

Master's Thesis

Learning-Augmented Online Algorithms for the 2-Server Problem on the Line and Generalizations

Alexander Lindermayr

Matriculation No. 3170905

June 29, 2020

First Examiner: Prof. Dr. Nicole Megow

Second Examiner: Prof. Dr. Sebastian Siebertz

Advisor: Prof. Dr. Nicole Megow, Dr. Bertrand Simon

Declaration of Authorship

I hereby confirm that the thesis I am submitting is my own original work. Any use other materials and works of other authors is properly acknowledged at their point of use.

Bremen, June 29, 2020

Alexander Lindermayr

Abstract

Online optimization captures relevant real-world problems, where the input is unknown in advance and arrives over time. Therefore, classical online algorithms make no assumptions on the future, which gives their analysis a pessimistic flavor. Indeed, in many real-world applications we can make statements about the future using modern machine-learning approaches to predict it. In order to still achieve performance guarantees in such a setting, Lykouris and Vassilvitskii [27] recently introduced a framework which measures the efficiency of such algorithms by analyzing them with respect to the quality of the given prediction, while making no assumptions on the generation of these.

We apply this framework to the *k-server problem on the line*, which is a well-studied and fundamental online problem. There exists a simple, deterministic, *k*-competitive online algorithm, which is optimal. We propose a family of learning-augmented algorithms for this problem and bound their competitive ratio with respect to the prediction's quality, such that they break the lower bound of *k* given good predictions. For some algorithms of the family we also prove a general performance guarantee, independently of the prediction.

For the special case of two servers, we show that this family is in a certain sense optimal among all learning-augmented algorithms for the problem. We also state a modification of the algorithms for tree metrics and achieve almost the same results as on the line. For the *2-taxi problem*, which was recently introduced by Coester and Koutsoupias [15] and generalizes the 2-server problem, we present a similar approach using machine-learned predictions, and again circumvent existing lower bounds. Finally, we empirically evaluate the proposed algorithms and illustrate their applicability in practice.

Contents

1	Preliminaries	5
1.1	Online algorithms	5
1.2	Adversaries and competitive analysis	7
1.3	Lower bounds	8
1.4	Untrusted predictions	9
1.5	Further related work	11
2	The k-server problem on the line	13
2.1	Problem definition	13
2.2	A history of the problem, applications and generalizations	15
2.3	The optimal offline solution	17
2.4	A lower bound	18
2.5	The greedy algorithm	20
2.6	The Double-Coverage algorithm	21
3	Two servers on the line with untrusted predictions	27
3.1	Improving Double-Coverage with untrusted predictions	27
3.1.1	Definition of a prediction model	27
3.1.2	First approach	29
3.1.3	Modifying Double-Coverage	30
3.2	The LAMBDADC algorithm	33
3.3	Competitive analysis	34
3.3.1	Competitive analysis with respect to the error	35
3.3.2	Consistency	37
3.3.3	Robustness	37
3.4	Optimality of LAMBDADC	40
4	The k-server problem on the line with untrusted predictions	45
4.1	LAMBDADC with k servers	45
4.2	Competitive analysis with respect to the error	46
4.3	Robustness with k servers	49

5	The k-server problem on trees with untrusted predictions	53
5.1	Problem definition	53
5.2	LAMBDADC for trees	54
5.3	Robustness on trees	60
6	The k-taxi problem	61
6.1	Problem definition	61
6.2	A deterministic online algorithm for the hard 2-taxi problem	63
6.3	The LAMBDABIASEDDC algorithm	65
6.4	Competitive analysis of LAMBDABIASEDDC	66
6.4.1	Competitive analysis with respect to the error	66
6.4.2	Consistency	68
6.4.3	Robustness	69
6.5	A worst-case instance for LAMBDABIASEDDC	71
6.5.1	Consistency	71
6.5.2	Robustness	75
7	Experiments	79
7.1	The k-server problem on the line	81
7.1.1	Uniformly generated instances	82
7.1.2	BrightKite instances	82
7.2	The 2-taxi problem on the line	83
8	Conclusion and open questions	85
	Bibliography	87

Introduction

Every year in autumn the *Bremer Freimarkt* takes place. It is a fair in Bremen and considered to be the largest fair in Northern Germany. Besides multiple amusement rides and roller coasters, there are also various food stands, including sweets stands. The latter usually have a very long counter, where miscellaneous sweets are displayed. Each type of candy belongs to its own part of the display. When a customer wants to buy a certain sweet, at least one of the two vendors has to move to that part of the counter. The customer also has to move to that location. After arriving there, the vendor gives the customer its requested candy and stays at that place until the next customer appears. While one vendor is serving the customer, the other one is also allowed to move to another location. Assume that only one customer can be served at a time at the stand, and that a customer buys only one type of sweet. The corridor where the vendors walk is very narrow, hence they cannot pass by another.

In contrast to other fairs, the vendors get paid per day according to the distance they walked on that day. To prevent the vendors from exploiting this, the owner of the stand specifies rules how the vendors have to behave based on their current positions when a certain candy is requested. Since this payment depends heavily on the overall number of customers of the stand, and the fair wants equality for all sweets stands, it supports the owners by a certain amount of money at the end of each day depending on the number of customers the stand served. This value is equal to the distance the vendors would have had walked if they had moved optimally, that is, the behavior of the vendors with the minimal walking distance in total if they had known the sequence of candy requests of the day in advance. Obviously, the owner tries to maximize his revenue and therefore wants to minimize the ratio between the payments of his vendors and the money from the fair. Among the owners of the sweets stands, this ratio is commonly known as *payment ratio*.

The fair decided that every owner has to publish his rules. However, Alice, a very malicious and intelligent owner, wants to overthrow the other owners. She analyzes the rules of a certain owner in the morning, and then buys a special sequence of candies over the whole day at the other owner's stand while blocking all customers. The sequence is composed in a way that maximizes the other owner's payment ratio for that day.

At some time, Bob, one of the sweets stand owners, realizes Alice's bad intentions and hence hires a snitch. The snitch shall spy on her every morning and reveal the sequence of candies she is going to buy on that day. Bob also adjusts his rules so that they depend on the snitch's information and thus can improve his payment ratio compared to his previous strategy. However, Bob does not know whether the information are correct or not. Furthermore, there is the possibility that the snitch is actually a double agent and works for Alice, hence provides information which make Bob's rules perform even worse. Since the only tool to affect his payment ratio is the design of the rules, the main question of Bob is how to specify the rules and how to prove that these are good.

We call this problem the *sweets-stand-problem*. In this thesis, we analyze the problem in a more abstract setting. However, all results are applicable for Bob, and hopefully help him to rescue his stand. One part of this abstraction is the so-called *k-server problem on the line*, where k distinct servers have to move to requested locations while minimizing the total distance moved. We can imagine that the vendors are the servers and the requests the locations of the requested candy. The other part concerns the role of the snitch. In the abstract setting, we consider a machine-learned model which gives us predictions on the requests. This idea arises from the progress of machine-learning in the last years which demonstrates the capabilities of it in many areas. Yet one of its biggest problems is that there is in general no *theoretical guarantee* that the predictions of a machine-learned model achieve a certain quality. Thus, we say that those are *untrusted*. The goal is to design an algorithm which uses those and for that we can still prove theoretical guarantees about the overall solution of the associated k -server instance, *i.e.* a bound on the payment ratio in the sweets-stand-problem. We call this abstraction the *k-server problem on the line with untrusted predictions*.

Several questions appear in the sweets-stand-problem. One of them is how Bob has to set his rules in the best possible way without having a snitch. In fact, this question consist of two questions, namely how to design the rules itself and how to argue that these are the best possible. We give a detailed answer to these questions in Chapter 2, and see that the best possible rules ensure a payment ratio of at most the amount of vendors. This also answers the question of what happens if Bob hires more than two vendors for his sweets stand in this setting. We also introduce the k -server problem in this Chapter and give an overview on the history and different approaches for it.

We can ask similar questions when we assume having a snitch. However, we additionally have to consider different qualities of information the snitch delivers. In Chapter 3 we study this setting with two vendors and illustrate that the payment ratio is at most $1 + \lambda$ if the information is correct, and $1 + \frac{1}{\lambda}$ if it is completely wrong or the snitch is a double agent for any λ between zero and one. By setting the value of λ , Bob can affect how much he trusts the snitch.

Afterwards, in Chapter 4 and Chapter 5 we look at different generalizations of that setting. In the former we consider having an arbitrary number of vendors and in the latter we analyze other structures than sweets stands, which generalize these and where similar problems can be formulated. We will see that we can naturally extend some results of the two-vendors case to these problems, including the result for having correct information. However, we encounter problems when arguing about the payment ratio if the snitch is a double agent.

Since some customers may cannot follow the vendor to the requested part of the counter, the vendors have to move to a second location after collecting the requested candy to give it to the customer. We consider a generalization of this modified problem in Chapter 6, which is called *k-taxi problem*, and demonstrate for having two vendors that with correct spy information the payment ratio is at most $3 + 4\lambda + 2\lambda^2$ whereas it is not more than $3 + \frac{4}{\lambda} + \frac{2}{\lambda^2}$ for poor information for any λ between zero and one.

Lastly, in Chapter 7 we simulate the problem with some of those rules, and empirically evaluate their performance for real world applications. But before we can do all of this, we have to introduce several concepts, tools and techniques in the following Chapter.

Chapter 1

Preliminaries

In this Chapter we introduce necessary definitions and techniques for the remaining thesis. We start by taking a closer look at classical online problems in Section 1.1, and illustrate different techniques to make arguments about the performance of algorithms for them in Section 1.2 and Section 1.3. The definitions for these three sections are based on [10]. Then, we outline ideas from the field of online algorithms with untrusted predictions in Section 1.4. Since this concept just recently emerged, we give in Section 1.5 an overview about related work and the results which had been established yet.

1.1 Online algorithms

In computer science and various other fields, many problems appear naturally with an uncertainty about the information we need to optimally solve them. One possible form of uncertainty is, that this information is separated over time. When a chunk of information arrives, we have to instantly execute some action which partly solves the problem, but can *not* be reverted. More formally, we have given a *sequence of requests*, and for each request we have to make a *decision*. We start this process by obtaining the first request of the sequence, and only get the next request if we have decided the current one. We can assume that each decision participates in some measurable and comparable *outcome* for the whole sequence, which enables us to compare different solutions for this kind of problem. This rough idea emerges the question of how we can determine the quality of a solution for such a problem. Before answering this question, we introduce general *optimization problems*.

Definition 1. An optimization problem $\mathcal{P}$ consists of a set $\mathcal{I}$ of possible *inputs* (also called *instances*) and a positive *cost function* c , where every input $I \in \mathcal{I}$ has a set of possible *solutions* $F(I)$, and every solution $O \in F(I)$ for an input I has associated cost $c(I, O)$.

For every input I of a given optimization problem $\mathcal{P}$ we can ask which solution is the one with minimal cost. To give a clear answer to this question we define the *optimal algorithm* or *optimal solution* for an input I .

Definition 2. For an optimization problem $\mathcal{P}$ with cost function c , the optimal algorithm OPT maps every input $I \in \mathcal{I}$ to the cost of its optimal solution, *i.e.*

$$\text{OPT}(I) = \min_{O \in F(I)} c(I, O).$$

If there is only a single instance I in the current context, we denote the cost of its optimal solution by OPT . Note that it does not matter if we want to maximize or minimize over some function, since we can obtain the opposite by negating the function.

Recall the setting from the beginning of this Section where we have given a sequence of requests over time. Observe that this problem can be seen as an optimization problem, where the input is the sequence of requests and a solution is the sequence of decisions for those. We call such optimization problems, where the input appears in sequential parts and the output has to be generated as soon as a part of the input arrives, *online problems*. Algorithms which compute a solution for an online problem are therefore called *online algorithms*. In contrast, optimization problems where the whole input is given at once are called *offline problems* with associated *offline algorithms*.

The *ski-rental problem* is an easy to understand rent-or-buy online problem. Assume that we want to ski for as many days as it snows, which number is unknown in advance. Since on the first day we do not own skies, we have to decide if we buy skies for a fixed price $b > 0$ and use them for the rest of the trip, or if we rent skies for a single day at price one. Our goal is to minimize the cost we pay for skiing. Formally, as input we receive a sequence of consecutive days with information about if it is going to snow on that day. For each day we have to decide whether we rent or buy skies. Note that this explains the name of this type of problems. The sequence ends on a day when it does not snow. Let n be the number of snowing days. The input sequence is composed of n requests with information that it snows on that day, and a final request which provides that it does not snow on the $n + 1$ th day. Clearly, an optimal solution is to buy skies on the first day if $n > b$ and rent otherwise. Thus, $\text{OPT} = \min(n, b)$. We can observe that any algorithm for the ski-rental problem rents skies for $i - 1$ days and buys them on day i for some $i \geq 0$. Hence, we denote this algorithm by ALG_i .

Every online problem can be transformed into an offline problem by removing the online setting of the input, *i.e.* we obtain all requests of the input at once, and we can compute the decisions for the requests without specific order. In the associated offline setting of the ski-rental problem, we would simply receive the number of days it is snowing, *i.e.* n , and hence could easily compute the optimal solution, that is, $\min(n, b)$. This must not be the case the other way around, since there can be inputs of general offline problems which we cannot turn into a meaningful sequence of requests. In the

next Section we will see how to use this concept to define an indicator of the quality of an online algorithm.

1.2 Adversaries and competitive analysis

This Section builds up on the previous one, where we defined the notion of an online problem and saw, that we can turn an online problem into an offline problem. Hence, there exists an optimal algorithm OPT for that online problem. Since the optimal algorithm looks at the offline setting of the problem, we call solutions computed by OPT *offline solutions*. This enables us to define an efficiency indicator for an online algorithm, *i.e.* a value which describes how good an online algorithm performs in terms of charged cost compared to the offline solution.

Definition 3. Let $\mathcal{P}$ be an online minimization problem with an optimal algorithm OPT , and let ALG be an online algorithm for $\mathcal{P}$. For an input I , we denote the cost of ALG for solving I by $\text{ALG}(I)$. We say ALG is *c-competitive* if $c \geq 1$ and there is a constant a such that for all finite inputs I ,

$$\text{ALG}(I) \leq c \cdot \text{OPT}(I) + a.$$

If an online algorithm is *c-competitive*, we also say that the algorithm attains a *competitive ratio* of c . If c is independent of the instance I , we say that the algorithm is *competitive*. Note that c still can be a function of some problem parameters such that the algorithm is competitive. If a in Definition 3 is less or equal than zero, we call the algorithm *strictly (c-)competitive*.

Let us go back to the ski-rental problem. Let n be the number of skiing days and b the price of the skies. Recall that ALG_b is the algorithm which rents for $b - 1$ days and buys on the b th day. In the last Section we saw that $\text{OPT} = \min(n, b)$. If $n \leq b - 1$, the algorithm only rents and hence pays the same as the optimal solution does, *i.e.*

$$\text{ALG}_b = n = \text{OPT},$$

but if $n \geq b$, the algorithm first rents for $b - 1$ days and then buys on day b , whereas, the optimal solution buys on the first day with cost b . Thus,

$$\text{ALG}_b = \underbrace{b - 1}_{\text{rent}} + \underbrace{b}_{\text{buy}} = 2b - 1 = \left(2 - \frac{1}{b}\right) \cdot b = \left(2 - \frac{1}{b}\right) \cdot \text{OPT}.$$

We conclude that ALG_b has a competitive ratio of $2 - \frac{1}{b}$. This means that for any instance, *i.e.* any number of skiing days n , ALG_b pays at most $2 - \frac{1}{b}$ times the cost the optimal solution pays.

It is a common way to figure the process of evaluating algorithms as a game between two players. In online optimization, one player is an online algorithm, and the other player the so-called *adversary*. The game works as follows. The algorithm publishes a strategy on how it processes a given instance. Based on that, the adversary creates an instance of the problem, where the algorithm's strategy behaves worst with respect to the competitive ratio. Since the adversary knows the whole input sequence, she also knows the optimal offline solution for it, and that enables her to compute the competitive ratio. Due to this property, the adversary is sometimes also called *offline player*.

Consider again the ski-rental example. An adversary knows that ALG_b rents for $b - 1$ days and buys skies on day b for cost b . If it rents on day b , there only would be cost 1 for that day. Hence, the worst thing that could happen for ALG_b is that there are exactly b skiing days, because the purchase of the skies on the b th day does not pay off, since the algorithm cannot save money from amortizing future rents. Of course, the adversary knows that, and therefore creates an instance with b skiing days. The adversary could then decide to buy the skies on the first day or rent them for b days, both for cost b . ALG_b first pays $b - 1$ for renting and then b for buying with total cost equal to $2b - 1$. This gives the exact competitive ratio of $2 - \frac{1}{b}$ for that instance. Above we have already seen that $2 - \frac{1}{b}$ is an upper bound on the competitive ratio of ALG_b . Since we have found an instance with exactly that competitive ratio, we showed that our competitive analysis of ALG_b is *tight*. The *tightness* of a competitive analysis ensures that there is no chance to prove a better competitive ratio for that specific algorithm, because there already is a concrete example instance where that competitive ratio is met. In other words, it illustrates that our analysis of the algorithm was the best possible.

1.3 Lower bounds

The competitive ratio of a concrete instance for a given online algorithm is a special form of a *lower bound*. It shows that the *upper bound*, *i.e.* the competitive ratio we established in the analysis of the algorithm, cannot drop below the lower bound. This form of lower bound always refers to a *specific* algorithm of a problem, not to the problem itself. A *general lower bound* for an online problem gives a lower bound on the competitive ratio of *any* online algorithm for the problem. Then, if we prove a competitive ratio for an algorithm which matches the lower bound, we can conclude that this algorithm is the best possible algorithm for the online problem with respect to the competitive ratio. Note that this also implies that the competitive analysis of that algorithm is tight.

We can also translate this idea to the adversarial game. The difference is, that we now do not propose a concrete algorithm for the problem, but an *arbitrary* algorithm for the problem. Then, if the adversary is able to create an instance for that algorithm, we can

analyze the algorithm's performance on that instance and obtain a lower bound on the competitive ratio.

Applying this technique to the ski-rental problem enables us to select the best possible algorithm for it. Consider algorithm ALG_i . As we have seen above for ALG_b , the worst case for algorithm ALG_i is when there are exactly i skiing days. Thus, an adversary creates an instance with i skiing days and pays $\min(i, b)$. It remains an analysis of the performance of ALG_i for that instance. Hence, if $i \leq b$,

$$\frac{\text{ALG}_i}{\text{OPT}} = \frac{i - 1 + b}{\min(i, b)} = \frac{i - 1 + b}{i} = 1 + \frac{b - 1}{i} \geq 1 + \frac{b - 1}{b} = 2 - \frac{1}{b}.$$

Similarly, if $i \geq b$,

$$\frac{\text{ALG}_i}{\text{OPT}} = \frac{i - 1 + b}{\min(i, b)} = \frac{i - 1 + b}{b} = 1 + \frac{i - 1}{b} \geq 1 + \frac{b - 1}{b} = 2 - \frac{1}{b}.$$

In other words, for each $i \geq 0$ the adversary finds an instance such that the competitive ratio of ALG_i is at least $2 - \frac{1}{b}$. Since we proved the same value as upper bound for ALG_b , we conclude that ALG_b is the best possible algorithm for the ski-rental problem and that our analysis is tight.

1.4 Untrusted predictions

In the last couple of years, machine learning became a very active research area in both theory and practice. Recently, there grew an interest in improving online algorithms using machine learning. Basically, given an online problem, the idea is to use a machine-learned model that produces *predictions*. Since these can have any *type*, *e.g.* predictions on the online input or on the behavior of an optimal algorithm, we have to introduce a *prediction model* which exactly defines of which type the predictions are. The key is to make no assumptions on the machine-learning algorithm itself, *i.e.* see it as black-box which produces predictions of a certain type. If we think back to the adversarial game, we can see this entity as a player, the so-called *predictor*, who gives the algorithm a prediction on what the adversary will do. But since in general the predictor does not know for sure what the adversary will do, we also call its predictions *untrusted*. The only measurement of the quality of predictions is some kind of *error*, which is a non-negative value and can be computed from the prediction and the ground truth, *i.e.* the instance or its optimal solution. The error depends on the prediction model, or said differently, the prediction model has to ensure that we can compute some kind of error for a prediction. There also can be multiple types of errors for single prediction model.

Given an online problem and an associated prediction with error η , the task is to design an algorithm which uses the prediction in the best possible way. Note that such an algorithm does not know the error of a prediction. In order to compare different

algorithms using predictions we have to compare an efficiency indicator of them. It turns out that this is possible for algorithms with predictions, but it is not as simple as for classical online algorithms. In fact, we have to consider different perspectives based on the quality of predictions. The most obvious way is to let the competitive ratio depend on the error of the prediction η , *i.e.* a function $f(\eta)$. But since we make no assumptions on the prediction quality, it might be that the error is very large, and hence the competitive ratio might be really bad. Thus, we want to make sure that our algorithm is not arbitrarily bad for arbitrarily bad predictions.

Lykouris and Vassilvitskii [27] observed the above problems and proposed the concept of *consistency* and *robustness* for online algorithms with predictions.

Definition 4. Let A be an online algorithm using predictions with a competitive ratio equal to $f(\eta)$, where η is the error of the prediction. We call A α -consistent if $f(0) = \alpha$ and β -robust if $f(\eta) \leq \beta$ for all $\eta \geq 0$.

We say that an online algorithm is (α, β) -competitive, if it is α -consistent and β -robust. Put into words, consistency describes how well an algorithm performs if it gets the best possible prediction in our setting, whereas robustness ensures that the algorithm cannot get too worse if it has given arbitrary bad predictions. In the context of the adversarial game we can imagine that consistency refers to the case where the predictor is aware of what the adversary will do with respect to the prediction error and tells the algorithm that. When considering robustness, we can think of the predictor and the adversary maliciously working together and giving the algorithm the worst possible pair of instance and prediction.

Purohit *et al.* [32] apply this idea to the ski-rental problem. For $\lambda \in [0, 1]$, they predict the number of skiing days y , and then buy on day $\lceil \lambda b \rceil$ if $y \geq b$ and otherwise on day $\lfloor b/\lambda \rfloor$. The prediction error is simply the absolute difference between the real number of snowing days and y . This algorithm is $(1 + \lambda)$ -consistent and $(1 + \frac{1}{\lambda})$ -robust algorithm for any $\lambda \in [0, 1]$. To prevent an extra case for $\lambda = 0$, we introduce the convention that $1 + \frac{1}{\lambda} = \infty$ for $\lambda = 0$, which means that the algorithm is *not* robust in this case. We will use this convention throughout the thesis. Note that this algorithm actually is a *family of algorithms*, *i.e.* an algorithm for each possible value of λ .

Since we now have multiple efficiency indicators for online algorithms with predictions, we can design algorithms for different requirements, *e.g.* we want an algorithm to be very robust, or to be close to optimal consistent. Note that this makes it difficult to argue that an online algorithm using predictions is the *best* algorithm for that problem, as we have seen for classical online algorithms. There, we used lower bounds to determine that an online algorithm has the best possible competitive ratio. In the setting with predictions, this is also possible, but we have to consider multiple efficiency indicators.

An approach towards this question is to look at how both consistency and robustness behave to each other for any algorithm of the problem. We can observe that for many

problems there exists a trade-off between those indicators, *i.e.* the better the consistency of an algorithm becomes, the worse the robustness gets. We can observe this behavior in the family of algorithms for the ski-rental problem, where for smaller values of λ the consistency approaches 1 whereas the robustness becomes arbitrarily large. As this example illustrates, there might be multiple algorithms for a problem which achieve a certain trade-off between consistency and robustness, but with different concrete values for these indicators. Hence, we might want to look at a family of algorithms and show that their members achieve the best possible trade-off. Angelopoulos *et al.* [2] introduced this approach and called a family of algorithms *pareto-optimal*, if they achieve the best possible trade-off between consistency and robustness. The following definitions are based on their work.

Definition 5. Let A and B online algorithms which are (α_A, β_A) and (α_B, β_B) competitive. Then A *dominates* B if $\alpha_A \leq \alpha_B$ and $\beta_A \leq \beta_B$.

Definition 6. A family of algorithms $\mathcal{A}$ is *pareto-optimal* if for every online algorithm B there exists $A \in \mathcal{A}$ such that A dominates B .

Note that in the setting of classical online algorithms, a c -competitive algorithm is (c, c) -competitive, and hence is pareto-optimal if it achieves the best possible competitive ratio for a problem, *i.e.* if c is provable the best possible competitive ratio for any online algorithm of the problem. Thus, the concept of pareto-optimality naturally extends the concepts of classical online algorithms for online algorithm with predictions.

Finally, we are ready apply this idea to the ski-rental problem. Angelopoulos *et al.* [2] propose a slight modification of the approach of Purohit *et al.* [32]. They define a family $\mathcal{A}$ of algorithms A_k for $1 \leq k \leq b$ such that they get a single bit prediction on whether to buy on day k or on day $b - 1$. They demonstrate that A_k is $(1 + (k - 1)/b)$ -consistent and $(1 + (b - 1)/k)$ -robust. Furthermore, they prove that $\mathcal{A}$ is pareto-optimal, *i.e.* any $(1 + (k - 1)/b)$ -consistent algorithm for the ski-rental problem using predictions is at least $(1 + (b - 1)/k)$ -robust.

1.5 Further related work

The field of online algorithms with predictions just recently emerged and built up several concepts and results. The first approach is considered to be from Medina and Vassilvitskii [29], who improve revenue maximization in auctions using machine-learned predictions. Lykouris and Vassilvitskii [27] illustrate how to add a similar concept to the online caching problem. They predict the number of requests between two requests of the same element and thereby enhance the competitive ratio of existing algorithms. They also introduce the idea of *consistency* and *robustness*. Based on that model, Purohit *et al.* [32] propose algorithms using machine-learned predictions for the ski-rental and non-clairvoyant job scheduling problem and show that those can be configured in a

way that they can become arbitrarily optimal if they get good predictions. Gollapudi and Panigrahi [18] improve this idea by adding multiple predictors to the ski-rental problem, whereas Angelopoulos *et al.* [2] slightly modify the initial ski-rental algorithm in order to prove pareto-optimality, as we have seen in the last Section. The latter also propose algorithms with predictions for the list update and bin packing problem, and demonstrate how to show lower bounds for algorithms with predictions.

Further approaches for different types of scheduling problems with machine-learned advice also emerged. Lattanzi *et al.* [26] break the lower bound for deterministic algorithms for the makespan minimization problem with restricted assignments using a prediction-based algorithm, while Mitzenmacher [31] predicts the processing times for jobs in different scheduling scenarios and introduces in this context the *price of misprediction*.

Kumar *et al.* [24] propose a similar model which they call *semi-online*. They assume that the input sequence of the online problem is composed of unknown and previously known elements, and obtain almost optimal results for the online bipartite matching problem using that model. In a subsequent work [25] they apply the semi-online model to the caching problem, where the request sequences are generated by paths on one or multiple access graphs, which are previously known to the algorithm, and prove tight bounds which depend on structural properties of those graphs.

In another line of research on the caching problem, Rohatgi [35] improves the initially proposed algorithm of Lykouris and Vassilvitskii and shows superior upper and lower bounds for it. Based on their work, Antoniadis *et al.* [3] introduce online algorithms using predictions for the online matching on the line problem and for general metrical task systems, for which the online caching problem is a special case.

Classical data structures also receive advances by the usage of predictions, as Kraska *et al.* [23] show for the indexing problem and Mitzenmacher [30] for bloom filters.

Finally, we want to mention that researchers use different terms to describe the kind of extra information, *e.g.* *machine-learned advice* [27, 35], *machine-learned predictions* [32], *untrusted advice* [2], *untrusted predictions* [3], *expert advice* [18], *approximate predictions* [29] or by adding the phrase *learned* to some part of the problem [23, 26, 30]. There are also different names for algorithms using those extra information, *e.g.* *prediction-based/prediction-augmented algorithm* [3], *learning-augmented algorithm* [25, 35] or *oracle-based algorithm* [27].

In this thesis, we stick to the phrase *untrusted prediction*. One reason for this is the fact that we cannot trust the predictor as mentioned in the previous Section. We also want to avoid the phrase *advice*, since it also refers to the field of *advice complexity* which is a different technique of improving online algorithms, see *e.g.* [9].

Chapter 2

The k -server problem on the line

In this Chapter we address the k -server problem, which was introduced by Manasse *et al.* in [28]. It is the problem of sequentially serving *requests* with k distinct *servers* in a metric space. There are both online and offline variants of the problem regarding the sequence of requests. The k -server problem on the line is a special case where the metric space is the real line.

In Section 2.1 we give a formal definition of the problem. Afterwards, we outline the history of the problem and variations of it. Then, we picture in Section 2.3 an optimal algorithm for it and prove a general lower bound in Section 2.4, before we switch to the restricted case in one dimension and analyze different approaches for the online k -server problem on the line in Section 2.5 and Section 2.6.

2.1 Problem definition

This Section is dedicated to all necessary definitions for the k -server problem. Since the k -server problem is based on a *metric space*, we start by defining it.

Definition 7. A *metric space* is a tuple $(\mathcal{M}, d)$ such that $\mathcal{M}$ is a set of points and d is a *metric* on $\mathcal{M}$, *i.e.* a function $d : \mathcal{M} \times \mathcal{M} \rightarrow \mathbb{R}_{\geq 0}$ that satisfies for all $x, y, z \in \mathcal{M}$

- 1) $d(x, x) = 0$,
- 2) $d(x, y) = d(y, x)$ and
- 3) $d(x, y) \leq d(x, z) + d(z, y)$.

The second property in Definition 7 is also known as *symmetry* and the third as *triangle inequality*. Let $(\mathcal{M}, d)$ be a metric space and $x_1, \dots, x_k \in \mathcal{M}$ be points in that space. We call $x_1, \dots, x_k$ *servers* and by x_i we denote the servers name as well as its position in space. A *configuration* $C = (x_1, \dots, x_k)$ is a snapshot of the positions of the servers at a fixed point in time. The *distance* $d(C, C')$ between two configurations $C = (x_1, \dots, x_k)$ and $C' = (x'_1, \dots, x'_k)$ is the sum of the distances a server moved

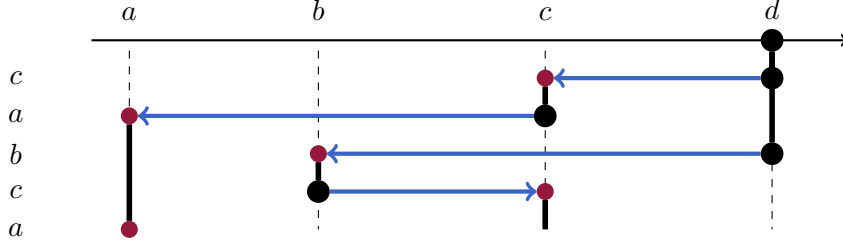


Figure 2.1 An example schedule with two servers and five requests.

between both configurations in $\mathcal{M}$, *i.e.*

$$d(C, C') = \sum_{i=1}^k d(x_i, x'_i).$$

An *instance* I for the k -server problem consists of two parts. The first one is a finite sequence of requests $r_1, \dots, r_n \in \mathcal{M}$, where each r_t denotes the name of the request as well as its position in $\mathcal{M}$. The second part is the initial configuration C_0 of the servers. For the remaining thesis, we use C_0 to refer to the initial configuration in the context of the k -server problem. Given that, the goal of the k -server problem is to find a sequence of configurations $C_1, \dots, C_n$, also called *schedule*, such that for each request r_t at least one server of configuration C_t serves it, *i.e.* $r_t \in C_t$, and that it minimizes the total cost for moving the servers, *i.e.*

$$\sum_{i=1}^n d(C_{i-1}, C_i).$$

In the whole thesis we are going to use a normalized visualization for schedules of the k -server problem on the line as shown in Figure 2.1. Fixed points on the line are marked by vertical dashed lines and are labeled on top by lower case letters. On the left we denote the sequence of requests from top to bottom which are selected from those fixed points on the line. Each request defines a step in time. Black circles are servers which get moved at the given step. On the top line they picture the initial configuration. Thick black vertical lines between two steps indicate that on these positions is at least one server. Red circles illustrate the position of the requests on the line. Horizontal arrows starting at servers display moves, and we will later use different colors to distinguish various algorithms and players.

There are two settings for the k -server problem: the *offline* and the *online* setting. In the former, the whole instance, *i.e.* the initial configuration and the sequence of requests, is known in advance. In the latter, we primarily obtain the initial configuration and the first request, and then get the next request only if we served the previous one.

Finally, we want to discuss two properties of algorithms for the k -server problem. An online algorithm for the k -server problem is *memoryless* if it moves its servers only based on the current configuration and the next request. The intention of this definition is, that

a memoryless algorithm has no knowledge of the history of requests, and hence cannot make its decisions based on it. However, as Koutsoupias [20] points out, this definition is problematic in certain spaces such as the euclidean spaces. There, a memoryless algorithm can encode the history of requests by only slightly moving its servers in a particular way, and then make its moves based on that. To overcome this issue, we also have the property of *laziness* for algorithms for the k -server problem. An algorithm is said to be *lazy* if it only moves the single server which directly serves the current request. Koutsoupias [20] demonstrates that we can postpone non-lazy moves of a non-lazy algorithm until they are necessary and hence lazy, and therefore can simulate every non-lazy algorithm using a lazy algorithm. Due to the triangle inequality of metric spaces, we do not pay additional cost by doing that. Thus, we can fix the definition of *memorylessness* by implicitly requiring that a memoryless algorithm has to be lazy.

2.2 A history of the problem, applications and generalizations

This Section gives an overview of the history of the k -server problem and of several algorithms and applications for it. A more in-depth, but not recent outline of the history of the k -server problem and its variations can be found in [20].

As already mentioned, the problem was introduced by Manasse *et al.* [28] in 1988. In their work, they showed that any deterministic algorithm for the k -server problem on a metric space with at least $k + 1$ points has a competitive ratio of at least k . We examine this result in Section 2.4. Due to this lower bound, they conjectured that there exists an online algorithm for the k -server problem which is k -competitive on any metric space with at least $k + 1$ points. Towards the proof of this conjecture, they proposed a 2-competitive algorithm for $k = 2$, but were unable to give results for a larger number of servers.

Soon after the conjecture was published, Chrobak *et al.* [13, 14] proved that it holds for certain types of metric spaces, namely the real line and trees. We extensively look into the former result in Section 2.6, and use techniques of the latter in Chapter 5.

Although Manasse *et al.* proposed the conjecture, there was no evidence that there even exists any competitive algorithm for $k > 2$ on general metric spaces. This has later been proved by Rabani *et al.* [33], who showed that there exists a competitive algorithm for the k -server problem where the competitive ratio depends exponentially on k . In 1994, this result was dramatically improved by the *Work Function Algorithm*, for which Koutsoupias and Papadimitriou [21] proved a competitive ratio of $2k - 1$. The Work Function Algorithm is a natural online algorithm based on dynamic programming which decides its moves based on all previously seen requests, thus is not memoryless.

Another important approach towards the k -server conjecture was the utilization of randomization for online algorithms. Raghavan and Snir [34] proposed the *Harmonic Al-*

gorithm, a simple memoryless randomized algorithm, and proved that it is $(k(k+1)/2)$ -competitive for metric spaces with exactly $k+1$ points. The upper bound on the competitive ratio of memoryless randomized algorithms on metrics spaces with exactly $k+1$ points was later improved to k by Coppersmith *et al.* [16]. For general metric spaces, Grove [19] showed that the Harmonic Algorithm is competitive and bounded its competitive ratio by $\mathcal{O}(k2^k)$.

There are also lower bounds on the competitive ratio of randomized algorithms. Fiat *et al.* [17] demonstrated that in uniform metric spaces, *i.e.* the metric space where all distances are equal, the competitive ratio for randomized algorithms is $\Theta(\log k)$. For general metric spaces with at least $k+1$ points, Bartal *et al.* [7, 8] proved the currently best known lower bound of $\Omega(\log k / \log \log k)$ on the competitive ratio of randomized algorithms for the k -server problem. Bansal *et al.* [4] proved the existence of a randomized algorithm which is $\tilde{\mathcal{O}}(\log^3 n \log^2 k)$ -competitive on general metric spaces with n points. This improves the $(2k-1)$ -bound of the deterministic Work Function Algorithm for the case where n is sub-exponential in k . For general instances, the Work Function Algorithm is still considered as the best performing algorithm for the k -server problem, even if randomization is allowed [4]. This shows that for randomized algorithms there is an exponential gap between the best known lower and upper bounds. Thus, the *randomized k -server conjecture* was established and says that there exists a $\mathcal{O}(\log k)$ -competitive randomized algorithm for the k -server problem for every metric space [20].

The k -server problem is a generalization of various other online problems. The most noticeable one is the *paging problem*, where we have given a set of m pages and a cache which can hold up to k pages in it. Each time a page is requested and not in the cache, we have to load it into the cache by paying a fixed cost, while possibly evicting another page in the cache. We can simply reduce the paging problem to the k -server problem by using a uniform metric space of size m , as illustrated in [28].

The paging problem can also be seen as the *simple* variant of the *caching problem*. In the *general caching problem*, each page has, in contrast to the fixed cost in the paging problem, its own associated cost, which an algorithm has to pay if it loads the page into the cache. Raghavan and Snir [34] showed that the general caching problem can also be reduced to the k -server problem.

For the CNN problem, which is a generalization of the k -server problem where the servers have to move to certain horizontal or vertical lines in a grid, Koutsoupias and Taylor [22] proposed a competitive algorithm for two servers. This problem is a special case of the so-called *general k -server problem*, where each server moves in its own metric space, and a request is a tuple of k requests, one from each metric space. Then, for a given request, at least one server has to serve the request of its metric space. For $k > 2$, there were no competitive algorithm known for this problem, until recently, on the one hand Bansal *et al.* [6] proved the existence of a competitive algorithm for uniform metric

spaces, which almost matches the best known lower bound established in [5], and on the other hand Chiplunkar and Vishwanathan [11] demonstrated the competitiveness of a randomized algorithm for weighted uniform metric spaces.

2.3 The optimal offline solution

When initially introducing the k -server problem, Manasse *et al.* [28] presented a procedure to compute the optimal offline solution for a given instance. Their approach was based on dynamic programming and not very efficient for instances where the size of the metric space is way larger than the number of requests. Thus, Chrobak *et al.* [13] proposed a more efficient and clearer algorithm, which searches a minimum cost flow in an acyclic flow network, and proved the following result.

Theorem 1 (Chrobak *et al.*, [13]). *There exists a deterministic offline algorithm for the k -server problem which computes an optimal schedule for an instance with n requests in time $\mathcal{O}(kn^2)$.*

Modelling optimization problems with flow networks is a common technique, since those networks can be nicely visualized and there exist various algorithms for finding a minimum cost flow in such a network [1].

We now demonstrate how to construct such a flow network for a given instance, which informally proves Theorem 1, and refer to the visualization of the network in Figure 2.2. Let I be an instance with initial configuration $C_0 = (s_1^0, \dots, s_k^0)$ and requests $r_1, \dots, r_n$. All edges have capacity one. We have two vertices s and t , k vertices $s_1, \dots, s_k$ for each server, and two vertices r_i and r'_i for each request r_i . Between r_i to r'_i is an edge with cost $-K$, where K is a large integer. There are edges from s to each server vertex s_i with cost zero and from each vertex r'_i to t with cost zero. There also exists an edge from s_i to t with cost zero. A solution uses the latter if that server does not serve any request. The remaining edges are the ones which are colored blue and brown. The blue edges are used for the first request a server serves. Since at that point the server is located at its initial position, an edge from s_i to r_j has a cost equal to $d(s_i^0, r_j)$. The brown edges are used by servers which are at a certain request r_i and want to serve another request r_j . Clearly, $i < j$, and the cost for that edge is equal to $d(r_i, r_j)$. Note that Figure 2.2 does not show all edges to make it more clear.

Solving this flow network yields a solution with k units of flow, and since all capacities are equal to one, and the solution is composed of k internal vertex-disjoint paths from s to t . Each of those paths indicate the movements of a distinct server. An example of a minimum cost flow with corresponding schedule is displayed in Figure 2.3. Note that this solution is optimal and moves exactly one server per request. Hence, we can assume that there always exists an optimal algorithm which is lazy. If we use the real line as metric space, there can be optimal solutions where a server s_i overtakes another server s_j .

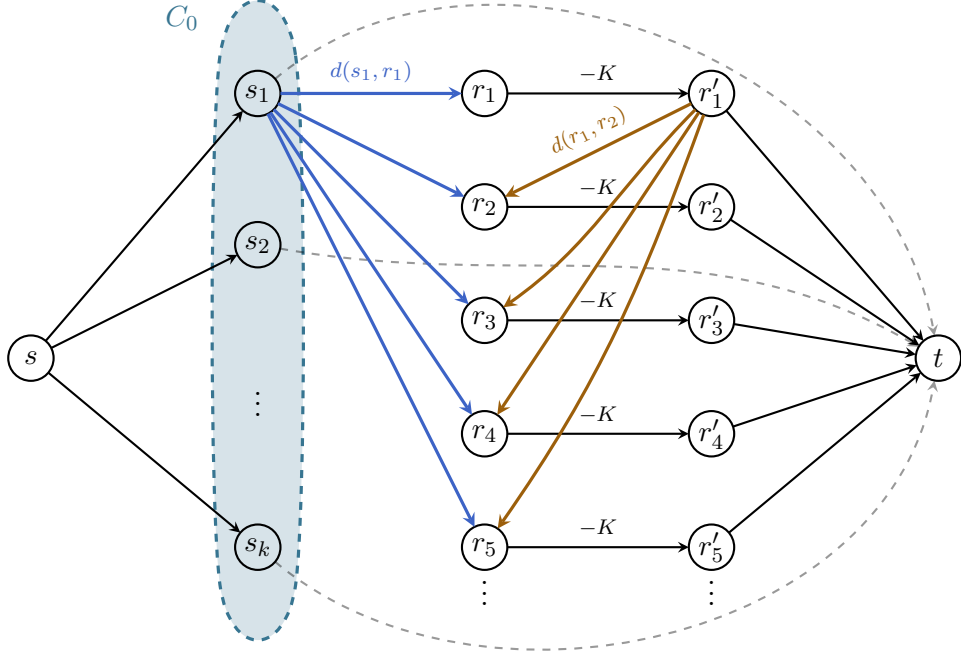


Figure 2.2 A flow network for solving a k -server instance. All edges have capacity one, and, if not indicated, cost zero.

If that is the case, there has to exist another solution without overtaking, because the request could have been served by s_j with lower or equal cost due to lazyness. Therefore, considering the real line as metric space, we will assume that an optimal solution never moves a server by another.

2.4 A lower bound

For the k -server problem, Manasse *et al.* [28] proved a lower bound on the competitive ratio equal to k , if the metric space is large enough, *i.e.* there are more points than servers. Note that otherwise a solution is trivial. We demonstrate their proof for that statement in this Section.

Theorem 2 (Manasse *et al.*, [28]). *For any metric space with at least $k + 1$ points, there is no deterministic online algorithm for the k -server problem with a competitive ratio less than k .*

Proof. Let A be an online algorithm and $(\mathcal{M}, d)$ a metric space with at least $k + 1$ points. Let $p_1, \dots, p_{k+1} \in \mathcal{M}$ such points and let the initial configuration of the servers $s_1, \dots, s_k$ be on k of those points. The sequence of request is adaptively defined by the algorithms

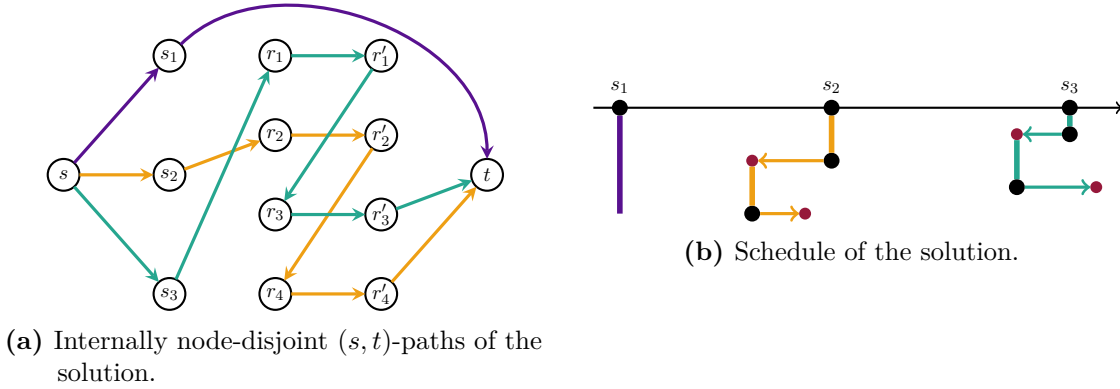


Figure 2.3 An example for the same solution computed in the flow network and on a schedule. Each server has its own color.

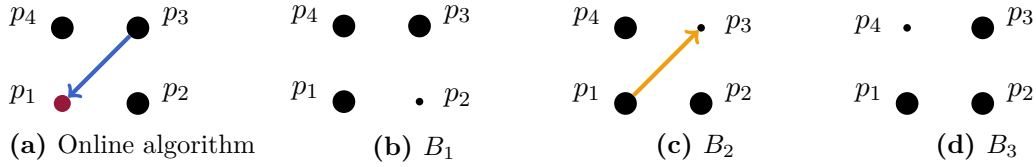


Figure 2.4 An example of the invariant in the proof of Theorem 2 for $k = 3$: when the online algorithm serves the request on p_1 with the server on p_3 , the offline algorithm B_3 will move its server from p_1 to p_3 such that all configurations are distinct. The costs for both moves are equal.

moves. Given a configuration, the next requests is the single free point of our fixed $k + 1$ points.

Now consider k offline algorithms $B_1, \dots, B_k$. Each of those algorithms has its own k servers. Since there are $k + 1$ fixed points, there are $k + 1$ possible configurations for k servers on those points. Thus, we maintain the following invariant. The configurations of $B_1, \dots, B_k$ and A are all different. We now argue that the invariant holds. Since the servers of all offline algorithms have to start at the same initial configuration as A , we have to bring them to their distinct configurations in the beginning to satisfy the invariant. We denote this initial cost for an offline algorithm B_i by $q(B_i)$. Note that those costs only depend on the initial configuration. If a request r_t arrives, A has to move one of its servers s_i from some point p_j to r_t with cost $\text{ALG}_t = d(p_j, r_t)$. Since previously all configurations were distinct by assumption, and $r_t \in \{p_1, \dots, p_{k+1}\}$, the configuration of A has to match the configuration of one of the offline algorithms after the move, say that is B_l . Thus, B_l has to move the server from point r_t to p_j with cost $c_t(B_l) = d(p_j, r_t)$. The other offline algorithms do not move. An example of this invariant is pictured in Figure 2.4.

Due to the symmetry of a metric space, the algorithm has the same cost for serving a single request as the sum of the costs of the offline algorithms for serving that request, *i.e.*

$$\text{ALG}_t = \sum_{i=1}^k c_t(B_i).$$

Hence,

$$\text{ALG} = \sum_{t=1}^T \text{ALG}_t = \sum_{t=1}^T \sum_{i=1}^k c_t(B_i) = \sum_{i=1}^k \sum_{t=1}^T c_t(B_i) = \sum_{i=1}^k c(B_i) + q(B_i).$$

Here, $c(B_i)$ denotes the total cost of algorithm B_i . Recall that $q(B_i)$ is the fixed cost for ensuring the invariant in the beginning. Since the cost of an optimal solution OPT for the instance is a lower bound on the cost of any offline algorithm,

$$\text{ALG} = \sum_{i=1}^k c(B_i) + q(B_i) \geq k \cdot \text{OPT} + \sum_{i=1}^k q(B_i) = k \cdot \text{OPT} + a,$$

where a is a constant which only depends on the initial configuration. We can conclude that the algorithm is at least k -competitive. $\square$

2.5 The greedy algorithm

For the rest of this Chapter, we restrict the k -server problem to one dimension, or to be more specific, the real line. Since the real line is also a metric space, we can apply the result from the last Section and see, that there is no hope for a deterministic algorithm with a competitive ratio better than k . Nevertheless, we want to get an intuition about what we have to bear in mind in order to design an efficient online algorithm.

We start by considering the *greedy algorithm*, which is the one that always moves its closest server to the request. It is greedy because it minimizes the cost for its move in each step. Unfortunately, as one can see in some other problems, we can find instances where the greedy approach behaves poorly. In our problem it even turns out that this algorithm is not competitive at all.

To see that we look at the following instance. Consider three points on the line a , b and c such that $|a - b| = 1$ and $|b - c| = 2$. We have k servers all sitting coincident at c . There sequence of requests is $abababab$ and so on. An optimal solution would simply move one server to a and then a different server to b with no further costs. The greedy algorithm however moves some server to a and then lets that server alternate between a and b for the rest of the sequence, because $|a - b| < |b - c|$. Thus, the competitive ratio

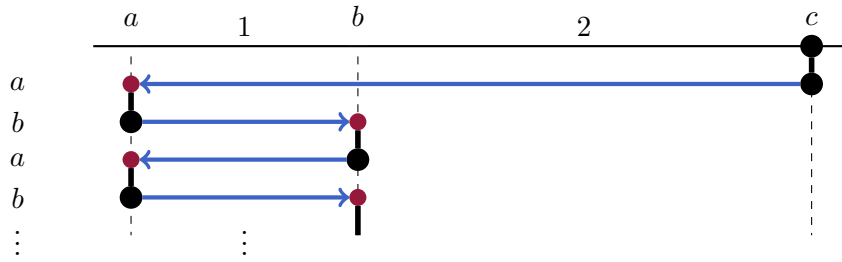


Figure 2.5 An instance with unbounded cost for the greedy algorithm.

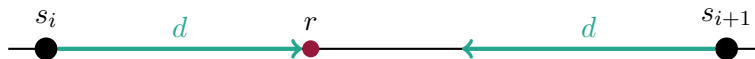


Figure 2.6 The DC move if the request appears between two servers.

of the greedy approach is unbounded for an increasing number of requests, *i.e.*

$$\frac{\text{ALG}}{\text{OPT}} = \frac{3 + n - 1}{3 + 2} \xrightarrow{n \rightarrow \infty} \infty.$$

The situation is visualized in Figure 2.5. Observe that if we always move the closest server, we either do something similar as the adversary or we do something completely different, since we assume that the adversary also moves at most one server per request. A way to improve this non-competitive behavior is to move both servers on either sides of the request in its direction, because by doing that we ensure that we may have at least something in common with the optimal solution. In the above example, that would have caused that another server at c would have slowly moved in the direction of b such that the algorithm would at some point be in the final configuration of the optimal solution and has no further cost. In the next Section we see how this idea was turned into an algorithm, and prove efficiency guarantees for it.

2.6 The Double-Coverage algorithm

As an answer to the k -server conjecture, Chrobak *et al.* [13] proposed Double-Coverage (DC), which is a simple deterministic algorithm for the k -server problem on the line. In this Section, we picture the algorithm and prove in Theorem 3 that it achieves a competitive ratio of k , which is optimal for deterministic online algorithms for the k -server problem, as we have seen in Theorem 2.

DC always serves the next request with its closest server, and in addition, if there is another neighboring server of the request, it gets moved the same distance as the closest server in the direction of the request (see Figure 2.6). Observe that DC is memoryless, since it moves its server only based on the current configuration and the next request.

In the remainder of this Section we detail the proof of the following result.

Algorithm 1 Double-Coverage (DC)**Require:** servers $s_1 \leq \dots \leq s_k$; next request r_t .

- 1: **if** $s_1 \geq r_t$ **then** Move s_1 to r_t .
- 2: **else if** $s_k \leq r_t$ **then** Move s_k to r_t .
- 3: **else**
- 4: Let $s_i \leq r_t \leq s_{i+1}$ and $d = \min\{|s_i - r_t|, |s_{i+1} - r_t|\}$.
- 5: Move s_i and s_{i+1} by distance d towards r_t .

Theorem 3 (Chrobak *et al.*, [13]). *For the k -server problem on the line, DC is k -competitive. That is, for every instance I ,*

$$\text{ALG}(I) \leq k \cdot \text{OPT}(I) + c,$$

where $\text{ALG}(I)$ is the cost of DC, $\text{OPT}(I)$ the cost of the optimal solution and $c \geq 0$ is a constant which only depends on the initial configuration.

Let I be any instance of the k -server problem on the line. We consider an adversary who knows I in advance, so makes the optimal moves. For each request we maintain the configuration of the adversary, *i.e.* the positions of her k servers $x_1, \dots, x_k$ on the line. Similarly, we simulate the algorithm on instance I and keep the configuration of its servers $s_1, \dots, s_k$ at each step. Note that we can always assume that the servers are ordered by index on the line, since neither DC nor the adversary moves one server by another.

This technique allows us to compare the algorithm's and adversary's costs using the *potential function*

$$\Phi = k \cdot \underbrace{\sum_{i=1}^k |s_i - x_i|}_{\Psi} + \underbrace{\sum_{i < j} (s_j - s_i)}_{\Theta}.$$

The potential function maps the configurations of the algorithm's and adversary's servers to a non-negative value, that is the *potential*. Intuitively, Ψ measures how much the algorithm's configuration differs from the adversary's configuration, and Θ stores the spreadness of the algorithm's servers. When one of the players moves, the potential changes. We can then argue about the changes both players incur on the potential for serving a single request, and bound those. We do this in Lemma 4. Then, we sum up those bounds based on Lemma 5 and derive a competitive ratio. The latter only works since we know that the potential cannot drop below zero by definition. This is a well-known proof technique for certain online problems. It was used in the original proof of DC by Chrobak *et al.* [13] and can be also found in standard literature for online optimization, *e.g.* [10].

An important requirement for this technique is that we have to analyze the moves for a single request in two steps. When a request arrives, we first let the adversary move

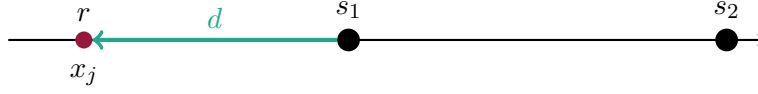


Figure 2.7 Proof of Lemma 4, case where DC moves and $r < s_1$.

her server, look at the change she charges on the potential and then let the algorithm make its move. The following Lemma states the bounds on the change of the potential for those two actions.

Lemma 4 (Chrobak *et al.*, [13]). *For each request, the following statements are true.*

1. *When the adversary serves the request, Φ increases by at most k times the cost the adversary charges for that move, i.e.*

$$\Delta\Phi \leq k \cdot \Delta\text{OPT}.$$

2. *When DC serves the request, Φ decreases by at least the cost the algorithm charges for that move, i.e.*

$$\Delta\Phi \leq -\Delta\text{ALG}.$$

Proof. Let r be the next request and let $(x_1, \dots, x_k)$ be the adversary's and $(s_1, \dots, s_k)$ the algorithm's current configuration.

1. Assume the adversary serves the request by moving x_j to r with cost ΔOPT . Thus, $|x_j - s_j|$ increases by at most ΔOPT when x_j moves away from s_j , which implies

$$\Delta\Phi \leq k \cdot \Delta\text{OPT},$$

since Θ is independent of the adversary's moves and does not change.

2. Now assume that the adversary already served r with x_j . There are two possible cases.

- a) **The request r is not between s_1 and s_k .** W.l.o.g. assume that $r < s_1$. The algorithm moves s_1 to r with cost $d := |r - s_1|$. Since $j \geq 1$, x_1 is not on the right side of r , and $|x_1 - s_1|$ decreases by d . Also, the distance from s_1 to each other server of the algorithm increases by d , and because there are $(k - 1)$ other servers, $\Delta\Theta = (k - 1) \cdot d$. Hence,

$$\Delta\Phi = -k \cdot d + (k - 1) \cdot d = -d = -\Delta\text{ALG}.$$

- b) **The request r is between s_i and s_{i+1} .** W.l.o.g. assume that s_i is closest to r and let $d := |s_i - r|$. Note that $\Delta\text{ALG} = 2d$. We first look at the change of Ψ . If $j \leq i$, x_i is not on the left side of r and thus $\Delta|x_i - s_i| = -d$. Since x_{i+1} can be on both sides of s_{i+1} , $\Delta|x_{i+1} - s_{i+1}| \leq d$.

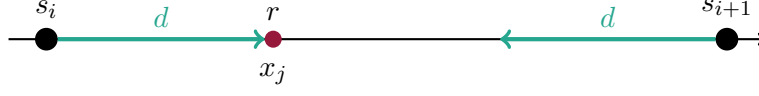


Figure 2.8 Proof of Lemma 4, case where DC moves and $r \in [s_i, s_{i+1}]$.

If $j > i$, x_{i+1} is not on the right side of r and similarly to the other case, $\Delta|x_i - s_i| \leq d$ and $\Delta|x_{i+1} - s_{i+1}| = -d$. In either case Ψ does not increase.

The change of Θ can be computed as follows. There are $(i - 1)$ servers on the left side and $(k - 1)$ on the right side of s_i and $(k - i - 1)$ servers on the right and i servers on the left side of s_{i+1} . The move of s_i increases the distances to the left servers by d but decreases those to the servers on the right by d . For s_{i+1} holds the opposite statement. Thus,

$$\Delta\Theta = \left(\underbrace{(i - 1) - (k - i)}_{\text{Changes by } s_i} + \underbrace{-i + (k - i - 1)}_{\text{Changes by } s_{i+1}} \right) d = -2d,$$

and

$$\Delta\Phi \leq -2d = -\Delta\text{ALG}.$$

□

If we have multiple requests, we denote the change of the cost for serving that request by ΔALG_t resp. ΔOPT_t and the change of the potential, i.e. $\Phi_t - \Phi_{t-1}$, by $\Delta\Phi_t$.

As described above, we now state and prove a Lemma which helps us to derive a competitive ratio from the bounds on the change of the potential. In the remaining thesis, we use it several times to simplify proofs.

Lemma 5. *Consider an instance composed of n requests and let $\alpha, \beta > 0$ such that for each $t \in \{1, \dots, n\}$*

$$\Delta\Phi_t \leq \alpha \cdot \Delta\text{OPT}_t - \beta \cdot \Delta\text{ALG}_t.$$

Then the algorithm is (α/β) -competitive, i.e.

$$\text{ALG} \leq \frac{\alpha}{\beta} \cdot \text{OPT} + \frac{\Phi_0}{\beta}.$$

Proof. First, we sum up the first inequality over all requests, that is

$$\begin{aligned} \sum_{t=1}^n \Delta\Phi_t &\leq \sum_{t=1}^n \alpha \cdot \Delta\text{OPT}_t - \beta \cdot \Delta\text{ALG}_t \\ &= \alpha \cdot \text{OPT} - \beta \cdot \text{ALG}. \end{aligned} \tag{2.1}$$

Applying a telescoping sum argument on the left side of the inequality yields

$$\sum_{t=1}^n \Delta \Phi_t = \sum_{t=1}^n \Phi_t - \Phi_{t-1} = \Phi_n - \Phi_0.$$

Hence, rewriting (2.1) gives

$$\Phi_n + \beta \cdot \text{ALG} = \alpha \cdot \text{OPT} + \Phi_0,$$

which implies due to $\Phi_n \geq 0$ and $\beta > 0$

$$\text{ALG} \leq \frac{\alpha}{\beta} \cdot \text{OPT} + \frac{\Phi_0}{\beta},$$

and concludes that the algorithm is (α/β) -competitive. $\square$

We are now ready to prove the main result of this Section, *i.e.* DC is k -competitive.

Proof of Theorem 3. Consider an instance composed of n requests $r_1, \dots, r_n$. Lemma 4 states for every $t \in \{1, \dots, n\}$

$$\Phi_t - \Phi_{t-1} = \Delta \Phi_t \leq k \cdot \Delta \text{OPT}_t - \Delta \text{ALG}_t.$$

Applying Lemma 5 yields

$$\text{ALG} \leq k \cdot \text{OPT} + \Phi_0.$$

Note that Φ_0 only depends on the initial configuration, and if we set all servers to the same position in the beginning, $\Phi_0 = 0$. $\square$

Chapter 3

Two servers on the line with untrusted predictions

In this Chapter we equip the 2-server problem on the line with predictions and propose a $(1+\lambda)$ -consistent and $(1+\frac{1}{\lambda})$ -robust algorithm for any $\lambda \in [0, 1]$. Given good predictions, this algorithm breaks the lower bound of 2 for deterministic algorithms for this problem.

We start by defining a prediction model in Section 3.1 and give some intuition on how we can improve DC. Then we state an algorithm in Section 3.2 and prove its efficiency in Section 3.3. Finally, we show in Section 3.4 that our algorithm is pareto-optimal.

3.1 Improving Double-Coverage with untrusted predictions

We start this Section by defining a suitable prediction model for the problem. Although we are looking at the 2-server problem on the line in this Chapter, we state the prediction model for the general case with k servers in an arbitrary metric space.

3.1.1 Definition of a prediction model

Definition 8. Let $(\mathcal{M}, d)$ a metric space with servers $s_1, \dots, s_k \in \mathcal{M}$ and let I be an instance composed of n requests $r_1, \dots, r_n \in \mathcal{M}$. The OPTIMALMOVES prediction model gives for each request $r_t \in I$ a prediction $p_t \in \{1, \dots, k\}$ on which server should serve it. A *prediction* for an instance is a sequence of predicted indices, *i.e.* $p_1, \dots, p_n$.

In the following, every prediction for the k -server problem on the line is based on OPTIMALMOVES. We define it for the general metric setting to make it suitable for other problems which are not restricted to the real line. Given servers $s_1, \dots, s_k$ of the algorithm and a single prediction p_t , we call s_{p_t} the *predicted server*.

Observe that there are multiple ways to express this kind of prediction. One way is that, given an instance with initial configuration C_0 and a prediction $p_1, \dots, p_n$, we can precisely determine how the schedule of the prediction looks like. This can be derived from the fact that the predictor never moves two servers simultaneously.

Thus, we can also see a prediction $p_1, \dots, p_n$ as a predicted schedule $C_0, C'_1, \dots, C'_n$ and vice versa. Furthermore, we could also predict the sequence of requests $\tilde{r}_1, \dots, \tilde{r}_n$ and compute, given an initial configuration C_0 , the *prediction-optimal solution*, *i.e.* the optimal schedule $C_0, \tilde{C}_1, \dots, \tilde{C}_n$ for the instance if we assume the predicted requests are correct. We can compute such a solution efficiently due to Theorem 1. Then, given this schedule, we can derive a server prediction $p_1, \dots, p_n$, and based on the previous equivalence obtain a schedule $C_0, C'_1, \dots, C'_n$ of that prediction for the true sequence of requests.

This observation raises the question of why we decided to predict the index of the server which should serve a request. One answer to that is, that this type of prediction is very compact, *i.e.* we only need n numbers which are all from the fixed set of integers $\{1, \dots, k\}$, and having compact predictions is important in practice, as stated in [26]. Furthermore, these predictions are very handy in proving efficiency bounds, as we will see later.

Before we think about how to employ those predictions, we have to define what the *cost* and the *error* of a prediction is.

Definition 9. Let I be an instance and $C'_0, \dots, C'_n$ a prediction. We write ΔPRD_t for the cost of bringing the servers from C'_{t-1} to C'_t , *i.e.*

$$\Delta\text{PRD}_t = d(C'_{t-1}, C'_t).$$

The total cost PRD of the prediction is defined as

$$\text{PRD} = \sum_{t=1}^n \Delta\text{PRD}_t.$$

Definition 10. Let I be an instance composed of the requests $r_1, \dots, r_n$ with a prediction $p_1, \dots, p_n$. For a request r_t , let $C_t = (x_1, \dots, x_k)$ be the configuration of the adversary's servers and $C'_t = (x'_1, \dots, x'_k)$ the configuration of the prediction's servers after both served it. The *error* η_t of the prediction p_t is given by

$$\eta_t = d(C, C') = \sum_{i=1}^k d(x_i, x'_i).$$

We call p_t *correct server prediction* if the adversary serves r_t with x_{p_t} , *i.e.* $x_{p_t} = r_t$. The total *error* η of the prediction is defined as

$$\eta = \sum_{t=1}^n \eta_t.$$

We call a prediction *perfect* if $\eta = 0$.

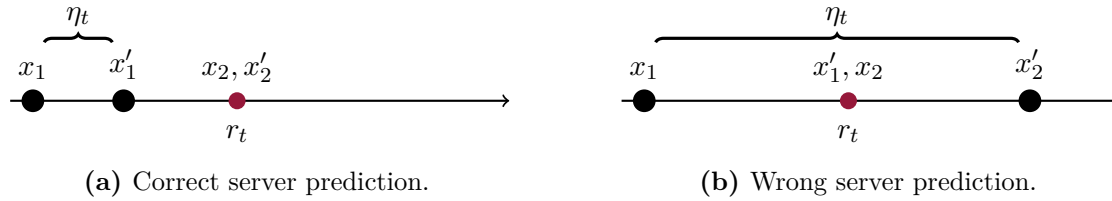


Figure 3.1 A visualization of the errors for OPTIMALMOVES on the line in Definition 10 for two servers.

In Section 2.1 we introduced the notion of *memorylessness* of a deterministic algorithm for the k -server problem. For algorithms using untrusted predictions, we can extend this definition. An algorithm with predictions is *memoryless* if it moves its servers only based on the current configuration, the next request and the next prediction for it. We also implicitly require a memoryless algorithm to be lazy by the same technique as for classic deterministic algorithms.

3.1.2 First approach

For the rest of this Chapter we consider the 2-server problem on the line. Suppose we have three points a, b, c on the line, such that $|a - b| = 1$ and $|b - c| = m$ for some integer m . We only have one request r at point b and two servers s_1 and s_2 which start at a and c . As we only have one request, the adversary would simply move s_1 to r (Figure 3.2a), because s_1 is closest to the request. An online algorithm does not know that there are no further requests. Consider it gets a prediction and the algorithm moves that server to the request. If that server is s_1 , we have a perfect prediction, and our algorithm has the same cost as the adversary, *i.e.* is 1-consistent. On the other hand, if s_2 is predicted, the algorithm pays m times the cost the optimal solution pays, hence is not robust, because m can be arbitrarily large (Figure 3.2b).

If we want to be robust in this example, we have to move s_1 to r and possibly s_2 by a distance independent of m . That is exactly what DC does, since it moves both servers with twice the cost of the adversary (Figure 3.2c). The disadvantage of DC is that it does not look at the prediction at all, and thus, if the prediction is perfect, is also only 2-consistent.

These two approaches show a dilemma. We could tell our algorithm to exactly follow the prediction, and get a good consistency, but are not robust, or we behave like DC and have a bad consistency, but are robust.

A nice thing would be to combine both approaches in a way that we are robust and have a consistency better than DC. We probably even want to tell the algorithm that it should focus on a good consistency than on robustness or the other way around. This choice also depends on the predictions. If we somehow know that the predictions are good, we also want the algorithm to know that, such that we benefit from these. In

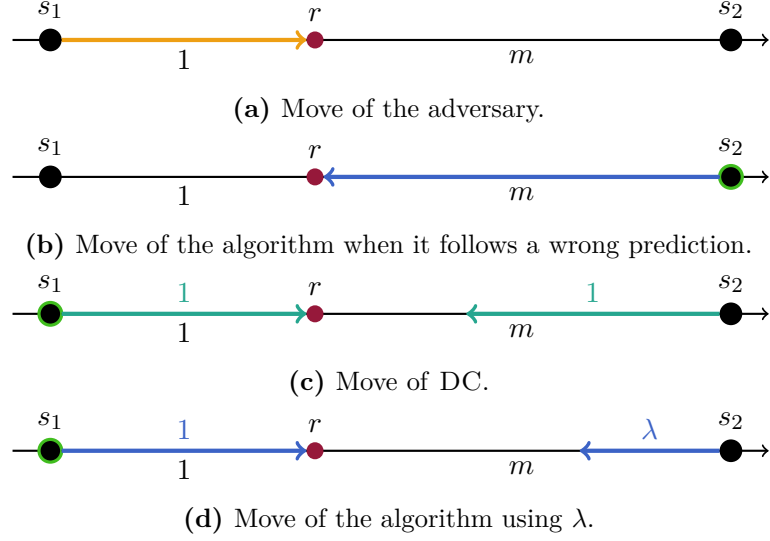


Figure 3.2 First approaches for an algorithm which uses predictions. Note that the predicted server is encircled green.

theory, this is of course not possible, since we assumed the predictor is a black-box, but in practice we often know whether the predictions are good or not.

We implement that choice by introducing a hyperparameter $\lambda \in [0, 1]$. If $\lambda = 0$ we want to exactly follow the predictions and if $\lambda = 1$ we want to behave like DC. Finally, we have to figure out how the algorithm should work if λ is between 0 and 1. We can achieve a useful behavior by interpolating between both approaches using λ . If we have a perfect prediction, *i.e.* s_1 is predicted, the simplest way of doing that is that we move s_2 by λ (Figure 3.2d). Then, the consistency is equal to $1 + \lambda$.

3.1.3 Modifying Double-Coverage

If in the previous example s_2 would be predicted, we cannot talk about consistency, since there is no chance that moving the far away server in a single request scenario is optimal. Thus, we now have a look at an example instance where the far away server is predicted and see how DC behaves in this situation: There are three positions a, b and c , where $|a - b| = 1$ and $|b - c| = m$ for some large integer m . Both servers start at the beginning at b . The sequence of requests is $cabababa$ and so on. The adversary moves s_2 to c and s_1 to a for the first two requests. Then she knows that there will not be any further request on the right of b and thus moves s_2 back to b at the third request (Figure 3.3). Since there are no moves necessary for the next requests, the adversary charges a cost of $2m + 1$.

DC starts with the same two trivial moves to c and a , but then needs exactly m iterations until s_2 reaches b (Figure 3.4). An iteration consists of the requests ba with a total cost of 3 for DC.

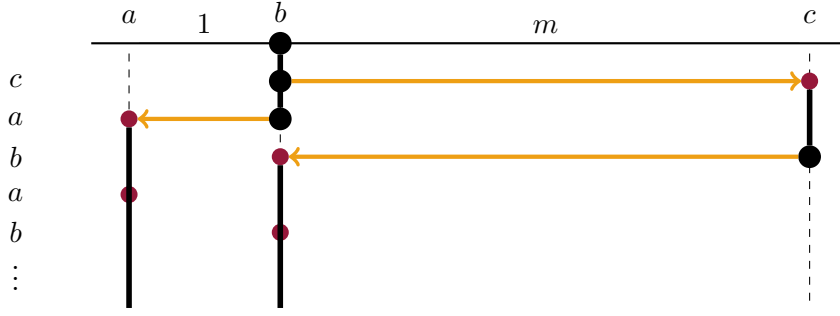


Figure 3.3 Moves of the adversary in the example instance.

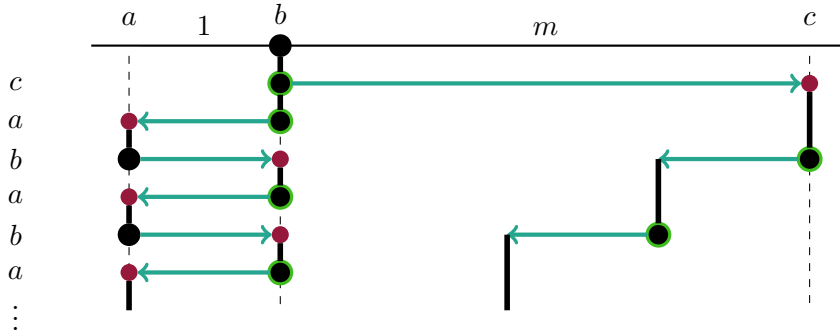


Figure 3.4 A worst-case instance for DC.

Thus, there is a total cost of

$$\text{ALG} = m + 1 + m \cdot (2 + 1) = 4m + 1,$$

which yields a competitive ratio for this instance equal to

$$\frac{\text{ALG}}{\text{OPT}} = \frac{4m + 1}{2m + 1} \xrightarrow{m \rightarrow \infty} 2.$$

The key observation here is that the cost of DC decreases if and only if there are fewer moves of s_2 from c in the direction of b . However, that happens if and only if the distance traveled by s_2 in each iteration increases.

Recall that we are still considering perfect predictions, which means, that we know that s_2 is the optimal choice for serving request b . Considering λ , if $\lambda = 0$ we want s_2 immediately to get moved to b and s_1 to stay at a . Similarly, we want the algorithm to behave exactly like DC if $\lambda = 1$. For all other values of λ we still want to increase the distance of the backward moves of s_2 in order to decrease the total cost depending on the exact value of λ . It turns out that a natural approach, which is also compatible to the previous case, is to increase the distance of the backward moves to $\frac{1}{\lambda}$. Since for any small values of λ this value can be arbitrarily large, we need to cap it when it reaches b .

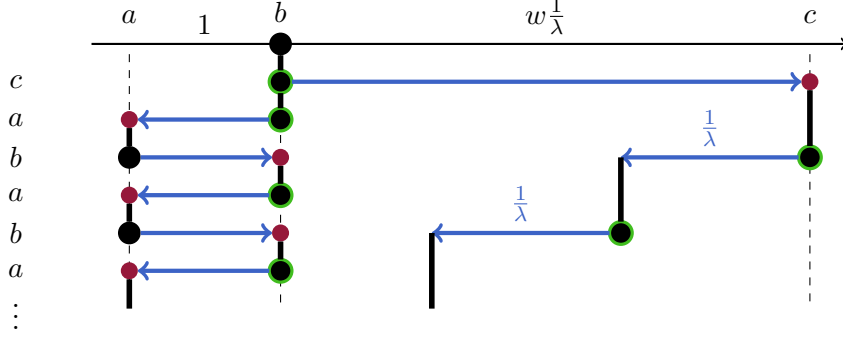


Figure 3.5 The algorithm improves the worst-case performance of DC for the example instance.

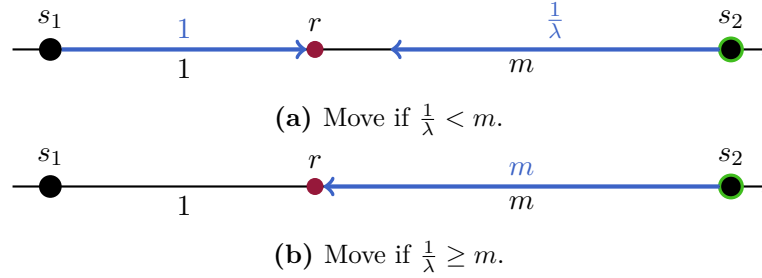


Figure 3.6 Moves of the algorithm if the predicted server is not closest to the request.

Let us check how this modification influences the competitive ratio. Consider the same instance as before, and slightly adjust $|b - c|$ such that $|b - c| = w\frac{1}{\lambda}$ for some natural number w . Our algorithm starts with the same two trivial moves to c and a as DC, but then needs only w iterations until s_2 reaches b , *i.e.*

$$w = |b - c| \lambda \leq m.$$

The moves of our algorithm are illustrated in Figure 3.5. The total cost is equal to

$$\text{ALG} = w\frac{1}{\lambda} + 1 + w(2 + \frac{1}{\lambda}) = 2w\frac{1}{\lambda} + 1 + 2w,$$

which implies a competitive ratio equal to

$$\frac{\text{ALG}}{\text{OPT}} = \frac{2w\frac{1}{\lambda} + 1 + 2w}{2w\frac{1}{\lambda} + 1} = 1 + \frac{2w}{2w\frac{1}{\lambda} + 1} = 1 + \lambda \frac{2w}{2w + \lambda} \xrightarrow{w \rightarrow \infty} 1 + \lambda.$$

This shows that the modification improved the performance in a worst-case instance of DC. Figure 3.6 pictures both moves for the cases where the predicted server is not closest.

For all $\lambda \leq \frac{1}{m}$, the algorithm still completely trusts the prediction, although we demanded this behavior only for $\lambda = 0$. Thus, consider the following instance pictured in

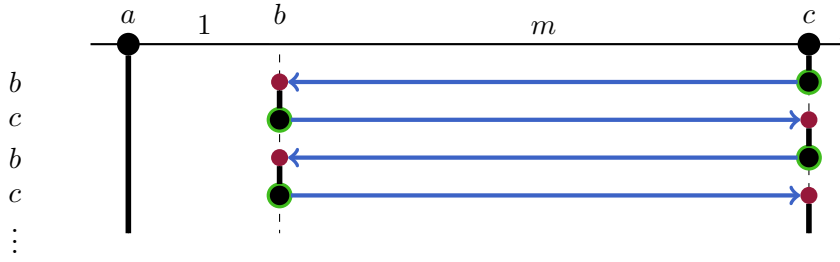


Figure 3.7 An instance with unbounded competitive ratio based on the rules of Figure 3.6.



Figure 3.8 Case where $\frac{1}{\lambda} \geq m$: The left server cannot move beyond the request.

Figure 3.7. As in the last example, we have three fixed positions a , b and c with $|a - b| = 1$ and $|b - c| = m$ such that $\frac{1}{\lambda} \geq m$. This time the servers start at a and c and the request sequence is $cbcbcbcb$ and so on. The adversary would just move s_1 to b with cost 1 and has no further cost. Now assume we have wrong predictions, i.e. s_2 is predicted for the first request b . Since $\frac{1}{\lambda} \geq m$ and s_1 is closest to b , we move s_2 to b without moving s_1 at all. For the next request we have to move s_2 back to c and are back at the initial configuration. By using the same predictions again, we can enforce that behavior over all requests and get an unbounded competitive ratio. Therefore, the algorithm is not robust in its current form. It is easy to see that the s_1 is responsible for that, because it does not move at all. If s_1 would just move a bit in the direction of b , it would reach b after a finite number of requests and there would be no further cost. Since

$$\frac{1}{\lambda} \geq m \iff 1 \geq \lambda \cdot m,$$

we can safely move s_1 by λm without moving s_1 beyond b , as illustrated in Figure 3.8.

We are now ready to compose the different cases to a functioning algorithm and analyze the competitive ratio depending on λ and η .

3.2 The LambdaDC algorithm

In the previous Section we observed how we can modify DC in order to make use of predictions. The next important step is to prove bounds on the efficiency of the algorithm under different prediction qualities. But before we turn towards that in the next Section, we precisely describe the algorithm, which we call LAMBDA DC.

Recall the three cases we reviewed in the previous Section and which are pictured in Figure 3.9. The important connection between the different cases can be described using

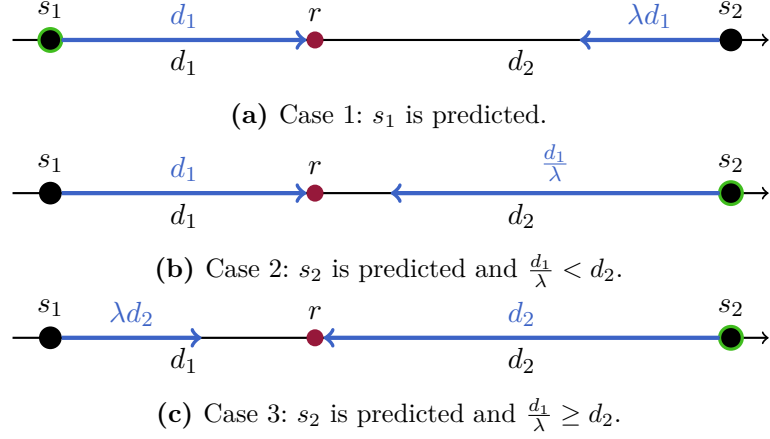


Figure 3.9 The LAMBADC algorithm; $d_1 \leq d_2$.

the term *speed*. We say a server moves with speed δ if it moves δ units in the same time as a server with speed 1 would move one unit.

In LAMBADC the predicted server moves $\frac{1}{\lambda}$ faster than the server on the other side of the request. In other words, the non-predicted server moves only with speed λ compared to the predicted server. When one server reached the request, both servers stop moving.

In Figure 3.9 we are omitting the trivial cases where the request is not between both servers. Since we now have found the essence of the three different cases, we can give a compact definition of LAMBADC in Algorithm 2. Note that LAMBADC is memoryless.

Algorithm 2 LAMBADC

Require: servers $s_1 \leq s_2$; next request r_t ; prediction p_t .

- 1: **if** $s_1 \geq r_t$ **then** Move s_1 to r_t .
 - 2: **else if** $s_2 \leq r_t$ **then** Move s_2 to r_t .
 - 3: **else**
 - 4: Move s_{p_t} with speed 1 and the other server with speed λ towards r_t .
 - 5: Stop both when one server reaches r_t .
-

3.3 Competitive analysis

In this Section we analyze the efficiency of LAMBADC. First, we show in Theorem 7 how the competitive ratio depends on the error η of the prediction, before we prove in Theorem 11 a general upper bound for any prediction, *i.e.* the robustness ratio. Both parts imply the following main Theorem of this Chapter.

Theorem 6. *Let I be an instance for the 2-server problem on the line with a prediction from the OPTIMALMOVES prediction model, which has the error η . Let OPT be the cost of an optimal solution for instance I . Then, for $\lambda \in [0, 1]$, LAMBADC has a competitive ratio for instance I of at most*

$$\min \left(1 + \frac{1}{\lambda}, (1 + \lambda) \left(1 + \frac{2\eta}{\text{OPT}} \right) \right).$$

3.3.1 Competitive analysis with respect to the error

This Section is devoted to the proof of the following Theorem, which bounds the cost of LAMBADC depending on the error η .

Theorem 7. *Let I be an instance of the 2-server problem on the line with a prediction from the OPTIMALMOVES prediction model. Let ALG be the cost of LAMBADC, OPT the cost of an optimal solution for instance I and η the error of the prediction. Then, for $\lambda \in [0, 1]$ holds*

$$\text{ALG} \leq (1 + \lambda)(\text{OPT} + 2\eta) + c,$$

where $c \geq 0$ only depends on the initial configuration of both servers.

The proof of Theorem 7 consists of two parts. Lemma 8 compares the algorithm's cost to the cost of the prediction. Lemma 9 bounds the prediction's cost by the cost of an optimal solution and the error of the prediction. Combining both parts yields a proof for Theorem 7.

Lemma 8. *Let I be an instance of the 2-server problem on the line and $p_1, \dots, p_n$ a prediction with cost PRD . Then,*

$$\text{ALG} \leq (1 + \lambda)\text{PRD} + c,$$

where $c \geq 0$ only depends on the initial configuration of both servers.

Proof. If $\lambda = 0$, LAMBADC follows exactly the prediction. Hence, $\text{ALG} = \text{PRD}$. Otherwise, let s_1, s_2 be the servers of the algorithm and x'_1, x'_2 be the servers of the prediction. We assume w.l.o.g. that $s_1 \leq s_2$ and $x'_1 \leq x'_2$. For $\lambda \in (0, 1]$ we define the potential function

$$\Phi = \frac{\lambda + 1}{\lambda} \cdot \underbrace{\left(|s_1 - x'_1| + |s_2 - x'_2| \right)}_{\Psi} + \underbrace{(s_2 - s_1)}_{\Theta}.$$

Upon the arrival of request r_t , we get a new configuration for our predicted servers (x'_1, x'_2) . This transition from the previous configuration increases the potential function by at

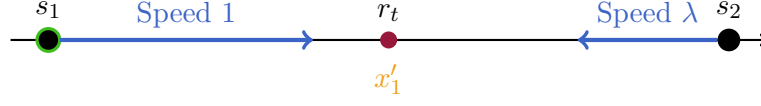


Figure 3.10 Proof of Lemma 8: case where the r_t appears between both servers, and the prediction served r_t with x'_1 .

most $\frac{1+\lambda}{\lambda}$ times the cost between both configurations, *i.e.*

$$\Phi_t - \Phi_{t-1} = \Delta\Phi_t \leq \frac{\lambda+1}{\lambda} \Delta\text{PRD}_t.$$

Now LAMBDADC serves the request with a total cost of ΔALG_t . There are two cases regarding the relative position of the request to both servers.

- a) **The request r_t is not between s_1 and s_2 .** W.l.o.g. assume that $r_t \leq s_1$. Then LAMBDADC moves s_1 to r_t . We can observe the changes

$$\Delta\Psi = -\Delta\text{ALG}_t \text{ and}$$

$$\Delta\Theta = \Delta\text{ALG}_t,$$

which result in a total change of

$$\Delta\Phi = -\frac{\lambda+1}{\lambda} \Delta\text{ALG}_t + \Delta\text{ALG}_t = -\frac{1}{\lambda} \Delta\text{ALG}_t.$$

- b) **The request r_t is between s_1 and s_2 .** W.l.o.g. assume that s_1 is predicted. Then LAMBDADC moves s_1 with speed 1 and s_2 with speed λ towards r_t . Since s_1 is predicted, we know that the server x'_1 is already on r_t , as pictured in Figure 3.10. We can assume by scaling the instance that s_1 travels distance 1. Due to the given speeds, s_2 travels distance λ . Hence, $\Delta|s_1 - x'_1| = -1$, $\Delta|s_2 - x'_2| \leq \lambda$ and $\Delta|s_1 - s_2| = -(1 + \lambda)$, which gives

$$\begin{aligned} \Delta\Phi &\leq \frac{1+\lambda}{\lambda}(\lambda - 1) - (1 + \lambda) \\ &= (1 + \lambda) - (1 + \lambda)\frac{1}{\lambda} - (1 + \lambda) = -\frac{1}{\lambda}(1 + \lambda) = -\frac{1}{\lambda} \Delta\text{ALG}_t. \end{aligned}$$

Therefore, the moves of the algorithm and the prediction for an arbitrary request r_t yield

$$\Delta\Phi_t \leq \frac{1+\lambda}{\lambda} \Delta\text{PRD}_t - \frac{1}{\lambda} \Delta\text{ALG}_t,$$

and we can conclude the proof by applying Lemma 5. □

Lemma 9. *Let I be an instance of the 2-server problem on the line, and $p_1, \dots, p_n$ a prediction with cost PRD and error η . Then,*

$$\text{PRD} \leq \text{OPT} + 2\eta.$$

Proof. For any request r_t , let $C_{t-1} = (x'_1, x'_2)$ be the previous configuration of the adversary and $C_t = (x_1, x_2)$ the configuration after the adversary served r_t . For the prediction we define similarly $C_{t-1}^P = (s'_1, s'_2)$ and $C_t^P = (s_1, s_2)$. The definition of the prediction error is

$$\begin{aligned} \eta_{t-1} &= |x'_1 - s'_1| + |x'_2 - s'_2| \text{ and} \\ \eta_t &= |x_1 - s_1| + |x_2 - s_2|. \end{aligned}$$

By using the triangle inequality we can conclude

$$\begin{aligned} \Delta \text{PRD}_t &= d(C_{t-1}^P, C_t^P) \\ &= |s'_1 - s_1| + |s'_2 - s_2| \\ &= |s'_1 - s_1 + x_1 - x_1 + x'_1 - x'_1| + |s'_2 - s_2 + x_2 - x_2 + x'_2 - x'_2| \\ &\leq |x'_1 - x_1| + |s'_1 - x'_1| + |s_1 - x_1| + |x'_2 - x_2| + |s'_2 - x'_2| + |s_2 - x_2| \\ &= d(C_{t-1}, C_t) + \eta_{t-1} + \eta_t = \Delta \text{OPT}_t + \eta_{t-1} + \eta_t, \end{aligned}$$

and then finish the proof by summing over all requests. $\square$

3.3.2 Consistency

Theorem 7 yields for $\eta = 0$ a competitive ratio equal to $1 + \lambda$. Thus, we have the following Corollary.

Corollary 10. *For the 2-server problem on the line, predictions from the OPTIMAL-MOVES prediction model and $\lambda \in [0, 1]$, LAMBDADC is exactly $(1 + \lambda)$ -consistent.*

Proof. Theorem 7 states an upper bound on the consistency equal to $1 + \lambda$. In Figure 3.5 we assumed having perfect predictions, i.e. $\eta = 0$, and showed that the competitive ratio of LAMBDADC can get arbitrarily close to $1 + \lambda$. Since this gives a lower bound for LAMBDADC on the consistency ratio, it must be exactly $1 + \lambda$. $\square$

3.3.3 Robustness

Recall that the robustness ratio of an online algorithm with predictions is an upper bound on the competitive ratio when making no assumption on the quality of the prediction. To get an intuition how the robustness of LAMBDADC behaves, we again look at the worst-case example of DC.

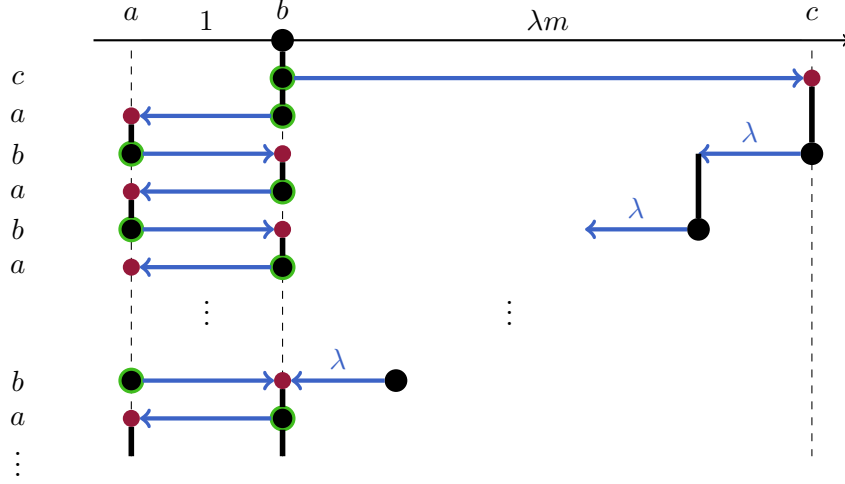


Figure 3.11 The schedule of LAMBADC for a worst-case instance.

There are three points on the line $a < b < c$ with distances $|a - b| = 1$ and $|b - c| = \lambda m$ for some integer m . The sequence of requests is $cababab\dots$ and both servers start at b . The adversary's cost is equal to $1 + 2\lambda m$, since she moves one server to c , the other one to a and then the former right back to b . For the remaining requests she does not have any further cost.

When the algorithm always predicts the right server for a request b , we expect a competitive ratio of at most $1 + \lambda$, since this prediction is perfect. Thus, we look at the case where the algorithm always predicts the left server. The schedule produced by LAMBADC is shown in Figure 3.11. Summing up all costs of the algorithm yields

$$\text{ALG} = \lambda m + 1 + m \cdot (\lambda + 2) = 2m + 2m\lambda + 1.$$

Hence,

$$\frac{\text{ALG}}{\text{OPT}} = \frac{2m + 2m\lambda + 1}{2\lambda m + 1} = 1 + \frac{2m}{2\lambda m + 1} = 1 + \frac{1}{\lambda + \frac{1}{2m}} \xrightarrow{m \rightarrow \infty} 1 + \frac{1}{\lambda},$$

and we conclude that the competitive ratio of LAMBADC approaches $1 + \frac{1}{\lambda}$ for an increasing m .

Whatever the prediction looks like we always have only two possible servers which can be predicted for each request. Thus, we can prove the robustness bound similarly to the proof of Lemma 8 by adding one more case.

Theorem 11. *For the 2-server problem on the line and $\lambda \in (0, 1]$, LAMBADC is exactly $(1 + \frac{1}{\lambda})$ -robust.*

If $\lambda = 0$, LAMBADC exactly follows the prediction, *i.e.* it only moves the predicted server to the request. Thus, we can give LAMBADC predictions which correspond to the

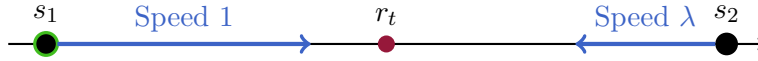


Figure 3.12 Moves of LAMBDADC if r_t is between both servers and s_1 is predicted.

behavior of a greedy algorithm. We have seen in Section 2.5 that the greedy approach is not competitive. Hence, LAMBDADC is not robust for $\lambda = 0$.

Since we illustrated in Figure 3.11 that there exists an instance where LAMBDADC achieves the competitive ratio of $1 + \frac{1}{\lambda}$ in the limit, it remains to show an upper bound on the robustness ratio in the following proof.

Proof of Theorem 11. Let I be an instance and let s_1, s_2 be the algorithm's servers and x_1, x_2 the adversary's servers. For $\lambda \in (0, 1]$ we define the potential function

$$\Phi = (1 + \lambda) \cdot \underbrace{(|s_1 - x_1| + |s_2 - x_2|)}_{\Psi} + \underbrace{|s_1 - s_2|}_{\Theta}.$$

Assume that r_t is the next request and the adversary moves x_j to r_t . Then Ψ increases by at most ΔOPT_t which is the cost the adversary incurs for serving request r_t . Hence,

$$\Delta \Phi \leq (1 + \lambda) \Delta \text{OPT}_t.$$

Now suppose that x_j is on r_t and the algorithm makes its move with a total cost of ΔALG_t . We show that in every case $\Delta \Phi \leq -\lambda \Delta \text{ALG}_t$. There are two cases.

- a) **The request r_t is not between s_1 and s_2 .** W.l.o.g. assume that $r_t \leq s_1$. When s_1 moves to r_t , we can observe the changes

$$\begin{aligned} \Delta \Psi &= -\Delta \text{ALG}_t \text{ and} \\ \Delta \Theta &= \Delta \text{ALG}_t, \end{aligned}$$

which result in a total change of

$$\Delta \Phi = -(1 + \lambda) \Delta \text{ALG}_t + \Delta \text{ALG}_t = -\lambda \Delta \text{ALG}_t.$$

- b) **The request r_t is between s_1 and s_2 .** W.l.o.g. assume that $p_t = 1$. Thus, the algorithm moves s_1 with speed 1 and s_2 with speed λ towards r_t . We can again assume by scaling that both server exactly move one time unit, and therefore travel distance 1 resp. λ . Hence, $\Delta \Theta = -(1 + \lambda)$ and $\Delta \text{ALG}_t = 1 + \lambda$. Note that in this case we cannot assume that r_t was served by x_1 by the adversary as we have done in Lemma 8. We rather have to consider both cases on which server the adversary uses to serve the request. If the adversary serves r_t with x_1 , we have a correct server prediction and hence $\Delta \Psi = \lambda - 1$. Otherwise, we have a wrong server prediction and the predicted server can move away from its corresponding adversary's server.

Thus, $\Delta\Psi = 1 - \lambda$. Since we want an upper bound on $\Delta\Phi$, it suffices to consider the case of a wrong server prediction, because for that case $\Delta\Phi$ is largest. Hence,

$$\begin{aligned}\Delta\Phi &\leq (1 + \lambda)(1 - \lambda) - (1 + \lambda) \\ &= (1 + \lambda) - \lambda(1 + \lambda) - (1 + \lambda) \\ &= -\lambda(\lambda + 1) = -\lambda\Delta\text{ALG}_t.\end{aligned}$$

In both cases we observe that

$$\Delta\Phi \leq (1 + \lambda)\Delta\text{OPT}_t - \lambda\Delta\text{ALG}_t.$$

Finally, we accomplish the proof by Lemma 5. □

3.4 Optimality of LambdaDC

In this Section we show that LAMBDA DC achieves the best possible trade-off between consistency and robustness of any memoryless algorithm for the 2-server problem on the line using predictions, *i.e.* LAMBDA DC is pareto-optimal by Definition 6. Especially, we prove the following result.

Theorem 12. *For any deterministic $(1 + \lambda, \beta)$ -competitive memoryless online algorithm A for the 2-server problem on the line with predictions from OPTIMALMOVES, and $\lambda \in [0, 1]$, it holds that $\beta \geq 1 + \frac{1}{\lambda}$.*

We immediately have the following Corollary by Theorem 7, Theorem 11 and Theorem 12.

Corollary 13. *LAMBDA DC is pareto-optimal.*

Proof of Theorem 12. Let A be a deterministic $(1 + \lambda)$ -consistent memoryless algorithm which uses predictions from OPTIMALMOVES.

If $\lambda = 0$, A is 1-consistent. This means that A has to follow exactly the prediction which, in the case of consistency, is perfect. But this means that we can create arbitrarily bad predictions for certain instances such that A has unbounded competitive ratio, *e.g.* if we always predict the closest server (see Figure 2.5). Thus, we always have an unbounded robustness ratio for $\lambda = 0$.

The case $\lambda = 1$ is trivial, since then the consistency matches the robustness and thus the robustness ratio is trivially bounded.

Now let be $\lambda \in (0, 1)$. Our goal is to show that there exists an instance for that A has a competitive ratio of at least $1 + \frac{1}{\lambda}$. First we consider an instance I with three points a , b and c on the line such that $|a - b| = 1$ and $|b - c| = m$ for some large integer m , for which

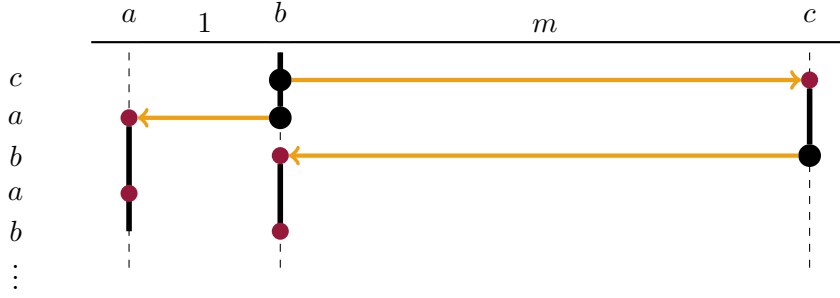


Figure 3.13 Proof of Theorem 12: the adversary's moves for instance I .

holds

$$m \geq \frac{1 + \lambda}{2(1 - \lambda)}. \quad (3.1)$$

Both servers s_1 and s_2 start at b . The sequence of requests is

$$cabababab \dots$$

An adversary moves s_2 to c , s_1 to a and then moves s_2 back to b without any further cost, as shown in Figure 3.13. Thereby she charges a total cost equal to

$$\text{OPT}(I) = 2m + 1.$$

Given the optimal solution, we can derive the following perfect predictions for instance I . We predict for each request b the server s_2 and for each request a the server s_1 :

$$(b, b), (b, c), (a, c), (a, b), (a, b), (a, b) \dots$$

As we are now given an instance with related perfect predictions, we can make statements about the behavior of algorithm A on that instance. Due to its consistency, we know that algorithm A has a competitive ratio of at most $1 + \lambda$. Since $\text{OPT}(I)$ is independent of the length of instance I , $\text{ALG}(I)$ has also to be independent of the length of I . Otherwise, it could not be competitive. As we have defined instance I as an infinite sequence of requests, we can conclude that algorithm A must bring its servers at some point to configuration (a, b) , because that is the unique configuration where the algorithm has no further cost for the remaining instance. Let R be the natural number such that algorithm A reaches configuration (a, b) after serving the $(R + 3)$ th request. We now want to compute a lower bound on the cost algorithm A charges for instance I . The initial moves to c and a are fixed. To serve the following R alternations between a and b , algorithm A can only move s_1 . Together with the fact that moving s_2 from c to b costs

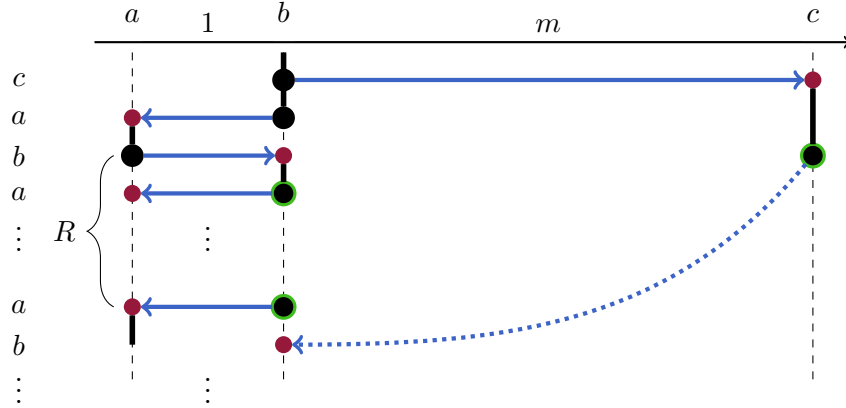


Figure 3.14 Proof of Theorem 12: algorithm A on instance I with perfect predictions.

at least m ,

$$\text{ALG}(I) \geq m + 1 + \underbrace{R}_{\text{alternations}} + \underbrace{m}_{\text{move back to } b} = R + 2m + 1.$$

The moves of algorithm A on instance I are visualized in Figure 3.14. The next step is to bound R . Hence, we chain the lower bound on the cost of the algorithm and the upper bound from the competitive ratio, that is

$$R + 2m + 1 \leq \text{ALG}(I) \leq (1 + \lambda)(2m + 1),$$

which implies

$$R \leq \lambda 2m + \lambda. \quad (3.2)$$

Observe that if we give algorithm A only the first $(R + 3)$ th requests of instance I together with perfect predictions, A is going to behave exactly the same as in instance I . The key to this statement is that we assume that A is memoryless and therefore decides its next move only based on the current configuration, the next request and the next prediction.

In the remaining proof, our goal is to use this observation to construct an instance I_R which has a competitive ratio of at least $1 + \frac{1}{\lambda}$. We start by defining the sequence S_R , which we call an *iteration*, as follows.

$$S_R = \underbrace{b, a, b, \dots, a, b, c}_{R+1 \text{ requests}}$$

Then,

$$I_R = c, a, \underbrace{S_R, \dots, S_R}_{\ell \text{ iterations}}.$$

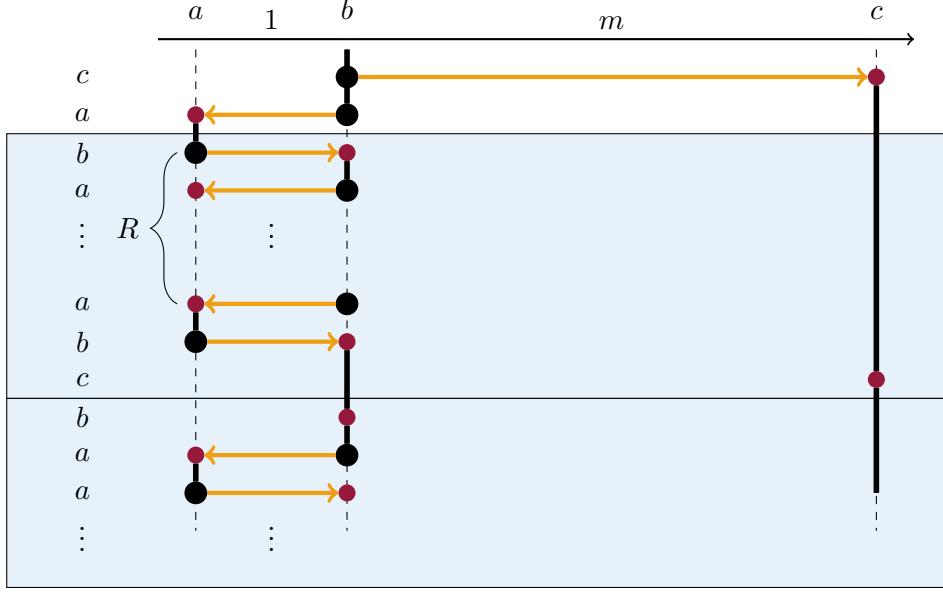


Figure 3.15 Proof of Theorem 12: an optimal solution for instance I_R .

Next we look at the behavior of an adversary on instance I_R . Essentially, the adversary has two possibilities for each iteration. Either s_2 stays on c and s_1 alternates between a and b with a total cost of $R + 1$ per iteration, or s_1 stays at a and s_2 moves at the start of an iteration to b and at the end back to c with a total cost of $2m$. By using (3.1) and (3.2) we can conclude that the first case is optimal, *i.e.*

$$R + 1 \leq \lambda 2m + \lambda + 1 \leq 2m.$$

The optimal solution is illustrated in Figure 3.15. Hence, its cost is equal to

$$\text{OPT}(I_R) = m + 1 + \ell \cdot R - 1 = \ell \cdot R + m.$$

Note that the decrease of 1 comes from putting the cost of the last alternation in each iteration to the next iteration. Thus, we have to remove a cost equal to 1 for the last iteration.

The last part of this proof is analyzing the behavior of algorithm A on instance I_R . For that we use the following predictions which are derived from the perfect predictions of instance I :

$$(b, b), (b, c), (a, c), \underbrace{(a, b), \dots, (a, b)}_{R+1}, (a, c), \underbrace{(a, b), \dots, (a, b)}_{R+1}, (a, c), \dots$$

This means that in each iteration we have the exact same predictions as for the 3rd to $(R + 3)$ th requests of instance I . Since each iteration starts in configuration (a, c) , algorithm A behaves in each iteration exactly like in instance I , as we have observed

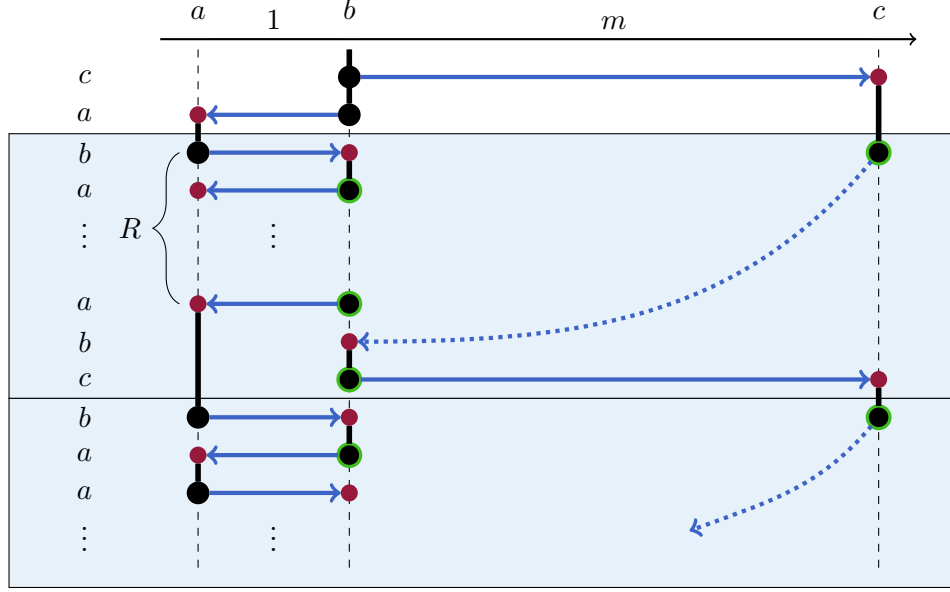


Figure 3.16 Proof of Theorem 12: algorithm A on instance I_R .

above. The moves of algorithm A are pictured in Figure 3.16. The algorithm charges a cost of at least $R + 2m$ in each iteration. Thus,

$$\text{ALG}(I_R) \geq m + 1 + \ell \cdot (R + 2m).$$

Finally, we compute a lower bound on the competitive ratio of algorithm A for instance I_R using (3.2), that is

$$\begin{aligned} \frac{\text{ALG}(I_R)}{\text{OPT}(I_R)} &\geq \frac{\ell \cdot (R + 2m) + m + 1}{\ell \cdot R + m} \\ &= \frac{\ell \cdot R}{\ell \cdot R + m} + \frac{2m\ell}{\ell \cdot R + m} + \frac{m + 1}{\ell \cdot R + m} \\ &= \frac{\ell \cdot R}{\ell \cdot R + m} + \frac{1}{\frac{R}{2m} + \frac{1}{2\ell}} + \frac{m + 1}{\ell \cdot R + m} \\ &\geq \frac{\ell \cdot R}{\ell \cdot R + m} + \frac{1}{\lambda + \frac{\lambda}{2m} + \frac{1}{2\ell}} + \frac{m + 1}{\ell \cdot R + m} \\ &\xrightarrow{\ell \rightarrow \infty} 1 + \frac{1}{\lambda + \frac{\lambda}{2m}} \xrightarrow{m \rightarrow \infty} 1 + \frac{1}{\lambda}. \end{aligned}$$

We conclude that the robustness ratio of algorithm A is at least $1 + \frac{1}{\lambda}$. $\square$

Chapter 4

The k -server problem on the line with untrusted predictions

In this Chapter we look at the k -server problem on the line and extend LAMBADC to k servers. In Section 4.1 we state this modification of LAMBADC, for which we prove a competitive ratio equal to $(1 + (k - 1)\lambda)(1 + 2\eta/\text{OPT})$ in Section 4.2. Finally, we look in Section 4.3 at the robustness of LAMBADC for k servers, state open questions and illustrate a result for certain values of λ .

4.1 LambdaDC with k servers

LAMBADC can easily be stated for k servers. When a request appears between two servers s_i and s_{i+1} , they behave like in the 2-server case, and otherwise the closest server gets moved to the request. The critical question in the first case is how to determine which server moves with which speed. If we get a prediction of OPTIMALMOVES, there is the possibility that none of the two adjacent servers s_i and s_{i+1} of the current request is predicted, *i.e.* the prediction does neither use x'_i nor x'_{i+1} to serve the request. This can happen then the algorithm's configuration is too far off of the prediction's configuration.

We solve this problem in the following way. Let $p_1, \dots, p_n$ be a prediction for an instance I . For a given request r_t , let s_i and s_{i+1} be the adjacent servers of r_t . If $p_t \leq i$, we move s_i with speed 1 and s_{i+1} with speed λ towards r_t and if $p_t \geq i + 1$, we move s_i with speed λ and s_{i+1} with speed 1. In the next Section we will see how we can use this to prove efficiency bounds for LAMBADC.

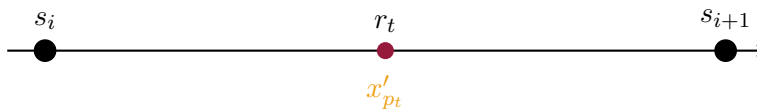


Figure 4.1 The predicted server x'_{p_t} must not have one of the indices of the two neighboring servers of the request, *i.e.* $p_t > i + 1$ or $p_t < i$.

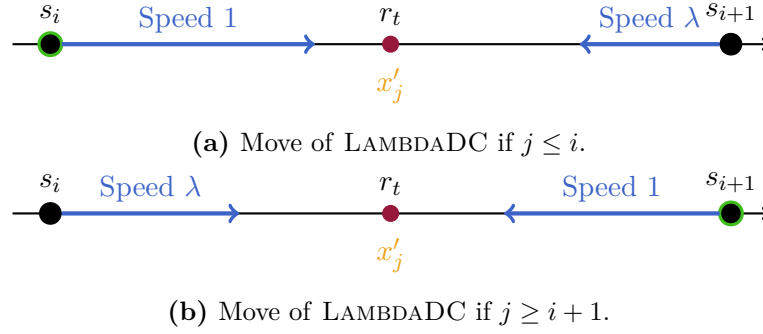


Figure 4.2 The moves of LAMBDADC depend on which side of the request the corresponding algorithm's server s_j of the predicted server x'_j is located.

Algorithm 3 LAMBDADC for k servers on the line

Require: servers $s_1 \leq \dots \leq s_k$; next request r_t ; prediction p_t .

- 1: **if** $s_1 \geq r_t$ **then** Move s_1 to r_t .
 - 2: **else if** $s_k \leq r_t$ **then** Move s_k to r_t .
 - 3: **else**
 - 4: Let $s_i \leq r_t \leq s_{i+1}$.
 - 5: **if** $p_t \leq i$ **then**
 - 6: Move s_i with speed 1 and s_{i+1} with speed λ towards r_t .
 - 7: **else**
 - 8: Move s_i with speed λ and s_{i+1} with speed 1 towards r_t .
 - 9: Stop both when one server reaches r_t .
-

4.2 Competitive analysis with respect to the error

In this Section we prove the main result of this Chapter, which is the following.

Theorem 14. *Let I be an instance for the k -server problem on the line with a prediction from the OPTIMALMOVES prediction model. Let ALG be the cost of LAMBDADC, OPT the cost of an optimal solution for instance I and η the error of the prediction. For $\lambda \in [0, 1]$ it holds*

$$\text{ALG} \leq (1 + (k - 1)\lambda)(\text{OPT} + 2\eta) + c,$$

where $c \geq 0$ only depends on the initial configuration of the servers.

As for Theorem 7, which states the same result for the case $k = 2$, we prove Theorem 14 by chaining the two bounds given in Lemma 15 and Lemma 16.

Lemma 15. *Let I be an instance of the k -server problem on the line and $p_1, \dots, p_n$ a prediction with cost PRD. Then,*

$$\text{ALG} \leq (1 + (k - 1)\lambda)\text{PRD} + c,$$

where $c \geq 0$ only depends on the initial configuration of the servers.

Proof. If $\lambda = 0$, the algorithm exactly behaves like the prediction. Hence, $\text{ALG} = \text{PRD}$. Otherwise, let $s_1, \dots, s_k$ be the servers of the algorithm and $x'_1, \dots, x'_k$ be the servers of the prediction. We assume w.l.o.g. that $s_1 \leq \dots \leq s_k$ and $x'_1 \leq \dots \leq x'_k$. For $\lambda \in (0, 1]$ we define

$$\Phi = \frac{1 + (k-1)\lambda}{\lambda} \cdot \underbrace{\sum_{i=1}^k |s_i - x'_i|}_{\Psi} + \underbrace{\sum_{i < j} (s_j - s_i)}_{\Theta}$$

as potential function.

Let r_t be the next request. Let ΔALG_t be the cost LAMBADC incurs and ΔPRD_t the cost the prediction incurs to serve r_t . Also, let $\Delta \Phi_t$ be the corresponding change of the potential function. We will show that

$$\Delta \Phi_t \leq \frac{1 + (k-1)\lambda}{\lambda} \Delta \text{PRD}_t - \frac{1}{\lambda} \Delta \text{ALG}_t \quad (4.1)$$

holds. Then, Lemma 5 concludes the proof.

First, we look at the prediction's move. Let $(x'_1, \dots, x'_k)$ be the new configuration on the prediction after serving r_t . This transition from the previous configuration increases the potential by at most $(1 + (k-1)\lambda)/\lambda$ times the cost between both configurations, *i.e.*

$$\Delta \Phi_t \leq \frac{1 + (k-1)\lambda}{\lambda} \Delta \text{PRD}_t. \quad (4.2)$$

Now we look at the algorithm's move. Assume that the prediction is already in its new configuration. We distinguish two cases:

- a) **The request r_t is not between s_1 and s_k .** W.l.o.g. assume that $r_t \leq s_1$. Then LAMBADC moves s_1 to r_t . Since we know that some server x'_j of the prediction must already be on r_t , and because $j \geq 1$, we can observe that

$$\begin{aligned} \Delta \Psi &= -\Delta \text{ALG}_t \text{ and} \\ \Delta \Theta &= (k-1)\Delta \text{ALG}_t, \end{aligned}$$

which yield a total change of

$$\begin{aligned} \Delta \Phi &= -\frac{1 + (k-1)\lambda}{\lambda} \Delta \text{ALG}_t + (k-1)\Delta \text{ALG}_t \\ &= -\frac{1}{\lambda} \Delta \text{ALG}_t - (k-1)\Delta \text{ALG}_t + (k-1)\Delta \text{ALG}_t = -\frac{1}{\lambda} \Delta \text{ALG}_t. \end{aligned} \quad (4.3)$$

- b) **The request r_t is between s_i and s_{i+1} .** Let $p_t = j$ and assume w.l.o.g. that $j \leq i$. Then LAMBADC moves s_i with speed 1 and s_{i+1} with speed λ towards r_t . For simplicity, we assume by scaling that s_i travels distance 1 and s_{i+1} distance λ . The algorithm incurs a total cost of $\Delta \text{ALG}_t = 1 + \lambda$. Since $j \leq i$, we know that x'_i must be located on the right of x'_j . Hence, the distance between s_i and x'_i decreases by 1,

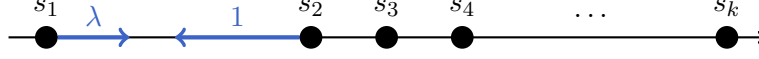


Figure 4.3 An upper bound for the change of Θ for the algorithm's move in the proof of Lemma 15.

but the distance between s_{i+1} and x'_{i+1} increases by at most λ . Thus,

$$\Delta\Psi = -1 + \lambda.$$

The change of Θ is bounded by the move for the case where s_1 moves with speed λ and s_2 with speed 1, as illustrated in Figure 4.3, that is

$$\Delta\Theta \leq \underbrace{(k-2)(1-\lambda)}_{\text{Change between } s_1, s_2 \text{ and } s_3, \dots, s_k} \underbrace{-(1+\lambda)}_{\text{Decrease between } s_1 \text{ and } s_2} = -(k-1)\lambda + k - 3.$$

Both bounds yield using the definition of Φ

$$\begin{aligned} \Delta\Phi &\leq \frac{1 + (k-1)\lambda}{\lambda}(\lambda - 1) - (k-1)\lambda + k - 3 \\ &= 1 - \frac{1}{\lambda} + (k-1)\lambda - (k-1) - (k-1)\lambda + k - 3 \\ &= 1 - \frac{1}{\lambda} - 2 = -\frac{1}{\lambda}(1 + \lambda) = -\frac{1}{\lambda}\Delta\text{ALG}_t. \end{aligned} \tag{4.4}$$

Finally, we can combine the changes from the prediction's move (4.2) and the algorithm's move (4.3), (4.4) and conclude (4.1). $\square$

The following statement compares the cost of a prediction to the cost of an optimal solution. We prove it for general metric spaces in order to reuse it later.

Lemma 16. *Let I be an instance of the k -server problem in the metric space $(\mathcal{M}, d)$ and $p_1, \dots, p_n$ a prediction with cost PRD and error η . Then,*

$$\text{PRD} \leq \text{OPT} + 2\eta.$$

Proof. Let r_t be any request. We will proof that

$$\Delta\text{PRD}_t \leq \Delta\text{OPT}_t + \eta_{t-1} + \eta_t \tag{4.5}$$

holds, and then finish the proof by summation over all requests. Let $C_{t-1} = (x'_1, \dots, x'_k)$ be the previous configuration of the adversary and $C_t = (x_1, \dots, x_k)$ the configuration after the adversary served r_t . For the prediction we define similarly $C_{t-1}^P = (s'_1, \dots, s'_k)$ and $C_t^P = (s_1, \dots, s_k)$. Assume that the adversary served r_t with x'_i and the prediction with s'_j . Since we can assume that the adversary and the prediction only move one server per request, we can compute the distance between two configurations by looking at the

distance these servers move, *i.e.*

$$\begin{aligned}\Delta\text{PRD}_t &= d(C_{t-1}^P, C_t^P) = d(s'_j, s_j) \text{ and} \\ \Delta\text{OPT}_t &= d(C_{t-1}, C_t) = d(x'_i, x_i).\end{aligned}$$

The definitions of the errors η_{t-1} and η_t give lower bounds concerning the moved servers, that is

$$\begin{aligned}\eta_{t-1} &= \sum_{\ell=1}^k d(x'_\ell, s'_\ell) \geq d(x'_j, s'_j) + d(x'_i, s'_i) \text{ and} \\ \eta_t &= \sum_{\ell=1}^k d(x_\ell, s_\ell) \geq d(x_j, s_j) + d(x_i, s_i).\end{aligned}$$

Using the triangle inequality and symmetry of d yields (4.5), *i.e.*

$$\begin{aligned}\Delta\text{PRD}_t &= d(C_{t-1}^P, C_t^P) \\ &= d(s'_j, s_j) + \underbrace{d(s'_i, s_i)}_{=0} \\ &\leq d(s'_j, x'_j) + d(x'_j, x_j) + d(x_j, s_j) + d(s'_i, x'_i) + d(x'_i, x_i) + d(x_i, s_i) \\ &= \underbrace{d(x'_j, x_j)}_{=0} + d(x'_i, x_i) + d(x'_j, s'_j) + d(x'_i, s'_i) + d(x_j, s_j) + d(x_i, s_i) \\ &\leq d(C_{t-1}, C_t) + \eta_{t-1} + \eta_t = \Delta\text{OPT}_t + \eta_{t-1} + \eta_t. \quad \square\end{aligned}$$

Setting $\eta = 0$ in Theorem 14 gives a bound on the consistency of LAMBDADC.

Corollary 17. *For the k -server problem on the line, predictions from the OPTIMAL-MOVES prediction model and $\lambda \in [0, 1]$, LAMBDADC is $(1 + (k - 1)\lambda)$ -consistent.*

4.3 Robustness with k servers

For $k = 2$, Theorem 11 states that LAMBDADC is $(1 + \frac{1}{\lambda})$ -robust for all $\lambda \in (0, 1]$. However, for $k > 2$ we do not know in full generality if LAMBDADC is robust. There is both missing a proof or an example where LAMBDADC is not robust for $k > 2$. The only result in that direction is Theorem 18 which proves robustness for a certain range of λ . We also show in Theorem 19 that this result is the best possible when using a similar potential function as in the previous proofs.

Theorem 18. *For the k -server problem on the line and $\lambda \in (\frac{k-2}{k-1}, 1]$, LAMBDADC has a robustness ratio of at most*

$$\frac{(k-1)\lambda + 1}{(k-1)\lambda - (k-2)}.$$

Note that for $k = 2$, Theorem 18 exactly matches the result of Theorem 11, hence it actually extends the result for two servers. We will even prove a slightly stronger result, namely that the result stated in Theorem 18 is the best possible if we use a potential function of a certain type.

Theorem 19. *For the k -server problem on the line, LAMBADC has a robustness ratio of at most*

$$\frac{(k-1)\lambda + 1}{(k-1)\lambda - (k-2)}$$

for all $\lambda \in (\frac{k-2}{k-1}, 1]$, and this is the best possible result when using a potential function of the form

$$\Phi = \alpha \cdot \sum_{i=1}^k |s_i - x_i| + \sum_{i < j} (s_j - s_i),$$

where $\alpha > 0$.

Proof. Let I be an instance and let $s_1 \leq \dots \leq s_k$ be the servers of the algorithm and $x_1 \leq \dots \leq x_k$ the servers of the adversary. For $\alpha > 0$ we define the potential function

$$\Phi = \alpha \cdot \underbrace{\sum_{i=1}^k |s_i - x_i|}_{\Psi} + \underbrace{\sum_{i < j} (s_j - s_i)}_{\Theta}.$$

Let r_t be the next request. Our goal is to find a value $\beta > 0$, such that

$$\Delta\Phi_t \leq -\beta \cdot \Delta\text{ALG}_t. \quad (4.1)$$

Having that, we can conclude by Lemma 5 that LAMBADC is (α/β) -robust, because the adversary increases the potential by at most $\alpha \cdot \Delta\text{OPT}_t$.

Thus, suppose that the adversary served r_t with x_j . We distinguish two cases whether r_t is between two servers of the algorithm or not. For each case we analyze how the values of α and β have to behave such that (4.1) is satisfied.

1. **The request r_t is not between two servers of the algorithm.** We assume that $r_t < s_1$. LAMBADC moves s_1 to r_t with cost ΔALG_t . Since some server of the adversary is already on r_t , Ψ decreases by ΔALG_t . By moving s_1 , Θ increases by ΔALG_t for each server of the algorithm except s_1 . Thus, (4.1) implies the following condition

$$\Delta\Phi = -\alpha \cdot \Delta\text{ALG}_t + (k-1) \cdot \Delta\text{ALG}_t \leq -\beta \cdot \Delta\text{ALG}_t.$$

Rewriting this yields

$$\alpha \geq \beta + k - 1. \quad (4.2)$$

2. **The request r_t is between s_i and s_{i+1} .** W.l.o.g we assume that s_{i+1} is predicted and that s_{i+1} exactly moves distance 1. The latter can be done by scaling the instance. Hence, s_i moves distance λ , and $\Delta \text{ALG} = 1 + \lambda$.

If $i + 1 \leq j$, x_{i+1} cannot be on the right side of x_j . Therefore, the move of s_{i+1} decreases Ψ by 1 whereas the move of s_i increases Ψ by at most λ , *i.e.*

$$\Delta \Psi \leq \lambda - 1.$$

If $i + 1 > j$, x_{i+1} has to be on the right side of x_j and moving s_{i+1} increases Ψ by at most 1. Since this also implies $i \geq j$, the move of s_i decreases Ψ by λ . Thus,

$$\Delta \Psi \leq 1 - \lambda.$$

We can conclude that Ψ increases by at most $1 - \lambda$. The increase of Θ is bounded by the case where $i = 1$, as we have seen in the proof of Lemma 15, *i.e.*

$$\Delta \Theta \leq -(k - 1)\lambda + k - 3.$$

Combining both bounds in the definition of Φ yields

$$\Delta \Phi \leq \alpha(1 - \lambda) - (k - 1)\lambda + k - 3.$$

Thus, satisfying (4.1) requires

$$\alpha(1 - \lambda) - (k - 1)\lambda + k - 3 \leq -\beta(1 + \lambda),$$

which is equal to

$$\alpha \leq \frac{-\beta(1 + \lambda) + (k - 1)\lambda - (k - 3)}{1 - \lambda}. \quad (4.3)$$

We now evaluate which values of α and β satisfy (4.1). Chaining (4.2) and (4.3) gives

$$\beta + k - 1 \leq \alpha \leq \frac{-\beta(1 + \lambda) + (k - 1)\lambda - (k - 3)}{1 - \lambda} \quad (4.4)$$

Hence, fulfilling (4.4) requires

$$\begin{aligned} \beta + k - 1 &\leq \frac{-\beta(1 + \lambda) + (k - 1)\lambda - (k - 3)}{1 - \lambda} \\ \iff \beta - \lambda\beta + (k - 1)(1 - \lambda) &\leq -\beta - \lambda\beta + (k - 1)\lambda - (k - 3) \\ \iff 2\beta &\leq 2(k - 1)\lambda - (k - 3) - (k - 1) \\ \iff \beta &\leq (k - 1)\lambda - (k - 2). \end{aligned}$$

This means that the potential function only exists if (4.4) is satisfied, and the latter only holds if

$$\beta \leq (k-1)\lambda - (k-2).$$

As we want the best possible competitive ratio, we select β as large as possible, that is

$$\beta = (k-1)\lambda - (k-2).$$

Since the application of Lemma 5 requires β to be positive, we have to restrict λ by

$$\beta = (k-1)\lambda - (k-2) > 0 \iff \lambda > \frac{k-2}{k-1}.$$

Similarly, we want for α the smallest possible value. Hence, we consider the lower bound on α in (4.4) and plug in the value of β , *i.e.*

$$\alpha = \beta + k - 1 = (k-1)\lambda - (k-2) + (k-1) = 1 + (k-1)\lambda.$$

Finally, we can conclude that LAMBDADC is (α/β) -competitive for $\lambda \in (\frac{k-2}{k-1}, 1]$, where

$$\frac{\alpha}{\beta} = \frac{(k-1)\lambda + 1}{(k-1)\lambda - (k-2)}. \quad \square$$

Chapter 5

The k -server problem on trees with untrusted predictions

In the last Chapter we saw that the generalization from two servers on the line to k servers on the line naturally extends the previous results, but meets problems when arguing about robustness. In this Chapter we generalize even further to tree metrics with k servers, while achieving similar results as on the line. We propose a modification of LAMBDADC that has a competitive ratio equal to $(1 + (k - 1)\lambda)(1 + 2\eta/\text{OPT})$ and is also $(1 + \frac{1}{\lambda})$ -robust for two servers. For $k > 2$ we neither have a robustness proof for any range of λ nor an example instance with unbounded competitive ratio.

5.1 Problem definition

A connected, weighted *tree* consist of a set of vertices V and a set of edges $E \subseteq V \times V$, where each edge has an associated non-negative weight and between any two distinct vertices $u, v \in V$ there exists a unique *simple* path. A *simple* path is a path that does not contain a cycle, *i.e.* no vertex appears more than once in the path. In this Chapter we always refer to simple paths when we talk about paths. Given a connected, weighted tree $T = (V, E)$, we can construct a metric space as follows. We set $\mathcal{M} = V$ and for any two points $u, v \in \mathcal{M}$, the distance $d(u, v)$ is the sum of the weights of the unique path between u and v in T . We call d *tree metric*.

The problem of finding an optimal schedule for a sequence of requests in a tree is called *k -server problem on trees*. For a given configuration $(s_1, \dots, s_k)$ of servers and a request r , we call a server s_i *active* if there is no other server on the (unique) path between r and s_i . If there are multiple servers coincident on a point and there is no other server between this point and the request, we select an arbitrary server of these servers to be active. If an active server s_i prevents another server s_j from being active, we say that s_i *blocks* s_j .

Chrobak and Larmore [14] propose a generalization of DC which works on trees. For a given request r , DC moves all active servers with the same speed towards r and stops

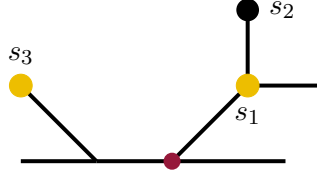


Figure 5.1 Example configuration of three servers s_1, s_2 and s_3 on a tree with a request. The active servers are yellow.

all if at least one server reaches r . During that, some servers may become inactive and hold at that moment. This is also compatible to the case where the metric space is the real line. There, the active servers are the neighboring servers of the request, and the special case where servers become inactive while moving does not occur.

In Section 2.4 we proved that k is a lower bound on the competitive ratio of any deterministic online algorithm for the k -server problem in any metric space with at least $k + 1$ points. Thus, the following result by Chrobak and Larmore [14] tells us that DC is the best possible deterministic algorithm for the k -server problem on trees with at least $k + 1$ points.

Theorem 20 (Chrobak and Larmore, [14]). *DC is k -competitive for the k -server problem on trees.*

We will not state the formal proof of Theorem 20 here. However, we use some ideas of its proof in the next sections, while enhancing that approach with predictions.

5.2 LambdaDC for trees

Similar to the generalization of DC for trees, we can modify LAMBDA DC. Before we think about how to do that, we need the notion of a *matching* between two configurations of servers.

Definition 11. Given two configurations $C = (x_1, \dots, x_k)$ and $C' = (x'_1, \dots, x'_k)$ of equal size, a *perfect matching* M between C and C' is a subset of $C \times C'$, such that every $x_i \in C$ is matched to a single $x'_j \in C'$, i.e. $(x_i, x'_j) \in M$, and every $x'_i \in C'$ is matched to single $x_j \in C$. The *cost* $c(M)$ of a perfect matching M between C and C' is defined as

$$c(M) = \sum_{(x_i, x'_j) \in M} d(x_i, x'_j).$$

We call a *perfect matching* only *matching* in the context of server configurations. On the line, we maintain an *index-based* matching between the algorithm's servers and the prediction's servers, i.e. the two servers with the same index are matched. Although we never explicitly mentioned it, we use this fact in the first part of the potential function Ψ in the proof of Lemma 15, which actually is equal to the cost of that matching. We also implicitly use it when we defined the *predicted server*, i.e. the server s_{p_t} for a given

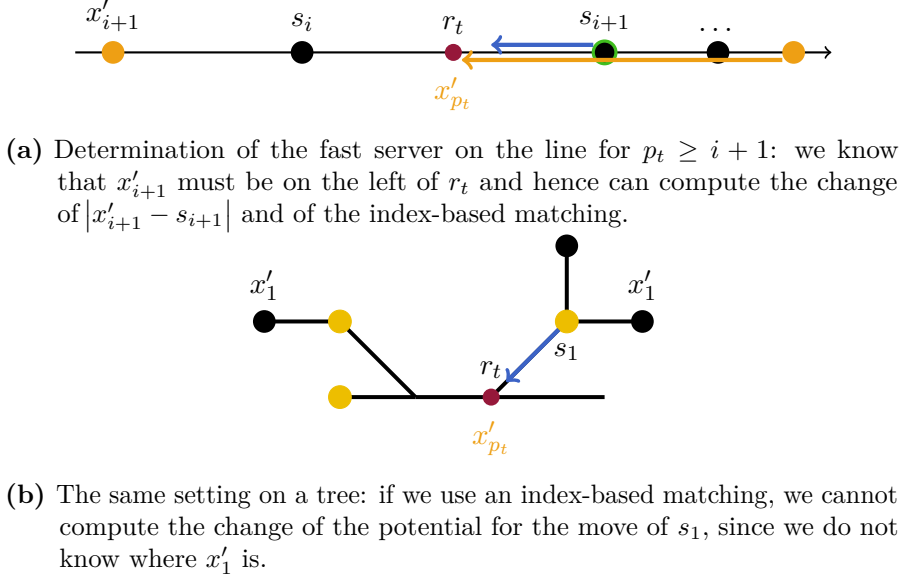


Figure 5.2 The differences of an index-based matching on the line and on a tree.

prediction p_t , since this server is matched to the server x'_{p_t} of the prediction that served the request.

For LAMBADC on the line with k servers, we observed that there is the possibility that the predicted server of the algorithm is not a neighbor of the current request. We solved this problem by looking on which side of the request the predicted server is and hence determined the speeds of the neighboring servers of the current request. The reason why this approach works is the fact that the servers are *ordered by index* on the line. This ordering allows us to argue about the change of Ψ , since we are able to *localize* on which *side* of the request a server with a certain index is, and hence can compute the change of the index-based matching.

Suppose that we would use the same index-based matching on trees. We can have a similar issue as on the line when the predicted server s_{p_t} is not active. The generalization to the solution on the line would be to look which active server s_i blocks the predicted server s_{p_t} . If we go on with that approach, we would give s_i a faster speed than other active servers. Then, on the line we could conclude that the prediction's server matched to s_i must be on the other side of the request, and therefore compute the exact change of the matching's cost and the potential's change. On trees however, this argument fails due to the absence of an ordering, and thus we cannot localize the matched server of s_i . The problem is visualized in Figure 5.2.

In a nutshell, on the line we can assume that the algorithm's servers and the prediction's server are matched by index and hence proof efficiency bounds using a potential function, but on trees we cannot do this assumption. However, if we would know how the servers are matched in the potential function, we could use a similar argumentation as on the

line for determining its change. That is, we move that active server with a faster speed than the others, if it is matched in the potential function to the prediction's server which served the request. Since both servers are matched, we can exactly determine the decrease of the potential due to that move. Note that we cannot use any matching for that purpose yet, but Lemma 21 tells us that there exists a matching with minimal cost where such an active server exists. But before proving that, we have to precise the notion of a matching with minimal cost in the context of server configurations.

Definition 12. Let M be a *perfect matching* between two configurations C and C' . We call M *minimum weight perfect matching* between C and C' if it has the minimal cost among all perfect matchings between C and C' .

For the remaining thesis, we call a minimum weight perfect matching in this context only *minimum matching*. Note that there may exist multiple minimum matchings between two configurations. As aforementioned, the following Lemma ensures that there exists at least one minimum matching with the required property for our algorithm.

Lemma 21. *Let r_t be the next request, $C_{t-1} = (s_1, \dots, s_k)$ be the algorithm's configuration before it serves r_t and $C_t^P(x'_1, \dots, x'_k)$ be the prediction's configuration after serving r_t with server x'_{p_t} . Then there exists a minimum matching between C_{t-1} and C_t^P where x'_{p_t} is matched to an active server.*

Proof. Let M be a minimum matching between C_{t-1} and C_t^P . If x'_{p_t} is matched to an active server in M , we return M and are finished. Otherwise, let s_i be the non-active server matched to x'_{p_t} in M . Since s_i is not active, it must be blocked by an active vertex s_a . Let x'_j be the matching partner of s_a in M . We construct another matching M' by changing the matches of those four server as follows. We match x'_{p_t} to s_a and x'_j to s_i , i.e.

$$M' = M - \{(s_i, x'_{p_t}), (s_a, x'_j)\} + \{(s_i, x'_j), (s_a, x'_{p_t})\}.$$

Note that M' satisfies the desired property. What remains is to show that M' is still minimum. In M , those four elements incur a cost equal to $d(x'_{p_t}, s_i) + d(x'_j, s_a)$ and in M' a cost of $d(x'_{p_t}, s_a) + d(x'_j, s_i)$. Note that it suffices to show that $c(M') \leq c(M)$, because M is already a minimum matching, i.e. $c(M) \leq c(M')$. As we only change the matching of those four elements, we only need to look at the change of the cost those four elements charge on the overall costs of the matchings. Since s_a blocks s_i from r_t , x'_{p_t} is on r_t , and there is a unique path between s_i and x'_{p_t} ,

$$d(x'_{p_t}, s_i) = d(x'_{p_t}, s_a) + d(s_a, s_i).$$

Using this equation and the triangle inequality of the metric space,

$$\begin{aligned}
 d(x'_{p_t}, s_a) + d(x'_j, s_i) &= d(x'_{p_t}, s_i) - d(s_a, s_i) + d(x'_j, s_i) \\
 &\leq d(x'_{p_t}, s_i) - d(s_a, s_i) + d(x'_j, s_a) + d(s_a, s_i) \\
 &= d(x'_{p_t}, s_i) + d(x'_j, s_a),
 \end{aligned}$$

which implies that $c(M') \leq c(M)$ and hence that M' is also minimum. $\square$

Given Lemma 21, we can formalize the previous ideas and apply them to LAMBDADC. Let I be an instance composed of the requests $r_1, \dots, r_n$ and an initial configuration C_0 , let $s_1, \dots, s_k$ be the servers of LAMBDADC, and $C_0^P, \dots, C_n^P$ the configurations of the prediction. Note that $C_0 = C_0^P$. For each request r_t , LAMBDADC first computes the minimum matching M between its current configuration C_{t-1} and the configuration of the prediction after it served the request C_t^P . By Lemma 21, we can assume that in M an active server is matched to the server of the prediction which served r_t . We call this active server of the algorithm *fast*. Then, LAMBDADC moves the fast server with speed 1 and all other active servers with speed λ towards r_t . The servers stop moving when one of them reaches the request.

Algorithm 4 LAMBDADC for k servers on trees

Require: servers $s_1, \dots, s_k$ in configuration C_{t-1} ; next request r_t ; configuration C_t^P of the prediction.

- 1: Let $s_1, \dots, s_q$ be the active servers w.r.t r_t .
 - 2: Compute a minimum matching M between C_{t-1} and C_t^P that satisfies Lemma 21.
 - 3: Let s_a be the matching partner of the prediction's server on r_t in M .
 - 4: Move s_a with speed 1 and the other active servers with speed λ towards r_t .
 - 5: Stop them when at least one server reaches r_t .
-

Before we analyze this variant of LAMBDADC for trees, we have to fix a small issue. Unlike to the real line, on tree metrics a server may have to stop while moving from one point of the space to another point. To overcome this, Coester and Koutsoupias [15] proposed a solution for a similar server problem by adding virtual points to the metric space where the server should have its destination. If a server moves to a virtual point, we leave it at its old position, but consider the virtual point in future computations. We can safely do this assumption, because the triangle inequality of the metric space ensures that we do not increase the overall distance travelled, and hence do not increase the overall cost of the algorithm. We call this technique *virtual augmentation* of the metric space.

The following Theorem is the main result of this Chapter. Note that it is the same bound as in Theorem 14.

Theorem 22. *Let I be an instance for the k -server problem on trees with a prediction from the OPTIMALMOVES prediction model. Let ALG be the cost of LAMBADC, OPT the cost of an optimal solution for instance I and η the error of the prediction. Then, for $\lambda \in [0, 1]$ it holds*

$$\text{ALG} \leq (1 + (k - 1)\lambda)(\text{OPT} + 2\eta) + c,$$

where $c \geq 0$ only depends on the initial configuration of the servers.

As for Theorem 14, we split up the proof of Theorem 22 into two parts, where in the first part we compare the cost of the algorithm to the cost of the prediction and bound in the second part the cost of the prediction by the cost of the optimal solution and the error of the prediction. For the second part we can apply Lemma 16, because it holds for any metric space. Hence, it suffices to prove Lemma 23.

Lemma 23. *Let I be an instance of the k -server problem on trees and $p_1, \dots, p_n$ a prediction with cost PRD . Then,*

$$\text{ALG} \leq (1 + (k - 1)\lambda)\text{PRD} + c,$$

where $c \geq 0$ is a constant which only depends on the initial configuration of the servers.

In the remaining Section we prove Lemma 23. The proof is very similar to the proof of Lemma 15.

Proof. If $\lambda = 0$, the algorithm exactly behaves like the prediction, thus $\text{ALG} = \text{PRD}$. Otherwise, let $s_1, \dots, s_k$ be the servers of the algorithm and $x'_1, \dots, x'_k$ be the servers of the prediction. Let M denote a minimum matching between the current configurations of the two sets of servers. For $\lambda \in (0, 1]$ we define the potential function

$$\Phi = \frac{1 + (k - 1)\lambda}{\lambda} \cdot c(M) + \underbrace{\sum_{i < j} d(s_i, s_j)}_{\Theta}.$$

Given the next request r_t , let ΔALG_t be the cost LAMBADC charges and ΔPRD_t the cost the prediction charges to serve r_t . Also, let $\Delta \Phi_t$ be the corresponding change of the potential function. We will show that

$$\Delta \Phi_t \leq \frac{1 + (k - 1)\lambda}{\lambda} \Delta \text{PRD}_t - \frac{1}{\lambda} \Delta \text{ALG}_t \quad (5.1)$$

holds, and then conclude the proof by Lemma 5.

As in previous proofs for server problems, we first analyze the changes due to the moves of the prediction and then look in a second step at the impact of the algorithm's move on Φ .

Suppose the prediction serves r_t with x'_{p_t} . Since x_{p_t} may moves away from its current matching partner in M , the matching cost increases by at most ΔPRD_t . Thus,

$$\Delta \Phi_t \leq \frac{1 + (k-1)\lambda}{\lambda} \Delta \text{PRD}_t. \quad (5.2)$$

Now assume that x'_{p_t} is already on r_t , and that $s_1, \dots, s_q$ are active. Furthermore, assume that M is the same minimum matching that the algorithm computed for this move. Since both matchings have minimal cost, we can safely do this assumption without changing the potential. Let s_q be the fast server with respect to M and w.l.o.g assume that s_q moves distance 1. Hence, LAMBADC charges cost for that move equal to

$$\Delta \text{ALG}_t = 1 + (q-1)\lambda.$$

Since s_q moves towards its matching partner x'_{p_t} in M , the matching cost decreases by 1. The moves of the other active servers increase $c(M)$ by at most $(q-1)\lambda$. Thus,

$$\Delta c(M) \leq -1 + (q-1)\lambda.$$

Lastly we analyze Θ . For an active server s_p , let ℓ_p be the number of servers s_p blocks plus one, *i.e.* it also counts the server s_p itself. Since the movement of each active server increases the distance to servers it blocks and decreases the distance to other active servers,

$$\Delta \Theta = \underbrace{(\ell_q - 1) - (k - \ell_q)}_{\text{Changes by fast server.}} + \underbrace{\sum_{p=1}^{q-1} (\ell_p - 1)\lambda - (k - \ell_p)\lambda}_{\text{Changes by other active servers.}}.$$

We are now ready to look at the overall change the algorithm incurs on Φ , which is

$$\begin{aligned} \Delta \Phi &= \frac{1 + (k-1)\lambda}{\lambda} \Delta c(M) + \Delta \Theta \\ &\leq \frac{1 + (k-1)\lambda}{\lambda} (-1 + (q-1)\lambda) + (\ell_q - 1) - (k - \ell_q) + \sum_{p=1}^{q-1} (\ell_p - 1)\lambda - (k - \ell_p)\lambda. \end{aligned}$$

Rearranging gives

$$\begin{aligned} \Delta \Phi &\leq -\frac{1}{\lambda} - 2k + (q-1)(1-\lambda) - (q-1)\lambda + 2\ell_q + 2 \sum_{p=1}^{q-1} \ell_p \lambda \\ &= -\frac{1}{\lambda} - (q-1) + 2(q-1)(1-\lambda) - 2k + 2\ell_q + 2 \sum_{p=1}^{q-1} \ell_p \lambda \\ &= -\frac{1}{\lambda} - (q-1) + 2(q-1)(1-\lambda) - 2k + 2\ell_q - 2\lambda\ell_q + 2 \sum_{p=1}^q \lambda\ell_p. \end{aligned}$$

Observe that $\sum_{p=1}^q \ell_p = k$, since every server is either blocked by some active server or active itself. Thus,

$$\begin{aligned}
\Delta\Phi &\leq -\frac{1}{\lambda} - (q-1) + 2(q-1)(1-\lambda) - 2k + 2\ell_q - 2\lambda\ell_q + 2\sum_{p=1}^q \lambda\ell_p \\
&= -\frac{1}{\lambda}(1 + (q-1)\lambda) + 2(q-1)(1-\lambda) - 2k + 2\ell_q - 2\lambda\ell_q + 2\lambda k \\
&= -\frac{1}{\lambda}\Delta\text{ALG}_t + 2(q-1 + \lambda - \lambda q - k + \lambda k + \ell_q - \lambda\ell_q) \\
&= -\frac{1}{\lambda}\Delta\text{ALG}_t + 2(q + \ell_q - 1 - k)(1-\lambda).
\end{aligned}$$

Since $\ell_q - 1$ counts only non-active servers, and q only active servers, $q + \ell_q - 1 \leq k$. Hence,

$$\Delta\Phi \leq -\frac{1}{\lambda}\Delta\text{ALG}_t, \tag{5.3}$$

and adding (5.2) and (5.3) yields (5.1), which concludes the proof. $\square$

5.3 Robustness on trees

Theorem 11 states that LAMBDADC is $(1 + \frac{1}{\lambda})$ -robust for two servers on the line. We can extend this result to the 2-server problem on trees. However, for $k > 2$, we were not able to apply the result of Theorem 18 to trees. Thus, for $k > 2$, there is missing a proof or an example instance on whether LAMBDADC is robust on trees or not.

Theorem 24. *For the 2-server problem on trees and $\lambda \in (0, 1]$, LAMBDADC is $(1 + \frac{1}{\lambda})$ -robust.*

Proof. We can reuse the proof of Theorem 11 by using the tree metric as distance function instead of the absolute value function. When only one server is active, we refer to the first case where the request is not between both servers, and otherwise to the second case. $\square$

Chapter 6

The k -taxi problem

After working on the k -server problem for different metric spaces in the last chapters, we now consider the k -taxi problem. It is a natural generalization of the k -server problem. There are k servers in a metric space and a sequence of requests which have to be served. However, those servers are *taxis*, and they do not just have to go to a certain location as in the k -server problem, but they pick up a customer there and then might drive her to another location.

We start this Chapter by exactly defining the problem and showing different properties in Section 6.1, before we look in Section 6.2 at the 2-taxi problem and analyze a simple, 9-competitive algorithm. These two sections are based on recent results of Coester and Koutsoupias [15]. Afterwards we enhance their algorithm with predictions in Section 6.3 and break existing lower bounds for deterministic algorithms. Finally, we demonstrate upper and lower bounds for it in Section 6.4 and Section 6.5.

6.1 Problem definition

In the k -taxi problem there are k distinct servers $s_1, \dots, s_k$ in a metric space $(\mathcal{M}, d)$. We again use the term *configuration* for the tuple of positions of those servers at a certain point in time. An *instance* remains unchanged and is a sequence of requests with a starting configuration of the servers. A *request* in the k -taxi problem is a tuple (x, y) of points of $\mathcal{M}$. When a server wants to serve a request (x, y) , it first has to move to x and then gets relocated to y . If $x = y$, we call the request *simple* and otherwise *relocation* request. For a simple request r , we also assume that r itself is the point of the simple request in the metric space. If the servers are currently in configuration C_t , and server s_i serves request $r_t = (x, y)$, s_i is at position y in the following configuration C_{t+1} . We name the move of server s_i to x *collection-move* and the move from x to y *relocation-move*. Given an instance composed of C_0 and $r_1, \dots, r_n$, the task of the k -taxi problem is to find a schedule $C_0, C_1, \dots, C_n$ such that each request $r_t = (x, y)$ is served.

There are two flavors of the problem. In the *easy* k -taxi problem, the objective is to minimize the total distance travelled, *i.e.* for each request $r_t = (x, y)$ the distance

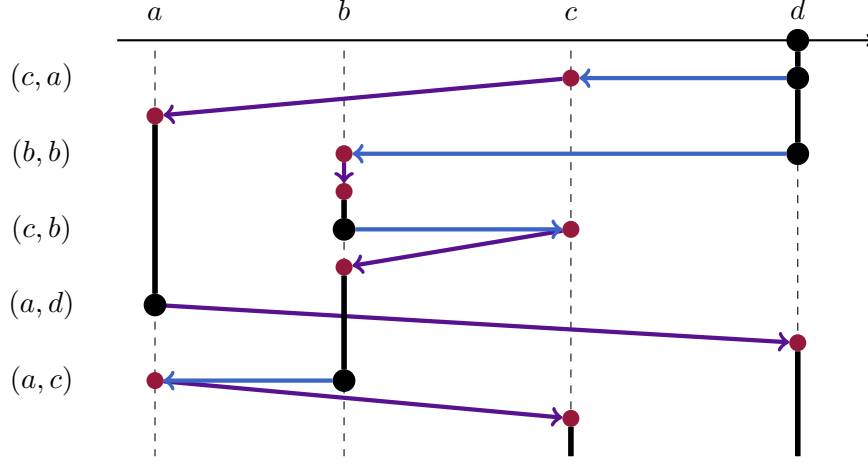


Figure 6.1 An example schedule with two servers and five requests on the line.

travelled to x and the distance travelled from x to y . In contrast to that, the *hard* k -taxi problem only minimizes the distance travelled without passengers, *i.e.* for each request $r_t = (x, y)$, the distance travelled to x . As mentioned before, the k -server problem is a special case of the k -taxi problem, where all requests are simple. Observe that in this case, the easy and hard k -taxi problem are equivalent. For the easy k -taxi problem, Coester and Koutsoupias [15] demonstrate the even stronger result that it is in general equivalent to the k -server problem. In other words, for both the easy k -taxi problem and the k -server problem there exists the same a lower bound, and there exists a c -competitive algorithm for the easy k -taxi problem if and only if there exists a c -competitive algorithm for the k -server problem.

Figure 6.1 displays a schedule for the 2-taxi problem on the line. The collection-moves are colored blue and the relocation-moves purple. The schedule looks slightly confusing, as there are two moves for one request and some of them also go by the other server. In order to properly work with these schedules, we want to make some assumptions. When we analyze algorithms for the k -taxi problem, we will argue with an adversary, which generates the instance in advance. Thus, we can make the following observation. An adversary has no disadvantage when splitting a relocation request (x, y) into a simple request (x, x) and a relocation request (x, y) , since the server which serves the simple request can then simply serve the relocation request without any additional cost. The adversary also knows that there is a relocation request after a simple request to a certain location, and thus can easily select the optimal server. An algorithm does not know that and therefore has only a disadvantage from that assumption. Thus, we assume w.l.o.g that a relocation request (x, y) only appears after the simple request (x, x) . Figure 6.2 shows the updated schedule. The new requests which are based on the assumption are colored green.

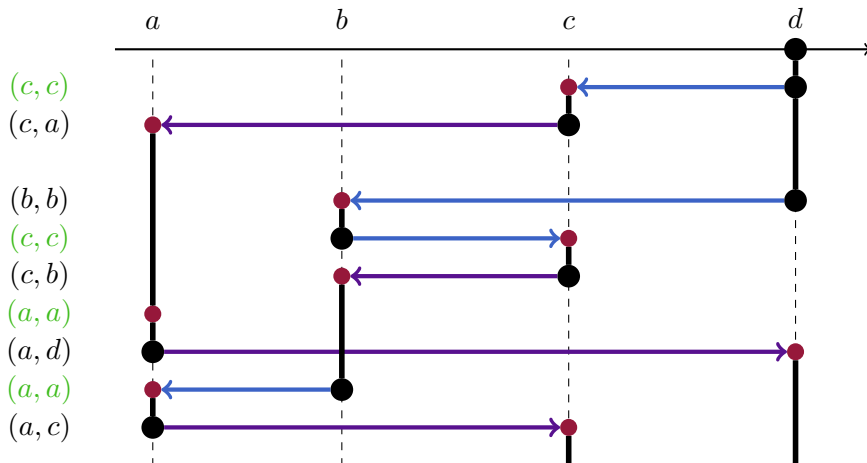


Figure 6.2 The updated example schedule based on the assumption, that there always is simple request (x, x) before a relocation request (x, y) .

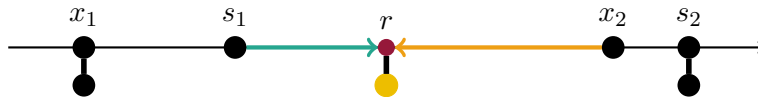


Figure 6.3 The adversary's server x_2 and the algorithm's server s_1 serve request r . They become active in the following configuration. Active servers are colored yellow.

Observe that we can efficiently compute an optimal solution for the hard k -taxi problem using an almost identical approach as demonstrated for the k -server problem in Section 2.3. We only have to change the distance between serving two requests $r = (x, y)$ and $r' = (x', y')$ to $d(y, x')$, since in the hard k -taxi setting we do not care about the cost of relocations. We can conclude that there is an optimal solution for an instance of the hard k -taxi problem which moves at most one server per request.

6.2 A deterministic online algorithm for the hard 2-taxi problem

BIASEDDC [15] is a deterministic online algorithm for the hard 2-taxi problem. Although it is an algorithm for any metric space, we stick with the real line for examples and figures. The algorithm is a modification of DC and based on the observation that in each configuration there is a pair of servers, one of the algorithm and one of the adversary, which are incident. In the beginning this is true, because the servers of both players start in the same configuration. After serving a request r , it is true for the pair of servers of the algorithm and the adversary, which served r . The latter case is shown in Figure 6.3. We call those servers *active*, and denote the adversary's active server by a and the algorithm's by $\tilde{a}$. The other two servers, which are possibly not incident, are called *passive* and are denoted with p (adversary) and $\tilde{p}$ (algorithm).

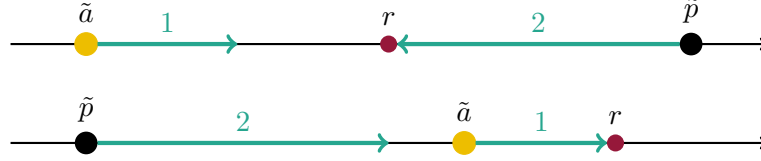


Figure 6.4 Moves of BIASDDC.

When a simple request r appears, BIASDDC moves $\tilde{a}$ with speed 1 and $\tilde{p}$ with speed 2 in the direction of r . Figure 6.4 illustrates these moves. This definition has a similar issue as LAMBDDC on tree metrics, when the destination point of a server is not in the metric space. Hence, we also use the *virtual augmentation* technique to solve this issue. Note that this technique was originally proposed by Coester and Koutsoupias for BIASDDC.

Algorithm 5 BIASDDC

Require: servers $\tilde{a}$ and $\tilde{p}$; last request served by $\tilde{a}$; next request $r = (x, y)$.

- 1: Move $\tilde{a}$ with speed 1 and $\tilde{p}$ with speed 2 towards x .
 - 2: Stop both when one server reaches x .
-

Coester and Koutsoupias prove in [15] that 9 is a lower bound for the competitive ratio of a deterministic online algorithm for the hard 2-taxi problem by giving a reduction from the layered width- k graph traversal problem. Thus, the following result shows that BIASDDC is the best possible deterministic online algorithm for the hard 2-taxi problem.

Theorem 25 (Coester and Koutsoupias, [15]). *BIASDDC is 9-competitive for the hard 2-taxi problem in any metric space.*

Proof. As in the proof of DC, we use a potential function based approach and argue similarly. Let M be a minimum matching between the algorithm's and adversary's servers and $\Phi = 3c(M)$ a potential function. For a single request r , let Δ_{OPT} be the cost of the adversary, Δ_{ALG} be the cost of the algorithm and $\Delta\Phi$ the change in the potential function for serving that request. By using Lemma 5 it remains to show that

$$\Delta_{\text{ALG}} + \Delta\Phi \leq 9 \cdot \Delta_{\text{OPT}}. \quad (6.1)$$

If r is a relocation request, *i.e.* $r = (x, y)$, we know that the previous request was a simple request $r = (x, x)$, thus a and $\tilde{a}$ are already on x and there are no costs for both the adversary and the algorithm. We can always assume that in the previous matching a was matched to $\tilde{a}$, since it had minimal cost and both served the same request. Hence, the matching after relocating both servers does not change, because a and $\tilde{a}$ are still matched, as they are still incident. Thus, the potential also remains unchanged. Since neither the algorithm nor the adversary have to charge cost for serving r , (6.1) holds.

In the remaining proof, we show that (6.1) holds for a simple request $r = (x, x)$. By scaling the instance, we can assume that the algorithm moves $\tilde{a}$ by one unit and $\tilde{p}$ by two units. Hence, $\Delta_{\text{ALG}} = 3$. The change of Φ depends on which server the adversary uses to serve the request, so we have two cases.

- a) **The adversary serves r with a .** Observe that $\Delta_{\text{OPT}} \geq 1$, since $\tilde{a}$ and a start at the same location and $\tilde{a}$ moves distance 1 towards r . Thus, the distance between a and $\tilde{a}$ increases by $\Delta_{\text{OPT}} - 1$ and the distance between p and $\tilde{p}$ by at most 2. Therefore, $c(M)$ increases by at most $\Delta_{\text{OPT}} + 1$ which yields (6.1), *i.e.*

$$\Delta_{\text{ALG}} + \Delta\Phi = 3 + 3 \cdot \Delta c(M) \leq 3 + 3(\Delta_{\text{OPT}} + 1) = 3 \cdot \Delta_{\text{OPT}} + 6 \leq 9 \cdot \Delta_{\text{OPT}}.$$

Note that the last inequality comes from our observation that the adversary moves a by at least distance 1. In the next case, we cannot do this assumption.

- b) **The adversary serves r with p .** Here we analyze the change of M in two steps where first the adversary and then the algorithm moves. By moving p to r , the matching can increase by at most Δ_{OPT} , if p moves away from its previous matching partner $\tilde{p}$. When the algorithm executes its move, $\tilde{a}$ moves distance 1 away from its matching partner a , but $\tilde{p}$ moves distance 2 towards its matching partner p , which already is on r . Thus,

$$\Delta c(M) \leq \Delta_{\text{OPT}} + 1 - 2 = \Delta_{\text{OPT}} - 1,$$

and (6.1) holds, that is,

$$\Delta_{\text{ALG}} + \Delta\Phi \leq 3 + 3(\Delta_{\text{OPT}} - 1) = 3 \cdot \Delta_{\text{OPT}} \leq 9 \cdot \Delta_{\text{OPT}}. \quad \square$$

6.3 The LambdaBiasedDC algorithm

As we developed LAMBDAADC by modifying DC to make use of predictions, we apply a similar approach to enhance BIASEDDC. First we select a prediction model. As we have seen in Section 6.1, an optimal solution moves at most one server per request. Thus, the problem of solving an instance reduces from moving servers to certain positions to selecting which server should serve the request. Since this is exactly the same as in the k -server problem, we employ the OPTIMALMOVES prediction model. Recall that in this prediction model, the predictor tells us which server should serve a specific request, *i.e.* gives us a possible solution to the instance. Also, OPTIMALMOVES was defined for any metric space and therefore is suitable for the 2-taxi problem.

Before we think about possible modifications for BIASEDDC, we briefly review its functionality. Consider we have given a request r and two servers $\tilde{a}$ and $\tilde{p}$, where $\tilde{a}$ served the previous request. Then, BIASEDDC would move $\tilde{a}$ with speed 1 and $\tilde{p}$ with

speed 2 towards r . Also, we introduce $\lambda \in [0, 1]$ with the same meaning as in the k -server problem. Now assume that $\tilde{a}$ is predicted. If the prediction is perfect, and we know that, we want to move $\tilde{p}$ as little as possible and if we have no information about the prediction, we may want to behave like BIASEDDC. Thus, we scale the speed of $\tilde{p}$ by some function of λ . Similarly, when $\tilde{p}$ is predicted, and we trust the predictor, we want it to move even faster than 2. It turned out that Algorithm 6, called LAMBDABIASEDDC, catches up this behavior and has nice properties.

As for BIASEDDC, the definition fails when the metric space has no point at the computed destination of the server. Thus, we again use the *virtual augmentation* technique to solve this issue.

Algorithm 6 LAMBDABIASEDDC

Require: servers $\tilde{a}$ and $\tilde{p}$; last request served by $\tilde{a}$; next request $r_t = (x, y)$.

Move $\tilde{a}$ with speed 1 towards x .

if $\tilde{a}$ is predicted **then**

Move $\tilde{p}$ with speed $1 + \lambda$ towards x .

else

Move $\tilde{p}$ with speed $1 + \frac{1}{\lambda}$ towards x .

Stop both when one server reaches x .

6.4 Competitive analysis of LambdaBiasedDC

6.4.1 Competitive analysis with respect to the error

In this Section we analyze the competitive ratio with respect to the error η of the prediction, and prove the following result.

Theorem 26. *Let I be an instance for the 2-taxi problem with a prediction from the OPTIMALMOVES prediction model. Let ALG be the cost of LAMBDABIASEDDC, OPT the cost of an optimal solution for instance I and η the error of the prediction. Then, for $\lambda \in [0, 1]$,*

$$\text{ALG} \leq (3 + 4\lambda + 2\lambda^2)(\text{OPT} + 2\eta).$$

Note that LAMBDABIASEDDC breaks the lower bound of 9 [15] for a deterministic online algorithm for the 2-taxi problem if $\lambda < 1$ and the predictions are good. As for the previous proofs concerning the competitive ratio with respect to the error of the prediction, we divide the proof of Theorem 26 into two parts, which are stated in Lemma 27 and Lemma 28.

Lemma 27. *Let I be an instance of the k -taxi problem and $p_1, \dots, p_n$ a prediction with cost PRD. Then,*

$$\text{ALG} \leq (3 + 4\lambda + 2\lambda^2)\text{PRD}.$$

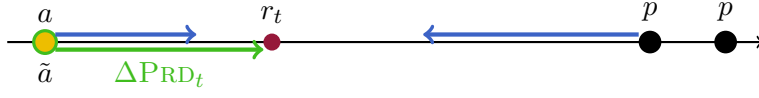


Figure 6.5 Case where server $\tilde{a}$ is predicted in the proof of Lemma 27.

Proof. Let r_t be any request of I and let $\tilde{a}, \tilde{p}$ be the servers (and their position) of LAMBDABIASEDDC and a, p the servers of the prediction. As in the proof of BIASEDDC we have active $(\tilde{a}, a)$ and passive $(\tilde{p}, p)$ servers. We can assume the active servers served the previous request and are incident. Let M be a minimum matching between the algorithm's servers and the prediction's servers. For $\lambda \in [0, 1]$ we define

$$\Phi = (1 + 2\lambda) \cdot c(M)$$

as potential function. We will show that

$$\Delta \text{ALG}_t + \Delta \Phi_t \leq (3 + 4\lambda + 2\lambda^2) \Delta \text{PRD}_t \quad (6.1)$$

holds, where ΔALG_t and ΔPRD_t are the algorithm's and prediction's cost for serving r_t and $\Delta \Phi_t$ is the change of the potential. Then, Lemma 5 concludes the proof.

Since we use a similar potential function as in the proof of Theorem 25, we can use the same argumentation as we did there to assume that r_t is a simple request. We distinguish the two cases whether $\tilde{a}$ or $\tilde{p}$ is predicted.

1. **The server $\tilde{a}$ is predicted.** This means that $\tilde{a}$ moves with speed 1 and $\tilde{p}$ with speed $1 + \lambda$. By scaling the instance, we can assume w.l.o.g. that $\tilde{a}$ moves 1 unit and $\tilde{p}$ moves $1 + \lambda$ units. The prediction serves r_t with server a , incurring a cost of ΔPRD_t . The situation is pictured in Figure 6.5. The key observation of this case is, that ΔPRD_t is at least 1, since $\tilde{a}$ and a start at the same position, but $\tilde{a}$ does not move beyond r_t . This also implies that a moves $\Delta \text{PRD} - 1$ units further than $\tilde{a}$. Thus, a moves $\Delta \text{PRD} - 1$ units and $\tilde{p}$ moves at most $1 + \lambda$ units away from their old matching partner. Therefore,

$$\Delta c(M) \leq \Delta \text{PRD}_t - 1 + 1 + \lambda = \Delta \text{PRD}_t + \lambda,$$

and by using $\Delta \text{PRD}_t \geq 1$ we verify (6.1), that is

$$\begin{aligned} \Delta \text{ALG}_t + \Delta \Phi_t &\leq 2 + \lambda + (1 + 2\lambda)(\Delta \text{PRD}_t + \lambda) \\ &= 2 + 2\lambda + 2\lambda^2 + (1 + 2\lambda)\Delta \text{PRD}_t \\ &\leq (2 + 2\lambda + 2\lambda^2 + 1 + 2\lambda)\Delta \text{PRD}_t \\ &= (3 + 4\lambda + 2\lambda^2)\Delta \text{PRD}_t. \end{aligned}$$

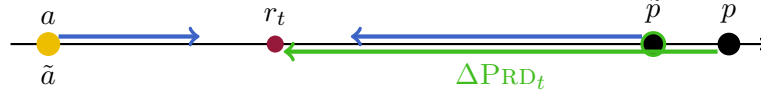


Figure 6.6 Case where server $\tilde{p}$ is predicted in the proof of Lemma 27.

2. **The server $\tilde{p}$ is predicted.** We can conclude that the prediction served r_t with p and that $\tilde{a}$ moves with speed 1 and $\tilde{p}$ with speed $1 + \frac{1}{\lambda}$. Again, we assume that $\tilde{a}$ moves 1 unit and $\tilde{p}$ moves $1 + \frac{1}{\lambda}$ units, as illustrated in Figure 6.6. Since p and $\tilde{p}$ may not start at the same position, there is no bound on ΔPRD_t . To analyze the change of M , we first look at the change the prediction incurs and then at the moves of LAMBDABIASEDDC. When the prediction moves, the matching cost increases by at most ΔPRD_t . By moving $\tilde{a}$, $c(M)$ increases by 1, since a did not move. By moving $\tilde{p}$ towards p , the matching decreases by $1 + \frac{1}{\lambda}$. Thus,

$$\Delta c(M) \leq \Delta\text{PRD}_t + 1 - (1 + \frac{1}{\lambda}) = \Delta\text{PRD}_t - \frac{1}{\lambda}.$$

We conclude that (6.1) holds, *i.e.*

$$\begin{aligned} \Delta\text{ALG}_t + \Delta\Phi_t &\leq 2 + \frac{1}{\lambda} + (1 + 2\lambda)(\Delta\text{PRD}_t - \frac{1}{\lambda}) \\ &= 2 + \frac{1}{\lambda} - \frac{2}{\lambda} - \frac{1}{\lambda^2} + (1 + 2\lambda)\Delta\text{PRD}_t \\ &= 2 - \underbrace{\frac{1}{\lambda} - \frac{1}{\lambda^2}}_{\leq 0} + (1 + 2\lambda)\Delta\text{PRD}_t \\ &\leq (1 + 2\lambda)\Delta\text{PRD}_t \leq (3 + 4\lambda + 2\lambda^2)\Delta\text{PRD}_t. \quad \square \end{aligned}$$

Lemma 28. *Let I be an instance of the k -taxi problem and $p_1, \dots, p_n$ a prediction with cost PRD and error η . Then,*

$$\text{PRD} \leq \text{OPT} + 2\eta.$$

Proof. The proof is based on the proof of Lemma 16, which is stated for general metric spaces and thus available. In its proof, we show that for each request (4.5) holds. Since simple requests are equivalent to requests in the k -server setting, we have nothing to prove for them. Hence, let $r_t = (s, t)$ be a relocation request. There are no cost for serving it, since we can assume that one server of the algorithm and one server of the prediction is already on s . Thus, $\Delta\text{PRD}_t = \Delta\text{OPT}_t = 0$, and (4.5) holds. $\square$

6.4.2 Consistency

Theorem 26 directly implies the following Corollary, which gives a bound on the consistency of LAMBDABIASEDDC.

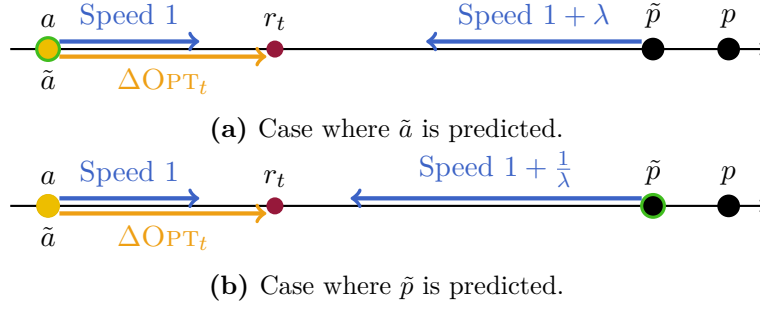


Figure 6.7 Proof of Theorem 30, cases where the adversary serves the requests with the active server.

Corollary 29. *For the 2-taxi problem, predictions from the OPTIMALMOVES prediction model and $\lambda \in [0, 1]$, LAMBDABIASEDDC is $(3 + 4\lambda + 2\lambda^2)$ -consistent.*

6.4.3 Robustness

Theorem 30. *For $\lambda \in (0, 1]$, LAMBDABIASEDDC is $(3 + \frac{4}{\lambda} + \frac{2}{\lambda^2})$ -robust for the 2-taxi problem.*

Proof. Let r_t be any request of I and let $\tilde{a}, \tilde{p}$ the servers of LAMBDABIASEDDC and a, p the servers of the adversary. Again, we have active $(\tilde{a}, a)$ and passive $(\tilde{p}, p)$ servers. Let M be a minimum matching between the algorithm's servers and the adversary's servers. Let

$$\Phi = \left(1 + \frac{2}{\lambda}\right) \cdot c(M)$$

be a potential function. We will show that

$$\Delta \text{ALG}_t + \Delta \Phi_t \leq \left(3 + \frac{4}{\lambda} + \frac{2}{\lambda^2}\right) \Delta \text{OPT}_t \quad (6.2)$$

holds, where ΔALG_t and ΔPRD_t are the algorithm's and adversary's cost for serving r_t and $\Delta \Phi_t$ is the change of the potential. Lemma 5 then concludes the proof.

Similarly to the proof of Theorem 25 we can ignore relocation requests. Assume that r_t is a simple request. The move of LAMBDABIASEDDC depends on whether $\tilde{a}$ or $\tilde{p}$ is predicted. In both cases $\tilde{a}$ moves with speed 1. We can assume w.l.o.g. that $\tilde{a}$ moves exactly one unit. The following four cases show all possible combinations of predictions and moves of the adversary for the given simple request. We prove for each case that (6.2) is true, and so conclude the proof.

1. **The adversary serves r_t with a .** Hence, $\Delta \text{OPT} \geq 1$, because a and $\tilde{a}$ were previously incident, and $\tilde{a}$ moves exactly one unit. This also implies that the movement of the active servers increases the cost of M by $\Delta \text{OPT} - 1$. The two subcases are displayed in Figure 6.7.

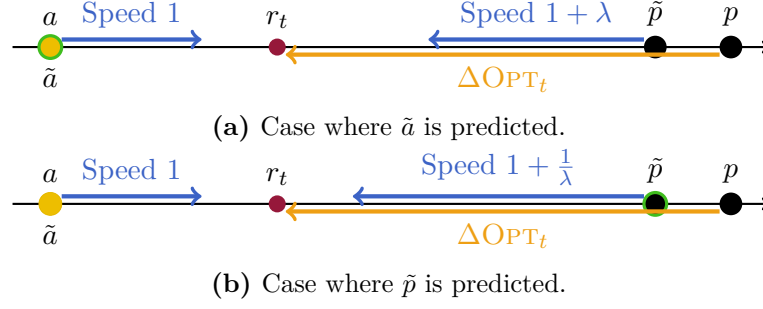


Figure 6.8 Proof of Theorem 30, cases where the adversary serves the requests with the passive server.

- a) If $\tilde{a}$ is predicted, $\tilde{p}$ moves exactly $1 + \lambda$ units. Hence, $c(M)$ increases by at most $1 + \lambda$, because $\tilde{p}$ may move away from its previous matching partner p . Thus, (6.2) holds, that is

$$\begin{aligned} \Delta \text{ALG} + \Delta \Phi &\leq 2 + \lambda + \left(1 + \frac{2}{\lambda}\right)(\Delta \text{OPT} + \lambda) \\ &\leq (2 + \lambda + \lambda + 2 + 1 + \frac{2}{\lambda})\Delta \text{OPT} \\ &= (5 + 2\lambda + \frac{2}{\lambda})\Delta \text{OPT} \leq (3 + \frac{4}{\lambda} + \frac{2}{\lambda^2})\Delta \text{OPT}. \end{aligned}$$

- b) If $\tilde{p}$ is predicted, $\tilde{p}$ moves exactly $1 + \frac{1}{\lambda}$ units. Again, this increases $c(M)$ by at most $1 + \frac{1}{\lambda}$, and (6.2) holds, *i.e.*

$$\begin{aligned} \Delta \text{ALG} + \Delta \Phi &\leq 2 + \frac{1}{\lambda} + \left(1 + \frac{2}{\lambda}\right)(\Delta \text{OPT} + \frac{1}{\lambda}) \\ &\leq (2 + \frac{1}{\lambda} + \frac{1}{\lambda} + \frac{2}{\lambda^2} + 1 + \frac{2}{\lambda})\Delta \text{OPT} \\ &= (3 + \frac{4}{\lambda} + \frac{2}{\lambda^2})\Delta \text{OPT}. \end{aligned}$$

2. **The adversary serves r_t with p .** In this case, there is no bound on ΔOPT . The situation is pictured in Figure 6.8.

- a) If $\tilde{a}$ is predicted, $\tilde{p}$ moves exactly $1 + \lambda$ units. In this case, $c(M)$ first increases by at most ΔOPT due to the move of p , but then decreases by $1 + \lambda$, because $\tilde{p}$ moves in the direction of p , which already is on the request. The movement of $\tilde{a}$ increases the cost of M by exactly 1, and we can conclude that (6.2) holds, *i.e.*

$$\begin{aligned} \Delta \text{ALG} + \Delta \Phi &\leq 2 + \lambda + \left(1 + \frac{2}{\lambda}\right)(\Delta \text{OPT} - \lambda) \\ &= 2 + \lambda + -\lambda - 2 + \left(1 + \frac{2}{\lambda}\right)\Delta \text{OPT} \\ &= \left(1 + \frac{2}{\lambda}\right)\Delta \text{OPT} \leq (3 + \frac{4}{\lambda} + \frac{2}{\lambda^2})\Delta \text{OPT}. \end{aligned}$$

- b) If $\tilde{p}$ is predicted, it moves exactly $1 + \frac{1}{\lambda}$ units. Here the move of p increases M by at most Δ_{OPT} , the movement of $\tilde{a}$ increases $c(M)$ by 1 and finally $\tilde{p}$ decreases $c(M)$ by exactly $1 + \frac{1}{\lambda}$. Again, (6.2) holds, *i.e.*

$$\begin{aligned} \Delta_{\text{ALG}} + \Delta\Phi &\leq 2 + \frac{1}{\lambda} + (1 + \frac{2}{\lambda})(\Delta_{\text{OPT}} - \frac{1}{\lambda}) \\ &= 2 + \underbrace{\frac{1}{\lambda} - \frac{1}{\lambda} - \frac{2}{\lambda^2}}_{\leq 0} + (1 + \frac{2}{\lambda})\Delta_{\text{OPT}} \leq (3 + \frac{4}{\lambda} + \frac{2}{\lambda^2})\Delta_{\text{OPT}}. \quad \square \end{aligned}$$

6.5 A worst-case instance for LambdaBiasedDC

In the final part of this Chapter, we propose an instance where LAMBDABIASEDDC achieves its exact consistency and robustness ratios bounded by Corollary 29 and Theorem 30.

Theorem 31. *For the 2-taxi problem, predictions from the OPTIMALMOVES prediction model and $\lambda \in [0, 1]$, the consistency ratio of LAMBDABIASEDDC is exactly $3 + 4\lambda + 2\lambda^2$.*

Theorem 32. *For the 2-taxi problem, predictions from the OPTIMALMOVES prediction model and $\lambda \in (0, 1]$, the robustness ratio of LAMBDABIASEDDC is exactly $3 + \frac{4}{\lambda} + \frac{2}{\lambda^2}$.*

We employ the real line as metric space. Let $a < b < c$ three points on the line. The initial configuration of the servers s_1, s_2 is (a, c) , and the sequence of requests is composed of consecutive iterations. An iteration consists of the requests $(a, a), (b, a)$ and (c, c) . Let ℓ be the number of iterations, and let $|a - b| = 1$ and $|b - c| = m$, where m is an upper bound on the algorithm's cost of an iteration. We will see that such a bound exists, by proving that the cost per iteration approaches a certain limit from below for an increasing number of iterations. Using this value for $|b - c|$ ensures that both servers will not meet.

An optimal solution is to serve the first two requests in each iteration with s_1 , while s_2 stays in place and serves the third request in each iteration. Thus, the only collection-move to be done is moving s_1 from a to b with cost 1, as pictured in Figure 6.9. Hence, $\text{OPT} = \ell$.

In the following two subsections, we first look at the case where we have perfect predictions and show that the competitive ratio approaches the established bound in Corollary 29. After that we prove the same for Theorem 30 assuming bad predictions.

6.5.1 Consistency

When looking at consistency, we have given perfect predictions, *i.e.* for the first two requests of an iteration s_1 is predicted and for the third request s_2 . LAMBDABIASEDDC processes the instance as illustrated in Figure 6.10. In the remaining Section, we compute the exact cost of LAMBDABIASEDDC for this instance. We start by computing the cost

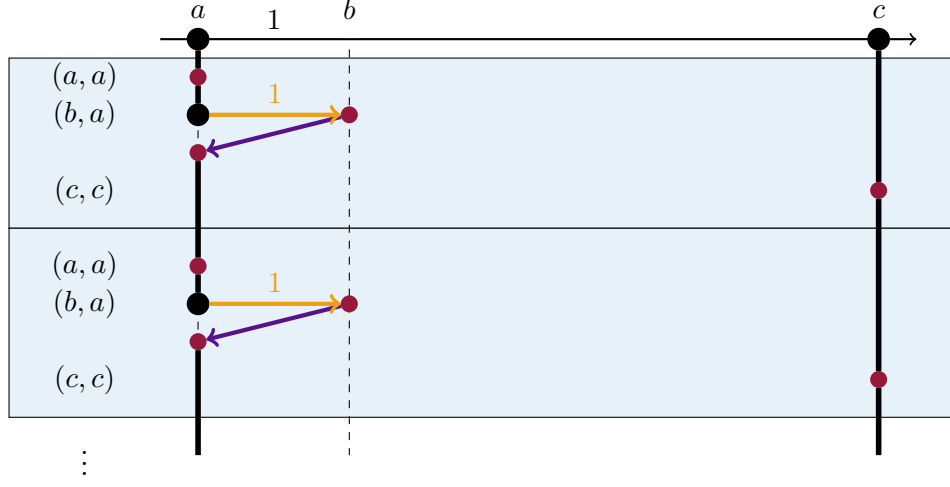


Figure 6.9 An optimal solution for the worst-case proposed in Section 6.5.

for a single iteration in Lemma 33 and then sum over all iterations to prove Theorem 31. Note that we start counting iterations by 0.

Lemma 33. *In the i th iteration, s_1 moves distance*

$$\alpha = \lambda \sum_{j=0}^{i-1} \left(\frac{\lambda}{1+\lambda} \right)^{2j}$$

to serve the first request of the iteration, and LAMBDABIASEDDC charges a total cost of

$$3 + 4\lambda + 2\lambda \sum_{j=1}^{2i-1} \left(\frac{\lambda}{1+\lambda} \right)^j + \lambda \left(\frac{\lambda}{1+\lambda} \right)^{2i}$$

for the iteration.

Proof. For $i = 0$, i.e. in the beginning of the sequence, s_1 is already on the request, and $\alpha = 0$. Now assume that for the $(i-1)$ th iteration

$$\alpha = \lambda \sum_{j=0}^{i-2} \left(\frac{\lambda}{1+\lambda} \right)^{2j}.$$

Since the last request of the previous iteration was c , we can assume that s_2 is on c and active. Furthermore, we know that for the first request of the iteration s_1 is predicted, and LAMBDABIASEDDC moves s_1 with speed

$$1 + \frac{1}{\lambda} = \frac{1+\lambda}{\lambda}$$

and s_2 with speed 1. As s_1 reaches the request first after travelling α units, s_2 must have been moved $\alpha(\frac{\lambda}{1+\lambda})$ units towards a . For the second request (b, a) , s_1 is active and

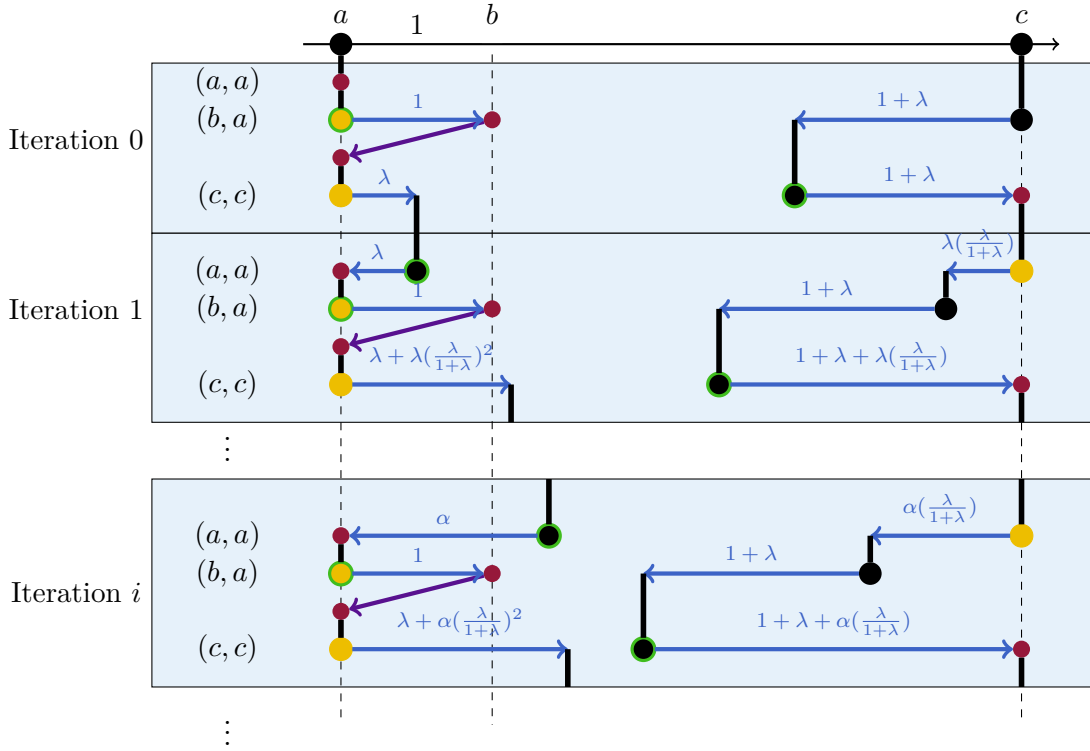


Figure 6.10 The schedule produced by LAMBDABIASEDDC on the worst-case instance given perfect predictions. The value of α is determined in Lemma 33.

predicted, and it moves distance 1 towards b while s_2 moves by distance $1 + \lambda$ towards b . Thus, s_2 is now

$$\alpha \left(\frac{\lambda}{1 + \lambda} \right) + (1 + \lambda)$$

units away from b . For the third request, which is (c, c) , s_2 is predicted, but s_1 is active. Therefore, s_2 moves back to c with speed $1 + \frac{1}{\lambda}$, whereas s_1 moves

$$\begin{aligned} \frac{\lambda}{1 + \lambda} \left(\alpha \left(\frac{\lambda}{1 + \lambda} \right) + 1 + \lambda \right) &= \lambda + \alpha \left(\frac{\lambda}{1 + \lambda} \right)^2 = \lambda + \lambda \sum_{j=0}^{i-2} \left(\frac{\lambda}{1 + \lambda} \right)^{2j+2} \\ &= \lambda + \lambda \sum_{j=1}^{i-1} \left(\frac{\lambda}{1 + \lambda} \right)^{2j} = \lambda \sum_{j=0}^{i-1} \left(\frac{\lambda}{1 + \lambda} \right)^{2j} \end{aligned}$$

units towards c . Since that is exactly the distance which s_1 has to move back to a in the next iteration i , we proved the first statement of the Lemma. For the second part we

sum up all the costs we observed in the first part, that is,

$$\begin{aligned}
& \underbrace{\alpha + \alpha \left(\frac{\lambda}{1+\lambda} \right)}_{\text{Cost for request } (a,a)} + \underbrace{1 + (1+\lambda)}_{\text{Cost for request } (b,a)} + \underbrace{\frac{\lambda}{1+\lambda} \left(\alpha \left(\frac{\lambda}{1+\lambda} \right) + 1 + \lambda \right) + \alpha \left(\frac{\lambda}{1+\lambda} \right) + 1 + \lambda}_{\text{Cost for request } (c,c)} \\
&= 3 + 3\lambda + \alpha + 2\alpha \left(\frac{\lambda}{1+\lambda} \right) + \alpha \left(\frac{\lambda}{1+\lambda} \right)^2.
\end{aligned}$$

Plugging in the value of α for iteration i yields the asserted result, that is

$$\begin{aligned}
& 3 + 3\lambda + \lambda \sum_{j=0}^{i-1} \left(\frac{\lambda}{1+\lambda} \right)^{2j} + 2\lambda \sum_{j=0}^{i-1} \left(\frac{\lambda}{1+\lambda} \right)^{2j+1} + \lambda \sum_{j=0}^{i-1} \left(\frac{\lambda}{1+\lambda} \right)^{2j+2} \\
&= 3 + 4\lambda + \lambda \sum_{j=1}^{i-1} \left(\frac{\lambda}{1+\lambda} \right)^{2j} + 2\lambda \sum_{j=0}^{i-1} \left(\frac{\lambda}{1+\lambda} \right)^{2j+1} + \lambda \sum_{j=0}^{i-2} \left(\frac{\lambda}{1+\lambda} \right)^{2j+2} + \lambda \left(\frac{\lambda}{1+\lambda} \right)^{2i} \\
&= 3 + 4\lambda + 2\lambda \sum_{j=1}^{i-1} \left(\frac{\lambda}{1+\lambda} \right)^{2j} + 2\lambda \sum_{j=0}^{i-1} \left(\frac{\lambda}{1+\lambda} \right)^{2j+1} + \lambda \left(\frac{\lambda}{1+\lambda} \right)^{2i} \\
&= 3 + 4\lambda + 2\lambda \sum_{j=1}^{2i-1} \left(\frac{\lambda}{1+\lambda} \right)^j + \lambda \left(\frac{\lambda}{1+\lambda} \right)^{2i}. \quad \square
\end{aligned}$$

Given the cost for a single iteration, we can now compute the limit of the competitive ratio of arbitrary many iterations and prove Theorem 31.

Proof of Theorem 31. First, we analyze how the cost of an iteration changes for an increasing number of iterations. Since only the last two parts of the total cost of an iteration of Lemma 33 depend on the index of that iteration, we compute its limit using the infinite geometric series, *i.e.*

$$\begin{aligned}
\lim_{i \rightarrow \infty} 2\lambda \sum_{j=1}^{2i-1} \left(\frac{\lambda}{1+\lambda} \right)^j + \lambda \left(\frac{\lambda}{1+\lambda} \right)^{2i} &= -2\lambda + \lim_{i \rightarrow \infty} 2\lambda \sum_{j=0}^{2i-1} \left(\frac{\lambda}{1+\lambda} \right)^j + \lambda \left(\frac{\lambda}{1+\lambda} \right)^{2i} \\
&= -2\lambda + 2\lambda \left(\frac{1}{1 - \frac{\lambda}{1+\lambda}} \right) \\
&= -2\lambda + 2\lambda(\lambda + 1) = 2\lambda^2.
\end{aligned}$$

Since the cost of the optimal solution is equal to the number of iterations, the limit of the competitive ratio for an increasing number of iterations approaches

$$\lim_{\ell \rightarrow \infty} \frac{1}{\ell} \left((3 + 4\lambda)\ell + \sum_{i=0}^{\ell} 2\lambda \sum_{j=1}^{2i-1} \left(\frac{\lambda}{1+\lambda} \right)^j + \lambda \left(\frac{\lambda}{1+\lambda} \right)^{2i} \right) = 3 + 4\lambda + 2\lambda^2,$$

and we conclude that the upper bound on the consistency ratio proved in Corollary 29 is tight. $\square$

6.5.2 Robustness

Showing tightness of the robustness ratio of LAMBDABIASEDDC works similar. In order to force that LAMBDABIASEDDC incurs large cost, we set up bad predictions, by always predicting the server that the optimal solution does not use for serving a request. That is, for the first two requests of an iteration we predict s_2 and for the third request s_1 . The resulting schedule of LAMBDABIASEDDC is displayed in Figure 6.11. We start again by computing the cost for a single iteration.

Lemma 34. *In the i th iteration, s_1 moves distance*

$$\alpha = \frac{1}{\lambda} \sum_{j=0}^{i-1} \left(\frac{1}{1+\lambda} \right)^{2j}$$

to serve the first request of the iteration, and LAMBDABIASEDDC charges a total cost of

$$3 + \frac{4}{\lambda} + \frac{2}{\lambda} \sum_{j=1}^{2i-1} \left(\frac{1}{1+\lambda} \right)^j + \frac{1}{\lambda} \left(\frac{1}{1+\lambda} \right)^{2i}$$

for the iteration.

Proof. For $i = 0$, *i.e.* in the beginning of the sequence, s_1 is already on the request, and $\alpha = 0$. Now assume that for the $(i - 1)$ th iteration

$$\alpha = \frac{1}{\lambda} \sum_{j=0}^{i-2} \left(\frac{1}{1+\lambda} \right)^{2j}.$$

Since the last request of the previous iteration was c , we can assume that s_2 is on c and active. Furthermore, we know that s_2 is predicted, and LAMBDABIASEDDC moves s_2 with speed 1 and s_1 with speed $1 + \lambda$. As s_1 reaches the request first after travelling α units, s_2 must have been moved $\alpha(\frac{1}{1+\lambda})$ units towards a . For the second request (b, a) , s_1 is active and s_2 is predicted. Hence, s_1 moves distance 1 towards b while s_2 moves by distance $1 + \frac{1}{\lambda}$ in the direction of b . Thus, s_2 is

$$\alpha \left(\frac{1}{1+\lambda} \right) + \left(1 + \frac{1}{\lambda} \right)$$

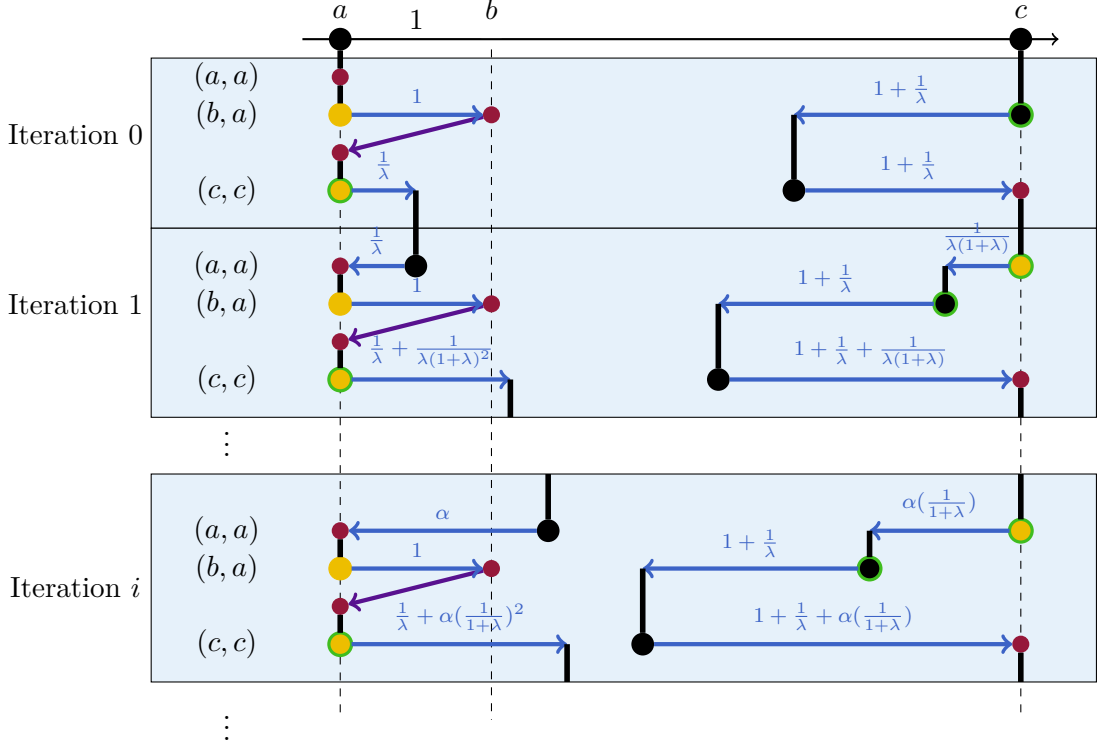


Figure 6.11 The schedule produced by LAMBDA BIASED DC given bad predictions. The value of α is determined in Lemma 34.

units away from b . For the third request, which is (c, c) , s_1 is predicted and active. Therefore, s_2 moves back to c with speed $1 + \lambda$, whereas s_1 moves with speed 1 by

$$\begin{aligned} \frac{1}{1+\lambda} \left(\alpha \left(\frac{1}{1+\lambda} \right) + 1 + \frac{1}{\lambda} \right) &= \frac{1}{\lambda} + \alpha \left(\frac{1}{1+\lambda} \right)^2 = \frac{1}{\lambda} + \frac{1}{\lambda} \sum_{j=0}^{i-2} \left(\frac{1}{1+\lambda} \right)^{2j+2} \\ &= \frac{1}{\lambda} + \frac{1}{\lambda} \sum_{j=1}^{i-1} \left(\frac{1}{1+\lambda} \right)^{2j} = \frac{1}{\lambda} \sum_{j=0}^{i-1} \left(\frac{1}{1+\lambda} \right)^{2j} \end{aligned}$$

units towards c . Since that is exactly the distance which s_1 has to move back to a in the next iteration i , we proved the first statement of the Lemma. For the second part we sum up all the costs we observed in the first part, *i.e.*

$$\begin{aligned} &\underbrace{\alpha + \alpha \left(\frac{1}{1+\lambda} \right)}_{\text{Cost for request } (a,a)} + \underbrace{1 + \left(1 + \frac{1}{\lambda} \right)}_{\text{Cost for request } (b,a)} + \underbrace{\frac{1}{1+\lambda} \left(\alpha \left(\frac{1}{1+\lambda} \right) + 1 + \frac{1}{\lambda} \right) + \alpha \left(\frac{1}{1+\lambda} \right) + 1 + \frac{1}{\lambda}}_{\text{Cost for request } (c,c)} \\ &= 3 + \frac{3}{\lambda} + \alpha + 2\alpha \left(\frac{1}{1+\lambda} \right) + \alpha \left(\frac{1}{1+\lambda} \right)^2 \end{aligned}$$

Plugging in the value of α for iteration i yields the stated result, that is

$$\begin{aligned}
& 3 + \frac{3}{\lambda} + \frac{1}{\lambda} \sum_{j=0}^{i-1} \left(\frac{1}{1+\lambda} \right)^{2j} + \frac{2}{\lambda} \sum_{j=0}^{i-1} \left(\frac{1}{1+\lambda} \right)^{2j+1} + \frac{1}{\lambda} \sum_{j=0}^{i-1} \left(\frac{1}{1+\lambda} \right)^{2j+2} \\
&= 3 + \frac{4}{\lambda} + \frac{1}{\lambda} \sum_{j=1}^{i-1} \left(\frac{1}{1+\lambda} \right)^{2j} + \frac{2}{\lambda} \sum_{j=0}^{i-1} \left(\frac{1}{1+\lambda} \right)^{2j+1} + \frac{1}{\lambda} \sum_{j=0}^{i-2} \left(\frac{1}{1+\lambda} \right)^{2j+2} + \frac{1}{\lambda} \left(\frac{1}{1+\lambda} \right)^{2i} \\
&= 3 + \frac{4}{\lambda} + \frac{2}{\lambda} \sum_{j=1}^{i-1} \left(\frac{1}{1+\lambda} \right)^{2j} + \frac{2}{\lambda} \sum_{j=0}^{i-1} \left(\frac{1}{1+\lambda} \right)^{2j+1} + \frac{1}{\lambda} \left(\frac{1}{1+\lambda} \right)^{2i} \\
&= 3 + \frac{4}{\lambda} + \frac{2}{\lambda} \sum_{j=1}^{2i-1} \left(\frac{1}{1+\lambda} \right)^j + \frac{1}{\lambda} \left(\frac{1}{1+\lambda} \right)^{2i}. \quad \square
\end{aligned}$$

Finally, given the cost for a single iteration, we can now compute the limit of the competitive ratio of arbitrary many iterations and prove Theorem 32.

Proof of Theorem 32. First, we analyze how the cost of an iteration changes for an increasing number of iterations. Since only the last two parts of the total cost of an iteration depend on the index of that iteration, we compute its limit using the infinite geometric series, *i.e.*

$$\begin{aligned}
\lim_{i \rightarrow \infty} \frac{2}{\lambda} \sum_{j=1}^{2i-1} \left(\frac{1}{1+\lambda} \right)^j + \frac{1}{\lambda} \left(\frac{1}{1+\lambda} \right)^{2i} &= -\frac{2}{\lambda} + \lim_{i \rightarrow \infty} \frac{2}{\lambda} \sum_{j=0}^{2i-1} \left(\frac{1}{1+\lambda} \right)^j + \frac{1}{\lambda} \left(\frac{1}{1+\lambda} \right)^{2i} \\
&= -\frac{2}{\lambda} + \frac{2}{\lambda} \left(\frac{1}{-\frac{1}{1+\lambda}} \right) \\
&= -\frac{2}{\lambda} + \frac{2(1+\lambda)}{\lambda(1+\lambda) - \lambda} = \frac{2}{\lambda^2}.
\end{aligned}$$

Since the cost of the adversary is equal to the number of iterations, the limit of the competitive ratio for an increasing number of iterations approaches

$$\lim_{\ell \rightarrow \infty} \frac{1}{\ell} \left(\left(3 + \frac{4}{\lambda} \right) \ell + \sum_{i=0}^{\ell} \frac{2}{\lambda} \sum_{j=1}^{2i-1} \left(\frac{1}{1+\lambda} \right)^j + \frac{1}{\lambda} \left(\frac{1}{1+\lambda} \right)^{2i} \right) = 3 + \frac{4}{\lambda} + \frac{2}{\lambda^2},$$

and we conclude that the upper bound on the robustness ratio proved in Theorem 30 is tight. $\square$

Chapter 7

Experiments

This is the final Chapter of the main part of the thesis. In the previous chapters, we established theoretical properties of different algorithms for various server problems. Now we empirically evaluate some of them. We focus on the k -server problem on the line and the 2-taxi problem on the line.

Setting The software is written in *Rust*¹ and can be found on *GitHub*². Rust enables a fast and parallel execution of the necessary algorithms and comfortable way to create command-line interfaces. The software is highly configurable, *e.g.* number of servers, length of the instances etc. The script for plotting the simulation results is written in *Python*³.

Instance generation To obtain instances, we use the following two techniques.

- One method is to sample the sequence of requests and the initial configuration from a *uniform distribution* between a given lower and upper bound. For k -taxi instances, we use the configurable parameter $r \in [0, 1]$ to set the probability that a request is a relocation request, and in that case sample a second value from the uniform distribution.
- Another method is using a real-world dataset. For generating sequences of *simple* requests, we use the *BrightKite* dataset [12]. It is composed of checkins into a social network from different users with the locations where these checkins happened. We will treat each sequence of a distinct user as an instance by converting the location to its latitude and apply some scaling. We will use this dataset only for the k -server problem on the line. It contains 402 instances with 2100 requests each. This dataset is commonly used for evaluating caching algorithms with untrusted predictions [3, 27].

¹<https://www.rust-lang.org/>

²<https://github.com/Mountlex/kserver>

³<https://www.python.org/>

Optimal solution Given an instance, we compute the optimal solution using a network flow algorithm and the proposed flow network from Section 2.3. We have also seen that there are optimal solutions with overtaking, and the flow network does not prevent these. However, it is easy to see that if a server s_i passes by s_j , we can serve requests with s_i instead of s_j and vice versa for the remaining sequence, and obtain a solution with equal cost.

Prediction generation Based on an instance and an associated solution, we compute predictions. Let OPT be the cost of the optimal solution for the instance, and $o_1, \dots, o_n \in \{1, \dots, k\}$ the sequence of servers the optimal solution uses to solve the instance. Afterwards we iteratively alter this sequence, *i.e.* use a different server for a request, and then compute the error η of the obtained prediction. We also use two configurable parameters b and p , where b is called *bin size* and p is the number of predictions we want to generate per instance. When we have computed a prediction, we compute its *bin* i by

$$i = \left\lfloor \frac{\eta/\text{OPT}}{b} \right\rfloor.$$

If $i > p$, we do not care about the prediction. Our goal is to have p predictions for an instance which are well-distributed, that is, we want a prediction for the first p bins with bin size b . We put the computed prediction into its bin, until in each bin there is at least a fixed number of predictions. Then, we randomly select a prediction from each bin. Note that the sequence of requests can have an arbitrary length n , and there are k^n possible predictions. In order to circumvent this exponential growth, we randomly select different predictions in a certain way, such that we might get all necessary predictions if and only if they can be found in general. Of course, this is not provable, but it works fine in practice. For more details on the prediction generation we refer to the source code. If this procedure does not find all necessary predictions, we discard the instance completely. Also, we always use the optimal solution as a prediction, since it is the unique prediction with $\eta = 0$. We can also observe that for different values of k resp. r , we can find different prediction qualities. For the presented results, we use the parameters displayed in Table 7.2.

Algorithms The evaluated algorithms are displayed with their corresponding theoretical bounds in Table 7.1. For the parameterized algorithms, we use 21 evenly distributed values for $\lambda \in [0, 1]$. We chose this number since it appeared to be a good trade-off between running time, memory consumption and precision. Also, this distribution gives proper rational values.

Simulation Given an instance with multiple predictions, we simulate the algorithms and compute their costs for different values of $k \geq 2$, *i.e.* number of servers. We generate 2000

Problem	Algorithm	Competitive ratio	Robustness
k -Server on the line	DC [13]	k	k
	LAMBDAADC	$(1 + (k - 1)\lambda)(1 + \frac{2\eta}{\text{OPT}})$	$1 + \frac{1}{\lambda}$ for $k = 2$
2-Taxi on the line	BIASEDDC [15]	9	9
	LAMBDABIASEDDC	$(3 + 4\lambda + 2\lambda^2)(1 + \frac{2\eta}{\text{OPT}})$	$3 + \frac{4}{\lambda} + \frac{2}{\lambda^2}$

Table 7.1 The theoretical efficiency bounds of the evaluated algorithms.

Instance type	Uniformly		BrightKite	
Parameter	p	b	p	b
$k = 2$	22	0.25	22	0.25
$k = 3$	22	0.5	22	0.5
$k = 5$	25	1	22	1
$k = 10$	27	3	22	1

(a) Parameters for predictions of the k -server problem for different k .

Instance type	Uniformly	
Parameter	p	b
$r = 0.0$	22	0.25
$r = 0.4$	24	0.5
$r = 0.8$	14	0.5
$r = 1.0$	15	0.5

(b) Parameters for predictions of the 2-server problem for different values of r .**Table 7.2** Parameters of the prediction generation for both problems.

instances for each value of k , where each instance is composed of 300 requests. When generating 2-taxi instances, we insert a simple request (x, x) right before a relocation request (x, y) , as we argued in Section 6.1. Hence, the exact length of the sequence may be longer for those settings, but the cost for both the algorithm and the optimal solution do not change. The number of predictions per instance, *i.e.* p , depends on the chosen parameters discussed in the previous paragraph.

Visualizations We combine all results for a specific value of k resp. r in two plots. The first plot displays the relationship between the relative error of the prediction and the averaged competitive ratio for fixed values of λ . Since plotting all 21 different values of λ would be confusing, we select 6 evenly spaced values for the visualization. Note that the line for $\lambda = 1$ corresponds to DC resp. BIASEDDC. The second plot switches λ and the relative error, meaning that it illustrates the relationship between λ and the averaged competitive ratio for fixed intervals of relative errors. Note that these intervals do not intersect, *i.e.* the first line averages over all predictions with $\eta = 0$, the next for all $\eta/\text{OPT} \in (0, x]$, then for $\eta/\text{OPT} \in (x, y]$ and so on. The size of those intervals are selected in a way that the overall plot is still neat, and depends on the value of k .

7.1 The k -server problem on the line

We selected several representative settings for the k -server problem on the line and evaluated them as described above. We want to stress two features which can be

identified in all plots for the k -server problem. First, the competitive ratio for $\lambda = 1$ is constant in the left figures and equal to the competitive ratio of DC, since in both cases the algorithms ignore the prediction. This fact can also be seen in the right figures where regardless of the error all lines are incident for $\lambda = 1$. The other characteristic is the consistency, which is at most $1 + \lambda(k - 1)$, hence 1 for $\lambda = 0$. We can see the consistency is described by the orange line in the right plots, which always interpolates between the optimal competitive ratio 1 for $\lambda = 0$ and the competitive ratio of DC for $\lambda = 1$.

7.1.1 Uniformly generated instances

Figure 7.1 displays the results for $k = 2$ resp. $k = 10$. As expected, uniformly generated instances yield evenly distributed results, that is, the change of the competitive ratio is clearly related to the change of λ and the relative error of the prediction. Also, the lines are very smooth, since we are able to generate many instances.

For $k = 2$ we can see in the right plot that for an increasing error the line approaches the shape of the robustness bound, *i.e.* $1 + \frac{1}{\lambda}$. This goes with our observation that trusting the predictions gives a larger competitive ratio if these are bad. For $\eta/\text{OPT} > 10$ the average competitive ratio is entirely above the line for DC, hence we cannot select a value of λ to improve on it. Interestingly, for $k = 10$ the simulation is not able to find prediction with a larger relative error, hence the line for the largest error is only for very small values of λ above the competitive ratio of DC in the right plot of Figure 7.1b. Thus, we can select $\lambda > 0.1$ and ensure an average improvement upon DC. There is still the possibility that there exist an instance and a prediction with a larger relative error. However, they seem to be quite rare and hence did not appear in the uniform samples.

7.1.2 BrightKite instances

For BrightKite instances, things look different. Since we only have 402 instances, the lines are not as smooth as for uniformly generated instances. In Figure 7.2a we can clearly see in the right plot that for two servers, we do better than DC for any relative error the simulation finds and any λ , even if we completely follow the predictions ($\lambda = 0$). A reason for that is, that for two servers there are not many possible predictions, and it seems that having at least *some* prediction already improves the average performance compared to DC for real-world data.

However, looking at ten servers in Figure 7.2b, there are much more possible predictions, and hence the only possibility to increase upon DC with $\lambda = 0$ is having perfect predictions, that is, $\eta/\text{OPT} = 0$. Yet we can identify an improvement compared to DC for $\lambda > 0.1$ and relative error not too big ($\eta/\text{OPT} \leq 16$). Interestingly, these observations are contrary to the uniform case regarding the number of servers, and indicate the differences between real-world data and randomly generated data.

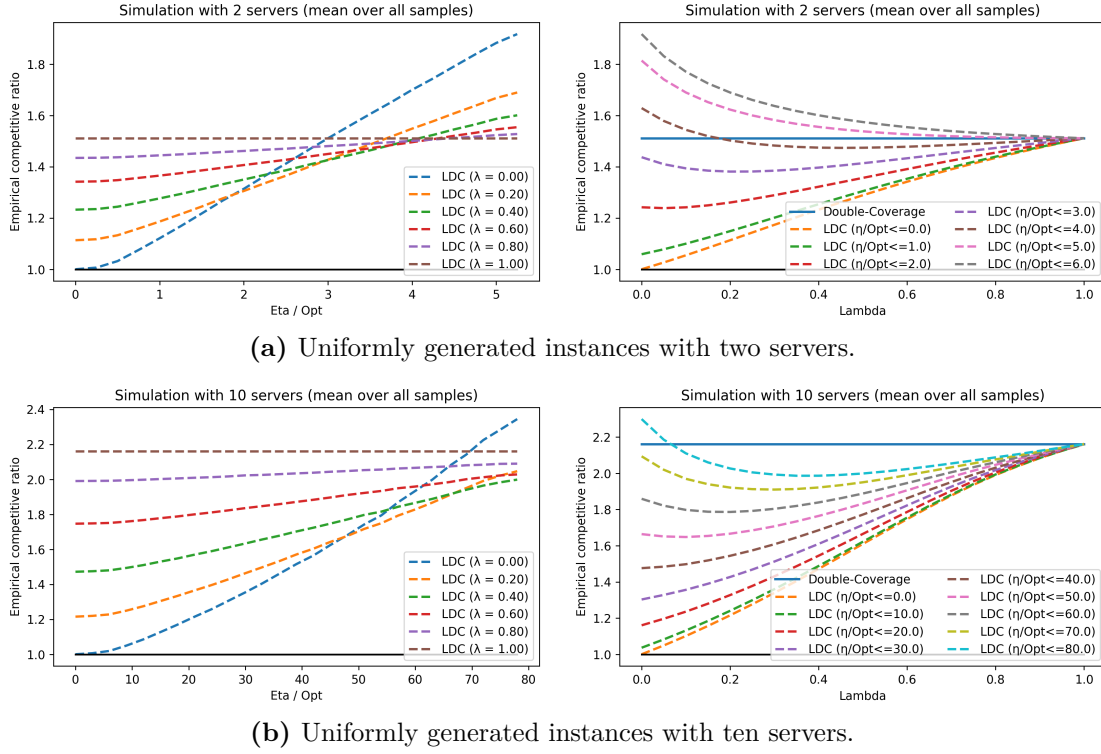
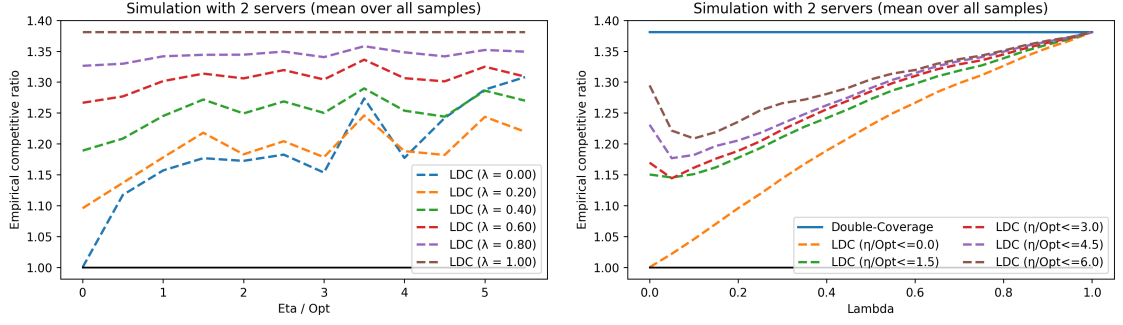


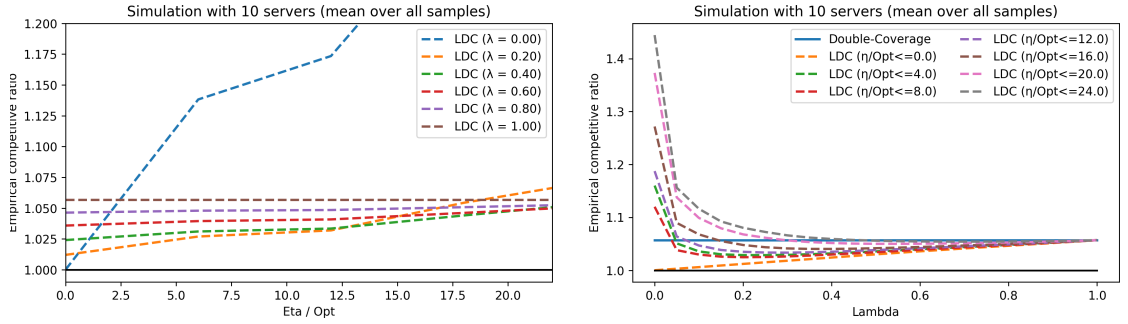
Figure 7.1 Results for uniformly generated instances.

7.2 The 2-taxi problem on the line

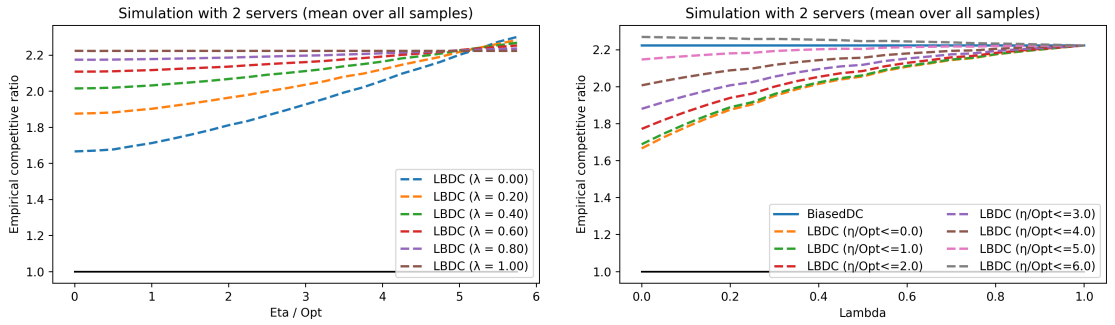
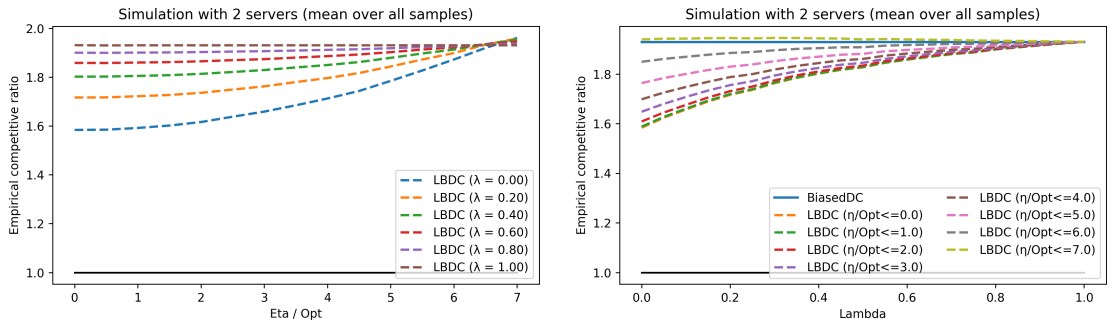
When looking at the 2-taxi problem on the line, we cannot vary the number of servers, but the expected amount of relocation requests, *i.e.* the value of r . We can observe in Figure 7.3 that the average competitive ratio decreases for an increasing number of relocation requests. Besides that, the results look very similar for different values of r , hence we select the two representative values $r = 0.4$ (Figure 7.3a) and $r = 1.0$ (Figure 7.3b). In contrast to the k -server problem, we do not have a consistency which is equal to 1 for a certain value of λ . However, we can observe in the left plots that LAMBDABIASEDDC improves in average upon BIASEDDC for all values of λ if $\eta/\text{OPT} \leq 5$ for $r = 0.4$ resp. if $\eta/\text{OPT} \leq 6.5$ for $r = 1.0$. However, for the largest found error, the average competitive ratio of LAMBDABIASEDDC for any λ is above the average competitive ratio of BIASEDDC, as we can see in the right plots.



(a) BrightKite-instances with two servers.



(b) BrightKite-instances with ten servers.

Figure 7.2 Results for instances derived from the BrightKite-dataset.(a) 2-Taxi instances with $r = 0.4$.(b) 2-Taxi instances with $r = 1.0$, *i.e.* only relocation requests.**Figure 7.3** Results for the 2-taxi problem.

Chapter 8

Conclusion and open questions

In this thesis, we proposed different algorithms using predictions for various cases of server problems. We considered the k -server problem, which is a well-studied, historical and fundamental online problem. Yet there are many questions open, especially regarding the gap between lower and upper bounds for general metric spaces. However, for the k -server problem on the line this question is answered and an optimal online algorithm exists with respect to the competitive ratio.

A more recent development is the field of learning-augmented online algorithms. It addresses on the one hand the problem of the pessimistic flavor of classical online algorithms, which make no assumption on the input and hence heavily depend on worst-case scenarios. On the other hand, it combines machine learning with online problems while still achieving theoretical efficiency bounds. These often break existing lower bounds of online problems and empirical experiments also indicate real-world performance improvements.

The main effort in this thesis was applying this new framework to various server problems. We started this process in Chapter 3 by introducing predictions for instances of the 2-server problem on the line. Using those, we proposed a speed-adjusted modification DC and showed that LAMBDADC is $(1 + \lambda)(1 + 2\eta/\text{OPT})$ -competitive and $(1 + \frac{1}{\lambda})$ -robust. Hence, given good predictions, LAMBDADC breaks the long-standing lower bound of 2 for 2 servers. In the final part of the Chapter, we also proved that LAMBDADC achieves the best possible trade-off between consistency and robustness for any learning-augmented online algorithm.

Based on that result, we extended LAMBDADC to an arbitrary number of servers in Chapter 4. For this case, we proved a competitive ratio equal to $(1 + (k - 1)\lambda)(1 + 2\eta/\text{OPT})$, which again breaks the lower bound of k for having good predictions. However, we were not able to show a general robustness bound in this case. Yet we illustrated that the competitive ratio is bounded independently of η for $\lambda > (k - 2)/(k - 1)$.

We further generalized the setting to tree metrics in Chapter 5. There, we again propose a speed-adjusted modification of DC, which turned out to be slightly more complex than LAMBDADC on the line, because we have no ordering of the servers

on a tree. Yet again, we were able to break the lower bound of k for deterministic online algorithms for the k -server problem by establishing a proof for a competitive ratio of $(1 + (k - 1)\lambda)(1 + 2\eta/\text{OPT})$ for LAMBADC on trees. However, we were not able to show any bound on the robustness ratio for $k > 2$, but still have the $1 + \frac{1}{\lambda}$ bound for the two server case.

The last generalization we considered is the k -taxi problem, which adds the relocation request type to the problem. In Chapter 6 we augmented the 9-competitive algorithm BIASEDDC for the hard 2-server taxi problem with predictions. For the resulted algorithm LAMBDA BIASEDDC, we demonstrated bounds on the competitive ratio with respect to the prediction error and the robustness ratio. The former also circumvents the previously known exact lower bound of 9, if it receives good predictions. We also illustrated that the proved consistency and robustness bounds are tight by proposing a worst-case example instance and analyzing LAMBDA BIASEDDC on that.

Finally, we empirically evaluated LAMBADC and LAMBDA BIASEDDC on randomly generated and real-world instances. The findings emphasize our theoretical results, and showed that for having a decent relative prediction error, there is a wide range of values of λ for that in average the learning-augmented algorithms outperform their deterministic bases. The findings also displayed that for randomly generated instances do not occur large prediction errors. We can conclude that the combination of a worst-case instance and an associated worst-case prediction, which is handled by the robustness case, is unlikely to occur in practice. We also saw that the relation of η/OPT , λ and the average competitive ratio does not significantly change when increasing the number of servers or changing the amount of relocation requests in the case of randomly generated instances.

There are still some open questions. The most important one is how the robustness of LAMBADC behaves for the k -server problem on the line for $k > 2$ and $\lambda \in (0, 1]$. If LAMBADC turns out to be not robust for all $\lambda \in (0, 1]$, it would be interesting if it is possible to modify LAMBADC in order to be robust. Regarding the k -taxi problem, it is unknown if LAMBDA BIASEDDC is pareto-optimal and in that context, if the current speeds for the servers are optimal. Another direction is improving the bound between the prediction's cost and the optimal solution with respect to the prediction error. Also, it would be interesting if it is possible to give a general upper bound for the relative prediction error η/OPT . Such a bound could also solve the question on whether LAMBADC is robust for more than two servers. The empirical results also indicate that such a bound may exist.

Bibliography

- [1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice hall, 1993.
- [2] S. Angelopoulos, C. Dürr, S. Jin, S. Kamali, and M. Renault. “Online Computation with Untrusted Advice”. In: *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020.
- [3] A. Antoniadis, C. Coester, M. Elias, A. Polak, and B. Simon. “Online metric algorithms with untrusted predictions”. In: *arXiv preprint arXiv:2003.02144* (2020).
- [4] N. Bansal, N. Buchbinder, A. Madry, and J. Naor. “A polylogarithmic-competitive algorithm for the k-server problem”. In: *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. IEEE. 2011, pp. 267–276.
- [5] N. Bansal, M. Eliáš, and G. Koumoutsos. “Weighted k-server bounds via combinatorial dichotomies”. In: *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2017, pp. 493–504.
- [6] N. Bansal, M. Eliáš, G. Koumoutsos, and J. Nederlof. “Competitive algorithms for generalized k-server in uniform metrics”. In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2018, pp. 992–1001.
- [7] Y. Bartal, B. Bollobás, and M. Mendel. “Ramsey-type theorems for metric spaces with applications to online problems”. In: *Journal of Computer and System Sciences* 72.5 (2006), pp. 890–921.
- [8] Y. Bartal, N. Linial, M. Mendel, and A. Naor. “On metric Ramsey-type phenomena”. In: *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. 2003, pp. 463–472.
- [9] H.-J. Böckenhauer, D. Komm, R. Kráľovič, R. Kráľovič, and T. Mömke. “On the advice complexity of online problems”. In: *International Symposium on Algorithms and Computation*. Springer. 2009, pp. 331–340.
- [10] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998, pp. I–XVIII, 1–414.
- [11] A. Chiplunkar and S. Vishwanathan. “Randomized Memoryless Algorithms for the Weighted and the Generalized k-server Problems”. In: *ACM Transactions on Algorithms (TALG)* 16.1 (2019), pp. 1–28.

- [12] E. Cho, S. A. Myers, and J. Leskovec. “Friendship and mobility: user movement in location-based social networks”. In: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2011, pp. 1082–1090.
- [13] M. Chrobak, H. Karloof, T. Payne, and S. Vishwnathan. “New results on server problems”. In: *SIAM Journal on Discrete Mathematics* 4.2 (1991), pp. 172–181.
- [14] M. Chrobak and L. L. Larmore. “An optimal on-line algorithm for k servers on trees”. In: *SIAM Journal on Computing* 20.1 (1991), pp. 144–148.
- [15] C. Coester and E. Koutsoupias. “The online k-taxi problem”. In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. 2019, pp. 1136–1147.
- [16] D. Coppersmith, P. Doyle, P. Raghavan, and M. Snir. “Random walks on weighted graphs and applications to on-line algorithms”. In: *Journal of the ACM (JACM)* 40.3 (1993), pp. 421–453.
- [17] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. “Competitive paging algorithms”. In: *Journal of Algorithms* 12.4 (1991), pp. 685–699.
- [18] S. Gollapudi and D. Panigrahi. “Online algorithms for rent-or-buy with expert advice”. In: *International Conference on Machine Learning*. 2019, pp. 2319–2327.
- [19] E. F. Grove. “The harmonic online k-server algorithm is competitive”. In: *Proceedings of the twenty-third annual ACM symposium on Theory of computing*. 1991, pp. 260–266.
- [20] E. Koutsoupias. “The k-server problem”. In: *Computer Science Review* 3.2 (2009), pp. 105–118.
- [21] E. Koutsoupias and C. Papadimitriou. “On the k-server conjecture”. In: *Proceedings of the twenty-sixth annual ACM symposium on Theory of Computing*. 1994, pp. 507–511.
- [22] E. Koutsoupias and D. S. Taylor. “The CNN problem and other k-server variants”. In: *Annual Symposium on Theoretical Aspects of Computer Science*. Springer. 2000, pp. 581–592.
- [23] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. “The case for learned index structures”. In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 489–504.
- [24] R. Kumar, M. Purohit, A. Schild, Z. Svitkina, and E. Vee. “Semi-Online Bipartite Matching”. In: *10th Innovations in Theoretical Computer Science Conference (ITCS 2019)*. Vol. 124. Leibniz International Proceedings in Informatics (LIPIcs). 2018, 50:1–50:20.

-
- [25] R. Kumar, M. Purohit, Z. Svitkina, and E. Vee. “Interleaved Caching with Access Graphs”. In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2020, pp. 1846–1858.
 - [26] S. Lattanzi, T. Lavastida, B. Moseley, and S. Vassilvitskii. “Online Scheduling via Learned Weights”. In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2020, pp. 1859–1877.
 - [27] T. Lykouris and S. Vassilvitskii. “Competitive Caching with Machine Learned Advice”. In: *International Conference on Machine Learning*. 2018, pp. 3296–3305.
 - [28] M. Manasse, L. McGeoch, and D. Sleator. “Competitive algorithms for on-line problems”. In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM. 1988, pp. 322–333.
 - [29] A. M. Medina and S. Vassilvitskii. “Revenue optimization with approximate bid predictions”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Curran Associates Inc. 2017, pp. 1856–1864.
 - [30] M. Mitzenmacher. “A model for learned bloom filters and optimizing by sandwiching”. In: *Advances in Neural Information Processing Systems*. 2018, pp. 464–473.
 - [31] M. Mitzenmacher. “Scheduling with Predictions and the Price of Misprediction”. In: *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020.
 - [32] M. Purohit, Z. Svitkina, and R. Kumar. “Improving online algorithms via ml predictions”. In: *Advances in Neural Information Processing Systems*. 2018, pp. 9661–9670.
 - [33] Y. Rabani, Y. Ravid, and A. Fiat. “Competitive k-server algorithms”. In: *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Oct. 1990, 454–463 vol.2.
 - [34] P. Raghavan and M. Snir. “Memory versus randomization in on-line algorithms”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1989, pp. 687–703.
 - [35] D. Rohatgi. “Near-Optimal Bounds for Online Caching with Machine Learned Advice”. In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2020, pp. 1834–1845.