UNIVERSITÉ PARIS.
DIDEROT (Paris 7)
SORBONNE PARIS CITÉ

ÉCOLE DOCTORALE: 386 – Sciences mathématiques de Paris Centre

Laboratoire d'Informatique Algorithmique: Fondements et Applications (LIAFA)

DOCTORAT

INFORMATIQUE

RENAULT Marc P

Lower and Upper Bounds for Online Algorithms with Advice

Bornes inférieures et supérieures pour les algorithmes en ligne avec conseil

Thèse dirigé par Adi Rosén, Directeur de recherche
(CNRS, Université Paris Diderot - Paris 7)

Soutenue le 15 septembre 2014.

JURY

MAGNIEZ Frédéric	Directeur de recherche (CNRS, Université Paris Diderot - Paris 7)	Président
ROSÉN Adi	Directeur de recherche (CNRS, Université Paris Diderot - Paris 7)	Directeur de thèse
BANSAL Nikhil	Professor (Eindhoven University of Technology)	Rapporteur
DÜRR Christoph	Directeur de recherche (CNRS, Université Pierre et Marie Curie)	Rapporteur
FRAIGNIAUD Pierre	Directeur de recherche (CNRS, Université Paris Diderot - Paris 7)	Examinateur
NAOR Seffi	Professor (Technion - Israel Institute of Technology)	Examinateur
VAN STEE Rob	Lecturer (University of Leicester)	Examinateur

Abstract

Online algorithms operate in a setting where the input is revealed piece by piece; the pieces are called requests. After receiving each request, online algorithms must take an action before the next request is revealed, i.e. online algorithms must make irrevocable decisions based on the input revealed so far without any knowledge of the future input. The goal is to optimize some cost function of the input. Online problems have many real world applications, e.g. paging, task scheduling, and routing. Competitive analysis is the standard method used to analyse the quality of online algorithms. The competitive ratio is the worst-case ratio, over all valid finite request sequences, of the online algorithm's performance as compared to the performance of an optimal offline algorithm for the same request sequence. The competitive ratio compares the performance of an algorithm with no knowledge about the future against an algorithm with full knowledge about the future.

Since it is not always reasonable to assume that an algorithm has absolutely no knowledge of the future, there have been many ad hoc attempts to enhance online algorithms with partial knowledge of the future such as with lookahead or locality of reference. Online computation with advice generalizes and abstracts these methods and quantifies the amount of future information known to the algorithm in a general way so as to consider the evolution of the competitive ratio as a function of the amount of information known about the future. Two models of advice have recently been proposed. In the model of [EFKR11], a fixed amount of advice is given with each request. In the model of [BKK$^+$09], prior to performing any computation, an oracle produces an advice tape from which the algorithm can read the advice as desired. Many problems have been studied in these models (e.g. Metrical Task Systems, knapsack, set cover and paging).

The focus of this thesis is on the advice per request model. The goal is to explore various classic online problems in this framework and evaluate how the competitive ratio changes as a function of the amount of advice. That is, exploring the importance

of future knowledge for online problems by establishing lower and upper bounds on the competitive ratio of algorithms with advice. We study the k-server problem, the bin packing problem, the dual bin packing problem, scheduling on m identical machines, the reordering buffer management problem, and the list update problem.

For the k-server problem on general metric spaces, we present an algorithm that is $\left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$-competitive, where $3 \leq b \leq \log k$ is the number of bits of advice per request. This is an improvement of almost a factor of 2 over the previously known upper bound. The more significant contribution is that our algorithm and our analysis are more intuitive and easier to understand than previous algorithms with advice for the k-server problem and, thus, may lead to further improvements in the upper bound. Also, we present a 1-competitive algorithm for finite trees with advice that is a function of the caterpillar dimension of the tree (a measure that is the height of the tree in the worst case).

For the bin packing problem and the dual bin packing problem, we show that 1 bit of advice gives a 3/2-competitive algorithm. For these problems and the scheduling problem on m identical machines, with the objective function of any of makespan, machine covering and the minimization of the ℓ_p norm, $p > 1$, we show that it is possible to arbitrarily approach a competitive ratio of 1 with a constant amount of advice per request. For the bin packing problem, a $(1+\varepsilon)$-competitive algorithm that uses $O\left(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon}\right)$ bits of advice per request and, for the dual bin packing problem, a $1/(1-\varepsilon)$-competitive algorithm that uses $O\left(\frac{1}{\varepsilon}\right)$ bits of advice per request are given. For the scheduling problem, we provide a framework for scheduling jobs in a nearly optimal manner using advice. We show that, for the objective functions of minimizing the makespan and the ℓ_p norm, this gives a $(1+\varepsilon)$-competitive algorithm and, for the objective function of maximizing the machine cover, this gives a $(1/(1-\varepsilon))$-competitive algorithm. For all the objective functions, the algorithms use $O\left(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon}\right)$ bits of advice per request. We complement those results by giving a lower bound showing that for any online algorithm with advice to be optimal, for any of the above packing or scheduling problems, a non-constant number, $O(\log N)$, of bits of advice per request is needed, where N is the number of (optimal) bins or machines.

For the reordering buffer management problem, we show that for any $\varepsilon > 0$ there is a $(1+\varepsilon)$-competitive algorithm for the problem which uses only a constant number of advice bits per request, depending on ε. This algorithm is interesting in that there

is no known polynomial-time approximation scheme (PTAS) for the offline (NP-hard) version of the problem. We complement the above result by presenting a lower bound of $\Omega(\log k)$ bits of advice per request for a 1-competitive algorithm, where k is the size of the buffer.

For the list update problem, we show a lower bound of 13/12 on the worst-case ratio between an offline optimal algorithm that can only perform free exchanges and an offline optimal algorithm that can perform paid and free exchanges. This implies a lower bound on the competitive ratio of any online algorithm that performs only free exchanges, including those with advice. This result answers a question that has been implicitly open since 1996 [RW96]. Also, we discuss some natural algorithms with advice. The upper bounds for these algorithms are due to known upper bounds for randomized algorithms. An algorithm with 1 bit of advice per request has a competitive ratio of 1.618 due to the TIMESTAMP algorithm and, for a list of length ℓ, an algorithm with $\lceil \log \ell \rceil$ bits of advice per request has a competitive ratio of 1.6 due to the COMB algorithm.

In studying these problems, in particular the packing and scheduling problems, we adapt the techniques employed for offline asymptotic polynomial time approximation schemes (APTAS) and polynomial time approximation schemes (PTAS) to develop online algorithms with advice that achieve a competitive ratio of $1 + \varepsilon$ and use $\tilde{O}(\frac{1}{\varepsilon})$ bits of advice per request. Although no PTAS is known for the offline version of the reordering buffer management problem, we develop a similarly flavoured algorithm. The lower bound technique that was first presented in [EFKR11] and then refined in [BHK+13] is used to lower bound the amount of advice need for a 1-competitive reordering buffer management algorithm. In this thesis, the other lower bounds pertaining to algorithms with advice describe the amount of advice needed to be optimal and use techniques that are more ad hoc. The techniques used in this thesis for upper and lower bounds on online algorithms with advice will ideally inspire similar results for other online problems.

To my wife, Nisa, and our two boys, Xavier, her thesis baby, and Brân, my thesis baby.

Acknowledgements

First and foremost, I would like to thank my supervisor, Adi Rosén, for supporting and guiding me through my Master's research project and my PhD. It was a pleasure working with Adi these past four plus years. I could not have asked for a better supervisor.

I would like to acknowledge the financial support of the ANR project NeTOC which enabled me to attend conferences and visits to colleagues.

I would like to thank the various researchers I had a chance to meet and work with during the course of my PhD. In particular, I would very much like to thank Rob van Stee for hosting me during my visit to MPI, Alex López-Ortiz for hosting me during my visits to the University of Waterloo and Magnús Halldórsson for hosting me during my visit to Reykjavik University. Also, a special mention to the other researchers that I was fortunate enough to have worked with and to have had meaningful discussions with in Paris and abroad during my PhD: Anna Adamaszek, Spyros Angelopoulos, Allan Borodin, Reza Dorrigiv, Christoph Dürr, Amos Fiat, Christian Konrad, Alejandro Salinger, Joachim Schauer, Norbert Zeh, and Uri Zwick. It was an enlightening experience working with everyone.

I am very grateful for having the opportunity to complete my PhD at LIAFA. I want to thank everyone (support staff, permanents and students) at the lab for creating a great learning environment. In particular, I would to thank the members of the Algorithms and Complexity group of which I have been a member for the past 4 years and, in particular, Fred and Sophie for wise counsel over the years.

Thanks to the wonderful people I had the pleasure of sharing an office with during my tenure at LIAFA (in more or less chronological order): Djamal, André, Christian, Loïck, Xavier, Antoine, Laure, Viginie, Mathieu, Nans, and Sven. Last but not least, a thanks to Nathanaël and Jamie, who technically didn't have a desk in my office.

Finally, I would also like to thank my wife, Nisa, for all her support.

Contents

Abstract **iii**

Dedication **vii**

Acknowledgements **ix**

Table of Contents **xi**

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**

1.1 Problems Considered 5

1.1.1 The k-Server Problem 5

1.1.1.1 Contributions 6

1.1.2 Bin Packing 7

1.1.2.1 Contributions 8

1.1.3 Dual Bin Packing 8

1.1.3.1 Contributions 9

1.1.4 Scheduling on m Identical Machines 9

1.1.4.1 Contributions 11

1.1.5 Reordering Buffer Management 11

1.1.5.1 Contributions 12
1.1.6 The List Update Problem 13
1.1.6.1 Contributions 14

2 Model and Techniques **15**
2.1 Online Computation 15
2.2 Online Computation with Advice 17
2.2.1 Online Advice Model 17
2.2.2 Semi-Online Advice Model 18
2.2.3 Comparing the Models with Advice 20
2.3 Techniques Used in this Thesis 20
2.3.1 A Lower Bound Technique 20
2.3.2 PTAS Inspired Algorithms 22

3 The k-Server Problem **25**
3.1 Preliminaries 25
3.2 An Upper Bound for General Metric Spaces 27
3.3 k-Server with Advice on Trees 33
3.3.1 Non-Lazy Optimum 34
3.3.2 Algorithm PATH-COVER 39
3.3.2.1 Algorithm and Advice for $r_1, \ldots, r_k$ 39
3.3.2.2 Algorithm and Advice for $r_{k+1}, \ldots, r_n$ 40
3.3.2.3 Analysis of PATH-COVER 41

4 Bin Packing **43**
4.1 Preliminaries 44
4.2 1.5-Competitive Online Algorithm with 1 bit of Advice per Request . . 45
4.3 $(1 + \varepsilon)$-Competitive Online Algorithms with Advice for Bin Packing . . 48
4.3.1 $(1 + 2\varepsilon)$-Competitive Algorithm 49
4.3.1.1 Formal Advice Definition 53
4.3.2 Strict $(1 + 3\varepsilon)$-Competitive Algorithm 53
4.4 Lower Bound on the Advice Required for Optimality 54

5 Dual Bin Packing **57**
5.1 Preliminaries . . . 58
5.2 1.5-Competitive Online Algorithm with 1 bit of Advice per Request . . 58
5.3 $(1/(1-\varepsilon))$-Competitive Algorithm with Advice . . . 61
5.3.1 $(1/(1-\varepsilon))$-Competitive Algorithm. . . . 61
5.3.2 Strict $(1/(1-2\varepsilon))$-Competitive Algorithm. . . . 65

6 Scheduling on Identical Machines **67**
6.1 Preliminaries . . . 68
6.2 Online Algorithms with Advice for Scheduling . . . 68
6.2.1 General Framework . . . 69
6.2.1.1 Formal advice definition . . . 72
6.2.2 Minimum Makespan . . . 74
6.2.3 Machine Covering . . . 74
6.2.4 The ℓ_p Norm . . . 76
6.3 Lower Bound on the Advice Required for Optimality . . . 79

7 Reordering Buffer Management **83**
7.1 Preliminaries . . . 83
7.2 1.5-Competitive Algorithm with 2 Bits of Advice per Request . . . 86
7.2.1 Definition of the Advice and the Algorithm . . . 86
7.2.2 The Analysis . . . 89
7.3 An Optimal Algorithm with $2 + \lceil \log k \rceil$ Bits of Advice per Request . . . 92
7.3.1 The Advice, the Waiting List and the Algorithm . . . 93
7.3.2 The Analysis . . . 94
7.4 A $(1+\varepsilon)$-Competitive Algorithm with $2 + \lceil \log k \rceil$ Bits of Advice per Request . . . 95
7.4.1 The Advice and the Algorithm . . . 95
7.4.1.1 Building the Advice Sequences . . . 96
7.4.2 The Analysis . . . 99
7.5 A $(1+\varepsilon)$-Competitive Algorithm with $O(\log(1/\varepsilon))$ Bits of Advice per Request . . . 104
7.5.1 The Advice and the Algorithm . . . 104
7.5.1.1 Building the Final Advice Sequences . . . 105

7.5.2 The Analysis . . . 105
7.6 Lower Bound on the Advice Required for a Competitive Ratio of 1 . . . 107
7.6.1 Ψ Request Sequences . . . 108
7.6.2 Tidy Algorithm . . . 109
7.6.3 A Predetermined Tidy Algorithm . . . 111
7.6.4 Reduction from the q-SGKH Problem . . . 113

8 The List Update Problem 117
8.1 Preliminaries . . . 117
8.2 Notes on Upper Bounds . . . 119
8.3 Lower Bound for OPT_FREE . . . 121

9 Conclusion 133

References 137

List of Figures

2.1 The online advice model 18
2.2 The semi-online advice model 19

3.1 An illustration of a caterpillar decomposition of a tree. 26
3.2 An illustration of the i-th cycle of server s as served by OPT and by CHASE. 31
3.3 An illustration of $x = \text{maxPath}(u, v)$. 35
3.4 An illustration of the definition of the advice bit x for r_i for the PATH-COVER algorithm. 41

4.1 An example of grouping and rounding of large items for the $(1 + \varepsilon)$-competitive bin packing algorithm. 50
4.2 An illustration of the packing produced by ABPA. 52

6.1 An example of grouping of the large jobs for the $(1 + \varepsilon)$-competitive scheduling framework. 69
6.2 An illustration of the permutation of the machines of S when scheduled by the online algorithm. 72

7.1 An illustration of the three possible initial item types. 87
7.2 An example of the procedure Split(B). 98
7.3 An example of the procedure Postpone(B). 99

8.1 The two possible list configurations of OFF for σ and a list of length 3. . 126

List of Tables

4.1 The size, weight and expansion for different types, according to the classification used by the HARMONIC algorithm. 46

8.1 A summary of the potential optimal free move algorithms for $R(L)$, where $L = y, x_1, x_2$ 124

1 Introduction

Online algorithms operate in a setting where the input is revealed piece by piece. These pieces are called requests. After receiving each request, online algorithms must take an action before the next request is revealed. That is, online algorithms must make irrevocable decisions based on the input revealed so far without any knowledge of the future input. Since the algorithm does not know future requests, this decision may prove to be suboptimal as more requests are revealed. The goal is to optimize some cost function of the input. The study of online algorithms therefore focuses on the quality of the solution without knowledge of the future as compared the optimal solution with full knowledge of the future. Online problems have many real world applications, e.g. paging, task scheduling, and routing.

A classic and illustrative online problem is called the ski rental problem (cf. [IK97]). Suppose that the cost to buy skis is b and renting skis costs b/r ($r \geq 1$) per day of skiing. The question is whether or not to buy skis. If we knew how many days we would go skiing the problem would be trivial. However, as the future is uncertain (e.g. weather, injuries, snow conditions), is there a way to solve this problem with a guaranteed result? There is an online deterministic algorithm that guarantees that we will not pay more than twice what is optimal, and this is the best possible for the deterministic case (cf. [IK97]). The algorithm works as follows: rent skis until the cost of renting is equal to or exceeds the cost to buy skis. That is, we would rent skis r times and then buy them. If we go skiing r times or less, we would be optimal. If we go skiing more than r times, we would spend $2b$ while it would have been optimal to buy the skis initially for a cost of b.

1. INTRODUCTION

Competitive analysis is the standard method used to analyse the quality of online algorithms. The competitive ratio is the worst-case ratio, over all valid finite request sequences, comparing the performance of an online algorithm against the performance of an optimal offline algorithm for the same request sequence. The competitive ratio is comparing the performance of an algorithm with no knowledge of the future against an algorithm with full knowledge of the future.

In many cases, competitive analysis can be too restrictive [BEY98, BLN01, DLO05] since there are many times when it is reasonable to assume that an algorithm has partial knowledge of the future. There have been many ad hoc approaches taken to handle the restrictive nature of competitive analysis. Several online algorithms have been augmented using methods such as a lookahead of future requests (e.g. [Gro95, Alb97, Alb98, Bre98]) or parametrized by a metric such as locality of reference (e.g. [BIRS95, IKP96, FM97, AFG05, AL08, ADLO08]).

Partially inspired by these approaches, two models of online computation with advice have been proposed in [EFKR11] and [BKK$^+$09]. Online computation with advice generalizes and abstracts these methods and quantifies the amount of future information known to the algorithm in a general way so as to consider the evolution of the competitive ratio as a function of the amount of information known about the future.

In addition, online computation with advice also models the scenario of a powerful server with the whole input and remote clients that will process the data in an online manner. We can imagine this scenario occurring with a remote robot that has some online task to perform and there is a significant cost in communication in terms of battery life. For instance, the remote robot must collect a series of core samples of different heights. The details (position and height) of the core samples are transmitted in an online manner to the robot. The storage of these core samples is a packing problem. With a limited amount of storage space, a few bits of advice could enable the robot to store the samples more efficiently, allowing it to collect more samples on a single trip.

This scenario can also occur in the context of big data computation. Imagine a powerful server or server farm that houses a massive amount data and simple remote clients that want to perform some calculation on the data, e.g. physicists working on a laptop accessing the Large Hadron Collider mainframe. In this setting, if the powerful

server would do some preprocessing of the data, it could aid the remote clients in their calculations by providing advice along with the data.

In the model of Emek et al. [EFKR11], termed in this thesis the *online advice model*, online algorithms with advice are algorithms that are given access to a quantified amount of knowledge about the future in an online manner (see Section 2.2.1 for a formal definition). The advice is given with each request and the definition of the advice is part of the specification of the algorithm. The advice given at request i is the value of some function u_i when applied to the whole request sequence, including future requests. The range of the function is 2^b binary strings, where b is the number of advice bits received per request. The interest is to see how the competitive ratio changes as a function of the amount of bits of advice per request. The majority of the work in this thesis is in this model.

In the other model of Böckenhauer et al. [BKK+09], termed in this thesis the *semi-online advice model*, online algorithms with advice are given access to an infinite advice tape from which the algorithm can read at any point (see Section 2.2.2 for a formal definition). The advice is written to the tape as function of the entire request sequence and is part of the specification of the algorithm. The interest is to see the evolution of the competitive ratio as a function of the total amount of bits of advice, as a function of the size of the input, read from the advice tape. This model is stronger than the online advice model since the algorithm can read all the advice bits before any of the requests, whereas, in the online advice model, the advice bits are revealed in an online manner with the requests.

Returning to the ski rental problem, we see that, with a single bit of advice indicating whether to rent or to buy skis, an online algorithm with advice can be optimal. It is not always the case for all online problems with advice that a constant number of advice bits can improve the competitive ratio to 1. There are known non-constant bounds on the amount of advice required to be optimal for some problems, e.g. Metrical Task System [EFKR11].

Online computation with advice provides a general model for situations of partial knowledge of the future and has been used to study a variety of classic online problems in both models, e.g. paging [BKK+09], the k-server problem [EFKR11, BKKK11, RR12, GKLO13], Metrical Task System [EFKR11], bin packing [BKLLO12, RRvS13], set cover [KKM12], the knapsack problem [BKKR12], various scheduling

problems [BKK+09, KK11, RRvS13, Doh13], buffer management problems [DHZ12, ARRvS13].

In this thesis, we consider the k-server problem, the bin packing problem, the dual bin packing problem (a multiple knapsack problem), the machine scheduling problem on m identical machines, the reordering buffer management problem and the list update problem. These problems are classical online problems that have been well studied in the standard online framework and, in some cases, also studied in the online advice framework. In studying these problems, in particular, the packing and scheduling problems, we adapt the techniques employed for offline asymptotic polynomial time approximation schemes (APTAS) and polynomial time approximation schemes (PTAS) to develop online algorithms with advice that achieve a competitive ratio of $1 + \varepsilon$ and use $\tilde{O}(\frac{1}{\varepsilon})$ bits of advice per request. Although no PTAS is known for the offline version of the reordering buffer management problem, we develop a similarly flavoured algorithm. The lower bound technique, as described in Section 2.3.1, was first presented in [EFKR11] and then refined in [BHK+13]. It is used, in this thesis, to lower bound the amount of advice needed for a 1-competitive reordering buffer management algorithm. In this thesis, the other lower bounds pertaining to algorithms with advice describe the amount of advice needed to be optimal and use techniques that are more ad hoc. The techniques used in this thesis for upper and lower bounds on online algorithms with advice will ideally inspire similar results for other online problems.

For the problems considered in this thesis, small gaps remain between the upper and lower bounds on the amount of advice needed for a given competitive ratio for the packing, scheduling and reordering buffer problems. Much larger gaps remain for the k server problem and the list update problem. Also, a natural follow up to this work would be to consider more general versions of the packing, scheduling and reordering buffer problems. In a broader sense, an interesting area to explore in more depth is the relation between online algorithms with advice and online randomized algorithms without advice.

The work in Chapter 3 on the k-server problem was a collaboration with Adi Rosén. A preliminary version appeared in the 9th Workshop on Approximation and Online Algorithms (WAOA 2011) [RR11] and a final version appeared in the *Theory of Computing Systems* special issue for WAOA 2011 [RR12]. The work in Chapter 4 and Chapter 6 on bin packing and scheduling on m machines was a collaboration with Adi Rosén and

Rob van Stee. The $(1+\varepsilon)$-competitive algorithms and the lower bound for scheduling can be found in [RRvS13]. The work in Chapter 7 on the reordering buffer management problem was a collaboration with Anna Adamaszek, Adi Rosén and Rob van Stee. A preliminary version of these results appeared in the 11th Workshop on Approximation and Online Algorithms (WAOA 2013) [ARRvS13]. The work in Chapter 8 on the list update problem was a collaboration with Alejandro López-Ortiz and Adi Rosén.

1.1 Problems Considered

1.1.1 The k-Server Problem

The k-server problem, originally proposed by Manesse et al. [MMS90], consists of k mobile servers on a metric space. Requests appear on the nodes of the metric space and a server must be moved to cover the requested node (see Section 3.1 for a formal definition). It is a generalisation of the paging problem (uniform metric space) [MMS90] and one of the fundamental online problems (cf. [BEY98]).

In complete generality, the best known algorithm for the k-server problem is the deterministic WORK FUNCTION ALGORITHM (WFA). In [KP95], Koutsoupias and Papadimitriou show that it is $(2k-1)$-competitive and conjecture it to be k-competitive. The deterministic lower bound on the competitive ratio is k [MMS90]. WFA is known to be k-competitive on restricted metrics such as the line, the star and $N \geq k+2$, where N is the number of nodes in the metric space [BK04]; $k=2$ [CL92]; and $k=3$ for the ℓ_1^2 metric (the Manhattan plane) [BCL02]. Chrobak and Larmore present a k-competitive algorithm for the tree metric [CL91], and Sleator and Tarjan present several k-competitive algorithms for the uniform metric space (paging) [ST85]. Recently, in [BBMN11], Bansal et al. present a randomized algorithm that is $\tilde{O}(\log^3 N \log^2 k)$-competitive in expectation which is lower than the competitive ratio of WFA, in expectation, for metrics where N is sub-exponential in k.

In [EFKR11], Emek et al. present the first online algorithm with advice for the k-server problem for the general metric space. It uses $\Theta(1) \leq b \leq \log k$ bits of advice per request and has a competitive ratio of $k^{O(1/b)}$.

This result was improved exponentially by Böckenhauer et al. to $2\left\lceil\frac{\lceil \log k \rceil}{b-1}\right\rceil$ for $b \geq 2$ [BKKK11]. In addition, Böckenhauer et al. present a lower bound on optimality for the general metric space of at least $\log \frac{k}{e}$ bits of advice per request, where e is Euler's

number. They also considered the Euclidean metric space and present an algorithm with a competitive ratio of $1/\left(1 - 2\sin\frac{\pi}{2^b}\right)$, using $b \geq 3$ bits of advice per request.

In addition, restricted metric spaces were considered in [Ren10]. Specifically, 1-competitive algorithms using 1 bit of advice per request for the line, the cycle, the star metrics, and $N = k + 1$ metrics, where N is the number of nodes in the metric space; and a 1-competitive algorithm using 2 bits of advice for spider metrics. In [DKP08], a 1-competitive algorithm using 1 bit of advice per request for the paging problem is implicit, and, in [Ren10], is shown to extend to the weighted paging problem. The algorithm for star metrics is a reduction from that algorithm for the weighted paging problem [Ren10]. Some of these results appear in [RR11] and the details of the algorithm for the line appear in [RR12].

Subsequent to the publication of our work on the k-server problem with advice [RR11, RR12], Gupta et al. [GKLO13] apply the algorithm for trees [RR12] to metric spaces that have bounded treewidth α, and metric spaces that admit a system of μ collective tree (q, r)-spanners. The former is $O(1)$-competitive, using $O(\log\alpha + \log\log N)$ bits of advice per request, and the later is $O(q+r)$-competitive, using $O(\log\alpha+\log\log N)$ bits of advice per request, where N is the number of nodes in the metric space. In addition, Gupta et al. show a tight lower bound for the line algorithm of [RR12]. That is, that a linear, in the length of the request sequence, amount of advice in total is required for optimality. For the metric spaces with bounded treewidth of $4 \leq \alpha \leq 2k$, Gupta et al. show that $(\log\alpha - 1.22)/2$ bits of advice per request are required for optimality.

1.1.1.1 Contributions

We consider the k-server problem on general metric spaces and on finite trees. That is, we improve the upper bound for deterministic k-server algorithms with advice on general metric spaces by giving a deterministic online algorithm with b bits of advice per request, for $3 \leq b \leq \log k$, whose competitive ratio is $\left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$. While the improvement over the previous result is only about a factor of 2, we believe that our algorithm and analysis are more intuitive and simpler than previous ones, and may lead to further improvements in the upper bound.

Also, we consider the class of metric spaces of finite trees, and give a 1-competitive deterministic online algorithm. The number of bits of advice per request used by this algorithm is $2 + 2\lceil\log(p + 1)\rceil$, where p is the caterpillar dimension of the tree

(cf. [Mat99]). The caterpillar dimension of the tree is preferable over other measures, such as height, because it remains constant for degenerate trees, such as the line, the spider and the caterpillar. Moreover, the caterpillar dimension is at most the height of the tree, and it is at most $\log N$, where N is the number of nodes in the tree [Mat99].

1.1.2 Bin Packing

The (1 dimensional) bin packing problem also termed the (1 dimensional) stock-cutting problem is a classic theoretical computer science problem that consists of a sequence of items of varying size (the maximum size of an item is the capacity of a bin) that must be packed in fixed size bins. The goal is to minimize the number of bins used. (See Section 4.1 for a formal definition.) The problem has been extensively studied in both the offline and online settings.

In the offline case, the problem is NP-complete [GJ79]. Fernandez de la Vega and Lueker [FdlVL81] presented an asymptotic polynomial time approximation scheme (APTAS) for the bin packing problem.

For online bin packing, one of the earliest uses of competitive analysis (although the term was coined later by Sleator and Tarjan [ST85]) was in the analysis of the online ANY FIT algorithms (e.g. NEXT FIT, FIRST FIT, and WORST FIT) for the bin packing problem by Garey et al. [GGU72], Johnson [Joh74] and Johnson et al. [JDU+74]. The best known lower bound on the competitive ratio is 1.54037 due to Balogh et al. [BBG12] and the best known deterministic upper bound on the competitive ratio is 1.58889 due to Seiden [Sei02]. The algorithm analysed by Seiden is HARMONIC++ which is an improved version of the HARMONIC algorithm of Lee and Lee [LL85]. Chandra [Cha92] showed that all known lower bounds can be shown to apply to randomized algorithms against an oblivious adversary.

Boyar et al. [BKLLO12] study the bin packing problem with advice, using the semi-online advice model of [BKK+09] and present a 3/2-competitive algorithm, using $\log n$ bits of advice in total and a $(4/3 + \varepsilon)$-competitive algorithm, using $2n + o(n)$ bits of advice in total, where n is the length of the request sequence. As both algorithms rely on reading $O(\log(n))$ bits of advice prior to receiving any requests, they would use $O(\log(n))$ bits of advice per request in the online advice model. The 3/2-competitive algorithm can be converted into an algorithm that uses 1 bit of advice per request that

would be similar to the algorithm presented in this thesis. We are not aware of a similar simple conversion for the $(4/3 + \varepsilon)$-competitive algorithm.

Further, in [BKLLO12], Boyar et al. show that, in total, linear advice is required for a competitive ratio better than 5/4 (in the semi-online advice model; in the online advice model, an algorithm has at least 1 bit of advice per request, i.e. at least linear advice in total). Finally, using essentially the same techniques as the lower bound presented here, they show that an online algorithm with advice requires at least $(n - 2N)\log N$ bits of advice to be optimal, where N is the optimal number of bins.

1.1.2.1 Contributions

We present two algorithms with advice for the bin packing problem. The first algorithm is inspired by the HARMONIC algorithm, uses 1 bit of advice per request and has a competitive ratio of 3/2. The second algorithm, for $0 < \varepsilon \leq 1/2$, achieves a competitive ratio of $1 + \varepsilon$, and uses $O\left(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon}\right)$ bits of advice per request.

These results are complemented by a lower bound of $\Omega(\log N)$ on the amount of advice required per request to be optimal, where N is the optimal number of bins.

1.1.3 Dual Bin Packing

The dual bin packing problem was first studied in the offline setting in [JLT78] and has applications to processor and storage applications in computers [CL79]. The goal of this problem is to maximize the number of items from a sequence that are packed into a fixed number of unit sized bins (see Section 5.1 for a formal definition). It is a special case of the multiple knapsack problem (MKP), where the profit is the same for all the items and the capacity is the same for all the knapsacks. This problem, MKP and the even more general Generalized Assignment Problem (GAP) have been greatly studied in the offline setting (cf. [KPP04]). The problem is NP-complete [GJ79] and a polynomial time approximation scheme (PTAS) is known [Kel99].

In the online setting, it has also been studied, but less so than the offline setting given that, in the general setting for this problem, no online algorithm can pack a constant fraction of the items packed by the offline optimal algorithm whether the algorithm is deterministic [BLN01] or randomized [CJ13]. Hence, the online work has focused on various settings that either strengthen the online algorithm or weaken the adversary, e.g. [BLN01, ABE$^+$02, EF03, CJ13]. When restricted to accommodating

sequences (sequences where all the items are packed in an optimal packing), First Fit has an asymptotically tight competitive ratio of 8/5 and a variant Unfair First Fit has an asymptotically tight competitive ratio of 3/2 [ABE+02].

The single knapsack problem with advice was studied in [BKKR12], using the semi-online advice model of [BKK+09]. In the online advice model of [EFKR11], this problem is trivially solvable, using only 1 bit of advice per request. In [BKKR12], for the case of unit profit, Böckenhauer et al. show that 1 bit of advice in total is sufficient for a competitive ratio of 2, and that $\log n$ bits of advice in total are required for a competitive ratio below 2. In the general case, Böckenhauer et al. show that no algorithm is competitive with less than $\log n$ bits of advice. Also, an algorithm is presented that is $(1+\varepsilon)$-competitive with $O((\log m)/\varepsilon)$ bits of advice in total, where m is the length of the request sequence. This results depends on the ability to encode the profit and the weight of the items efficiently in the advice, whereas the $(1+\varepsilon)$-competitive algorithm presented in this thesis for the dual bin packing problem (multiple knapsacks) does not have this restriction.

1.1.3.1 Contributions

We present two algorithms with advice for the dual bin packing problem. The first algorithm uses 1 bit of advice per request and packs the items using First Fit. It has a competitive ratio of at most 3/2. The second algorithm, for $0 < \varepsilon < 1/2$, achieves a competitive ratio of $1/(1-\varepsilon)$, and uses $O\left(\frac{1}{\varepsilon}\right)$ bits of advice per request.

1.1.4 Scheduling on m Identical Machines

The problem of scheduling on m identical machines consists of m identical machines, an objective function and a sequence of jobs of varying processing times. The goal is to schedule the jobs in an optimal manner for the objective function (see Section 6.1 for a formal definition). The objective function can be to minimize or maximize some cost. The framework presented in Chapter 6 is applicable to many objective functions; we focus on makespan, minimizing the maximum load; machine cover, maximizing the minimum load; and minimizing the ℓ_p norm. For these objective functions, the problem is NP-hard [GJ79].

In the offline case, Hochbaum and Shmoys [HS87] developed a polynomial time approximation scheme (PTAS) for the makespan minimization problem on m identical

machines. Subsequently, Woeginger [Woe97] presented a PTAS for the machine covering problem on m identical machines and Alon et al. [AAWY97] presented a PTAS for the ℓ_p norm minimization problem on m identical machines.

The online problem of scheduling on m identical machines for the makespan objective is one of the first uses of competitive analysis [Gra66] (again the term was not coined until [ST85]). For the makespan objective without advice, Rudin and Chandrasekaran [RC03] presented the best known deterministic lower bound on the competitive ratio for minimizing the makespan of 1.88. The best known deterministic upper bound on the competitive ratio for minimizing the makespan, due to Fleischer et al.[FW00], is 1.9201 as $m \to \infty$. The best known randomized lower bound on the competitive ratio for minimizing the makespan is $1/(1-(1-1/m)^m$ which tends to $e/(e-1) \approx 1.58$ as $m \to \infty$ was proved independently by Chen et al. [CvVW94] and Sgall [Sga97], and the best known randomized algorithm, due to Albers [Alb02], has a competitive ratio of 1.916. For machine covering, Woeginger [Woe97] proved tight $\Theta(m)$ bounds on the competitive ratio for deterministic algorithms, and Azar and Epstein [AE98] showed a randomized lower bound of $\Omega(\sqrt{m})$ and a randomized upper bound of $O(\sqrt{m}\log m)$. For minimizing the ℓ_p norm, Avidor et al. [AAS98] showed a deterministic lower bound on the competitive ratio of $3/2-\Theta(1/p)$ and a deterministic upper bound of $2-\Theta(\ln p/p)$.

In [AE98], Azar and Epstein considered the case where the optimal value is known to the algorithm and showed that, for $m \geq 4$, no deterministic algorithm can achieve a competitive ratio better than 1.75.

In the semi-online advice model, Böckenhauer et al. [BKK$^+$09] and Komm et al. [KK11] consider a special case of the job shop scheduling problem. The problem considered in [BKK$^+$09, KK11] consists of m machines, two jobs consisting of m tasks of unit processing time such that each task per job must be run on a different machine. That is, each job is a permutation of $1, \ldots, m$. The tasks must be assigned sequentially and a machine can process one task per time unit. The objective function is the minimization of the makespan. In [BKK$^+$09], the authors show that $\Omega(m)$ bits of advice in total are required for optimality and, in [KK11], the authors present an algorithm that uses, for $d \in o(m)$, at most $\log d$ bits of advice in total and has a competitive ratio that tends to $(1+1/d)$ as the number of tasks d grow.

In a technical report, Dohrau [Doh13] considers the makespan objective for machine scheduling on 2 identical machines, using the semi-online advice model, and shows that $n-2$ bits of advice in total are required to be optimal. (The trivial optimal algorithm with advice uses $n-1$ bits of advice in total.) Additionally, an algorithm that is $(1+\varepsilon)$-competitive using $O(\log \frac{1}{\varepsilon})$ bits of advice in total is presented for machine scheduling on 2 identical machines for the makespan objective.

1.1.4.1 Contributions

For scheduling on m identical machines, a general framework is presented for an online algorithm to produce a schedule such that (up to a permutation of the machines) the load of machine i is within ε times the load of machine i in the optimal schedule. This general framework is applicable to any objective function of the form $\sum_{i=1}^{m} f(L_i(S))$ ($L_i(S)$ is the load on machine i for a schedule S) such that f satisfies the property that if $x \leq (1+\varepsilon)y$ then $f(x) \leq (1+O(1)\varepsilon)f(y)$ (and a weak assumption regarding the optimal schedule given the maximum size of jobs used in a geometric classification scheme [see Section 6.2]). In this thesis, we consider the objective functions of makespan, machine covering and minimizing the ℓ_p norm for $p > 1$. For any of these, we present online algorithms with advice that, for $0 < \varepsilon < 1/2$, are $(1+\varepsilon)$-competitive ($(1/(1-\varepsilon))$-competitive for machine covering) and use $O\left(\frac{1}{\varepsilon} \log \frac{1}{\varepsilon}\right)$ bits of advice per request.

We complement our results by showing that, for any of the scheduling problems we consider, an online algorithm with advice needs $\Omega(\log m)$ bits of advice per request to be optimal.

1.1.5 Reordering Buffer Management

The reordering buffer management problem was introduced in 2002 by Räcke et al. [RSW02]. The problem models a service station that must process a sequence of coloured items. At any time, the service station is configured to process items of a certain colour c. Changing the configuration of a service station to a different colour generates a cost. The total cost is the number of colour switches performed while serving a sequence of items. In order to reduce this cost, the service station is equipped with a reordering buffer that has the capacity to hold k items. The service station can process any of the items contained in its buffer. Using this reordering buffer, the goal is to find a schedule that minimizes the number of colour switches. In the online setting,

the colour of each item is known only once the item enters the reordering buffer (see Section 7.1 for a formal definition).

This problem is a natural model for job shops such as paint shops and blow moulding facilities, where there is a large cost in terms of time and materials to switch colours (paint or plastic) due to the need clean out the old colour and set-up the equipment in order to correctly produce the new colour. For this reason, such production plants are equipped with a reordering buffer to preprocess the sequence of incoming cars or production orders [GSV04]. This framework also has many applications in various areas, such as production engineering, storage systems, network optimization and computer graphics (see [AER10, BB02, GSV04, KRSW04, RSW02] for more details).

Since its introduction in 2002, this problem has been extensively studied. In the online setting, the best known results are a deterministic $O(\sqrt{\log k})$-competitive algorithm by Adamaszek et al. [ACER11], and a randomized $O(\log\log k)$-competitive algorithm by Avigdor-Elgrabli and Rabani [AER13b]. To complement this, there are nearly matching lower bounds of $\Omega(\sqrt{\log k/\log\log k})$ and $\Omega(\log\log k)$ on the competitive ratio of any online deterministic and randomized algorithms, respectively [ACER11]. In the offline setting, the problem is known to be NP-hard [CMSvS12, AKM12], and the best known result is a constant factor deterministic approximation algorithm by Avigdor-Elgrabli and Rabani [AER13a].

More general versions of the problem have been studied, where the context switching cost for switching from an item of colour c to an item of colour c' depends on c' (e.g. [EW05, ACER11]), or on both c and c' [ERW10].

1.1.5.1 Contributions

This is the first work on this problem in the advice framework. First, we present an online algorithm with advice for the reordering buffer management problem that has a competitive ratio of 1.5 and uses 2 bits of advice per request. We extend this algorithm using more bits of advice per request. The bits of advice per request are still a constant amount and the algorithm achieves a competitive ratio that is arbitrary close to 1. Specifically, for any $\varepsilon > 0$, the algorithm has a competitive ratio of $1 + \varepsilon$, using only $O(\log(1/\varepsilon))$ advice bits per request.

We complement the above result by showing that in order for an online algorithm to be 1-competitive, the number of bits of advice must depend on k, the size of the buffer. More precisely, $\Omega(\log k)$ bits are required.

1.1.6 The List Update Problem

The list update problem consists of a linked list of ℓ items and a sequence of requests to the items. The requested items are accessed from the head of the list and this costs the current index of the accessed item in the list. After the access, the requested item can be move forward in the list at no cost. This is called a *free exchange*. At any point, two adjacent items can be swapped for a cost of 1. This is called a *paid exchange*. The goal is to minimize the cost over the request sequence (see Section 8.1 for a formal definition).

The list update problem (also called the list access problem) was one of the two problems (the other being the paging problem) studied in the seminal work on competitive analysis of Sleator and Tarjan in [ST85] and is a fundamental problem of online algorithms that has been intensely studied, particularly, due to its importance in compression [BSTW86]. In [ST85], Sleator and Tarjan presented a 2-competitive deterministic algorithm called MOVE TO FRONT (MTF) that Irani showed to be an optimal online deterministic algorithm [Ira91]. As its name implies, MTF moves every requested item to the front, using free exchanges.

Also, in [Ira91], Irani presented the first online randomized algorithm, for the list update problem, with a competitive ratio of 15/8. Reingold and Westbrook present the first barely random online algorithm called BIT that has a competitive ratio of at most 7/4 [RWS94]. A barely random algorithm uses a number of random bits that is independent of the length of the request sequence. In the case of BIT, it uses ℓ random bits to assign a random value of 0 or 1 to each item of the list. When an item is request, if the bit of the item is a 0 BIT moves it to the front, and, in either case, BIT toggles the bit. The best online randomized algorithm is COMB due to Albers [AvSW95]. It has a competitive ratio 1.6 [AvSW95] and randomly chooses between using the BIT algorithm and the deterministic TIMESTAMP algorithm (see Chapter 8 for more details about TIMESTAMP).

The offline problem is known to NP-complete [Amb00]. It is not known if this holds if only free exchanges are permitted. In [ST85], Sleator and Tarjan claim that

an optimal algorithm that uses paid exchanges and free exchanges can be converted to an optimal algorithm that uses only free exchanges without increasing the cost. This claim turns out not to be true as Reingold and Westbrook present the counter-example of $\langle 3,2,2,3 \rangle$ for a list of length 3 with a starting configuration of $1,2,3$ [RW96]. An optimal algorithm serves this sequence for a cost of 8 while an optimal algorithm using only free moves serves this sequence for a cost of 9.

In [BKLLO14], Boyar et al. study the list update problem, using the semi-online advice model of [BKK$^+$09]. They show that the total amount of advice must be linear in the size of the request sequence for a competitive ratio better than 15/14. Also, they present an algorithm that uses 2 bits of advice in total that is 5/3 competitive. However, even though this algorithm uses advice, it still has a worse competitive ratio than the best randomized competitive ratio of 1.6 (the algorithm COMB [AvSW95]).

1.1.6.1 Contributions

The main result, for the list update problem, is a comparison between the cost of an optimal offline algorithm that can only perform free exchanges, OPT_FREE, and an optimal optimal offline algorithm that can use both paid and free exchanges, OPT. We show a lower bound of 13/12 on the worst-case ratio over all possible request sequences between the performance of an OPT_FREE and an OPT. This implies a lower bound on any algorithm that performs only free exchanges, including those with advice. Also, it answers a question that has been implicitly open since 1996 when Reingold and Westbrook showed that there was at least an additive constant in the difference between OPT_FREE and OPT [RW96]. They showed this in their counter-example to the claim that OPT_FREE was equivalent to OPT by Sleator and Tarjan in [ST85].

2 Model and Techniques

In this chapter, we formally define online computation and competitive analysis (cf. [BEY98]). Then, we define and compare the two models of online computation with advice. That is, the online advice model of Emek et al. [EFKR11] and the semi-online advice model of Böckenhauer et al. [BKK$^+$09]. Finally, we review some of the techniques used in this thesis.

2.1 Online Computation

In *online computation*, algorithms receive their input piece by piece. Each piece, termed a *request*, is an element of some possible infinite set R of requests. The algorithm receives a *request sequence* denoted $\sigma = r_1, \ldots, r_n$, where $n = |\sigma|$ and r_i is the i-th request. An online algorithm must perform some action on r_i before r_{i+1} is revealed. In general, these actions are irrevocable and define some portion of the output. In addition, the online algorithm incurs some cost performing these actions. The goal is to optimize this cost for σ in spite of having no knowledge about the length of σ or the subsequent requests.

Using the request-answer system of [BDBK$^+$90] (see also [BEY98]), we formally define deterministic online algorithms and randomized online algorithms as follows.

Definition 2.1 (Deterministic Online Algorithm)**.** *Given a request set R for some online problem, a deterministic online algorithm consists of*

- *a sequence of finite nonempty answer sets $A_1, A_2, \ldots$ and*

- *a sequence of cost functions* $\text{cost}_n : R^n \times A_1 \times A_2 \times \cdots \times A_n \to \mathbb{R}^+ \cup \{\infty\}$ *for* $n = 1, 2, \ldots$.

The actions of the deterministic online algorithm are the functions $g_i : R^i \to A_i$ *for* $i = 1, 2, \ldots$.

From this definition, for request r_i, the action a_i of an online deterministic algorithm is $g_i(r_1, \ldots, r_i)$, i.e. a function of the requests received so far.

Definition 2.2 (Randomized Online Algorithm)**.** *A randomized online algorithm is a distribution over deterministic online algorithms.*

As online problems are minimization or maximization problems, we are trying to minimize or maximize some cost over a request sequence, $\sigma = \langle r_1, \ldots, r_n \rangle$. Let $\text{ALG}(\sigma)$ and $\text{OPT}(\sigma)$ be the cost to an online algorithm, ALG, and the optimal algorithm, OPT, over σ respectively. Using the request-answer system of [BDBK+90], we formally define the cost of an online algorithm in the following.

Definition 2.3 (Cost of an online algorithm)**.** $\text{ALG}(\sigma) = \text{cost}_n(\sigma, \text{ALG}[\sigma])$*, where* $\text{ALG}[\sigma] = \langle a_1, \ldots, a_n \rangle \in A_1 \times \cdots \times A_n$*,* $a_i = g_i(r_1, \ldots, r_i)$ *and* $\sigma = \langle r_1, \ldots, r_n \rangle$*.*

We will analyse the performance of an online algorithm through *competitive analysis*[ST85]. That is, a worst-case analysis of the performance of the online algorithm as compared to an offline optimal algorithm which gives us the competitive ratio as defined in the following.

Definition 2.4 (Competitive Ratio for Deterministic Algorithms)**.** *For a minimization problem, we say that a deterministic algorithm is* c-competitive *or has a* competitive ratio *of c, if, for every finite request sequence* σ*,*

$$\text{ALG}(\sigma) \leq c \cdot \text{OPT}(\sigma) + \zeta \ ,$$

where ζ *does not depend on* σ *and* $\text{OPT}(\sigma)$ *is the optimal cost over* σ*.*

For a maximization problem, an algorithm ALG *is c-competitive if*

$$\text{ALG}(\sigma) \geq \frac{1}{c} \cdot \text{OPT}(\sigma) - \zeta \ .$$

For randomized algorithms, we consider only the *oblivious adversary* in this thesis. That is an adversary that does not know the values of the random coin tosses of the algorithm (cf. [BEY98]). For a randomized algorithm and an oblivious adversary, the definition of the competitive ratio is analogous except that we consider the expected cost of the algorithm over the distribution of the random bits of the algorithm.

Definition 2.5 (Competitive Ratio for Randomized Algorithms)**.** *For a minimization problem, we say that a randomized algorithm is* c-competitive *against an oblivious adversary or has a* competitive ratio *of c against an oblivious adversary, if, for every finite request sequence σ,*

$$\mathbb{E}[\text{ALG}(\sigma)] \leq c \cdot \text{OPT}(\sigma) + \zeta \ ,$$

where ζ does not depend on σ, $\mathbb{E}[\text{ALG}(\sigma)]$ is the expected cost of ALG *over the values of the random bits for σ and* $\text{OPT}(\sigma)$ *is the optimal cost over σ.*

For a maximization problem, an algorithm ALG *is c-competitive if*

$$\mathbb{E}[\text{ALG}(\sigma)] \geq \frac{1}{c} \cdot \text{OPT}(\sigma) - \zeta \ .$$

In all cases, when $\zeta = 0$, this is known as a *strict* competitive ratio, and we say that the algorithm is *strictly* c-competitive.

2.2 Online Computation with Advice

2.2.1 Online Advice Model

For this thesis, we focus on the model of online computation with advice introduced in [EFKR11]. In this thesis, this model is termed the *online advice* model or the *advice per request* model. In this model, the algorithm has access via a query to an advice space, U, which is a finite set. The advice space has a size of 2^b, where $b \geq 0$ is the number of bits of advice provided to the algorithm with each request. The advice query is part of the algorithm's specification and its value is a function of the whole request sequence (see Figure 2.1). Using the request-answer system of [BDBK+90], we formally define deterministic online algorithms and randomized online algorithms in this online advice framework as follows.

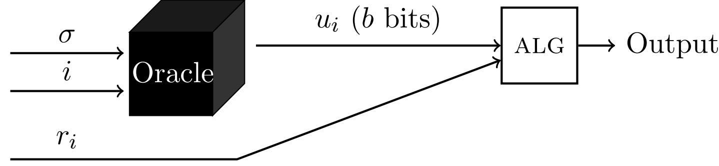

Figure 2.1: The online advice model as present by Emek et al. [EFKR11] consists of an advice oracle that provides the online algorithm an advice string, u_i, that is fixed amount of bits of advice, b, received with each request, r_i. The advice is received by the algorithm online at the same time as the request. The advice bits are a function of the entire request sequence, σ, including future requests.

Definition 2.6 (Deterministic Online Algorithm with Online Advice)**.** *Given a request set R for some online problem, a deterministic online algorithm with $b \geq 0$ bits of advice per request consists of*

- *a sequence of finite nonempty answer sets $A_1, A_2, \ldots$,*
- *a sequence of cost functions $cost_n : R^n \times A_1 \times A_2 \times \cdots \times A_n \to \mathbb{R}^+ \cup \{\infty\}$ for $n = 1, 2, \ldots$,*
- *a finite set U, for the advice space, of size 2^b,*
- *a sequence of query functions $u_i : R^* \to U$, indexed by the request serial, and*
- *a sequence of action functions $g_i : R^i \times U^i \to A_i$ for $i = 1, 2, \ldots$.*

For a request $r_i \in \sigma = r_1, r_2, \ldots$, the action a_i of a online deterministic algorithm with b bits of advice is $g_i(r_1, \ldots, r_i, u_1(\sigma), \ldots, u_i(\sigma))$, i.e. a function of all the requests and the advice received so far.

Definition 2.7 (Randomized Online Algorithm with Online Advice)**.** *A randomized online algorithm with b bits of advice is a distribution over deterministic online algorithms with b bits of advice.*

2.2.2 Semi-Online Advice Model

The other model of online computation with advice was introduced in [BKK$^+$09]. In this thesis, this model is termed the *semi-online advice* model or the *advice tape* model since the advice is provided to the algorithm in an offline manner via an infinite advice tape that can be read as needed by the algorithm, allowing for sublinear in n advice bits in total, where n is the length of the request sequence. The bits written to the

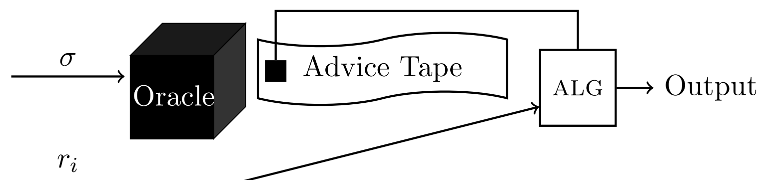

Figure 2.2: The semi-online advice model, another model of online computation with advice, as presented by Böckenhauer et al. [BKK+09] consists of an infinite advice tape that is written to by an oracle. The advice bits written by the oracle are a function of the entire request sequence, σ. The advice bits are read on demand by the algorithm and, hence, a stronger model than that of [EFKR11].

advice tape are a function of the entire request sequence (see Figure 2.2). For a request sequence of length n and a given algorithm, let $b(n)$ be the maximum number of bits of advice read in total by the algorithm over all request sequences of length n. For a given σ of length n, an arbitrary string of length $b(n)$ from the advice space U of size $2^{b(n)}$ is written to the start of the advice tape such that the prefix of the string corresponds to the definition of the advice for the algorithm and the given σ. The rest of the tape is arbitrary filled with bits. Using the request-answer system of [BDBK+90], we formally define deterministic online algorithms and randomized online algorithms in this semi-online advice framework as follows.

Definition 2.8 (Deterministic Online Algorithm with Semi-Online Advice)**.** *Given a request set R for some online problem, a deterministic online algorithm with semi-online advice with at most $b(n) \geq 0$ bits of advice in total for inputs of length n consists of*

- *a sequence of finite nonempty answer sets $A_1, \ldots,$*
- *a sequence of cost functions $cost_n : R^n \times A_1 \times A_2 \times \cdots \times A_n \to \mathbb{R}^+ \cup \{\infty\}$ for $n = 1, 2, \ldots,$*
- *a sequence of finite advice spaces U_n of size $|U_n| = 2^{b(n)}$,*
- *a sequence of query functions $u_n : R^n \to U_n$,*
- *a sequence of functions $s_i : R^i \times \{0,1\}^* \to \{\textsc{read}, \textsc{stop}\}$, and*
- *a sequence of action functions $g_i : R^i \times \{0,1\}^* \to A_i$ for $i = 1, 2, \ldots$.*

The algorithm has access to an infinite advice tape T which is a sequence of advice bits $t_1, t_2, \ldots$, where $\langle t_1, \ldots, t_{b(n)} \rangle := u_n(\sigma)$ for $n = |\sigma|$ and the remaining bits are arbitrary, and σ is the request sequence. Let B be the sequence of advice bits read from T that is initially empty. When request $r_i \in \sigma = r_1, r_2, \ldots$ is revealed, if $s_i(r_1, \ldots, r_i, B)$ is READ*, an additional bit is read from the advice tape and appended to B.This is repeated until $s_i(r_1, \ldots, r_i, B)$ is* STOP*. The action a_i of a online deterministic algorithm is $g_i(r_1, \ldots, r_i, B)$, i.e. a function of all the requests received so far and all the bits read from the advice tape by the algorithm (as determined by the function s_i) prior to r_{i+1} being revealed.*

The functions u_n and s_i are defined so as to guarantee that the algorithm does not read more than $b(n)$ bits of advice from T on any input of length n.

Definition 2.9 (Randomized Online Algorithm with Semi-Online Advice)**.** *A randomized online algorithm with at most $b(n)$ bits of advice in total is a distribution over deterministic online algorithms with at most $b(n)$ bits of advice.*

2.2.3 Comparing the Models with Advice

The semi-online advice model is a stronger model than that of the online advice model. Algorithms in the online advice model can be run in the semi-online advice model. The converse is not the case in general as the semi-online advice model allows for algorithms that use an amount of advice bits in total that is sublinear in n, the length of the request sequence, whereas the online advice model uses an amount of advice bits in total that is at least linear in n since a fixed amount of advice is received with each request.

In the semi-online advice model, it is possible to read all the advice bits prior to serving any requests. This implies that lower bounds in the semi-online advice model apply to the online advice model. Consider a lower bound such that $b(n)$ bits in total are required for a competitive ratio of r on request sequences of length n. Such a bound implies a lower bound in the online advice model of $b(n)/n$ advice bits per request for a competitive ratio of r.

2.3 Techniques Used in this Thesis

2.3.1 A Lower Bound Technique

This general lower bound technique was first presented in [EFKR11]. They defined a problem called the generalized matching pennies problem (GMP). The technique

presented in [EFKR11] provide lower bounds on b, the bits of advice per request, where $\Phi \geq 4$ is number of possible values of the penny and $1 \leq b \leq \frac{\log \Phi}{3}$ and applies to both randomized and deterministic algorithms. This technique was refined in [BHK+13] in that the restriction on Φ is improved such that $\Phi \geq 2$ and the range for b is increased to the full range of $[1, \log \Phi]$, but it only applies to deterministic algorithms. The problem in [BHK+13] is called the string guessing problem with known history (q-SGKH) and Φ from GMP is equivalent to q, the size of the alphabet, in q-SGKH. The formal definition of q-SGKH is as follows.

Definition 2.10. q-SGKH [BHK+13]. *The string guessing problem with known history over an alphabet Σ of size $q \geq 2$ (q-SGKH) is an online minimization problem. The input consists of n and a request sequence $\sigma = r_1, \ldots, r_n$ of the characters, in order, of an n length string. An online algorithm A outputs a sequence $a_1, \ldots, a_n$ such that $a_i = f_i(n, r_1, \ldots, r_{i-1}) \in \Sigma$ for some computable function f_i. Request r_i is revealed immediately after the algorithm outputs a_i. The cost of A is the Hamming distance between $a_1, \ldots, a_n$ and $r_1, \ldots, r_n$.*

The problem q-SGKH is useful for establishing lower bounds on the amount of advice required for a certain competitive ratio due to the following theorem.

Theorem 2.11. [BHK+13] *Consider an input string of length n for q-SGKH. The minimum number of advice bits per request for any online algorithm that is correct for more than αn characters, for $1/q \leq \alpha < 1$, is $(1 - H_q(1-\alpha)) \log_2 q$, where $H_q(p) = p \log_q(q-1) - p \log_q p - (1-p)\log_q(1-p)$ is the q-ary entropy function.*

For an online problem $\mathcal{P}$, a lower bound on the amount of advice require to achieve a given competitive ratio for deterministic algorithms can be established by first showing a reduction from the q-SGKH problem to the $\mathcal{P}$ problem. This is done by assuming that there is a ρ-competitive algorithm, P, for the $\mathcal{P}$ problem that uses b_P bits of advice per request. An algorithm, Q, for the q-SGKH problem that uses b_Q bits of advice per request uses P as a black box such that Q is correct for at least cn characters of the requests string, where $0 < c < 1$. For the reduction to work, $b_P|\sigma_{\mathcal{P}}| \leq b_Q|\sigma_{q\text{-SGKH}}|$, where $\sigma_{\mathcal{P}}$ is the request sequence created for P in the reduction and $\sigma_{q\text{-SGKH}}$ is the original request sequence for q-SGKH. Also, it is assumed that the bits of advice are received at the beginning before the request sequence is seen. Finally, a lower bound is established by applying Theorem 2.11 to Q with the appropriately set parameters for P.

2.3.2 PTAS Inspired Algorithms

The algorithms presented in this thesis for the bin packing problem, the dual bin packing problem, the scheduling problems on m machines and the reordering buffer management problem have the flavour of polynomial time approximation schemes (PTAS) in that the competitive ratio obtained is a function of some ε and the advice is a function of $1/\varepsilon$. A PTAS is an offline algorithm parametrized by an $\varepsilon > 0$, where the approximation ratio is a function of ε and the running time is a function of $1/\varepsilon$. In the advice case, we use more advice for a better competitive ratio and, for a PTAS, running time is sacrificed for a better approximation ratio.

For the offline versions of these problems, PTAS (or asymptotic PTAS [APTAS]) are known for the bin packing problem [FdlVL81], the dual bin packing problem [Kel99], the scheduling problems on m machines for the different objective functions considered [HS87, Woe97, AAWY97]. No PTAS is known for the reordering buffer management problem and the algorithm presented in this thesis may provide some insight in how to develop a PTAS.

Common to all our algorithms for the packing and scheduling problems is the technique of classifying the input items, according to their size, into a constant number of classes, depending on ε. For the bin packing problem and the dual bin packing problem, there are a constant number of groups of a constant number of items, both depending on ε. For the scheduling problems, the sizes of the items in one class differ only by a multiplicative constant factor, depending on ε. We classify all the items but the smallest ones (bin packing and scheduling) or a subset of the items (dual bin packing) in this way, where, for bin packing and scheduling, the bound on the size of the items not classified again depends on ε, and, for dual bin packing, the subset of items classified depends on an optimal packing and ε. This classification is done explicitly in the scheduling algorithms, and implicitly in the packing algorithms. We then consider an optimal packing (or schedule) for the input sequence and define *patterns* for the bins (the machines) that describe how the critically sized items (jobs) are packed (scheduled). The advice bits indicate with each input item into which bin (machine) pattern it should be packed (scheduled). For packing problems, all but the largest classified items can be packed into the optimal number of bins, according to the assigned pattern. The remaining items cause an ε increase in the number of bins used or items rejected.

For the scheduling problems, since items in the same class are "similar" in size, we can schedule the items such that the class of the items of each machine matches the class of those in the optimal while being within an ε factor of the optimal. For bin packing and scheduling, the very small items (jobs) have to be treated separately as are the non-classified items for the dual bin packing problem; in both cases, the items (jobs) are packed (scheduled) while remaining with an ε factor of the optimal.

Our techniques for these algorithms are similar to those of [FdlVL81, HS87, Woe97, AAWY97, Kel99]. In particular, we use the technique of rounding and grouping the items. The main difficulty in getting our algorithms to work stems from the fact that we must encode the necessary information using only a constant number of advice bits per request. In particular, the number of advice bits per request cannot depend on the size of the input or the size of the instance (number of bins/machines). Further, for the online advice mode, the advice is received per request and this presents additional challenges as the advice has to be presented sequentially per request such that the algorithm will be able to schedule the items in an online manner.

3 The k-Server Problem

In this chapter, we consider the k-server problem (see Section 3.1 for a formal definition) on general metric spaces and on finite trees. We present an upper bound on the competitive ratio of $\left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$ for the k-server problem on general metric spaces, using b bits of advice per request for $3 \leq b \leq \log k$. This is an improvement of about a factor of 2 over the previous best known upper bound. Also, we present a 1-competitive deterministic algorithm for metric spaces of finite trees, where the advice per request is $2 + 2\lceil \log(p+1) \rceil$ and p is the caterpillar dimension of the tree.

If $\log k < b$, where b is the bits of advice per request, then the trivial algorithm with advice, i.e. encoding the server id used by an optimal algorithm per request could be used instead of the algorithms presented in this chapter.

3.1 Preliminaries

The *k-server problem* consists of a metric space, $\mathcal{M}$, k mobile servers, an initial configuration of servers, and a finite request sequence, σ. Let $\mathcal{M} = (M, d)$, where M is a set of nodes, $d : M \times M \to \mathbb{R}^+$ is a distance function on M and $|M| = N > k$. Each request of σ will be to a node of $\mathcal{M}$, and a server must be moved to the requested node before the algorithm will receive the subsequent request. The goal is to minimize the total distance travelled by the k servers over σ.

Definition 3.1 (k-Server Configuration)**.** *A* configuration *is a multiset X such that $|X| = k$ and, for all $x \in X$, $x \in M$. Each $x \in X$ indicates the position one of the k servers in the metric space M.*

Definition 3.2 (Lazy k-Server Algorithm)**.** *A lazy k-server algorithm is an algorithm that, upon each request, only moves a single server to the request if it is uncovered.*

The following definitions for *adjacent* servers, *caterpillar dimension* and *caterpillar decomposition* only apply to metric spaces that are trees. Let T be a metric space which is a tree, $E(T)$ denote the edges of T and $V(T)$ denote the nodes of T.

Definition 3.3 (Adjacent Server)**.** *For T, server s, is* adjacent *to a request, r_i, if, along the unique path between the positions of s and r_i, there are no other servers.*

The *caterpillar dimension* of T with root r is denoted by $\mathrm{cdim}(T, r)$, or simply $\mathrm{cdim}(T)$ when the rooting of the tree is clear from the context. We will define it as it is defined in [Mat99] which is as follows.

Definition 3.4 (Caterpillar dimension of a rooted tree T)**.** *For T composed of a single node, $cdim(T) = 0$.*

For T, with two or more nodes, $cdim(T) = m + 1$ if there exist edge disjoint paths, $P_1, \ldots, P_q$, beginning at the root r such that each component T_j of $T \setminus E(P_1) \setminus \ldots \setminus E(P_q)$ has $cdim(T_j) \leq m$. The components T_j are rooted at their unique vertex lying on some P_i.

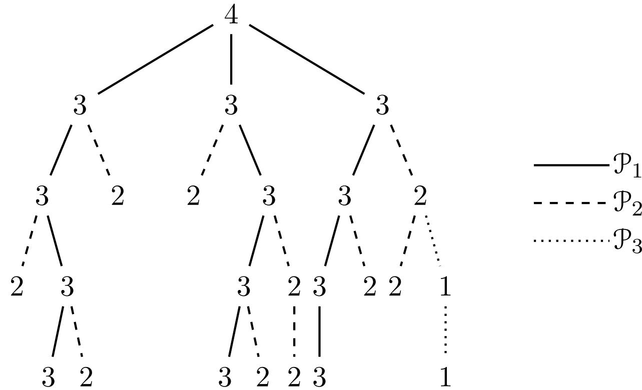

Figure 3.1: An illustration of a caterpillar decomposition of a tree, where the nodes are marked by their assigned path level. $\mathcal{P}_1$ is the set of edge disjoint paths rooted at the root, $\mathcal{P}_2$ is the set of edge disjoint paths rooted at a node of some path $\in \mathcal{P}_1$ and $\mathcal{P}_3$ is the set of edge disjoint paths rooted at a node of some path $\in \mathcal{P}_2$.

As described in the definition of the caterpillar dimension of T, the decomposition of T into edge disjoint paths is called the *caterpillar decomposition* of the tree. Given a

caterpillar decomposition of T, each node will be assigned a *path level*. For each level of the decomposition, the paths, $P_1, \ldots, P_q$ are distinguished and m is the maximum caterpillar dimension of the components T_j. All the nodes of P_i, $1 \leq i \leq q$, except the root, are assigned path level $m+1$. The root is assigned path level $m+2$ (see Figure 3.1). The formal recursive definition is as follows.

Definition 3.5 (Path Level of node i)**.** *Given a caterpillar decomposition of T, consider the edge disjoint paths, $P_1, \ldots, P_q$, beginning at the root r. If i is r, the path level of i is cdim(T) + 1, else, if $i \in \bigcup_{j=1}^{q} V(P_j)$, the path level of i is cdim(T). Otherwise, i belongs to a component T_j of $T \setminus E(P_1) \setminus \ldots \setminus E(P_q)$, and the path level of i is based on the caterpillar decomposition of T_j.*

Note that the root of the tree has a path level one more than the caterpillar dimension of the tree. In what follows, we refer to the caterpillar dimension of unrooted trees as defined in the following manner.

Definition 3.6 (Caterpillar dimension of an unrooted tree G)**.** *Given G, the caterpillar dimension of G is the minimum over all nodes, $v \in V(G)$, of the caterpillar dimension of G when rooted at v. That is, cdim(G) = $\min_{v \in V(G)}$ cdim(G, v) .*

In this chapter, with a risk of a slight abuse of notation, we will note the cost of a subsequence of σ to an online algorithm, ALG, as

$$\begin{aligned}\text{ALG}(r_i, \ldots, r_j) = \text{cost}_j(r_1, \ldots, r_j, \text{ALG}[r_1, \ldots, r_j]) \\ - \text{cost}_{i-1}(r_1, \ldots, r_{i-1}, \text{ALG}[r_1, \ldots, r_{i-1}]) \ ,\end{aligned}$$

where $r_i, \ldots, r_j$ is a subsequence of σ containing all the requests from r_i to $r_j \in \sigma$, and the prefix is understood implicitly.

3.2 An Upper Bound for General Metric Spaces

In this section, we present a $\left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$-competitive deterministic online algorithm with advice, called CHASE, for the k-server problem on general metric spaces with b bits of advice per request, where $b \geq 3$. For convenience of notation, we use $\alpha = \left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$.

In order to clearly present the algorithm and the proof, we will first design and analyze the algorithm such that it gets a variable number of bits of advice with each request. The algorithm will receive at least 3 bits of advice with each request, and the

total number of advice bits will not exceed bn for any prefix of n requests. Afterwards, we will show how to adapt the algorithm so that it gets at most b bits of advice with each request using a transformation of [BKKK11].

Roughly speaking, our algorithm works as follows: given a request sequence, σ, we consider an optimal algorithm for this sequence. Based on this optimal algorithm, we partition σ into k subsequences, σ^s for $1 \leq s \leq k$, such that all the requests of σ^s are served according to the optimal algorithm by server s. With $\lceil \log k \rceil$ bits of advice per request, we can indicate, with each request of σ^s, the identity of the server s, and, thus, our online algorithm with advice would precisely follow the optimal algorithm. If, however, we have only $b < \lceil \log k \rceil$ bits of advice per request, we can do that only roughly every $\lceil \log k \rceil / b$ requests of σ^s. We call these requests *anchors*. The rest of the requests of σ^s are served in a greedy manner, i.e. they are served by the closest server to the request which then returns to its previous position. By serving requests in this way, server s always stays at its last anchor. Thus, the cost of serving the $(\lceil \log k \rceil / b) - 1$ non-anchor requests of σ^s between any two anchors is bounded from above by $2 \lceil \log k \rceil / b$ times the distance from the last anchor to the furthest non-anchor request. This gives us a competitive ratio of $O(\log k / b)$. Some fine tuning of the above ideas gives us our result. In what follows, we formally define the algorithm and prove its competitive ratio.

Algorithm CHASE: At the beginning, all servers are unmarked.
Given a request, r_j, and the advice, do:

- If the advice is 00, serve r_j with the closest server to r_j and return it to its previous position.
- If the advice is 10, serve r_j with the closest unmarked server and mark this server. Do not return the server to its previous position.
- If the advice is $11t$, where t is a server number encoded in $\lceil \log k \rceil$ bits, serve the request with server number t.

In order to define the advice, we will fix an optimal algorithm, OPT, that we assume to be a lazy algorithm. We will then partition the request sequence into k subsequences, $\sigma^1, \ldots, \sigma^k$, where σ^s is the trace of the server s in OPT. In other words, σ^s consists of the requests served by server s in the lazy optimum. It should be noted that the

requests of σ^s are not necessarily consecutive requests in σ. Let r_j^s be the j-th request served by server s over σ^s. Recall that $\alpha = \left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$. Independently, for each server, we choose an index $1 \leq q^s \leq \alpha$. The way to choose this index will be defined later. The request sequence σ^s is divided into α-length cycles starting at $r_{q^s+1}^s$. We will denote the i-th cycle of σ^s by c_i^s. The first cycle, c_1^s, which starts at request r_1^s and ends at request $r_{q^s}^s$, may have a length less than α. Let C^s be the total number of cycles in σ^s.

The advice will be defined as follows for request r_j^s:

- 10, if $j = q^s$, i.e. the last request of the first cycle.
- $11t$, if $j = q^s + i\alpha$ for some $i \geq 1$, i.e. the last request of all cycles except the first one. Here, t is the server number that serves request $r_{q^s}^s$ in CHASE encoded in $\lceil \log k \rceil$ bits.
- 00, if $j \neq q^s + i\alpha$, i.e. everywhere else.

The first two bits of the advice per request will be referred to as the control bits.

First, we state a technical lemma that we will use in our proof.

Lemma 3.7. *Given a sequence of α non-negative values, $a_1, \ldots, a_\alpha$, there is an integral value, q, where $1 \leq q \leq \alpha$, such that*

$$\sum_{i=1}^{q}(2(q-i)+1)a_i + \sum_{i=q+1}^{\alpha}(2(\alpha+q-i)+1)a_i \leq \alpha \sum_{i=1}^{\alpha} a_i \ .$$

Proof. Summing the expression over all possible values of q, we get

$$\sum_{q=1}^{\alpha}\left[\sum_{i=1}^{q}(2(q-i)+1)a_i + \sum_{i=q+1}^{\alpha}(2(\alpha+q-i)+1)a_i\right]$$
$$= \sum_{q=1}^{\alpha}\left[\sum_{i=1}^{\alpha}(2(\alpha+q-i)+1)a_i - \sum_{i=1}^{q}2\alpha a_i\right]$$
$$= \left[\sum_{q=1}^{\alpha}\big(2(\alpha-q)+1\big)\right]\cdot\sum_{i=1}^{\alpha}a_i + \left[\sum_{q=1}^{\alpha}4q\right]\cdot\sum_{i=1}^{\alpha}a_i - \sum_{q=1}^{\alpha}\sum_{i=1}^{\alpha}2ia_i - \sum_{q=1}^{\alpha}\sum_{i=1}^{q}2\alpha a_i$$

$$= \left[\sum_{q=1}^{\alpha} \big(2(\alpha - q) + 1\big)\right] \cdot \sum_{i=1}^{\alpha} a_i + (2\alpha^2 + 2\alpha) \sum_{i=1}^{\alpha} a_i - \left[2\alpha \sum_{i=1}^{\alpha} i a_i + 2\alpha \sum_{i=1}^{\alpha} (\alpha - i + 1) a_i\right]$$
$$= \left[\sum_{q=1}^{\alpha} \big(2(\alpha - q) + 1\big)\right] \cdot \sum_{i=1}^{\alpha} a_i + (2\alpha^2 + 2\alpha) \sum_{i=1}^{\alpha} a_i - (2\alpha^2 + 2\alpha) \sum_{i=1}^{\alpha} a_i$$
$$= \left[\sum_{q=1}^{\alpha} \big(2(\alpha - q) + 1\big)\right] \cdot \sum_{i=1}^{\alpha} a_i$$
$$= \alpha^2 \sum_{i=1}^{\alpha} a_i \ .$$

It follows that one of the α possible values of q gives at most the average value, i.e. $\alpha \sum_{i=1}^{\alpha} a_i$. The lemma follows.

□

Now, we prove the main theorem of this section.

Theorem 3.8. *For every $b \geq 3$, algorithm* CHASE *is a $\left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$-competitive k-server algorithm for general metric spaces with b bits of advice per request.*

Proof. For the proof, we will compare the cost of CHASE and OPT separately for every subsequence σ^s, and cycle by cycle within each σ^s. Recall that $\alpha = \left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$. Note that the first cycle and the last cycle may be of length less than α.

If the last cycle is less than α for some σ^s, we can repeat the last request of σ^s until the last cycle is of length α. The repeated requests for each server should be appended to σ. Let σ' be the request sequence with the additional requests. Clearly, CHASE$(\sigma) \leq$ CHASE(σ'). As the additional requests of σ' will be to nodes containing the servers of OPT, OPT(σ') = OPT(σ). Hence, CHASE$(\sigma) \leq \alpha$OPT(σ) if CHASE$(\sigma') \leq \alpha$OPT(σ'). Therefore, for this proof, we can assume without loss of generality that the last cycle for each server is of length α.

Consider the i-th cycle of server s in OPT for $i > 1$ (we will deal with the first cycle later). Figure 3.2 illustrates the idea of the upper bound for the i-th cycle. Let t be the server in CHASE that serves request $r^s_{q^s}$. We will denote the position of $r^s_{(i-2)\alpha+q^s}$, the last request of the previous cycle, by INIT^s_i. We claim that, just before the cycle starts, both OPT and CHASE will have a server at INIT^s_i. This is true because the advice for request $r^s_{(i-2)\alpha+q^s}$ indicates to CHASE to bring server t to INIT^s_i and, by the definition of the algorithm, t will always return to INIT^s_i between $r^s_{(i-2)\alpha+q^s}$ and $r^s_{(i-2)\alpha+q^s+1}$. By the definition of the subsequence σ^s, OPT serves $r^s_{(i-2)\alpha+q^s}$ with s and does not move s between request $r^s_{(i-2)\alpha+q^s}$ and request $r^s_{(i-2)\alpha+q^s+1}$.

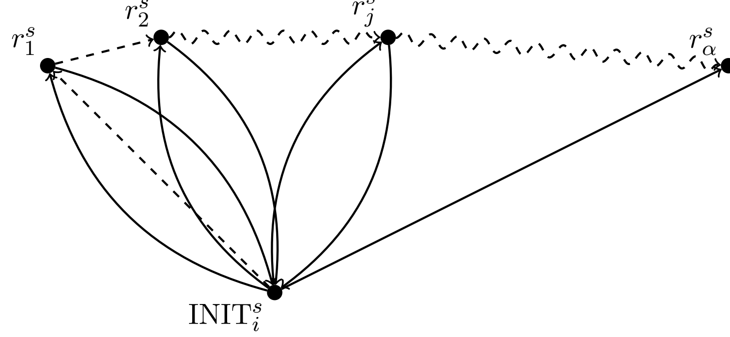

Figure 3.2: An illustration of the i-th cycle of server s as served by OPT and bounded by CHASE in the worst case. The dashed line represents the trace of server s in OPT for the i-th cycle and the solid line represents the worst case trace of the server in CHASE to serve the requests of the i-th cycle of server s. INIT_i^s is the last request of the $(i-1)$-th cycle and $r_1^s, \ldots, r_j^s, \ldots, r_\alpha^s$ are the requests of the i-th cycle of server s.

Also, observe that just before each of the requests between $r^s_{(i-2)\alpha+q^s+1}$ and $r^s_{(i-1)\alpha+q^s}$ inclusive, i.e. the requests of the i-th cycle, server t of CHASE is at INIT_i^s. Recall that CHASE serves these requests except the last one by using the closest server to the request and, then, returns that server to its prior position. Therefore, the cost to CHASE for any request $r^s_{(i-2)\alpha+q^s+j}$, where $1 \leq j \leq \alpha - 1$, i.e. the requests of cycle i except the last one, is

$$\text{CHASE}(r^s_{(i-2)\alpha+q^s+j}) \leq 2d(\text{INIT}_i^s, r^s_{(i-2)\alpha+q^s+j}) \ . \tag{3.1}$$

By the triangle inequality and Inequality (3.1),

$$\text{CHASE}(r^s_{(i-2)\alpha+q^s+j}) \leq 2\sum_{l=1}^{j} d(r^s_{(i-2)\alpha+q^s+l-1}, r^s_{(i-2)\alpha+q^s+l}) \ . \tag{3.2}$$

For request $r^s_{(i-1)\alpha+q^s}$, i.e. the last request of cycle i, CHASE serves the request using server t that is at $r^s_{(i-2)\alpha+q^s}$. We have, by the triangle inequality,

$$\begin{aligned}\text{CHASE}(r^s_{(i-1)\alpha+q^s}) &= d(\text{INIT}_i^s, r^s_{(i-1)\alpha+q^s}) \\ &\leq \sum_{l=1}^{\alpha} d(r^s_{(i-2)\alpha+q^s+l-1}, r^s_{(i-2)\alpha+q^s+l}) \ . \end{aligned} \tag{3.3}$$

Observe that the cost of OPT to serve $r^s_{(i-2)\alpha+q^s+j}$ for $1 \leq j \leq \alpha$, i.e. the requests of cycle i, is $d(r^s_{(i-2)\alpha+q^s+j-1}, r^s_{(i-2)\alpha+q^s+j})$. Using this fact and Inequalities (3.2) and

(3.3), we can bound the cost of CHASE over a cycle by the cost of OPT as follows:

$$\sum_{j=1}^{\alpha} \text{CHASE}(r^s_{(i-2)\alpha+q^s+j}) \leq \sum_{j=1}^{\alpha-1} \left(2 \sum_{l=1}^{j} \text{OPT}(r^s_{(i-2)\alpha+q^s+l})\right) + \sum_{l=1}^{\alpha} \text{OPT}(r^s_{(i-2)\alpha+q^s+l}) = \sum_{j=1}^{\alpha} [2(\alpha - j) + 1]\text{OPT}(r^s_{(i-2)\alpha+q^s+j}) \ . \tag{3.4}$$

The analysis of the first cycle is, essentially, the same as the analysis of the i-th cycle, $i > 1$, with the exception that an additive constant is introduced per request of the first cycle. The additive constant results from the fact that, during the first cycle of σ^s, CHASE does not necessarily maintain a server at the initial position of s. Recall that in CHASE, s may have been used to follow the trace of another server of OPT that is not s. In such a case, s would be marked and may be at a position in the metric space that is not the initial position of s. Nevertheless, by the definition of CHASE, there will always be an unmarked server in one of the locations of the initial configuration. Let Δ be the diameter of the initial configuration. Therefore, for any request of the first cycle, r^s_l, of σ^s, analogously to Inequality (3.2), we have

$$\text{CHASE}(r^s_l) \leq 2\Big(\Delta + \sum_{m=1}^{l} d(r^s_{m-1}, r^s_m)\Big) \ , \tag{3.5}$$

where r^s_0 is the initial position of s. Analogous to Inequality (3.4), summing Inequality (3.5) over all requests of the first cycle of s, gives

$$\sum_{l=1}^{q^s} \text{CHASE}(r^s_l) \leq \sum_{l=1}^{q^s} [2(q^s - l) + 1]\text{OPT}(r^s_l) + 2\alpha\Delta \ . \tag{3.6}$$

Note that the first cycle is of length $q^s \leq \alpha$. If we define the cost for requests with indexes less than 0 to be 0 for both OPT and CHASE, we can rewrite Inequality (3.6) to be more congruent with Inequality (3.4) as follows:

$$\sum_{j=1}^{\alpha} \text{CHASE}(r^s_{-\alpha+q^s+j}) \leq \sum_{j=1}^{\alpha} [2(\alpha - j) + 1]\text{OPT}(r^s_{-\alpha+q^s+j}) + 2\alpha\Delta \ . \tag{3.7}$$

Using Inequalities (3.4) and (3.7), and summing over all cycles, gives

$$\text{CHASE}(\sigma^s) \leq \sum_{i=1}^{C^s} \sum_{j=1}^{\alpha} [2(\alpha - j) + 1]\text{OPT}(r^s_{(i-2)\alpha+q^s+j}) + 2\alpha\Delta \ . \tag{3.8}$$

Now, we deal with assigning the values of q^s. Define $a_1, \ldots, a_\alpha$ to be $a_j = \sum_{i=1}^{C^s} \text{OPT}(r^s_{(i-1)\alpha+j})$, i.e. the cost of OPT for the requests in σ^s in jumps of α requests. We can rewrite Inequality (3.8) as

$$\text{CHASE}(\sigma^s) \leq \sum_{i=1}^{q^s} (2(q^s - i) + 1)a_i + \sum_{i=q^s+1}^{\alpha} (2(\alpha + q^s - i) + 1)a_i + 2\alpha\Delta \; . \tag{3.9}$$

By Lemma 3.7, there is a value $1 \leq q^s \leq \alpha$ such that

$$\text{CHASE}(\sigma^s) \leq \alpha \sum_{i=1}^{\alpha} a_i + 2\alpha\Delta = \alpha\text{OPT}(\sigma^s) + 2\alpha\Delta \; .$$

We chose this q^s separately for each server s in order to define the cycles. Summing over all k subsequences σ^s concludes the proof of the competitive ratio.

Finally, we show that the algorithm uses at most bn bits of advice over any prefix of n requests. There are two control bits with each request. Let t be the server in CHASE that serves $r^s_{q^s}$, i.e. the last request of the first cycle of σ^s. There are at least α requests of σ^s between any two requests, where the id of t is given in the advice. Since $\alpha = \left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$, the claim follows.

□

In order to adapt the algorithm so that it receives b bits of advice per request, we use a transformation of [BKKK11]. Two control bits will be given with each request, and the remaining $b - 2$ bits will contain portions of server ids. The control bits will be as defined previously. We then define a string as the concatenation of all server ids given for the whole sequence. This string will be broken into $(b - 2)$-bit chunks and a single chunk will be given with each request. The algorithm can store these $(b - 2)$-bit chunks in a FIFO queue and will have $\lceil \log k \rceil$ bits available to be read from the queue when dictated by the control bits.

3.3 k-Server with Advice on Trees

In this section, we describe a deterministic online algorithm with advice for the k-server problem on finite trees, called PATH-COVER, that is 1-competitive and uses $2+2\lceil \log(p+1) \rceil$ bits of advice per request, where p denotes the caterpillar dimension of the tree.

The algorithm and advice are such that the actions of the algorithm will mimic the actions of a non-lazy optimal algorithm, denoted by OPT_{nl} which has specific properties

with respect to the given request sequence. First, we will describe the construction of the non-lazy algorithm and show that it has optimal cost. Then, we will analyze the online algorithm with advice based on OPT_{nl}.

In this section, the definition of the algorithm, the advice and OPT_{nl} is based on the caterpillar decomposition of the tree that minimizes the caterpillar dimension. If there is not a unique caterpillar decomposition that minimizes the caterpillar dimension, one of the minimizing caterpillar decompositions can be chosen arbitrarily to define the algorithm, the advice and OPT_{nl}. Further, in this section, an ancestor of a node is with respect to the root used for the chosen caterpillar decomposition and we assume that a node is an ancestor of itself.

The intuition for our algorithm can be seen most easily using the height of a rooted tree, h, instead of the caterpillar dimension. Consider any lazy optimal algorithm on the tree. A server move on the tree of such an algorithm can be broken into two parts. The first part is to the lowest common ancestor of the server and the request and the second part is to the request. Such an optimum can be altered into a non-lazy optimal algorithm that before the initial request and immediately after serving each request, it moves the server to the lowest common ancestor between the server's current position and the next request it would serve in the lazy optimum. $2 \log h$ bits of advice are sufficient to communicate the height of the node containing the server used for the request in the non-lazy optimum and the height of the lowest common ancestor between this request and the next request handled by the active server in the non-lazy optimum. Therefore, modulo an initialization phase, an algorithm with $2 \log h$ bits of advice can follow the non-lazy algorithm for each request by using a server at the position indicated by the advice and, after serving the request, moving the server to the second position indicated by the advice. For the algorithm described below, we choose the caterpillar dimension over the height of the tree since it gives a 1-competitive algorithm with a constant number of bits of advice per request for degenerate trees such as the line or a caterpillar. Furthermore, the caterpillar dimension is at most the height of the tree, and is at most $\log N$, where N is the number of nodes in the tree [Mat99].

3.3.1 Non-Lazy Optimum

Let OPT_l be any lazy optimum for the given request sequence. For a given caterpillar decomposition of the tree, OPT_{nl} will be constructed from OPT_l such that OPT_{nl} has

three properties. Let s be the server used by OPT_{nl} to serve request r_i. Then, given that OPT_{nl} can choose the starting configuration, the three properties are:

1. Immediately after serving r_i and before serving r_{i+1}, s and only s can be moved, and, if s is moved, it can only be to the nearest node of a higher path level in the tree.

2. The position of s, immediately before r_i is given, is at the same path level or higher than r_i.

3. Immediately before r_i is given, s is adjacent to r_i.

Let T be the tree representing the metric space $\mathcal{M}$. Given the caterpillar decomposition of T that minimizes $\text{cdim}(T)$ as described above, we first construct OPT'_{nl} which has the first two properties. For any $u, v \in T$, let $\text{maxPath}(u, v)$ be the ancestor of u with the maximum path level on the path between u and v (see Figure 3.3). The initial position of each server, $1 \leq s \leq k$, in OPT'_{nl} is $\text{maxPath}(u_s, v_s)$, where u_s is the initial position of server s in OPT_l and v_s is the position of the first request served by s in OPT_l.

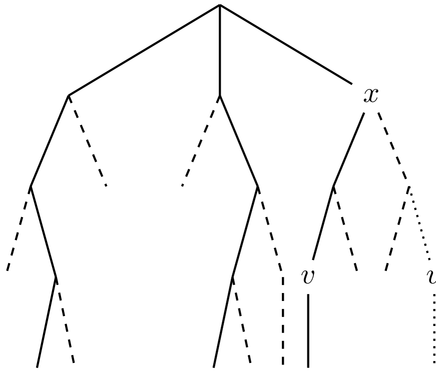

Figure 3.3: An illustration of $x = \text{maxPath}(u, v)$, using the caterpillar decomposition from Figure 3.1.

For each request, r_i in σ, let s_i be the server used by OPT_l to serve r_i. For r_i, OPT'_{nl} will:

1. Serve r_i with s_i.

2. Immediately after serving r_i, move s_i to $\text{maxPath}(r_i, r_j)$, where r_j is the next request served by s_i in OPT_l.

Fact 3.9. *OPT'_{nl} has the first two properties.*

Proof. The definition of the algorithm guarantees Property 1.

As OPT'_{nl} uses the same server as OPT_l for each request, the initial configuration of OPT'_{nl} guarantees that each server s will be at the same level or higher than the first request it serves; and the second step for r_i guarantees Property 2 for the rest of the request sequence.

□

Fact 3.10. *For any σ, $\text{OPT}'_{nl}(\sigma) = \text{OPT}_l(\sigma)$.*

Proof. The claim follows from the fact that the trajectories followed by each of the servers according to OPT'_{nl} and OPT_l are the same. The only difference being that some of the moves are done earlier in OPT'_{nl} than OPT_l.

□

Next, we construct OPT_{nl} from OPT'_{nl} so that Property 3 will be satisfied, properties 1 and 2 will be maintained and the cost will not increase.

Intuitively, we will iterate over all the server moves of OPT'_{nl} and, in each instance such that a non-adjacent server moves to a request, we will swap the non-adjacent server and the adjacent server, i.e. the adjacent server will serve the request and the non-adjacent server will be used for the next move of the adjacent server.

OPT_{nl} will be defined by induction on the server moves of OPT'_{nl}. Let $T^* = (t_1^*, q_1), \ldots, (t_m^*, q_m)$ be the sequence, in order, of server moves performed by OPT'_{nl} such that the ordered pair (x, y) represents a move of a server from position x to position y in the metric space. Note that $m \geq n$. All requests are represented in the sequence even if there is a server at the position of a request. In that case, the two positions of the ordered pair are the same. We will build T^i, which is a sequence of location pairs representing the server moves of an optimal algorithm, inductively for $i \geq 0$ where $T^0 = T^*$. For each T^i, $i \geq 0$, Property 1 and Property 2 hold for the entire sequence and Property 3 holds for the subsequence $(t_1^i, q_1), \ldots, (t_i^i, q_i)$.

Assume that T^{i-1} is defined and has the above properties. This is trivially true for T^0. In order to construct T^i, we need to first consider q_i. If q_i is not a request, then $T^i = T^{i-1}$. If q_i is a request and, in the configuration obtained in T^{i-1} after the $(i-1)$-th pair, the server at t_i^{i-1} is adjacent to q_i, then $T^i = T^{i-1}$. Otherwise, there is a server at some position u_i between t_i^{i-1} and q_i. In this case, T^i will be defined as

T^{i-1} from q_1 to q_{i-1}. That is, for all $1 \leq j < i$, $(t^i_j, q_j) = (t^{i-1}_j, q_j)$. The server at u_i will be used in T^i for request q_i, i.e. $(t^i_i, q_i) = (u_i, q_i)$. The remainder of the server moves will be the same in T^i as T^{i-1} except for the *next* time after i that a server at the position u_i is moved in T^{i-1}. In that case, a server at position t^{i-1}_i will be used. More formally, let l be the *first* index of the ordered pairs of T^{i-1} where $t^{i-1}_l = u_i$ such that $i < l \leq m$. In T^i, $(t^i_l, q_l) = (t^{i-1}_i, q_l)$.

Lemma 3.11. *For a given caterpillar decomposition of the tree, the cost of* OPT_{nl} *is no more than the cost of* OPT_l*, and* OPT_{nl} *has three properties. Let s be the server used by* OPT_{nl} *to serve request r_i. The three properties are:*

1. *Immediately after serving r_i and before serving r_{i+1}, s and only s can be moved, and, if s is moved, it can only be to the nearest node of a higher path level in the tree.*

2. *The position of s, immediately before r_i is given, is at the same path level or higher than r_i.*

3. *Immediately before r_i is given, s is adjacent to r_i.*

Proof. We will prove by induction on i that, for T^i, the cost does not increase compared to OPT_l, Property 1 and Property 2 hold for the enter sequence T^i, and Property 3 holds for $(t^i_1, q_1), \ldots, (t^i_i, q_i)$.

Cost: We will show by induction on i that the cost of T^i is no more than OPT_l. For $i = 0$, as T^0 is OPT'_{nl}, the claim follows from Fact 3.10. For the inductive step from T^{i-1} to T^i, note that the two sequences differ by at most two moves in the construction above. If $T^{i-1} = T^i$, then the cost remains the same between T^{i-1} and T^i. Using the notation in the construction above, if the two sequences differ at a single move then it is the move at index i. T^i uses a server at position u_i which is on the path between t^{i-1}_i and q_i and, therefore, the cost of T^i is less than that of T^{i-1}. If two moves differ between T^{i-1} and T^i, then they are the moves at indexes i and l. The cost to T^i for i and l is

$$\begin{aligned}
&d(u_i, q_i) + d(t^{i-1}_i, q_l) \\
&\qquad \leq d(u_i, q_i) + d(t^{i-1}_i, u_i) + d(u_i, q_l) \text{ , by the triangle inequality,} \\
&\qquad = d(t^{i-1}_i, q_i) + d(u_i, q_l) \text{ , as } u_i \text{ is on the path between } t^{i-1}_i \text{ and } q_i, \\
&\qquad = d(t^{i-1}_i, q_i) + d(t^{i-1}_l, q_l) \text{ , as } u_i = t^{i-1}_l,
\end{aligned}$$

which is the cost to T^{i-1} for i and l.

Property 1: Property 1 restricts the moves that an algorithm can make between serving requests. Specifically, the algorithm can only move a server from the current request. The intuition of the proof is that, in the construction above, the only moves that differ between T^{i-1} and T^i are the moves to serve requests while any moves between requests are consistent between T^{i-1} and T^i.

We will show that Property 1 holds for the entirety of T^i by induction on i. The claim is trivial for T^0 as Property 1 holds for OPT'_{nl}. For the inductive step from T^{i-1} to T^i, if $T^i = T^{i-1}$, the claim must hold as it was true for T^{i-1}. If only the move at index i differs between T^{i-1} and T^i, then, according to the construction, this move must be that of a server to a request. Specifically, the difference between T^{i-1} and T^i is the position of the server that serves r_i, i.e. the first item of the i-th ordered pair is the only difference between T^{i-1} and T^i. So, the claim holds as it was true for T^{i-1}. If the moves at index i and l, as defined in the construction above, differ between T^{i-1} and T^i, the move at index i is that of a server to a request as in the previous case. The second change is the first move of the server at u_i, in T^{i-1}, after (t_i^{i-l}, q_i) which must be to a request in T^{i-1} as Property 1 holds for T^{i-1}. Therefore, the second move changed between T^{i-1} and T^i is of a server to a request as well, which implies that Property 1 holds for the whole sequence T^i.

Property 2: By induction on i, we will show that Property 2 holds for the entirety of T^i. The claim is trivial for T^0 as Property 2 holds for OPT'_{nl}. For the inductive step from T^{i-1} to T^i, if $T^i = T^{i-1}$, the claim holds as it was true for T^{i-1}. If only the move at index i differs between T^{i-1} and T^i, then the claim holds as the different move, using the notation above, uses a server at u_i instead of t_i^{i-1}, and by the construction above, u_i is on the path from t_i^{i-1} to q_i. Therefore, u_i must be at the same path level or higher than that of q_i due to the monotonic nature of the caterpillar decomposition and that t_i^{i-1} is at the same path level or higher than q_i which follows from the induction hypothesis. If both the moves at indexes i and l, as defined in the construction above, differ between T^{i-1} and T^i, then the claim holds for the move at index i by the same argument of the previous case. For the difference at index l, the move must be that of a server to a request (as shown in the proof for Property 1). As shown in the previous case, t_i^{i-1} is at the same path level or higher than u_i. Therefore, t_i^{i-1} will be at the same path level or higher than any subsequent request served by the server at u_i in T^{i-1} since Property 2 holds for the whole of T^{i-1}. Hence, Property 2 holds for the whole sequence T^i.

Property 3: The fact that Property 3 holds for $(t_1^i, q_1), \ldots, (t_i^i, q_i)$ is immediate from the inductive construction.

□

3.3.2 Algorithm PATH-COVER

There will be two stages to the algorithm. The initial stage will be for the first k requests and will be used to match the configuration of PATH-COVER to that of OPT_{nl} as defined in the previous section. Over the remaining requests, PATH-COVER will be designed to act exactly as OPT_{nl}. PATH-COVER will receive $2(l+1)$ bits of advice per request, where $l = \lceil \log(p+1) \rceil$ and p is the minimal caterpillar dimension of the tree. The advice will be of the form $wxyz$, where w and x will be 1 bit in length, and y and z will be l bits in length.

Note that, in the definition of the algorithm and the advice that follows, we assume that the servers are arbitrarily numbered from 1 to k.

3.3.2.1 Algorithm and Advice for $r_1, \ldots, r_k$

From r_1 to r_k, PATH-COVER will serve each request with the nearest server regardless of the advice. As for the advice, for request r_i, where $1 \leq i \leq k$,

- if $w = 1$, the algorithm stores the ancestor node nearest r_i which has the path level y.
- if $x = 1$, the algorithm stores the ancestor node nearest the initial position of server i which has the path level z.

Note that both w and x can be 1 for request r_i. Immediately after serving r_k, PATH-COVER will use the first k stored nodes as a server configuration and will move to this configuration at minimal cost (minimum matching).

Immediately before serving r_{k+1}, the k servers of OPT_{nl} will be at their initial position or a higher path level; or at one of the first k requests or a higher path level. The advice over the first k requests is defined so as to encode the configuration of OPT_{nl} immediately before OPT_{nl} serves r_{k+1}. For $1 \leq i \leq k$, the advice for r_i will be defined as follows:

w:	1,	if the server used for r_i in OPT$_{nl}$ does not serve another request up to r_k.
	0,	otherwise
x:	1,	if server i does not serve any of the first k requests in OPT$_{nl}$.
	0,	otherwise
y:		A number in binary indicating the path level of the node to which the server used for r_i is moved to in OPT$_{nl}$ after serving r_i.
z:		A number in binary indicating the path level of the node to which server i is moved to before r_1 in OPT$_{nl}$.

Note that, over the first k requests, w and x will be 1 a total of k times (once for each of the k servers of OPT$_{nl}$). For r_i, when w is 1, this implies that, immediately before r_{k+1}, OPT$_{nl}$ will have a server at the nearest ancestor of r_i with path level y. When x is 1, this implies that, immediately before r_{k+1}, the server i of OPT$_{nl}$ will be at the nearest ancestor of the initial position of server i with path level z. Observe that this indeed encodes the configuration of OPT$_{nl}$ just before r_{k+1} in the advice bits of the first k requests.

3.3.2.2 Algorithm and Advice for $r_{k+1}, \ldots, r_n$

From r_{k+1} to r_n, given a request, r_i, where $k+1 \leq i \leq n$, and the advice, let P be the path formed by all the adjacent nodes of path level y that includes the nearest ancestor of r_i with path level y. That is, P is the unique path from the chosen caterpillar decomposition of T with nodes of path level y that intersects the path between r_i and the root. Now, define a path, Q, on the tree as follows:

- if $x = 1$, Q runs from r_i to the end of P nearest the root.
- if $x = 0$, Q runs from r_i to the end of P furthest from the root.

PATH-COVER will serve r_i with the closest server along Q. After serving r_i, PATH-COVER will move this server to the nearest ancestor of r_i with path level z.

The advice is defined so that PATH-COVER and OPT$_{nl}$ will use a server from the same position for each request. For $k+1 \leq i \leq n$, the advice for r_i will be defined as follows:

w:	0	(not used)
x:		Let s be the position of the server used by OPT_{nl} to serve r_i, and let c be the ancestor of r_i with the same path level as s (see Figure 3.4).
	1,	if s is between c and the root of the tree.
	0,	otherwise.
y:		A number in binary indicating the path level of s.
z:		A number in binary indicating the path level of the node to which the server used for r_i is moved to in OPT_{nl} immediately after serving r_i.

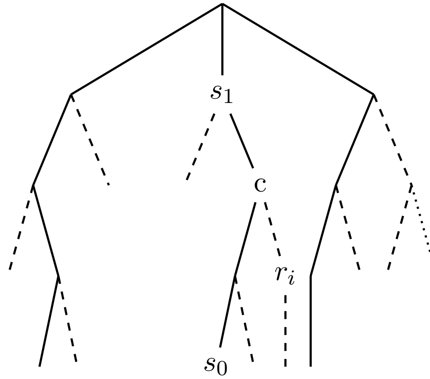

Figure 3.4: An illustration of the definition of the advice bit x for r_i for the PATH-COVER algorithm, where $k+1 \leq i \leq n$, using the caterpillar decomposition from Figure 3.1. s_1 represents a position of s such that $x = 1$ and s_0 represents a position of s such that $x = 0$.

Fact 3.12. *The algorithm uses* $2 + 2\lceil \log(p+1) \rceil$ *bits of advice per request, where* p *is the minimal caterpillar dimension of the tree.*

Proof. For each request, the bits of advice are composed of two control bits and two numbers encoded in binary, representing a path level. The two numbers encoded in binary range from 1 to $p+1$ where p is the minimal caterpillar dimension of the tree. □

3.3.2.3 Analysis of PATH-COVER

Theorem 3.13. PATH-COVER *is* 1*-competitive on finite trees.*

Proof. From r_1 to r_k, all the requests are served by the nearest server. This cost can be bounded by $k\Delta$, where Δ is the diameter of the tree. Immediately after serving r_k, PATH-COVER matches the configuration of OPT_{nl}. The cost to match a configuration can also be bounded by $k\Delta$.

We will show, by induction on $i \geq k+1$, that:

1. Just before r_i, the configuration of PATH-COVER and the configuration of OPT_{nl} match.

2. To serve r_i, they use a server from the same position.

By the definition of the algorithm and the advice, the configurations of PATH-COVER and OPT_{nl} match just before r_{k+1}. This establishes point 1 for the base case of r_{k+1}. Property 2 of OPT_{nl} guarantees that the position of the server, s_{k+1}, used by OPT_{nl} to serve r_{k+1} is at the same path level or higher that r_{k+1}. Property 3 of OPT_{nl} ensures that, just before serving r_{k+1}, there are no servers between s_{k+1} and r_{k+1}. Given that the configurations of PATH-COVER and OPT_{nl} match just before r_{k+1}, the first server, in PATH-COVER, along the path defined by the advice is at s_{k+1}. This establishes point 2 for the base case of r_{k+1}.

Assume that the induction hypothesis is true for j, $k+1 \leq j \leq i-1$. From the induction hypothesis, we know that the configurations of PATH-COVER and OPT_{nl} prior to serving r_{i-1} match, and that both PATH-COVER and OPT_{nl} will move a server from the same position in the tree to r_{i-1}. Therefore, the configurations of PATH-COVER and OPT_{nl} still match upon serving r_{i-1}. Property 1 of OPT_{nl} guarantees that only the server used for r_{i-1} can be moved if there is a server moved between the time that r_{i-1} is served and just before r_i is served. In that case, by the definition of the advice and the algorithm, PATH-COVER will move the server located at r_{i-1} to the same position that it is moved in OPT_{nl}. Property 1 also guarantees that this is the only server moved by OPT_{nl}. Therefore, the configurations of PATH-COVER and OPT_{nl} will match immediately before r_i, proving point 1. As shown in the base case, properties 2 and 3 of OPT_{nl}, and the fact that the configurations of OPT_{nl} and PATH-COVER match just before r_i, ensures that the first server along the path defined by the advice is at the same position as the server used by OPT_{nl}. This proves point 2.

It follows that PATH-COVER mimics the moves of OPT_{nl} from r_{k+1} to r_n, and, therefore,

$$\text{PATH-COVER}(\sigma) \leq \text{OPT}(\sigma) + 2k\Delta \ .$$

□

4 Bin Packing

Presented in this chapter is an algorithm for the online bin packing problem (see Section 4.1 for a formal definition), using 1 bit of advice per request, based on the classic HARMONIC algorithm due to [LL85], and an algorithm, inspired by the APTAS algorithms for offline bin packing problem, that is $(1+\varepsilon)$-competitive, using $O\left(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon}\right)$ bits of advice per request.

For the $(1+\varepsilon)$-competitive algorithm, the advice is defined based on an optimal packing of the request sequence. That is, the offline oracle must solve an NP-hard problem. This is possible in this model as no computational restrictions are placed on the oracle. However, it should be noted that the algorithm presented here creates a packing that is $(1+\varepsilon)$-competitive with respect to some packing S^* which does not necessarily have to be an optimal packing. If the computational power of the oracle were restricted, a $(1+\varepsilon)$-competitive (asymptotic) algorithm could be achieved by defining the advice based on the $(1+\varepsilon')$-approximate packing S^* created via a bin packing APTAS, e.g. the scheme of Fernandez de la Vega and Lueker [FdlVL81], (albeit requiring slightly more bits of advice as ε would have to be adjusted according to ε').

To complement the upper bound, we present a lower bound of $\Omega(\log N)$ bits of advice per request on the amount of advice required to be optimal, where N is the optimal number of bins.

4.1 Preliminaries

The *online bin packing problem* consists of a request sequence σ, and an initially empty set B of bins of capacity 1. Each $r_i \in \sigma$ is an item with size $0 < s(r_i) \leq 1$. The goal is to assign all the items of σ to bins such that, for each bin $b_j \in B$, $\sum_{r_i \in b_j} s(r_i) \leq 1$ and $|B|$ is minimized. N is used to denoted the optimal number of bins ($N = |B^{OPT}|$).

An assignment of the items of σ to the bins in a set of bins is called a *packing*. The cardinality of packing B, denoted by $|B|$, indicates the number of bins in B in the packing. A bin is *valid* if the total size of the packed items in the bin is at most 1. We say that an item r_i is *packed* in a bin b if r_i is assigned to the bin b, or that r_i is *packed* in a packing B, according to a some rule that will assign r_i to a bin in B. A bin b can *accommodate* an item r_i if the bin remains valid after packing r_i. In that case, we can say that r_i *fits* into b. A packing is valid if all the bins in a packing are valid. For an item $r_i \in b$, where b is a bin in the packing B, we will write $r_i \in B$.

In our algorithms, we use two common heuristics for bin packing: FIRST FIT (FF) and NEXT FIT (NF) (cf. [Joh73]). Both algorithms are online algorithms which consider each item once and pack it irrevocably.

Definition 4.1 (FIRST FIT)**.** *To pack σ in a* FIRST FIT *manner, $r_i \in \sigma$ is packed in the first bin, in an ordering of the bins, that can accommodate r_i. If no such bin exists, a new bin is opened to pack r_i.*

Definition 4.2 (NEXT FIT)**.** *To pack σ in a* NEXT FIT *manner, a sequence $B = \langle b_1, \ldots, b_q \rangle$, $q \geq 1$, of ordered open bins is used.An item r_i is packed in the first bin in B that can accommodate r_i. If no such bin exists, a new bin is opened to pack r_i and appended to B. Let $b_j \in B$ be the j-th bin in the ordering and be the bin used to pack r_i. All $b_k \in B$ such that $k < j$ are closed, i.e. removed from the sequence B. That is, the bins that cannot accommodate r_i are closed and no subsequent items will be packed in them.*

In the following, for simplicity of presentation, we assume that $1/\varepsilon$ is a natural number.

4.2 1.5-Competitive Online Algorithm with 1 bit of Advice per Request

The algorithm with advice, called HARMONIC WITH ADVICE, presented in this section is based on the classic bin packing algorithm called HARMONIC [LL85] and uses 1 bit of advice per request. It has a competitive ratio of 1.5 which is below the randomized lower bound for the online bin packing [Cha92].

The HARMONIC algorithm [LL85] classifies the requested items, where an item with size in the range $(1/(i+1), 1/i]$ is said to be of type i for $i = 1, \ldots, k-1$. The items with size in the range $(0, 1/k]$ are of type k (k to be determined later). Items with a type 2 or greater will be called *small items* and type 1 items will be called *large items.* Note that exactly i, $1 \leq i < k$, items of type i can fit into a single bin. Bins with i items of type i for $1 \leq i < k$ are called *complete bins.* The HARMONIC algorithm packs the items based on item type as follows.

- Items of type i, $1 \leq i < k$, are packed into bins dedicated to packing type i items in a first fit manner.
- Items of type k are packed into completed bins in a first fit manner. If a type k item does not fit into any of the completed bins, it is packed into a bin dedicated to type k items in a first fit manner.

For an item r_i, the *expansion* of r_i is $f(s(r_i))/s(r_i)$, where $f : (0,1] \rightarrow \mathbb{R}^+$ such that $f(s(x)) \geq s(x)$ is a weight function. The expansion ratio is an upper bound on the contribution of any combination of items of size at most $s(r_i)$ to the competitive ratio. That is, over all possible combinations of items of size at most $s(r_i)$ into a single bin, an algorithm will need at most $\lceil f(s(r_i))/s(r_i) \rceil$ bins to pack the items in the worst case.

The worst case analysis for HARMONIC can be done with a such a weighting function. For $1 \leq i < k$, the weight for type i is $1/i$ and an upper bound on the expansion is the ratio of the weight to the largest size for an item of type $i+1$ (see Table 4.1).

The worst case analysis for this algorithm, using the weighting function for HARMONIC, shows that the highest possible weight of any bin results from a bin with items of size $1/2 + \varepsilon, 1/3 + \varepsilon, 1/7 + \varepsilon, 1/43 + \varepsilon, \ldots$. The competitive ratio is at most $1 + 1/2 + 1/6 + 1/42 + \cdots = 1.691\ldots$ [LL85].

Type	Size	Weight	Expansion
1	$(1/2, 1]$	1	2
2	$(1/3, 1/2]$	1/2	3/2
3	$(1/4, 1/3]$	1/3	4/3
4	$(1/5, 1/4]$	1/4	5/4
5	$(1/6, 1/5]$	1/5	6/5
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k-1$	$(1/k, 1/(k-1)]$	$1/(k-1)$	$k/(k-1)$
k	$(0, 1/k]$	$1/k$	$k/(k-1)$

Table 4.1: The size, weight and expansion for different types, according to the classification used by the HARMONIC algorithm.

We are now ready to define the algorithm HARMONIC WITH ADVICE. It is based on the HARMONIC algorithm and is enhanced with 1 bit of advice per request. It will pack the requested items into different sets of bins based on the item type as defined for HARMONIC and the bit of advice. Let B^{ALG} be the set of all the bins opened by the algorithm. This set is split into two disjoint sets H^{ALG} and G^{ALG}. The set H^{ALG} is the set of the bins that are packed exactly as the HARMONIC algorithm without advice. The set $G^{\text{ALG}} := B^{\text{ALG}} \setminus H^{\text{ALG}}$ is split into three disjoint sets based on the types of items packed in the bins. The three sets are:

- G_1^{ALG}, the set of bins with only a type 1 item;
- $G_{3,4,5}^{\text{ALG}}$, the set of bins with a type 3, 4, or 5 item (and possibly a type 1 item); and
- G_{6+}^{ALG}, the set of bins with one or more type 6 or greater items (and possibly a type 1 item).

Given a $k > 2$, let β_i be the bit of advice received with request r_i and let t_i be the type of r_i, HARMONIC WITH ADVICE will pack r_i in the following manner. If $\beta_i = 0$, r_i is packed in H^{ALG}, according to the HARMONIC algorithm. Otherwise $\beta_i = 1$ and r_i is packed in the following manner. (Note that β_i is never set to 1 for items of type 2 as shown below.)

- If $t_i = 1$, pack r_i in a bin in $G^{\text{ALG}}_{3,4,5} \cup G^{\text{ALG}}_{6+}$ without a type 1. If no such bin exists, pack r_i in a new bin and add it to G^{ALG}_1.
- If $3 \leq t_i \leq 5$, pack r_i in a bin $b \in G^{\text{ALG}}_1$ if $|G^{\text{ALG}}_1| > 0$. Set $G^{\text{ALG}}_1 := G^{\text{ALG}}_1 \setminus b$ and $G^{\text{ALG}}_{3,4,5} := G^{\text{ALG}}_{3,4,5} \cup b$. If $|G^{\text{ALG}}_1| = 0$, pack r_i in a new bin and add it to $G^{\text{ALG}}_{3,4,5}$.
- If $t_i \geq 6$, pack r_i, in a first fit manner, into the set of bins G^{ALG}_{6+} such that the total size of the items with type 6 or greater does not exceed 1/3 after packing r_i. If r_i does not fit into G^{ALG}_{6+}, pack r_i in a bin $b \in G^{\text{ALG}}_1$ if $|G^{\text{ALG}}_1| > 0$. Set $G^{\text{ALG}}_1 := G^{\text{ALG}}_1 \setminus b$ and $G^{\text{ALG}}_{6+} := G^{\text{ALG}}_{6+} \cup b$. If $|G^{\text{ALG}}_1| = 0$, pack r_i in a new bin and add it to G^{ALG}_{6+}.

Let $B^{\text{OPT}}_{>1.5} \subseteq B^{\text{OPT}}$ be the bins in an optimal packing with a weight greater than 1.5 (see Table 4.1). Let $G^{\text{OPT}}_2 \subseteq B^{\text{OPT}}_{>1.5}$ be the bins with a type 2 item, let $G^{\text{OPT}}_{3,4,5} \subseteq B^{\text{OPT}}_{>1.5}$ the bins with a type 3,4 or 5 item and let G^{OPT}_{6+} be the bins with a type 1 item and the rest of the items are of type 6 or greater. Note that the sets G^{OPT}_2, $G^{\text{OPT}}_{3,4,5}$ and G^{OPT}_{6+} are disjoint. Let $\sigma_{2,6+}$ be a subsequence of σ restricted to the items of type 6 or greater from the bins of $G^{\text{OPT}}_2 \cup G^{\text{OPT}}_{6+}$, and let $G^{\text{FF}}_{2,6+}$ be the first fit packing of the items of $\sigma_{2,6+}$ into $|G^{\text{OPT}}_2| + |G^{\text{OPT}}_{6+}|$ bins of capacity 1/3. Given a B^{OPT}, the advice β_i for r_i is defined in the following manner.

$$\beta_i\text{:}\quad \begin{cases} 1, & \text{if } (r_i \in G^{\text{FF}}_{2,6+}) \\ & \text{OR } (r_i \in G^{\text{OPT}}_2 \cup G^{\text{OPT}}_{3,4,5} \cup G^{\text{OPT}}_{6+} \text{ and } t_i = 1) \\ & \text{OR } (r_i \in G^{\text{OPT}}_{3,4,5} \text{ and } r_i \text{ is the second largest item in} \\ & \text{the bin}) \\ 0, & \text{otherwise.} \end{cases}$$

Note that all the r_i from the bins in B^{OPT} with a weight at most 1.5 will have an advice bit of 0. To have an advice bit of 1, r_i must be in the set $G^{\text{OPT}}_2 \cup G^{\text{OPT}}_{3,4,5} \cup G^{\text{OPT}}_{6+}$ which, by definition, is a subset of $B^{\text{OPT}}_{>1.5}$.

With the following lemma that upper bounds the size of type 1 items in bins with a weight greater than 1.5, we are ready to prove the main theorem of this section.

Lemma 4.3. *A bin with a weight greater than 1.5 must contain a type 1 item of size less than* 2/3.

Proof. From the expansion of 3/2 for type 2 items, a bin with a weight greater than 1.5 must contain a type 1 item.

Let b be a bin with a type 1 item and a weight greater than 1.5. Assume that the type 1 item has a size of at least 2/3. The second largest item is at most 1/3 in size and is of type 3 or greater. Type 3 items have an expansion of 4/3. Therefore, an upper bound on the weight of b is $1 + \frac{1}{3} \cdot \frac{4}{3} < 1.5$ which contradicts the assumption that the weight of b is greater than 1.5. □

Theorem 4.4. HARMONIC WITH ADVICE *is* 1.5*-competitive.*

Proof. Let $B^{\text{OPT}}_{\leq 1.5} \subseteq B^{\text{OPT}}$ be the bins with a weight of at most 1.5. HARMONIC WITH ADVICE packs the items of $B^{\text{OPT}}_{\leq 1.5}$, using HARMONIC, and uses at most $1.5|B^{\text{OPT}}_{\leq 1.5}|$ bins.

By Lemma 4.3, we know that every bin with a weight greater than 1.5 has a type 1 item of size at most 2/3. This implies that each bin in G^{ALG} will have a type 1 item.

HARMONIC WITH ADVICE packs the second largest item of $G^{\text{OPT}}_{3,4,5}$ with a type 1 item, using exactly $|G^{\text{OPT}}_{3,4,5}|$ bins. The remaining items with a total size of at most $\frac{|G^{\text{OPT}}_{3,4,5}|}{3}$ are packed according to the HARMONIC algorithm. The largest of the remaining items has a type of 3 and an expansion of 4/3. Therefore, at most $\frac{4|G^{\text{OPT}}_{3,4,5}|}{9}$ bins are needed to pack the remaining items.

HARMONIC WITH ADVICE packs the type 2 items of G^{OPT}_{2} into dedicated bins, using at most $|G^{\text{OPT}}_{2}|/2$ bins. The remaining space which is at most 1/6 in size contains type 6 and greater items.

The type 6 and greater items from $G^{\text{OPT}}_{2} \cup G^{\text{OPT}}_{6+}$ are packed into $|G^{\text{OPT}}_{2} \cup G^{\text{OPT}}_{6+}|$ bins, using up to 1/3 of the bin. The total size of these items is at most $\frac{|G^{\text{OPT}}_{2}|}{6} + \frac{|G^{\text{OPT}}_{6+}|}{2}$. The items are packed into $|G^{\text{ALG}}_{2} \cup G^{\text{ALG}}_{6+}| = |G^{\text{OPT}}_{2} \cup G^{\text{OPT}}_{6+}| = |G^{\text{OPT}}_{2}| + |G^{\text{OPT}}_{6+}|$ bins, using at least $\frac{|G^{\text{OPT}}_{2}|+|G^{\text{OPT}}_{6+}|}{3} - \frac{|G^{\text{OPT}}_{2}|+|G^{\text{OPT}}_{6+}|}{6} = \frac{|G^{\text{OPT}}_{2}|+|G^{\text{OPT}}_{6+}|}{6}$ space. The expansion of the items is 7/6. Therefore, the extra bins used are at most $\left(\frac{|G^{\text{OPT}}_{2}|}{6} + \frac{|G^{\text{OPT}}_{6+}|}{2} - \frac{|G^{\text{OPT}}_{2}|+|G^{\text{OPT}}_{6+}|}{6}\right)\frac{7}{6} = \frac{7|G^{\text{OPT}}_{6+}|}{18}$.

Overall, to pack the items of σ, the HARMONIC WITH ADVICE algorithm uses at most

$$\frac{3|B^{\text{OPT}}_{\leq 1.5}|}{2} + \frac{3|G^{\text{OPT}}_{2}|}{2} + \frac{13|G^{\text{OPT}}_{3,4,5}|}{9} + \frac{25|G^{\text{OPT}}_{6+}|}{18} \leq \frac{3|B^{\text{OPT}}|}{2} \text{ bins.}$$

□

4.3 $(1 + \varepsilon)$-Competitive Online Algorithms with Advice for Bin Packing

In this section, we present an online bin packing algorithm with advice called BPA. The main result of this section is the following.

Theorem 4.5. *Given ε, $0 < \varepsilon \leq 1/2$, the competitive ratio for* BPA *is at most* $1 + 3\varepsilon$ *and uses at most* $\frac{1}{\varepsilon}\log\left(\frac{2}{\varepsilon^2}\right) + \log\left(\frac{2}{\varepsilon^2}\right) + 3$ *bits of advice per request.*

Initially, we will present an algorithm, ABPA that uses less than $\frac{1}{\varepsilon}\log\left(\frac{2}{\varepsilon^2}\right)+\log\left(\frac{2}{\varepsilon^2}\right)+3$ bits of advice per request and is asymptotically (in the number of optimal bins) $(1+2\varepsilon)$-competitive, i.e. non-strictly $(1+2\varepsilon)$-competitive. Then, with a small modification to ABPA, we will present BPA, an algorithm that is $(1+3\varepsilon)$-competitive for any number of optimal bins, i.e. strictly $(1+3\varepsilon)$-competitive, and uses 1 more bit of advice per request than ABPA.

4.3.1 $(1+2\varepsilon)$-Competitive Algorithm

We begin by creating a rounded input σ' based on σ, using a scheme of Fernandez de la Vega and Lueker [FdlVL81]. The scheme works by grouping items based on their size into a finite number of groups and rounding the size of all the items of each group up to the size of the largest item in the group (see Figure 4.1).

More formally, given an ε, an item is called *large* if it has size more than ε. Items with size at most ε are called *small* items. Let L be the number of large items in σ. Sort the large items of σ in order of nonincreasing size. Let $h = \varepsilon^2 L$. For $i = 0, \ldots, 1/\varepsilon^2 - 1$, assign the items $ih+1, \ldots, ih + \lceil \varepsilon^2 L \rceil$ to group $i+1$. A large item of *type* i denotes a large item assigned to group i. The last group may contain less than $\lceil \varepsilon^2 L \rceil$ items. For each item in group i, $i = 1, \ldots, 1/\varepsilon^2$, round up its size to the size of the largest element in the group.

Let σ' be the subsequence of σ restricted to the large items with their sizes rounded up as per the scheme of Fernandez de la Vega and Lueker. We now build a packing S' for σ based on a packing of σ'. The type 1 items will be packed one item per bin in the set of bins B_1.

For the remaining *large* items, i.e. types 2 to $1/\varepsilon^2$, in σ', a packing, B_2' that uses at most N bins can be found efficiently [FdlVL81]. The packing of each bin $b_i \in B_2'$ can be described by a vector of length at most $1/\varepsilon$ (the maximum number of large items that can fit in a single bin), denoted $\mathbf{p_i}$, where each value in the vector ranges from 1 to $1/\varepsilon^2$ representing the type of each of the large items in b_i. This vector will be called a *bin pattern*. Let B_2 be a set of bins such that $|B_2| = |B_2'|$ and each $b_i \in B_2$ is assigned the bin pattern $b_i \in B_2'$. The items of σ' can be assigned sequentially to the bins of B_2,

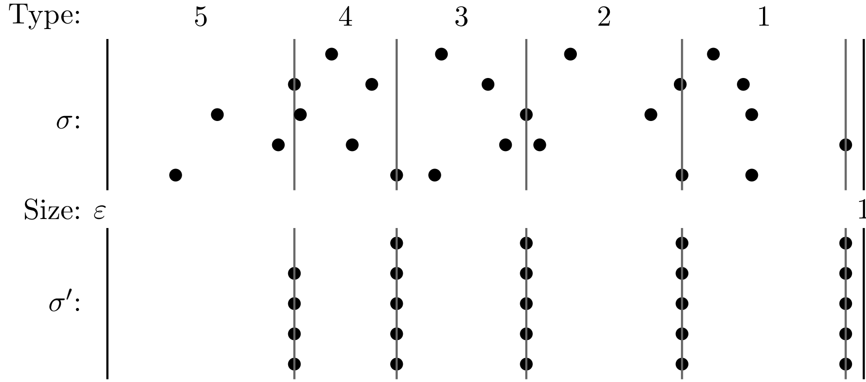

Figure 4.1: An example of grouping and rounding of large items for $\varepsilon = \sqrt{1/5}$. The top illustration show the size of the 24 large items, denoted by the black dots, from σ grouped into 5 groups of 5 items (except for the last group that contains 4 items), according to a sorting of the items by size. The bottom illustration denotes the same items in the same grouping with their sizes rounded up as in σ'.

using the following procedure. Initially, the bins of B_2 are all closed. For each $r_i \in \sigma'$, assign r_i with type t_i to the oldest open bin, b_j, such that there are less items of type t_i packed in the bin than are described in $\mathbf{p_j}$. If no such bin exists, open a closed bin with a pattern that contains the type t_i and pack r_i in this bin. Note that such a bin must exist by the definition of B_2.

The packing S' is defined to be $B_1 \cup B_2$ with the original (non-rounded up) sizes of the packed large items. By definition, $|B_1| = \lceil \varepsilon^2 L \rceil$. Since large items have size at least ε, $N \geq \varepsilon L$. This implies the following fact.

Fact 4.6. $|B_1| \leq \varepsilon N$

The bins of S' are numbered from 1 to $|S'|$ based on the order that the bins would be opened when σ' is processed sequentially. That is, for $i < j$ and every $b_i, b_j \in S'$, there exists an $r_p \in b_i$ such that, for all $r_q \in b_j$, $p < q$. From Fact 4.6 and that $|B_2| \leq N$ by definition, we have the following fact.

Fact 4.7. $|S'| \leq (1 + \varepsilon)N$

We now extend S' to include the small items and define S. Sequentially, by the order the small items arrive, for each small item $r_i \in \sigma$, pack r_i into S', using NEXT FIT. Additional bins are opened as necessary. The following lemma shows that S is

a near-optimal packing. Note that this bound implies that S may pack one more bin than $(1+2\varepsilon)$ times the optimal, giving it a non-strict competitive ratio of $1+2\varepsilon$.

Lemma 4.8. $|S| \leq (1+2\varepsilon)N + 1$

Proof. After packing the small items, if no new bins are opened then the claim follows from Fact 4.7. If there are additional bins opened, all the bins of S, except possible the last one, are filled to at least $(1-\varepsilon)$. Since the total size of the items is at most N, we have $(|S|-1)(1-\varepsilon) \leq N$ and, therefore, $|S| \leq \frac{N}{1-\varepsilon} + 1 \leq (1+2\varepsilon)N + 1$. □

We now define ABPA. It is defined given a σ, and an ε, $0 < \varepsilon \leq 1/2$. ABPA uses two (initially empty) sets of bins L_1 and L_2. L_1 is the set of bins that pack small items and 0 or more large items. L_2 is the set of bins that pack only large items. BPA and the advice will be defined such that the items are packed exactly as S.

With the first N items, the advice bits indicate a bin pattern. These N bin patterns will be the patterns of the bins in order from S. As the bin patterns are received, they will be queued. Also, with each item, the advice bits indicate the type of the item. Small items will be of type -1. If the item is large, the bits of advice will also indicate if it is packed in S in a bin that also includes small items or not.

During the run of BPA, bins will be opened and assigned bin patterns. The bins in each of the sets of bins are ordered according to the order in which they are opened. When a new bin is opened, it is assigned an empty bin pattern if the current item is small. If the current item is type 1, the bin is assigned a type 1 bin pattern. Otherwise, the current item is of type 2 to $1/\varepsilon^2$, and the next pattern from the queue of bin patterns is assigned to the bin. Note that, by the definition of S, this pattern must contain an entry for an item of the current type.

For each $r_i \in \sigma$, the items are packed and bins are opened as follows:

Small Items. For packing the small items, BPA maintains a pointer into the set L_1 indicating the bin into which it is currently packing small items. Additionally, the advice for the small items includes a bit (the details of this bit will be explained subsequently) to indicate if this pointer should be moved to the next bin in L_1. If this is the case, the pointer is moved prior to packing the small item and, if there is no next bin in L_1, a new bin with an empty pattern is opened and added to L_1. Then, the small item is packed into the bin referenced by the pointer.

Large Items. BPA receives an additional bit y as advice that indicates if r_i is packed in a bin in S that also includes small items.

Type 1 *items:* If the item r_i is packed into a bin with small items ($y = 1$), r_i is packed in the oldest bin with an empty pattern. If no such bin exists, then r_i is packed into a new bin that is added to L_1. If r_i is packed into a bin without small items ($y = 0$), then r_i is packed into a new bin that is added to L_2. In all the cases, the bin into which r_i is packed is assigned a type 1 bin pattern.

Type $i > 1$ *items:* Let t_i be the type of r_i. If r_i is packed with small items ($y = 1$), then r_i is packed into the oldest bin of L_1 such that the bin pattern specifies more items of type t_i than are currently packed. If no such bin exists, then r_i is packed in the first bin with an empty bin pattern and the next bin pattern from the queue is assigned to this bin. If there are no bins with empty bin patterns, a new bin is added to pack r_i. If r_i is not packed with small items ($y = 0$), r_i is packed analogously but into the bins of L_2.

The advice bit used to move the pointer for packing small items (see Section 4.3.1.1 for a formal definition) is defined so that BPA will pack the same number of small items in each bin as S. Further, BPA packs both the small and large items in the order each item arrives in the least recently opened bin just as S which implies the following fact.

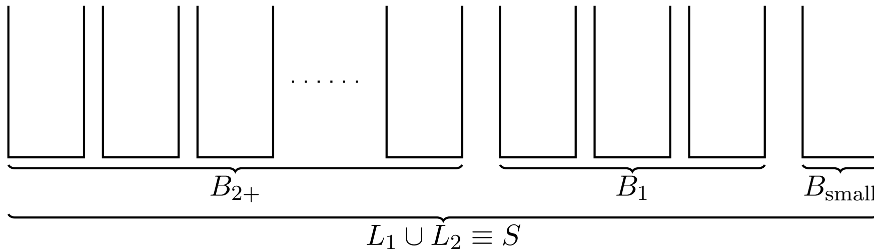

Figure 4.2: An illustration of the packing produced by ABPA, $L_1 \cup L_2$, that is equivalent to the packing S. B_{2+} packs items of type 2 to $1/\varepsilon^2$ into N bins. B_1 represents the set of εN bins dedicated to packing type 1 items and B_{small} represents the (possibly empty) set of at most $\varepsilon N + 1$ bins dedicated to packing the overflow of small items from the NEXT FIT packing of the small items into the bins of $B_1 \cup B_{2+}$.

Fact 4.9. *$L_1 \cup L_2$ is the same packing as S.*

Therefore, $|L_1 \cup L_2| \leq (1 + 2\varepsilon)N + 1$ by Lemma 4.8.

4.3.1.1 Formal Advice Definition

Item Patterns. Instead of sending the entire vector representing a bin pattern, we enumerate all the possible vectors and the advice will be the index of the vector from the enumeration encoded in binary. The bin pattern vectors have a length of at most $1/\varepsilon$ and there are at most $1/\varepsilon^2$ different possible values. For the enumeration, item pattern vectors with length less than $1/\varepsilon$ items will be padded to a length $1/\varepsilon$ with addition entries with a new value of $\perp$.

The algorithm requires less than $\lceil \frac{1}{\varepsilon} \log\left(\frac{1}{\varepsilon^2}+1\right) \rceil < \frac{1}{\varepsilon}\log\left(\frac{2}{\varepsilon^2}\right)+1$ bits of advice per request to encode the index of the item pattern from an enumeration of all possible item patterns in binary.

Advice per Request. In order to define the advice, for each bin $b_i \in S$, we define a value κ_i that is the number of small items packed in b_i.

Per request, the advice string will be xyz, where x is $\lceil \log\left(1/\varepsilon^2+1\right) \rceil < \log\left(2/\varepsilon^2\right)+1$ bits in length to indicate the type of the item; y is 1 bit in length to indicate whether the large items are packed with small items, or to indicate to small items whether or not to move to a new bin; z has a length less than $\frac{1}{\varepsilon}\log\left(\frac{2}{\varepsilon^2}\right)+1$ to indicate a bin pattern. xyz is defined as follows for request r_i:

x:		The type of r_i encoded in binary.
y:	r_i is a small item:	Let s be the number of small items in $\langle r_1, \ldots, r_{i-1}\rangle$. If there exists and an integer $1 \leq j \leq N$ such that $\sum_{k=1}^{j} \kappa_k = s$, then the first bit is a 1. Otherwise, the first bit is a 0.
	r_i is a large item:	1, if $\kappa_i > 0$, where b_i is the bin in which r_i is packed in S, i.e. b_i packs small items. Otherwise, 0.
z:	$i \leq N$	The bits of z is a number in binary indicating the vector representing the bin pattern of the i-th bin opened by S'.
	$i > N$	Not used. All zeros.

4.3.2 Strict $(1+3\varepsilon)$-Competitive Algorithm

BPA is defined such that it will behave in two different manners, depending on N, the number of bins in an optimal packing, and ε. One bit of advice per request, denoted by w, is used to distinguish between the two cases. The two cases are as follows.

Case 1: $N > 1/\varepsilon$ **(**$w = 0$**).** BPA will run ABPA as described previously. The only difference is that the advice per request for ABPA is prepended with an additional bit for w. Since $N > 1/\varepsilon$, a single bin is at most εN bins. Therefore, we get the following corollary to Lemma 4.8.

Corollary 4.10. $|S| \leq (1 + 3\varepsilon)N$

Case 2: $N \leq 1/\varepsilon$ **(**$w = 1$**).** In this case, for each $r_i \in \sigma$, after w, the next $\lceil \log(1/\varepsilon) \rceil$ bits of advice per request define the bin number in which r_i is packed in an optimal packing. BPA will pack r_i into the bin as specified by the advice. Including w, this case requires less than $\log(1/\varepsilon) + 2 < \frac{1}{\varepsilon} \log\left(\frac{2}{\varepsilon^2}\right) + \log\left(\frac{2}{\varepsilon^2}\right) + 3$ (the upper bound on the amount of advice used per request in case 1) bits of advice per request and the packing produced is optimal.

The definition of the algorithm and the advice, Fact 4.9 and Corollary 4.10 prove Theorem 4.5.

4.4 Lower Bound on the Advice Required for Optimality

For the lower bound, we assume that the algorithm knows N, the optimal number of bins.

Let

$$\begin{aligned} k &= n - 2N, \\ \sigma_1 &= \left\langle \frac{1}{2^{k+2}}, \frac{1}{2^{k+3}}, \ldots, \frac{1}{2^{k+N+1}}, \frac{1}{2^2}, \ldots, \frac{1}{2^{k+1}} \right\rangle \text{ and} \\ \sigma_2 &= \langle x_1, x_2, \ldots, x_N \rangle \ , \end{aligned}$$

where x_i will be defined later in an adversarial manner. The entire adversarial request sequence will be $\sigma = \langle \sigma_1, \sigma_2 \rangle$. The adversarial sequence will be chosen such that the adversary will pack the items in to the N bins such that each will be filled to capacity while any algorithm using less than $k \log N$ bits of advice will be force to pack the items into $N + 1$ bins.

Fact 4.11. *Every subset of the requests of* σ_1 *has a unique sum that is less than* $1/2$*.*

Let T be the set of all possible packings in N bins of the requests of σ_1. The adversary will pack each of the first N requests of σ_1 in a distinct bin. This distinguishes the N bins from one another. Let V be the set of all possible packings of the last k requests of σ_1 into the N distinct bins. Note that $V \subset T$ and that $|V| = N^k$. Let $P_{\sigma_1}^{\text{ADV}} \in V$ be the adversarial packing of the items of σ_1. $P_{\sigma_1}^{\text{ADV}}$ will be determined later. Let $x_i = 1 - \sum_{r_j \in b_i} s(r_j)$ for $b_i \in P_{\sigma_1}^{\text{ADV}}$. Note that, by using Fact 4.11, we have that the N values x_i, $1 \leq i \leq N$, are distinct.

Observation 4.12. *For every $P_{\sigma_1} \in V \setminus P_{\sigma_1}^{\text{ADV}}$, every possible packing of the items of σ_2 into P_{σ_1} results in packing of at least $N+1$ bins.*

Proof. The sum of the size of the items of σ_1 is less than 1/2 which implies that the size for each x_i is greater than 1/2. Therefore, any bin that packs more than one item from σ_2 is not valid. We, therefore, consider a packing P_σ that packs at most a single item from σ_2 per bin.

Since the total size of all the items of σ is N, if there is a $b_j \in P_\sigma$ such that $\sum_{r_i \in b_j} s(r_i) < 1$, then $|P_\sigma| \geq N+1$. We can assume by way of contradiction that in P_σ all bins are filled to capacity. Then, Fact 4.11 implies that P_{σ_1} equals $P_{\sigma_1}^{\text{ADV}}$, a contradiction. □

We are now ready to prove the main theorem of the section.

Theorem 4.13. *Any online algorithm with advice needs at least $(n - 2N)\log N$ bits of advice in order to be optimal for the bin packing problem, where N is the optimal number of bins and n is the length of the request sequence.*

Proof. Let ALG be an arbitrary deterministic online algorithm with advice for the bin packing problem. Let $P_{\sigma_1}^{\text{ALG}}$ be the packing produced by ALG for σ_1. If $P_{\sigma_1}^{\text{ALG}} \in T \setminus V$, i.e. $P_{\sigma_1}^{\text{ALG}}$ is such that the first N requests are not packed in a different bin, then, by Observation 4.12, $P_{\sigma}^{\text{ALG}} \geq N+1$. Therefore, we will assume that the algorithm will pack the first N requests in N distinct bins, i.e. $P_{\sigma_1}^{\text{ALG}} \in V$.

Assume that the online algorithm with advice receives all the advice bits in advance. This only strengthens the algorithm and, thus, strengthens our lower bound. Let ALG(s,u) be the packing produced by ALG for request sequence s when receiving advice bits u. Let U be the advice space of ALG. Since ALG gets less than $k \log N$ bits of advice, $|U| < N^k$. It follows that $|\{\text{ALG}(\sigma_1, u) | u \in U\}| < N^k = |V|$. Therefore, given ALG, $P_{\sigma_1}^{\text{ADV}}$ is chosen by the adversary such that $P_{\sigma_1}^{\text{ADV}} \in V \setminus \{\text{ALG}(\sigma_1, u) | u \in U\}$. Note that this choice defines σ_2.

We now have, by Observation 4.12, that P_{σ}^{ALG} uses at least $N+1$ bins. □

5 Dual Bin Packing

In this chapter, we consider the dual bin packing problem (see Section 5.1 for a formal definition) and present an algorithm that uses 1 bit of advice per request that is 1.5-competitive and an algorithm, inspired by the PTAS algorithms for the offline dual bin packing problem, that is $(1/(1-\varepsilon))$-competitive and uses $O(1/\varepsilon)$ bits of advice per request.

The approach of rounding and grouping items used in this chapter for the $(1/(1-\varepsilon))$-competitive algorithm is similar to the approach of the $(1+\varepsilon)$-competitive bin packing algorithm in Chapter 4. However, in the algorithm for the dual bin packing problem, bins containing more than $1/\varepsilon$ items are handled differently than bins with less items. This allows for a small savings in the advice used in that we do not have to worry about small items. The main savings comes from the fact that, for the bins with at most $1/\varepsilon$ items, we only need to split them into $1/\varepsilon$ groups as opposed to the $1/\varepsilon^2$ groups for bin packing. This allows for a savings in the amount of advice used per request by a factor of $\log\left(\frac{1}{\varepsilon}\right)$ (up to constant factors). As with the $(1+\varepsilon)$-competitive bin packing algorithm in Chapter 4, the advice is based on an optimal packing and could be replaced with a packing produced by a PTAS for the dual bin packing problem.

The lower bound on the amount of advice required to be optimal of $\Omega(\log N)$ of Theorem 4.13 from Section 4.4 for the bin packing problem holds for the dual bin problem. That is, the lower bound construction for the dual bin packing problem uses the same sets of items as for the bin packing problem lower bound and N bins. The optimal packing for the dual bin packing problem would pack all the items. A bin packing solution which requires $N+1$ bins is equivalent to rejecting at least 1 item.

5.1 Preliminaries

The *dual bin packing problem* consists of a sequence σ of items and a set B of n bins (initially empty and numbered from 1 to n) with capacity 1. Each $r_i \in \sigma$ is an item with size $0 < s(r_i) \leq 1$ and $|\sigma| = m$. The goal is to maximize the number of the items of σ assigned to the n bins such that, for all bins $b_i \in B$, $\sum_{r_i \in b_i} s(r_i) \leq 1$. That is, to pack as many items as possible in the n bins. Items that are not packed are called *rejected items*.

The notions of a packing, valid packings, valid bins, item being packed into bins, and such are defined for the dual bin packing problem as they are for the bin packing problem in Section 4.1.

For an algorithm ALG and a σ, the set of rejected items is denoted by R_σ^{ALG}, and the set of accepted items by A_σ^{ALG}. Let $p_\sigma^{\text{ALG}} = \lfloor s(r_j)^{-1} \rfloor$, where $r_j \in R_\sigma^{\text{ALG}}$ is the smallest item rejected by ALG. Let P_σ^{ALG} be a packing produced by ALG for the items of σ. For a given P_σ^{ALG}, let $k_{\sigma,i}^{\text{ALG}}$ be the number of items in the i-th bin, let $B_{\sigma,i}^{\text{ALG}}$ be the set of items packed in the i-th bin, and let $v_{\sigma,i}^{\text{ALG}}$ be the total size of the items packed in $B_{\sigma,i}^{\text{ALG}}$. For two algorithms, ALG$_1$ and ALG$_2$, such that $R_\sigma^{\text{ALG}_2} \subseteq R_\sigma^{\text{ALG}_1}$, the set $D_\sigma^{\text{ALG}_1,\text{ALG}_2}$ is defined to be $R_\sigma^{\text{ALG}_1} \setminus R_\sigma^{\text{ALG}_2}$. When it is clear by the context, the superscripts and subscripts will be suppressed.

A definition of the FIRST FIT (FF) heuristic used in this chapter can be found in Section 4.1.

5.2 1.5-Competitive Online Algorithm with 1 bit of Advice per Request

In this section, we present an online algorithm with advice called SUBSEQUENCE FIRST FIT (SFF) that uses 1 bit of advice. First, we consider an offline algorithm called RESTRICTED SUBSEQUENCE FIRST FIT (RSFF) which we show to be 3/2-competitive. This bound holds for SFF as we show that SFF will always pack at least as many items as RSFF.

RESTRICTED SUBSEQUENCE FIRST FIT. Initially, define $\sigma^{\text{RSFF}} = \sigma$. Then, simulate the FF packing of σ^{RSFF}. If $|R^{\text{FF}}(\sigma^{\text{RSFF}})| > 0$, remove the largest item from σ^{RSFF} (for

ties remove the item with the smallest index). Repeat the process until all the items of σ^{RSFF} can be packed in the n bins, using FF. This is the packing produced by RSFF.

The key properties of RSFF are that the items are packed using FF and that the largest items have been rejected, i.e. the size of all the rejected items is at least as large as any packed item. The following lemma shows that the minimum number of items in every bin for a packing produced by RSFF is at least $p^{\text{RSFF}} = \lfloor s(r_j)^{-1} \rfloor$, where r_j is the smallest rejected item.

Lemma 5.1. *For any σ, $k_i^{\text{RSFF}} \geq p^{\text{RSFF}}$ for $1 \leq i \leq n$.*

Proof. Suppose that $k_i^{\text{RSFF}} < p := p^{\text{RSFF}}$ for some $1 \leq i \leq n$. Given that the largest items are rejected by RSFF, the packed items in the i-th bin have size less than or equal to $s(r_j)$, where r_j is the smallest item rejected by RSFF. The free space in the i-th bin is greater than or equal to $1 - (p-1)(s(r_j)) \geq s(r_j)$ implying that r_j should have been packed in the i-th bin, which is a contradiction. □

The following lemma shows that there always exists an optimal packing that rejects the largest items.

Lemma 5.2. *For any σ, there exists an optimal packing such that R^{OPT} is the $|R^{\text{OPT}}|$ largest items (for ties reject the item with smallest index).*

Proof. Fix an optimal packing for σ. Let $r_j \in R^{\text{OPT}}$ be the smallest rejected item. Consider the largest $r_i \in A^{\text{OPT}}$. If $s(r_i) > s(r_j)$, replace r_i by r_j in the packing and add r_i to R^{OPT}. If $s(r_i) = s(r_j)$ and $i < j$, replace r_i by r_j in the packing and add r_i to R^{OPT}. Repeat this procedure until $r_j \geq r_i$ for all $r_i \in A^{\text{OPT}}$. The packing resulting from this procedure is still optimal and all the rejected items are at least as large as all the packed items. □

In the following, we will always assume that OPT uses the optimal packing as described by Lemma 5.2. In the next lemma, we use this optimal packing and show that the difference between the number of rejected items for RSFF and an optimal algorithm is less than n.

Lemma 5.3. *For any σ, there exists an* OPT *such that $|D| = |R^{\text{RSFF}} \setminus R^{\text{OPT}}| \leq n - 1$.*

Proof. Fix an optimal packing for σ. By Lemma 5.2 and the definition of RSFF, we can assume that $R^{\text{OPT}} \subseteq R^{\text{RSFF}}$ and R^{OPT} is the $|R^{\text{OPT}}|$ largest items of R^{RSFF} (ties are broken by rejecting the item with the smallest index). Let $s(r_j)$ be the size of

the smallest rejected item in R^{RSFF}. The total size of the items in D is at most the empty space of P^{RSFF} which is less than $s(r_j)n$. Therefore, $s(r_j)(|R^{\text{RSFF}}| - |R^{\text{OPT}}|) \leq \sum_{\eta \in R^{\text{RSFF}} \setminus R^{\text{OPT}}} s(\eta) < s(r_j)n$. Hence, $|R^{\text{RSFF}} \setminus R^{\text{OPT}}| = |R^{\text{RSFF}}| - |R^{\text{OPT}}| < n$. $\square$

We are now ready to prove the upper bound on RSFF.

Theorem 5.4. *For any σ, the approximation ratio of* RSFF *is* $3/2$.

Proof. If there are empty bins, then RSFF has packed all the items and $\frac{\text{OPT}(\sigma)}{\text{RSFF}(\sigma)} = 1$. In the following, we only consider instances with rejected items. Also, we always assume that OPT uses the optimal packing as described by Lemma 5.2.

Let $s(r_j)$ be the size of the smallest rejected item in R^{RSFF} and let $p := p^{\text{RSFF}}$.

Case 1: $p \geq 2$ ($s(r_j) \leq \frac{1}{2}$). In the packing by RSFF, all the bins contain at least p items by Lemma 5.1. Therefore, $|A^{\text{RSFF}}| \geq pn$. Thus, using Lemma 5.3, the approximation ratio is

$$\frac{\text{OPT}(\sigma)}{\text{RSFF}(\sigma)} = \frac{|A^{\text{RSFF}}| + |D|}{|A^{\text{RSFF}}|} < \frac{pn + n}{pn} = \frac{p+1}{p} \leq \frac{3}{2}\ , \text{ for } p \geq 2.$$

Case 2: $p = 1$ ($\frac{1}{2} < s(r_j)$). Let q be the number of bins packed with at least 2 items and let A^L be the set of accepted large items (size $> 1/2$). The first fit packing guarantees that at most one of the $n - q$ bins with a single item contains a small item (size $\leq 1/2$). Therefore, $|A^L| \geq n - q - 1$.

By the definition of OPT (Lemma 5.2), OPT packs the items of D and A^L and the items of $D \cup A^L$ are large. Since at most one large item can be packed per bin and the number of bins is n, we have that $|D| + |A^L| \leq n$. Adding this to $|D| \leq n - 1$ from Lemma 5.3, we have that

$$|D| \leq \frac{2n - 1 - |A^L|}{2} \leq \frac{n+q}{2}\ . \tag{5.1}$$

Using (5.1) and the fact that $|A^{\text{RSFF}}| \geq 2q + (n - q) = n + q$, we get that the approximation ratio is

$$\frac{\text{OPT}(\sigma)}{\text{RSFF}(\sigma)} = \frac{|A^{\text{RSFF}}| + |D|}{|A^{\text{RSFF}}|} \leq \frac{n + q + \frac{n+q}{2}}{n+q} = \frac{3}{2}\ .$$

$\square$

Now, we define an online algorithm with 1 bit of advice per request and show that it always packs at least as many items as RSFF.

Subsequence First Fit with 1 bit of Advice. If the bit of advice is a 1, pack the requested item using FF. Otherwise, the bit is a 0 and the item is rejected. The advice is based on the final subsequence σ^{RSFF} of σ that would be packed by the algorithm RSFF as defined previously. The advice bit for r_i is defined to be 1 if $r_i \in \sigma^{\text{RSFF}}$ and 0 otherwise.

Theorem 5.5. *For any σ, the competitive ratio of* SFF *is* $3/2$.

Proof. The final sequence of items of σ^{RSFF} is defined such that they can be packed using FF into the n bins. This fact and Theorem 5.5, imply that SFF is 3/2-competitive. □

5.3 $(1/(1-\varepsilon))$-Competitive Algorithm with Advice

In this section, an algorithm with advice is presented that has a competitive ratio of $1/(1-2\varepsilon)$, using $O\left(\frac{1}{\varepsilon}\right)$ bits of advice per request for $0 < \varepsilon < 1/2$. This shows that an algorithm can arbitrarily approach a competitive ratio of 1 with advice bits that are inversely proportional to the approximation guarantee. It should be noted that, if the amount of bits of advice per request is $\log(n+1)$, then the trivial algorithm with advice can be used to be optimal. That is, an algorithm that is given the bin number ($n+1$ for a rejected item) in which to pack the item. For ease of presentation, $1/\varepsilon$ is assumed to be a natural number.

Initially, we present an algorithm ADBPA that is $(1/(1-\varepsilon))$-competitive and uses $O\left(\frac{1}{\varepsilon}\right)$ bits of advice per request. Then, with a slight modification to ADBPA and one additional bit of advice, we present an algorithm DBPA that is strictly $(1/(1-2\varepsilon))$-competitive.

5.3.1 $(1/(1-\varepsilon))$-Competitive Algorithm.

The algorithm presented in this section is called ADBPA and has a competitive ratio of $1/(1-\varepsilon)$. It uses $O\left(\frac{1}{\varepsilon}\right)$ bits of advice per request for $0 < \varepsilon < 1$.

The advice for ADBPA is defined such that ADBPA can distinguish between two types of items in an optimal packing. They are: (1) items that belong to bins with more than $1/\varepsilon$ items and (2) items that belong to bins with at most $1/\varepsilon$ items. Another bit of advice will indicate for each item of type (1) whether to pack it in a first fit manner into a subset of the available bins or reject it. The items of type (2) will be classified

into groups based on the size of the item (each group being an ε fraction of the total number of items). The advice will indicate for each item of type (2) the index of the group to which the item belongs and the type of bin (see bin pattern below) in which it should be packed. The only items of type (2) that are rejected are the items from the group containing the largest items. The algorithm and the advice are formally defined in the following.

Let P^{OPT} be an optimal packing for σ. We assume without loss of generality that OPT is the optimum described in Lemma 5.2. We split the bins of P^{OPT} into two sets of bins P_1 and P_2 such that all the bins of P_1 have more than $1/\varepsilon$ items packed in each bin and $P_2 = P^{\text{OPT}} \setminus P_1$.

Let $\sigma^{>\frac{1}{\varepsilon}}$ be the subsequence of items from σ that are packed in the bins of P_1. Further, let σ^{SFF} be the subsequence of items from $\sigma^{>\frac{1}{\varepsilon}}$ that would be packed by SFF (see Section 5.2) if given $|P_1|$ bins to pack $\sigma^{>\frac{1}{\varepsilon}}$.

Let $\sigma^{\leq\frac{1}{\varepsilon}}$ be the subsequence of items from σ that are packed in the bins of P_2. The items of $\sigma^{\leq\frac{1}{\varepsilon}}$ are sorted in non-increasing order of size and, based on that ordering, the items are partitioned into $\lfloor 1/\varepsilon \rfloor - 1$ groups of size $\left\lceil \varepsilon|\sigma^{\leq\frac{1}{\varepsilon}}| \right\rceil$ and, for the remaining smallest items, 1 group of size at most $\left\lceil \varepsilon|\sigma^{\leq\frac{1}{\varepsilon}}| \right\rceil$. The items are assigned a type from 1 to $1/\varepsilon$ based on their group in the sorted order. The bins of P_2 have at most $1/\varepsilon$ items. Therefore, the packing of each bin can be described by a $1/\varepsilon$ length vector where the entries are the types of the packed items. Note that the item types have values in the range $[1, 1/\varepsilon]$. This vector will be called a *bin pattern*.

Items that are rejected by the OPT, i.e. $r_i \notin \sigma^{>\frac{1}{\varepsilon}} \cup \sigma^{\leq\frac{1}{\varepsilon}}$, are also assigned a type of 1. This is a technical detail to ensure that ADBPA rejects all the items rejected by OPT without using an additional bit of advice.

We now define a process that assigns bin patterns to the items. The assigned bin patterns will be received as part of the advice to the algorithm. Let Q be the multiset of the bin patterns of P_2. Each item of type $i > 1$ in $\sigma^{\leq\frac{1}{\varepsilon}}$ is assigned a bin pattern in Q as follows. Let B be a set of $|P_2|$ bins to which bin patterns will be assigned. Initially, the bins of B have no patterns assigned. Consider each item $r_i \in \sigma^{\leq\frac{1}{\varepsilon}}$ with type $t_i > 1$ in the order they arrive as defined by σ. Let $t'_i = t_i - 1$ and pack r_i in a bin b in B such that the number of items packed in b with type t_i is less than the number of items of type t'_i as specified by the pattern of b. If no such bin exists, pack r_i in an empty bin b without an assigned pattern, assign a pattern $p \in Q$ that contains type t'_i to b and set

$Q = Q \setminus p$. Item r_i is assigned the bin pattern of b. Note that the assignment and the packing are feasible since the number of items of type i is at least the number of items of type $i-1$ and all the items of type i are smaller in size than the items of type $i-1$.

We now define the online algorithm with advice. ADBPA maintains two sets of bins L_1 and L_2. As ADBPA processes σ, the n bins will be assigned to either L_1 or L_2 such that, after processing σ, $|L_1| = |P_1|$ and $|L_2| = |P_2|$. Initially, none of the bins are assigned to either set. L_1 is the set of bins that pack the items in $\sigma^{\text{SFF}} \subseteq \sigma^{>\frac{1}{\varepsilon}}$ and L_2 is the set of bins that pack the type $i > 1$ items in $\sigma^{\leq\frac{1}{\varepsilon}}$. As bins are added to L_2, they will be assigned bin patterns. For each r_i in σ, ADBPA gets a bit of advice per request to indicate if $r_i \in \sigma^{>\frac{1}{\varepsilon}}$ or not and packs r_i as follows.

$r_i \in \sigma^{>\frac{1}{\varepsilon}}$: For each $r_i \in \sigma^{>\frac{1}{\varepsilon}}$, an additional bit of advice indicates if $r_i \in \sigma^{\text{SFF}}$. If so, ADBPA packs r_i into the bins of L_1 in a first fit manner. If r_i does not fit, an unassigned bin is assigned to L_1 and r_i is packed in that bin. Otherwise, if the bit indicates that $r_i \notin \sigma^{\text{SFF}}$, then r_i is rejected.

$r_i \notin \sigma^{>\frac{1}{\varepsilon}}$: For each $r_i \notin \sigma^{>\frac{1}{\varepsilon}}$, ADBPA receives the item type, t_i, as advice. If $t_i = 1$, the item is rejected. Otherwise, let $t'_i = t_i - 1$. In this case, ADBPA also receives an assigned bin pattern, p_i (as defined previously) such that t'_i is an entry of p_i, as advice. The algorithm packs r_i in a bin $b_j \in L_2$ with a pattern p_i such that there are less items of type t_i are packed in b_j than there are entries of t'_i in the pattern. If no such bin exists, ADBPA will assign an unassigned bin to L_2, assign it the pattern p_i, and pack r_i in it.

From the definition of ADBPA, we have that the items of σ^{SFF} are packed in $|P_1|$ bins and the items of $\sigma^{\leq\frac{1}{\varepsilon}}$ are packed in $|P_2|$ which implies the following fact.

Fact 5.6. ADBPA *will use n bins to pack the non-rejected items of σ.*

In the next two lemmas, we bound from below the number of items of $\sigma^{>\frac{1}{\varepsilon}}$ and the number of items of $\sigma^{\leq\frac{1}{\varepsilon}}$ that ADBPA packs as compared to OPT. For $x \in \{\sigma^{>\frac{1}{\varepsilon}}, \sigma^{\leq\frac{1}{\varepsilon}}\}$, let ADBPA^x and OPT^x be the number of items packed from σ^x by ADBPA and OPT, respectively. Note that, $\text{ADBPA}(\sigma) = \text{ADBPA}^{\sigma^{>\frac{1}{\varepsilon}}} + \text{ADBPA}^{\sigma^{\leq\frac{1}{\varepsilon}}}$ and $\text{OPT}(\sigma) = \text{OPT}^{\sigma^{>\frac{1}{\varepsilon}}} + \text{OPT}^{\sigma^{\leq\frac{1}{\varepsilon}}} = |\sigma^{>\frac{1}{\varepsilon}}| + |\sigma^{\leq\frac{1}{\varepsilon}}|$.

Lemma 5.7. $\text{ADBPA}^{\sigma^{>\frac{1}{\varepsilon}}} > (1-\varepsilon)\text{OPT}^{\sigma^{>\frac{1}{\varepsilon}}}$.

Proof. By the definition of $\sigma^{>\frac{1}{\varepsilon}}$, OPT packs all the items of $\sigma^{>\frac{1}{\varepsilon}}$ in $|P_1|$ bins. Therefore, for any optimal packing of $\sigma^{>\frac{1}{\varepsilon}}$ in $|P_1|$ bins, the set of rejected items is empty, i.e. $\left|R^{\text{OPT}}_{\sigma^{>\frac{1}{\varepsilon}}}\right| = 0$. For $|P_1|$ bins and request sequence $\sigma^{>\frac{1}{\varepsilon}}$, Lemma 5.3 gives

$$\left|R^{\text{RSFF}}_{\sigma^{>\frac{1}{\varepsilon}}} \setminus R^{\text{OPT}}_{\sigma^{>\frac{1}{\varepsilon}}}\right| = \left|R^{\text{RSFF}}_{\sigma^{>\frac{1}{\varepsilon}}}\right| < |P_1| \; . \tag{5.2}$$

Let $R^{\sigma^{>\frac{1}{\varepsilon}}}$ be the rejected items of $\sigma^{>\frac{1}{\varepsilon}}$ by ADBPA when run on σ. ADBPA packs the items of $\sigma^{>\frac{1}{\varepsilon}}$ into $|P_1|$ bins. This produces the same packing as SFF and, by the definition of SFF, the same packing as RSFF when packing $\sigma^{>\frac{1}{\varepsilon}}$ into $|P_1|$ bins. This implies that $\left|R^{\text{RSFF}}_{\sigma^{>\frac{1}{\varepsilon}}}\right| = |R^{\sigma^{>\frac{1}{\varepsilon}}}|$ when the number of bins used by RSFF to pack $\sigma^{>\frac{1}{\varepsilon}}$ is $|P_1|$. Further, using (5.2), we have that

$$|R^{\sigma^{>\frac{1}{\varepsilon}}}| < |P_1| < \varepsilon \text{OPT}^{\sigma^{>\frac{1}{\varepsilon}}} \; ,$$

where the last inequality follows from the fact that $\text{OPT}^{\sigma^{>\frac{1}{\varepsilon}}} > \frac{1}{\varepsilon}|P_1|$ since, in OPT, the items of $\sigma^{>\frac{1}{\varepsilon}}$ are packed into $|P_1|$ bins such that there are more than $1/\varepsilon$ items in each bin. Therefore,

$$\text{ADBPA}^{\sigma^{>\frac{1}{\varepsilon}}} = \text{OPT}^{\sigma^{>\frac{1}{\varepsilon}}} - |R^{\sigma^{>\frac{1}{\varepsilon}}}| > (1-\varepsilon)\text{OPT}^{\sigma^{>\frac{1}{\varepsilon}}}$$

□

Lemma 5.8. $\text{ADBPA}^{\sigma^{\leq\frac{1}{\varepsilon}}} > (1-\varepsilon)\text{OPT}^{\sigma^{\leq\frac{1}{\varepsilon}}} - 1$.

Proof. ADBPA, by definition, will reject the type 1 items of $\sigma^{\leq\frac{1}{\varepsilon}}$ and pack the rest of the items of $\sigma^{\leq\frac{1}{\varepsilon}}$. Therefore, there are $\left\lceil \varepsilon|\sigma^{\leq\frac{1}{\varepsilon}}|\right\rceil < \varepsilon\text{OPT}^{\sigma^{\leq\frac{1}{\varepsilon}}} + 1$ items rejected by ADBPA. So, $\text{ADBPA}^{\sigma^{\leq\frac{1}{\varepsilon}}} > (1-\varepsilon)\text{OPT}^{\sigma^{\leq\frac{1}{\varepsilon}}} - 1$. □

Formal Advice Definition. The bin pattern vectors, padded to a length of exactly $1/\varepsilon$, have at most $\frac{1}{\varepsilon}+1$ different possible values per entry (an additional value for the padding). Since the order of the items in a bin does not matter, vectors will have the entries ordered from largest to smallest. The number of patterns is less than the number of ways to pull $1/\varepsilon$ names out of a hat with $1/\varepsilon+1$ names, where repetitions are allowed and order is not significant. Therefore, there are at most $\binom{2/\varepsilon}{1/\varepsilon} \leq \left(\frac{2e/\varepsilon}{1/\varepsilon}\right)^{1/\varepsilon} = (2e)^{\frac{1}{\varepsilon}}$ different bin patterns and at most $\left\lceil \frac{\log(2e)}{\varepsilon} \right\rceil$ bits of advice are need to specify a pattern from an enumeration of all possible patterns, where e is Euler's number.

Per request, the advice string will be xyz, where x is 1 bit in length, y has a length of $\lceil \log(1/\varepsilon) \rceil$ bits to indicate the job type, and z has a length of $\left\lceil \frac{\log(2e)}{\varepsilon} \right\rceil$ bits. The advice string xyz is defined as follows for request r_i.

x**:**	1	if $r_i \in \sigma^{>\frac{1}{\varepsilon}}$
	0	otherwise.
y**:**	if $x = 1$:	If $r_i \in \sigma^{\text{SFF}}$, then the first bit of y is 1. Otherwise, $r_i \notin \sigma^{\text{SFF}}$ and the first bit of y is 0. This is the second advice bit used to pack the $\sigma^{>\frac{1}{\varepsilon}}$ items. The remaining bits are unused and all the bits are set to 0.
	if $x = 0$:	y is the type of r_i encoded in binary.
z**:**		z is a number in binary indicating the bin pattern p_i assigned to r_i. If p_i is undefined (r_i is type 1), then the bits are unused and set to 0.

Immediate from the advice definition is the following fact.

Fact 5.9. ADBPA *uses* $O(\frac{1}{\varepsilon})$ *bits of advice per request.*

Lemma 5.10. *For any* ε*,* $0 < \varepsilon < 1$*,* $\text{ADBPA}(\sigma) > (1-\varepsilon)\text{OPT}(\sigma) - 1$.

Proof. Using Lemma 5.7 and Lemma 5.8, we get that

$$\begin{aligned}\text{ADBPA}(\sigma) &= \text{ADBPA}^{\sigma^{>\frac{1}{\varepsilon}}} + \text{ADBPA}^{\sigma^{\leq\frac{1}{\varepsilon}}} \\ &> (1-\varepsilon)\text{OPT}^{\sigma^{>\frac{1}{\varepsilon}}} + (1-\varepsilon)\text{OPT}^{\sigma^{\leq\frac{1}{\varepsilon}}} - 1 \\ &= (1-\varepsilon)\text{OPT}(\sigma) - 1 \ .\end{aligned}$$

□

5.3.2 Strict $(1/(1-2\varepsilon))$-Competitive Algorithm.

We now define the algorithm DBPA. It behaves, depending on OPT(σ), the optimal number of packed items, in two different manners. One bit, denoted by w, is used to distinguish between the cases. The bit w is sent as advice with each item. In the first case, DBPA runs ADBPA as previously described. In the second case, DBPA is able to pack the items optimally in one of two different ways. The choice of which way to pack the items depends on ε and n. The details of the two cases are as follows.

Case 1: OPT$(\sigma) > 1/\varepsilon$ $(w = 0)$. DBPA will run ADBPA as described previously. The only difference is that the advice per request for ABPA is prepended with an additional bit for w. Since OPT$(\sigma) > 1/\varepsilon$, we get the following corollary to Lemma 5.10 for this case.

Corollary 5.11. *If* $w = 0$*, then, for any* ε*,* $0 < \varepsilon < 1/2$*,* $\text{DBPA}(\sigma) > (1-2\varepsilon)\text{OPT}(\sigma)$.

Case 2: OPT$(\sigma) \leq 1/\varepsilon$ $(w = 1)$. For this case, given w, the algorithm can determine if the number of bins is more than $1/\varepsilon$ as ε and the number of bins is known. If the number of bins given to the algorithm is at least $1/\varepsilon$, then the algorithm packs one item per bin and is optimal.

Otherwise, the number of bins is less then $1/\varepsilon$. For each $r_i \in \sigma$, we define the advice (after w) to be the bin number in which r_i is packed in an optimal packing. This can be done with $\lceil \log(1/\varepsilon) \rceil$ bits. DBPA will pack r_i into the bin as specified by the advice. Note that the packing produced in this case is optimal.

Immediate from the definition of the algorithm and the advice, and Corollary 5.11, we get the following Theorem.

Theorem 5.12. *For any ε, $0 < \varepsilon < 1/2$, DBPA is strictly $\frac{1}{1-2\varepsilon}$-competitive and uses $O\left(\frac{1}{\varepsilon}\right)$ bits of advice per request.*

6

Scheduling on Identical Machines

In this chapter, we present a general framework for the online scheduling problem on m identical machines (see Section 6.1 for a formal definition). This framework depends on a positive $\varepsilon < 1/2$, $U > 0$, and the existence of an optimal schedule S^*, where all jobs with a processing time greater than U are scheduled on a machine without any other jobs. The framework will produce a schedule S such that, up to a permutation of the machines of S, the load of machine i in S is within $\varepsilon L_i(S^*)$ of the load of machine i in S^*, where $L_i(S)$ denotes the load of machine i in the schedule S. This is done using $O\left(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon}\right)$ bits of advice per request. We show that this nearly optimal schedule is sufficient for $(1+\varepsilon)$-competitive algorithms for the makespan and minimizing the ℓ_p norm objectives, and a $(1/(1-\varepsilon))$-competitive algorithm for the machine cover objective.

As is the case with the $(1+\varepsilon)$-competitive algorithm for bin packing problem presented in Chapter 4 and the $(1/(1-\varepsilon))$-competitive algorithm for the dual bin packing problem presented in Chapter 5, the algorithms with a competitive ratio of $(1+\varepsilon)$ (resp. $(1/(1-\varepsilon))$) presented in this chapter could use advice that was based on a schedule produced by a PTAS for the desired objective function as opposed to an optimal schedule.

Using the same techniques as in the lower bound on the amount of advice required to be optimal for the bin packing problem in Section 4.4, we present the analogous results of $\Omega(\log m)$ bits of advice per request are needed for optimality for the objective functions of makespan, machine cover and minimizing the ℓ_p norm.

If $\log m$ is less than the bits of advice per request, then the trivial algorithm with advice that encodes the machine number to schedule each job could be used to obtain a 1-competitive agorithm instead of the framework presented in this chapter.

6.1 Preliminaries

The *online scheduling problem on m identical machines* consists of m identical machines and a request sequence σ. Each $r_i \in \sigma$ is a job with a processing time $v(r_i) > 0$. An assignment of the jobs to the m machines is called a *schedule*. For a schedule S, $L_i(S) = \sum_{r_j \in M_i} v(r_j)$ will denote the *load* of machine i in S, where M_i is the set of jobs assigned to machine i in S. In this chapter, we focus on the following objective functions:

- *Makespan:* minimizing the maximum load over all the machines;
- *Machine cover:* maximizing the minimum load;
- *ℓ_p norm:* minimizing the ℓ_p norm, $1 < p \leq \infty$, of the load of all the machines. For a schedule S, the ℓ_p norm is defined to be $\|L(S)\|_p = \left(\sum_{i=1}^{m}(L_i(S))^p\right)^{1/p}$. Note that minimizing the ℓ_∞ norm is equivalent to minimizing the makespan.

In this chapter, we will use the NEXT FIT heuristic. For machine scheduling, it can be defined analogously to Definition 4.2 for Bin Packing. The machines can be viewed as bins, the jobs as items and the processing time of a job as the size of the item. Finally, NEXT FIT is well defined given an upper bound on the load of a machine in that the upper bound on the load of a machine corresponds to the capacity of a bin.

For simplicity of presentation, we assume that $1/\varepsilon$ is a natural number.

6.2 Online Algorithms with Advice for Scheduling

In this section, we present a general online framework for approximating an optimal schedule, using advice. We apply the general framework to the problems of minimizing the makespan, maximizing machine cover and minimizing the ℓ_p norm. For the minimization problems, we get algorithms with a competitive ratio of $1 + \varepsilon$ and, for the maximization problems, a competitive ratio of $1/(1 - \varepsilon)$. The amount of advice per request is $O\left(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon}\right)$ for $0 < \varepsilon < 1/2$. We note that with $\log m$ bits of advice per

request, the trivial algorithm, that indicates the machine in which to schedule the item, would be optimal.

6.2.1 General Framework

The machines are numbered from 1 to m. Given an ε, $0 < \varepsilon < 1/2$, and $U > 0$, the requested jobs will be classified into a constant number of *types*, using a geometric classification. U is a bound which depends on the objective function of the schedule. Formally, jobs with processing times in the interval $(\varepsilon U, U]$ are called *large jobs*. A large job is assigned type i if its processing time is in the interval $(\varepsilon(1+\varepsilon)^i U, \varepsilon(1+\varepsilon)^{i+1} U]$ for $i \in [0, \lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil)$. Jobs with processing times at most εU are considered *small jobs* and are assigned a type of -1. Jobs with processing times greater than U are considered *huge jobs* and are assigned a type of $\lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil$. The online algorithm does not need to know the actual value of the threshold, U, or the value of ε.

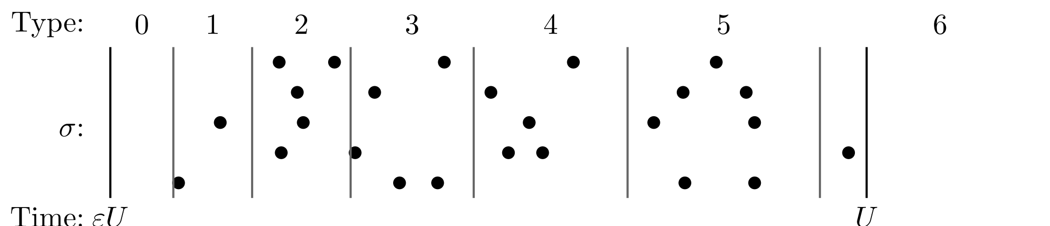

Figure 6.1: An example of the geometric grouping of large jobs, jobs with a processing time in the range $(\varepsilon U, U]$, for $\varepsilon = 1/4$. The illustration shows the processing time of the 25 large jobs, denoted by the black dots, from σ grouped into 7 groups, according to a sorting of the jobs by processing time. The i-th group consists of jobs with a processing time in the interval $(\varepsilon(1+\varepsilon)^i U, \varepsilon(1+\varepsilon)^{i+1} U]$. Even though the range for type 6 is greater than U, only jobs with a processing time at most U will be assigned a type 6.

Let S^* be an optimal schedule for the input at hand with respect to the objective function. In what follows, we will define a schedule S' based on S^* such that, for all i, $L_i(S') \in [L_i(S^*) - \varepsilon U, L_i(S^*) + \varepsilon U]$. Then, based on S', we will define a schedule S such that $L_i(S) \in [(1-\varepsilon)L_i(S^*) - \varepsilon U, (1+\varepsilon)L_i(S^*) + \varepsilon U]$. The advice will be defined so that the online algorithm will produce the schedule S.

The framework makes the following assumption. For all the objective functions that we consider, we will show that there exist optimal schedules for which this assumption holds.

Assumption 6.1. *S^* is an optimal schedule such that each huge job is scheduled on a machine without any other jobs.*

The general framework is defined given a σ, an ε, a U, and an S^* under Assumption 6.1. For the schedule S^*, we assume without loss of generality that the machines are numbered from 1 to m, according to the following order. Assign to each machine the serial number of the first large job scheduled on it. Order the machines by increasing order of this number. Machines without a large job scheduled are placed at the end in an arbitrary order.

We define S' in two steps. In the first step, we schedule the large and huge jobs and, in the second step we schedule the small jobs into the schedule containing the large and huge jobs. Initially, S' is S^* with the small jobs removed. At this point, S' can be described by m patterns, one for each machine. Each such pattern will be called a *machine pattern*. For machine i, $1 \leq i \leq m$, the machine pattern indicates that (1) the machine schedules large or huge jobs, or (2) an empty machine (such a machine may schedule only small jobs). In the first case, the machine pattern is a vector with one entry per large or huge job scheduled on machine i in S'. These entries will be the job type of these jobs on machine i ordered from smallest to largest. Let v denote the maximum length of the machine pattern vectors for S'. The value of v will be dependent on the objective function and U. We later show that for all the objective functions we consider, $v \leq 1/\varepsilon + 1$.

In the following lemma, we extend S' to include the small jobs. The schedule produced is nearly optimal.

Lemma 6.2. *The small jobs of σ can be scheduled on the machines of S' sequentially in a next fit manner from machine 1 to machine m, such that the load for each machine i will be in $[L_i(S^*) - \varepsilon U, L_i(S^*) + \varepsilon U]$.*

Proof. Consider the small jobs in the order in which they arrive. Denote the size of the j-th small job in this order by x_j for $j = 1, \ldots$. For $i = 1, \ldots, m$, let y_i be the total size of small jobs assigned to machine i in S^*. Let $i(0) = 0$, and for $k = 1, \ldots, m$, let $i(k)$ be the minimum index such that $\sum_{j=1}^{i(k)} x_j \geq \sum_{i=1}^{k} y_i$. Finally, for $k = 1, \ldots, m$, assign the small jobs $i(k-1)+1, \ldots, i(k)$ to machine k. (If $i(k) = i(k-1)$, machine k receives no small jobs.)

By definition of $i(k)$ and the fact that all small jobs have size at most εU, the total size of small jobs assigned to machines $1, \ldots, k$ is in $[\sum_{i=1}^{k} y_i, \sum_{i=1}^{k} y_i + \varepsilon U]$ for

$k = 1, \ldots, m$. By taking the difference between the total assigned size for the first $k-1$ and for the first k machines, it immediately follows that the total size of small jobs assigned to machine k is in $[y_k - \varepsilon U, y_k + \varepsilon U]$. □

Note that some machines may not receive any small jobs in this process. We will use the advice bits to separate the machines that receive small jobs from the ones that do not, so that we can assign the small jobs to consecutive machines.

We now define the schedule S, using the following procedure. Assign the machine patterns of S' in the same order to the machines of S. For each large or huge item $r_i \in \sigma$, in the order they appear in σ, assign r_i with type t_i to the first machine in S such that the number of items with of type t_i currently scheduled is less than the number of items of type t_i indicated by the machine pattern. After all the large and huge jobs have been processed, assign the small jobs to the machines of S exactly as they are assigned in S' in Lemma 6.2.

Lemma 6.3. *For* $1 \leq i \leq m$, $L_i(S) \in [(1-\varepsilon)L_i(S^*) - \varepsilon U, (1+\varepsilon)L_i(S^*) + \varepsilon U]$

Proof. By Lemma 6.2 and the fact that jobs of the same type differ by a factor of at most $1+\varepsilon$, we have $L_i(S) \in \left[\frac{1}{1+\varepsilon}L_i(S^*) - \varepsilon U, (1+\varepsilon)L_i(S^*) + \varepsilon U\right]$. The claim follows since $1/(1+\varepsilon) > 1 - \varepsilon$ for $\varepsilon > 0$. □

We have thus shown that the load on every machine is very close to the optimal load in S (for an appropriate choice of U). Note that this statement is independent of the objective function.

We now define the online algorithm with advice, for the general framework, which produces a schedule equivalent to S up to a permutation of the machines. For simplicity of presentation, we assume that this permutation is the identity permutation.

For the first m requests, the general framework receives, as advice, a machine pattern and a bit y, that indicates whether this machine contains small jobs or not. For r_j, $1 \leq j \leq m$, if $y = 0$, the framework assigns the machine pattern to the *highest* machine number without an assigned pattern. Otherwise, the framework will assign the machine pattern to the *lowest* machine number without an assigned pattern. For each request r_i in σ, the type of r_i, denoted t_i, is received as advice. The framework schedules r_i, according to t_i, as follows:

Small Jobs ($t_i = -1$). For scheduling the small jobs, the algorithm maintains a pointer to a machine (initially machine m) indicating the machine that is currently scheduling

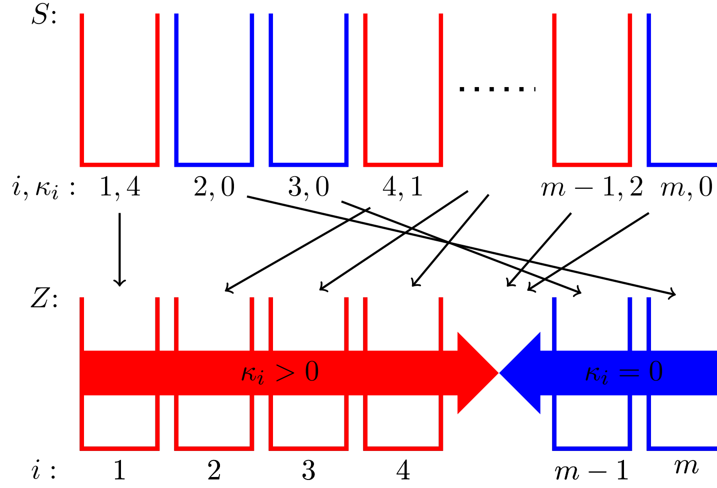

Figure 6.2: In order to produce the exact same schedule as S, the online algorithm must permute the machines of S based on the number of small jobs scheduled per machine (denoted by κ_i for machine i). This figure illustrates such a permutation. Z denotes the schedule produced by the online algorithm. Note that, in Z, machines with small jobs, coloured in red ($\kappa_i > 0$), are in the same relative order as S and machines without small jobs, coloured in blue ($\kappa_i = 0$), are in reverse relative order. This allows the jobs for machines with small jobs to be scheduled from left to right and jobs for machines without small jobs to be scheduled from right to left.

small jobs. With each small job, the algorithm gets a bit of advice x that indicates if this pointer should be moved to the machine with the next serial number. If so, the pointer is moved prior to scheduling the small job. Then, r_i is scheduled on the machine referenced by the pointer.

Large and Huge Jobs ($0 \leq t_i \leq \lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil$). The algorithm schedules r_i on a machine such that the number of items of type t_i is less than the number indicated by its pattern.

6.2.1.1 Formal advice definition

Machine Patterns. For the first m request, a machine pattern is received as advice. Specifically, all possible machine patterns will be enumerated and the id of the pattern, encoded in binary, will be sent as advice for each machine. For large jobs, there are at most v jobs in a machine pattern vector, and each job has one of $\lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil$ possible types. The machine patterns can be described with the jobs ordered from smallest to largest since the order of the jobs on the machine is not important. This is equivalent

to pulling v names out of $\lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil + 1$ names (one name to denote an empty entry and $\lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil$ names for each of the large item types), where repetitions are allowed and order is not significant. Therefore, there are

$$\binom{v + \lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil}{\lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil} \leq \left\lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \right\rceil^{v}$$

different possible machine patterns for the machines scheduling large jobs. Additionally, there is a machine pattern for machines with only small jobs and a machine pattern for machines with only a huge job. Hence, at most $\beta(v) \leq \lceil \log(2 + \lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil^{v}) \rceil < v \log\left(\frac{3\log(1/\varepsilon)}{\log(1+\varepsilon)}\right) + 1$ bits are required to encode the index of a machine pattern in an enumeration of all possible machine patterns in binary. For the cases of makespan, machine cover and ℓ_p norm, $v \leq 1/\varepsilon + 1$ and $\beta(v) < \frac{1+\varepsilon}{\varepsilon} \log\left(\frac{3\log(1/\varepsilon)}{\log(1+\varepsilon)}\right) + 1$.

Advice per Request. In order to define the advice, for each machine $m_i \in S$, we define a value κ_i that is the number of small items scheduled on m_i.

Per request, the advice string will be $wxyz$, where w has a length of $\lceil \log(2 + \lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil) \rceil < \log\left(\frac{3\log(1/\varepsilon)}{\log(1+\varepsilon)}\right) + 1$ bits to indicate the job type, x and y are 1 bit in length (as described above), and z has a length of $\beta(v)$ bits to indicate a machine pattern. $wxyz$ is defined as follows for request r_i:

w:		A number in binary representing the type of r_i.
x:	r_i is a small job:	$x = 1$ if the small job should be scheduled on the next machine. Otherwise, $x = 0$. More formally, let s be the number of small jobs in $\langle r_1, \ldots, r_{i-1} \rangle$. If there exists and an integer $1 \leq j \leq m$ such that $\sum_{k=1}^{j} \kappa_k = s$, then $x = 1$. Otherwise, $x = 0$.
	otherwise:	x is unused and the bit is set to 0.
y:	$i \leq m$:	If $\kappa_i > 0$, $y = 1$. Otherwise, $y = 0$.
	$i > m$:	This bit is unused and set to 0.
z:	$i \leq m$:	z is a number in binary indicating the machine pattern of machine i in S'.
	$i > m$:	z is unused and all the bits are set to 0.

Fact 6.4. *This framework uses less than* $\log\left(\frac{3\log(1/\varepsilon)}{\log(1+\varepsilon)}\right) + \beta(v) + 4$ *bits of advice per request.*

The following theorem, which follows immediately from definition of the general framework and Lemma 6.3, summarizes the main result of this section.

Theorem 6.5. *For any σ, an ε, $0 < \varepsilon < 1/2$, and a $U > 0$ such that there exists an S^* under Assumption 6.1, the general framework schedules σ such that for all machines, $1 \le i \le m$, $L_i(S) \in [(1-\varepsilon)L_i(S^*) - \varepsilon U, (1+\varepsilon)L_i(S^*) + \varepsilon U]$.*

6.2.2 Minimum Makespan

For minimizing the makespan on m identical machines, we define $U = \text{OPT}$, where OPT is the minimum makespan for σ.

Fact 6.6. *If $U = \text{OPT}$, there are no huge jobs as the makespan is at least as large as the largest processing time of all the jobs.*

By the above fact, we know that Assumption 6.1 holds.

Lemma 6.7. *The length of the machine pattern vector is at most $\frac{1}{\varepsilon}$.*

Proof. This lemma follows from the fact that a machine in S^* with more than $\frac{1}{\varepsilon}$ jobs with processing times greater than εOPT is more than the maximum makespan, a contradiction. □

From Lemma 6.7, $v = \frac{1}{\varepsilon}$. Using this value with Fact 6.4 of the general framework, gives the following.

Fact 6.8. *The online algorithm with advice, based on the general framework, uses at most $\frac{2}{\varepsilon}\left(\log\left(\frac{3\log(1/\varepsilon)}{\log(1+\varepsilon)}\right)\right) + 5$ bits of advice per request.*

Theorem 6.9. *Given a request sequence σ, $U = \text{OPT}$ and an ε, $0 < \varepsilon < 1/2$, the online algorithm with advice, based on the general framework schedules the jobs of σ such that the online schedule has a makespan of at most $(1+2\varepsilon)\text{OPT}$.*

Proof. By Fact 6.6, Assumption 6.1 holds and Theorem 6.5 applies.

Let j be a machine with the maximum load in S^*. By Theorem 6.5, $L_i(S) \le (1+\varepsilon)L_i(S^*) + \varepsilon U \le (1+2\varepsilon)\text{OPT}$ as $U = \text{OPT} = L_j(S^*) \ge L_i(S^*)$ for all $1 \le i \le m$. □

6.2.3 Machine Covering

For maximizing the minimum load, i.e. machine covering, on m identical machines, we define $U = \text{OPT}$, where OPT is the load of the machine with the minimum load in S^*.

Lemma 6.10. *There exists an optimal schedule S such that any job with processing time more than that of the minimum load, i.e. a huge job, will be scheduled on a machine without any other jobs.*

Proof. In S, let machine i be the machine with the minimum load and assume that scheduled on some machine $j \neq i$ is a huge job and one or more large or small jobs. We will denote the set of non-huge jobs scheduled on machine j by J. We will define another schedule S^* to be the same schedule as S for all the machines but i and j. In S^*, machine i will schedule the same jobs as S plus all the jobs in J and machine j will only schedule the huge job scheduled on machine j in S. The load on machine j in S^* is greater than OPT as it contains a huge job and the load on machine i in S^* is greater than OPT given that it was OPT in S and jobs were added to it in S^*. If the load of machine i in S is a unique minimum, then S^* contradicts the optimality of S. Otherwise, there exists another machine, $k \neq i$ and $k \neq j$, with the same load as i in S. Machine k has the same load in S^* as it does in S. Therefore, S^* is an optimal schedule. This process can be repeated until a contradiction is found or an optimal schedule is created such that no huge job is scheduled on a machine with any other jobs. □

Lemma 6.11. *There exists an the optimal schedule S such that there are at most $1 + \frac{1}{\varepsilon}$ non-small items scheduled on each machine and huge jobs are scheduled on a machine without any other jobs.*

Proof. By Lemma 6.10, we can transform any optimal schedule S to an optimal schedule S', where all the huge jobs are scheduled on machines without any other jobs.

In S', let machine i be the machine with the minimum load and assume that some machine $j \neq i$ has more than $1 + \frac{1}{\varepsilon}$ large items. We will define another schedule S^* to be the same schedule as S' for all the machines but i and j. Note that machine i has at most $\frac{1}{\varepsilon}$ jobs scheduled and, since its load is U, it cannot contain a huge job which have processing times more than U. In S^*, machine i will schedule the same jobs as S' plus the all the small jobs and the largest job scheduled on machine j in S'. The load on machine j in S^* is greater than OPT as it has at least $1 + \frac{1}{\varepsilon}$ large items scheduled on it and the load on machine i in S^* is greater than OPT given that it was OPT in S' and jobs were added to it in S^*. If the load of machine i in S' is a unique minimum, then S^* contradicts the optimality of S'. Otherwise, there exists another machine, $k \neq i$ and $k \neq j$, with the same load as i in S'. Machine k has the same load in S^* as it does in S'. Therefore, S^* is an optimal schedule. This process can be repeated until a contradiction is found or an optimal schedule is created such that no machine has more than $1 + \frac{1}{\varepsilon}$ non-small items scheduled. □

From Lemma 6.11, $v = 1 + \frac{1}{\varepsilon}$. Using this value with Fact 6.4 of the general framework, gives the following.

Fact 6.12. *The online algorithm with advice, based on the general framework, uses at most* $\frac{3}{\varepsilon}\left(\log\left(\frac{3\log(1/\varepsilon)}{\log(1+\varepsilon)}\right)\right)+5$ *bits of advice per request.*

Theorem 6.13. *Given a request sequence* σ*,* $U = \text{OPT}$ *and an* ε*,* $0 < \varepsilon < 1/2$*, the online algorithm with advice, based on the general framework, schedules the jobs of* σ *such that the online schedule has a machine cover at least* $(1-2\varepsilon)\text{OPT}$*.*

Proof. By Lemma 6.11, Assumption 6.1 holds and Theorem 6.5 applies.

Let j be a machine with the minimum load in S^*. By Theorem 6.5, $L_i(S) > (1-\varepsilon)L_i(S^*) - \varepsilon U \geq (1-2\varepsilon)\text{OPT}$ as $\text{OPT} = L_j(S^*) \leq L_i(S^*)$ for all $1 \leq i \leq m$. □

6.2.4 The ℓ_p Norm

For the objective function of minimizing the ℓ_p norm on m identical machines, we will define $U = \frac{W}{m}$, where W is the total processing time of all the jobs.

The following technical lemma and corollaries are used to prove the lemmas regarding the size of the machine pattern vector and regarding Assumption 6.1.

Lemma 6.14. *If* $\mathbf{x}$ *and* $\mathbf{y}$ *are* m *length vectors with positive real entries such that for some* i *and some* $j \neq i$*:*

- $y_j > y_i$*,* $x_i = y_i + v$ *and* $x_j = y_j - v$*, where* $0 < v < y_j - y_i$*;*
- *for all* $h \neq i$ *and* $h \neq j$*,* $1 \leq h \leq m$*,* $x_h = y_h$*,*

then $\|\mathbf{x}\|_p < \|\mathbf{y}\|_p$*.*

Proof. Define a and b such that $by_i = v$ and $(a+b)y_i = y_j$. Since $by_i = v < y_j - y_i = (a+b)y_i - y_i$, we get that $a > 1$.

$$\begin{aligned}
\|\mathbf{x}\|_p &= \left(\sum_{k=1}^{m}(x_k)^p\right)^{1/p} \\
&= \left((y_i+v)^p + (y_j-v)^p - y_i^p - y_j^p + \sum_{k=1}^{m}(y_k)^p\right)^{1/p} \\
&= \left((y_i+by_i)^p + (ay_i)^p - y_i^p - ((a+b)y_i)^p + \sum_{k=1}^{m}(y_k)^p\right)^{1/p}
\end{aligned}$$

$$
\begin{aligned}
&= \left(y_i^p((1+b)^p + a^p - 1 - (a+b)^p) + \sum_{k=1}^{m}(y_k)^p \right)^{1/p} \\
&= \left(y_i^p \left(\sum_{i=1}^{p-1} \binom{p}{i} b^i - \sum_{i=1}^{p-1} \binom{p}{i} a^{p-i} b^i \right) + \sum_{k=1}^{m}(y_k)^p \right)^{1/p} \\
&< \left(\sum_{k=1}^{m}(y_k)^p \right)^{1/p} \quad \text{, as } a > 1, \\
&= \|\mathbf{y}\|_p
\end{aligned}
$$

□

The following lemma shows that transfering load from a machine over the average, assuming the amount transfered does not drop the load on the machine below the average, to a machine below the average load will strictly improve the ℓ_p norm.

Lemma 6.15. *For any schedule S with machines i and j, where $L_i(S) < W/m$ and $L_j(S) \geq v + W/m$, moving some load v from some machine j to a machine i results in a new schedule S' such that $\|L(S')\|_p < \|L(S)\|_p$.*

Proof. Since $v \leq L_j(S) - W/m$ and $L_i(S) < W/m$, then $v < L_j(S) - L_i(S)$. The new schedule S' is defined such that the load of machine j is $L_j(S) - v$ and the load of machine i is $L_i(S) + v$. By Lemma 6.14, $\|L(S')\|_p < \|L(S)\|_p$. □

The following corollary follows from Lemma 6.15.

Corollary 6.16. *For the ℓ_p norm, $(\sum_{i=1}^{m}(W/m)^p)^{1/p} < \|L(S)\|_p$ for any S.*

Proof. If the load of each machine is not W/m, then there must be at least two machines, i and j, in S such that the load of i is strictly less than W/m and the load of j is strictly greater than W/m. Let $v = L_j(S) - W/m$. Define a new schedule S' such that the load of machine j is W/m and the load of machine i is $L_i(S) + v$. That is, move some fraction of the jobs on machine j equal in processing time to v, splitting a job if necessary, to machine i. By Lemma 6.15, $\|L(S')\|_p < \|L(S)\|_p$. This process can be repeated until all the machines have a load of W/m. □

Lemma 6.17. *In any optimal schedule S, any job with processing time greater than $\frac{W}{m}$, i.e. a huge job, will be scheduled on a machine without any other jobs.*

Proof. There can be at most $m-1$ huge jobs otherwise the total processing time of the jobs would be more than W. In a schedule with a huge job, the machine with the minimum load must be less than $\frac{W}{m}$ (and cannot contain a huge job) otherwise the total processing time of the jobs would be more than W.

If, in the optimal schedule, there is a huge job scheduled with other jobs on machine j, we can move these jobs, one by one, to machine i with minimum load. Note that machine j will have load greater than W/m and machine i must have load less than W/m. Therefore, by Lemma 6.15, this process decreases the ℓ_p norm, contradicting that we started with an optimal schedule. □

Lemma 6.18. *In any optimal schedule S, there are at most $\frac{1}{\varepsilon}$ non-small items scheduled on each machine.*

Proof. By Lemma 6.17, in an optimal schedule, any machine with a huge job will have only one item.

In S, let machine i be the machine with the minimum load and assume that some machine $j \neq i$ has more than $\frac{1}{\varepsilon}$ large jobs. The load of j is at least $(1+\varepsilon)\frac{W}{m}$ and, hence, the load of i is strictly less than $\frac{W}{m}$. This implies that i has less than $\frac{1}{\varepsilon}$ large jobs. Further, removing any large job from j leaves at least $\frac{1}{\varepsilon}$ large jobs on machine j and a load of at least $\frac{W}{m}$ which is still strictly larger than the load of i. By Lemma 6.15, moving a large job from machine j to machine i will decreases the ℓ_p norm, contradicting that S is an optimal schedule. □

From Lemma 6.18, $v = \frac{1}{\varepsilon}$. Using this value with Fact 6.4 of the general framework, gives the following.

Fact 6.19. *The online algorithm with advice, based on the general framework, uses at most $\frac{2}{\varepsilon}\left(\log\left(3\frac{\log(1/\varepsilon)}{\log(1+\varepsilon)}\right)\right)+5$ bits of advice per request.*

Theorem 6.20. *Given a request sequence σ, $U = \frac{W}{m}$ and an ε, $0 < \varepsilon < 1/2$, the general framework schedules the jobs of σ such that the resulting schedule has a ℓ_p norm of at most $(1+2\varepsilon)$*OPT.

Proof. By Lemma 6.17, Assumption 6.1 holds and Theorem 6.5 applies.

The algorithm schedules the jobs such that

$$\begin{aligned}
\|L(S)\|_p &= \left(\sum_{i=1}^{m} (L_i(S))^p\right)^{1/p} \\
&\le \left(\sum_{i=1}^{m} \left((1+\varepsilon)L_i(S^*) + \varepsilon\frac{W}{m}\right)^p\right)^{1/p} && \text{by Theorem 6.5} \\
&\le \left(\sum_{i=1}^{m}(1+\varepsilon)L_i(S^*)^p\right)^{1/p} + \left(\sum_{i=1}^{m}\left(\varepsilon\frac{W}{m}\right)^p\right)^{1/p} \\
&\le (1+\varepsilon)\textsc{opt} + \varepsilon\textsc{opt} && \text{by Corollary 6.16} \\
&= (1+2\varepsilon)\textsc{opt}\ ,
\end{aligned}$$

where we have used the Minkowski inequality in the third line. □

6.3 Lower Bound on the Advice Required for Optimality

Using the same techniques as in the lower bound on the amount advice required for optimality for the bin packing problem (see Section 4.4), we show that $(n-2m)\log m$ bits of advice in total (at least $\left(1-\frac{2m}{n}\right)\log m$ bits of advice per request) is required for any online scheduling algorithm with advice on m identical machines to be optimal for makespan, machine cover or the ℓ_p norm.

Let

$$\begin{aligned}
k &= n - 2m, \\
\sigma_1 &= \left\langle \frac{1}{2^{k+2}}, \frac{1}{2^{k+3}}, \ldots, \frac{1}{2^{k+m+1}}, \frac{1}{2^2}, \ldots, \frac{1}{2^{k+1}} \right\rangle \text{ and} \\
\sigma_2 &= \langle x_1, x_2, \ldots, x_m \rangle\ ,
\end{aligned}$$

where x_i will be defined later in an adversarial manner. The entire adversarial request sequence will be $\sigma = \langle \sigma_1, \sigma_2 \rangle$. The adversarial sequence will be chosen such that the adversary will have a balanced schedule (a load of 1 on each machine) while any algorithm using less than $k \log m$ bits of advice will not. That is, such algorithm will have at least one machine with load greater than 1, and, hence, at least one machine with load less than 1. Such an algorithm will, therefore, not be optimal for makespan, machine cover or the ℓ_p norm.

Fact 6.21. *Every subset of the requests of σ_1 has a unique sum that is less than $1/2$.*

Let T be the set of all possible schedules on m identical machines for the requests of σ_1. The adversary will schedule each of the first m requests of σ_1 on a distinct machine. This distinguishes the m machines from one another. Let V be the set of all possible schedules of the last k requests of σ_1 onto the m distinct machines. Note that $V \subset T$ and that $|V| = m^k$. Let $S_{\sigma_1}^{\text{ADV}} \in V$ be the adversarial schedule of the items of σ_1. $S_{\sigma_1}^{\text{ADV}}$ will be determined later. Let $x_i = 1 - L_i(S_{\sigma_1}^{\text{ADV}})$. Note that, by using Fact 6.21, we have that the N values x_i, $1 \leq i \leq N$, are distinct. Further, note that σ allows for a balanced schedule, where all machines have load 1.

Observation 6.22. *For every $S_{\sigma_1} \in V \setminus S_{\sigma_1}^{\text{ADV}}$, every possible scheduling of the jobs of σ_2 into S_{σ_1} results in a schedule S_σ such that there are at least 2 machines i and j, where $L_i(S_\sigma) < 1$ and $L_j(S_\sigma) > 1$.*

Proof. The sum of the processing time of all jobs of σ_1 is less than 1/2 which implies that the processing time for each x_i is greater than 1/2. Therefore, any machine that schedules more than one job from σ_2 will have a load greater than 1. It follows that such a schedule also has a machine that does not have any job from σ_2, and, hence, has load less than 1. We, therefore, consider a schedule S_σ that schedules a single job from σ_2 on each machine.

Since the sum of the processing time of all the jobs of σ is m, if we have a machine with a load greater than 1, then there must be a machine with a load less than 1. We can therefore assume by way of contradiction that in S_σ all machines have load exactly 1. As each job x_i of σ_2 is scheduled on a distinct machine, we have that in S_σ the total processing time of the jobs from σ_1 on the machine that has job x_i is exactly $1 - x_i$. Therefore, Fact 6.21 implies that S_{σ_1} equals $S_{\sigma_1}^{\text{ADV}}$, a contradiction. □

We are now ready to prove the main theorem of the section. The proof follows the same logic as the analogous proof for the bin packing problem, except that we have to show that a unbalanced scheduled produced by an algorithm using less than $(n - 2m)\log m$ bits of advice in total is not optimal for the makespan, machine cover and the ℓ_p norm objectives.

Theorem 6.23. *Any online algorithm with advice needs at least $(n-2m)\log m$ bits of advice in order to be optimal for the makespan, machine cover and the ℓ_p norm objectives, where m is the number of machines and n is the length of the request sequence.*

Proof. Let ALG be an arbitrary (deterministic) online algorithm with advice for the scheduling problem. Let S_{σ_1} be the schedule produced by ALG for σ_1. If $S_{\sigma_1} \in T \setminus V$,

i.e. S_{σ_1} is such that the first m requests are not scheduled on distinct machines, then, by Observation 6.22, S_σ is not balanced. Therefore, we will assume that the algorithm will schedule the first m requests on m distinct machines, i.e. $S_{\sigma_1} \in V$.

Assume that the online algorithm with advice receives all the advice bits in advance. This only strengthens the algorithm and, thus, strengthens our lower bound. Let $\textsc{alg}(s, u)$ be the schedule produced by $\textsc{alg}$ for request sequence s when receiving advice bits u. Let U be the advice space of $\textsc{alg}$. Since $\textsc{alg}$ gets less than $k \log m$ bits of advice, $|U| < m^k$. It follows that $|\{\textsc{alg}(\sigma_1, u)|u \in U\}| < m^k = |V|$. Therefore, given $\textsc{alg}$, $S_{\sigma_1}^{\textsc{adv}}$ is chosen by the adversary such that $S_{\sigma_1}^{\textsc{adv}} \in V \setminus \{\textsc{alg}(\sigma_1, u)|u \in U\}$. Note that this choice defines σ_2.

We now have, by Observation 6.22, that S_σ has at least 2 machines i and j such that $L_i(S_\sigma) < 1$ and $L_j(S_\sigma) > 1$. Given that there is a balanced schedule with all machines having load 1 for σ, S_σ is not optimal for makespan due to machine j, S_σ is not optimal for machine cover due to machine i, and S_σ is not optimal for the ℓ_p norm by Corollary 6.16. □

7 Reordering Buffer Management

In this chapter, we study the reordering buffer management problem (see Section 7.1 for a formal definition). Initially, we present an algorithm that has a competitive ratio of exactly 1.5, using 2 bits of advice per request. We then enhance this algorithm with more advice per request but still constant advice per request such that we can achieve a competitive ratio arbitrarily close to 1. That is, an algorithm with a competitive ratio of $1 + \varepsilon$ that uses $O(\log(1/\varepsilon))$ advice bits per request.

We complement these results by presenting a lower bound of $\Omega(\log k)$ on the amount of bits of advice required per request for a competitive ratio of 1, where k is the size of the buffer.

If the bits of advice per request are at least $\lceil \log k \rceil$, then the trivial algorithm with advice that encodes the index of an item in the buffer of the next colour to output can be used for an optimal algorithm.

7.1 Preliminaries

The *reordering buffer management problem* (RBM) consists of a random access buffer of size k, a service colour, and a finite request sequence. Initially, the buffer is empty and the service colour is not set to any colour. The request sequence σ consists of a sequence of items that are each characterized by a colour. For $e \in \sigma$, the colour of e is denoted by c_e.

Definition 7.1 (Current request)**.** *The* current request *(or* current item*) of σ is the first item in σ that has not entered the buffer.*

Let request r be the current request. The algorithm does not know the colour of r until it enters the buffer. As long as the buffer is not empty or there exists a current request, the algorithm can perform one of the following actions.

- Any item in the buffer with the same colour as the service colour can be output, i.e. removed from the buffer.
- The service colour can be switched to another colour.
- If there exists a current request, r, and the buffer is not full, r can enter the buffer and the next request in σ becomes the current request.

The goal is to serve σ with as few colour switches as possible.

The following observation is immediate from the fact that any algorithm must perform a number of colour switches at least equal to the number of distinct colours in a request sequence.

Observation 7.2. *The number of distinct colours in σ is a lower bound on the optimal cost.*

In the following, we define the notation of a colour block.

Definition 7.3 (Colour Block)**.** *For an input sequence σ and an algorithm* ALG*, a* colour block *is a set of items from σ with the same colour that are output sequentially by* ALG *on a single colour switch.*

Note that we can have more than one colour block of the same colour in the buffer.

We now define the notion of a lazy reordering buffer management algorithm.

Definition 7.4 (Lazy RBM Algorithm)**.** *An algorithm for the reordering buffer management problem is called* lazy *if it has the two following properties.*

- *If the buffer is not full and there is a current item r, r is brought into the buffer.*
- *The algorithm only changes service colour when the buffer is full or there is no current item, and only if there are no items in the buffer with the current service colour.*

Without loss of generality, we will always assume that OPT is lazy. We can do this as any algorithm ALG can be transformed into a lazy algorithm ALG_{LAZY} at no additional

cost [RSW02]. Let C_σ^{ALG} be the sequence of colour switches performed by ALG for σ. ALG_{LAZY} will serve σ in a lazy manner and maintain an index i that is initially 1. When there are no more items of the service colour in the buffer, ALG_{LAZY} switches to the next colour in C_σ^{ALG} beginning at index i that is in the buffer and updates i to the index used. Hence, ALG_{LAZY} performs at most as many colour switches as ALG.

In this chapter, the notion of time corresponds to the index of the current request. That is, when the current request is r_τ, it is time τ. Hence, at time τ, the previous $r_1, \ldots, r_{\tau-1}$ requests have entered the buffer and, possibly, have been output. An online algorithm does not know yet the colour of r_τ (or the associated advice bits). An algorithm with advice has received the advice bits for $r_1, \ldots, r_{\tau-1}$. After r_n has entered the buffer, the time advances to $n+1$ and the algorithm must serve all the remaining items in the buffer. We say that an item r_i is *older* than r_j if $i < j$ and, conversely, that r_j is *younger* that r_i. Without loss of generality, we will assume that, whenever a lazy algorithm has more than one item of the active colour in the buffer, it will output the oldest item first.

Consider a block of items B output by a lazy OPT beginning at some time τ. In the following lemma, we show that, with a single colour switch, a lazy algorithm can output all the items of B at time τ or later.

Lemma 7.5. *Let $e_1, \ldots, e_\ell$ be a block of items B output by a lazy* OPT *with a single colour switch that occurs at time τ. If, for some i, $1 \leq i \leq \ell$, at some time $\tau' \geq \tau+i-1$, e_i is the oldest item of B in the buffer of a lazy algorithm* ALG*, then, with a single colour switch that occurs at time τ',* ALG *can output all the items of B not output before τ'.*

Proof. At time τ, OPT has the items $e_1, \ldots, e_m$ in its buffer for some $m \in [1, \ell]$ (along with $k-m$ other items). It switches to the colour c of e_1. Since OPT is lazy, every time an item of B is output, a new item enters the buffer, advancing the time ℓ time steps as the items of B be are served (or until $n+1$ if the end of the request sequence is reached). At the start of the time step after serving e_ℓ, OPT has read the input until item $\tau+\ell-1$. The buffer content of OPT at time $\tau+\ell$ is the same as at time τ, except that the m items of colour c have been replaced by a set S of at most m new items with colours other than c that enter the buffer in the interval $[\tau, \min\{\tau+\ell-1, n\}]$.

At time $\tau' \geq \tau+i-1$, based on the assumptions of the lemma, ALG has the items $e_i, \ldots, e_{m'}$ in its buffer for some $m' \in [m, \ell]$ and $k-m'+i-1$ other items. At time τ', ALG can switch to colour c and start outputting the items of B. Since ALG is lazy, it would continue outputting this colour until the buffer contains no items of colour

c. As there are at least $m' - i + 1$ items of colour c in the buffer, the algorithm will read at least $m' - i + 1$ items of colours other than c unless it reaches the end of the request sequence in which case all the items have enter the buffer and ALG will be able to output all the items of B.

Let τ'' be the time when ALG has read $m' - i + 1$ items of colours other than c. At time τ', ALG has read $i - 1$ additional items appearing after time τ. Of these $i - 1$ items, $m' - m$ of them have colour c and $i - 1 - m' + m$ of them have a colour other than c. From τ to τ'', ALG has read $m' - i + 1 + i - 1 - m' + m = m$ new items with a different colour from c. Therefore, ALG has read as many new items as OPT, is at least as far in the input and has served the entire block. □

In this chapter, we use regular expressions to define the language to which strings of advice belong and follow standard notations (cf. [LP97]). The Kleene star * denotes that a string occurs 0 or more times, e.g. the a^*bc language contains bc, abc, $aabc$, $aaabc$, etc. We use **?** to denote that a string occurs 0 or 1 times, e.g. the $ab?c$ language contains ac and abc. The symbol ∪ is used to denote two possible strings, e.g. the $a \cup b$ language contains a and b. The parentheses **(** and **)** indicated sub-expressions which should be evaluated from the inner most set of parentheses out.

7.2 1.5-Competitive Algorithm with 2 Bits of Advice per Request

In this section, we present an algorithm that has a tight competitive ratio of 1.5 and uses 2 bits of advice per request. The advice received with each item of the input indicates the manner in which it is served by a lazy optimal algorithm.

7.2.1 Definition of the Advice and the Algorithm

The Advice. Each item $e \in \sigma$ is assigned a *type*, denoted t_e. There are three possible values, $t_e \in \{\text{HOLD}, \text{LIST}, \text{COMP}\}$. For item e, the advice bits are the value of t_e encoded in two bits.

For the sake of analysis, given σ, we define a parallel advice sequence of item types $T_{2\text{BL}}(\sigma)$. It is defined based on a fixed lazy optimum OPT in the following manner. For each $e \in \sigma$, t_e is defined based on the colour block B of OPT that contains e (see

Figure 7.1). Let f be the youngest item in B that was in the buffer at the time that OPT began serving B. For $e \in B$, t_e is

HOLD if e is older than f,

LIST if e is no older than f and not the youngest item of B,

COMP if e is the youngest item of B. (This is the last item of the block and the block is considered to be *complete*.)

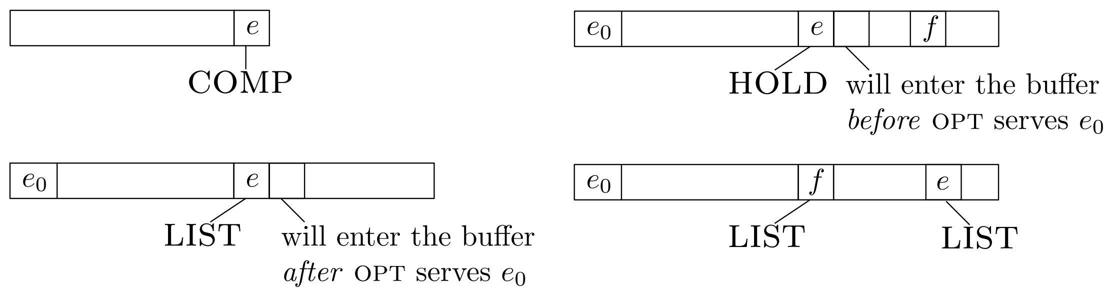

Figure 7.1: An illustration of a block B and the three possible types of items as based on a fixed lazy OPT. Let f be the youngest item in B that was in the buffer at the time that OPT began serving B. Top-left: item e is assigned type COMP since e is youngest item of B. Top-right: item e is assigned type HOLD since e is older than f. Bottom-left: item e is assigned LIST since e is f. Bottom-right: item e is assigned LIST since e is younger than f and e is not the youngest item of B. The LIST type indicates that OPT starts serving this colour block before all items have been read into the buffer; the remaining items enter the buffer while the colour block is being served (and while other items of the colour block are being removed).

From the definition of the advice, we get the following observation that each block contains 0 or more HOLD items followed by 0 or more LIST items followed by a COMP item.

Observation 7.6. *The string produced by the concatenation of types of the items in a colour block when ordered from oldest to youngest forms an expression in the language defined by the regular expression,*

$$\text{HOLD}^*\text{LIST}^*\text{COMP} .$$

We can now define the notion of the type of a block at time τ. Contrary to an item type that is fixed, the type of a block changes over time depending on the youngest

item of the block currently in the buffer of a given algorithm. At time τ, let e be the youngest item of block B in the buffer. The type of B, denoted by $t_B(\tau)$, is t_e. Blocks of type LIST and COMP are ranked based on the age of their items. At time τ, let two blocks, B_1 and B_2, be of the same type t. Let r_i be the oldest item of type t in B_1, and let r_j be the oldest item of type t in B_2. (Note that r_i and r_j may have already been output at time τ.) We say that B_1 is older than B_2 if r_i is older than r_j and B_1 is younger than B_2 if r_i is younger than r_j.

The Algorithm 2 BIT LAZY. 2 BIT LAZY (2BL) is a lazy algorithm that works in following manner. At time t, if the buffer is not full, the current request, r_t, is brought into the buffer. If the buffer is full and contains an item with the same colour as the active colour c, then the oldest item of colour c is output. Otherwise, the buffer is full and there are no items with the same colour as the active colour. At this point, the algorithm will perform a colour switch.

In order to choose the next colour to use, we need to define the notion of an advice block. An *advice block* is defined based on a sequence of items, that is some prefix of the items of σ, and their type. For the sequence of items $\mathcal{B} = \langle r_1, \ldots, r_j \rangle$, $1 \leq j \leq n$, sort the items according to colour and, within each colour, sort the items in decreasing order according to their age. Each advice block $B \in \mathcal{B}$ of colour c consists of the maximum number of consecutive items of colour c terminated by an item of type COMP or the last item of colour c in the sorted sequence such that the string produced by the concatenation of the types of the items in B is in the language HOLD*LIST*COMP?. Note that, for σ, all the advices blocks of 2BL are disjoint and correspond to different colour blocks of OPT. That is, an advice block B corresponds to a colour block B_{OPT} of OPT if, for all $e_i \in B$, $e_i \in B_{\text{OPT}}$. For an advice block B, the corresponding colour block in OPT is denoted by B_{OPT}. We say that an advice block B is in the buffer at some time τ if there is an item $e \in B$ that is in the buffer at the beginning of time step τ. The type and the age of an advice block at a given time is defined in the same manner as the type of a colour block at a given time.

At time step τ, the next colour is chosen based on the advice blocks in the buffer of 2BL as defined for $\langle r_1, \ldots, r_{\tau-1} \rangle$, according to the following rules arranged in order of precedence. Note that all these requests have entered the buffer of 2BL at time τ and the algorithm has received the associated advice.

1. If there is an advice block B with type COMP, then switch to the colour of the oldest COMP advice block,

2. else, if there is an advice block B with type LIST and at least 2 of the items of B_{OPT} that have type LIST have entered the buffer, then switch to the colour of B,

3. else, switch to the colour of the advice block B with type LIST that has the largest cardinality. (Ties are broken by choosing the colour of the oldest advice block.)

7.2.2 The Analysis

For the analysis, unless explicitly stated, we will always consider the advice blocks as defined with respect to the entire request sequence, σ, and their types, $T_{2\text{BL}}(\sigma)$. Along with the definition of 2BL, the next two lemmas show that 2BL is well defined for any request sequence and will be able to output all the items.

Lemma 7.7. *Whenever* 2BL *has a full buffer, it contains a* LIST *or a* COMP *type item.*

Proof. Assume for contradiction that at some time τ, $k < \tau \leq n+1$, the buffer of 2BL is full and all the items have type HOLD. By the definition of the advice, for the advice blocks currently in the buffer of 2BL, the items of type LIST and COMP are younger than the HOLD items and, by the definition of 2BL, the HOLD items will be served before the LIST and COMP items. Therefore, for all the advice blocks in the buffer, the LIST and COMP items arrive at a time greater than τ. By the definition of the advice, the items of type HOLD are kept in the buffer of OPT and are present in the buffer of OPT at time τ. So, at time τ, OPT cannot remove any of the items in its buffer before it reads another item with a matching colour from the input sequence. This is a contradiction, as OPT has to remove an item at time τ since its buffer is full. □

Lemma 7.8. *At time $n+1$, the buffer of* 2BL *contains only advice blocks of type* COMP.

Proof. By the definition of the advice, the last item of every advice block is COMP and, at time $n+1$, all the items have entered the buffer which implies that the COMP item of each advice block has entered the buffer. If there is an advice block in the buffer of 2BL without a COMP item, this means that a COMP item was served before an older item which contradicts the definition of 2BL. □

Based on the order of precedence for the colour switches of 2BL, the first priority are advice blocks of type COMP and, by the definition of the advice, the youngest item

of the advice block is in the buffer and all the items of the advice block that have not yet been output can be output on a single colour switch. The second priority are advice blocks of type LIST such that at least two LIST items have entered the buffer (and possibly have been output). The following two lemmas show that there can be at most one advice block of type LIST in the buffer of 2BL for which at least two LIST items have entered the buffer and that the remaining items of the advice block can be output on a single colour switch. The first lemma also shows that the second colour switching priority is well defined.

Lemma 7.9. *At any time, the buffer of* 2BL *contains at most one advice block B of type* LIST *such that at least two* LIST *items of B have entered the buffer of* 2BL*.*

Proof. At time some time τ, consider a buffer of 2BL with two advice blocks, B' and B'', both of type LIST such that at least two LIST items of B'_{OPT} and B''_{OPT} have entered the buffer. Since B' and B'' are of type LIST neither block contains an item of type COMP. At time τ, the definition of the advice implies that OPT is serving both B'_{OPT} and B''_{OPT} which is a contradiction. □

Lemma 7.10. *At time τ', let B be an advice block of type* LIST *such that at least two* LIST *items of B have entered the buffer of* 2BL*. With a single colour switch, at time τ' or later,* 2BL *can serve all the items of B that have not yet been output.*

Proof. At time τ', let e_i be the oldest item of B_{OPT} in the buffer of 2BL, where i is the index of e_i in B_{OPT} when the items are sorted by decreasing age. At time τ', by Lemma 7.9, there is only one advice block of type LIST with at least two LIST items that have entered the buffer and, from the definition of the algorithm and the advice, all the other advice blocks have type LIST or HOLD, and their items are output after τ' by OPT.

Let e_j be the oldest item of B_{OPT} in the buffer of OPT at time τ', where j is the index of e_j in B_{OPT} when the items are sorted by decreasing age. The item e_j must not be older than e_i otherwise, at time τ' the buffer of B_{OPT} would contain all the items in the buffer of 2BL at τ' plus $i - j + 1$ items which is more than k. Therefore, $j \geq i$ and $\tau' = \tau + j - 1 \geq \tau + i - 1$, where τ is the time that OPT starts serving B_{OPT}. The claim then follows from Lemma 7.5. □

The third priority are advice blocks that are of type LIST with only one LIST item in the buffer. 2BL will only switch to the colour of such a block if its buffer does not contain advice blocks of a higher priority which implies that the buffer of 2BL only contains

advice blocks of the third priority or HOLD advice blocks. The following lemma shows that, in this case, by switching to the colour of the largest cardinality LIST advice block either all items in corresponding OPT block are output, or an item of LIST or COMP of another LIST advice block in the buffer will enter the buffer, ensuring a higher priority advice block for the subsequent colour switch.

Lemma 7.11. *At time τ, let $\mathcal{B}_{\text{LIST}}$ be the set of* LIST *advice blocks in the buffer of* 2BL. *If, at time τ, all the advice blocks are of type* HOLD *or* LIST *such that, for all blocks, at most a single* LIST *item has entered the buffer, then switching to the colour of the largest advice block $B^* \in \mathcal{B}_{\text{LIST}}$ at τ guarantees that either all the items of B^* are output before the next colour switch, or a* LIST *or* COMP *item from another* LIST *advice block in $\mathcal{B}_{\text{LIST}}$ enters the buffer.*

Proof. By the definition of the advice, the algorithm and the assumption of the lemma regarding the state of the buffer of 2BL, 2BL is at least at the same time step as OPT when OPT begins outputting one of the blocks, B'_{OPT}, that correspond to $B' \in \mathcal{B}_{\text{LIST}}$ and, for all $B \in \mathcal{B}_{\text{LIST}}$, the oldest item of B (and B_{OPT}) is in the buffer of 2BL. By Lemma 7.5, if B^* is B', all the items of B^* will be output. Otherwise, B^* is at least as large as B', for which 2BL is at least at the same time step as OPT when OPT begins outputting B'_{OPT}. Therefore, 2BL must bring in the next item e of B' otherwise OPT would not be able to serve B'_{OPT} with a single colour switch. By the definition of the advice, e must be of type LIST or COMP. □

Essentially, from the previous lemma and Lemma 7.10, we see that, every time 2BL makes two colour switches to serve an optimal block B_{OPT}, there is another block that is output by 2BL with a single colour switch, giving a competitive ratio of 1.5.

Theorem 7.12. 2BL *is* 1.5*-competitive.*

Proof. Let b^{COMP} be the colour blocks of OPT without a LIST item. By the definition of the algorithm and the advice, 2BL will perform b^{COMP} colour switches to serve those blocks.

Consider the colour blocks of OPT that contain a LIST item. Let b_1^{LIST} be the colour blocks of OPT that 2BL serves with a single colour switch, and let b_2^{LIST} be the colour blocks of OPT that 2BL serves with more than a single colour switch. Each colour block B_{OPT} counted in b_2^{LIST} will be served with two colour switches by 2BL. The first colour switch occurs after the first LIST item of B_{OPT} enters the buffer of 2BL and the second colour switch occurs after the second LIST item of B_{OPT} enters the buffer of 2BL. By

Lemma 7.10, the items of B_{OPT} not yet output will be output by the second colour switch by 2BL. Hence, for 2BL, the blocks counted in b_2^{LIST} take $2b_2^{\text{LIST}}$ colour switches.

From the definition of the advice, the algorithm and Lemma 7.11, the b_2^{LIST} blocks can only occur when a colour switch is made to the largest advice block B of type LIST in a buffer containing only LIST advice blocks, where at most single LIST item has entered the buffer, and HOLD type advice blocks. By Lemma 7.11, after serving B, when all the items of B_{OPT} are not output, either a LIST or a COMP type item e from another LIST advice block B' enters the buffer of 2BL. If t_e is COMP, all the items of B'_{OPT} are in the buffer and 2BL will output all of them on the next colour switch to the colour of e. If t_e is LIST, then 2BL will output the all the items of colour block B'_{OPT} on the next colour switch to the colour of e by Lemma 7.10. Therefore, every time there is a colour block counted in b_2^{LIST}, there is a colour block that 2BL will serve in its entirety that is counted in b_1^{LIST} which implies $b_2^{\text{LIST}} \leq b_1^{\text{LIST}}$.

From Observation 7.2, the cost of OPT is the number of colours blocks which is $b^{\text{COMP}} + b_1^{\text{LIST}} + b_2^{\text{LIST}}$. From this and the fact that $b_2^{\text{LIST}} \leq b_1^{\text{LIST}}$, we get that $b_2^{\text{LIST}} \leq \text{OPT}(\sigma)/2$. Therefore, the number of colour switches of performed by 2BL is $b^{\text{COMP}} + b_1^{\text{LIST}} + 2b_2^{\text{LIST}} \leq \frac{3}{2}\text{OPT}(\sigma)$. □

In the following theorem, we complement the previous result by showing that the upper bound on 2BL is tight.

Theorem 7.13. *The competitive ratio of* 2BL *is at least* 1.5.

Proof. Let $R(c_1, c_2) = \left\langle c_1^{\lceil \frac{k}{2} \rceil}, c_2^{\lfloor \frac{k}{2} \rfloor}, c_2^{\lceil \frac{k}{2} \rceil}, c_1^k \right\rangle$ be a request sequence where x^y denotes a item of colour x requested y times in a row and c_1, c_2 are different colours. Arbitrarily long request sequences can be created from R by alternatively swapping the colours c_1 and c_2 for c_3 and c_4. That is,

$$\sigma = \langle R(c_1, c_2), R(c_3, c_4), R(c_1, c_2), R(c_3, c_4), \ldots \rangle \ .$$

For each R, 2BL will make three colour switches while OPT will make only 2. □

7.3 An Optimal Algorithm with $2 + \lceil \log k \rceil$ Bits of Advice per Request

In this section, we consider the 1.5-competitive algorithm from the previous section and enhance it by giving it access to the order in which OPT serves the LIST colour blocks.

This enables the algorithm to serve σ with no more colour switches than OPT. However, at any given time step, there could be $k - 1$ other LIST advice blocks in the buffer. So, encoding the position of an advice block in an ordered list of LIST advice blocks requires at worst $\lceil \log k \rceil$ bits which is actually enough to run the trivial algorithm with advice that indicates the colour switch at each time step. The ordered list of LIST advice blocks is called the waiting list and the algorithm described in this section is called 2 BIT LAZY WITH WAITING LIST (2BWL). In the two subsequent sections, we will modify the advice string of the algorithm so as to obtain a $(1 + \varepsilon)$-competitive algorithm with constant bits of advice per request. That is, we will allow for a multiplicative factor of ε more colour switches but generate the item types such that the position of the LIST advice blocks in the waiting list can be encoded with $O(\log \frac{1}{\varepsilon})$ bits.

7.3.1 The Advice, the Waiting List and the Algorithm

The Advice. Given a σ and a fixed lazy OPT, each item $e \in \sigma$ is assigned a type $t_e \in \{\text{HOLD}, \text{LIST}, \text{COMP}\}$ as defined in the previous section, i.e. $T_{2\text{BWL}}(\sigma) = T_{2\text{BL}}(\sigma)$.

Consider the advice blocks of σ as described by $T_{2\text{BWL}}(\sigma)$. Each advice block B with an item of type LIST is assigned a value u_B that is a number from 0 to $k - 1$. The value u_B is defined based on the oldest LIST item in each block with an item of type LIST. For the oldest LIST item $e \in B$, u_B is defined to be the position of B_{OPT} in an ordering (beginning at position 0), from the first colour block output by OPT to the last colour block output by OPT, of the corresponding colour blocks of OPT to the LIST advice blocks present in the buffer of 2BWL at the time step that e enters the buffer. For the sake of analysis, we define another sequence, $U_{2\text{BWL}}(\sigma)$, to be the sequence of the values of u_B ordered, from oldest to youngest, by the age of the oldest LIST item in each advice block with an item of type LIST in σ.

The Waiting List. The algorithm maintains a *waiting list* of advice blocks of type LIST. This list contains only advice blocks that have, at some time, type LIST. At the beginning of the algorithm, the waiting list is empty. Whenever an advice block B of type LIST appears (i.e. the first item of type LIST for some advice block is read from the input), it is inserted into the waiting list. The *initial* position of the block on the waiting list, i.e., the position where the block is inserted, is defined by the value u_B of the block. A value of 0 denotes the head of the list. Note that when a new advice block

is inserted into or removed from the waiting list, the position of the other advice blocks on the list can change and, whenever an advice block on the waiting list is output, that advice block is removed from the waiting list.

The Algorithm 2 BIT LAZY WITH WAITING LIST. 2BWL is a lazy algorithm that runs just as 2BL. The difference is the manner in which 2BWL will choose the next colour for each colour switch. At time step τ, the next colour is chosen based on the advice blocks in the buffer of 2BWL as defined for $\langle r_1, \ldots, r_{\tau-1} \rangle$, according to the following rules arranged in order of precedence.

1. If there is an advice block B with type COMP, then switch to the colour of the oldest COMP advice block,
2. else, if there is an advice block with type LIST, switch to the advice block B at the front of the waiting list and remove B from the waiting list.

7.3.2 The Analysis

For the analysis, unless explicitly stated, we will always consider the advice blocks as defined with respect to the entire request sequence, σ, and their types, $T_{2\text{BWL}}(\sigma)$. 2BWL uses an advice sequence $T_{2\text{BWL}}(\sigma) = T_{2\text{BL}}(\sigma)$ and processes items in essential same manner as 2BL. That is, 2BL and 2BWL switch to the colour of a COMP or a LIST advice blocks in the same order of precedence. The difference being the way in which each algorithm will choose which colour of which LIST blocks to use for the next colour switch. Therefore, Lemma 7.7 and Lemma 7.8 hold for 2BWL as they held for 2BL which implies that 2BWL is well defined for any request sequence.

Theorem 7.14. 2BWL *is optimal.*

Proof. Let b^{COMP} be the colour blocks OPT without a LIST item. By the definition of the algorithm and the advice, 2BWL will perform b^{COMP} colour switches to serve those blocks.

Let b^{LIST} be the colour blocks of OPT with a LIST item. By the definition of the algorithm and the advice, 2BWL will only serve LIST advice blocks if the buffer contains only HOLD and LIST items and, for every LIST advice block B in the buffer, the oldest item of B_{OPT} is in the buffer. From this and the fact that the algorithm serves the LIST advice blocks in the same order as OPT serves the corresponding colour blocks, 2BWL

will be at the same time step or further ahead than OPT when 2BWL begins outputting B as compared to when OPT begins outputting B_{OPT}. With Lemma 7.5, this implies that 2BWL will output every block in b^{LIST} with a single colour switch.

Therefore, the number of colour switches of 2BWL is $b^{\text{COMP}} + b^{\text{LIST}} = \text{OPT}(\sigma)$. □

7.4 A $(1+\varepsilon)$-Competitive Algorithm with $2+\lceil\log k\rceil$ Bits of Advice per Request

In this section, we modify the 2BWL algorithm and the advice from the previous section and present an algorithm that is $(1+\varepsilon)$-competitive and uses $2+\lceil\log k\rceil$ bits of advice per request. This algorithm has a worse competitive ratio than 2BWL, but, for a cost of an additional multiplicative factor of ε colour switches, the number of LIST advice blocks that are placed in the waiting list is significantly reduced. In the next section, we will modify the algorithm presented in this section such that the position of the advice blocks in the waiting list will be encoded in a constant number of bits.

7.4.1 The Advice and the Algorithm

A New Item Type. Given a σ and a fixed lazy OPT, each item $e \in \sigma$ is a assigned a type $t_e \in \{\text{HOLD}, \text{LIST}, \text{READY}, \text{COMP}\}$, where HOLD, LIST and COMP are defined as in Section 7.2. READY is a new item type that is used along with COMP to split an advice block with an item of type LIST into two separate advice blocks such that the algorithm can output both advice blocks with a single colour switch or to delay the time when an advice block with an item of type LIST will be output (see below for the details). Note that the four item types can still be encoded in 2 bits.

We extend the notion of an advice block for items of type READY. The definition remains the same except that the string produced by the concatenation of the types of the items in advice block is in the language $\text{HOLD}^*(\text{LIST}^* \cup \text{READY}^*)\text{COMP}?$. With the addition of the type READY as detailed below, the advice blocks are still disjoint, but there can be more advice blocks than colour blocks of OPT. Specifically, there could be two advice blocks that are the prefix and the suffix of the items of a colour block of OPT.

An advice block B in the buffer of the algorithm that contains an item of type READY is of type READY if the COMP item of B is not yet in the buffer. At time τ, for

two advice blocks of type READY in the algorithm's buffer, we say that B_1 is older than B_2 if the oldest READY item in B_1 is older than the oldest READY item in B_2.

The Algorithm 2 BIT LAZY WITH REDUCED WAITING LIST. 2 BIT LAZY WITH REDUCED WAITING LIST (2BRWL) is a lazy algorithm that runs just as 2BL and 2BWL. The difference is the manner in which 2BRWL will choose the next colour for each colour switch. At time step τ, the next colour is chosen based on the advice blocks in the buffer of 2BRWL as defined for $\langle r_1, \ldots, r_{\tau-1} \rangle$, according to the following rules arranged in order of precedence.

1. If there is an advice block B with type COMP, then switch to the colour of the oldest COMP advice block,
2. else, if there is an advice block with type READY, then switch to the colour of the oldest READY advice block,
3. else, if there is an advice block with type LIST, switch to the advice block B at the front of the waiting list and remove B from the waiting list.

7.4.1.1 Building the Advice Sequences

The advice sequence containing the types of the items, $T_{2\text{BRWL}}(\sigma)$, and the advice sequence containing the waiting list positions, $U_{2\text{BRWL}}(\sigma)$, are constructed *offline*, based on an *optimal* lazy solution OPT for the instance σ. The idea of the construction is as follows. We initially assign each input item e type $t_e \in \text{HOLD}, \text{LIST}, \text{COMP}$ as it is assigned for 2BL. That is, initially, $T_{2\text{BRWL}}(\sigma) = T_{2\text{BL}}(\sigma)$ and $U_{2\text{BRWL}}(\sigma) = U_{2\text{BWL}}(\sigma)$. This means that OPT is the number of advice blocks initially described by σ and $T_{2\text{BRWL}}(\sigma)$. Then, 2BRWL is simulated on σ with the advice sequences $T_{2\text{BRWL}}(\sigma)$ and $U_{2\text{BRWL}}(\sigma)$. Given some constant C (defined later), whenever the waiting list of 2BRWL contains at least C advice blocks of sizes that differ by given multiplicative factor, they are removed from the waiting list at a cost of 2 additional colour switches. This is accomplished by replacing LIST type items with type READY and type COMP in $T_{2\text{BRWL}}(\sigma)$. This modification introduces items of type READY and increases the number of advice blocks as defined by σ and $T_{2\text{BRWL}}(\sigma)$ (and updates $U_{2\text{BRWL}}(\sigma)$ as advice blocks with READY items are not put into the waiting list). The number of colour switches performed by 2BRWL increases as the number of advice blocks defined for σ by $T_{2\text{BRWL}}(\sigma)$ increases, but the

number of advice blocks in the waiting list at any given time will be bounded by a constant and this will allow us, in the next section, with one more alteration, to encode the position of the blocks in the waiting list with a constant amount of advice.

In order to formally define the procedure to generate $T_{2\mathrm{BRWL}}(\sigma)$ and $U_{2\mathrm{BRWL}}(\sigma)$, we will first define three procedures that are used to modify the advice data. The input to the procedures are the full advice blocks, i.e. the advice blocks defined with respect to the entire request sequence σ and $T_{2\mathrm{BRWL}}(\sigma)$.

The first procedure removes an advice block B from the waiting list and adjusts the position information for the advice blocks that are inserted behind B in the waiting list.

Procedure Remove(B)

- Remove u_B from $U_{2\mathrm{BRWL}}(\sigma)$.
- For each advice block B' that was inserted into the waiting list, at a position behind B, after B was inserted but prior to B being output, decrease $u_{B'}$ by one.

The next procedure will split an advice block B into two advice blocks such that both will end with an item of type COMP by reassigning the oldest LIST item to type COMP. Additionally, the other items in B with LIST type are changed to READY. This ensures that neither of the new blocks will be placed on the waiting list (see Figure 7.2). The first advice block is called an *early* block and the second advice block is called a *late* block.

Procedure Split(B)

- Run Remove(B).
- Reassign type COMP to the first item of block B which has type LIST assigned.
- Reassign type READY to the remaining items of B which have type LIST assigned.

Let B be an advice block that is processed by the Split procedure that produces an early block B' and a late block B''. The colour block of OPT that corresponds to B, B_{OPT}, is the same block that corresponds to both B' and B'', i.e. $B_{\mathrm{OPT}} = B'_{\mathrm{OPT}} = B''_{\mathrm{OPT}}$.

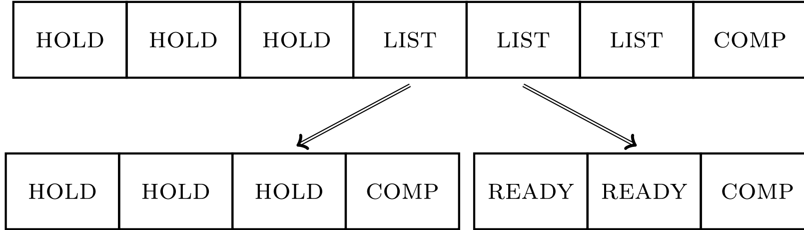

Figure 7.2: An example of the procedure Split(B) as applied to an advice block B (top) of items with types HOLD, LIST and COMP. B is split into two new advice blocks (bottom). The early block (bottom left) is made up of the HOLD items and the first LIST item of B which is changed to COMP. The late block (bottom right) has the rest of the items of B with all the LIST items changed to READY.

The procedure Split(B) makes 2BRWL output the items from the early block of B before they are output by OPT, generating free space in the buffer of 2BRWL. This allows some advice blocks to stay in the buffer until the next item of the advice block is read from the input, and only then to be output by 2BRWL. These are the advice blocks for which we run procedure Postpone(B) (see Figure 7.3).

Procedure Postpone(B)

- Run Remove(B).
- Reassign type HOLD to the first item of B which has type LIST assigned.
- Reassign type READY to the remaining items of B which have type LIST assigned.

Advice blocks processed in this way are called *postponed blocks*. Note that no additional advice blocks are produced by the procedure Postpone. For a block B, that is processed by the Postpone procedure, B remains the only advice block that corresponds to B_{OPT}.

An advice block B that is removed from the waiting list by these procedures will never be inserted into the waiting list of 2BRWL. This is the reason, for both procedures Split(B) and Postpone(B), that the procedure Remove(B) is run.

For this algorithm, the goal is to ensure that there are not too many advice blocks of a similar size in the waiting list. To formalize the notion of similar size, we define the class of a LIST advice block. In the definition, $s(B)$ is the number of items in the buffer of OPT when OPT starts serving B_{OPT}. That is, one LIST item and all the HOLD items of B. Note that, for an early block B' resulting from Split(B), we have that the number of items in B' is $s(B)$.

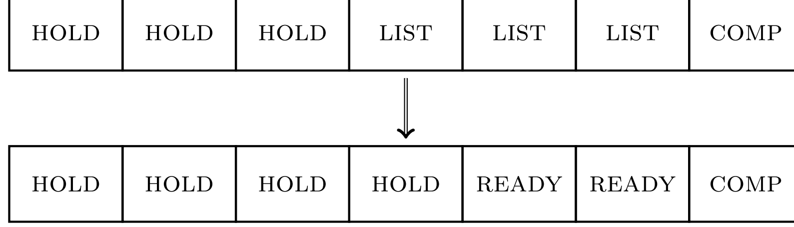

Figure 7.3: An example of the procedure Postpone(B) as applied to an advice block B (top) of items with types HOLD, LIST and COMP. The first LIST item of B is changed to HOLD and the remaining LIST items are changed to READY as shown in the postponed block (bottom).

Definition 7.15 (Class of an Advice Block). *The* class *of an advice block B of type* LIST *is $\lfloor \log s(B) \rfloor$, where $s(B)$ is one more than number of items in B of type* HOLD.

From the definition of a class of a LIST advice block B, we have that, for class i, $s(B) \in [2^i, 2^{i+1})$. Therefore, for a buffer of size k, there are $\lfloor \log k \rfloor + 1$ possible classes, i.e. classes run from 0 to $\lfloor \log k \rfloor$.

We are now ready to formally define $T_{\text{2BRWL}}(\sigma)$ and $U_{\text{2BRWL}}(\sigma)$. Initially, $T_{\text{2BRWL}}(\sigma) = T_{\text{2BL}}(\sigma)$ and $U_{\text{2BRWL}}(\sigma) = U_{\text{2BWL}}(\sigma)$. Simulate 2BRWL on σ with $T_{\text{2BRWL}}(\sigma)$ and $U_{\text{2BRWL}}(\sigma)$. Let n_i^τ be the number of advice blocks of class i in the waiting list at time τ. Let

$$(\tau^*, i^*) = \operatorname*{arg\,max}_{\tau \in [1,n], i \in [0, \lfloor \log k \rfloor]} n_i^\tau \ .$$

If $n_{i^*}^{\tau^*} \geq C$, let B_1 and B_2 be the two last advice blocks of class i^* in the waiting list at time τ^*, and perform the two following operations.

- Run Split(B_1) and Split(B_2).
- For all remaining advice blocks, B', of class i^* in the waiting list at τ^*, run Postpone(B').

By running these procedures, the advice sequences are updated such that 2BRWL will not insert these $n_{i^*}^{\tau^*}$ blocks into the waiting list. This process is repeated using the updated advice sequences until, for all $1 \leq \tau \leq n$ and $0 \leq i \leq \lfloor \log k \rfloor$, $n_i^\tau < C$.

7.4.2 The Analysis

For the analysis, unless explicitly stated, we will always consider the advice blocks as defined with respect to the entire request sequence, σ, and their types, $T_{\text{2BRWL}}(\sigma)$. The

definition of 2BRWL and the two following lemmas show that 2BRWL is well defined for any request sequence. The proof of the first lemma follows that of Lemma 7.7 for the items of type HOLD in both $T_{\text{2BL}}(\sigma)$ and $T_{\text{2BRWL}}(\sigma)$ plus a second case for those items of type HOLD in $T_{\text{2BRWL}}(\sigma)$, from running the procedure Postpone, that are type LIST in $T_{\text{2BL}}(\sigma)$. The proof of the second lemma is analogous to that of Lemma 7.8.

Lemma 7.16. *Whenever* 2BRWL *has a full buffer, it contains a* LIST*,* READY *or* COMP *type item.*

Proof. Assume for contradiction that at some time τ, $k < \tau \leq n + 1$, the buffer of 2BRWL is full and all the items have type HOLD and the original type of these items was HOLD. That is, the type of the items in $T_{\text{2BL}}(\sigma)$ is HOLD. By the definition of the advice sequence $T_{\text{2BRWL}}(\sigma)$, for the advice blocks currently in the buffer of 2BRWL, the items of type LIST, READY and COMP are younger than the HOLD items and, by the definition of 2BRWL, the HOLD items will be served before the LIST and COMP items. Therefore, for all the blocks in the buffer, the LIST, READY and COMP items arrive at a time greater than τ. By the definition of the advice, the items of type HOLD are kept in the buffer of OPT and are present in the buffer of OPT at time τ. So, at time τ, OPT cannot remove any of the items in its buffer before it reads another item with a matching colour from the input sequence. This is a contradiction, as OPT has to remove an item at time τ since its buffer is full.

If the buffer contains HOLD items that were originally of type LIST (LIST in $T_{\text{2BL}}(\sigma)$), then the procedure Postpone was run on the blocks containing these items which reassigned the type from LIST to HOLD. Hence, these blocks are postponed blocks. If at time τ, OPT has already finished outputting the items of any postponed block B in the buffer of 2BRWL, the item of B of type COMP would have been already read from the input and it would be in the buffer of 2BRWL which contradicts the fact that the buffer is full of HOLD items. That means that at time τ, OPT is outputting the items of one of the postponed blocks B, and all the $k - s(B)$ other items from the buffer of 2BRWL are still in the buffer of OPT and have type HOLD. As B is a postponed block, from the definition of $T_{\text{2BRWL}}(\sigma)$, B is postponed after two blocks, B_1 and B_2 of the same class as B are split. Since $s(B), s(B_1)$ and $s(B_2) \in [2^i, \ldots, 2^{i+1})$, the two early blocks are of size greater than $s(B)/2$ each. At time τ, B_1 and B_2 are still in the buffer of OPT while the early blocks are already output by 2BRWL. This is a contradiction as OPT does not have enough buffer space to keep at least $k - s(B) + 2(s(B)/2 + 1) = k + 2$ items in the buffer. □

Lemma 7.17. *At time* $n + 1$*, the buffer of* 2BRWL *contains only blocks of type* COMP*.*

Lemma 7.16 leaves open the possibility that 2BRWL may be able to serve only *part* of an advice block. This could only happen if all the items of the entire block do not enter the buffer while 2BRWL is serving its colour. In such a case, 2BRWL would have to return to this colour several times, and its cost would be higher than the number of advice blocks given by the advice sequence $T_{2\text{BRWL}}(\sigma)$. We will show that this does not happen, and 2BRWL always outputs the complete advice blocks as defined for the entire request sequence σ and $T_{2\text{BRWL}}(\sigma)$. Then, we will bound the competitive ratio of 2BRWL.

First, we present the following technical lemma on the item types contained in an advice block of $T_{2\text{BRWL}}(\sigma)$ that shows that each block is 0 or more HOLD items followed by 0 or more LIST or READY items followed by a COMP item.

Lemma 7.18. *In the advice sequence $T_{2\text{BRWL}}(\sigma)$, the string produced by the concatenation of types of the items in an advice block when ordered from oldest to youngest form an expression in the language defined by the regular expression,*

$$\text{HOLD}^*(\text{LIST}^* \cup \text{READY}^*)\text{COMP} \ .$$

Proof. Initially, in the definition of $T_{2\text{BRWL}}(\sigma)$, $T_{2\text{BRWL}}(\sigma) = T_{2\text{BL}}(\sigma)$. So, from Observation 7.6, initially, all the advice blocks in $T_{2\text{BRWL}}(\sigma)$ are of the form HOLD*LIST*COMP. In the process to generate the final $T_{2\text{BRWL}}(\sigma)$, only the advice blocks of this form are run through the procedures Split and Postpone. Let B be an advice block of the form HOLD*LIST*COMP. Applying the procedure Split(B) generates an early block of the form HOLD*COMP and a late block of the form READY*COMP as, for the early block, the LIST item is changed to COMP and, for the late block, all the LIST items are changed to READY. Applying the procedure Postpone(B) generates an advice block of the form HOLD*READY*COMP as the oldest LIST item is changed to HOLD and the remaining LIST items are changed to READY. □

Now, we consider all the types of advice blocks which 2BRWL outputs before they become complete and show that OPT cannot keep all the items of such advice blocks longer in the buffer. For the various cases where this situation can occur, applying Lemma 7.5 gives us the following result.

Lemma 7.19. *When the algorithm 2BRWL starts outputting an advice block, the entire advice block will be output with no additional colour switches.*

Proof. The only advice blocks which 2BRWL will start outputting before reading all the items of the advice block from the input are: READY blocks and LIST blocks. All the READY blocks are either postponed blocks from the Postpone procedure or late blocks from the Split procedure.

First, consider the postponed blocks of type READY. From the construction of $T_{2\text{BRWL}}(\sigma)$, we know that 2BRWL starts outputting a postponed block B later than OPT. That is, OPT begins outputting B_{OPT} strictly before reading the block item f that follows the first item e of type LIST, but the advice for e gets modified to HOLD in the Postpone procedure and, therefore, 2BRWL only begins outputting B *after* reading f of type READY into the buffer. Also, we know that the oldest item of B_{OPT} is in the buffer of 2BRWL at that time. Hence, from Lemma 7.5, 2BRWL can finish outputting all the items of the block with a single colour switch.

Now, consider the late blocks of type READY. These blocks were produced by the Split procedure. Let τ' be the time when 2BRWL starts outputting a late READY block B. From the definition of the Split procedure, the first READY item of a late block B is the second LIST item of the original advice block. Let τ be the time that OPT starts outputting B_{OPT}. By the definition of the advice, $\tau < \tau'$ and, since the COMP item is not in the buffer at τ', OPT is still outputting the items of B_{OPT} at time τ'. The buffer of 2BRWL at time τ' only contains LIST advice blocks and HOLD advice blocks that contain items that are served after τ' by OPT. The early block that corresponds to B_{OPT} has been output by 2BRWL before τ'. Therefore, $\tau' \geq \tau + i$, where i is the number of items in the early block that corresponds to B_{OPT}. By Lemma 7.5, 2BRWL can output all the items of the block without a colour change at time τ'.

Finally, consider the advice blocks of type LIST. Let τ' be the time when 2BRWL starts outputting an advice block B of type LIST. By the definition of the advice, advice block B was not processed by the procedures Split or Postponed and is, therefore, an advice block as defined for σ, using $T_{2\text{BL}}(\sigma)$. Also, the oldest item of B_{OPT} is in the buffer of 2BRWL at the time that 2BRWL begins outputting B. By showing that OPT starts outputting B_{OPT} at some time $\tau \leq \tau'$, the claim follows from Lemma 7.5.

Assume that OPT starts outputting B_{OPT} at a time $\tau > \tau'$. As B has type LIST at time τ', we get that at time τ' there are no advice blocks of type READY or COMP in the buffer of 2BRWL. All the advice blocks in the buffer of 2BRWL at time τ' are

(1) original advice blocks of type HOLD as defined for σ, using $T_{2\text{BL}}(\sigma)$,

(2) original advice blocks of type LIST as defined for σ, using $T_{2\text{BL}}(\sigma)$,

(3) postponed blocks of type HOLD.

By the definition of the advice, OPT still has the items of the advice blocks of (1) in the buffer at time τ'. The advice blocks of (2) are in the waiting list behind B. This implies that OPT still has the items of the the advice blocks of (2) in its buffer at time τ'. The advice blocks of (3) are postponed blocks and, because of the reassignment of the type in the Postpone procedure, OPT can be in the process of outputting the first of them at time τ'. In this case, OPT would have the items of two early blocks, $s(B_1)$ and $s(B_2)$, of the same class as B in its buffer which have a total size greater than $s(B)$ since $s(B), s(B_1)$ and $s(B_2) \in [2^i, \ldots, 2^{i+1})$. The early blocks have already been output by 2BRWL at time τ'. This is a contradiction as OPT does not have enough buffer space to keep at least $k - s(B) + 2(s(B)/2+1) = k+2$ items in the buffer at time τ'. □

In the remaining part of this section, we will bound the competitive ratio of 2BRWL. For this, we will need a lower bound on the cost of OPT, as well as an upper bound on the cost of 2BRWL. From the construction of the advice sequence, we have that the number of advice blocks defined for σ by $T_{2\text{BRWL}}(\sigma)$ is at least the number of advice blocks defined for σ by $T_{2\text{BL}}(\sigma)$. We get the following two observations.

Observation 7.20. *The cost of* 2BRWL *is at most the number of advice blocks defined with respect to σ and $T_{2\text{BRWL}}(\sigma)$.*

Observation 7.21. *The cost of* OPT *is at least the number of non-late blocks defined with respect to σ and $T_{2\text{BRWL}}(\sigma)$.*

Therefore, to bound the competitive ratio, it is enough to bound the number of late blocks defined with respect to σ and $T_{2\text{BRWL}}(\sigma)$ as compared to the number of advice blocks of other types. Let $\text{LATE}_{2\text{BRWL}}$ denote the set of late blocks created by generating $T_{2\text{BRWL}}(\sigma)$ and $U_{2\text{BRWL}}(\sigma)$.

Lemma 7.22. *Let* POSTPONED *and* EARLY *denote the sets of postponed and early blocks created when constructing $T_{2\text{BRWL}}(\sigma)$. Hence,* $|\text{LATE}_{2\text{BRWL}}| \leq \frac{2}{C}(|\text{POSTPONED}|+|\text{EARLY}|)$.

Proof. Advice blocks from the set $\text{LATE}_{2\text{BRWL}}$ are a result of the Split procedure. The Split procedure is run on 2 blocks out of a set of blocks of size at least C. This procedure produces a total of two early blocks and two late blocks. The remaining advice blocks of which there are at least $C-2$ become postponed blocks. The inequality follows as there are at least C early and postponed blocks for every two late blocks. □

Using Lemma 7.22 above and setting $C = \lceil 2/\varepsilon \rceil$, gives the following theorem.

Theorem 7.23. *For any* $\varepsilon > 0$, 2BRWL *is* $(1+\varepsilon)$*-competitive for the reordering buffer management problem.*

Proof. Let $C = \lceil 2/\varepsilon \rceil$. From Lemma 7.22 and Observation 7.21, we get that

$$|\text{LATE}_{2\text{BRWL}}| \leq \frac{2}{C}\text{OPT}(\sigma) \leq \varepsilon\text{OPT}(\sigma) \ .$$

Therefore, $2\text{BRWL}(\sigma) \leq (1+\varepsilon)\text{OPT}(\sigma)$. □

7.5 A $(1+\varepsilon)$-Competitive Algorithm with $O(\log(1/\varepsilon))$ Bits of Advice per Request

In this section, we modify the 2BRWL algorithm and the advice in such a way so as to be able to encode the position of the LIST blocks in the waiting list, using a constant number of bits of advice per request. This is the main upper bound result of the chapter.

The idea to the final modification is the following. Let B be an advice block with an item of type LIST, and let time τ be the time that the first LIST item of B enters the buffer of the algorithm. It is at time τ that the algorithm needs to know the value u_B and there are at least $s(B)$ items in the buffer at this time. Recall that $s(B)$ for an advice block with an item of type LIST is defined to be the number of HOLD items plus one LIST item which is exactly the number and types of the items in the buffer when the position of B in the waiting list is required. Let D be a constant to be defined later. If $u_B < D^{s(B)}$, then u_B can be encoded in base D with at most $s(B)$ digits. These digits can be sent as advice with each one of the first $s(B)$ items of the advice block B. If $u_B \geq D^{s(B)}$, then u_B cannot be encoded in base D with $s(B)$ digits. So, the advice block B is removed from the waiting list by splitting B into an early and late block, using the Split procedure define previously. The latter case will not happen too often as the number of LIST advice blocks was significantly reduced going from 2BWL to 2BRWL.

7.5.1 The Advice and the Algorithm

The Advice. Given a σ and a fixed lazy OPT, each item $e \in \sigma$ is a assigned a type $t_e \in \{\text{HOLD}, \text{LIST}, \text{READY}, \text{COMP}\}$, where HOLD, LIST, READY and COMP are defined as in Section 7.4.1.

In addition, each item e will have a value $v_e \in [0, \ldots, D-1]$. For an advice block B with an item of type LIST, the first $s(B)$ items of B encode a digit of u_B in base D, ordered from the most significant to the least significant digit, using $\lceil \log D \rceil$ bits. For all other items, v_e is unused and set to 0.

The Algorithm CONSTANT ADVICE WITH WAITING LIST. CONSTANT ADVICE WITH WAITING LIST (CAWL) is defined in essentially the same manner as the algorithm 2BRWL. The only difference is the manner in which CAWL determines the position in the waiting list for the LIST advice blocks. For an advice block B of type LIST, when the first LIST item enters the buffer it is placed in the waiting list. The position in which to insert B is determined by the v_e values of the first $s(B)$ items of B. Each v_e represents a digit of a base D number that is the position of B in the waiting list.

7.5.1.1 Building the Final Advice Sequences

We construct the advice sequence containing the types of the items, $T_{\text{CAWL}}(\sigma)$, and the advice sequence containing the waiting list positions, $U_{\text{CAWL}}(\sigma)$, offline in the following manner. Initially, $T_{\text{CAWL}}(\sigma) = T_{\text{2BRWL}}(\sigma)$, $U_{\text{CAWL}}(\sigma) = U_{\text{2BRWL}}(\sigma)$ and $\tau = 0$. In a simulation of CAWL using the advice sequences $T_{\text{CAWL}}(\sigma)$ and $U_{\text{CAWL}}(\sigma)$, let advice block B be the first new advice block of type LIST after time τ. If $u_B < D^{s(B)}$, then set the values of v_e appropriately for each of the first $s(B)$ items of B. Otherwise, $u_B \geq D^{s(B)}$. In this case, run Split(B). This creates an early block and a late block, and removes B from the waiting list. Set τ to the time step when the first LIST item of B entered the buffer in the current simulation and repeat this procedure, using the updated advice sequences of $T_{\text{CAWL}}(\sigma)$ and $U_{\text{CAWL}}(\sigma)$.

7.5.2 The Analysis

For the analysis, unless explicitly stated, we will always consider the advice blocks as defined with respect to the entire request sequence, σ, and their types, $T_{\text{CAWL}}(\sigma)$. CAWL is defined essentially as 2BRWL. One difference is the encoding of the positions of LIST advice blocks in the waiting list. The other difference is that some of the advice blocks with an item of type LIST as defined for σ by $T_{\text{2BRWL}}(\sigma)$ have been changed to early and late blocks by the Split procedure when defining $T_{\text{CAWL}}(\sigma)$. This means that CAWL and 2BRWL would process σ in the same manner if $T_{\text{CAWL}}(\sigma) = T_{\text{2BRWL}}(\sigma)$ and

$U_{\text{CAWL}}(\sigma) = U_{2\text{BRWL}}(\sigma)$. Therefore, all the lemmas and observations of Section 7.4.2 hold for CAWL as they held for 2BRWL.

As with 2BRWL, in order to bound the competitive ratio, we need to bound the number of late blocks in $T_{2\text{BRWL}}(\sigma)$ as compared to the number of advice blocks of other types. Lemma 7.22 bounds the number of late blocks created by generating $T_{2\text{BRWL}}(\sigma)$ and $U_{2\text{BRWL}}(\sigma)$ which is the initial state of $T_{\text{CAWL}}(\sigma)$ and $U_{\text{CAWL}}(\sigma)$. Let $\text{LATE}_{\text{CAWL}}$ denote the set of late blocks created by generating $T_{\text{CAWL}}(\sigma)$ and $U_{\text{CAWL}}(\sigma)$ that are not in the set $\text{LATE}_{2\text{BRWL}}$.

Lemma 7.24. *Let* LISTED *denotes the set of advice blocks which contain items of type* LIST *as defined with respect to σ and $T_{2\text{BRWL}}(\sigma)$. Hence, $|\text{LATE}_{\text{CAWL}}| \leq \frac{C}{D-1}|\text{LISTED}|$.*

Proof. Let $B' \in \text{LATE}_{\text{CAWL}}$ be a late block produced by applying Split(B) to some LIST advice block B when generating $T_{\text{CAWL}}(\sigma)$ and $U_{\text{CAWL}}(\sigma)$. This only occurs in the case when $u_B \geq D^{s(B)} \geq D^{2^i}$, where i is the class of B.

Let $\mathcal{B}(B)$ be the set of advice blocks in the waiting list at a position before u_B when the first LIST item of B enters the buffer. Therefore, $|\mathcal{B}(B)| = u_B$ and each block from the set $\mathcal{B}(B)$ belongs to the set LISTED.

Let B_0 be an advice block in LISTED, and let τ be the time step when the first item of B_0 was output and B_0 was removed from the waiting list. For any advice block $B' \in \text{LATE}_{\text{CAWL}}$ such that $B_0 \in \mathcal{B}(B)$, the advice block B must have been in the waiting list at the time τ. From the construction of $T_{2\text{BRWL}}(\sigma)$ and $U_{2\text{BRWL}}(\sigma)$, the number of advice blocks of class i which are in the waiting list at time τ is less than C. So, B_0 belongs to less than C different sets of $\mathcal{B}(B)$, where B is in class i.

Now, for each advice block $B' \in \text{LATE}_{\text{CAWL}}$, we charge a cost of $\frac{1}{u_B} \leq \frac{1}{D^{2^i}}$ to each of the u_B advice blocks from the set $\mathcal{B}(B)$. The total cost charged to all advice blocks in LISTED equals $|\text{LATE}_{\text{CAWL}}|$. From the reasoning above, we know that each advice block from the set LISTED belongs to less than C different sets of $\mathcal{B}(B)$ for each class i. For a buffer of size k, there are $\lfloor \log k \rfloor$ classes of blocks. Therefore, each block in LISTED is charged a total cost less than

$$\sum_{i=0}^{\lfloor \log k \rfloor} \frac{C}{D^{2^i}} \leq \frac{C}{D-1} \ .$$

That gives us that $|\text{LISTED}| \cdot \frac{C}{D-1} \geq |\text{LATE}_{\text{CAWL}}|$. ☐

Combining Lemma 7.22 and Lemma 7.24, and setting $C = \lceil 2/\varepsilon \rceil$ and $D = \lceil C/\varepsilon \rceil + 1$ gives us the main upper bound of the chapter.

Theorem 7.25. *For any $\varepsilon > 0$,* CAWL *is $(1+\varepsilon)$-competitive for the reordering buffer management problem, using $O(\log(1/\varepsilon))$ bits of advice per input item.*

Proof. Let $C = \lceil 2/\varepsilon \rceil$ and $D = \lceil C/\varepsilon \rceil + 1$. From Lemma 7.22, Lemma 7.24 and Observation 7.21, we get that

$$|\text{LATE}_{\text{2BRWL}}| + |\text{LATE}_{\text{CAWL}}| \le \left(\max\left\{\frac{2}{C}, \frac{C}{D-1}\right\}\right) \text{OPT}(\sigma) \le \varepsilon \text{OPT}(\sigma) \ .$$

Therefore, $\text{ALG}(\sigma) \le (1+\varepsilon)\text{OPT}(\sigma)$.

The number of advice bits per input item e is 2 bits to encode the item type (t_e) and $\lceil \log D \rceil$ to encode in binary a digit in the base D encoding of the position of a block in the waiting list (v_e). In total, that is $2 + \lceil \log D \rceil = O(\log(1/\varepsilon))$ bits of advice per item. □

7.6 Lower Bound on the Advice Required for a Competitive Ratio of 1

In this section, we show that $\Omega(\log k)$ bits of advice per request are needed for a deterministic algorithm to be 1-competitive. That is, for a deterministic algorithm ALG that uses $o(\log k)$ bits of advice there is no constant α independent of the request sequence σ such that $\text{ALG}(\sigma) \le \text{OPT}(\sigma) + \alpha$.

Initially, we present a family of request sequences, denoted by Ψ, where every $\sigma \in \Psi$ is composed of a sequence of phases. Then, we consider a class of *tidy* algorithms and show that, for Ψ, any online algorithm can be converted to an online tidy algorithm without increasing the cost. Next, we present a subclass of tidy algorithms called *predetermined tidy* algorithms, and show that, for Ψ, we can construct a predetermined tidy algorithm, ALG_{PTA} from any tidy algorithm, ALG_{TIDY}, such that the number of times ALG_{PTA} serves a phase of Ψ optimally is at least as often as ALG_{TIDY}. Finally, we are able to reduce the q-SGKH problem (see Section 2.3.1) to the reordering buffer management problem, using a predetermined tidy algorithm and Ψ. The lower bound follows from this reduction and Theorem 2.11 from [BHK$^+$13] that is a refinement of the lower bound framework first proposed in [EFKR11] as discussed in Section 2.3.1.

Throughout this section, we assume without loss of generality that the algorithm is lazy. Also, the buffer size is assumed to be at least 2.

7.6.1 Ψ Request Sequences

For a buffer of size $k \geq 2$ and an even $r > 0$, we define a family of request sequences $\Psi(r, \pi_1, \pi_3 \ldots, \pi_{r-1})$ such that, for $\sigma \in \Psi$,

$$\begin{aligned}\sigma = \big\langle c_1, c_2, \ldots, c_k, & \pi_1(1), c_{k+1}, \pi_1(2), c_{k+2}, \ldots, \pi_1(k), c_{2k}, \\ & \pi_2(1), c_{2k+1}, \ldots, \pi_2(k), c_{3k}, \\ & \vdots \\ & \pi_{r-1}(1), c_{(r-1)k+1}, \ldots, \pi_{r-1}(k), c_{rk}, \\ & \pi_r(1), \pi_r(2), \ldots, \pi_r(k)\big\rangle \ ,\end{aligned}$$

where $c_1, \ldots, c_{rk}$ are distinct colours. A *new colour* is the first request to a colour c_i in the request sequence. If i is odd, π_i is a random permutation of the k most recently requested new colours. If i is even, π_i is the identity permutation of the k most recently requested new colours. Note that $|\sigma| = 2rk$.

In other words, a request sequence in Ψ, consists of $r+1$ phases. The initial phase, phase 0, consists of k items with k new colours. For the next $r-1$ phases, a phase i, $1 \leq i \leq r-1$, alternates the k new colours from the $(i-1)$-th phase with k new colours. For an odd phase i, the k colours from phase $i-1$ are ordered according to a random permutation. For an even phase i, the k colours from phase $i-1$ are ordered according to the identity permutation. The final phase, phase r, is only one additional request to each of the last k new colours. As shown below, the identity permutation of the even phases from 1 to $r-2$ is used to ensure that, for tidy algorithms (defined below), the state of the buffer after every such even phase consists of the k most recent new colours. This allows us to consider pairs of sequential odd and even phases when constructing the lower bound.

In the following lemma, we show that an optimal algorithm with a full buffer must switch to the colour c_i when the colour of the next request waiting to enter the buffer is c_i, i.e. the items are served in exactly the same order as the permutations, for a cost equal to the number of distinct colours.

Lemma 7.26. *Given $\sigma \in \Psi$, let S_σ be a sequence of colours such that S_σ is, for i from 1 to $r-1$, $\pi_i(1), \ldots, \pi_i(k)$. S_σ is the $(r-1)k$ prefix for all optimal sequences of colour switches.*

Proof. Let OFF be an offline algorithm for σ that performs rk colour switches, denoted by C_σ^{OFF}, such that the first $(r-1)k$ colour switches are those of S_σ and the last k colour switches are $\pi_r(1), \pi_r(2), \ldots, \pi_r(k)$.

The correctness proof for OFF is by induction on j over the indexes of C_σ^{OFF}. Initially, OFF fills its buffer with the first k requests which are k distinct colours. By definition, the request waiting to enter the buffer is $\pi_1(1)$ which is the colour switched to by OFF. OFF serves both requests of colour $\pi_1(1)$ in σ with one colour switch and a new colour enters the buffer. Assume that for every colour switch for $j = 1$ to $u-1$, the two requests of colour $C_\sigma^{\text{OFF}}(j)$ are served. For the inductive step from $u-1$ to u, let σ_u be σ with the pairs of requests with the colours of $C_\sigma^{\text{OFF}}(1, \ldots, u-1)$ removed. The 2 requests of colour $C_\sigma^{\text{OFF}}(u)$ occur in the first $k+1$ requests of σ_u. Therefore, both requests will be served when OFF switches to $C_\sigma^{\text{OFF}}(u)$. Since OFF correctly serves σ and the number of colour switches is equal to the number of distinct colours in the request sequence, then OFF is optimal by Observation 7.2.

Finally, we show by induction on j over the indexes of S_σ, that S_σ is the only $(r-1)k$ prefix for a sequence of colour switches that an optimal algorithm can perform. Note that no algorithm can perform better than the number of distinct colour switches. Therefore, by induction, we will show that any deviation from S_σ results in an additional colour switch which implies more colour switches than OFF. At the point of the first colour switch, the buffer of an algorithm will contain the first k distinct colours and $S_\sigma(1)$ is waiting to enter the buffer. If any other colour besides $S_\sigma(1)$ is switched to at this point, the algorithm will only serve the first request of that colour resulting in an additional colour switch to serve the second request of the same colour. Therefore, an optimal algorithm must switch to $S_\sigma(1)$ initially. Assume that this is the case for j from 1 to $u-1$. For the inductive step from $u-1$ to u, let σ_u be σ with the pairs of requests with colours of $S_\sigma(1, \ldots, u-1)$ removed. The first k requests of σ_u will be the first request to k distinct colours including $S_\sigma(u)$ and the $(k+1)$-th request will have the colour $S_\sigma(u)$. As in the base case, an optimal algorithm must switch to $S_\sigma(u)$ at this point. □

7.6.2 Tidy Algorithm

In this section, we consider a class of algorithms called *tidy* algorithms. Informally, a tidy algorithm will always switch to a colour that has the second item of that colour in the buffer before other colours. For Ψ, that means switching to a colour when the second request of that colour is sure to be served. Tidy algorithms are formally defined

in the following.

Definition 7.27 (A tidy algorithm)**.** *For $\sigma \in \Psi$, when performing a colour switch, a* tidy *algorithm will switch (in order of priority) to a colour c in the buffer such that:*

1. *the last request to c is in the buffer;*
2. *$c = \pi_i(j)$, where $\pi_i(j)$ is the first request outside the buffer and i is even;*
3. *c is any colour in the buffer.*

The following lemma shows that we can, for $\sigma \in \Psi$, without loss of generality, restrict our attention to *tidy* online algorithms.

Lemma 7.28. *Any online algorithm A for $\sigma \in \Psi$ can be converted to an online tidy algorithm $\hat{A}$ such that, for all $\sigma \in \Psi$, $\hat{A}(\sigma) \leq A(\sigma)$.*

Proof. Let C_σ^A be the sequence of colour switches performed by A on σ, and let u be the first index in C_σ^A that violates the colour switching rules of a tidy algorithm, and let c be the colour that a tidy algorithm would switch to at u.

We define $\hat{A}$ to be another algorithm that simulates A in an online manner. The algorithm $\hat{A}$ maintains a pointer p, which is initially 1, in the sequence C_σ^A which is generated online as $\hat{A}$ simulates A on the request received. For the first $u-1$ colour switches, $\hat{A}$ performs the same colour switches as A and updates p each time. Since A does not violate the tidy properties until u, $\hat{A}$ is tidy until this point. Note that the buffers of $\hat{A}$ and A are in the same state at this point and that $C_\sigma^{\hat{A}}(1,\ldots,u-1) = C_\sigma^A(1,\ldots,u-1)$. For the u-th colour switch, $\hat{A}$ will switch to colour c. For the $u+1$ colour switch, $\hat{A}$ will perform the $C_\sigma^A(u)$ colour switch and update p to $u+1$. From the $(u+1)$-th colour switch of A and $\hat{A}$, $\hat{A}$ is at least as far into the input as A. From this point on, $\hat{A}$ performs the same colour switches as A, dropping any colour switches to colours that are no longer in the buffer. The droped switches correspond to colours that are served more efficiently by $\hat{A}$ than A. In total, $\hat{A}$ performs at most as many colour switches as A. As defined, $\hat{A}$ may not be tidy, however this process can be repeated on $\hat{A}$ until a tidy algorithm is produced.

More formally, we will define a sequence of algorithms such that $A_0 = A$ and $A_q = \hat{A}$, where $\hat{A}$ is an online tidy algorithm and A_i is define in an online manner based on A_{i-1}. Let $u_{A_{i-1}}$ denote the first index of the colour switches of an algorithm A_{i-1}, for $0 < i \leq q$, that violates a tidy rule. If $u_{A_{i-1}}$ is undefined, A_{i-1} is an online tidy algorithm and the process is done. Let c_{A_i} be the colour that would not violate the tidy rule at $u_{A_{i-1}}$. Define algorithm A_i as $\hat{A}_{i-1}$. That is, at $u_{A_{i-1}}$, the previous

$u_{A_{i-1}} - 1$ colour switches were served exactly as the online simulation of A_{i-1} and A_i switches to c_{A_i}. If u_{A_i} exists, the process is repeated to define A_{i+1}. Note that this process is online over a single pass over σ since the colour switches $C_\sigma^{A_i}(1, \ldots, u_{A_i} - 1) = C_\sigma^{A_j}(1, \ldots, u_{A_i} - 1)$ for all A_j, $j \geq i$. In addition, u_{A_i} can be determined online by simulating A_i and using the simulated colour switches until the simulated colour switch would violate a tidy property. As shown previously, $A_i(\sigma) \leq A_{i-1}(\sigma)$. Therefore, the claim follows since $A(\sigma) = A_0(\sigma) \geq \cdots \geq A_q(\sigma) = \hat{A}(\sigma)$. □

Since we can assume without loss of generality that any online algorithm processing $\sigma \in \Psi$ is tidy, the following lemma implies that the buffer of such an algorithm contains the k most recently requested new colours after serving the last request of an even permutation.

Lemma 7.29. *For $\sigma \in \Psi$, after serving $\pi_i(k)$, where i is even, the buffer of a tidy algorithm contains $c_{ik+1}, \ldots, c_{ik+k}$.*

Proof. Assume for the sake of contradiction, that, after serving $\pi_i(k)$, there exists a colour c in the buffer such that $c \notin \{c_{ik+1}, \ldots, c_{ik+k}\}$. If $c \in \{c_1, \ldots, c_{ik}\}$, it would imply that the algorithm served the first request of $\{c_{ik+1}, \ldots, c_{ik+k}\}$ over the last request of c which violates the first rule of a tidy algorithm, contradicting the fact that the algorithm is tidy. If $c \in \{c_{ik+k+1}, \ldots, c_{rk}\}$ for $r > i$, it would imply that the algorithm had either served a colour in $\{c_{ik+1}, \ldots, c_{ik+k}\}$ over the next item in π_i which violates the second rule of a tidy algorithm or served a colour in $\{c_{ik+1}, \ldots, c_{ik+k}\}$ over the last request of an item in π_i which violates the first rule of a tidy algorithm. Either case is a contradiction to the fact that the algorithm is tidy. □

7.6.3 A Predetermined Tidy Algorithm

A *predetermined tidy* algorithm is a tidy algorithm that, prior to bringing in the first request of some permutation $\pi_i \in \sigma \in \Psi$, i odd, into its buffer, will decided on a fixed order, π_i^*, to serve the colours of π_i. That is, setting a specific order for the third tidy rule. The formal definition of a predetermined tidy algorithm is the following.

Definition 7.30 (A predetermined tidy algorithm)**.** *For $\sigma \in \Psi$, prior to a colour switch that will bring $\pi_i(1)$, i odd, into the buffer, a* predetermined tidy *algorithm will fix a π_i^*. When performing a colour switch, a* predetermined tidy *algorithm will switch (in order of priority) to a colour c in the buffer such that:*

1. *the last request to c is in the buffer;*

2. *$c = \pi_i(j)$, where $\pi_i(j)$ is the first request outside the buffer and i is even;*

3. *$c = \pi_i^*(\ell)$, where $\pi_i^*(j)$ is in the buffer and, for all $j < \ell$, $\pi_i^*(j)$ is not in the buffer.*

In the following, for an arbitrary $\sigma \in \Psi$, we are going bound the number of times a predetermined tidy algorithm ALG_{PTA} perfectly serves all the colours of a permutation as compared to a given tidy algorithm ALG_{TIDY}. For $\sigma \in \Psi$, let $m_{\text{ALG}}(\sigma)$ be the number of pairs of permutation π_i and π_{i+1}, for i odd, such that some tidy algorithm, ALG_{TIDY}, performs more than $2k$ colour switches to serve the colours of π_i and π_{i+1}. Given the definition of m_{ALG}, we get the following fact.

Fact 7.31. *For $\sigma \in \Psi$ and any ALG_{TIDY}, $\text{ALG}_{\text{TIDY}}(\sigma) \geq 2kr + m_{\text{ALG}_{\text{TIDY}}}(\sigma)$.*

In the following lemma, we define a predetermined tidy algorithm, ALG_{PTA}, using b bits of advice per request, that uses an arbitrary tidy algorithm, ALG_{TIDY}, with b bits of advice per request, as a black box to determine π_i^*, where i is odd.[1] We will show that ALG_{PTA} serves π_i and π_{i+1} optimally (with $2k$ colour switches) when ALG_{TIDY} serves them optimally.

Lemma 7.32. *From any online tidy algorithm with b bits of advice per request for the reordering buffer management problem, ALG_{TIDY}, that receives all the advice in advance, a predetermined online tidy algorithm with b bits of advice per request, ALG_{PTA}, can be constructed that receives all the advice in advance, such that, for all $\sigma \in \Psi$, $m_{\text{PTA}}(\sigma) \leq m_{\text{TIDY}}(\sigma)$.*

Proof. We will define a predetermined tidy algorithm, ALG_{PTA}, using b bits of advice per request that are received in advance, that uses any tidy algorithm, ALG_{TIDY}, using b bits of advice per request that are received in advance.

For each request in σ, the b bits of advice received by ALG_{PTA} are defined to be the bits of advice required by ALG_{TIDY}. This is feasible since both algorithms use b bits advice per request.

Intuitively, to determine every π_i^* (the order in which ALG_{PTA} attempts to serve π_i), for i odd, ALG_{PTA} will provide ALG_{TIDY} a permutation of the k most recent colours based on the advice received in advance and the requests already processed. The permutation

[1] It should be noted that we assume that ALG_{PTA} receives all the advice in advance. This strengthens the algorithm and, in the following, for the lower bound, we make the same assumption. This only makes our result stronger.

π_i^* will be determined by the colour switches of ALG_{TIDY}. The idea is to provide ALG_{TIDY} the only input instance for which it could serve the colours of π_i and π_{i+1} with $2k$ colour switches.

More formally, for i odd, to determine every π_i^*, ALG_{PTA} defines a σ_i' to be given as input to a simulation of ALG_{TIDY} with the advice received by ALG_{PTA} and an input request sequence σ_i' defined in the following manner. The prefix of σ_i' is $\sigma(1), \ldots, \sigma(2k(i-1)+k)$, i.e. the request seen by ALG_{PTA} to this point. Since ALG_{TIDY} is tidy, by Lemma 7.29, immediately after bringing $\sigma(2k(i-1)+k)$ into the buffer, the buffer of ALG_{TIDY} contains the k most recently requested new colours: $C = \{c_{2k(i-1)+1} \ldots, c_{2ik-k}\}$. For the next $1 \leq j \leq k$ subsequent colour switches of ALG_{TIDY}, if a colour switch c_j is in C, we append $c_j, c_{2ik-k+1}$ to σ_i', where $c_{2ik-k+1}$ is a new colour, set $\pi_i^*(j) = c_j$ and have ALG_{TIDY} continue serving σ_i'. If c_j is not in C or ALG_{TIDY} is unable to make a colour switch (e.g. due to an inconsistency between σ' and the advice), then the simulation stops. In this case, the remaining items of π_i^* are defined in an arbitrary manner.

To show that $m_{\text{PTA}}(\sigma) \leq m_{\text{TIDY}}(\sigma)$, we need to show that every time ALG_{TIDY} serves π_i, π_{i+1}, i odd, with $2k$ colour switches so does ALG_{PTA}. By Lemma 7.26, if ALG_{TIDY} serves π_i, π_{i+1} in σ with $2k$ colour switches, then the first k colour switches to serve the colours in $\pi_i \cup \pi_{i+1}$ must be π_i. Let π_i' be first k colour switches of ALG_{TIDY} to serve the colours in $\pi_i \cup \pi_{i+1}$. If $\pi_i' = \pi_i$, it follows from the construction of π_i^* above that $\pi_i^* = \pi_i$ and, therefore, ALG_{PTA} will serve the colours of π_i, π_{i+1} with $2k$ colour switches. □

7.6.4 Reduction from the q-SGKH Problem

For the lower bound, we will assume that the algorithm receives all the advice in advance and use the lower bound framework as discussed in Section 2.3.1. That is, we can reduce the q-SGKH problem to the reordering buffer management problem. Further, the σ_{RBM} produced by the reduction is in the set Ψ, as defined above, which, along with the previous lemmas, allows us to assume that the reordering buffer management algorithm is a tidy algorithm and use the ALG_{PTA} defined in Lemma 7.32.

Lemma 7.33. *Suppose that there exists a ρ-competitive algorithm with b bits of advice per request for the reordering buffer management problem with buffer of size k. Then, there exists an algorithm with $5kb$ bits of advice per request for the q-SGKH problem that is correct for more than $(1-(\rho-1)2k)n$ characters of the n length string, where $q = k!$, $k \geq 2$ and $1 \leq \rho \leq 1 + \frac{1}{2k}$.*

Proof. Let ALG_{RBM} be a ρ-competitive algorithm with b bits of advice per request for

the reordering buffer management problem. We will design an algorithm $\text{ALG}_{q\text{-SGKH}}$ that will generate a request sequence, σ_{RBM}, that can be processed by ALG_{RBM}.

The first k request of σ_{RBM} are k different colours. The remaining requests of σ_{RBM} are defined, as follows, in an online manner such that it belongs to the set Ψ. Let Π be an enumeration of all the possible permutations of length k and let $g : \Sigma \to \{1, \ldots, k!\}$ be a bijection between Σ, the alphabet of the q-SGKH problem, and an index of a k length permutation in Π. Let $c_{(2i-1)k+1}, \ldots, c_{2ik+k}$ be $2k$ arbitrary new colours. After receiving request $\sigma_{q\text{-SGKH}}(i)$,

$$\left\langle \pi_{2i-1}(1), c_{(2i-1)k+1}, \ldots, \pi_{2i-1}(k), c_{2ik}, \pi_{2i}(1), c_{2ik+1}, \ldots, \pi_{2i}(k), c_{2ik+k} \right\rangle$$

is appended to σ_{RBM}, where π_{2i-1} is the permutation at $\Pi(g(\sigma_{q\text{-SGKH}}(i)))$, π_{2i} is the identity permutation, and both permutations are a permutation of the newest k colours immediately before the first request of each permutation.

For $\sigma_{q\text{-SGKH}}(i)$, the actions of $\text{ALG}_{q\text{-SGKH}}$ are based on the permutation π^*_{2i-1} determined by ALG_{PTA}, as defined in Lemma 7.32, based on ALG_{RBM}. Prior to the i-th guess, ALG_{PTA} has determined π^*_{2i-1} based on the requests seen so far, i.e. $\sigma_{\text{RBM}}(1), \ldots, \sigma_{\text{RBM}}(4ik-4k)$ as generated from $\sigma_{q\text{-SGKH}}(1), \ldots, \sigma_{q\text{-SGKH}}(i-1)$. So, the i-th guess of $\text{ALG}_{q\text{-SGKH}}$ is the inverse function of g on the index of π^*_{2i-1} in Π.

For each request in $\sigma_{q\text{-SGKH}}$, at most $5k$ requests are generated for σ_{RBM} which for ALG_{RBM} requires at most $5kb$ bits of advice. This is feasible since $\text{ALG}_{q\text{-SGKH}}$ has $5kb$ bits of advice per request. The advice provided to ALG_{RBM} is from the advice provided to $\text{ALG}_{q\text{-SGKH}}$.

From the definitions of σ_{RBM} and $\text{ALG}_{q\text{-SGKH}}$, we get that

$$\begin{aligned}
\#\text{ of misses} &= m_{\text{PTA}}(\sigma_{\text{RBM}}) \\
&\leq m_{\text{RBM}}(\sigma_{\text{RBM}}) \text{ , from Lemma 7.32,} \\
&\leq \text{ALG}_{\text{RBM}}(\sigma_{\text{RBM}}) - \text{OPT}_{\text{RBM}}(\sigma_{\text{RBM}}) \text{ , from Fact 7.31,} \\
&\leq (\rho - 1)\text{OPT}_{\text{RBM}}(\sigma_{\text{RBM}}) \text{ , as ALG}_{\text{RBM}} \text{ is } \rho\text{-competitive,} \\
&= (\rho - 1)2kn \text{ , since OPT}_{\text{RBM}} = 2kn.
\end{aligned}$$

□

We are now ready to prove the main lower bound theorem.

Theorem 7.34. *A deterministic online algorithm with advice for the reordering buffer management problem with buffer of size $k \geq 2$ requires at least $\frac{1}{10} \cdot ((1 - H_{k!}(1-\alpha)) \log_2 k)$ bits of advice per request to be ρ-competitive for $1 < \rho \leq 1 + \frac{1}{2k} - \frac{1}{2k \cdot k!}$, where H_q is the q-ary entropy function and $\alpha = 1 - (\rho - 1)2k$.*

Proof. For $1 < \rho \leq 1 + \frac{1}{2k} - \frac{1}{2k \cdot k!}$, let ALG_{RBM} be a ρ-competitive deterministic algorithm for the reordering buffer management problem with b bits of advice per request. Define $q = k!$. By Lemma 7.33, there exists an algorithm for the q-SGKH problem with $5kb$ bits of advice per request that is correct for more than $(1 - (\rho - 1)2k)n$ characters of the n length string. The bounds on ρ and k imply that $q \geq 2$ and $1/q \leq \alpha < 1$. Theorem 2.11 implies that $5kb \geq (1 - H_q(1 - \alpha)) \log q$ and, therefore,

$$\begin{aligned} b &\geq \frac{(1 - H_q(1 - \alpha))}{5k} \log q \\ &\geq \frac{(1 - H_{k!}(1 - \alpha))}{10} \log_2 k \text{ , as } q = k! \geq k^{k/2}. \end{aligned}$$

□

From Theorem 7.34, we get the following corollary.

Corollary 7.35. *A deterministic online algorithm with advice requires $\Omega(\log k)$ bits of advice per request to be 1-competitive for the reordering buffer management problem with buffer of size $k \geq 2$.*

Proof. Let ALG be a ρ-competitive deterministic online algorithm, using b bits of advice per request, for the reordering buffer management problem with buffer of size $k \geq 2$, where $1 < \rho \leq 1 + \frac{1}{2k} - \frac{1}{2k \cdot k!}$. Let $\alpha = 1 - (\rho - 1)2k$. From Theorem 7.34, we have

$$b \geq \frac{(1 - H_{k!}(1 - \alpha))}{10} \log_2 k \ .$$

Note that $0 < 1 - \alpha \leq 1 - 1/k!$. Since $H_{k!}(x) \leq H_2(x)$ for all $k! \geq 2$ and $x \in (0, 1 - 1/k!]$,

$$b \geq \frac{(1 - H_2(1 - \alpha))}{10} \log_2 k \tag{7.1}$$

As approaches ρ approaches 1 from above, α approaches 1 from below and $H_2(1 - \alpha)$ approaches 0. Hence, (7.1) approaches $\frac{\log_2 k}{10}$ as ρ approaches 1. □

8 The List Update Problem

In this chapter, we consider the list update problem (see Section 8.1 for a formal definition). The main result in this section is to show that there is a multiplicative gap of at least 13/12 between the cost of an offline optimal algorithm restricted to free exchanges and an unrestricted offline optimal algorithm that can use both paid and free exchanges. This question has implicitly been open since the work of Reingold and Westbrook [RW96] showed that the gap was at least an additive constant with a counter-example to the claim that there was no gap of Sleator and Tarjan in [ST85].

8.1 Preliminaries

The *list update problem* consists of a linked list of ℓ items and a finite request sequence of accesses. Each request is to access an item of the list. Each item access begins at the head of the list and there is a cost of 1 to the algorithm for each item accessed until the requested item is found. This is, the cost to access the i-th item in the list is i. Then, the requested item can be moved forward in the list at no cost and such a move is called a *free exchange*. At any time, two adjacent items may be swapped at a cost of 1 and the swaps are called *paid exchanges*. The goal is to dynamically rearrange the list over the request sequence so as to minimize the total cost of accesses and paid exchanges over the request sequence.

We will use OPT to denote an unrestricted optimal offline algorithm, and we will use OPT_FREE to denote an optimal offline algorithm restricted to using only free exchanges.

For the request sequences, we will denote multiple requests in a row to the same item by using exponents, e.g. x^k would mean that x is requested k times in a row.

In [RW96], Reingold and Westbrook consider the offline version of the list update problem and show several properties of an offline optimum using both paid and free exchanges such as the following corollary.

Corollary 8.1. [RW96] *If an item x is requested 3 or more times consecutively, then an optimal algorithm must move it to the front before the second access.*

Also, in [RW96], Reingold and Westbrook define the notion of a subset transfer and show that there exists an optimal algorithm that only performs such moves.

Definition 8.2 (Subset Transfer)**.** *Let x be a requested item. A* subset transfer *is a move of a subset of the items ahead of x in the list to the position immediately after x such that the relative order of the items in the subset is maintained.*

Theorem 8.3. [RW96] *There is an optimal algorithm that does only subset transfers.*

Using Corollary 8.1 and Theorem 8.3, we get the following theorem.

Theorem 8.4. *Let $\sigma = \left\langle x_1^{k_1}, \ldots, x_j^{k_j} \right\rangle$, where, for all i, $k_i \geq 3$ and, for $i < j$, $x_i \neq x_{i+1}$. For any initial list configuration, every* OPT_FREE *moves each x_i, $1 \leq i \leq j$, to the front of the list immediately after the first access to x_i of $x_i^{k_i}$ in σ.*

Proof. For σ, by Corollary 8.1, an optimal algorithm must move x_i, $i = 1$ to j, to the front before the second request to the item. Let OPT be an optimal algorithm that only performs subset transfers. By Theorem 8.3, such an optimal algorithm exists. Observe that, after moving x_i to the front by a subset transfer immediately before the first access to x_i, OPT performs no other moves over the remaining consecutive requests to x_i by the fact that, for the remaining requests, x_i is at the front of the list and the property that OPT only performs subset transfers.

The action by OPT of moving x_i to the front immediately before the first request to x_i can be accomplished for the same cost by either accessing x_i and then moving it to the front, or moving x_i to the front with paid exchanges and then accessing x_i. Therefore, there exists an optimal algorithm that moves x_i to the front immediately after the first access of x_i with a free exchange. □

Informally, the next theorem shows that, for an arbitrary algorithm that only performs free moves, denoted by ALG_FREE, and, for a sequence of consecutive requests

to an item x, such that x is moved forward to the position β during these consecutive requests, if ALG_FREE would move x to β immediately after the first request, it would reduce its cost. This holds for both offline and online algorithms, but online algorithms are not able take advantage of this fact given that they do not in general know the subsequent requests.

Theorem 8.5. *Let $\sigma = \langle \sigma_1, \nu, \sigma_2 \rangle$, where ν is at least two consecutive requests to the same item x. Let β be the position of x immediately after ν for an arbitrary algorithm* ALG_FREE*. There exists an algorithm* ALG_FREE$'$ *that moves x to β immediately after the first request of ν such that* ALG_FREE$'(\sigma) \leq$ ALG_FREE(σ)*, and* ALG_FREE$'$ *serves σ_1 and σ_2 exactly as* ALG_FREE*.*

Proof. The algorithm ALG_FREE$'$ is defined to serve σ_1 in the same manner as ALG_FREE, move x to position β immediately after the first request of ν and to serve σ_2 in the same manner as ALG_FREE. Note that the list configurations of ALG_FREE$'$ and ALG_FREE match prior to and after serving ν. Therefore, the cost to both algorithms is the same for σ_1 and σ_2.

Let $\alpha \geq \beta$ be the position that ALG_FREE moves x immediately after the first request of ν. Then, there is at least one request of ν that has an access cost of α for ALG_FREE. The same request has an access cost of β for ALG_FREE$'$. Therefore, ALG_FREE$'(\sigma) \leq$ ALG_FREE(σ) □

8.2 Notes on Upper Bounds

In this section, we consider some natural algorithms with advice and provide some upper bounds on the competitive ratio of these algorithm via known upper bounds for randomized algorithms (without advice) whose performance is not better than the algorithms with advice. Initially, we consider algorithms with advice that only perform free exchanges and then consider an algorithm with advice, using paid exchanges.

In [Alb95], Albers describes the TIMESTAMP family of algorithms. The parameterized version of TIMESTAMP is denoted by TIMESTAMP$_p$ and works as follows. For a request r_j in σ to an item x, let r_i be the previous request (if it exists) to x. Let y be the item nearest to the front that was either not requested between r_i and r_j, or that was requested once between r_i and r_j and that request was processed by the procedure Stamp (defined below). With probability p, TIMESTAMP$_p$ moves x to the front and,

with probability $1-p$, TIMESTAMP$_p$ runs procedure Stamp(r_j).

Procedure Stamp(r_j)

- If r_j is the first request to x, do not move x.
- Otherwise, x has been requested previously and y is well defined. Move x to the position immediately in front of y.

The algorithm TIMESTAMP without the parameter p is equivalent to $p = 0$ and is deterministic, i.e. TIMESTAMP = TIMESTAMP$_0$. The optimal choice of $p = \frac{3-\sqrt{5}}{2}$ gives a Φ-competitive algorithm, where $\Phi = \frac{1-\sqrt{5}}{2} > 1.618$ is the golden ratio. Also, this algorithm can be implemented as barely random algorithm where the items are fixed to the two different actions with a probability p and $1-p$ respectively, using ℓ random bits, before serving the sequence [Alb95]. Therefore, implicit in the definition of this algorithm is a Φ-competitive algorithm that uses 1 bit of advice per request or ℓ bits of advice in total. These algorithms simply make the choices according to the advice bits.

For the problem with advice, there is a very natural family of online algorithms to consider. We denote this family of algorithms as MF$_{\mathcal{P}}$ which stands for "move forward to a position in the set $\mathcal{P}$". For a set of positions $\mathcal{P}$, at most $\lceil \log |\mathcal{P}| \rceil$ bits of advice per request are needed to encode the index of a position in an enumeration of $\mathcal{P}$. The advice is based on the algorithm that will optimally serve the request sequence when limited to moving forward to one of the positions in $\mathcal{P}$ after each request. Some examples of algorithms in the family MF$_{\mathcal{P}}$, are

MF$_{\{1,r\}}$: For request r, 1 bit of advice indicates whether to move the item to the front of the list (position 1) or not (position r).

MF$_{\text{GEO}}$: GEO is the set of positions of the head of sub-lists in a geometric partitioning of the positions of the list into disjoint, contiguous sub-lists of increasing size of 2^i for $i \in [0, \ldots, \lceil \log \ell \rceil)$, where the first sub-list begins at position 1. These positions can be encoded with $\lceil \log \lceil \log \ell \rceil \rceil$ bits of advice.

MF$_{[\ell]}$: $\lceil \log \ell \rceil$ bits of advice indicate the position in the list to which the accessed item is moved.

Note that the performance of the $\text{MF}_{[\ell]}$ algorithm is equivalent to the optimal offline algorithm restricted to free moves. Further, the lower bound for OPT_FREE presented in Section 8.3 is a lower bound for the family of MF_x algorithms.

Let $\text{MF}_{\supseteq\{1,r\}}$ be all of the $\text{MF}_{\mathcal{P}}$ algorithms that contain, as a subset of possible forward moves, the option to move the item to the front or to not move the item. The COUNTER algorithm, a randomized algorithm of Reingold et al. [RWS94] that is a generalization of the BIT algorithm that moves items to the front in a barely random fashion, has an upper bound of $\sqrt{3}$ which holds for all $\text{MF}_{\supseteq\{1,r\}}$. For OPT_FREE and $\text{MF}_{[\ell]}$, the algorithm COMB of Albers et al. [AvSW95] which randomly uses the BIT algorithm with a probability of 4/5 and the non-parameterized TIMESTAMP algorithm with a probability of 1/5 implies an upper bound of 1.6. This is the currently the best known upper bound for randomized algorithms, OPT_FREE and $\text{MF}_{[\ell]}$.

If we allow the algorithm with advice to use paid exchanges, at most $\ell - 1$ bits of advice per request or $\sum_{i=1}^{n}(r_i - 1)$ bits of advice in total are required for an optimal algorithm. This follows immediately from the subset transfer theorem of Reingold and Westbrook of [RW96] (see Definition 8.2 and Theorem 8.3).

8.3 Lower Bound for OPT_FREE

In this section, we present a lower bound for the free move optimal offline algorithm as compared to the unrestricted optimal offline algorithm. That is, we are comparing the power of paid exchanges and free exchanges versus only free exchanges.[1] We show that, for the case of a list of length at least 3, the performance ratio between OPT_FREE and OPT is at least $13/12 > 1.083$ in the worst case. More formally, there exists an infinite family of finite request sequences σ_r, $r > 0$, where the cost of an offline algorithm that uses paid exchanges, OFF, increases with r, such that $\frac{\text{OPT_FREE}(\sigma_r)}{\text{OPT}(\sigma_r)} \geq \frac{\text{OPT_FREE}(\sigma_r)}{\text{OFF}(\sigma_r)} \geq 13/12$. This implies that, for any $\varepsilon > 0$ and any additive constant η that does not depend on the length of the request sequence, there does not exist a free move algorithm, ALG_FREE, such that $\text{ALG_FREE}(\sigma) \leq \left(\frac{13}{12} - \varepsilon\right)\text{OPT}(\sigma) + \eta$ for all σ.

To prove the claim, we begin by defining a request sequence $R(L)$. For a given initial list configuration L, we define $R(L)$ and a deterministic offline algorithm OFF

[1] In [RW96], Reingold and Westbrook show that free exchanges can be replaced by paid exchanges without increasing the cost. So, an optimal offline algorithm restricted to only paid exchanges is not weaker than an optimal offline algorithm that can use both paid and free exchanges.

that uses paid exchanges. By relabelling the list of OFF after having served $R(L)$ to match L, we can define an arbitrarily long request sequence σ consisting of repeated requests to $R(L)$ based on a relabelling of the list state of OFF after each $R(L)$. If the list is of length more than 3, we ignore all but 3 items. In what follows, without loss of generality, we consider a list of length 3.

For a list of length 3 with a starting list configuration $L = y, x_1, x_2$, we define the request sequence $R(L) = \langle x_2, x_1, x_1^3, x_2^3 \rangle$. (Note that x_1 is requested 4 times in a row.)

Offline Paid Exchange Algorithm. Let OFF be an unrestricted offline algorithm defined as follows. Before the first request of $R(L)$, using two paid exchanges, x_1, x_2 are moved to the front of the list. Then, immediately before the second request to any x_i, $1 \leq i \leq 2$, OFF moves x_i to the front.

Immediately from the definition of OFF, we have the following fact.

Fact 8.6. *Given a starting list configuration of $L = y, x_1, x_2$, after serving $R(L)$, the list configuration of* OFF *is x_2, x_1, y.*

The following lemma shows the cost of OFF for $R(L)$, assuming the starting list configuration is L.

Lemma 8.7. *Given a starting list configuration of $L = y, x_1, x_2$, the cost of* OFF *to serve $R(L)$ is* 12

Proof. The cost to bring x_1, x_2 to the front by paid exchanges is 2 and the list configuration is now x_1, x_2, y. The cost of the first access to x_2 is 2, the cost to the next four requests of x_1 is 4. The second access to x_2 costs 2 and then x_i is brought to the front and the remaining two accesses cost 2.

Overall, the cost to OFF is 12. □

Arbitrarily Long Request Sequence. For an initial list configuration of $L = y, x_1, x_2$, from Fact 8.6, the configuration of the list of OFF after serving $R(L)$ is x_2, x_1, y. Therefore, after serving $R(L)$, with a relabelling of the list of OFF to that of L, $R(L)$ can subsequently be requested again, and this can be repeated to create arbitrarily long request sequences.

Let $\sigma_r = \langle R_1(L_1), R_2(L_2), \ldots, R_r(L_r) \rangle$ such that $R_j(L_j)$ is based on L_j, where L_j is the configuration of the list of OFF after serving $R_1, \ldots, R_{j-1}$ for $1 < j \leq r$ and $L_1 = L$

is the initial configuration of the list. We will use the term round to signify an $R(L)$ in σ_r.

Offline Free Move Algorithm. Let $L = y, x_1, x_2$ be the initial list configuration. Define MFF_1 to be an algorithm for the request sequence $R(L)$ as follows. MFF_1 moves x_1 to the front of the list on the first request to the item, and moves x_2 to the front of the list only on the second request to the item.

Immediately from the definition of MFF_1, we have the following fact about the configuration of the list after MFF_1 serves $R(L)$.

Fact 8.8. *Given a starting list configuration of $L = y, x_1, x_2$, after serving $R(L)$, the list configuration of* MFF_1 *is x_2, x_1, y.*

Note that, when starting from the same initial list configuration and serving $R(L)$, the list configuration of MFF_1 is exactly that of OFF after serving $R(L)$ which implies the following observation.

Observation 8.9. MFF_1 *is well defined for σ_r.*

The Last 4 Requests of σ. In the following lemma, we show that any OPT_FREE moves any item x to the front of the list immediately after the first access of three consecutive requests to x in $R_r(L_r)$ of σ_r, i.e. the last round of σ_r.

Lemma 8.10. *For $\sigma_r = \langle R_1(L_1), \ldots, R_r(L_r)\rangle$, every* OPT_FREE *moves any item x to the front of the list immediately after the first access of three consecutive requests to x in $R_r(L_r)$, where $L_1 = y, x_1, x_2$ and L_j, $1 < j \leq r$, is the list configuration of* OFF *after serving $\langle R_1(L_1), \ldots, R_{j-1}(L_{j-1})\rangle$.*

Proof. Let $L_r = y, x_1, x_2$ and let A be an arbitrary OPT_FREE algorithm. For the sake of contradiction, assume that A does not move some $x_i \in R_r(L_r)$ to the front immediately after the first request to $x_i^3 \in R_r(L_r)$.

Let $\sigma' = \langle R_1(L_1), \ldots, R_{r-1}(L_{r-1}), x_2\rangle$ and $\sigma'' = \left\langle x_1^4, x_2^3 \right\rangle$. Define $\hat{A}$ to be a free move algorithm that serves σ' exactly as A and moves all $x_j \in \sigma''$ to the front immediately after the first request to each item in σ''.

Since A and $\hat{A}$ serve σ' in the same manner, $\hat{A}(\sigma') = A(\sigma')$ and the list configurations of A and $\hat{A}$ are the same immediately after σ'. From Theorem 8.4, given the list configuration of both A and $\hat{A}$ after serving σ', $\hat{A}$ is optimal over the remainder of the sequence and A is not. That is, for the list configuration of both A and $\hat{A}$ after serving

σ', $\hat{A}(\sigma'') = \textsc{opt}(\sigma'') < A(\sigma'')$. Therefore, $\hat{A}(\sigma_r) = A(\sigma') + \textsc{opt}(\sigma'') < A(\sigma_r)$ which contradicts the fact that A is an optimal free move algorithm. □

For an initial configuration $L = y, x_1, x_2$ and $R(L) = \langle x_2, x_1, x_1^3, x_2^3 \rangle$, the following lemma shows that $\textsc{mff}_1$ is an optimal free move algorithm for $R(L)$.

Lemma 8.11. *For an initial list configuration* $L = y, x_1, x_2$, $\textsc{mff}_1(R(L)) = 13$ *and* $\textsc{opt_free}(R(L)) = 13$.

Proof. Irrespective of the free move algorithm, the access cost for the first request is 3 and there are 3 possible list configurations after the access. They are y, x_1, x_2 (this corresponds to $\textsc{mff}_1$); y, x_2, x_1; and x_2, y, x_1. By Theorem 8.4, after serving the first request, every OPT_FREE moves x_1 and x_2 to the front of the list on the next request to each item. Table 8.1, summarizes the costs of the 3 possible ways to serve $R(L)$. The actions of $\textsc{mff}_1$ on $R(L)$ correspond to the y, x_1, x_2 column which is a minimum.[1] □

		List Configuration		
Request		y, x_1, x_2	y, x_2, x_1	x_2, y, x_1
1	x_2	3	3	3
2	x_1	2	3	3
3	x_1^3	3	3	3
6	x_2	3	3	2
7	x_2^2	2	2	2
Total:		13	14	13

Table 8.1: For an initial list configuration of $L = y, x_1, x_2$, this table summarizes the potential optimal free move algorithms for $R(L) = \langle x_2, x_1, x_1^3, x_2^3 \rangle$. From Theorem 8.4, we know that after the first request every OPT_FREE moves all the items to the front of the list for the remaining requests. Therefore, the only variable is the configuration of the list immediately after the first request. Columns $3 - 5$ represent the three possible list configurations. Column 1 is the index in $R(L)$ of the request listed in column 2. From the table, the first and third list configurations are optimal, and $\textsc{mff}_1$ corresponds to the first list configuration. The third list configuration corresponds to an algorithm that moves items to the front on every request which also happens to be optimal for this $R(L)$.

[1] The reason we consider $\textsc{mff}_1$ over MTF is that we believe that this lower bound construction can be extended to use a list of arbitrary length such that there are m x's and k y's, giving an $L = y_1, \ldots, y_k, x_1, \ldots, x_m$ and an $R(L) = \langle x_m, \ldots, x_1, x_1^3, \ldots, x_m^3 \rangle$, where the case considered here is $m = 2$ and $k = 1$. In general, for such an initial list configuration and request sequence, MTF is not a OPT_FREE, whereas it can be shown that $\textsc{mff}_k$ is an OPT_FREE for the general $R(L)$.

In the next lemma, we show that there exists an OPT_FREE that moves any item x to the front after the first access of three consecutive requests.

Lemma 8.12. *For $\sigma_r = \langle R_1(L_1), \ldots, R_r(L_r)\rangle$, there exists an* OPT_FREE *that moves any item x to the front of the list immediately after the first access of three consecutive requests to x, where $L_1 = y, x_1, x_2$ and L_j, $1 < j \leq r$, is the list configuration of* OFF *after serving $\langle R_1(L_1), \ldots, R_{j-1}(L_{j-1})\rangle$.*

Proof. In this proof, for σ_r, we consider an arbitrary OPT_FREE algorithm A and show that, if the property does not hold for A, then there exists another OPT_FREE algorithm A' that moves item x to the front of the list immediately after the first access of three consecutive requests to x such that $A'(\sigma_r) \leq A(\sigma_r)$. This will be done by defining a sequence of algorithms A_q, starting with $A_0 = A$, and by reverse induction on i and j over all the $x_i^3 \in R_j(L_j) \in \sigma_r$. That is, we consider the rounds from $R_r(L_r)$ to $R_1(L_1)$ and the consecutive three requests in each round from the last consecutive three requests to the first. For each x^3, if A_q does not move x to the front immediately after the first request, we define A_{q+1}, based on A_q, such that the desired property holds for x^3 and all subsequent consecutive three requests, and we show that the cost does not increase.

In the proof, we use the following notations. Let x^3 be three consecutive requests in $R_j(L_j)$ for which A_q does not have the desired property. We will denote all the requests in σ_r before x^3 by σ_1. The requests after x^3 will be denoted by σ_2. Note that σ_2 could be an empty sequence. For the analysis, we will often (Case 2 and Case 3 below) further partition σ_2 into disjoint partitions $\langle\sigma_3, \ldots, \sigma_p\rangle$ such that $\sigma_r = \left\langle\sigma_1, x^3, \sigma_3, \ldots, \sigma_p\right\rangle$. At a risk of a slight abuse of notation, we will denote the cost of a subsequence of an arbitrary σ_r to an algorithm, ALG, that serves all of σ_r, as $\text{ALG}(r_i, \ldots, r_j) = \text{ALG}(r_1, \ldots, r_j) - \text{ALG}(r_1, \ldots, r_{i-1})$, where the prefix and the suffix are understood implicitly. That is, $\text{ALG}(r_i, \ldots, r_j)$ is the cost accrued by ALG over the requests $r_i, \ldots, r_j$ of σ_r given that ALG has served the prefix $r_1, \ldots, r_{i-1}$ and will serve the remaining requests. Therefore, we have that $\text{ALG}(\sigma_r) = \text{ALG}(\sigma_1) + \text{ALG}(x^3) + \text{ALG}(\sigma_3) + \cdots + \text{ALG}(\sigma_p)$. By Theorem 8.5, we can assume without loss of generality that A_q does not move x further ahead in the list on the second or third requests of x^3.

For a list of length 3, there are two alternating list configurations for OFF (i.e. values for L_j) before each $R_j(L_j)$, y, x_1, x_2 and x_2, x_1, y (see Figure 8.1). Therefore, x_1 is requested in every $R_j(L_j)$, and x_2 and y are requested in alternating $R_j(L_j)$s.

For $x_i \in R_j(L_j)$, which is the last point in σ_r for which A_q does not move x_i to the front immediately after the first request of three consecutive requests, we can

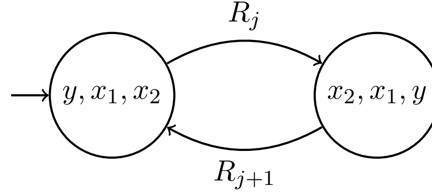

Figure 8.1: An automaton describing the two possible list states of OFF for σ and a list of length 3, where the initial list configuration is y, x_1, x_2, $R_j = \langle x_2, x_1, x_1^3, x_2^3 \rangle$ and $R_{j+1} = \langle y, x_1, x_1^3, y^3 \rangle$.

distinguish between three cases: (1) x_i is never requested again in σ_r; (2) x_i is requested again in $R_{j+1}(L_{j+1})$, i.e. the next round; (3) x_i is requested again in $R_{j+2}(L_{j+2})$, i.e. in the round after the next round. Note that this partitioning is exhaustive.

At each inductive step such that A_q does not have the desired property, we define an algorithm A_{q+1} based on A_q, for $q \geq 0$ as follows.

Definition of $\hat{A}_q$. For $\sigma = \langle \sigma_1, x_i^3, \sigma_2 \rangle$ and algorithm A_q as defined previously, let $\hat{A}_q$ be an algorithm that serves σ_1 in the same manner as A_q and then moves x_i from position $\alpha > 1$ to the front of the list at the first request of x_i. Immediately after serving x_i, the configuration of the list of A_q is B, x_i, C and the configuration of the list of $\hat{A}_q$ is x_i, B, C, where B is the set of items ahead of x_i in the configuration of A_q at this point and C is the set of items behind x_i in the configuration of A_q. As long as the list configuration of A_q and $\hat{A}_q$ differ, for each $x_j \in \sigma_2$, if A_q moves x_j to the front, $\hat{A}_q$ moves x_j to front. Otherwise, $\hat{A}_q$ does not move x_j forward. Once the list configurations of A_q and $\hat{A}_q$ match, $\hat{A}_q$ will serve the remaining requests exactly as A_q. Note that it is possible that the list configuration of $\hat{A}_q$ may never match that of A_q (see Case 1 below).

From the definition of $\hat{A}_q$, and the fact that the list has length of 3, we have the following useful properties.

$$\hat{A}_q(\sigma_1) = A_q(\sigma_1) \ , \tag{8.1}$$

$$|B| \geq 1 \ , \tag{8.2}$$

$$|C| = 2 - |B| \ , \tag{8.3}$$

$$\beta = |B| + 1 \ , \tag{8.4}$$

where β is the position to which x_i is moved by A_q. Further, given that A_q moves x_i from α to β, $1 < \beta \leq \alpha$, and $\hat{A}_q$ moves x_i from α to the front of the list, we have the

following properties.

$$A_q(x_i^3) = \alpha + 2\beta \ , \tag{8.5}$$
$$\hat{A}_q(x_i^3) = \alpha + 2 \tag{8.6}$$
$$= A_q(x_i^3) - 2\beta + 2 \ , \tag{8.7}$$

where (8.7) comes from replacing α in (8.6) by the value of α in (8.5).

Case 1: $x_i \in R_j(L_j)$ is never requested again in σ_r. Recall that $\sigma = \langle \sigma_1, x_i^3, \sigma_2 \rangle$. When $j = r$, this case follows immediately from Lemma 8.10 by defining A_{q+1} to be the algorithm defined in the proof of Lemma 8.10.

When $j < r$, we define A_{q+1} to be $\hat{A}_q$ as defined above. For a list of length 3, this only occurs when $j = r-1$ and $i = 2$, where $L_{r-1} = y, x_1, x_2$ and, hence, $R_{r-1}(L_{r-1}) = \langle x_2, x_1, x_1^3, x_2^3 \rangle$. For $L_{r-1} = y, x_1, x_2$, $R_r(L_r) = \langle y, x_1, x_1^3, y^3 \rangle$ and x_2 is not requested.

Let $\sigma_1 = \langle R_1(L_1), \ldots, R_{r-2}(L_{r-2}), x_2, x_1^4 \rangle$,
$\sigma_2 = \langle y, x_1, x_1^3, y^3 \rangle$.

Cost for $\sigma_2 = \langle y, x_1, x_1^3, y^3 \rangle$. By the case $j = r$, for σ_2, A_q will move x_1 and y to the front of the list on the first request to x_1 and the second request to y after which the configuration of the lists of $\hat{A}_q$ and A_q will match.

If y is in B, then the total cost to access y for $\hat{A}_q$ over σ_2 is at most 2 more than that of A_q over σ_2. This follows from the fact that there are two requests to y before A_q must move y to the front, according to the induction hypothesis and, if y is in B, then $\hat{A}_q$ has x_1 in front of y, whereas A_q does not.

If y is in C, the total cost to access y for $\hat{A}_q$ over σ_2 is at most 1 more than that of A_q over σ_2. This can occur if, on the first access to y, A_q were to move y between x_1 and x_2 in its list. Then, on the second access, y is one item closer to the front in the list of A_q as compared to the list of $\hat{A}_q$.

By the induction hypothesis, A_q must move x_1 to the front on the first request to x_1 in σ_2. Therefore, if x_1 is in B, the first access costs 1 more to $\hat{A}_q$ as compared to A_q and, if x_1 is in C, the cost for the first access is the same for both $\hat{A}_q$ and A_q.

This gives that for σ_2,

$$\hat{A}_q(\sigma_2) \leq A_q(\sigma_2) + 2|B| + |C| - 1$$
$$= A_q(\sigma_2) + |B| + 1 \ , \tag{8.8}$$

where the last line comes from applying (8.3).

Using (8.1), we get

$$\begin{aligned}\hat{A}_q(\sigma_r) &= A_q(\sigma_1) + \hat{A}_q(x_i^3, \sigma_2) \\ &\le A_q(\sigma_r) - 2\beta + |B| + 3 \text{ , using (8.7) and (8.8),} \\ &= A_q(\sigma_r) - |B| + 1 \text{ , using (8.4),} \\ &\le A_q(\sigma_r) \text{ , by (8.2).}\end{aligned}$$

Case 2: $x_i \in R_j(L_j)$ and $x_i \in R_{j+1}(L_{j+1})$, i.e. x_i is requested in the next round. We define A_{q+1} to be $\hat{A}_q$. For $L_j = y, x_1, x_2$, $R_j(L_j) = \langle x_2, x_1, x_1^3, x_2^3 \rangle$ and $R_{j+1}(L_{j+1}) = \langle y, x_1, x_1^3, y^3 \rangle$.
Let $\sigma_1 = \langle R_1(L_1), \ldots, R_{j-1}(L_{j-1}), x_2, x_1 \rangle$,
$\sigma_3 = \langle x_2^3, y, x_1, x_1^3 \rangle$,
$\sigma_4 = \langle y^3, R_{j+2}(L_{j+2}), \ldots, R_r(L_r) \rangle$.
Note that $\sigma_2 = \langle \sigma_3, \sigma_4 \rangle$.[1]

Cost for $\sigma_3 = \langle x_2^3, y, x_1, x_1^3 \rangle$. After serving x_1^3, the configuration of the list of $\hat{A}_q$ is x_1, B, C and the list of A_q is B, x_1, C. According to the induction hypothesis, A_q will move x_2 to the front on its first request in σ_3. This request and the request to y will each cost 1 more to $\hat{A}_q$ than A_q if they are in B. If they are in C, there is no additional cost to $\hat{A}_q$ as compared to A_q. Finally, on the first request to x_1 in σ_3, x_1 is no further from the front in $\hat{A}_q$ than it is in A_q. Then, by the induction hypothesis, A_q moves x_1 to the front for the remaining requests to x_1 in σ_3 as does $\hat{A}_q$. Therefore,

$$\hat{A}_q(\sigma_3) \le A_q(\sigma_3) + |B| \ . \tag{8.9}$$

List Configuration after σ_3. By the induction hypothesis, A_q will move x_2 and x_1 to the front of the list immediately after the first access to each one in σ_3. Consider the state of the list of A_q and $\hat{A}_q$ immediately after serving σ_3, depending on whether or not A_q moves y to the front. If A_q does not move y to the front of the list, the configuration of its list will be x_1, x_2, y, and, by the definition of $\hat{A}_q$, its list configuration will also be x_1, x_2, y. If A_q does move y to the front of the list, the configuration of its list will be x_1, y, x_2, and, by the definition of $\hat{A}_q$, its list configuration will also be x_1, y, x_2.

[1] Since x_1 is requested four times in a row, there is an x_1^3 (the last three x_1's) and an x_1^4 (all four x_1's). We only consider the last x_1^3 in $R_j(L_j)$ as the x_1^4 in $R_j(L_j)$ is to the advantage of $\hat{A}_q$ given that x_1 is requested an additional time. The additional request to x_1 gives an additional savings of $\beta - 1$ to $\hat{A}_q$ over σ_2 as compared to the case of the last three x_1's. So, if the claim holds considering only x_1^3, it must hold for x_1^4.

Cost for $\sigma_4 = \langle y^3, R_{j+2}, \ldots, R_r \rangle$. After serving σ_3, the configurations of the lists of $\hat{A}_q$ and of A_q are the same. Therefore,

$$\hat{A}_q(\sigma_4) = A_q(\sigma_4) \ . \tag{8.10}$$

Summing (8.1), (8.7), (8.9), and (8.10), we get that the cost for $\hat{A}_q$ over σ_r is

$$\begin{aligned} \hat{A}_q(\sigma_r) &\leq A_q(\sigma_r) - 2\beta + 2 + |B| \\ &= A_q(\sigma_r) - |B| \ , \text{ using (8.4)}, \\ &< A_q(\sigma_r) \ , \text{ by (8.2)}. \end{aligned}$$

Case 3: $x_i \in R_j(L_j)$ and $x_i \in R_{j+2}(L_{j+2})$, i.e. x_i is requested in the round after next. We define A_{q+1} to be $\hat{A}_q$. For $L_j = y, x_1, x_2$, $R_j(L_j) = \langle x_2, x_1, x_1^3, x_2^3 \rangle$, $R_{j+1}(L_{j+1}) = \langle y, x_1, x_1^3, y^3 \rangle$, and $R_{j+2}(L_{j+2}) = \langle x_2, x_1, x_1^3, x_2^3 \rangle$.
Let $\sigma_1 = \langle R_1(L_1), \ldots, R_{j-1}(L_{j-1}), x_2, x_1, x_1^3 \rangle$,
$\sigma_3 = \langle y, x_1, x_1^3, y^3 \rangle$, and
$\sigma_4 = \langle R_{j+2}(L_{j+2}), \ldots, R_r(L_r) \rangle$.
Note that $\sigma_2 = \langle \sigma_3, \sigma_4 \rangle$.

Cost for $\sigma_3 = \langle y, x_1, x_1^3, y^3 \rangle$. After serving σ_1, x_2^3, the configuration of the list of $\hat{A}_q$ is x_2, B, C and the configuration of the list of A_q is B, x_2, C. By the induction hypothesis, A_q will move y to the front of the list on the second request in σ_3 and x_1 to the front of the list on the first request to it in σ_3. This is exactly the same scenario as σ_2 for Case 1. Similarly to (8.8) for Case 1,

$$\hat{A}_q(\sigma_3) \leq A_q(\sigma_3) + |B| + 1 \ . \tag{8.11}$$

Since both y and x_1 are moved to the front of the list in σ_3, the configuration of the lists of A_q and $\hat{A}_q$ must be y, x_1, x_2 after σ_3.

Cost for $\sigma_4 = \langle R_{j+2}(L_{j+2}), \ldots, R_r(L_r) \rangle$. After serving σ_3, the configurations of the lists of $\hat{A}_q$ and of A_q are the same. Therefore,

$$\hat{A}_q(\sigma_4) = A_q(\sigma_4) \ . \tag{8.12}$$

Summing (8.1), (8.7), (8.11), and (8.12), we get that the cost for $\hat{A}_q$ over σ_r is

$$\begin{aligned} \hat{A}_q(\sigma_r) &\leq A_q(\sigma_r) - 2\beta + |B| + 3 \\ &= A_q(\sigma_r) - |B| + 1 \ , \text{ using (8.4)}, \\ &\leq A_q(\sigma_r) \ , \text{ by (8.2)}. \end{aligned}$$

To conclude, for each of the three cases possible at each inductive step, we have shown that there exists an algorithm with the desired property. Overall, we have shown that $A'(\sigma_r) = A_q(\sigma_r) \leq \cdots \leq A_0(\sigma_r) = A(\sigma_r)$ which concludes the proof. $\square$

For σ_r as defined above, Lemma 8.12 shows that there exists an OPT_FREE that moves x to the front when x is requested at least three times in a row. Let OPT_FREE* be such an OPT_FREE. This guarantees that the list configuration of OPT_FREE* after each $R_j(L_j) \in \sigma_r$ is the same as OFF. For an initial list configuration of $L = y, x_1, x_2$, Lemma 8.11 shows that the algorithm MFF$_1$ is an optimal free move algorithm for $R(L)$. Combined with the previous fact, this implies that MFF$_1$ is an optimal free move algorithm for σ_r since the list configuration of MFF$_1$ after serving $R_1(L_1), \ldots, R_j(L_j)$, $1 \leq j \leq r$, is the same as OPT_FREE*. Hence, MFF$_1$ serves all $R_{j+1}(L_{j+1})$ at a cost no more than that of OPT_FREE*. This is formally stated in the following lemma.

Lemma 8.13. *For* $\sigma_r = \langle R_1(L_1), \ldots, R_r(L_r)\rangle$, MFF$_1(\sigma_r)$ = OPT_FREE(σ_r), *where* $L_1 = y, x_1, x_2$ *and* L_j, $1 < j \leq r$, *is the list configuration of* OFF *after serving* $\langle R_1(L_1), \ldots, R_{j-1}(L_{j-1})\rangle$.

Proof. By Lemma 8.12, there exists an OPT_FREE that will have the same configuration as OFF and MFF$_1$ immediately before $R_j(L_j)$, $1 \leq j \leq r$. Let OPT_FREE* be such an OPT_FREE. Since the list configuration of MFF$_1$ and OPT_FREE* match prior to serving every $R_j(L_j)$, Lemma 8.11 implies that MFF$_1(\sigma_r)$ = OPT_FREE(σ_r). $\square$

In the following lemma and theorem, given that, for any $r > 0$, MFF$_1$ is an optimal free move algorithm for σ_r, we can lower bound the worst-case ratio between OPT_FREE(σ) and OPT(σ) by analysing the ratio between MFF$_1(\sigma_r)$ and OFF(σ_r) for $\sigma_r = \langle R_1(L_1), \ldots, R_r(L_r)\rangle$, where $L_1 = y, x_1, x_2$ and L_j, $1 < j \leq r$, is the list configuration of OFF after serving $\langle R_1(L_1), \ldots, R_{j-1}(L_{j-1})\rangle$.

Lemma 8.14. *For* $r > 0$, $\frac{\text{OPT_FREE}(\sigma_r)}{\text{OPT}(\sigma_r)} \geq \frac{\text{OPT_FREE}(\sigma_r)}{\text{OFF}(\sigma_r)} = \frac{13}{12} > 1.083$.

Proof. Let $\sigma = \langle R_1(L_1), \ldots, R_r(L_r)\rangle$, where $L_1 = y, x_1, x_2$ and L_j, $1 < j \leq r$, is the list configuration of OFF after serving $\langle R_1(L_1), \ldots, R_{j-1}(L_{j-1})\rangle$.

From Lemma 8.13 and Lemma 8.11, MFF$_1$ is an optimal free move algorithm for σ_r with a cost of $13r$ and, from Lemma 8.7, the cost of OFF for σ_r is $12r$. Therefore, $\frac{\text{OPT_FREE}(\sigma_r)}{\text{OPT}(\sigma_r)} \geq \frac{\text{OPT_FREE}(\sigma_r)}{\text{OFF}(\sigma_r)} = \frac{13}{12}$. $\square$

Theorem 8.15. *Let* ALG_FREE *be a free move algorithm such that* ALG_FREE$(\sigma) \leq \alpha$OPT$(\sigma) + \eta$. *For any* η *not dependant on* σ, $\alpha \geq 13/12$.

Proof. For the sake of contradiction, let $\alpha = \frac{13}{12} - \varepsilon$ for an $\varepsilon > 0$. Hence, $\text{ALG_FREE}(\sigma_r) \leq \left(\frac{13}{12} - \varepsilon\right) \text{OPT}(\sigma_r) + \eta$. Solving for ε and σ_r such that $\text{ALG_FREE}(\sigma_r) > \eta$,

$$\begin{aligned} \varepsilon &\leq \frac{13}{12} - \frac{\text{ALG_FREE}(\sigma_r) - \eta}{\text{OPT}(\sigma_r)} \\ &\leq \frac{13}{12} - \frac{\text{ALG_FREE}(\sigma_r) - \eta}{\text{OFF}(\sigma_r)} \\ &\leq \frac{\eta}{\text{OFF}(\sigma_r)} \end{aligned} \tag{8.13}$$

by the fact that $\text{OFF}(\sigma_r) \geq \text{OPT}(\sigma_r)$ and Lemma 8.14. Since η does not depend on r, $\text{ALG_FREE}(\sigma_r)$ increases as r increases and $\text{OFF}(\sigma_r)$ increases as r increases, a contradiction is found by choosing a sufficiently large r such that $\text{ALG_FREE}(\sigma_r) > \eta$ and (8.13) is no longer true. □

9 Conclusion

In this thesis, we considered online computation with advice as a means to study the impact of knowledge about the future on the competitive ratio of online algorithms. Specifically, we focused on the online advice model of [EFKR11] over the semi-online advice model of [BKK+09]. The advantage of the online advice model is that the advice is provided in an online manner which is very natural in the online setting. The semi-online advice model is stronger than the online model in that the advice is received before any requests are seen. This allows for algorithms that use amounts of advice in total that are sublinear in the length of the request sequence.

The problems we considered were the k-server problem, the bin packing problem, the dual bin packing problem, the problem of scheduling on m identical machines, the reordering buffer problem, and the list update problem. For the k-server problem, we improved the upper bound for general metric spaces and presented a novel algorithm for finite trees that uses advice that is, in the worst case, logarithmic in the height of the tree. An open question is that of establishing better bounds on the competitive ratio of the k-server problem as a function of the advice.

For the packing and scheduling problems, we showed that it is possible to approach a competitive ratio of 1 with a constant amount of bits of advice per request, yet $\Omega(\log N)$ bits of advice per request are required to be optimal, where N is the (optimal) number of bins or machines. A natural follow up to this work would be to explore more general versions of these problems, e.g. multi-dimensional packing, variable sized bins, related and unrelated machines, variable profits for items. Many of the offline versions of the generalized problems have PTAS'(APTAS'), but not all. It would be interesting to

see if our techniques here could be extended to the more generalized versions of the problems. For the packing and scheduling problems considered here, except for the bin packing problem [BKLLO12], only lower bounds on the amount of advice required for optimality are known. It would be interesting to find lower bounds on the amount of advice needed for a competitive ratio, where the amount of advice is a function of the competitive ratio. In addition, for the dual bin packing problem, we showed that an algorithm that uses 1 bit of advice per request has a competitive ratio of 3/2. This algorithm ensures that the largest items are rejected and packs the remaining items in a reasonably efficient manner (FIRST FIT). We conjecture that the competitive ratio of this algorithm is actually 4/3 which is the competitive ratio of the offline FIRST FIT INCREASING algorithm [JLT78].

For the reordering buffer management problem, we presented an algorithm that has the flavour of a PTAS. It has a competitive ratio of $1 + \varepsilon$ and uses $O(\log \frac{1}{\varepsilon})$ bits of advice per request. We complimented this by providing a lower bound of $\Omega(\log k)$ on the number of bits of advice per request for an online algorithm to achieve a competitive ratio of 1. Since no PTAS is known for the offline version of this problem which is NP-hard, it would be interesting to try adapting our approach to the offline version of the problem in hopes of developing a PTAS. As with the packing and scheduling problems, it would be interesting to see how to extend our results to the more general cases, e.g. when the cost to switch from colour c to colour c' depends on c and c'.

The main result presented in this thesis for the list update problem shows that the difference in performance between an offline optimal algorithm restricted to free moves and an unrestricted offline optimal algorithm is at least a multiplicative factor of 13/12. This question has essentially been open since 1996 [RW96]. Due to some preliminary results and some computer simulations, we believe that the sequence which only uses 3 distinct items for the lower bound of 13/12 can be generalized to a sequence that uses any number of distinct items such that a lower bound of $3 - \sqrt{3}$ can be achieved. In addition, we presented some natural online algorithms with advice for the list update problem and remarked that known randomized algorithms provide immediate trivial upper bounds for the online algorithms with advice. There are no known algorithms with advice (aside from the trivial algorithm using ℓ bits of advice per request) that beat the competitive ratio of the best known randomized algorithm COMB of 1.6 [AvSW95].

This is an important open question that would also provide an upper bound for an optimal free move algorithm.

In the wider context of online algorithms with advice as a whole, there remains many online problems that have not yet been studied in the context of advice and, for those that have been studied, still many questions remain open. Furthermore, on a more fundamental direction, an important direction to study in the relation between online algorithms with advice and online randomized algorithms without advice.

References

[AAS98] Adi Avidor, Yossi Azar, and Jiří Sgall. Ancient and new algorithms for load balancing in the lp norm. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 426–435, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics. 10

[AAWY97] N. Alon, Y. Azar, G.J. Woeginger, and T. Yadid. Approximation schemes for scheduling. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 493–500. Society for Industrial and Applied Mathematics, 1997. 10, 22, 23

[ABE+02] Yossi Azar, Joan Boyar, Leah Epstein, Lene M. Favrholdt, Kim S. Larsen, and Morten N. Nielsen. Fair versus unrestricted bin packing. *Algorithmica*, 34(2):181–196, 2002. 8, 9

[ACER11] Anna Adamaszek, Artur Czumaj, Matthias Englert, and Harald Räcke. Almost tight bounds for reordering buffer management. In Lance Fortnow and Salil P. Vadhan, editors, *STOC*, pages 607–616. ACM, 2011. 12

[ADLO08] S. Angelopoulos, R. Dorrigiv, and A. López-Ortiz. List update with locality of reference. *LATIN 2008: Theoretical Informatics*, pages 399–410, 2008. 2

[AE98] Yossi Azar and Leah Epstein. On-line machine covering. *J Scheduling*, 1(2):67–77, 1998. 10

[AER10] Noa Avigdor-Elgrabli and Yuval Rabani. An improved competitive algorithm for reordering buffer management. In Moses Charikar, editor, *SODA*, pages 13–21. SIAM, 2010. 12

[AER13a] Noa Avigdor-Elgrabli and Yuval Rabani. A constant factor approximation algorithm for reordering buffer management. In Sanjeev Khanna, editor, *SODA*, pages 973–984. SIAM, 2013. 12

[AER13b] Noa Avigdor-Elgrabli and Yuval Rabani. An optimal randomized online algorithm for reordering buffer management. *CoRR*, abs/1303.3386, 2013. 12

[AFG05] S. Albers, L.M. Favrholdt, and O. Giel. On paging with locality of reference. *Journal of Computer and System Sciences*, 70(2):145–175, 2005. 2

[AKM12] Yuichi Asahiro, Kenichi Kawahara, and Eiji Miyano. Np-hardness of the sorting buffer problem on the uniform metric. *Discrete Applied Mathematics*, 160(10-11):1453–1464, 2012. 12

[AL08] S. Albers and S. Lauer. On list update with locality of reference. *Automata, Languages and Programming*, pages 96–107, 2008. 2

[Alb95] Susanne Albers. Improved randomized on-line algorithms for the list update problem. In Kenneth L. Clarkson, editor, *SODA*, pages 412–419. ACM/SIAM, 1995. 119, 120

[Alb97] S. Albers. On the influence of lookahead in competitive paging algorithms. *Algorithmica*, 18(3):283–305, 1997. 2

[Alb98] S. Albers. A competitive analysis of the list update problem with lookahead. *Theoretical Computer Science*, 197(1):95–109, 1998. 2

[Alb02] Susanne Albers. On randomized online scheduling. In *In Proc. 34th Symp. Theory of Computing (STOC*, pages 134–143. ACM, 2002. 10

[Amb00] Christoph Ambühl. Offline list update is np-hard. In Paterson [Pat00], pages 42–51. 13

[ARRvS13] Anna Adamaszek, Marc P. Renault, Adi Rosén, and Rob van Stee. Reordering buffer management with advice. In *WAOA*, 2013. To appear. 4, 5

[AvSW95] Susanne Albers, Bernhard von Stengel, and Ralph Werchner. A combined bit and timestamp algorithm for the list update problem. *Inf. Process. Lett.*, 56(3):135–139, 1995. 13, 14, 121, 134

[BB02] Daniel K. Blandford and Guy E. Blelloch. Index compression through document reordering. In *DCC*, pages 342–351. IEEE Computer Society, 2002. 12

[BBG12] János Balogh, József Békési, and Gábor Galambos. New lower bounds for certain classes of bin packing algorithms. *TCS*, 440-441(0):1–13, 2012. 7

[BBMN11] Nikhil Bansal, Niv Buchbinder, Aleksander Madry, and Joseph Naor. A polylogarithmic-competitive algorithm for the k-server problem. In Rafail Ostrovsky, editor, *FOCS*, pages 267–276. IEEE, 2011. 5

[BCL02] Wolfgang W. Bein, Marek Chrobak, and Lawrence L. Larmore. The 3-server problem in the plane. *Theor. Comput. Sci.*, 289(1):335–354, 2002. 5

[BDBK+90] Shai Ben-David, Allan Borodin, Richard M. Karp, Gábor Tardos, and Avi Wigderson. On the power of randomization in online algorithms (extended abstract). In *STOC*, pages 379–386. ACM, 1990. 15, 16, 17, 19

[BEY98] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, New York, NY, USA, 1998. 2, 5, 15, 16

[BHK+13] Hans-Joachim Böckenhauer, Juraj Hromkovic, Dennis Komm, Sacha Krug, Jasmin Smula, and Andreas Sprock. The string guessing problem as a method to prove lower bounds on the advice complexity. In Ding-Zhu Du and Guochuan Zhang, editors, *COCOON*, volume 7936 of *Lecture Notes in Computer Science*, pages 493–505. Springer, 2013. v, 4, 21, 107

[BIRS95] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50(2):244–258, 1995. 2

[BK04] Yair Bartal and Elias Koutsoupias. On the competitive ratio of the work function algorithm for the k-server problem. *Theor. Comput. Sci.*, 324(2-3):337–345, 2004. 5

[BKK+09] Hans-Joachim Böckenhauer, Dennis Komm, Rastislav Královic, Richard Královic, and Tobias Mömke. On the advice complexity of online problems. In Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra, editors, *ISAAC*, volume 5878 of *Lecture Notes in Computer Science*, pages 331–340. Springer, 2009. iii, 2, 3, 4, 7, 9, 10, 14, 15, 18, 19, 133

[BKKK11] Hans-Joachim Böckenhauer, Dennis Komm, Rastislav Královic, and Richard Královic. On the advice complexity of the k-server problem. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP (1)*, volume 6755 of *Lecture Notes in Computer Science*, pages 207–218. Springer, 2011. Also as technical report at ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/7xx/703.pdf. 3, 5, 28, 33

[BKKR12] Hans-Joachim Böckenhauer, Dennis Komm, Richard Královic, and Peter Rossmanith. On the advice complexity of the knapsack problem. In David Fernández-Baca, editor, *LATIN*, volume 7256 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 2012. 3, 9

[BKLLO12] Joan Boyar, Shahin Kamali, Kim S. Larsen, and Alejandro López-Ortiz. Online bin packing with advice. *CoRR*, abs/1212.4016, 2012. 3, 7, 8, 134

[BKLLO14] Joan Boyar, Shahin Kamali, Kim S. Larsen, and Alejandro López-Ortiz. On the list update problem with advice. In Adrian Horia Dediu, Carlos Martín-Vide, José Luis Sierra-Rodríguez, and Bianca Truthe, editors, *LATA*, volume 8370 of *Lecture Notes in Computer Science*, pages 210–221. Springer, 2014. 14

[BLN01] Joan Boyar, Kim S. Larsen, and Morten N. Nielsen. The accommodating function: A generalization of the competitive ratio. *SIAM J. Comput.*, 31(1):233–258, 2001. 2, 8

[Bre98] D. Breslauer. On competitive on-line paging with lookahead. *Theoretical Computer Science*, 209(1):365–375, 1998. 2

[BSTW86] Jon Louis Bentley, Daniel Dominic Sleator, Robert Endre Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Commun. ACM*, 29(4):320–330, 1986. 13

[Cha92] Barun Chandra. Does randomization help in on-line bin packing? *Inf. Process. Lett.*, 43(1):15–19, August 1992. 7, 45

[CJ13] Marek Cygan and Łukasz Jeż. Online knapsack revisited. In *WAOA*, 2013. To appear. 8

[CL79] Edward G Coffman, Jr and Joseph YT Leung. Combinatorial analysis of an efficient algorithm for processor and storage allocation. *SIAM Journal on Computing*, 8(2):202–217, 1979. 8

[CL91] Marek Chrobak and Lawrence L. Larmore. An optimal on-line algorithm for k-servers on trees. *SIAM J. Comput.*, 20(1):144–148, 1991. 5

[CL92] Marek Chrobak and Lawrence L Larmore. The server problem and on-line games. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 7:11–64, 1992. 5

[CMSvS12] Ho-Leung Chan, Nicole Megow, René Sitters, and Rob van Stee. A note on sorting buffers offline. *Theor. Comput. Sci.*, 423:11–18, 2012. 12

[CvVW94] Bo Chen, André van Vliet, and Gerhard J. Woeginger. A lower bound for randomized on-line scheduling algorithms. *Information Processing Letters*, 51(5):219 – 222, 1994. 10

[DHZ12] Reza Dorrigiv, Meng He, and Norbert Zeh. On the advice complexity of buffer management. In Kun-Mao Chao, Tsan sheng Hsu, and Der-Tsai Lee, editors, *ISAAC*, volume 7676 of *Lecture Notes in Computer Science*, pages 136–145. Springer, 2012. 4

[DKP08] Stefan Dobrev, Rastislav Královič, and Dana Pardubská. How much information about the future is needed? In *SOFSEM'08: Proceedings of the 34th conference on Current trends in theory and practice of computer science*, pages 247–258, Berlin, Heidelberg, 2008. Springer-Verlag. 6

[DLO05] Reza Dorrigiv and Alejandro López-Ortiz. A survey of performance measures for on-line algorithms. *SIGACT News*, 36(3):67–81, 2005. 2

[Doh13] Jérôme Dohrau. Online makespan scheduling with sublinear advice. Technical report, Eidgenössische Technische Hochschule Zürich, 2013. 4, 11

[EF03] Leah Epstein and Lene M. Favrholdt. On-line maximizing the number of items packed in variable-sized bins. *Acta Cybern.*, 16(1):57–66, 2003. 8

[EFKR11] Yuval Emek, Pierre Fraigniaud, Amos Korman, and Adi Rosén. Online computation with advice. *Theor. Comput. Sci.*, 412(24):2642–2656, 2011. iii, v, 2, 3, 4, 5, 9, 15, 17, 18, 19, 20, 21, 107, 133

[ERW10] Matthias Englert, Harald Räcke, and Matthias Westermann. Reordering buffers for general metric spaces. *Theory of Computing*, 6(1):27–46, 2010. 12

[EW05] Matthias Englert and Matthias Westermann. Reordering buffer management for non-uniform cost models. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 627–638. Springer, 2005. 12

[FdlVL81] Wenceslas Fernandez de la Vega and George S. Lueker. Bin packing can be solved within 1+epsilon in linear time. *Combinatorica*, 1(4):349–355, 1981. 7, 22, 23, 43, 49

[FM97] A. Fiat and M. Mendel. Truly online paging with locality of reference. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 326–335. IEEE, 1997. 2

[FW00] Rudolf Fleischer and Michaela Wahl. Online scheduling revisited. In Paterson [Pat00], pages 202–210. 10

[GGU72] M. R. Garey, Ronald L. Graham, and Jeffrey D. Ullman. Worst-case analysis of memory allocation algorithms. In Patrick C. Fischer, H. Paul Zeiger, Jeffrey D. Ullman, and Arnold L. Rosenberg, editors, *STOC*, pages 143–150. ACM, 1972. 7

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979. 7, 8, 9

[GKLO13] Sushmita Gupta, Shahin Kamali, and Alejandro López-Ortiz. On advice complexity of the k-server problem under sparse metrics. In Thomas Moscibroda and Adele A. Rescigno, editors, *SIROCCO*, volume 8179 of *Lecture Notes in Computer Science*, pages 55–67. Springer, 2013. 3, 6

[Gra66] Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966. 10

[Gro95] Edward F. Grove. Online bin packing with lookahead. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, SODA '95, pages 430–436, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics. 2

[GSV04] Kai Gutenschwager, Sven Spiekermann, and Stefan Voß. A sequential ordering problem in automotive paint shops. *Internat. J. Production Research*, 42(9):1865–1878, 2004. 12

[HS87] Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM*, 34(1):144–162, January 1987. 9, 22, 23

[IK97] Sandy Irani and Anna R. Karlin. On online computation. In *Approximation Algorithms for NP-Hard Problems, chapter 13*, pages 521–564. PWS Publishing Company, 1997. 1

[IKP96] S. Irani, A.R. Karlin, and S. Phillips. Strongly competitive algorithms for paging with locality of reference. *SIAM Journal on Computing*, 25(3):477–497, 1996. 2

[Ira91] Sandy Irani. Two results on the list update problem. *Inf. Process. Lett.*, 38(6):301–306, 1991. 13

[JDU+74] David S. Johnson, Alan J. Demers, Jeffrey D. Ullman, M. R. Garey, and Ronald L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. Comput.*, 3(4):299–325, 1974. 7

[JLT78] Edward G. Coffman Jr., Joseph Y.-T. Leung, and D. W. Ting. Bin packing: Maximizing the number of pieces packed. *Acta Inf.*, 9:263–271, 1978. 8, 134

[Joh73] D.S. Johnson. *Near-optimal Bin Packing Algorithms*. PhD thesis, MIT, 1973. 44

[Joh74] David S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. Syst. Sci.*, 9(3):256–278, 1974. 7

[Kel99] Hans Kellerer. A polynomial time approximation scheme for the multiple knapsack problem. In Dorit S. Hochbaum, Klaus Jansen, José D. P. Rolim, and Alistair Sinclair, editors, *RANDOM-APPROX*, volume 1671 of *Lecture Notes in Computer Science*, pages 51–62. Springer, 1999. 8, 22, 23

[KK11] Dennis Komm and Richard Královic. Advice complexity and barely random algorithms. *RAIRO - Theor. Inf. and Applic.*, 45(2):249–267, 2011. 4, 10

[KKM12] Dennis Komm, Richard Královic, and Tobias Mömke. On the advice complexity of the set cover problem. In Edward A. Hirsch, Juhani Karhumäki, Arto Lepistö, and Michail Prilutskii, editors, *CSR*, volume 7353 of *Lecture Notes in Computer Science*, pages 241–252. Springer, 2012. 3

[KP95] Elias Koutsoupias and Christos H. Papadimitriou. On the k-server conjecture. *J. ACM*, 42(5):971–983, 1995. 5

[KPP04] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer, 2004. 8

[KRSW04] Jens Krokowski, Harald Räcke, Christian Sohler, and Matthias Westermann. Reducing state changes with a pipeline buffer. In Bernd Girod, Marcus A. Magnor, and Hans-Peter Seidel, editors, *VMV*, page 217. Aka GmbH, 2004. 12

[LL85] C. C. Lee and D. T. Lee. A simple on-line bin-packing algorithm. *J. ACM*, 32(3):562–572, July 1985. 7, 43, 45

[LP97] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1997. 86

[Mat99] Jiří Matoušek. On embedding trees into uniformly convex banach spaces. *Israel Journal of Mathematics*, 114:221–237, 1999. 7, 26, 34

[MMS90] Mark S. Manasse, Lyle A. McGeoch, and Daniel Dominic Sleator. Competitive algorithms for server problems. *J. Algorithms*, 11(2):208–230, 1990. 5

[Pat00] Mike Paterson, editor. *Algorithms - ESA 2000, 8th Annual European Symposium, Saarbrücken, Germany, September 5-8, 2000, Proceedings*, volume 1879 of *Lecture Notes in Computer Science*. Springer, 2000. 138, 142

[RC03] John F. Rudin, III and R. Chandrasekaran. Improved bounds for the online scheduling problem. *SIAM J. Comput.*, 32(3):717–735, March 2003. 10

[Ren10] Marc Renault. Online algorithms with advice. Master's thesis, MPRI – Université Paris Diderot - Paris 7, 2010. 6

[RR11] Marc P. Renault and Adi Rosén. On online algorithms with advice for the k-server problem. In Roberto Solis-Oba and Giuseppe Persiano, editors, *WAOA*, volume 7164 of *Lecture Notes in Computer Science*, pages 198–210. Springer, 2011. 4, 6

[RR12] Marc P. Renault and Adi Rosén. On online algorithms with advice for the k-server problem. *Theory of Computing Systems*, pages 1–19, 2012. 3, 4, 6

[RRvS13] Marc P. Renault, Adi Rosén, and Rob van Stee. Online algorithms with advice for bin packing and scheduling problems. *CoRR*, abs/1311.7589, 2013. 3, 4, 5

[RSW02] Harald Räcke, Christian Sohler, and Matthias Westermann. Online scheduling for sorting buffers. In Rolf H. Möhring and Rajeev Raman, editors, *ESA*, volume 2461 of *Lecture Notes in Computer Science*, pages 820–832. Springer, 2002. 11, 12, 85

[RW96] Nick Reingold and Jeffery Westbrook. Off-line algorithms for the list update problem. *Inf. Process. Lett.*, 60(2):75–80, 1996. v, 14, 117, 118, 121, 134

[RWS94] Nick Reingold, Jeffery Westbrook, and Daniel Dominic Sleator. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11(1):15–32, 1994. 13, 121

[Sei02] Steven S. Seiden. On the online bin packing problem. *J. ACM*, 49(5):640–671, 2002. 7

[Sga97] Jiří Sgall. A lower bound for randomized on-line multiprocessor scheduling. *Inf. Process. Lett.*, 63(1):51 – 55, 1997. 10

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985. 5, 7, 10, 13, 14, 16, 117

[Woe97] Gerhard J. Woeginger. A polynomial-time approximation scheme for maximizing the minimum machine completion time. *Oper. Res. Lett.*, 20(4):149 – 154, 1997. 10, 22, 23