Dissertation zur Erlangung des Doktorgrades der Technischen
Fakultät der Albert-Ludwigs-Universität Freiburg im Breisgau

Serving Online Requests with Mobile Resources

Abdolhamid Ghodselahi

Department of Computer Science
University of Freiburg
July 2018

Serving Online Requests with Mobile Resources

Abdolhamid Ghodselahi

Dekan:	Prof. Dr. Oliver Paul
Erstgutachter:	Prof. Dr. Fabian Kuhn
Zweitgutachter:	Prof. Dr. Christian Scheideler
Abgabedatum:	4. June 2018
Disputation:	9. July 2018

Abstract

Resource allocation problems have a large variety of applications in different areas of computer science and operations research. A resource allocation problem seeks to find an *optimal allocation* of a given type of expensive or limited resource to a set of clients that request the services of the given resource. Some of these problems have an *online nature*: The requests sequence is not revealed at the beginning, but the requests arrive in an *online fashion*. An algorithm for an online resource allocation problem must make its decision in response to a newly arriving request in an *online fashion*, i.e., typically before the subsequent request arrives. Depending on the definition of an online resource allocation problem, it might be allowed to postpone serving a request or to change an already made decision. However in this case, postponing a service and changing a decision come at a cost.

In this thesis, we also study online allocation problems in the distributed setting, where in contrast to a centralized system, there is no central unit that controls everything and that is aware of the global state of the system. In addition to the uncertainty about the request sequence arising from the online arrivals, there is also uncertainty at each node in the network because the node does not directly learn about requests arriving in other parts of the network. The nodes of a distributed system therefore need to communicate in order to coordinate their actions and one typically assumes that this communication does not come for free.

Two online problems are mainly studied in this thesis. First, we consider the *distributed queuing problem* as a basic problem that coordinates mutually exclusive access to shared data in distributed systems. We devise a randomized distributed queuing algorithm with an expected competitive ratio of $O(\log n)$ on general network topologies. We utilize the well-known probabilistic tree embedding of Fakcharoenphol, Rao, and Talwar [STOC 2003] that approximates the distances of a general metric space by mapping it to a special family of tree topologies known as hierarchically well-separated trees and often just referred to as *HSTs*. Our randomized distributed queuing algorithm is obtained by running the ARROW algorithm—a well-known distributed queuing algorithm—on top of the HST that is produced by applying the above embedding to the distances of the underlying network. It is shown that (under some assumptions) the simple and elegant ARROW algorithm outperforms all existing significantly more complicated distributed queueing algorithms. The second main problem that is studied in a centralized setting is the *online facility location problem*. We introduce the *online mobile facility location (OMFL)* problem, where the facilities are mobile. A lower bound for the OMFL problem that even holds on uniform metrics is provided. A natural approach to solve the OMFL problem for general metric spaces is to again use the above embedding into an HST and to directly solve the OMFL problem on HSTs. In this thesis, we provide a first step in this direction by solving a generalized version of the OMFL problem on uniform metrics. A simple deterministic online algorithm is devised and a tight analysis is provided for the algorithm. The second step remains as an open question. We further introduce and study another variant of the OMFL problem that is closer to the k-server problem, arguably one of the most influential problem in the area of online algorithms and *competitive analysis*.

Zusammenfassung

Probleme zur Zuteilung von Ressourcen finden eine Vielzahl von Anwendungen in unterschiedlichen Bereichen der Informatik und der Operations Research. Bei einem Problem zur Zuteilung von Ressourcen geht es darum bestimmte Arten von kostspieligen oder begrenzten Ressourcen optimal einer Menge von Klienten zuzuordnen, welche Anfragen an die entsprechende Ressource stellen. Manche dieser Probleme sind von ihrer Natur her *online*, dass heißt die Reihenfolge in der die Anfragen gestellt werden ist anfangs unbekannt. Ein Algorithmus für ein *Online*-Problem zur Ressourcenzuteilung muss seine Entscheidung bei jeder neu ankommenden Anfrage *online* treffen, also typischerweise bevor die nächste Anfrage ankommt. Abhängig von der Definition des *Online*-Problems zur Ressourcenzuteilung kann es erlaubt sein das Bedienen einer Anfrage zu verschieben oder eine schon getroffene Entscheidung zu ändern. In diesem Fall werden dem Verschieben oder Verändern einer Entscheidung Kosten zugeordnet.

In dieser Thesis untersuchen wir zudem Online-Zuteilungsprobleme in verteilten Systemen, bei welchem es, im Gegensatz zu einem zentralisierten Problem, keine zentrale Steuerungseinheit gibt die den globalen Zustand des Systems kennt. Zusätzlich zur Unsicherheit der Anfragereihenfolge bei einem Online-Problem, gibt es hier Unsicherheit an jedem Knoten des Netzwerkes eines verteilten Systems, da einzelnen Knoten die Anfragen welche an andere Knoten gestellt werden zunächst unbekannt sind. Aus diesem Grund müssen die Knoten eines verteilten Systems miteinander kommunizieren um Ihre Handlungen abzustimmen, wobei man typischerweise annimmt das diese Kommunikation Kosten verursacht.

In dieser Thesis werden hauptsächlich zwei Online Probleme untersucht. Zuerst betrachten wir verteilte Queuing-Probleme als grundlegendes Problem um wechselseitig exklusiven Zugriff auf gemeinsame Daten eines verteilten Systems zu koordinieren. Wir entwerfen einen randomisierten verteilten queuing Algorithmus, mit erwartetem kompetitivem Faktor von $O(\log n)$ auf allgemeinen Netzwerk Topologien. Wir benutzen die bekannte Probabilistic Tree Embedding von Fakcharoenphol, Rao, and Talwar [STOC 2003], welche die Distanzen eines allgemeinen metrischen Raumes approximiert indem wir diese auf eine spezielle Familie von Baum-Topologien abbilden die als Hierarchically Well-Separated Trees bekannt ist und oft mit *HSTs* abgekürzt wird. Wir erhalten unseren randomisierten, verteilten Queuing-Algorithmus indem wir den ARROW Algorithmus—ein bekannter verteilter Algorithmus—auf dem HST ausführen, welches wir erhalten indem wir die oben genannten Einbettung der Distanzen auf das zugrunde liegende Netzwerk anwenden. Es wird gezeigt, dass der einfache und elegante ARROW Algorithmus (unter einigen Annahmen) eine bessere Performanz bietet als alle bisher existierenden, wesentlich komplizierteren verteilten Queuing-Algorithmen. Das zweite Hauptproblem ist das *Online Facility Location* Problem, welches wir in in einem zentralisierten Szenario untersuchen. Wir führen das *Online Mobile Facility Location (OMFL)* Problem ein bei welchem die facilities mobil sind. Wir geben eine untere Schranke für das OMFL Problem, welches sogar in metrischen Räumen gilt in dem sich alle Distanzen gleichen. Ein natürlicher Ansatz um das OMFL Problem für allgemeine metrische Räume zu lösen, ist die oben gegebene Einbettung in einen HST erneut zu benutzen um das OMFL Problem direkt anhand von HSTs zu lösen. In dieser Thesis geben wir einen ersten Schritt in diese Richtung, indem wir eine verallgemeinerte Version des OMFL Problems auf metrischen Räumen mit uniformen Distanzen lösen. Wir leiten einen einfachen, deterministischen Algorithmus her und geben eine scharfe Analyse dieses Algorithmus. Der zweite Schritt verbleibt als offene Fragestellung. Des weiteren definieren und untersuchen wir eine Variante des OMFL Problems welches nahe verwandt ist mit dem k-Server Problem, eines der wohl einflussreichsten Probleme im Bereich der Online-Algorithmen und der kompetitiven Analyse.

Acknowledgements

To my advisor, Fabian Kuhn: because I owe it all to you. I have always had the signal honor of being your PhD student. I am happy that I have learned many things from you and I wish that I could have learned more. I admire you for the way you deal with the situations. Many thanks.

I would also like to thank my co-examiner Christian Scheideler for reviewing my thesis. It has been my honor to have him in the thesis defense committee.

I am grateful to Philipp Schneider, my colleague and office mate, who has translated the abstract of my thesis into German. I am also grateful to my colleague Yannic Maus who gave me useful comments, especially when I was preparing my presentations. I want to thank Mohamad Ahmadi who has been beyond just a colleague for me. I will miss our interesting discussions. I am also grateful to all my colleagues in the chair of Algorithms and Complexity in the university of Freiburg.

A very special gratitude goes to our system admin Wolfgang Paulat for his unfailing support and assistance not only for the technical issues I had, but also for any questions I had in general. I was always happy to have such a warm-hearted colleague in our chair.

My forever interested, encouraging, and always enthusiastic father: you were always keen to know what I was doing and how I was proceeding. I miss your screams of joy whenever a significant momentous was reached. My mother: I have always been very proud of you. I owe you a lot and I am sorry that I cannot be with you in hard times.

And finally, last but by no means least, I also want to thank my brothers, who have been providing me through moral and emotional support in my life.

Contents

1 Online Allocation Problems **1**

1.1 Allocation Problems 1

1.2 Online Allocation Problems 2

1.2.1 Formal Definitions 3

1.3 Centralized Service for Sequential Requests 4

1.4 Models 5

1.4.1 Centralized vs. Distributed Systems 5

1.4.2 Network Model 5

1.4.3 Communication Model 6

1.4.4 Synchrony vs. Asynchrony 6

1.4.5 Input Sequence 7

1.4.6 Hierarchically Well-Separated Trees 7

1.5 Distributed Service for Dynamic Requests 8

1.6 Request-Answer Games 9

1.6.1 Adversary Models 10

1.6.2 Competitive Analysis 10

1.7 An Abstract Online Allocation Problem 11

1.8 Our Results 13

I Distributed Service for Dynamic Requests **15**

2 The Distributed Queuing Problem: Preliminaries **17**

2.1 Model and Problem Statement 17

2.2 Arrow Algorithm 18

2.3 Greedy Nature of Arrow Algorithm 20

2.3.1 Synchronous Executions 20

2.3.2 Asynchronous Executions 21

2.4 Cost Model 24

2.5 Further Related Work 25

3 The Distributed Queuing Problem: Dynamic Requests **29**
3.1 Introduction 29
3.2 One-Shot Executions 31
3.3 Preliminaries 32
3.4 Analysis of the Optimal Offline Cost 34
3.4.1 Characterizing ARROW By A Hierarchical Partition of $\boldsymbol{R}$ 34
3.4.2 HST Conversion 37
3.4.3 Lower Bounding The Optimal Manhattan Cost 40
3.5 Analysis of the Online Queueing Cost 45
3.5.1 Constructing a Spanning Tree 45
3.5.2 Bounding the Manhattan Cost of the Spanning Tree 46
3.5.3 Bounding the Total Latency Cost 48
3.6 Queueing Cost in the Asynchronous Model 49
3.7 Minimum Spanning Tree Approximation 51
3.8 Chapter Notes 52

II Centralized Service for Sequential Requests **55**

4 Online Mobile Facility Location Problem: A Lower Bound **57**
4.1 Introduction 57
4.2 Online MFL Problem 58
4.2.1 Further Related Work 59
4.3 A Lower Bound for OMFL 60
4.3.1 Outline of the Lower Bound Proof 60
4.4 Towards OMFL on General Metrics 61
4.5 Chapter Notes 62

5 Minimizing Combined Cost in Generalized Uniform OMFL: A Tight Analysis **63**
5.1 Generalized OMFL with Uniform Distances 63
5.1.1 Further Related Work 65
5.2 Problem Statement 65
5.2.1 Service Cost Function Properties 67
5.3 Algorithm Description 67
5.4 Analysis Overview 68
5.5 Upper Bound Analysis 69
5.5.1 Optimal Service Cost Analysis 71
5.5.2 Optimal Offline Algorithm Total Cost 79
5.5.3 Online Algorithm Total Cost 79
5.6 Lower Bound Analysis 81
5.6.1 Lower Bound Execution 81
5.6.2 Online Algorithm Total Cost 82
5.6.3 Optimal Offline Algorithm Total Cost 84

5.7 Chapter Notes 84

6 Minimizing Movement in Generalized Uniform OMFL: A Lower Bound 85

6.1 Introduction 85

6.1.1 Problem Statement 85

6.1.2 Further Related Work 87

6.2 A Lower Bound 87

6.2.1 Lower Bound Analysis 87

6.3 Chapter Notes 91

Bibliography 93

My Publications 99

Chapter 1

Online Allocation Problems

1.1 Allocation Problems

The study of *allocation problems* has numerous applications in the fields of computer science, mathematics, economics, and engineering. In the following, three applications of different allocation problems are described to give a taste of what is to come.

A sales manager of a company is identifying locations in a city to ship the products of the company from some new sites to the customers. The sales manager would find the demands for the products in different locations and neighborhoods of the city. Further, the cost for constructing a new site at each potential location is calculated. The sales manager creates a table that includes the potential locations, the cost of constructing a new site at each location, and the distance between each customer's demand and each site. The goal is to choose a subset of these potential locations to construct the new sites at their locations such that the total cost of building new sites together with the cost for shipping the products to the demands' points is minimized. The above scenario is an application of problems called *facility location* that aim to locate a set of facilities to efficiently meet a set of demands [27, 33, 56].

For the second problem, assume there is a shared object such as a linked list in which a set of processes might need to get exclusive access to the linked list to perform some task, for instance to remove some elements of the linked list. A distributed computer system must coordinate the accesses to the linked list such that only one process can access the linked list at a time in order to keep the linked list in a consistent state. This scenario is an application of the *mutual exclusion problem* [31].

As the last scenario, consider a set of machines that are located at different locations and that are interconnected by a *network*. A set of jobs are initiated by users in the network at arbitrary times and locations. The machine at the machine of each job arrival only knows about the newly arrived job. After each job arrival, one needs to decide whether to execute the job locally or whether to send it to some other machine with less load or more power. In case of sending the job to some other machine, however, one needs to pay a transferring cost that might be proportional to the distance between the two machines. The goal is to provide an efficient global scheduling of all jobs at all machines such that the **total completion time** of all jobs is minimized (where the completion time of a job is the time between the time of arrival of the job and the time the job is completed). This scenario is an application of a class of problems known *distributed job scheduling problems* [3, 12].

The common denominator of all above scenarios together with a large volume of applications of allocation problems in areas such as industrial planning, network design, content distribution in networks, or data clustering is an abstract problem as follows. On the one hand, there is a set of different potential resources that are either costly or limited in number. On the other hand, there is a set of requests to access/use these resources. Serving each request might incur some coast as well. The goal is to devise algorithms to efficiently serve the requests by using the available potential resources. These problems can also be seen from the point of view of optimization problems. In this thesis, wherever it is said that a problem is efficiently solved, we mean that the goal is to minimize the total cost. Alternatively, one might also be concerned with maximizing some kind of profit.

1.2 Online Allocation Problems

There are applications of allocation problems in the real world where the requests are fed into the system incrementally and where the requests have to processed as they arrive. In comparison to the *offline* setting, where the whole request sequence is known from the beginning, such an incremental scenario is known as an *online* setting [2, 21]. At each point, such a system must make its decision when to assign a request and to which available resource to assign the request, without knowledge of the future inputs and only based on the request sequence it has seen so far. One either assumes that the assignment of a request to a resource is irrevocable even if the future inputs show that the decision has been suboptimal, or one assumes that the assignment can be changed in the future. However, in the latter case, the system has to pay some cost. Such problems are called *online* allocation problems.

Some fundamental allocation problems such as facility location are traditionally studied in the offline setting (see, e.g., the first scenario that is described in Section 1.1 as an application of a facility location problem in the offline setting). However, considering online versions of such problems is interesting from a theoretical point of view—e.g., since efficient online solutions might provide additional insights into the structure of the problem—, but it is also interesting regarding their applications in real life. The online version of facility location problems has numerous applications in areas such as network design and data clustering [38, 57]. For example, assume a network is established using servers and physical links between the servers and some clients. Now some new clients need to be added to the network. Since changing the physical links previously installed between the servers and the clients is very expensive, the assignment of clients to servers is done in an irrevocable fashion. Hence, one might need to provide some new servers and new physical links to connect the new clients. The goal is to minimize the sum of the total cost of servers that are purchased and the total cost of physical links. In clustering applications, assigning requests to a cluster center might not be irrevocable since merging the cluster centers might not be expensive. Therefore, upon arrival of a new request, one can assign the new request to an existing cluster, open a new cluster center and assign the newly arrived request, or possibly also merge two already open cluster centers and assign the newly arrived request to the cluster center resulted by merging. The formal definition of *online facility location problems* together with definitions of some other classic online allocation problems are presented in Section 1.2.1.

1.2.1 Formal Definitions

In this thesis we study some online allocation problems. We next provide the formal definitions of some of the classic online allocation problems that are related to the problems studied in this thesis.

Online Facility Location Problem (OFL): There are different variants of online facility location problems. Here, we provide the definition of the online facility location problem that was first given by Meyerson [57]. There is a metric space with a set of nodes. Each node can potentially host a facility or demand. To use a facility, the facility must have been opened, where an **opening cost** occurs. A set of demands are issued one by one each at a time and each request is assigned to the closest open facility once it is issued. The cost for serving a demand is the distance between the demand and its closest open facility. This cost called the **service cost**. The decisions of assigning demands to facilities are irrevocable. The challenge is to decide when and where to open a new facility (without knowledge of future demands) such that at the end, the total opening cost of facilities and the total service cost of demands is as small as possible.

We next consider an allocation problem that has an intrinsic online nature. The classic *k-server problem* [16, 54, 55] is perhaps the most and best studied problem among all online allocation problems and generally among all online problems. The k-server problem is a generalization of the *paging problem* [5, 11, 35, 68], which we formally define first.

Paging Problem: Virtual memory hierarchies are implemented in operating systems to provide a large memory address space and to combine fast and small memory units (such as a processor cache) with a slow and large memory unit (such as the main RAM). A main part of a virtual memory hierarchy is a memory management program by which the system transfers data from and to the slow memory to use in the cache (fast memory). The memory is partitioned into blocks (called pages) of equal sizes. The system receives a sequence of requests, each at a time, where each request specifies a page in the memory system. A request can be served immediately if the referenced page is available in cache. If the requested page is not in cache, a **page fault** occurs. The missing page is then loaded from slow memory into cache so that the request can be served. At the same time a page is evicted from the cache to make room for the newly loaded page. The choose the pages to evict so as to minimize the number of page faults. Now we can formally define the k-server problem as a generalization of paging problem.

k-Server Problem: There is an n-point metric space and k mobile servers. The k points of the metric that host the servers correspond to the k pages in the cache. The other $n - k$ points of the metric correspond to the remaining pages in the slow memory. Requests arrive at the points of the metric, one at a time. Each request must be served by a server at the requested point after the request arrives and before new requests arrive. Therefore, if a server is not available at the point where a new request arrives, some server must be moved to the point of the request. This corresponds to evicting some page in the fast memory and moving the requested page to the fast memory. While in the paging problem, the goal is to minimize the total number of page faults, in the k-server problem, the objective is to minimize the total distance by which the servers need to be moved around. The paging problem is equivalent to the k-server problem if the underying metric is a uniform metric, where the distance between any pair of distinct points is 1. Some key questions regarding the k-server problem are still open [55] and the work on the problem has been a major driving force for the developments in the area of *online algorithms* and even of various problems where the goal more generally is to *serve online requests with mobile resources*.

1.3 Centralized Service for Sequential Requests

Consider a variant of facility location problem where the facilities are mobile. There are practical scenarios [28, 39], where the goal in a facility location problem is to plan the motion of facilities and demands such that each demand will be at a point that also hosts some facility. The *mobile facility location* (MFL) problem in general metrics has been introduced in [28] as a movement problem. It can be seen as a generalization of the facility location problem [39]. Initially, there are k mobile facilities located at some points of the metric space. Further, there is a set of demands where each demand needs to be assigned to some facility. In contrast to the standard facility location problem, where essentially each demand moves to some facility, in MFL a facility can move close to some sufficiently large cluster of demands such that the total movements of demands and facilities are minimized. MFL is modeled in such a way that the algorithm needs to move each facility and demand such that in the final configuration, each demand is at a node with some facility. The goal is to minimize the total movement cost of facilities and demands. Equivalently, one can also only move the facilities. The total cost at the end then is the total movement cost of the facilities plus the sum of the distances of each demand to its closest facility.

We introduce an online variant of MFL that we call the *online mobile facility location (OMFL)* problem. The problem is formally defined in Chapter 4 in Part II. In the OMFL problem, initially we are given a set of k mobile facilities with their starting locations. One by one, demands are added. After each demand arrives, we can make some changes to the facility locations to ensure we have a feasible solution before the next client arrives and the new demand is assigned to the closest facility. In other words, facilities can be moved, each demand is always assigned to the nearest facility and the cost of this assignment is the distance from the demand to the facility. Hence, if some facility j moves from v_1 to v_2 in one step and then from v_2 to v_3 in another step, the total movement distance $d(v_1, v_2) + d(v_2, v_3)$ is counted toward the total movement cost of facility j. Two types of costs are considered. The first type is the total distance traveled by the facilities. The second, is the assignment/service cost of demands. Since the assignment between facilities and demands can change, we always take the *current assignment cost* into account. The goal is to at all times minimize the total movement cost and the current total assignment cost.

In Part II, in Chapter 5 and Chapter 6, we study two generalized versions of OMFL with uniform distances.

1.4 Models

1.4.1 Centralized vs. Distributed Systems

In this thesis, we study online problems in the classic *centralized* sequential setting, but also in a *distributed* setting. In centralized systems, there is a central unit that controls everything and is aware of the global state of the system. The global state includes the configuration of the system such as for example the locations of demands and facilities in an online facility location problem. It further includes the complete request sequence up to the current point in the execution. A centralized system relies heavily on the central unit, which controls the complete execution. The problems discussed in Section 1.3 are defined in the centralized setting.

By contrast, in distributed systems such a "mastermind" does not exist. There are different types of operational units in the system that only act based on local information and that cooperate in order to achieve a global goal. Hence, in distributed systems, in contrast to centralized systems, many computational units can be active at the same time. There are two main types of distributed systems with respect to the way they exploit cooperation between the processors. One type of distributed systems controls and uses the power of processors to solve large and complex problems in collaboration while the other type exploits cooperation to achieve individual goals in which different processors do not disturb each other [60].

The latter type of distributed systems that uses cooperation in a preventive sense is the one we consider in this thesis. Generally to solve distributed allocation problems such as the distributed k-server problem [18], the distributed file allocation problem [10], or the distributed facility location problem [58], a distributed system must serve the requests of individual users who often do not share any common interest and who are instead merely interested in their own activities. The cooperation in such distributed systems is done to guarantee that the resource allocation is achieved correctly and as efficiently as possible. In this thesis, in Part I we study the *queuing problem* [48] in a distributed setting.

1.4.2 Network Model

Many applications of online allocation problems are indeed problems that occur in networks (see for example the last scenario in Section 1.1). In such a case, a network topology is usually modeled by a simple connected graph where the nodes of the graph represent the processors of the network and the edges represent the communication links between the processors. We sometimes assume that the network graph is weighted. The weight of each edge can provide different interpretations regarding the communication link between the corresponding adjacent processors in the network. In the context of this thesis, the weight of an edge will always refer to the propagation delay of the communication link between the two corresponding processors in the network or more generally to the cost of using the link. [8, 60].

1.4.3 Communication Model

Although also in a centralized online problem, the algorithm does not have complete information about the input (it does not know the future), the partial information that is available is completely known by the algorithm. In distributed setting, there is also incomplete knowledge at the different processors if a piece of information has already fed into the system and a processor does not know about that information while some other processors might already know the information. Thus, when an online problem is studied in the distributed setting, a processor might suffer from lack of information not only because of the online nature of the problem, but also because the available information is distributed among the processors. Therefore a processor needs to communicate with other processors to obtain additional knowledge about the global state of the system in order to make decisions.

A common way to exchange this information is by using *message passing*. In a message passing system, neighboring processors in the network can exchange messages with each other. Note that even if an online problem is defined in a network context, if we study the online problem in the centralized setting, we typically do not take into account the communication cost of informing the nodes of the network about the steps they have to carry out. In contrast, when considering a distributed online problem, communication is an integral part of the overall cost. The cost of communication can be measured by the delay and/or the communication volume that are incurred in the process of answering the requests.

Note that distributed algorithms generally have an online flavor even if all the requests are revealed from the beginning. In a distributed algorithm, the processors face different sources of uncertainty and nondeterminism even in an offline setting. The uncertainty for example comes from unpredictable message delays (asynchrony), failures, and generally from the fact that a node cannot instantly learn about non-local events in the network and the node can thus never know the entire current global state of the network. This limitation to not have a global picture of the current system state makes the work of an online algorithm in a distributed system harder than in a centralized system.

1.4.4 Synchrony vs. Asynchrony

The communication links can have different delays for sending messages. Synchronous and asynchronous models are two cornerstone models that have been defined to capture the notion of timing in distributed systems. In a synchronous message passing model, message delays are bounded and a distributed protocol has access this timing information and to a global clock. Often, it is supposed that time is divided into synchronous clock pulses that are known as rounds. In each round, each processor executes three steps: a) send message to some neighbors, b) receive messages from some neighbors, and c) perform some local computation.

In an asynchronous model, in contrast, the messages arrive to their destinations after a finited but unbounded amount of time. Messages that take a longer path may arrive earlier and the receiver of a message can never distinguish whether a message is still in transit or whether it has not been sent at all. Processors do not have access to a clock and all their actions are event-driven.

In the real world, distributed systems might be neither completely synchronous nor completely asynchronous. However, from a theoretical point of view it is most relevant to study problems with respect to the two extreme models. An impossibility result for a problem in the synchronous setting holds for the problem in any practical model defined between synchronous and asynchronous models. Analogously, a distributed algorithm that solves a problem in an asynchronous system can solve the problem in any other intermediate practical model with at least the same quality. In Part I, the *distributed queuing problem* is studied in both synchronous and asynchronous systems.

1.4.5 Input Sequence

In everyday life we might face situations where we need to make decisions that should minimize our future cost or maximize our future profit. One could for example be faced the abstract problem of paying a large amount of money for using a facility for a long time or instead paying a smaller amount of money for using the facility for a much (over-proportionally) shorter time. The difficulty of making such a decision comes from the fact that one might not know in advance how often one is going to use the facility.

The available information about the input is playing a crucial role in devising efficient solutions for computational problems. In this thesis, we consider different ways in how requests arrive over time. In classic online algorithm, one assumes that the requests are arriving **sequentially** one by one and each request has to be processed before the next request arrives. Especially, when we consider online problems in a distributed system, it makes sense to study a general **dynamic** setting, where requests can arrive possibly concurrently at arbitrary times and arbitrary nodes of the system. Recently, the dynamic setting has been also considered in the centralized setting [15, 19]. The classic sequential arrivals of requests can be seen as an extreme case of arbitrary dynamic arrivals. In this thesis, we study a distributed online allocation problem with a dynamic request arrival pattern in Part I and we study two versions of a centralized online allocation problem with sequential request arrivals in Part II.

1.4.6 Hierarchically Well-Separated Trees

Embeddings of a metric space into (a probability distributions over) tree metrics has found many important applications [15, 16, 26]. The notion of a *hierarchically well-separated tree* (in the following referred to as an HST) was defined by Bartal in [17].

Definition 1.1. *Given a parameter $\alpha > 1$, an α-HST of depth h is a rooted tree with the following properties. All children of the root are at distance α^{h-1} from the root. Further, every subtree of the root is an α-HST of depth $h-1$ that is characterized by the same parameter α (i.e., the children 2 hops away from the root are at distance α^{h-2} from their parents). A tree is an HST if it is an α-HST for some $\alpha > 1$.*

The probabilistic tree embedding result of [34] shows that for every metric space (X, d) with minimum distance normalized to 1 and for every constant $\alpha > 1$, there is a randomized construction of an α-HST T with a bijection f of the points in X to the leaves of T such that a) the distances on T are dominating the distances in the metric space (X, d), i.e., $\forall x, y \in X$: $d_T(f(x), f(y)) \geq d(x, y)$ and such that b) the expected tree distance is $\mathbb{E}\big[d_T(f(x), f(y))\big] = O(\alpha \log |X| / \log \alpha) \cdot d(x, y)$ for every $x, y \in X$.

Utilizing HSTs as a tool in solving the online allocation problems is a common approach [15, 16, 26]. The procedure of finding an approximate optimal solution for a given online minimization problem is roughly as follows: an HST is sampled according to the distribution defined by the embedding. The problem is then solved on the HST with a competitive ratio of γ. Since the distances in the HST are at least the corresponding distances in the original graph/metric, the solution on the HST provides a solution on the original graph of at most the same cost. However, to bound the cost of the optimal solution with respect to the distances in the HST from above by the corresponding distances in the original graph, we lose a $O(\log n)$ factor. Consequently, we get an expected $O(\gamma \cdot \log n)$ competitive ratio. The technical results of Part I on the distributed queueing problem are obtained by following this general approach.

1.5 Distributed Service for Dynamic Requests

Consider a family of allocation problems where the request sequence dynamically and possibly concurrently arrives in an online fashion at the points of a metric space. There is a server initially located at some point of the metric space and the goal is to provide an order for serving the requests by that server. The server needs to move to the locations of the requests to serves them. The goal in all cases is to provide a schedule for serving all requests. However, there are different ways in which to measure the cost of a solution, leading to different solutions. As in all such problems postponing serving the requests is sometimes unavoidable and thus is allowed. In Part I, we study a problem that fall into the described family of problems.

A problem of the above family, called the *online service with delay (OSD) problem* [15], has recently been introduced in the centralized setting. In the OSD problem the **delay of a request** is the time that it takes from when the request arrives in the system until it is served. It is assumed that the server can instantaneously move between different points of the metric, however the total distance that is traveled by the server is accounted for. The quality of a solution in the centralized setting for the OSD problem is evaluated w.r.t. the total movement plus the total delay cost. Another problem of the family that has been studied also in the centralized setting is the *online TSP problem* [9, 19]. In the online TSP problem, the server moves at unit speed to the requested points. The objective in the online TSP problem is to minimize the total time until all requests have been served.

The third problem of this family is the *distributed queueing problem* [48]. This problem is one of the problems studied in this thesis in Part I. At the core of many distributed directory implementations is the following basic distributed queueing problem that allows to order potential concurrent access requests to a shared object [48]. The nodes of a network issue queueing requests (e.g., requests to access a shared object) in a completely dynamic and possibly arbitrarily concurrent manner. A queueing algorithm needs to globally order all the requests so that they can be acted on consecutively. Formally, each request has to find its *predecessor request* in the order. That is, when enqueueing a request r issued by some node v, a queueing algorithm needs to find the request r' that currently forms the tail of the queue and inform the node v' of request r' about the new request r. Hence, the **delay of a request** is the difference between the times the request is issued and ordered. The quality of a solution for the distributed queuing problem evaluated w.r.t. the sum of delays for ordering the requests. In Chapter 3, we study the distributed queuing problem when the requests dynamically arrive.

1.6 Request-Answer Games

In interactive computations where the inputs are fed into the system in an online fashion, an algorithm must make a decision in response to each newly arrived piece of the input. Hence, the algorithm produces a sequence of decisions without knowing the future inputs. An algorithm that solves such an online problem is called an **online algorithm**. Before we formally define what an online algorithm is, we provide the definition of a more general framework called *request-answer* systems for studying online algorithms, in particular online allocation problem.

A request-answer system has three main entities. A request set R, a sequence of finite non-empty answer sets $A_1, A_2, \ldots, A_n$, and a sequence of cost functions $cost_1, cost_2, \ldots, cost_n$ where $n \in \mathbb{N}^+$. The cost function $cost_i$ where $i \in \{1, 2, \ldots n\}$ is defined as follows [21].

$$cost_i : R^i \times A_1 \times A_2 \times \ldots \times A_i \to \mathbb{R}^+ \cup \{\infty\} .$$

For example, the k-server problem where the number of points in the metric is N is a request-answer system where the request set is $\{1, 2, \ldots, N\}$ and the answer set is $\{0, 1, 2, \ldots, k\}$. The answer i to a request j implies that the server i moves to the requested point j. The answer 0 in response to the request j implies that there is server at the requested point j and no server is moved. The total cost of serving the first j requests is $cost_j = d(v_i, v_j) + cost_{j-1}$ where $d(u, v)$ generally denotes the distance of points u and v. The distance d is nonnegative, symmetric, and satisfies the triangle inequality. Further, $cost_0 := 0$ and also v_j and v_i are the points that host the request j and the server i, respectively.

Online Algorithms: A *deterministic* online algorithm always produces the same sequence of answers for a particular sequence of requests. Formally, a deterministic online algorithm ALG is a sequence of functions $g_i : R^i \to A_i$ for $i \in \mathbb{N}^+$. For a given request sequence $\sigma = \langle r_1, r_2, \ldots, r_n \rangle$ let ALG$[\sigma] = \langle a_1, a_2, \ldots, a_n \rangle$ denote the answer sequence of ALG in response to σ where $a_j = g_j(r_1, r_2, \ldots, r_j)$. The cost of ALG for serving the whole request sequence σ denoted by $\mathrm{ALG}(\sigma) = cost_n(\sigma, \mathrm{ALG}[\sigma])$. A *randomized* online algorithm is a probability distribution over the set of all deterministic online algorithms $\{\mathrm{ALG}_x\}$. Note that for a given request sequence σ both the answer sequence $\mathrm{ALG}[\sigma]$ and the cost $\mathrm{ALG}(\sigma)$ are random variables [21].

Since the answer sequence produced by an online algorithm as well as the cost incurred by the online algorithm is a function of the request sequence, such an interaction system can be seen as a game between two players: the online algorithm and an *adversary* that produces the request sequence.

1.6.1 Adversary Models

There are different types of adversaries. An adversary aims to maximize the cost of an online algorithm by producing the worst request sequence based on what it knows about the online algorithm. At the same time, the adversary aims to minimize the cost incurred by an optimal offline algorithm. We can view the adversary as the *offline player* who wants to force the *online player* to make costly decisions. When the online algorithm is deterministic, then the adversary knows everything about the online player and therefore it exactly knows what will be the answer of the online algorithm in response to each request. This adversary is the most powerful adversary.

When randomization is used, the online algorithm has more power, or in other words, the adversary is comparatively less powerful since the answers of the online player are not certain anymore. Generally, in the case of randomized online algorithms, there are two types of adversaries, *oblivious* and *adaptive* adversaries. Both of these adversaries know the online algorithm description including the probability distribution used by the online algorithm. However, the oblivious adversary must produce the request sequence in advance without knowledge of the online algorithm's answers. By contrast, the adaptive adversary at each time knows all the answers that have been made by the online algorithm up to that point in time. Hence, it can produce the next request dependent on the history of what the online algorithm has done. There are different types of adaptive adversaries, however with respect to the problems studied in this thesis we only consider the oblivious adversary and the adaptive adversary that knows everything about the online algorithm including the random answers of the online algorithm. In this case, it is known that in the sequential setting, randomization does not help against an adaptive adversary [21].

We next describe how to measure the quality of online algorithms. The standard method is to compare the costs of the online and offline players.

1.6.2 Competitive Analysis

Competitive analysis is a mathematical framework that was proposed in [68] for analyzing online algorithms. In this framework of analysis, the cost of an online algorithm for solving a problem is compared to the cost of an optimal offline algorithm that solves the same instance of the problem. The optimal offline cost incurred by an optimal offline algorithm OPT for a given request sequence $\sigma \in R^n$ is defined as follows,

$$\mathrm{OPT}(\sigma) = \min \left\{ cost_n(\sigma, a) : a \in A_1 \times A_2 \times \ldots \times A_n \right\}.$$

The deterministic online algorithm ALG provides a competitive ratio $\alpha \geq 1$ if for any given request sequence σ we have

$$\mathrm{ALG}(\sigma) \leq \alpha \cdot \mathrm{OPT}(\sigma) + \beta,$$

where β is an additive constant. Now, we define the competitiveness of a randomized online algorithm, say again ALG, against an oblivious adversary. The randomized online algorithm ALG is said to be α-competitive if for every (fixed) request sequence σ,

$$\mathrm{ALG}(\sigma) = \mathbb{E}[\mathrm{ALG}_x(\sigma)] \leq \alpha \cdot \mathrm{OPT}(\sigma) + \beta,$$

where again β is supposed to be an additive constant. Competitive analysis is also used for analyzing distributed online algorithms [3, 18].

1.7 An Abstract Online Allocation Problem

To sum up this chapter, we provide an abstract online allocation problem that captures many of the standard online problem as a special case. We are given a metric space with n points. There is a set of servers and there is a set of requests that is fed into the system in an online fashion and each request needs to be served by some server. The goal is to minimize some cost function.

This abstract online problem lies at the heart of many centralized and distributed online applications. A concrete online problem can be specified based on how the servers can act, on how the servers serve the requests, on how the way in which the online requests arrive, and on how the cost function is defined. The problems that are studied in this thesis are applications of this abstract online problem. In the following, we provide a discussion of the possible variations and on how some specific online allocation problems fit into the general setting.

Arrival of Requests: The requests are fed into the system in an online fashion. As described in Section 1.4.5, the requests can either arrive one by one sequentially or they can arrive in a general dynamic and possibly concurrent way (i.e., each request can arrive at an arbitrary location and at an arbitrary point in time). The latter case generalizes the sequential scenario.

Servers: Either a set of $1 \leq k < n$ servers are *available* and initially located at some points of the metric space or each point of the given metric can be *potentially* host a server. In the latter case, the servers need to be established, which typically incurs an **opening cost**. The servers either can be *mobile* and move around the metric space or they can have a *fixed* location.

Service Scenario: After a request arrives at time t, the request might need to be either **permanently** served (i.e., the request needs to be served at all times $t' \geq t$) or it only needs to be **temporarily** served (i.e., the request only needs to be served once at time t). If a request needs to be permanently served, the decision of assigning the request to some server is either *irrevocable* or it is *changeable*. Further, requests must be served at the *requested point* or they can be *remotely* served. In the latter case, there is a **service cost** for the request that is typically proportional to the distance between the server and request. Further, the requests must either be served *immediately* after they are issued, or an online algorithm can wait and serve some request *later*. In the latter case, however, waiting incurs some **latency cost**.

Cost Function: The goal can be to minimize a *singular* cost function such as the total movement cost of all servers. It can be also to minimize a *combined* cost function that usually consists of two singular cost functions. When the goal is to minimize a combined cost, there is a trade-off between the two singular cost functions. Therefore, to be competitive against an optimal offline algorithm, it is necessary to strike a balance between the singular cost functions in the combined cost.

Table 1.1 includes some applications (including some of the problems studied in this thesis) of the abstract problem and shows which combination of above characterization defines each problem.

Table 1.1: Translation of the abstract allocation problem in different applications.

	k-Server	DQU [1]	OSD	OMFL	OFL
Server(s)	available mobile	available mobile	available mobile	available mobile	potential fixed
Service Scenario	requested point immediately	requested point late	requested point late	remotely immediately permanently changeable	remotely immediately permanently irrevocable
Online Fashion	sequential	dynamic	dynamic	sequential	sequential
Cost Function	singular total movement	singular total latency	combined total movement total latency [2]	combined total movement current service cost	combined service cost opening cost

[1] DQU stands for distributed queuing problem.
[2] As mentioned in Section 1.5, the latency in the OSD problem is defined differently from the latency in the distributed queuing problem.

Remark 1.2. *Some combinations of cases are impossible. For example, the case when $k < n$ servers are available with another scenario where each request must be served at the requested point both together imply that the requests cannot be permanently served and the servers must be mobile.*

1.8 Our Results

In the following, we overview the technical contributions of this thesis. The first part of the thesis mainly deals with the distributed queueing problem. In the second part of the thesis, we study generalized versions of the online mobile facility location (OMFL) problem with uniform distances.

Distributed Service for Dynamic Requests: The ARROW protocol [59, 63, 70] is a simple and elegant distributed algorithm to coordinate exclusive access to a shared object in a network. The algorithm solves the underlying distributed queuing problem by using path reversal on a pre-computed spanning tree (or any other tree topology simulated on top of the given network). It has been previously shown that the ARROW protocol is also an efficient way of solving the distributed queuing problem on a tree topology. In [48], it was shown that on any tree T and for arbitrary dynamic arrivals of requests, the ARROW protocol is $O(\log D)$-competitive in synchronous executions, where D is the diameter of the tree T. We note that here, the competitive ratio is measured by comparing the ARROW protocol to the best offline algorithm on the same tree T. If the tree T stretches the distances on an underlying graph G by a factor s, the competitive ratio w.r.t. the best solution on G might grow to as much as $O(s \cdot \log D)$.

In this thesis, in Chapter 3, we significantly generalize the result of [48]. As the main technical contribution of Chapter 3, we show that when running the ARROW algorithm on top of an HST T, it has a constant competitive ratio (when compared to an optimal offline solution on T). While the analysis in [48] is based on a reduction of the problem to an analysis of the nearest neighbor heuristic for the TSP problem, the analysis on HSTs given in this thesis is based on a completely novel approach. In combination with the probabilistic embedding of arbitrary metrics into HSTs of [34], we show that when run on an HST that is obtained by using the randomized construction of [34], our result implies that ARROW is $O(\log n)$-competitive on general network topologies, even if the requests in a arbitrarily dynamic and possibly concurrent fashion. We further show that the result even holds if communication is asynchronous.

Centralized Service for Sequential Requests: In the second part of the thesis, we study variants of the online mobile facility location problem. In Chapter 4, we provide a lower bound on the achievable competitive ratio. The lower bound even holds for the OMFL problem on uniform metric spaces, that is, when all pairwise distances are equal to 1. In that case, it shows that there is not *strictly* competitive with a small competitive ratio. More specifically, we (roughly) prove that if an online algorithm guarantees that for some $\beta = \Omega(k/\log k)$, the service cost of the algorithm is less than an additive β above the optimal service cost, then it holds that $\mathrm{cost}^{\mathrm{ALG}} \geq \big(1 + \Omega(k/\beta)\big) \cdot \mathrm{cost}^{\mathrm{OPT}} + O(k \log k)$, where $\mathrm{cost}^{\mathrm{ALG}}$ and $\mathrm{cost}^{\mathrm{OPT}}$ refer to the total costs of the online algorithm and an optimal offline algorithm, respectively.

Chapter 5 is devoted to the main technical result of Part II. We study the following generalization of the OMFL problem on uniform metric spaces. There are n nodes, each node can host 0, 1, or also more than 1 servers. The requests arrive at the nodes, the service cost at each node is defined by a general cost function. The service cost is a function of the number servers and the number of requests at the node. We note that the standard OMFL problem on uniform metric spaces is a special case, where the cost is either 0 (if there is a server at the node) or it is equal to the number of requests at the node. For this generalized uniform OMFL problem, we provide a deterministic online algorithm that almost matches the lower bound of Chapter 4. Concretely, we show that at the cost of an additive term which is roughly linear in k (k is the number of facilities/servers), it is possible to achieve a competitive ratio of $(1 + \varepsilon)$ for every constant $\varepsilon > 0$.

Finally, we introduce a movement version of the OMFL problem. The problem is specified by two parameters $\alpha \geq 1$ and $\beta \geq 0$ such that $\max\{\alpha - 1, \beta\} \geq 1$. A solution at time t is called feasible if the service cost is within a multiplicative factor α and an additive term β of the optimal service cost at time t. An algorithm has to move servers in order to guarantee that the solution remains feasible at all times. We show that for all α and β and all $k \in \{1, \ldots, n-1\}$, any deterministic online algorithm for the problem necessarily has a competitive ratio of at least $\Omega(n)$.

Organization of the Thesis

The thesis consists of two parts. In Part I, we mainly study the distributed queuing problem. Chapter 2 provides essential preliminaries that are needed in the technical analysis of the problem in Chapter 3, where as the main technical contribution, we analyze the ARROW protocol in hierarchically well-separated trees (HSTs), when the requests can arrive in a completely dynamic fashion.

Part II starts with introducing an online instance of mobile facility location (OMFL) problem in Chapter 4. In Chapter 5, we analyze the problem for the special case of having a metric space with uniform distances (i.e., all pair-wise distances are the same). While we can only provide an analysis for this special case, we study a generalized version of the problem in which each point of the metric can host more than one servers and we accordingly define a general cost function that captures the cost of serving a certain number of requests with a given number of servers at each point of the metric space. In Chapter 6, a movement version of the problem introduced in Chapter 5 is studied.

Part I

Distributed Service for Dynamic Requests

Chapter 2

The Distributed Queuing Problem: Preliminaries

2.1 Model and Problem Statement

Coordinating the access to shared data is a fundamental task that is at the heart of almost any distributed system. For example, when implementing a distributed shared memory system on top of a message passing system, each shared register has to be kept in a coherent state despite possibly a large number of concurrent requests to read or write the shared register. In a distributed transactional memory system, each transaction might need to operate on several shared objects, which need to be kept in a consistent state [47, 67, 71]. When implementing a shared object on top of large-scale network, a *distributed directory algorithm* can be used to improve the scalability of the system [1, 7, 13, 23, 29, 47, 67]. When a network node requires access to a shared object, the directory moves a copy of the object to the node requesting the object. If the node changes the state of the shared object, the directory algorithm has to make sure that all existing copies of the object are kept in a consistent state.

Communication Model: In this part, i.e., Part I, we consider a standard message passing model on a network modeled by a graph $G = (V, E)$. In some cases, the edges of G have weights $w : E \to \mathbb{R}_{>0}$, which are assumed to be normalized such that $w(e) \geq 1$ for all $e \in E$. We distinguish between synchronous and asynchronous executions. In a *synchronous execution*, the delay for sending a message from a node u to a node v over an edge e connecting u and v is exactly 1 if the edge is unweighted and exactly $w(e)$ otherwise. In an *asynchronous execution*, message delays are arbitrary, however when analyzing an asynchronous execution, we assume that the message delay over an edge e is upper bounded by the edge weight $w(e)$ (or by 1 in the unweighted case).

The Distributed Queueing Problem: In the *distributed queueing problem* on a graph $G = (V, E)$, there is a shared object and a set R of queueing requests $r_i = (v_i, t_i)$ are issued at the nodes of V in an arbitrarily dynamic fashion. The goal of a queueing algorithm is to order all the requests to access the shared object. Specifically, if a request $r_i = (v_i, t_i)$ is issued at node v_i at time $t_i \geq 0$, the algorithm needs to enqueue the request r_i by informing the node v_j of the predecessor request $r_j = (v_j, t_j)$ in the constructed global order. For this purpose, every queueing algorithm in particular has to send (possibly indirectly) a respective message from node v_i to v_j. We say that the request r_i can be enqueued as soon as the predecessor request r_j is in the system and as soon as node v_j knows about request r_i. The delay of request r_i is the time it takes until it is enqueued. The quality of a solution for the distributed queuing problem evaluated with respect to the sum of delays for ordering the requests. We assume that at time 0, when an execution starts, the tail of the queue is at a given node $v_0 \in V$. Formally, this is modeled as a request $r_0 = (v_0, 0)$ which has to be ordered first by any queueing algorithm. We sometimes refer to r_0 as the dummy request.

Note that for two integers a and b, $a \leq b$, we use $[a, b] := \{a, \ldots, b\}$ to denote the set of all integers between a and b. Further, for an integer $a \geq 1$, we use $[a]$ as a short form to denote $[a] := [1, a]$.

Organization of the Chapter: We continue this chapter as follows. Section 2.2 describes the ARROW protocol, which is a distributed queuing algorithm that has been introduced in [59, 63, 70]. We mainly analyze this algorithm in this first part of the thesis. Then, in Section 2.3 we formally show that ARROW a has greedy nature even in asynchronous systems. In the subsequent Section 2.4, we provide a formal cost model for analyzing distributed queuing algorithms. At the end of this chapter, in Section 2.5, we discuss some related work.

2.2 Arrow Algorithm

The ARROW algorithm [59, 63, 70] is a distributed queueing algorithm that operates on a tree network $T = (V, E)$. At each point in time, each node $v \in V$ has exactly one outgoing link (arrow) pointing either to one of the neighbors of v or to the node v itself. In a quiescent state, the arrow of the node of the request at the tail of the queue points to itself and all other arrows point towards the neighbor on the path towards the tail of the queue (i.e., the tree is directed towards the current tail).

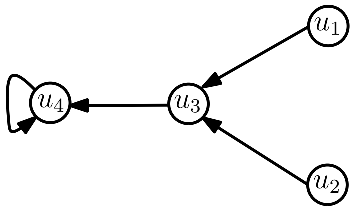


Figure 2.1: ARROW algorithm: initial system state. There is a unique path from each node to the current tail of the queue.

When a new request at a node $v \in V$ arrives, a "find predecessor" message is sent along the arrows until it finds the predecessor request. While following the path, to the direction of the arrows are reversed. More formally, a request r at node v is handled as follows.

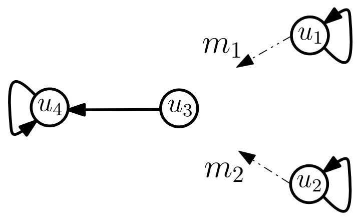


Figure 2.2: ARROW algorithm: the nodes u_1 and u_2 issue requests to access the shared object in a distributed (possibly asynchronous) system.

1. **New request r at v:** If the arrow of v points to v itself, r is queued directly behind the previous request issued at v. Otherwise if the arrow points to neighbor u, *atomically*, a "find predecessor" message (including the information about request r) is sent to u and the arrow of v is redirected to v itself.

2. **Upon u receiving a "find by v" message from node w:** If a node u receives a "find predecessor" message for request r from a neighbor w, if the arrow of u points to itself, *atomically*, the request r is queued directly behind the last request issued by node u and the arrow of u is redirected to node w. Otherwise, if the arrow of u points to neighbor x, *atomically*, the "find predecessor" message is forwarded to node x and the arrow of node u is redirected to node w.

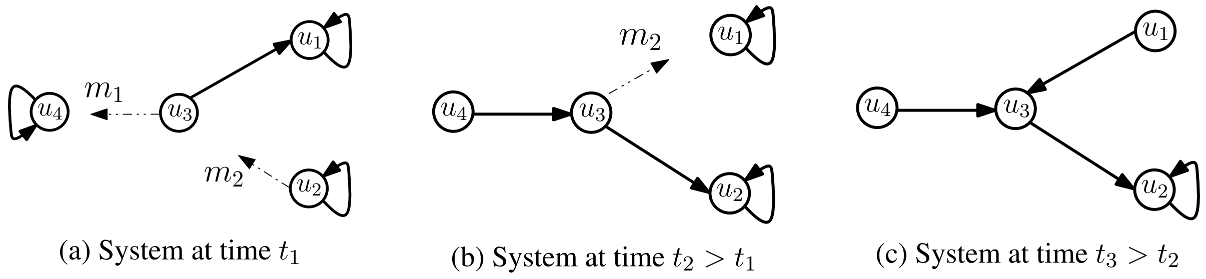


Figure 2.3: ARROW algorithm: intermediate steps. The "find predecessor" messages follow the arrows, reversing their directions along their ways. (a) At time t_1, the message m_1 (cf. Figure 2.2) reaches u_3 while m_2 is on its way towards u_3 and hence m_1 forwarded to u_4. (b) At time $t_2 > t_1$, m_1 reaches u_4 and thus the issued request by u_1 is ordered behind the last request ordered on u_4. Therefore, the shared object is moved to u_1. Also, the "find predecessor" message m_2 reaches u_3 and is deflected towards u_1. (c) The "find predecessor" message m_2 reaches u_1 at time $t_3 > t_2$ and therefore the request issued by u_2 is ordered behind the request issued by u_1. The shared object is moved from u_1 to u_2 and the current tail of the queue is now the request issued by u_2.

See Figure 2.1, Figure 2.2, and Figure 2.3 all together as a simple example of how ARROW behaves. For a more detailed description of the ARROW algorithm and of how ARROW handles concurrent requests, we refer the reader to [30, 46]. It was shown in [30] that the ARROW algorithm correctly orders a given sequence of requests even in an asynchronous network. Moreover as shown in [30, 46], when operating on tree T, the algorithm always finds the predecessor of a request on the direct path on T. As a result, if two requests r' and r are at distance d on T and if r' is the predecessor of r in the queueing order, the "find predecessor" message initiated by request r finds the node of request r' in time exactly d in the synchronous setting and in time at most d in the asynchronous model. Further, it is shown in [46] that the successor request of a request r at node v in the queue is always the remaining request r'' that first reaches v on a direct path. This "greedy" nature of the ARROW ordering was used in [48], where it was shown that in the one-shot case when all requests arrive at time 0, the ARROW order corresponds to a greedy (nearest neighbor) TSP path through the requests, whereas an optimal offline algorithm corresponds to an optimal TSP path on the request set. The competitive ratio on trees then follows from the fact that the *nearest neighbor heuristic* provides a logarithmic approximation of the metric TSP problem [64]. In [46], this analysis was extended and it was shown that even in the fully dynamic case, it is possible to reduce the problem to a (generalized) TSP nearest neighbor analysis.

2.3 Greedy Nature of Arrow Algorithm

In this section, we show that the ARROW algorithm can be interpreted as a distributed greedy algorithm. The greedy property of the ARROW algorithm in the synchronous setting is formally captured by Lemma 2.1 in Section 2.3.1, whereas the corresponding property in the asynchronous setting is formally captured by Lemma 2.3 in Section 2.3.2. The technical results in Chapter 3 rely on the greedy nature of ARROW. Throughout Section 2.3, we assume that we are given a fixed tree T and a set of dynamic requests R are placed at the leaves of T. Further, we assume that an execution of ARROW (either synchronous or asynchronous) with request set R on T is given. For convenience, we relabel the requests in R so that they are ordered according to the queueing order resulting from the given ARROW execution on T. That is, we assume that for all $i \in \{0, \ldots, |R| - 1\}$, request $r_i = (v_i, t_i)$ is the i^{th} request in ARROW's order. Note that $r_0 = (v_0, 0)$ is the dummy request defining the initial tail of the queue. As discussed in Section 2.2, the ARROW order can be seen as a greedy ordering in the following sense. Assume we are given the first $i - 1$ requests in the order. Assume that for each of the remaining requests $r = (v, t)$, at time t, we send a message from node v to the node v_{i-1} of request r_{i-1}. The i^{th} request r_i is a request $r = (v, t)$ from the subset of the remaining requests for which this message can reach node v_{i-1} first.

2.3.1 Synchronous Executions

The greedy behavior is formally captured by the following basic lemma for synchronous executions. For a more thorough discussion, we also refer to [46].

Lemma 2.1. *Consider a synchronous execution of* ARROW *on tree T and consider two arbitrary requests r_i and r_j for which $1 \leq i < j$ (i.e., r_j is ordered after r_i by* ARROW*). Then it* *holds that*

1. $t_i + d_T(v_{i-1}, v_i) \leq t_j + d_T(v_{i-1}, v_j)$ *and*
2. $t_i \leq t_j + d_T(v_i, v_j)$.

Proof. The first claim of the lemma follows immediately from Definition 3.5 and from Lemma 3.8 and Lemma 3.9 in [46]. The second claim follows from the first claim of the lemma and the triangle inequality. □

2.3.2 Asynchronous Executions

We next generalize the greedy property that was captured by Lemma 2.1 for synchronous ARROW executions to the asynchronous setting.

Basic Properties of Asynchronous Arrow Executions: We have seen that a synchronous ARROW execution can be seen as a greedy queueing order. In the asynchronous setting, an analogous property is true. However, we need to be a bit more careful and we need to argue about the arrival time of the "find predecessor" message on the whole path from the node of a request to its predecessor.

Let us assume that we are given an asynchronous execution of ARROW that enqueues the requests $r_0, r_1, \ldots, r_{|R|-1}$ in this order. It has been shown in [30] that even in a concurrent asynchronous ARROW execution, every request r_i finds the node v_{i-1} of its predecessor r_{i-1} on a direct path. To formally specify the greedy property of ARROW in the asynchronous setting, we need to study the progress of messages on the whole path from a request to its predecessor. For any two nodes u, v of T, we use $P_{u,v}$ to denote the direct path from u to v on tree T. The following Lemma 2.3 formally establishes the greedy behavior of asynchronous ARROW executions.

We first introduce some terminology defined in [46]. For all $i \in [0, |R| - 1]$, we define F_i to be a configuration of the tree network, where all arrows are pointing towards the node v_i of request r_i. Further, let R_i be the set of requests $[r_{i+1}, |R| - 1]$ that are ordered after request r_i. Finally, let E_i be an execution of the ARROW algorithm starting from configuration F_i and in which only the requests in R_i are issued. It is shown in Lemma 3.7 in [46] that for all i, except for request r_i no request in R_{i-1} can distinguish locally between executions E_{i-1} and E_i. More specifically, all these requests see exactly the same arrows in both executions. This implies that the "find predecessor" message of every request r_i sees exactly the same arrows as if the network started in configuration F_{i-1} and only request r_i was issued. To study the behavior of the requests in R_{i-1}, it therefore suffices to study an execution that starts in configuration F_{i-1} and where only the requests in R_{i-1} are issued.

Lemma 2.2. *Consider an asynchronous* ARROW *execution for a request set R on a tree T. Let $i \in [1, |R| - 1]$ and consider the path $P_{v_i,v_{i-1}} = (u_0, u_1, \ldots, u_s)$ from node $u_0 = v_i$ of request r_i to the node $u_s = v_{i-1}$ of the predecessor r_{i-1}. For every node u_k on the path, the "find predecessor" message of request r_i is the first "find predecessor" message that reaches* *node u_k (or is generated at node u_k) among all the "find predecessor" of requests r_j for $j \in [i, |R| - 1]$.*

Proof. In order to prove the claim of the lemma, we can assume that requests $r_0, \dots, r_{i-1}$ have already found their predecessors and therefore the tree is in configuration F_{i-1}. Lemma 3.7 in [46] implies that this does not affect the behavior of any of the remaining queueing requests in R_{i-1}.

Assume for contradiction that the claim of the lemma is not true. Let $x \in [0, \dots, s]$ be the maximal value such that the "find predecessor" message of request r_i is not the first one among the requests in R_{i-1} reaching u_k. Note that we need to have $k < s$ because by the definition of the ARROW algorithm, the first message reaching $u_s = v_{i-1}$ is the successor request of r_{i-1}. Let $r = (v, t)$ be the first request in R_{i-1} that reaches node u_s. In configuration F_{i-1}, the arrow of node u_k points to u_{k+1}. In order to change this, a "find predecessor" message first has to be sent from node u_k to u_{k+1}. Because r is the first request reaching u_k, when the "find predecessor" message of r reaches u_k, this has not happened and therefore the arrow still points from u_k to u_{k+1}. When reaching u_k, in an atomic step, the "find predecessor" message of r is therefore forwarded to u_{k+1}. As long as the message is in transit between the two nodes, there is no arrow across the edge $\{u_k, u_{k+1}\}$ and therefore the "find predecessor" message of r also reaches u_{k+1} before the "find predecessor" message of r_i reaches u_{k+1}. This is a contradiction to the assumption on the maximality of k and therefore the claim of the lemma holds. $\square$

The above lemma shows that if the "find predecessor" messages of two requests reach the same node v, then the earlier ordered request reaches v first. To have an analogous statement for Lemma 2.1, we would like to have a statement saying that a request r reaches a node v on the path to the predecessor request before any request r' that is ordered after r (not only for a request r' that actually reaches v). To achieve this, we extend a given ARROW execution to simplify the analyses provided in Chapter 3. Whenever a request $r = (v, t)$ is issued at node v at time t, a "find predecessor" message leaves v at time t and it travels on the direct path to the predecessor request r' of r. For the proof, we assume that instead of only going to the predecessor, the "find predecessor" message is sent as a broadcast to the whole network. We think of the additional messages to complete this broadcast as virtual messages that are only used for the analysis and have no influence on the queueing algorithm. Given an asynchronous execution of ARROW, we assume that the actual messages sent by the ARROW algorithm keep their message delays (to ensure an equivalent execution). All the virtual messages are assumed to have the maximum possible message delay. That is, the delay of sending a virtual message from u to v is equal to the length $d_T(u, v)$ of the respective tree edge. Further, to make sure that virtual messages can never overtake real messages, if a real message and a virtual message reach a node at the same time, the node always first processes the real message. In this way, for every request $r = (v, t)$, the delay of the respective "find predecessor" message is defined for all nodes. For a request r and a node $u \in V$, we introduce the following notation:

$$\Delta(r, u) := \text{ time of "find predecessor" message of request } r \text{ to reach node } u. \tag{2.1}$$

We note that for $r = (v, t)$ and any node $u \in V$, we have $\Delta(r, u) \leq d_T(u, v)$ (recall that in the asynchronous setting, for the analysis, the delay of a message is assumed to be at most the length of the respective edge). The next lemma will be used as a replacement of the main statement of Lemma 2.1 in the asynchronous analysis.

Lemma 2.3. *Consider an asynchronous execution of* ARROW *for a set of requests R on tree T and consider two arbitrary requests r_i and r_j for which $1 \leq i < j$ (i.e., r_j is ordered after r_i by* ARROW*). Then for any node v on the path from v_i to v_{i-1}, it holds that*

$$t_i + \Delta(r_i, v) \leq t_j + \Delta(r_j, v).$$

Proof. Similarly to the proof of Lemma 2.2, we apply Lemma 3.7 from [46] and we assume that the network starts in configuration F_{i-1}. Consequently, initially, all arrows are pointing towards v_{i-1} and only the requests in R_{i-1} still need to be ordered.

We first show that for every arrow pointing from a node u_1 to a node u_2 in configuration F_{i-1}, the first message sent from u_1 to u_2 has to be a real message. For contradiction, assume otherwise and assume that the first arrow along which a virtual message is sent before a real message is pointing from node w_1 to node w_2. Further, assume that message $\mathcal{M}$ is the first such message that is sent by w_1 over the edge. Note that this also implies that $\mathcal{M}$ is the first message sent from w_1 to w_2. Assume that this virtual message $\mathcal{M}$ belongs to a request $r = (v, t)$. First note that $\mathcal{M}$ is the first message arriving at node w_1. Otherwise, some other message would have been sent from w_1 to w_2. If message $\mathcal{M}$ arrives at w_1 as a real message, it is forwarded as a real message to node w_2. We can therefore conclude that message $\mathcal{M}$ reaches w_1 as a virtual message (say from neighbor w_0). Because $\mathcal{M}$ is the first message that reaches w_1, it is also the first message sent from w_0 to w_1 (note that as a virtual message, it has the maximum possible message delay, so it cannot overtake any other message). Because in configuration F_{i-1}, there also is an arrow from w_0 to w_1, this is a contradiction to the assumption that the arrow from w_1 to w_2 is the first on which a virtual message is sent before a real one.

To conclude the proof, observe that in configuration F_{i-1}, all neighbors u of the path $P_{v_i,v_{i-1}} = (u_0, \ldots, u_s)$ from $u_0 = v_i$ to $u_s = v_{i-1}$ have an arrow pointing from u to the neighbor on the path. Hence, on each edge connecting to the path, the first message that reaches the path is a real message. The same is true for all edges of the path in the direction from node $u_0 = v_i$ to node $u_s = v_{i-1}$. The only way a virtual message can therefore reach a node u_k of the path before a real message does is when a virtual message for a request r is sent from a node u_{k+1} to node u_k. Assume that this is the case and assume that u_x for $x \geq k+1$ is the first node on the path that is reached by the message of r. There are two cases to consider, either the message of r reaches node u_x from a neighbor outside the path $P_{v_i,v_{i-1}}$ or the request is issued at node u_x. Because the first message reaching the path $P_{v_i,v_{i-1}}$ from a neighbor of the path has to be a real message, Lemma 2.2 implies that the "find predecessor" message of request r_i reaches u_x before any message from outside the path reaches u_x. However, in that case, the "find predecessor" message of r_i also reaches all earlier nodes on path $P_{v_i,v_{i-1}}$ (and thus in particular node u_k) before the message of r does. If the request r is issued at node u_x, Lemma 2.2 also implies that this has to happen after the "find predecessor" message of u_i reaches u_x. □

The following lemma is a simple consequence of Lemma 2.3.

Lemma 2.4. *Consider an asynchronous execution of* ARROW *for a given set of requests R on a tree T and consider two arbitrary requests r_i and r_j for which $i < j$ (i.e., r_i is ordered before r_j). Then, the following two statements hold:*

1. $t_i \leq t_j + d_T(v_i, v_j)$,
2. *if* $i \geq 1$, $t_i + \Delta(r_i, v_{i-1}) \leq t_j + d_T(v_j, v_{i-1})$.

Proof. If $i = 0$, we only need to prove the first claim, in which this clearly holds because $t_0 = 0$ and $t_j \geq 0$ for all $r_j \in R$. Let us therefore assume that $i \geq 1$. We consider the part of the tree T induced by the paths between the nodes v_i, v_j, and the node v_{i-1} of the predecessor request r_{i-1} of r_i. Let x be the (unique) node on the tree on which the three paths P_{v_i,v_j}, $P_{v_i,v_{i-1}}$, and $P_{v_j,v_{i-1}}$ intersect. Because x in particular is a node on the path $P_{v_i,v_{i-1}}$, from Lemma 2.3, we get that

$$t_i + \Delta(r_i, x) \leq t_j + \Delta(r_j, x). \tag{2.2}$$

The term $\Delta(r_j, x)$ is the delay of the message of request r_j to reach node x from node v_j. Because the message delay is upper bounded by the length of the path and because x is on the path P_{v_i,v_j}, we have $\Delta(r_j, x) \leq d_T(v_j, x) \leq d_T(v_j, v_i)$ and thus, the first claim of the lemma follows directly from (2.2) (note that $\Delta(r_i, x) \geq 0$). The second claim can also be proved based on (2.2):

$$\begin{aligned} t_i + \Delta(r_i, v_{i-1}) &= t_i + \Delta(r_i, x) + \big(\Delta(r_i, v_{i-1}) - \Delta(r_i, x)\big) \\ &\overset{(2.2)}{\leq} t_j + \Delta(r_j, x) + \big(\Delta(r_i, v_{i-1}) - \Delta(r_i, x)\big) \\ &\leq t_j + d_T(v_j, x) + d_T(x, v_{i-1}) \\ &= t_j + d_T(v_{i-1}, v_j). \end{aligned}$$

The second inequality follows because the message delay of an edge is at most the length of the edge. □

2.4 Cost Model

In Chapter 3, we will study the distributed queuing problem and some closely related problems by analyzing the ARROW algorithm and its modified versions on HSTs. In this section we formally provide our cost model. Assume when applying some queueing algorithm ALG to a set of request R, the resulting ordering of the requests can be seen as a permutation π_{ALG} of R such that the request ordered at position i in the order is $r_{\pi_{\text{ALG}(i)}}$. For every $i \in \{1, \ldots, |R| - 1\}$, we define the *cost of ordering* $r_{\pi_{\text{ALG}}(i)}$ *after* $r_{\pi_{\text{ALG}}(i-1)}$ as the time it takes a queueing algorithm to enqueue the request $r_{\pi_{\text{ALG}}(i)}$ as the successor of $r_{\pi_{\text{ALG}}(i-1)}$. More specifically, we assume that request $r_{\pi_{\text{ALG}}(i)}$ can be enqueued as soon as the predecessor request $r_{\pi_{\text{ALG}}(i-1)}$ is in the system and as soon as node $v_{\pi_{\text{ALG}}(i-1)}$ knows about request $r_{\pi_{\text{ALG}}(i)}$. Assume that algorithm ALG informs node $v_{\pi_{\text{ALG}}(i-1)}$ (through a message) about $r_{\pi_{\text{ALG}}(i)}$ at time $t_{\text{ALG}}(i)$.

The cost (latency) $L_{\mathsf{ALG}}(r_{\pi_{\text{ALG}}(i-1)}, r_{\pi_{\text{ALG}}(i)})$ incurred for enqueueing request $r_{\pi_{\text{ALG}}(i)}$ and the overall cost (latency) $cost_{\text{ALG}}$ of ALG are then defined as follows.

$$L_{\text{ALG}}(r_{\pi_{\text{ALG}}(i-1)}, r_{\pi_{\text{ALG}}(i)}) := \max\left\{t_{\text{ALG}}(i), t_{\pi_{\text{ALG}}(i-1)}\right\} - t_{\pi_{\text{ALG}}(i)}, \tag{2.3}$$

$$cost_{\text{ALG}}(\pi_{\text{ALG}}) := \sum_{i=1}^{|R|-1} L_{\text{ALG}}(r_{\pi_{\text{ALG}}(i-1)}, r_{\pi_{\text{ALG}}(i)}). \tag{2.4}$$

We next specify the above cost more concretely for ARROW and for an optimal offline algorithm. Assume that we have an execution $\mathcal{A}$ of the ARROW algorithm that operates on a tree T. Let $\pi_{\mathcal{A}}$ be the ordering induced by the ARROW execution $\mathcal{A}$. When the "find predecessor" message of a request $r_{\pi_{\mathcal{A}}(i)}$ arrives at the node of the predecessor request $r_{\pi_{\mathcal{A}}(i-1)}$, clearly the request $r_{\pi_{\mathcal{A}}(i-1)}$ has already occurred and thus we always have $L_{\mathcal{A}}(r_{\pi_{\mathcal{A}}(i-1)}, r_{\pi_{\mathcal{A}}(i)}) = t_{\mathcal{A}}(i) - t_{\pi_{\mathcal{A}}(i)}$ for any ARROW execution. Further note, that in a synchronous execution of arrow on tree T, because ARROW always finds the predecessor on the direct path, this latency cost is always equal to the distance between the respective nodes in T.

When studying in the cost of an optimal offline queueing algorithm $\mathcal{O}$, we assume that $\mathcal{O}$ knows the whole sequence of requests in advance. However, $\mathcal{O}$ still needs to send messages from each request to its predecessor request. The message delays are not under the control of the optimal offline algorithm. When lower bounding the cost of $\mathcal{O}$, we can therefore assume that all communication is synchronous even in the asynchronous case. Note that a synchronous execution is a possible strategy of the asynchronous scheduler. When operating on a graph G, the latency cost of $\mathcal{O}$ for ordering a request r_j as the successor of a request r_i is then exactly $L^G_{\mathcal{O}}(r_i, r_j) = \max\{t_i - t_j, d_G(v_i, v_j)\}$. As we analyze ARROW on an HST T that is simulated on top of an underlying network G, we directly define the optimal offline w.r.t. synchronous executions on the tree T as follows.

$$L^T_{\mathcal{O}}(r_{\pi^T_{\mathcal{O}}(i-1)}, r_{\pi^T_{\mathcal{O}}(i)}) := \max\left\{d_T(v_{\pi^T_{\mathcal{O}}(i-1)}, v_{\pi^T_{\mathcal{O}}(i)}), t_{\pi^T_{\mathcal{O}}(i-1)} - t_{\pi^T_{\mathcal{O}}(i)}\right\}, \tag{2.5}$$

$$cost^T_{\mathcal{O}}(\pi_{\mathcal{O}}) := \sum_{i=1}^{|R|-1} L^T_{\mathcal{O}}(r_{\pi^T_{\mathcal{O}}(i-1)}, r_{\pi^T_{\mathcal{O}}(i)}). \tag{2.6}$$

The ordering $\pi_{\mathcal{O}}$ is chosen such that the total cost $cost^T_{\mathcal{O}}(\pi_{\mathcal{O}})$ in (2.6) is minimized.

2.5 Further Related Work

The ARROW protocol is a particularly simple and elegant solution for the distributed queueing problem. The protocol was introduced independently (in slightly different forms) by Naimi and Trehel, Raymond, as well as van de Snepscheut in the context of distributed mutual exclusion [59, 63, 70]. The algorithm was later reinvented by Demmer and Herlihy [30], who used ARROW to implement a distributed directory [23]. It has been shown in [30] that the ARROW algorithm correctly solves the queueing problem even in an asynchronous system even if the requests are issued in a completely dynamic and possibly concurrent way. Over the years, ARROW has been used and analyzed in different contexts [45, 46, 49, 50, 61, 69]. The algorithm has been implemented as a part of Aleph Toolkit [45] and shown to outperform centralized schemes significantly in practice [50]. Several other tree-based distributed queueing algorithms that are similar to the ARROW algorithm have also been proposed in the literature. An algorithm that combines the ideas of ARROW with path compression has been implemented in the Ivy system [52]. The amortized cost to serve a single request is only $O(\log n)$ [41], however the algorithm needs a complete graph as the underlying network topology. There are also other similar algorithms that operate on fixed trees. The Relay algorithm [71] has been introduced as a distributed transactional memory algorithm. It is run on top of a fixed spanning tree similar to ARROW, however to more efficiently deal with aborted transactions, it does not always move the shared object to the node requesting it. Further, in [7], a distributed directory algorithm called Combine has been proposed. Combine runs on a fixed overlay tree and it is in particular shown in [7] that Combine is starvation-free.

The first paper to study the competitive ratio of concurrent executions of a distributed queueing algorithm is [48]. The paper shows that in synchronous executions of ARROW on a tree T, if we have a one-shot execution where all requests are issued at the same time, say at time 0, the total cost of ARROW is within a factor $O(\log |R|)$ compared with the optimal queueing cost on tree T. This analysis has later been extended (and slightly strengthened) to the general concurrent setting where requests are issued in an arbitrarily dynamic fashion. In [46], it is shown that in this case, the total cost of ARROW is within a factor $O(\log D)$ of the optimal cost on the tree T. Later, the same bounds have also been proven for the Relay algorithm [71] and the Combine algorithm [7]. Typically, these algorithms are run on a spanning tree or an overlay tree on top of an underlying general network topology. While the cost of all these algorithms is small when compared with the optimal queueing cost on the tree, the cost of the algorithms might be much larger when compared with the optimal cost on the underlying topology. In this case, the competitive ratio becomes $O(s \cdot \log D)$, where s is the stretch of the tree. There are underlying graphs (e.g., cycles) for which every spanning tree and even every overlay tree has stretch $\Omega(n)$ [43, 62]. The fact that even the best spanning tree might have large stretch initiated the work on distributed queueing algorithms that run on more general hierarchical structures. In [47], an algorithm called Ballistic is introduced and analyzed for the sequential and the one-shot case. Ballistic has competitive ratio $O(\log D)$, however the algorithm requires the underlying distance metric to have bounded doubling dimension and it thus cannot be applied in general networks. The best algorithm known for general networks is Spiral, which was introduced in [67]. Spiral is based on a hierarchy of overlapping clusters that cover the graph. It's general structure is thus somewhat resembling the classic sparse partitions and mobile objects solutions by Awerbuch and Peleg [13, 14]. The competitive ratio of Spiral is shown to be $O(\log^2 n \cdot \log D)$ for sequential and one-shot executions in [67]. In [66], a general framework to analyze the cost of concurrent executions of hierarchical queueing and directory algorithms has been presented. In particular, in [66], the competitive analysis of Spiral and also of the classic mobile object algorithm of Awerbuch and Peleg [13, 14] has been extended to the dynamic setting. In [46], the authors provide a sketch for how the competitive analysis of the ARROW protocol can be generalized to the asynchronous case. However, [46] does not provide a formal proof that the algorithm performs well in an asynchronous setting.

Apart from analyzing the ARROW protocol on HSTs, this thesis also shows how to formally treat the asynchronous case. We note that while we analyze the protocol on HSTs in the thesis, the generalization to asynchronous executions can also be directly applied to the analysis of [46] for general tree topologies.

Chapter 3

The Distributed Queuing Problem: Dynamic Requests

3.1 Introduction

In this chapter we study the distributed queuing problem as defined in Section 2.1 where the requests arrive in an arbitrarily dynamic and possibly concurrent fashion. As the main technical contribution, we show that when run on an HST T, the ARROW algorithm achieves a constant competitive ratio, even in the fully dynamic case. When running ARROW on an HST T, we assume that all requests R are issued at the leaf nodes of T.

The analysis of ARROW on HSTs in particular strengthens the result of [46], where it was shown that ARROW has a logarithmic competitive ratio for dynamically arriving requests on general tree topologies. In combination with the probabilistic tree embedding result of [34], our analysis implies that when run on the HST resulting from the randomized HST construction of [34], the ARROW algorithm is $O(\log n)$-competitive even on general network topologies. The best previously known competitive ratio for the distributed queueing problem with arbitrarily dynamically injected requests on general graphs is $O(\log^2 n \cdot \log D)$ as shown in [66] for the hierarchical schemes defined of [13, 67]. This shows that (under some assumptions), the simple and elegant ARROW algorithm outperforms all existing significantly more complicated distributed queueing algorithms. We note that our algorithm is based on a randomized tree construction and its competitive ratio is w.r.t. an oblivious adversary. Other algorithms with polylogarithmic competitive ratio are deterministic and they therefore also work in the presence of an adaptive adversary. For a more detailed comparison of our results with existing algorithms, we refer to the discussion in Section 2.5.

The main technical contribution is stated by the following theorem.

Theorem 3.1. *Assume that we are given an HST T with parameter 2 and queueing requests R that arrive in an arbitrarily dynamic manner at the leaves of T. When using the* ARROW *algorithm on tree T, the total cost for ordering the requests in R is within a constant factor of the cost of an optimal offline algorithm for ordering the requests R on T. This even holds if communication is asynchronous.*

Remark 3.2. *We remark that the fact that the stement of Theorem 3.1 also holds for asynchronous executions has an important implication even for synchronous systems, where the propagation delay of each edge is fixed. Assume that we are given a general synchronous network G. When using the embedding result of [34] to construct an HST T that approximately captures the distances of G (in a probabilistic sense), T is built as an overlay topology that runs on top of the underlying network G. The distance between two leaves of T is guaranteed to be lower bounded by the distance between the corresponding nodes in G. However the distance is G might be smaller and therefore the propagation delay when sending a message between two leaves of T might be smaller than the distance between the leaves in T if T if communication on T is emulated by running a synchronous on G. Since the theorem applies to general asynchronous executions on T, it in particular applies to synchronous (and asynchronous) executions on the underlying graph G.*

For a precise description of the Arrow algorithm and the definition of queueing cost, we refer to Section 2.2 and Section 2.4, respectively. Further, the formal definition of HSTs provided in Section 1.4.6. When combining Theorem 3.1 with the celebrated probabilistic tree embedding of Fakcharoenphol, Rao, and Talwar [34], we get our main result for general graphs. In [34], it is shown that there is a randomized algorithm that given an arbitrary n-point metric (X, d) constructs an HST T such all points X are mapped to leaves of T, all distances in (X, d) are upper bounded by the respective distances in T, and the expected distance between any two leaves in T is within an $O(\log n)$ factor of the distance between the corresponding two points in X. When constructing such an HST T for a given graph G and when assuming an oblivious adversary[1], this implies that the expected total cost of Arrow on T is within an $O(\log n)$ factor of the optimal offline queueing cost on G. We also note that an efficient distributed construction of the HST embedding of [34] has been given in [40].

Theorem 3.3. *Assume that we are given an arbitrary graph $G = (V, E)$ and queueing requests R that arrive in an arbitrarily dynamic manner at the nodes of G. There is a randomized construction of an HST T that can be simulated on G such that when running* Arrow *on T, we get a distributed queueing algorithm for G with competitive ratio at most $O(\log n)$ against an oblivious adversary providing the sequence of requests. This even holds if communication is asynchronous.*

Organization of the Chapter: The remainder of this chapter is organized as follows. In Section 3.2, we first (informally) show that Arrow is optimal when it is run on top of an HST for one-shot executions where the requests arrive at the same time. Section 3.3 contains some lemmas that establish some basic properties that are needed for the rest of the chapter. Section 3.4 analyzes the cost of an optimal offline algorithm on an HST T by relating it to the total weight of an MST defined on the set of requests. In Section 3.5, we introduce a general framework to analyze the queueing cost of distributed queueing algorithms on an HST T and the framework is applied to synchronous executions of the Arrow algorithm. The analysis of

[1]That is, when assuming that the sequence of requests is statistically independent of the randomness used to construct the HST T.

asynchronous executions appears in Section 3.6. In Section 3.7, we prove a general minimum spanning tree (MST) approximation result that is used in Section 3.4 and Section 3.5. Some open problems are discussed, finally, in Section 3.8.

3.2 One-Shot Executions

As a warm-up, we first provide a simple (informal) analysis that shows that when run on an HST T, the ARROW protocol provides an optimal solution for so-called one-shot execution, that is, for executions, where all requests are entered into the system at time 0. The special case of one-shot executions was already considered in [49]. Consider the metric space (R, d_T) defined by the set of requests R and where the distance between two requests $r, r' \in R$ is the distance between the corresponding leaf nodes in T. It was already observed in [49] that the optimal ordering of the requests on T corresponds to an optimal TSP path that starts at the dummy request and the optimal offline cost therefore equals the length of an optimal TSP path. By applying the greedy property of the ARROW algorithm (cf. Lemma 2.1), the ARROW order corresponds to the TSP path that results from using the nearest neighbor heuristic, i.e., the successor of a request r is a request r' that minimizes $d_T(r, r')$ among all the remaining requests. It is known that for a general metric space of size n, the nearest neighbor heuristic always produces a TSP path that is within an $O(\log n)$-factor of an optimal TSP path [64]. As a consequence, it is shown in [49] that for one-shot scenarios, ARROW is $O(\log |R|)$-competitive.

To analyze the one-shot scenario on HSTs, we therefore need to understand the TSP nearest neighbor heuristic for a metric space that is induced by the distances between leaves of an HST T. Recall that an HST T is a tree in which all nodes at depth ℓ (i.e., at ℓ hops from the root) have the same distance to the root and in which all leaves are at the same depth h (i.e., the same hop distance from the root). The distance between two leaves therefore only depends on the depth of the nearest common ancestor and it monotonically increases when the depth of the nearest common ancestor decreases. As a consequence, when running the nearest neighbor heuristic, a TSP path will always first visit all requests in the current subtree before leaving the subtree. For $\ell \geq 0$, n_ℓ be the number of subtrees that are rooted at a node at depth ℓ of T and for which there is at least one node that has a request (e.g., n_0 is the number of subtrees of the root of T in which there is at least one request). As the nearest neighbor TSP path leaves each subtree of T exactly once, the number of times it passes through a node at depth at most ℓ is exactly $n_\ell - 1$. If δ_ℓ is the distance between two leaves of T for which the nearest common ancestor is at depth ℓ, the total length of a nearest neighbor TSP path is therefore

$$\mathrm{TSP}_{\mathrm{NN}} = \sum_{\ell=0}^{h-1} (n_\ell - n_{\ell-1}) \cdot \delta_\ell,$$

where for convenience n_{-1} is defined as $n_{-1} := 0$. It is clear that any spanning tree that connects all requests must have at least $n_\ell - 1$ edges of length at least δ_ℓ because all n_ℓ subtrees with requests rooted at nodes at depth ℓ need to be connected. The length of a nearest neighbor tour is therefore even equal to the total length of all edges of a minimum spanning tree and thus also to the total length of an optimal TSP path. We therefore conclude that for synchronous, one-shot executions on an HST T, the ARROW protocol provides an optimal solution. We further note that with a similar, slightly more complicated argument, based on the asynchronous greedy property of Lemma 2.2, one can even show that ARROW is optimal for asynchronous one-shot executions on an HST T.

3.3 Preliminaries

In this section, we provide some basic lemmas that we need for the analysis of the ARROW protocol in the dynamic setting The first lemma shows that when using the randomized HST construction of [34], the expected cost (2.6) is within a logarithmic factor of the optimal offline cost on the underlying network graph G.

Lemma 3.4. *Assume T is an HST that is constructed on top of an n-node network graph G by using the randomized algorithm of [34] and assume that there is a dynamic set of queueing requests issued at the nodes of G. If the sequence of requests is independent of the randomness of the randomized HST construction, the expected optimal total cost on T (as defined in (2.6)) is within a factor $O(\log n)$ of the optimal offline queueing cost on G.*

Proof. Let $\pi_{\mathcal{O}}^G$ and $\pi_{\mathcal{O}}^T$ be the optimal orderings w.r.t. the optimal offline costs $L_{\mathcal{O}}^G(r_i, r_j)$ and $L_{\mathcal{O}}^T(r_i, r_j)$ on G and T, respectively, as defined above. We have

$$\begin{aligned}
\mathbb{E}\left[cost_{\mathcal{O}}^T(\pi_{\mathcal{O}}^T)\right] &= \mathbb{E}\left[\sum_{i=1}^{|R|-1} L_{\mathcal{O}}^T(r_{\pi_{\mathcal{O}}^T(i-1)}, r_{\pi_{\mathcal{O}}^T(i)})\right] \\
&\leq \sum_{i=1}^{|R|-1} \mathbb{E}\left[L_{\mathcal{O}}^T(r_{\pi_{\mathcal{O}}^G(i-1)}, r_{\pi_{\mathcal{O}}^G(i)})\right] \\
&= \sum_{i=1}^{|R|-1} \mathbb{E}\left[\max\left\{d_T(v_{\pi_{\mathcal{O}}^G(i-1)}, v_{\pi_{\mathcal{O}}^G(i)}), t_{\pi_{\mathcal{O}}^G(i-1)} - t_{\pi_{\mathcal{O}}^G(i)}\right\}\right] \\
&\leq 2 \cdot \sum_{i=1}^{|R|-1} \max\left\{\mathbb{E}\left[d_T(v_{\pi_{\mathcal{O}}^G(i-1)}, v_{\pi_{\mathcal{O}}^G(i)})\right], t_{\pi_{\mathcal{O}}^G(i-1)} - t_{\pi_{\mathcal{O}}^G(i)}\right\} \\
&\leq 2 \cdot \sum_{i=1}^{|R|-1} \max\left\{O(\log n) \cdot d_G(v_{\pi_{\mathcal{O}}^G(i-1)}, v_{\pi_{\mathcal{O}}^G(i)}), t_{\pi_{\mathcal{O}}^G(i-1)} - t_{\pi_{\mathcal{O}}^G(i)}\right\} \\
&\leq O(\log n) \cdot \sum_{i=1}^{|R|-1} \max\left\{d_G(v_{\pi_{\mathcal{O}}^G(i-1)}, v_{\pi_{\mathcal{O}}^G(i)}), t_{\pi_{\mathcal{O}}^G(i-1)} - t_{\pi_{\mathcal{O}}^G(i)}\right\} \\
&\leq O(\log n) \cdot cost_{\mathcal{O}}^G(\pi_{\mathcal{O}}^G).
\end{aligned}$$

The first inequality follows from the fact that $\pi_{\mathcal{O}}^T$ is an optimal ordering w.r.t. the cost $L_{\mathcal{O}}^T(r_i, r_j)$ and by linearity of expectation. The second inequality follows because for every non-negative random variable X and every fixed (possibly negative) constant c, it holds that $\mathbb{E}[\max\{X, c\}] \leq 2 \cdot \max\{\mathbb{E}[X], c\}$. The third inequality follows from the expected stretch bound of the HST construction of [34], and the fourth inequality follows because for all values $\lambda \geq 1$, $a \geq 0$ and $b \in \mathbb{R}$, it holds that $\max\{\lambda a, b\} \leq \lambda \cdot \max\{a, b\}$. □

Given Theorem 3.1 (which will be proven as the main technical result of this chapter) and Lemma 3.4, we immediately get Theorem 3.3. We note in light of the remark following the statement of Theorem 3.1, the statement of Theorem 3.3 is also true for synchronous executions on the underlying graph G.

Manhattan Cost: In the dynamic competitive analysis of ARROW on general trees in [46], it has been shown that it is useful to study the optimal ordering w.r.t. to the following *Manhattan cost* on a tree T between two queueing requests $r_i = (v_i, t_i)$ and $r_j = (v_j, t_j)$.

$$c^T_{\mathcal{M}}(r_i, r_j) := d_T(v_i, v_j) + |t_i - t_j|. \tag{3.1}$$

As the cost function $c_{\mathcal{M}}(r_i, r_j)$ defines a metric space on the request set, the problem of finding an optimal ordering w.r.t. the cost $c_{\mathcal{M}}(r_i, r_j)$ is a metric TSP problem.[2] As a result, we will for example use that the total weight of an MST on the set of request w.r.t. the weight function $c_{\mathcal{M}}(r_i, r_j)$ is within a factor 2 of the cost of an optimal TSP path. The following definition is inspired by Lemma 3.12 in [46].

Definition 3.5 (Condensed Request Set)**.** *A set R of queueing requests $r_i = (v_i, t_i)$ on a tree T is called* condensed *if for any two requests $r_i = (v_i, t_i)$ and $r_j = (v_j, t_j)$ that are consecutive w.r.t. time of occurrence, there exits requests $r_a = (v_a, t_a)$ and $r_b = (v_b, t_b)$ such that $t_a \leq t_i$, $t_b \geq t_j$, and $d_T(v_a, v_b) \geq t_b - t_a$.*

It is shown in [46] that for condensed request sets, the total optimal Manhattan cost is within a constant factor of the optimal offline queueing cost.

Lemma 3.6 (Lemma 3.17 in [46] rephrased)**.** *If the request set R is condensed, then on any tree T and for every ordering π on the requests, it holds that*

$$\sum_{i=1}^{|R|-1} c^T_{\mathcal{M}}(r_{\pi(i-1)}, r_{\pi(i)}) \leq 12 \cdot \sum_{i=1}^{|R|-1} L^T_{\mathcal{O}}(r_{\pi(i-1)}, r_{\pi(i)}).$$

For synchronous executions on trees, it is also shown in [46] that every request set R can be transformed into a condensed request set without changing the ordering (and the cost) of ARROW and without increasing the optimal offline cost.

Lemma 3.7 (Lemma 3.11 in [46] rephrased)**.** *Let R be a set of queueing requests issued on a tree T and let $r_i = (v_i, t_i)$ and $r_j = (v_j, t_j)$ be two requests of R that are consecutive w.r.t. time of occurrence. Further, choose two requests $r_a = (v_a, t_a)$ with $t_a \leq t_i$ and $r_b = (v_b, t_b)$ with $t_b \geq t_j$ minimizing $\delta := t_b - t_a - d_T(v_a, v_b)$. if $\delta > 0$, every request $r = (v, t)$ with $t \geq t_j$ can be replaced by a request $r' = (v, t - \delta)$ without changing the synchronous* ARROW *order and without increasing the optimal offline cost.*

[2]The relation of ARROW and the TSP was already exploited in [46] when analyzing ARROW on general trees.

Lemma 3.7 implies that every request set R can be transformed into a condensed set R' without changing the synchronous order of ARROW and without increasing the optimal offline cost. For the analysis of ARROW in synchronous systems, we can thus w.l.o.g. assume that the request set is condensed. In Section 3.6, we show that this also holds in asynchronous systems.

3.4 Analysis of the Optimal Offline Cost

This and the next section discuss the main technical contribution of this chapter and analyzes the total cost of a synchronous ARROW execution when run on an HST T. Throughout this section, we assume that a fixed HST T, a set of dynamic requests R placed at the leaves of T, and a synchronous execution of ARROW with request set R on T are given. For convenience, we relabel the requests in R so that they are ordered according to the queueing order resulting from the given ARROW execution on T. That is, we assume that for all $i \in \{0, \ldots, |R|-1\}$, request $r_i = (v_i, t_i)$ is the i^{th} request in ARROW's order. Note that $r_0 = (v_0, 0)$ is still the dummy request defining the initial tail of the queue. Before delving into the details of the analysis, we give a short outline. In the first step in Section 3.4.1, we study the ordering generated by ARROW in more detail and show that it implies a hierarchical partition of the requests R in a natural way. To simplify the next Section 3.4.2 transforms the given HST T into a new tree such that inside each subtree, if ordering the request by time of occurrence, the gap between the times of consecutive requests cannot be too large (whenever such a gap is too large, we split the corresponding subtree into two trees). Section 3.4.3 then shows that the optimal offline cost can be characterized by the total Manhattan cost of a spanning tree that respects the hierarchical structure of the HST T in a best given way. Finally, in Section 3.5, we give a general framework to compare the queueing cost of an online distributed algorithm on an HST T to the optimal offline cost on T and we apply this method to synchronous ARROW executions. In Section 3.6, we show that the same framework can also be applied to general asynchronous ARROW executions.

3.4.1 Characterizing ARROW By A Hierarchical Partition of R

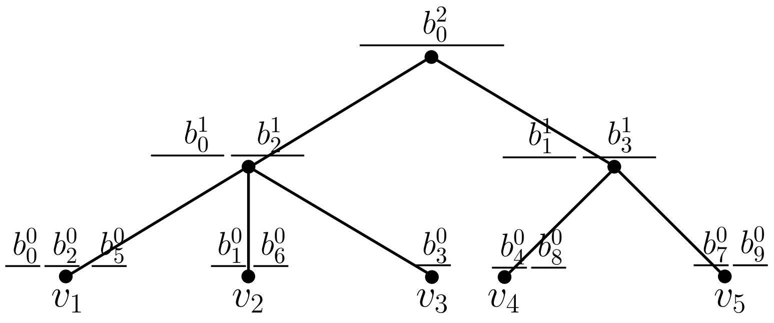
We hierarchically partition the requests R according to the ARROW queueing order and the hierarchical structure of the HST T. On each level ℓ of T, we partition the requests into blocks, where a block of requests is a maximal set of requests that are ordered consecutively by ARROW inside some level-ℓ subtree of T. In the following, for non-negative integers s and t, we use the abbreviations $[s] := \{0, \ldots, s-1\}$ and $[s,t] := \{s, \ldots, t\}$. Formally, instead of partitioning the set of requests R directly, we partition the set of indexes $[|R|]$. Recall that the requests in R are indexed consecutively according to the queueing order of ARROW.

Definition 3.8 (Hierarchical Block Partition). *For each level $\ell \in [0, h]$, we partition $[|R|]$ into $n(\ell)$ blocks $\left\{b_0^\ell, b_1^\ell, \cdots, b_{n(\ell)-1}^\ell\right\}$ such that*

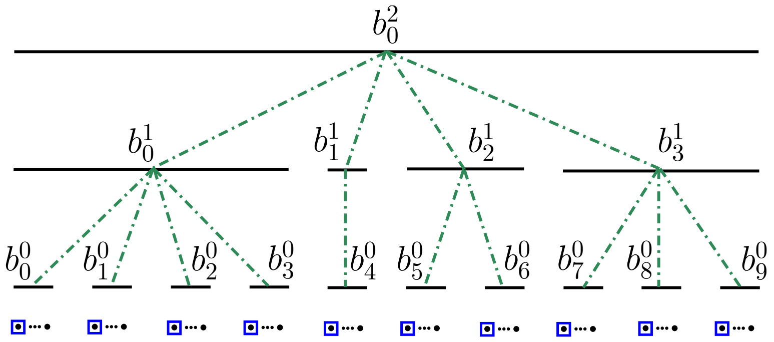
1. *each block is a consecutive set of integers (i.e., a consecutively ordered set of requests),*

2. *for every block b_i^ℓ, all requests r_p for $p \in b_i^\ell$ are in the same level-ℓ subtree of T, and*
3. *for all $i, j \in [n(\ell)]$ and all $p \in b_i^\ell$ and $q \in b_j^\ell$, $i < j \implies p < q$.*

For each block b, we further define the first request of b to be the one that has minimum index in b.



(a) Blocks within the same subtree



(b) Tree induced by the block hierarchy

Figure 3.1: The partition of R. (a) An HST with height 2 and 5 leaves. The leaves issue requests at different times. The issued requests by nodes v_1, v_2, and v_3 are partitioned into the blocks b_0^1 and b_2^1 on level 1. These two blocks are called neighbor blocks at a subtree rooted at height 1. (b) The corresponding 4 level-wise partition based on ARROW's order that forms a parent-child relation between the blocks on different levels. Blue boxes include the requests that are ordered first by ARROW among all requests in blocks b_i^0 for all $i \in [0, 9]$.

Note that for each level ℓ and for the first block of this level, the first request of the block has index 0. The block partition defined in Definition 3.8 is illustrated in Figure 3.1. Figure 3.1a shows the blocks within the HST structure, whereas Figure 3.1b shows the hierarchical partition induced by the blocks. To simplify the presentation of our analysis, we also define a level -1 block b_i^{-1} for each individual request r_i. Note that we have $n(-1) = |R|$. The following definition allows to navigate through the block hierarchy.

Definition 3.9 (Children Blocks). *The set of children blocks of a block b_i^ℓ on a level $\ell \in [0, h]$ is defined as $child(b_i^\ell) := \{b_j^{\ell-1} : b_j^{\ell-1} \subseteq b_i^\ell\}$. Block b_i^ℓ is called the parent block of each of the blocks in $child(b_i^\ell)$.*

In Figure 3.1b, block b_2^1 is the parent block of its children blocks b_5^0 and b_6^0. Block b_1^1 has only one child block b_4^0 and thus $b_1^1 = b_4^0$.

The blocks $\left\{b_0^\ell, b_1^\ell, \cdots, b_{n(\ell)-1}^\ell\right\}$ of level ℓ belong to the subtrees rooted at height ℓ of the HST T. Note that by the definition of the block partition, no two consecutive blocks at the same level ℓ belong to the same level-ℓ subtree of T. The next definition specifies notation to argue about blocks of the same subtree of T.

Definition 3.10 (Blocks of Same Subtree). *If two blocks b_i^ℓ and b_j^ℓ belong to the same level-ℓ subtree of T, this is denoted by $\widehat{b_i^\ell b_j^\ell}$. Moreover, $|\widehat{b_i^\ell b_j^\ell}| := \left|\left\{w : i < w < j \wedge \widehat{b_i^\ell b_w^\ell} \text{ holds}\right\}\right|$. Two blocks b_i^ℓ and b_j^ℓ are called* neighbor blocks *if $\widehat{b_i^\ell b_j^\ell}$ and $|\widehat{b_i^\ell b_j^\ell}| = 0$.*

In Figure 3.1a, blocks b_0^0, b_2^0, and b_5^0 are within the same subtree rooted at node v_1. Blocks b_0^0 and b_5^0 are not neighbor blocks, however blocks b_0^0 and b_2^0, as well as blocks b_2^0 and b_5^0 are neighbor blocks. The next lemma lists a number of simple properties of the block partition.

Lemma 3.11. *The block partition of Definition 3.8 satisfies the following properties:*

1. *For every block b_i^ℓ and for all $p, q \in b_i^\ell$, we have $d_T(v_p, v_q) \leq \delta(\ell)$.*
2. *For each level ℓ and all level-ℓ blocks b_i^ℓ and b_j^ℓ, if $\widehat{b_i^\ell b_j^\ell}$ holds, for any $p \in b_i^\ell$ and $q \in b_j^\ell$, we have $d_T(v_p, v_q) \leq \delta(\ell)$.*
3. *For each level ℓ and all level-ℓ blocks b_i^ℓ and b_j^ℓ, if $\widehat{b_i^\ell b_j^\ell}$ does* not *hold, for all $p \in b_i^\ell$ and $q \in b_j^\ell$, we have $d_T(v_p, v_q) \geq \delta(\ell + 1)$.*
4. *Assume $\ell < h$ and consider two blocks b_i^ℓ and b_j^ℓ that have a common parent block $b_w^{\ell+1}$, but for which $\widehat{b_i^\ell b_j^\ell}$ does not hold. Then, for all $p \in b_i^\ell$ and $q \in b_j^\ell$, we have $d_T(v_p, v_q) = \delta(\ell + 1)$.*

Proof. Recall that the distance between two leaves u, v of the HST T is equal to $\delta(\ell)$ if the least common ancestor of u and v is on level ℓ. The first claim then holds because all requests in a block b_i^ℓ at level ℓ are issued at nodes in the same level-ℓ subtree of T and therefore the least common ancestor of any two of them is on level at most ℓ. The second claim holds for a similar reason. If $\widehat{b_i^\ell b_j^\ell}$ holds for two blocks b_i^ℓ and b_j^ℓ, both blocks consist of requests in the same level-ℓ subtree of T. For the third claim, note that when $\widehat{b_i^\ell b_j^\ell}$ does not hold for two blocks b_i^ℓ and b_j^ℓ, the two blocks do not belong to the same subtree at level ℓ. Therefore for any two requests $p \in b_i^\ell$ and $q \in b_j^\ell$, the least common ancestor has to be on level at least $\ell + 1$ and thus the distance $d_T(v_p, v_q) \geq \delta(\ell + 1)$. Finally, the fourth claim holds by combining the second claim (applied to block $b_w^{\ell+1}$ on level $\ell + 1$) and the third claim. $\square$

We have seen that in a synchronous ARROW execution, the latency cost for ordering request r_{i+1} as the successor of r_i is exactly the distance $d_T(v_i, v_{i+1})$ between the nodes of the two requests. The total cost of ARROW therefore directly follows from the structure of the block partition.

Lemma 3.12. *The total cost of a synchronous* ARROW *execution on the HST T with corresponding hierarchical block partition is given by*

$$cost_{\mathcal{A}}(\pi_{\mathcal{A}}) = \sum_{\ell=0}^{h-1} \big(n(\ell) - n(\ell+1)\big) \cdot \delta(\ell+1).$$

Proof. It follows from claim 4 of Lemma 3.11 that for any two requests r and r', $d_T(r, r') = \delta(\ell+1)$ for the smallest ℓ for which r and r' are in the same level-ℓ block. The block partition implies that for every level ℓ, there are $n(\ell) - 1$ consecutive requests r_i and r_{i+1} which are in different level-ℓ blocks. For every $\ell \in \{0, \ldots, h-1\}$, the number of consecutive request pairs at distance at least $\delta(\ell+1)$ is therefore equal to $n(\ell) - 1$. The claim of the lemma now follows because $cost_{\mathcal{A}}(\pi_{\mathcal{A}}) = \sum_{i=1}^{|R|-1} d_T(v_{i-1}, v_i)$. □

3.4.2 HST Conversion

In this section, a recursive (top-down) splitting procedure is provided so that the original HST is converted into a new HST with better properties. The conversion does not change the total cost of ordering the requests by ARROW (in fact, it does not change the block partition). Further, the total Manhattan cost of optimal offline algorithm's order asymptotically remains unchanged as well. We describe how the splitting procedure works and we then argue its properties. For a set R' of queueing request (and sometimes by overloading notation also for a set of request indexes), we define $t_{\min}(R')$ and $t_{\max}(R')$ to be the minimum and the maximum issue time t of any request $r = (v, t) \in R'$, respectively.

Splitting Procedure: We describe the splitting procedure as it is applied to a subtree T' that is rooted at a given level $\ell \in \{0, \ldots, h\}$ of T. If $\ell = 0$, the tree T' is returned unchanged. Otherwise ($\ell \geq 1$), we go through all level-($\ell-1$) subtrees T'' of T'. As long as the tree T'' has two neighbor blocks $b_i^{\ell-1}$ and $b_j^{\ell-1}$ (for $i < j$) for which the following condition (3.2) is true, the subtree T'' is split into two separate subtrees T_1'' and T_2'' of T'.

$$t_{\min}(b_j^{\ell-1}) - t_{\max}(b_i^{\ell-1}) \geq \delta(\ell). \tag{3.2}$$

The splitting of T'' into T_1'' and T_2'' works as follows. The topology of T_1'' and T_2'' is identical to the topology of T''. Each request $r = (v, t)$ that is issued at some node v of T'' is either placed on the isomorphic copy of v in T_1'' or in T_2''. All requests r in blocks $b_x^{\ell-1}$ of T'' for $x \leq i$ are placed in tree T_1'' and all request in blocks $b_y^{\ell-1}$ of T'' for $y \geq j$ are placed in tree T_2''. We perform such splittings for trees T' of level ℓ as long as there are subtrees of T' on level $\ell - 1$ with neighbor blocks that satisfy Condition (3.2). As soon as no such neighbor blocks exist, the procedure is applied recursively to all trees T'' at level $\ell - 1$ (including the new subtrees). The whole conversion is started by applying the procedure to the complete HST T.

Lemma 3.13. *The above splitting procedure does not change the hierarchical block partition and it thus also preserves* ARROW*'s queueing order $\pi_{\mathcal{A}}$ and its total cost $cost_{\mathcal{A}}(\pi_{\mathcal{A}})$.*

Proof. We prove that a single splitting step does not change the block partition or the ARROW cost. The lemma then follows by induction on the number of splits in the above procedure. Assume that we are working on tree T' on level ℓ and that we are splitting subtree T'' of T' into T''_1 and T''_2 as a result of two neighbor blocks $b_i^{\ell-1}$ and $b_j^{\ell-1}$ satisfying Condition (3.2).

We first show that w.r.t. ARROW's ordering $\pi_{\mathcal{A}}$ before the splitting step, the block partition remains the same. W.r.t. the ordering $\pi_{\mathcal{A}}$, the block partition can only change if some block of level $\ell' \leq \ell - 1$ at a subtree of T'' is split into two blocks. Note that any subtree τ of T that is rooted at some node v outside T'' either does not contain any node of T'' or it contains the whole subtree T''. In both cases, the request set of τ does not change and w.r.t. ordering $\pi_{\mathcal{A}}$ therefore also their blocks on the level of node v remain the same. Because the blocks at some level $\ell' < \ell - 1$ of tree T'' are a refinement of the blocks on level $\ell - 1$, if some block of some level $\ell' \leq \ell - 1$ at a subtree of T'' is split, there is also a level-$(\ell - 1)$ block of tree T'' is split into two blocks. However this cannot happen because the splitting procedure moves each level-$(\ell - 1)$ block of T'' either completely to T''_1 or to T''_2. Hence, w.r.t. the ordering $\pi_{\mathcal{A}}$ before the splitting, the block partition remains the same.

We next show that this implies that for all pairs of requests (r_i, r_{i+1}) ordered consecutively by ARROW, the tree distance $d_T(v_i, v_{i+1})$ remains the same. If it does not remain the same, it means that v_i and v_{i+1} are both within T'' and thus before the split $d_T(v_i, v_{i+1}) \leq \delta(\ell - 1)$ (their least common ancestor is some node in T''). Hence, r_i and r_{i+1} are in the same block on level $\ell - 1$. To see this, recall that the blocks of level $\ell - 1$ of T'' are the maximal set of requests inside tree T'' that are ordered consecutively by ARROW. Because r_i and r_{i+1} are ordered consecutively, they therefore have to be in the same level $\ell - 1$ block of T''. After the split, we then have $d_T(v_i, v_{i+1}) = \delta(\ell)$ and thus r_i and r_{i+1} cannot be in the same block at level ℓ' any more. As the splitting does not change the block partition (w.r.t. the original ordering $\pi_{\mathcal{A}}$), this cannot happen. Hence, we have that for every $i \in \{0, \ldots, |R| - 2\}$, $d_T(v_i, v_{i+1})$ remains unchanged. All other distances can only increase. Hence, even after the split, for every $i \in \{0, \ldots, |R| - 2\}$, request r_{i+1} still minimizes $t + d_T(v, v_i)$ among all non-ordered requests $r = (v, t)$. Lemma 2.1 therefore implies that $\pi_{\mathcal{A}}$ is still a valid ARROW ordering. Because the block partition remains the same, Lemma 3.12 also immediately implies that $cost_{\mathcal{A}}(\pi_{\mathcal{A}})$ remains unchanged. Because when splitting tree T'', every level-$(\ell - 1)$ block of T'' either completely goes to tree T''_1 or to tree T''_2, the splitting does not divide any block. Hence, if we assume that the queueing order $\pi_{\mathcal{A}}$ is preserved, also the block partition is preserved. □

The next lemma shows that if a tree T'' is split into two trees T''_1 and T''_2 such that all requests in T''_1 are ordered before all requests in T''_2, there is a significant time of occurrence gap between the requests ending up in subtrees T''_1 and T''_2.

Lemma 3.14. *Assume that we are performing a single splitting. Further, assume that we are working on a tree T' on level ℓ and that we are splitting a subtree T'' of T' into T''_1 and T''_2 such that T''_1 obtains the blocks that are scheduled first by* ARROW. *If R_1 and R_2 are the request sets of T''_1 and T''_2, respectively, we have $t_{\min}(R_2) - t_{\max}(R_1) \geq \delta(\ell) - \delta(\ell - 1)$.*

Proof. Assume that the split of the tree T'' is caused by two neighbor blocks $b_i^{\ell-1}$ and $b_j^{\ell-1}$ satisfying Condition (3.2). We first show that $t_{\min}(R_2) = t_{\min}(b_j^{\ell-1})$. To see this, we generally show that for any subset of blocks $b^x_{i_1}, b^x_{i_2}, \dots$ of some tree $\bar{T}$ rooted at level x, if $b^x_{i_1}$ is the first of these blocks ordered by ARROW, then the first request ordered in $b^x_{i_1}$ has the smallest time of occurrence among all requests in blocks $b^x_{i_1}, b^x_{i_2}, \dots$. To see this, note that whenever ARROW enters a level-x block b^x_i of tree $\bar{T}$, the predecessor request r is at a node v outside tree $\bar{T}$. As a consequence, all leaf nodes in $u \in \bar{T}$ and thus all requests in $\bar{T}$ are at the same distance from v in the HST T. Therefore Lemma 2.1 implies that the successor of r is a request with minimum time of occurrence.

It remains to show that

$$t_{\max}(R_1) \le t_{\max}(b^{\ell-1}_i) + \delta(\ell - 1). \tag{3.3}$$

Assume that $r_p = (v_p, t_p)$ is a request from R_1 with $t_p = t_{\max}(R_1)$. Further, assume that $r_q = (v_q, t_q)$ is the last request ordered by ARROW among the requests in R_1. Note that request r_q needs to be inside block $b^{\ell-1}_i$ because that is the last level-$(\ell-1)$ block that is assigned to tree T''_1. Hence, we clearly have $t_q \le t_{\max}(b^{\ell-1}_i)$. Therefore, if $r_p = r_q$ (3.3) clearly holds. We can therefore assume that r_p is ordered before r_q by ARROW. Consider the predecessor r_{p-1} of request r_p. From the second claim of Lemma 2.1, we have

$$t_p - t_q \le d_T(v_p, v_q). \tag{3.4}$$

Since both r_p and r_q are in T'' then $d_T(v_p, v_q) \le \delta(\ell - 1)$ thus (3.3) holds. □

It remains to show that the splitting also does not affect the optimal offline cost in a significant way. The following lemma shows that the Manhattan cost $c_{\mathcal{M}}(r, r')$ for any two requests r and r' can increase by at most a factor 3. Hence, also the total Manhattan cost of an optimal ordering cannot increase by more than a factor 3.

Lemma 3.15. *For any two requests r and r', the splitting procedure does not increase the Manhattan cost $c_{\mathcal{M}}(r, r')$ by more than a factor 3.*

Proof. We prove that a) by every single splitting, the Manhattan cost $c_{\mathcal{M}}(r, r')$ can at most increase by a factor of 3 and b) the Manhattan cost $c_{\mathcal{M}}(r, r')$ is affected by at most one splitting. Assume that $r = (v, t)$ and $r' = (v', t')$. Clearly, the issue times t and t' are not affected by the splitting. The Manhattan cost can therefore only change because $d_T(v, v')$ changes. We first show that this can happen at most once. When working on tree T' at level ℓ, a splitting divides a subtree T'' at level $\ell - 1$ into two subtrees T''_1 and T''_2. Hence, when working on level ℓ, if two nodes are affected by the splitting their distance in T' increases from at most $\delta(\ell - 1)$ to exactly $\delta(\ell)$. Therefore, after separating two nodes v and v' because of a splitting for a tree T' on level ℓ, the two nodes cannot be affected by another splitting on a level $\ell' \ge \ell$. Claim b) now follows because we do the splitting in a top-down way, i.e., throughout the splitting procedure the levels on which we split are monotonically non-increasing.

To prove claim a), let us assume that $r = (v, t)$ and $r' = (v', t')$ are affected by a splitting when a tree T'' at level $\ell - 1$ is split into two trees T''_1 and T''_2. We have already seen that this implies that after the splitting, we have $d_T(v, v') = \delta(\ell)$. It further follows from Lemma 3.14 that $|t - t'| \ge \delta(\ell) - \delta(\ell - 1) > \delta(\ell)/2$. Hence, before the splitting, we have $c_{\mathcal{M}}(r, r') \ge |t - t'|$ and after the splitting, we have $c_{\mathcal{M}}(r, r') \le |t - t'| + d_T(v, v') < 3 \cdot |t - t'|$. □

For the remainder of the analysis in this section (and also in Section 3.6), we assume that the HST T is an HST that is obtained after applying the splitting procedure recursively. We therefore assume that for every level ℓ and every subtree T' at level ℓ, there is no level-($\ell - 1$) subtree T'' of T' that contains two neighbor blocks that satisfy Condition (3.2).

3.4.3 Lower Bounding The Optimal Manhattan Cost

In this section, we construct a tree S^* that spans all requests in R. The tree S^* has a nice hierarchical structure: For each subtree T' of T, the set edges of S^* induced by the request set of the subtree T' forms a spanning tree of the request set of T'. Apart from this useful structural property, we will show that the total Manhattan cost of the spanning tree S^* is within a constant factor of minimum spanning tree (MST) of the request set R w.r.t. the Manhattan cost. We have seen that on condensed request sets, the optimal TSP path of the request set w.r.t. the Manhattan cost is within a constant factor of the optimal offline queueing cost. Note that because any TSP path is also a spanning tree, this implies that the total Manhattan cost of the MST and thus also the total Manhattan cost of the tree S^* are lower bounding the optimal offline queueing cost within a constant multiplicative factor.

Throughout this section, for convenience, we add one more level to the HST T. Instead of placing the requests at the leaves on level 0, we assume that each level 0 node v has a child node on level -1 for each of the requests issued at node v. Hence, the new leaf nodes are on level -1 and each leaf node receives exactly one request. Note that subtrees of T that do not have any queuing requests can be ignored and therefore, we can w.l.o.g. assume that every leaf node issues some queueing request. The distance between a level -1 node and its parent on level 0 is set to be 0.

Spanning Tree Construction: The spanning tree S^* is constructed greedily in a bottom-up fashion. For each subtree T' of T, we recursively define a tree $S^*(T')$ as follows. For the leaf nodes on level -1, the tree consists of the single request placed at the node. For a tree T' rooted at a node v on level $\ell \geq 0$, the tree $S^*(T')$ consists of the recursively constructed trees $S^*(T''_1), S^*(T''_2), \dots$ of the subtrees $T''_1, T''_2, \dots$ of T'' and of edges connecting the trees $S^*(T''_1), S^*(T''_2), \dots$ to a spanning tree of the set of request issued at leaves of tree T'. The edges for connecting the trees $S^*(T''_1), S^*(T''_2), \dots$ are chosen so that they have minimum total Manhattan cost. That is, to connect the trees $S^*(T''_1), S^*(T''_2), \dots$, we compute an MST of the graph we get if each of the trees $S^*(T''_i)$ is contracted to a single node. We can therefore for example choose the edges to connect the trees $S^*(T''_1), S^*(T''_2), \dots$ in e greedy way: Always add the lightest (w.r.t. Manhattan cost) edge that does not close a cycle with the already existing edges, including the edges of the trees $S^*(T''_1), S^*(T''_2), \dots$.

MST Approximation: In the following, it is shown that the total Manhattan cost of the tree $S^* = S^*(T)$ is within a constant factor of the cost of an MST w.r.t. the Manhattan cost. Where convenient, we identify a tree τ with its set of edges, i.e., we also use S^* to denote the set of edges of the tree S^*. Further, the cost of an edge $e = \{r, r'\}$ is the Manhattan cost $c_{\mathcal{M}}(r, r')$.

We also slightly abuse notation and use $c_{\mathcal{M}}(e)$ to denote this cost. The proof applies a general MST approximation result that appears in Theorem 3.26 in Section 3.7. Together with the following technical lemma, Theorem 3.26 directly implies that the total Manhattan cost of S^* is within a factor 4 of the MST Manhattan cost. For a subtree T' of T, we use $R(T')$ to denote the subset of the requests R that are issued at nodes of T'.

Lemma 3.16. *Consider the constructed spanning tree S^* and consider an arbitrary edge e of S^*. Let S_1^* and S_2^* the two subtrees that result when removing edge e from S^*. Further, assume e^* be an edge that connects the two subtrees S_1^* and S_2^* and that has minimum Manhattan cost among all such edges. We then have $c_{\mathcal{M}}(e) \leq 4 \cdot c_{\mathcal{M}}(e^*)$.*

Proof. Assume that the edge $e = \{r_p, r_q\} \in S^*(\tau)$ is an edge that connects two subtrees of a subtree τ of T that is rooted at some level $\ell \in [0, h]$. Further, let $V_{S_1^*}$ and $V_{S_2^*}$ be the node sets of the two subtrees of S_1^* and S_2^*.

Let us first assume that $|t_p - t_q| \leq 3 \cdot \delta(\ell)$. All edges including e^* from the metric $(R, c_{\mathcal{M}})$ that cross the cut $(V_{S_1^*}, V_{S_2^*})$ have length at least $\delta(\ell)$ since $d_T(v_w, v_z) \geq \delta(\ell)$ for all $r_w \in V_{S_1^*}$ and $r_z \in V_{S_2^*}$. Since $d_T(v_p, v_q) = \delta(\ell)$, we then have $c_{\mathcal{M}}(e) \leq 4 \cdot \delta(\ell)$. Hence, the claim of the lemma holds.

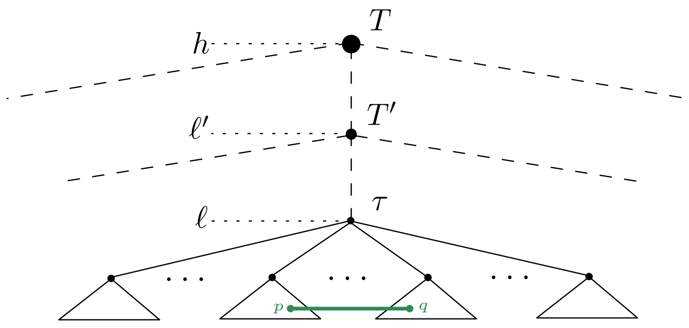


Figure 3.2: The HST T and the edge $\{r_p, r_q\}$. The subtree T' is the highest subtree that includes r_p and r_q while $|t_p - t_q| > 3 \cdot \delta(\ell')$ where T' is rooted at level $\ell' \geq \ell$.

Let us therefore assume that $|t_p - t_q| > 3 \cdot \delta(\ell)$. Let $\ell' \in [\ell, h]$ be the largest level for which $|t_p - t_q| > 3 \cdot \delta(\ell')$ and let T' be the subtree of T that is rooted on level ℓ' and that contains both requests r_p and r_q (see Figure 3.2). Note that this implies that

$$|t_p - t_q| \leq 3 \cdot \delta(\ell' + 1) \quad \text{and thus} \quad c_{\mathcal{M}}(e) \leq 3 \cdot \delta(\ell' + 1) + \delta(\ell). \tag{3.5}$$

We can partition each of the sets $V_{S_1^*}$ and $V_{S_2^*}$ into two sets where one of the sets in each case includes the requests in the subtree T' and the other set includes the requests outside subtree T' (see Figure 3.3). The edge e obviously connects the two components $V_{S_1^*} \cap R(T')$ and $V_{S_2^*} \cap R(T')$ since r_p and r_q are both in $R(T')$. If the edge e is removed then edge e^* is an edge connecting one of the two components $V_{S_1^*} \cap R(T')$ and $V_{S_1^*} \setminus R(T')$ in $V_{S_1^*}$ to one of the two components $V_{S_2^*} \cap R(T')$ and $V_{S_2^*} \setminus R(T')$ in $V_{S_2^*}$. The four different types of such edges are shown by the dashed edges in Figure 3.3.

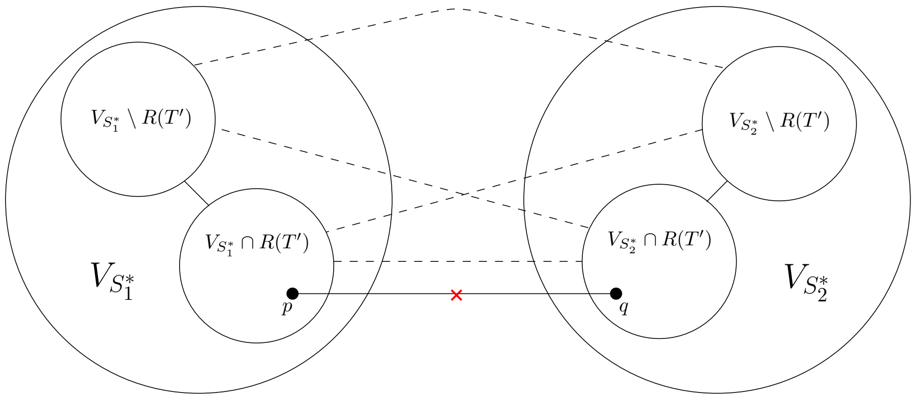


Figure 3.3: The spanning tree S^* when there is a subtree T' that is rooted at height $\ell' \in [\ell, h]$ and is the highest subtree where $|t_p - t_q| > 3 \cdot \delta(\ell')$. If the edge $\{p, q\}$ is removed then the edge e^* could be one of the dashed edges.

Any edge that connects the two components $V_{S_1^*} \setminus R(T')$ and $V_{S_2^*} \cap R(T')$ has length at least $\delta(\ell'+1)$ since $d_T(v_w, v_z) \geq \delta(\ell'+1)$ for all $r_w \in V_{S_1^*} \setminus R(T')$ and $r_z \in V_{S_2^*} \cap R(T')$. By symmetry, the same also holds for the edges that connect the two components $V_{S_1^*} \cap R(T')$ and $V_{S_2^*} \setminus R(T')$. Hence, if e^* is an edge of one of these two types, we have $c_{\mathcal{M}}(e^*) \geq \delta(\ell'+1)$. It then follows directly from (3.5) that $c_{\mathcal{M}}(e) \leq 4 \cdot c_{\mathcal{M}}(e^*)$ and thus the claim of the lemma holds.

Let us therefore move to the case where e^* connects the two components $V_{S_1^*} \setminus R(T')$ and $V_{S_2^*} \setminus R(T')$, i.e., $e^* = \{r_x, r_y\}$ connects to nodes v_x and v_y outside tree T'. Recall that the tree S^* is constructed in a bottom-up way such that the subtree $S^*(T'')$ of S^* is connected for every subtree T'' of T. Hence, removing edge e inside subtree T' does not affect subtrees $S^*(T'')$ for trees T'' that do not contain T'. Therefore if two nodes u and v outside tree T' end up on different sides of the cut $(V_{S_1^*}, V_{S_2^*})$, the least common ancestor of v_x and v_y has to be an ancestor of T' and it is thus at level at least $\ell'+1$. Hence, if e^* connects the two components $V_{S_1^*} \setminus R(T')$ and $V_{S_2^*} \setminus R(T')$, we also have $c_{\mathcal{M}}(e^*) \geq \delta(\ell'+1)$ and therefore again (3.5) implies the claim of the lemma.

It remains to show that all edges that connect the two components $V_{S_1^*} \cap R(T')$ and $V_{S_2^*} \cap R(T')$ are also large enough. W.l.o.g., we assume that $p < q$, i.e., the request r_p is ordered before the request r_q by ARROW. Further w.l.o.g., we assume that the dummy request is in $V_{S_1^*}$.

We next show that $t_q > t_p$. If $p = 0$ then the $t_q \geq t_p$ because $t_p = 0$ and because $|t_p - t_q| > 3 \cdot \delta(\ell) \geq 0$. Otherwise, for the sake of contradiction, let us assume that $t_q \leq t_p$. By the second claim of Lemma 2.1 we have

$$t_p - t_q \leq d_T(v_p, v_q) \leq \delta(\ell).$$

This together with our assumption $t_q \leq t_p$ contradicts the fact that $|t_p - t_q| > 3 \cdot \delta(\ell)$. Therefore, $t_q > t_p$.

Recall that e connects the two requests r_p and r_q inside level-ℓ tree τ. Consider the subtree $S^*(\tau)$ of S^* and let $S_1^*(\tau)$ and $S_2^*(\tau)$ be the two subtrees of $S^*(\tau)$ that are obtained when removing edge e from $S^*(\tau)$. By the construction of the tree S^*, the edge e is one with minimum Manhattan cost among all edges connecting the requests in $S_1^*(\tau)$ and $S_2^*(\tau)$. We know that for all $r_w \in V_{S_1^*(\tau)}$ and $r_z \in V_{S_2^*(\tau)}$ we have $d(v_w, v_z) = \delta(\ell)$. These facts imply that $t_p = t_{\max}(V_{S_1^*(\tau)})$ and $t_q = t_{\min}(V_{S_2^*(\tau)})$.

Now we show that there is an ARROW edge (r_x, r_{x+1}) where $r_x \in V_{S_1^*(\tau)}$ and $r_{x+1} \in V_{S_2^*(\tau)}$. For any two neighbor blocks b_i^ℓ and b_j^ℓ at subtree τ and with $i < j$, we know that

$$t_{\min}(b_j^\ell) - t_{\max}(b_i^\ell) < \delta(\ell + 1)$$

as otherwise because of the split condition (3.2), the subtree τ would have been split. Thus, we have

$$t_{\min}(b_j^\ell) - t_{\max}(b_i^\ell) < 3 \cdot \delta(\ell)$$

since $\delta(\ell+1) \leq 3 \cdot \delta(\ell)$ for $\alpha = 2$. Let $b_{i_1}^\ell, b_{i_2}^\ell, \ldots, b_{i_s}^\ell$ be the level-ℓ blocks of the subtree τ and assume that $i_1 < i_2 < \cdots < i_s$. As $t_q - t_p > 3 \cdot \delta(\ell)$ and because $t_p = t_{\max}(V_{S_1^*(\tau)})$ and $t_q = t_{\min}(V_{S_2^*(\tau)})$, for any two neighbor blocks $b_{i_j}^\ell$ and $b_{i_{j+1}}^\ell$, the requests $r = (v, t)$ from $b_{i_j}^\ell$ with $t = t_{\max}(b_{i_j}^\ell)$ and the requests $r' = (v', t')$ from $b_{i_{j+1}}^\ell$ with $t' = t_{\min}(b_{i_{j+1}}^\ell)$ either all have to be in in $V_{S_1^*(\tau)}$ or they all have to be in $V_{S_2^*(\tau)}$. We show that this implies that there has to be a block $b_{i_j}^\ell$ at tree τ for which the first request is in $V_{S_1^*(\tau)}$ and which contains some request from $V_{S_2^*(\tau)}$. First note that because of Lemma 2.1 and because we assumed that the dummy request is in $V_{S_1^*(\tau)}$, the first request of $b_{i_1}^\ell$ is in $V_{S_1^*(\tau)}$. If all the first requests of blocks $b_{i_j}^\ell$ are in $V_{S_1^*(\tau)}$, it follows from the fact that $V_{S_2^*(\tau)}$ needs to be non-empty that there has to be a block $b_{i_j}^\ell$ for which the first request is in $V_{S_1^*(\tau)}$ and which contains some request from $V_{S_2^*(\tau)}$. Otherwise, assume that $b_{i_j}^\ell$ (for $j \geq 2$) is the first block for which the first request is in $V_{S_2^*(\tau)}$. Because by Lemma 2.1, the first request of a block is always one with smallest issue time, the above observation implies that the request with the largest issue time in $b_{i_{j-1}}^\ell$ is in $V_{S_2^*(\tau)}$ and then $b_{i_{j-1}}^\ell$ there has the first request is in $V_{S_1^*(\tau)}$ and which contains some request from $V_{S_2^*(\tau)}$. In a block, where the first request is from $V_{S_1^*(\tau)}$ and there is some request from $V_{S_2^*(\tau)}$, there also have be two consecutive requests r_x and r_{x+1} (and thus an ARROW edge), such that $r_x \in V_{S_1^*(\tau)}$ and $r_{x+1} \in V_{S_2^*(\tau)}$.

We next show that the ARROW edge (r_x, r_{x+1}) is the only such ARROW edge even with respect to the tree T' containing tree τ. Specifically, we show that for all $r_w \in V_{S_1^*} \cap R(T')$ and all $r_z \in V_{S_2^*} \cap R(T')$ we have $w \leq x$ and $z \geq x+1$. In other words, r_x is the last request ordered in $V_{S_1^*} \cap R(T')$ and r_{x+1} is the first request ordered in $V_{S_2^*} \cap R(T')$. For the sake of contradiction, let us assume that there is a request $r_w \in V_{S_1^*} \cap R(T')$ for which $w > x$ or that there is a request $r_z \in V_{S_2^*} \cap R(T')$ for which $z < x+1$. We first assume the existence of request r_w. Since $(x, x+1)$ is an ARROW edge, we have $w > x+1$ and using the second claim of Lemma 2.1 we get

$$t_{x+1} - t_w \leq d(v_w, v_{x+1}) \leq \delta(\ell').$$

However, we know that $t_q - t_p \leq t_{x+1} - t_w$ and therefore

$$t_q - t_p \leq \delta(\ell').$$

This contradicts the fact that $t_q - t_p > 3 \cdot \delta(\ell')$. Consequently, there does not exist any requests $r_w \in V_{S_1^*} \cap R(T')$ for which $w > x$. Now, let us assume that there is a request $r_z \in V_{S_2^*} \cap R(T')$ for which $z < x+1$. Again since $(x, x+1)$ is an ARROW edge, we have $z < x$ and using the second claim of Lemma 2.1 we get

$$t_z - t_x \leq d(v_z, v_x) \leq \delta(\ell').$$

However, we know that $t_q - t_p \leq t_z - t_x$ and therefore

$$t_q - t_p \leq \delta(\ell').$$

Again, this is a contradiction to the fact that $t_q - t_p > 3 \cdot \delta(\ell')$. Consequently, there does not exist any requests $r_z \in V_{S_2^*} \cap R(T')$ with $z < x+1$.

Finally we show that for all $r_w \in V_{S_1^*} \cap R(T')$ and all $r_z \in V_{S_2^*} \cap R(T')$ the Manhattan cost $c_{\mathcal{M}}(r_w, r_y)$ is at most $3 \cdot \delta(\ell')$. Using the second claim of Lemma 2.1 we have

$$t_{x+1} - t_z \leq d(v_z, v_{x+1}) \leq \delta(\ell'). \tag{3.6}$$

We can similarly bound $t_w - t_x$. If $w = 0$ we have $t_w \leq t_x$ and otherwise, using the second claim of Lemma 2.1 we have

$$t_w - t_x \leq d(v_x, v_w) \leq \delta(\ell'). \tag{3.7}$$

Using (3.6) and (3.7) we then get

$$t_{x+1} - t_x \leq t_z - t_w + 2 \cdot \delta(\ell'). \tag{3.8}$$

We know that the Manhattan cost of (r_x, r_{x+1}) is at least the Manhattan cost of (r_p, r_q) because $t_q - t_p \leq t_{x+1} - t_x$ and because for all $r_f \in V_{S_1^*(\tau)}$ and $r_g \in V_{S_2^*(\tau)}$, we have $d(v_f, v_g) = \delta(\ell)$. That is, we have

$$c_{\mathcal{M}}(r_p, r_q) \leq c_{\mathcal{M}}(r_x, r_{x+1}).$$

Further, because for all $r_f \in V_{S_1^*} \cap R(T')$ and $r_g \in V_{S_2^*} \cap R(T')$, we have $d(v_f, v_g) \geq \delta(\ell)$, by using (3.8), we obtain

$$c_{\mathcal{M}}(r_p, r_q) \leq c_{\mathcal{M}}(r_x, r_{x+1}) \leq c_{\mathcal{M}}(r_w, r_z) + 2 \cdot \delta(\ell'). \tag{3.9}$$

Therefore, by using the facts that $t_q - t_p > 3 \cdot \delta(\ell')$ and $t_q - t_p \leq t_{x+1} - t_x$, and by using (3.8), we get that $t_z - t_w \geq t_{x+1} - t_x - 2\delta(\ell') \geq \delta(\ell')$ and we thus have $c_{\mathcal{M}}(r_w, r_z) \geq \delta(\ell')$. By applying (3.9), we thus get that

$$c_{\mathcal{M}}(r_p, r_q) < 3 \cdot c_{\mathcal{M}}(r_w, r_z).$$

Consequently, also if e^* connects the two components $V_{S_1^*} \cap R(T')$ and $V_{S_2^*} \cap R(T')$, its Manhattan cost is within a factor 3 of the Manhattan cost of e. Hence, the claim of the lemma holds. □

Corollary 3.17. *The total Manhattan cost of the spanning tree S^* is at most 4 times the total Manhattan cost of an MST spanning all the requests.*

Proof. Follows directly from Lemma 3.16 and Theorem 3.26.

□

3.5 Analysis of the Online Queueing Cost

In this section, we give a general framework to compare the queueing cost of an online queueing algorithm on HST T with the bound of the offline queueing cost as established in Section 3.4. At the end of the section, we apply the method to analyze synchronous ARROW executions on T. As in Section 3.4.3, for convenience, we add one more level to the HST T so that each level 0 node v has a child node on level -1 for each of the requests issued at node v. The new leaf nodes are on level -1 and each leaf node receives exactly one request.

We first state two basic locality properties of ARROW and possibly other online queueing algorithms. We will then show that those properties are sufficient to prove a constant competitive ratio compared to the optimal offline queueing cost on T. We define the notion of a *distance-respecting queueing order* and the notion of *distance-respecting latency cost* of a queueing algorithm.

Definition 3.18 (Distance-Respecting Order). *Let R be a set of requests $r_i = (v_i, t_i)$ issued at the nodes of a tree T and let π be permutation on $[0, |R|-1]$. The ordering $r_{\pi(0)}, r_{\pi(1)}, \ldots, r_{\pi(|R|-1)}$ induced by π is called* distance-respecting *if whenever $\pi(i) < \pi(j)$, we have $t_i - t_j \leq d_T(v_i, v_j)$.*

Definition 3.19 (Distance-Respecting Latency Cost). *An online distributed queueing algorithm* ALG *is said to have* distance-respecting latency cost *if for any request set R and any possible queueing order π_{ALG} of* ALG*, for all $1 \leq i < j < |R|$, it holds that*

$$t_{\pi_{\text{ALG}}(i)} + L_{\text{ALG}}(r_{\pi_{\text{ALG}}(i), \pi_{\text{ALG}}(i-1)}) \leq t_{\pi_{\text{ALG}}(j)} + d_T(v_{\pi_{\text{ALG}}(j)}, v_{\pi_{\text{ALG}}(i-1)}).$$

3.5.1 Constructing a Spanning Tree

As the first part of the online queueing cost analysis, we construct a new tree $\mathbb{S}$ that spans all requests in R. It will be shown that the total Manhattan cost of $\mathbb{S}$ asymptotically equals the total Manhattan cost of the tree S^* constructed in the previous section.

We construct a new tree $\mathbb{S}$ on R based on an ordering π of the set of requests. We assume that the ordering of the requests given by π is $r_{\pi(0)}, r_{\pi(1)}, \ldots, r_{\pi(|R|-1)}$. For each index i with $i \in [0, |R|-2]$, we define the *local successor* as

$$next(i) := \min\left\{ j \in [i+1, |R|-1] \,:\, d_T(v_{\pi(i)}, v_{\pi(j)}) = \min_{k \in [i+1, |R|-1]} d_T(v_{\pi(i)}, v_{\pi(k)}) \right\}. \tag{3.10}$$

Hence, among the requests ordered after $r_{\pi(i)}$ by order π, $next(i)$ is the position of a request in the order π with minimum tree distance to $v_{\pi(i)}$ and among those, of the first one ordered by π. Note that this means that for all requests $r_{\pi(k)}$ for which $i < k < next(i)$, we have $d_T(v_{\pi(i)}, v_{\pi(k)}) > d_T(v_{\pi(i)}, v_{\pi(next(i))})$ and for all requests $r_{\pi(k)}$ for which $k \geq next(i)$, we have $d_T(v_{\pi(i)}, v_{\pi(k)}) \geq d_T(v_{\pi(i)}, v_{\pi(next(i))})$.

The spanning tree $\mathbb{S}$ is constructed as follows. For every request $r_{\pi(i)}$ for all $i \in [0, |R|-2]$, we add the edge $\{r_{\pi(i)}, r_{\pi(next(i))}\}$ to the tree $\mathbb{S}$. Note that $\mathbb{S}$ is indeed a spanning tree: If directing each edge from $r_{\pi(i)}$ to $r_{\pi(next(i))}$, each node has out-degree 1 and we cannot have cycles because $next(i) > i$. The following observation shows that in addition, $\mathbb{S}$ has the same useful hierarchical structure as the tree S^* constructed in Section 3.4.3.

Observation 3.20. *As the tree S^*, also the tree $\mathbb{S}$ has the property that for any subtree T' of T, the subgraph of $\mathbb{S}$ induced by only the requests at nodes in T' is a connected subtree of $\mathbb{S}$. This follows directly from the definition of the local successor $r_{\pi(next(i))}$. Except for the last ordered request inside T', the local successor of any other request of T' is inside T' (because the local successor is a request with minimum tree distance).* □

In light of Observation 3.20, for any subtree T' of T, we use $\mathbb{S}(T')$ to denote the subtree of $\mathbb{S}$ induced by the requests issued at nodes in T'.

3.5.2 Bounding the Manhattan Cost of the Spanning Tree

The following lemma shows that if the spanning tree $\mathbb{S}$ is constructed by using a distance-respecting ordering π, the total Manhattan cost of the spanning tree $\mathbb{S}$ is asymptotically equal the total Manhattan cost of S^*.

Lemma 3.21. *Let $C_{\mathcal{M}}(\mathbb{S})$ and $C_{\mathcal{M}}(S^*)$ be the total Manhattan costs of $\mathbb{S}$ and of S^*. If the tree $\mathbb{S}$ is constructed using a distance-respecting ordering π, we have $C_{\mathcal{M}}(\mathbb{S}) \leq 3 \cdot C_{\mathcal{M}}(S^*)$.*

Proof. Consider some subtree τ of T that is rooted at a node on level $\ell \in [0, h]$. Assume that v has m children an that the subtrees of T rooted at the m children are $\tau_1, \tau_2, \ldots, \tau_m$. Using Observation 3.20, we know that $\mathbb{S}(\tau_1), \mathbb{S}(\tau_2), \ldots, \mathbb{S}(\tau_m)$ are subtrees of $\mathbb{S}(\tau)$ trees that are connected to each other with $m - 1$ edges to form the spanning tree $\mathbb{S}(\tau)$. Let us call this set of edges $\mathbb{I}(\tau)$. Note that for $\ell = 0$ the subtrees of τ are single requests at level -1. Similarly, the construction of S^* implies that the spanning tree $S^*(\tau)$ results from connecting the spanning trees $S^*(\tau_1), S^*(\tau_2), \ldots, S^*(\tau_m)$ with $m - 1$ edges. Let $I^*(\tau)$ denote this set of these $m-1$ edges. Recall that the edges in $I^*(\tau)$ are chosen such that they have minimum total Manhattan cost among all sets of m edges connecting the trees $S^*(\tau_1), S^*(\tau_2), \ldots, S^*(\tau_m)$. We also emphasize that for all $i \in [1, m]$, the trees $\mathbb{S}(\tau_i)$ and $S^*(\tau_i)$ consist of the same set of nodes (the requests inside tree τ_i). Let $C_{\mathcal{M}}(\mathbb{I}(\tau))$ and $C_{\mathcal{M}}(I^*(\tau))$ be the total Manhattan costs of the edges in $\mathbb{I}(\tau)$ and $I^*(\tau)$, respectively. To prove the lemma, it suffices to show that

$$\forall \text{ subtree } \tau \text{ of } T\colon C_{\mathcal{M}}(\mathbb{I}(\tau)) \leq 3 \cdot C_{\mathcal{M}}(I^*(\tau)). \tag{3.11}$$

Let $e = (r_{\pi(w)}, r_{\pi(z)}) \in \mathbb{I}(\tau)$ be an arbitrary edge of $\mathbb{I}(\tau)$ and let $\mathbb{S}_1(\tau)$ and $\mathbb{S}_2(\tau)$ be the two subtrees of $\mathbb{S}(\tau)$ resulting from removing e from $\mathbb{I}(\tau)$. Let $V_{\mathbb{S}_1(\tau)}$ and $V_{\mathbb{S}_2(\tau)}$ be the set of nodes (requests) of the trees $\mathbb{S}_1(\tau)$ and $\mathbb{S}_2(\tau)$ and assume, w.l.o.g., that $w < z$ and that $r_{\pi(w)} \in V_{\mathbb{S}_1(\tau)}$ and $r_{\pi(z)} \in V_{\mathbb{S}_2(\tau)}$. Also, consider an edge e^* that crosses the cut $(V_{\mathbb{S}_1(\tau)}, V_{\mathbb{S}_2(\tau)})$ and has minimum Manhattan cost among all edges in $S^*(\tau)$ that cross this cut. Note that because for all i the trees $\mathbb{S}(\tau_i)$ and $S^*(\tau_i)$ consist of the same set of node, node e^* must be from the set $I^*(\tau)$. In order to prove (3.11), it suffices to show that

$$c_{\mathcal{M}}(e) \leq 3 \cdot c_{\mathcal{M}}(e^*). \tag{3.12}$$

Inequality (3.11) then directly follows from Theorem 3.26.

From the definition of local successor, we know that $z = next(w)$. This implies that for all requests $r_{\pi(x)}$ where $w < x < z$, we have $d_T(v_{\pi(w)}, v_{\pi(x)}) > \delta(\ell)$ since $d_T(v_{\pi(w)}, v_{\pi(z)}) = \delta(\ell)$. Therefore, all requests that are ordered between $r_{\pi(w)}$ and $r_{\pi(z)}$ by ARROW are not in $R(\tau)$ (i.e., in the set of requests of tree τ). This means that all requests in $R(\tau)$ are ordered either before $r_{\pi(w)}$ or after $r_{\pi(z)}$ by ARROW. More precisely, the claim is that for all requests $r_{\pi(x)} \in V_{\mathbb{S}_1(\tau)}$ we have $x \leq w$ and for all requests $r_{\pi(x)} \in V_{\mathbb{S}_2(\tau)}$ we have $x \geq z$. To show this, we first observe that by the definition of e, $\mathbb{S}_1(\tau)$ and $\mathbb{S}_2(\tau)$, among all edges of $\mathbb{S}(\tau)$, the edge $e = \left\{r_{\pi(w)}, r_{\pi(z)}\right\}$ is the only edge that crosses the cut $(V_{\mathbb{S}_1(\tau)}, V_{\mathbb{S}_2(\tau)})$.

We now first show that for all requests $r_{\pi(x)} \in V_{\mathbb{S}_2(\tau)}$ we have $x \geq z$. For contradiction, let us assume that there is a request $r_{\pi(x)} \in V_{\mathbb{S}_2(\tau)}$ for which $x < z$ and therefore $x < w$. This implies that there must be a largest $y < w$ such that $r_{\pi(y)} \in V_{\mathbb{S}_2(\tau)}$. Note that because $r_{\pi(y)}$ is not the last request ordered in τ, $r_{\pi(next(y))}$ must be in τ and it therefore must be in $V_{\mathbb{S}_1(\tau)}$. This implies that the edge $\left\{r_{\pi(y)}, r_{\pi(next(y))}\right\}$ of $\mathbb{S}(\tau)$ crosses the cut $(V_{\mathbb{S}_1(\tau}, V_{\mathbb{S}_2(\tau)})$, which is not possible because the edge $\left\{r_{\pi(w)}, r_{\pi(z)}\right\}$ is the only edge of $\mathbb{S}(\tau)$ crossing this cut.

We next show that for all requests $r_{\pi(x)} \in V_{\mathbb{S}_1(\tau)}$, we have $x \leq w$. Again assume that there is a request $r_{\pi(x)} \in V_{\mathbb{S}_1(\tau)}$ such that $x > w$ and thus $x > z$. Therefore, there must be smallest $y > w$ for which $r_{\pi(y)} \in V_{\mathbb{S}_1(\tau)}$. This implies that $r_{\pi(y)}$ is the local successor of some request in $V_{\mathbb{S}_2(\tau)}$. This again contradicts the fact that the edge $e = \left\{r_{\pi(w)}, r_{\pi(z)}\right\}$ is the only edge of $\mathbb{S}(\tau)$ crossing the cut $(V_{\mathbb{S}_1(\tau)}, V_{\mathbb{S}_2(\tau)})$.

Finally we show that for all $r_{\pi(p)} \in V_{\mathbb{S}_1(\tau)}$ and $r_{\pi(q)} \in V_{\mathbb{S}_2(\tau)}$ the Manhattan cost of e is at most $3 \cdot c_{\mathcal{M}}(r_{\pi(p)}, r_{\pi(q)})$. Because π is distance-respecting, we have

$$t_{\pi(z)} - t_{\pi(q)} \leq d_T(v_{\pi(q)}, v_{\pi(z)}) \leq \delta(\ell). \tag{3.13}$$

Further, if $p = 0$, we have $t_{\pi(p)} = 0$ and thus $t_{\pi(p)} \leq t_{\pi(w)}$. Otherwise, because π is distance-respecting, we get

$$t_{\pi(p)} - t_{\pi(w)} \leq d_T(v_{\pi(p)}, v_{\pi(w)}) \leq \delta(\ell). \tag{3.14}$$

Using (3.13) and (3.14) we have

$$t_{\pi(z)} - t_{\pi(w)} \leq t_{\pi(q)} - t_{\pi(p)} + 2 \cdot \delta(\ell). \tag{3.15}$$

We continue by distinguishing the two cases $t_{\pi(z)} \geq t_{\pi(w)}$ and $t_{\pi(w)} > t_{\pi(z)}$. First assume that $t_{\pi(z)} \geq t_{\pi(w)}$. Then, using $d_T(v_{\pi(z)}, v_{\pi(w)}) = d_T(v_{\pi(p)}, v_{\pi(q)}) = \delta(\ell)$ and (3.15) we obtain

$$c_{\mathcal{M}}(r_{\pi(z)}, r_{\pi(w)}) \leq c_{\mathcal{M}}(r_{\pi(p)}, r_{\pi(q)}) + 2 \cdot \delta(\ell).$$

Moreover, because $d_T(v_{\pi(p)}, v_{\pi(q)}) = \delta(\ell)$, we know that $\delta(\ell) \leq c_{\mathcal{M}}(r_{\pi(p)}, r_{\pi(q)})$. Thus,

$$c_{\mathcal{M}}(e) \leq 3 \cdot c_{\mathcal{M}}(r_{\pi(p)}, r_{\pi(q)}).$$

Let us therefore consider the second case where $t_{\pi(w)} > t_{\pi(z)}$. It is clear that $w \neq 0$ as otherwise $t_{\pi(w)} = 0$ and thus $t_{\pi(z)} \geq t_{\pi(w)}$. Because π is distance-respecting, we have

$$t_{\pi(w)} - t_{\pi(z)} \leq d_T(v_{\pi(w)}, v_{\pi(z)}) = \delta(\ell).$$

Using the assumption that $t_{\pi(w)} > t_{\pi(z)}$, we then have

$$c_{\mathcal{M}}(r_{\pi(z)}, r_{\pi(w)}) = |t_{\pi(w)} - t_{\pi(z)}| + d_T(v_{\pi(w)}, v_{\pi(z)}) = t_{\pi(w)} - t_{\pi(z)} + \delta(\ell) \leq 2 \cdot \delta(\ell).$$

Finally, we can again use that $c_{\mathcal{M}}(r_{\pi(p)}, r_{\pi(q)}) \geq d_T(v_{\pi(p)}, v_{\pi(q)}) = \delta(\ell)$ and thus get that

$$c_{\mathcal{M}}(e) \leq 2 \cdot c_{\mathcal{M}}(r_{\pi(p)}, r_{\pi(q)}).$$

This concludes the proof of the lemma. □

3.5.3 Bounding the Total Latency Cost

It remains to prove the main claim and show that the total online queueing cost on the HST T is within a constant factor of the optimal offline cost on T. The following theorem states that this is generally true for algorithms with distance-respecting latency cost (Definition 3.19) and which produce distance-respecting queueing orders (Definition 3.18), as long as the request set R is condensed (Definition 3.5).

Theorem 3.22. *Assume that we are given an HST T and a condensed set of requests issued at the leaves of R. Further, assume that we are given a distributed queueing algorithm* ALG *that has distance-respecting latency cost and that always produces a distance-respecting queueing order π. Then, the total latency cost of* ALG *is within a constant factor of the optimal offline cost on T.*

Proof. Because the request set R is condensed, Lemma 3.6 implies that the optimal offline cost is within a constant factor of the Manhattan cost of an optimal TSP path connecting all the requests. The optimal offline cost therefore also is within a constant factor of the total Manhattan cost of an MST of the request set. Hence, Corollary 3.17 implies that also the total Manhattan cost of S^* is within a constant factor of the cost of an optimal offline solution on T. Because the ordering π generated by ALG is distance-respecting, by Lemma 3.21, the same is true for the total Manhattan cost $C_{\mathcal{M}}(\mathbb{S})$ of the tree $\mathbb{S}$. It therefore remains to show that $cost^T_{\text{ALG}}(\pi) = O(C_{\mathcal{M}}(\mathbb{S}))$.

Because ALG has distance-respecting latency cost, for all $i \in [0, |R| - 2]$, we have

$$t_{\pi(i+1)} + L^T_{\text{ALG}}(r_{\pi(i)}, r_{\pi(i+1)}) \leq t_{\pi(next(i))} + d_T(v_{\pi(i)}, v_{\pi(next(i))}).$$

Note that we have $next(i) \geq i + 1$. Subtracting $t_{\pi(i)}$ on both sides yields

$$t_{\pi(i+1)} - t_{\pi(i)} + L^T_{\text{ALG}}(r_{\pi(i)}, r_{\pi(i+1)}) \leq t_{\pi(next(i))} - t_{\pi(i)} + d_T(v_{\pi(i)}, v_{\pi(next(i))}).$$

If we sum up the above inequality for all $i \in [0, |R| - 2]$, we get

$$\sum_{i=0}^{|R|-2} \big(t_{\pi(i+1)} - t_{\pi(i)} + d_T(v_{\pi(i)}, v_{\pi(i+1)})\big) \leq \sum_{i=0}^{|R|-2} \big(t_{\pi(next(i))} - t_{\pi(i)} + d_T(v_{\pi(i)}, v_{\pi(next(i))})\big)$$

The sum of the latencies on the left-hand side exactly equals the total queueing cost $cost_{\text{ALG}}^{T}(\pi)$ of ALG. To bound the right-hand side, note that $t_{\pi(next(i))} - t_{\pi(i)} + d_T(v_{\pi(i)}, v_{\pi(next(i))}) \leq c_{\mathcal{M}}(r_{\pi(i)}, r_{\pi(next(i))})$. Together, we get

$$t_{\pi(|R|-1)} - t_{\pi(0)} + cost_{\text{ALG}}^{T}(\pi) \leq C_{\mathcal{M}}(\mathbb{S}).$$

As specified in Section 2.1, we assume that $t \geq 0$ for every request $r = (v, t)$ and that every queueing algorithm first has to order the dummy request $r_0 = (v_0, 0)$. We therefore have $t_{\pi(|R|-1)} \geq 0$ and $t_{\pi(0)} = t_0 = 0$, which completes the proof of the theorem. □

Corollary 3.23. *The total latency cost of a synchronous execution of* ARROW *on an HST T is within a constant factor of the optimal offline queueing cost on T.*

Proof. First note that by Lemma 3.7, w.l.o.g., for synchronous ARROW executions, we can assume that the request set R is condensed. The corollary therefore follows from Theorem 3.22 if we show that synchronous ARROW's ordering is distance-respecting and that synchronous ARROW has distance-respecting latency cost. The former follows from the second claim of Lemma 2.1, the latter follows from the first claim of Lemma 2.1 and the fact that the latency cost of synchronous ARROW for ordering a request r_i as the predecessor of request r_{i+1} is exactly $d_T(v_i, v_{i+1})$. □

Remark 3.24. *The above corollary proves Theorem 3.1 for synchronous executions on the HST T. The full statement of Theorem 3.1 for general asynchronous executions is proven in Section 3.6. There, it is shown that also for asynchronous executions,* ARROW *has distance-respecting latency cost and produces distance-respecting queueing orders. In addition, we also show that we can still restrict attention to condensed request sets. The claim of Theorem 3.1 for the asynchronous case then follows from Theorem 3.22 in the same way as in the above corollary.*

3.6 Queueing Cost in the Asynchronous Model

In this section, we show that the generic analysis of Section 3.5 also applies to asynchronous executions of the ARROW algorithm on T. In order to use the framework of Section 3.5 in the asynchronous setting, we mostly importantly need to show that ARROW has distance-respecting latency cost (Definition 3.19) and that it generates distance-respecting queueing orders (Definition 3.18) also in the asynchronous case. To show this, we need asynchronous variants of the basic Lemma 3.7 and the Lemma 2.4.[3] In addition, we also need to generalize Lemma 3.7 to show that also in the asynchronous setting, w.l.o.g., we can assume that the given request set R is condensed (Definition 3.5).

As in Section 3.4, we relabel the requests for convenience. Throughout the section, we assume that an asynchronous execution of ARROW is given and we label the requests according

[3]The greedy nature of ARROW when the communication is asynchronous has been formally provided in Section 2.3.2.

the order. That is, r_0 is the dummy request and for every $i \geq 1$, r_i is the i^{th} non-dummy request ordered by the asynchronous ARROW execution.

In the following we adapt the basic Lemma 3.7 to the asynchronous setting.

Lemma 3.25. *Let R be a set of queueing requests issued on a tree T and let $r_i = (v_i, t_i)$ and $r_j = (v_j, t_j)$ be two requests of R that are consecutive w.r.t. time of occurrence. Further, choose two requests $r_a = (v_a, t_a)$ with $t_a \leq t_i$ and $r_b = (v_b, t_b)$ with $t_b \geq t_j$ minimizing $\delta := t_b - t_a - d_T(v_a, v_b)$. If $\delta > 0$, every request $r = (v, t)$ with $t \geq t_j$ can be replaced by a request $r' = (v, t - \delta)$ without decreasing the worst-case cost of* ARROW *and without increasing the optimal offline cost.*

Proof. Because the optimal offline cost is computed w.r.t. synchronous executions, the proof that the optimal offline cost is not increased follows directly from Lemma 3.7. To show that the worst-case ARROW cost does not decrease, we show that if all the message delays remain the same, the execution can still produce the same ARROW order with the same total cost.

Let $R_{\leq}$ be the set of requests with issue time $\leq t_i$ and let $R_{\geq}$ be the set of requests with issue time $\geq t_j$. Note that $R = R_{\leq} \cup R_{\geq}$. We first show that when replacing every request $r = (v, t)$ in $R_{\geq}$ by a request $r' = (v, t - \delta + \varepsilon)$ for an arbitrary $\varepsilon > 0$, if we do not change any of the message delays, we obtain exactly the same ARROW ordering and cost.[4] To see this, first observe that in this case, the first claim of Lemma 2.4 implies that all requests in $R_{\leq}$ are ordered before any request in $R_{\geq}$ is ordered. Let $r_x = (v_x, t_x)$ be the last request ordered in $R_{\leq}$ and let $r_y = (v_y, t_y)$ be the first request ordered in $R_{\geq}$ in the original execution. Because all requests in $R_{\geq}$ are shifted by the same amount and they are still all ordered after the requests in $R_{\leq}$, also after the shifting, the "find predecessor" request of r_y is the first one to arrive at node v_x and therefore r_y still is the successor of r_x. Because the time differences inside $R_{\geq}$ do not change, also the rest of the ordering does not change. The argument holds even if we let ε go arbitrarily close to 0. In the limit, the argument therefore still holds as long as whenever a node receives several messages at the same time, the asynchronous scheduler processes messages corresponding to requests in $R_{\leq}$ before processing messages corresponding to $R_{\geq}$. We have therefore shown that for every initial ARROW execution, the asynchronous scheduler can enforce an equivalent execution with the same cost with the shifted request. This proves the claim of the lemma. □

Considering the formal discussion provided in Section 2.3.2, we now have everything needed to prove Theorem 3.1 stating that the total cost of an asynchronous execution of ARROW on an HST T is within a constant factor of the optimal offline queueing cost on T.

Proof of Theorem 3.1. The above Lemma 3.25 shows that we can (iteratively) transform the initial request set R into a condensed set of requests without decreasing the cost of ARROW and without increasing the optimal offline cost. We can therefore assume that we are given a

[4] A bit more precisely, the asynchronous scheduler has to generate the same message delays and whenever several messages arrive at some node at exactly the same time, the scheduler needs to process them in the same order.

condensed set of requests. The claim of the theorem now follows if we can show that the latency cost of asynchronous ARROW is distance-respecting and that any asynchronous ARROW execution generates a distance-respecting queueing order. However, these statements follow directly from the second and the first claims of Lemma 2.4, respectively. □

3.7 Minimum Spanning Tree Approximation

In this section, we prove a general minimum spanning tree (MST) approximation result. Assume that we are given a spanning tree $\tau = (V, E_\tau)$ of a graph $G = (V, E)$. Together with τ, every edge $e \in E_\tau$ induced a cut of G as follows. When removing e from τ, we obtain a spanning forest consisting of two connected subtrees of τ. Let S and $V \setminus S$ be the node sets of these two connected components. We say that $(S, V \setminus S)$ is the *cut induced by removing* e *from* τ. The next theorem shows that if for every edge $e \in E_\tau$, the weight of e is within a factor λ of the weight of the lightest edge crossing the cut induced by removing e from τ, then the total weight of τ is within a factor λ of the weight of an MST. We expect that this results is already known, however, we have not found a proof of it in the literature. The next theorem proves a slightly more general statement.

Theorem 3.26. *Let $\lambda \geq 1$ be some number and let $G = (V, E, w)$ be a weighted connected graph with non-negative edge weights $w(e) \geq 0$ and let $\tau \subseteq E$ and $\tau^* \subseteq E$ be two arbitrary spanning trees of G. If for every edge e of τ, the lightest edge e' of τ^* crossing the cut induced by removing e from τ has weight $w(e') \geq w(e)/\lambda$, then the total weight of all edges in τ is at most a λ-factor larger than the total weight of the edges in τ^*.*

Proof. In the following, we slightly abuse notation and we identify a spanning tree τ with the set of edges contained in τ. For an edge set $F \subseteq E$, we also use $w(F)$ to denote the total weight of the edges in F. We prove the stronger statement that

$$w(\tau \setminus \tau^*) \leq \lambda \cdot w(\tau^* \setminus \tau). \tag{3.16}$$

We show (3.16) by induction on $|\tau \setminus \tau^*| = |\tau^* \setminus \tau|$. First note that if $|\tau \setminus \tau^*| = 0$, we have $\tau = \tau^*$ and thus (3.16) is clearly true. Further, if $|\tau \setminus \tau^*| = 1$, there is exactly one edge $e \in \tau \setminus \tau^*$ and exactly one edge $f \in \tau^* \setminus \tau$. Because τ and τ^* are spanning trees, f connects the two sides of the cut $(V_{e,1}, V_{e,2})$ induced by removing e from τ and we therefore have $w(e) \leq \lambda \cdot w(f)$, implying (3.16).

Let us therefore assume that $|\tau \setminus \tau^*| = k \geq 2$ and let e be a maximum weight edge of $\tau \setminus \tau^*$. Let $(V_{e,1}, V_{e,2})$ be the cut induced by removing e from τ. Further, let τ' be a spanning tree of G that is obtained by removing e from τ and by adding some edge $f \in \tau^* \setminus \tau$ that connects $V_{e,1}$ and $V_{e,2}$. Note that by the assumptions of the theorem, we have $w(e) \leq \lambda \cdot w(f)$. To prove (3.16), it thus suffices to show that $w(\tau' \setminus \tau^*) \leq \lambda \cdot w(\tau^* \setminus \tau')$. We have $|\tau' \setminus \tau^*| = k - 1$ and thus, if the spanning tree τ' satisfies the conditions of the theorem, $w(\tau' \setminus \tau^*) \leq \lambda \cdot w(\tau^* \setminus \tau')$ and (3.16) follows from the induction hypothesis. We therefore need to show that τ' satisfies the conditions of the theorem.

Consider an arbitrary edge $e' \in \tau' \setminus \tau^*$ and let (U_1, U_2) be the partition of V induced by removing e' from tree τ'. Since e' is an edge of one of the two subtrees of τ resulting after removing e, e' either connects two nodes in $V_{e,1}$ or two nodes in $V_{e,2}$. W.l.o.g., assume that e' connects two nodes in $V_{e,2}$ and let $V_{e,2,1}$ and $V_{e,2,2}$ be the partition of $V_{e,2}$ induced by removing e' from the subtree of τ induced by $V_{e,2}$. We need to show that for every edge $f' \in \tau^*$ connecting U_1 and U_2, it holds that $w(e') \leq \lambda \cdot w(f')$. Any edge f' crossing the cut has to either connect $V_{e,1}$ with $V_{e,2}$ or it has to connect $V_{e,2,1}$ with $V_{e,2,2}$. In the first case, we have $w(e') \leq w(e) \leq \lambda \cdot w(f')$ (recall that we chose e to be the heaviest edge from $\tau \setminus \tau^*$). In the second case, f' also crosses the cut induced by removing e' from the original tree τ and therefore we also have $w(e') \leq \lambda \cdot w(f')$. This concludes the proof. □

3.8 Chapter Notes

To conclude the first part of this thesis, we discuss some open problems. In Section 1.5, we have discussed three problems, i.e., the distributed queuing, the OSD, and the online TSP problems that consider the dynamic requests. The distributed queuing problem has been studied in this chapter. Based on this study, studying the distributed version of the OSD problem on HSTs is an interesting open problem. Note that the OSD problem with $k \geq 1$ servers is a generalization of a variant of the classic k-server problem, where the requests can arrive at any time rather than sequentially [15]. Therefore, the distributed version of OSD can be seen as the distributed variant of the k-server problem [18], where the requests arrive dynamically and possibly concurrently. In [15], the authors left it as an open question whether there is an online algorithm that provides a constant competitive ratio for the OSD problem in the centralized setting. The resemblances between the OSD and the distributed queuing problems[5] together with the constant competitive ratio provided in this chapter for the distributed queuing problem on HSTs build up our hopes that it is maybe possible to achieve a constant competitive ratio for the distributed OSD problem if the problem is restricted to HSTs.

In the instance of the distributed queuing problem that has been studied in this thesis, we do not take into account the time that is required by each processor of the network to process the "find predecessor" messages. In particular, if several messages arrive at some node of the network at time t, all of them can be handled at the same time. From a practical point of view, it would certainly be interesting to study the distributed queuing problem in a model with *congestion*, where the delay for the processing the messages are taken into account.

Finally, in Section 3.5, we have introduced a general framework to analyze the queueing cost of distributed queueing algorithms on an HST T. In order to use the framework of Section 3.5 for some other distributed queuing algorithm, we mainly need to show that the algorithm has distance-respecting latency cost (Definition 3.19) and that it generates distance-respecting queueing order (Definition 3.18). The question is whether one can apply Theorem 3.22 to some other distributed queuing algorithms specially those that are run on top of the overlay networks such as RELAY [71] and COMBINE [7] (refer to Section 2.5 for further

[5]As discussed in Section 1.5, the main difference between the OSD and the distributed queuing problems is the way in which their cost functions are defined.

information about these algorithms).

Part II

Centralized Service for Sequential Requests

Chapter 4

Online Mobile Facility Location Problem: A Lower Bound

4.1 Introduction

The mobile facility location (MFL) problem has been introduced by Demaine et al. in [28] and studied by Friggstad and Salavatipour in [39] as a variant of facility location problem. The motivation behind the MFL problem is to model scenarios where it potentially makes sense to move facilities to regions where there are a lot of demands in order to reduce the overall cost to serve the demands. We note that the classic facility location problem can be interpreted as a problem, where there is a fixed set of facilities (possibly with an opening cost) and a set of movable demands. The goal is to move each demand to the location of some (open) facility such that the total movement cost (plus potentially the total facility opening cost) is minimized. In this context, the MFL problem is a variant of the classic facility location problem, where also the facilities are movable. In this case, by adding an fixed additive term c_i to all distances of a facility i, one can even capture the cost of opening facility i as a part of the total movement cost of i. We refer the reader to [28, 39] for some real-world applications of MFL in logistics and networks.

Formally in MFL problem, we are given a metric space (V, d), where V is the point set and $d : V \times V \to \mathbb{R}_+$ is the distance function. We are further given k mobile facilities initially located at $F = \{f_1, \dots, f_k\} \subseteq V$, and a demand set $D \subseteq V$. As (V, d) is a metric space, it is assumed that the distance function is symmetric and satisfies the triangle inequality. The goal is to seek a destination f_j^* for each facility j so that the total cost of moving each facility j from f_j to f_j^* and each demand $v \in D$ to the nearest facility destination is minimized. More precisely, the goal is to find a set of k destinations $F^* = \{f_1^*, \dots, f_k^*\}$ that minimizes $\sum_{j=1}^{k} d(f_j, f_j^*) + \sum_{v \in D} d(v, F^*)$ where for any set of points $S \subseteq V$ and any point v, $d(v, S) := \min_{p \in S} d(v, p)$.

Many applications of the MFL problem inherently have an online nature. For example, think of a distributed service that is offered on a large network such as the Internet. To offer the service, a provider might have a budget to place k servers in the network. The best placement of servers depends on the distribution of the users of the distributed service. As the set of users might grow (or even change arbitrarily) over time, from time to time, we might have to move some of the servers, even though migrating a whole server might be a relatively costly thing to do. We would like to minimize the total movement cost of the servers and the assignment cost (or the movement cost) of users to the servers. Note that since the assignment between the users and servers can change over time, the *current* assignment cost is taken into account. Hence, the assignment cost is a function of the current configuration and not cumulative.

Organization of the Chapter: In the rest of this chapter, we provide a formal definition of the online mobile facility location (OMFL) problem in Section 4.2. We also provide some related work in this section. In Section 4.3, a lower bound for OMFL provided. Since the same lower bound holds for a variant of OMFL that is introduced in Chapter 5, the analysis for this lower bound will be provided there. However, an outline of the analysis is provided in Section 4.3. In Section 4.4, an approach that could solve the OMFL problem on general metrics is discussed. This approach is the main motivation behind the problem that is defined and studied in Chapter 5. Finally, in Section 4.5 another online variant of OMFL is discussed.

4.2 Online MFL Problem

An instance of the online MFL (OMFL) problem is defined as follows. We are given an arbitrary finite metric space $\mathcal{M} = (V, d)$ with $|V| = n$ points and a non-negative, symmetric distance function $d : V \times V \to \mathbb{R}_+$, which satisfies the triangle inequality. There are demands/requests $1, 2, \ldots$ that are adversarially issued at points one at a time and there is a set of k mobile facilities/servers. We define a configuration (at time $t = 1, 2, \ldots$) to be a function $\mathcal{S}(t) : [k] \to V$ that specifies which of the servers $\{1, \ldots, k\}$ is located at which point at time t, i.e., $\mathcal{S}_j(t)$ denotes the point that hosts server j at time t.[1] At each time $t = 1, 2, 3, \ldots$ there is exactly one request placed at some point in an adversarial manner and $\mathcal{J}(t)$ denotes the point at which the single request at time t is placed.

Formally, "serving all requests" at time t is a function $\mathcal{I} : [t] \to [k]$, which assigns any request $i \in [t]$ to one of the k servers. We assume there is no bound on the number of requests that can be assigned to the same server. If request i is assigned to server j at time t then this induces the cost $d(\mathcal{J}(i), \mathcal{S}_j(t))$, denoted as *service cost* of request i at time t. The *overall service cost* of a configuration $F \subset V$ at time t denoted by $S_t(F)$ is the sum of the service costs of all requests at this time, i.e., $\sum_{i=1}^{t} d\left(\mathcal{J}(i), \mathcal{S}_{\mathcal{I}_i(t)}(t)\right)$. Hence, the *optimal service cost* at time t denoted by S_t^* equals $\min_F S_t(F)$. Note that the overall service cost of an optimal offline algorithm at time t does not necessarily equal the optimal service cost at time t. In order to keep the overall service cost small, an algorithm can move the servers between the points, which implies some additional cost, called the *movement cost*. The movement cost equals the distance between the points, i.e., if server j is at point v at time t and at point v' at time $t + 1$, the cost of movement of server j at time $t + 1$ equals the distance $d(v, v')$.

[1]We remark that for two integers $a \leq b$, $[a, b] := \{a, \ldots, b\}$ and for an integer $a \geq 1$, we use $[a]$ as a short form to denote $[a] := [1, a]$.

The total cost of algorithm ALG denoted by $\mathcal{A}$ at time t is the sum of the total movement cost by time t and the overall service cost at time t and defined as follows

$$cost_t^{\mathcal{A}} := S_t^{\mathcal{A}} + M_t^{\mathcal{A}} \tag{4.1}$$

where

$$S_t^{\mathcal{A}} := \sum_{i=1}^{t} d\left(\mathcal{J}(i), \mathcal{S}^{\mathcal{A}}_{\mathcal{I}_i^{\mathcal{A}}(t)}(t)\right)$$

and

$$M_t^{\mathcal{A}} := \sum_{i=1}^{t}\sum_{j=1}^{k} d\left(\mathcal{S}_j^{\mathcal{A}}(i-1), \mathcal{S}_j^{\mathcal{A}}(i)\right).$$

The competitive ratio is then defined as the total cost of ALG divided by the optimal MFL solution cost. An optimal algorithm for MFL can wait until all requests arrive and just perform all the necessary server movements at the very end.

4.2.1 Further Related Work

The MFL problem in general metrics was introduced in [28, 39] as a movement problem. It can be seen as a generalization of the standard k-median and facility location problems [39]. The k-median and facility location problems have been widely studied in both operations research and computer science [6, 22, 24, 33, 42, 51]. In [39], the MFL problem is modeled in such a way that the algorithm moves each facility and client to a point where in the final configuration, each client is at a node with some facility. The goal is to minimize the total movement cost of facilities and clients. The movement cost between the clients and the final configuration points could be interpreted as a service cost.

There exist various natural models in which the locations of requests are not known in advance, and a solution must be built or maintained gradually over time without any knowledge about future requests. The first algorithm for online facility location was introduced in [57]. For a broad discussion of models and results on online facility location problem, we refer to the survey in [38]. Another online type of the problem, known as incremental facility location (and also k-median), is studied in [36]. In this variant of the problem, it is possible to close open facilities and assign all their demands to another open facility. Incremental algorithms were introduced in the context of clustering applications where merging clusters is quite natural [25]. A relaxed variant of incremental facility location is presented in [32] (also see [37]). In this variant, the facilities can also be moved to reduce some cost. However unlike in the OMFL problem that was defined in Section 4.2, in [32], moving a facility is for free. For a broader and deeper discussion of models and results on online and incremental facility location problems, we refer to the detailed survey of Fotakis [38].

4.3 A Lower Bound for OMFL

In this section, we provide a lower bound for the OMFL problem. Because the formal proof of the theorem requires some of the tools developed in the next chapter, the formal proof of the following theorem is deferred to Chapter 5.

Theorem 4.1. *Consider any deterministic online algorithm $\mathcal{A}$ and assume that $\mathcal{O}$ is an optimal offline algorithm. Further assume that $\mathcal{A}$ guarantees that at all times $t > 0$, the additive difference between the service cost of $\mathcal{A}$ and the optimal service cost at time t is less than β. Then, there exist an execution and a time $t_0 > 0$ such that the total cost of $\mathcal{A}$ can be lower bounded as follows.*

- *If $\beta = O(k/\varepsilon)$ for any $\varepsilon > 0$, it holds that*

$$cost_{t_0}^{\mathcal{A}} \geq (1+\varepsilon) \cdot cost_{t_0}^{\mathcal{O}} + \Omega(k \log k - \varepsilon k).$$

- *If $\beta = O\left(\frac{k \cdot \log k}{\log \log k}\right)$ and for every $\varepsilon \geq \frac{\log \log k}{\log^{1-\delta} k}$ for any constant $0 < \delta \leq 1$, we obtain*

$$cost_{t_0}^{\mathcal{A}} \geq (1+\varepsilon) \cdot cost_{t_0}^{\mathcal{O}} + \Omega\left(\frac{k \cdot \log k}{\log \log k} - \varepsilon k\right).$$

The lower bound theorem essentially says that if we want to guarantee a small multiplicative competitive ratio, we have to tolerate a relatively large additive term. Basically, if we want to keep the additive term within a bound of β, the multiplicative competitive ratio becomes at least $1 + \Omega(k/\beta)$ (as long as β is not too small).

4.3.1 Outline of the Lower Bound Proof

We consider a metric space with uniform distances, i.e., by scaling appropriately, we can assume that the distance between each pair of points is equal to 1. Assume that we are given an algorithm ALG denoted by $\mathcal{A}$ which guarantees that

$$S_t^{\mathcal{A}} < S_t^* + \beta \tag{4.2}$$

at all times t for some parameter β. In the following, let OPT be any optimal offline algorithm denoted by $\mathcal{O}$. We essentially compute the total cost of ALG and OPT at any time t as functions of the optimal service cost at time t. Given ALG, we construct an execution in which ALG has to perform a large number of movements while the optimal service cost does not grow too much. We divide time into phases such that in each phase, ALG has to move $\Omega(k)$ servers and the optimal service cost grows as slowly as possible. For p phases, we define a sequence of integers $k/3 \geq n_1 \geq n_2 \geq \dots n_p \geq 1$ and values $\Gamma_1 < \Gamma_2 < \cdots < \Gamma_p$. In the following, let v be a free node if v does not have a server. Roughly, at the beginning of a phase i, we choose a set N_i of n_i (ideally) free nodes and make sure that all these nodes have Γ_i requests. Note that constructing an execution means to determine where to add the request in each iteration. The value Γ_i is chosen large enough such that throughout phase i a service cost of $n_i\Gamma_i$ is sufficiently large to force an algorithm to move. Hence, whenever there are n_i free nodes with Γ_i requests, ALG has to move at least one server to one of these nodes. For each such movement, we pick another free node that currently has less than Γ_i requests and make sure it has Γ_i requests. We proceed until there are k nodes with Γ_i requests at which point the main part of the phase ends. Except for the nodes in N_i, each of the k nodes with Γ_i requests leads to a movement of ALG and therefore, ALG has to move at least $k - n_i = \Omega(k)$ servers in phase i. At the end of phase i, we can guarantee that there are exactly k nodes with Γ_i requests, n_i nodes with Γ_{i-1} requests, $n_{i-1} - n_i$ nodes with Γ_{i-2} requests, etc. The optimal service cost after phase p, therefore, is $n_p\Gamma_{p-1} + \sum_{i=3}^{p}(n_{i-1} - n_i)\Gamma_{i-2}$. The service cost paid by ALG at time t can not be smaller than S_t^*.

By contrast, the optimal offline algorithm can wait until all requests have arrived and just perform all the necessary server movements at the very end to have an optimal configuration. Therefore by the end of phase p, OPT has to pay at most k as the total movement cost, while ALG has to pay $\Theta(pk)$ for the movement cost in total by this time. The service cost of OPT equals the optimal service cost at the end of phase p. By choosing the values n_i appropriately, we obtain the claimed bounds.

4.4 Towards OMFL on General Metrics

It seems hard to deal with the OMFL problem on a general metric. Hence, as the first step towards solving the OMFL problem on general metrics, we study the problem on a uniform metric space (i.e., on a metric space, where all pairwise distances are equal). However, we consider a generalized version of the OMFL problem (which we call the *generalized uniform OMFL* problem), where each point of the metric space can potentially host several servers/facilities. We assume that if a node v has some servers/facilities, all requests/demands at v are served by these servers. Depending on the number of servers and the number of requests at a node $v \in V$, an algorithm has to pay some service cost to serve the requests located at v. This service cost of node v is defined by a *service cost function* σ_v such that $\sigma_v(x, y) \geq 0$ is the cost for serving y requests if there are x servers at node v. The service cost function σ is guaranteed to satisfy a number of natural properties at any time. First of all, for every $v \in V$, $\sigma_v(x, y)$ has to be monotonically decreasing in the number of servers x that are placed at node v and monotonically increasing in the number of requests y at v. Further, the effect of adding additional servers to a node v should become smaller with the number of servers and it should not decrease if the number of requests gets larger. The concrete definition of the generalized uniform OMFL problem will be provided in Chapter 5.

As described in Section 1.4.6, utilizing HSTs as a tool in solving the online allocation problems is a common approach [15, 16, 26]. Due to the special structure of HSTs, the task of solving problems on top of them can sometimes be reduced to the task of solving a more general (and thus harder) problem, but on a uniform metric. An interesting and important application of this approach is the k-server problem that has been studied on HSTs [16, 26]. In [16], it is shown that exploiting the randomized low-stretch hierarchical tree decomposition of [34], it is possible to obtain a poly-logarithmic competitive ratio for the k-server problem.

Combined with the general cost functions as described above, a similar approach as [16] could also work for the OMFL problem. The online algorithm for the generalized uniform OMFL that will be described in Chapter 5 can potentially be used as a building block to devise an online algorithm which recursively solves the OMFL on an HST. Roughly speaking, the idea would be that each internal node of the HST runs an instance of the generalized uniform OMFL that determines how to distribute the available servers among its children nodes. Starting from the root, which has k servers, the recursive calls to the generalized uniform OMFL determine the number of servers at each leaf of the HST, giving a feasible OMFL solution.

4.5 Chapter Notes

We have mentioned in Section 4.2.1 that the MFL problem in general metrics has been introduced as a movement problem. In the instance of OMFL that is defined in Section 4.2, the assignment/service costs of requests are not cumulative. Hence, another interesting version of OMFL that is closer to the movement problem introduced by [28] can be defined as follows: initially we are given a collection of servers with their starting locations. One by one, requests (along with, perhaps, new locations in the metric) are added. After each request arrives, we make some changes to the requests/server locations to ensure we have a feasible solution before the next request arrives. That is, when a new request arrives we may reassign server and request locations (including the new request). The cost of this change is the total distance travelled by all requests and servers.

The overall cost (after all requests have arrived) is the total cost of all moves made by all servers and requests. The competitive ratio could then be defined as the total cost of all movements divided by the optimal MFL solution cost.

Chapter 5

Minimizing Combined Cost in Generalized Uniform OMFL: A Tight Analysis

5.1 Generalized OMFL with Uniform Distances

In Section 4.4, we discussed that it is easier to study the OMFL problem on HSTs rather than the general metrics. As described in Section 1.4.6, providing a γ competitive ratio for the OMFL problem on $O(1)$-HSTs yields an expected competitive ratio of $O(\gamma \cdot \log n)$ for the OMFL problem on the general metrics. As mentioned before, the task of solving problems on top of HSTs can sometimes be reduced to the task of solving a more general and thus harder problem, but on a uniform metric. In the following, we define a generalized version of OMFL problem in metrics with uniform distances.

There is a set V of n nodes and there are k mobile servers, where each server has to be placed at one of the nodes. Further, there are requests that arrive at the nodes in an online fashion each at a time. We assume that any node can potentially host an arbitrary number of servers. Formally, the cost for serving the requests at each node v, which is called *service cost* of node v, is given by a general cost function that depends on v, on the number of requests at node v, as well as on the number of servers placed at v. Generally, the more requests there are at some node, the more it costs to serve these requests. Further, if we place more servers at a given node, the cost for serving the requests at this node becomes smaller (formally defined in Section 5.2.1).[1] As the requests arrive one by one, we study online algorithms that possibly have to react whenever a new request arrives. We assume that the servers are movable and the objective of an algorithm is to move around some of the servers in order to keep the *overall service cost* (i.e., the sum of the service costs of all nodes in V) low. As moving a server is an expensive operation, we need to keep the number of movements as small as possible. An online algorithm has to be competitive with an optimal offline algorithm that can wait until all

[1]The most basic cost function would incur a service cost of x whenever x requests are at a node with no server and a service cost of 0 for all requests at nodes with at least one server.

requests have arrived and just perform all the necessary server movements at the very end.

The total cost of an algorithm at time t is the sum of the total number of movements up to time t and the overall service cost at time t (shown by $cost_t^{\mathcal{A}}$ for a given algorithm ALG). In Section 5.3 we give a simple deterministic online algorithm that guarantees to bound the number of movements while keeping the overall service cost close to optimal at all times. More specifically, we fix two parameters $\alpha \geq 1$ and $\beta \geq 0$ and we require that given α and β, an algorithm has to guarantee that at all times, the overall service cost of the online algorithm is within a multiplicative factor α and an additive term β of the current optimal service cost. Recall that the optimal service cost at time any time t is obtained if we have the optimum configuration of the servers and it is not necessarily equal the configuration of the servers by an optimal offline algorithm as discussed in Section 4.2. In particular our algorithm a) only moves when it needs to move because the configuration is not feasible any more and b) always moves a server which improves the service cost as much as possible.

We show that the total number of movements up to a time t of this online greedy algorithm can be upper bounded as a function of the optimal service cost S_t^* at time t. Most significantly, we show that even for $\alpha = 1$, for any $\varepsilon > 0$, as long as $\beta = \Omega(k + k/\varepsilon)$, at all times t, the cost $cost_t^{\mathcal{A}}$ of the greedy algorithm can be upper bounded by the cost $cost_t^{\mathcal{O}}$ of an optimal algorithm as $cost_t^{\mathcal{A}} \leq (1+\varepsilon)cost_t^{\mathcal{O}} + O(\beta + k\log k)$. We also show that this result is almost tight. In particular, an additive term which is at least linear in k is unavoidable (even for much larger multiplicative competitive ratio).

In Section 5.3, we describe a simple, deterministic online algorithm ALG with the following properties. For two parameters $\alpha \geq 1$ and $\beta \geq 0$, ALG guarantees that at all times $t \geq 0$, $S_t^{\mathcal{A}} < \alpha S_t^* + \beta$. The algorithm ALG achieves this while keeping the total movement cost small. More precisely, we prove the following main theorem.

Theorem 5.1. *Let $\mathcal{O}$ denote an optimal offline algorithm. There is a deterministic algorithm $\mathcal{A}$ such that for all times $t \geq 0$, the following statements hold.*

- *If $\alpha = 1$ and $\beta = \Omega\left(k + \frac{k}{\varepsilon}\right)$ for an abitrary $\varepsilon > 0$,*

$$cost_t^{\mathcal{A}} \leq (1+\varepsilon)cost_t^{\mathcal{O}} + O(\beta + k\log k).$$

- *If $\alpha = 1$ and $\beta = \Omega\left(\frac{k\cdot\log k}{\log\log k}\right)$, for every $\varepsilon \geq \log\log k/\log^{1-\delta} k$ and any constant $0 < \delta \leq 1$,*

$$cost_t^{\mathcal{A}} \leq (1+\varepsilon)cost_t^{\mathcal{O}} + O(\beta).$$

Choosing $\alpha > 1$: The results of the above theorem all hold for $\alpha = 1$, i.e., an algorithm is always forced to move to a configuration which is optimal up to the additive term β. Even if α is chosen to be larger than 1, as long as we want to guarantee a reasonably small multiplicative competitive ratio (of order $o(k)$), an additive term of order $\Omega(k)$ is unavoidable. In fact, in order to reduce the additive term to $O(k)$, α has to be chosen to be of order k^δ for some constant $\delta > 0$. Note that in this case, the multiplicative competitive ratio grows to at least $\alpha \gg 1$. However, it might still be desirable to choose $\alpha > 1$. In that case, it can be shown that the movement cost $M_t^{\mathcal{A}}$ of our simple greedy algorithm ALG only grows logarithmically with the optimal service cost S_t^* (where the basis of the logarithm is α). As an application, this for example allows to be $(1+\varepsilon)$-competitive for any constant $\varepsilon > 0$ against an objective function of the form $\gamma \cdot S_t^{\mathcal{A}} + M_t^{\mathcal{A}}$ even if γ is chosen of order $k^{-O(1)}$.

5.1.1 Further Related Work

As we have defined the service cost for the OMFL problem in Section 4.2, the cost of serving a request in the facility location problem is classically given by the distance from the request to the facility to which it is assigned. In a uniform metric, this corresponds to the most basic cost function that can be studied in our framework (service cost is equal to the number of requests at nodes with no servers). As described, we significantly generalize this basic service cost model. In the context of facility location, a similar approach was used in [44]. More concretely, in [44], it is assumed that the cost of a facility increases as a function of the requests it needs to serve.

The generalized uniform OMFL problem studied in this chapter has some resemblance to learning problems [4, 53, 65]. Somewhat similarly to expert learning algorithms where in essence, one converges to the "right set of experts", our algorithm has to converge to the "right set of nodes" to place its servers. However, in our case, the cost will usually be dominated by the total movement cost, i.e., the total cost for replacing the servers. In learning, switching to a different set of experts is usually not considered a (main) cost.

Organization of the Chapter: In the remainder of this chapter, the formal statement of the generalized OMFL problem on the uniform metrics is given in Section 5.2. We also define a general cost function in this section for the problem and provide some properties that are satisfied with the general cost function. In Section 5.3, our simple and greedy deterministic online algorithm for the generalized OMFL problem on uniform metrics is described. In Section 5.4, we provide an overview of our analysis of the online algorithm. The analysis of the online algorithm is then provided in Section 5.5. In Section 5.6 we show that the claim of Theorem 4.1 holds. Note that our lower bound claimed by Theorem 4.1 even holds for the generalized OMFL problem in uniform metrics. At the end of this chapter, in Section 5.7, we discuss the main open question related to this chapter of solving the OMFL problem on general metrics.

5.2 Problem Statement

We are given a set V of n nodes and there is a set of k servers. Further, there are requests $1, 2, \ldots$ that adversarially arrive one at a time. We assume that at time $t \geq 1$, request t arrives at node $v(t) \in V$. For a node $v \in V$, let $r_{v,t}$ be the number of requests at node v after t requests have arrived, i.e.,

$$r_{v,t} := |\{i \leq t : v(i) = v\}| .$$

In order to keep the *overall/total service cost* small, an algorithm can move the servers between the nodes (if necessary, for answering one new request, we allow an algorithm to also move more than one server). However throughout the execution, each of the k servers is always placed at one of the nodes $v \in V$. We define a *configuration* of servers by integers $f_v \in \mathbb{N}_0$ for each $v \in V$ such that $\sum_{v \in V} f_v = k$. We describe such a configuration by a set of pairs as

$$F := \{(v, f_v) : v \in V\}.$$

The initial configuration is denoted by F_0.

Service Cost: We implicitly assume that if a node v has some servers, all requests at v are served by these servers. This also implies that the "assignment" of requests to servers can change over time and the service cost is not cumulative. Depending on the number of servers and the number of requests at a node $v \in V$, an algorithm has to pay some service cost to serve the requests located at v. This service cost of node v is defined by a *service cost function* σ_v such that $\sigma_v(x, y) \geq 0$ is the cost for serving y requests if there are x servers at node v. For convenience, for $t \geq 1$, we also define $\sigma_{v,t}(x) := \sigma_v(x, r_{v,t})$ to be the service cost with x servers at node v at time t. For some configuration F, we denote the total service cost at time t by

$$S_t(F) := \sum_{v \in V} \sigma_{v,t}(f_v) = \sum_{v \in V} \sigma_v(f_v, r_{v,t}).$$

Let S_t^* be the *optimal total service cost* at time t, i.e.

$$S_t^* := \min_F S_t(F).$$

Note that S_t^* is not necessarily the same as the total service cost $S_t^{\mathcal{O}}$ of an optimal algorithm OPT at time t. As mentioned before, an optimal offline algorithm such as OPT can wait until all requests have arrived and just perform all the necessary server movements at the very end. We say that a configuration F^* is an *optimal configuration* at time t if $S_t(F^*) = S_t^*$.

Feasible Solution: For a given algorithm ALG, we denote the solution at time t by $\mathcal{F}_t^{\mathcal{A}} := \{F^{\mathcal{A}}(i) : i \in [0, t]\}$, where $F^{\mathcal{A}}(t)$ is the configuration after reacting to the arrival of request t and where $F^{\mathcal{A}}(0) = F_0$. The service cost of an algorithm ALG at time t is denoted by $S_t^{\mathcal{A}} := S_t(F^{\mathcal{A}}(t))$.

Movement Cost: We define the movement cost $M_t^{\mathcal{A}}$ of given algorithm ALG to be the total number of server movements by time t. Generally, for two configurations, $F = \{(v, f_v) : v \in V\}$ and $F' = \{(v, f'_v) : v \in V\}$, we define the distance $\chi(F, F')$ between the two configurations as follows:

$$\chi(F, F') := \sum_{v \in V} \max\{0, f_v - f'_v\} = \frac{1}{2} \cdot \sum_{v \in V} |f_v - f'_v|. \tag{5.1}$$

The distance $\chi(F, F')$ is equal to the number of movements that are needed to get from configuration F to configuration F' (or vice versa). Based on the definition of χ, we can express the movement cost of an algorithm ALG with solution $\mathcal{F}_t^{\mathcal{A}} = \left\{F^{\mathcal{A}}(i) : i \in [0,t]\right\}$ as $M_t^{\mathcal{A}} = \sum_{i=1}^{t} \chi\left(F^{\mathcal{A}}(i-1), F^{\mathcal{A}}(i)\right)$. The total cost of an algorithm ALG is

$$cost_t^{\mathcal{A}} := S_t^{\mathcal{A}} + M_t^{\mathcal{A}}.$$

The competitive ratio is defined as the total cost of ALG divided by the solution cost of OPT.

5.2.1 Service Cost Function Properties

The service cost function σ has to satisfy a number of natural properties. First of all, for every $v \in V$, $\sigma_v(x, y)$ has to be monotonically decreasing in the number of servers x that are placed at node v and monotonically increasing in the number of requests y at v.

$$\forall v \in V\, \forall x, y \in \mathbb{N}_0 \quad : \quad \sigma_v(x,y) \geq \sigma_v(x+1, y) \tag{5.2}$$

$$\forall v \in V\, \forall x, y \in \mathbb{N}_0 \quad : \quad \sigma_v(x,y) \leq \sigma_v(x, y+1) \tag{5.3}$$

Further, the effect of adding additional servers to a node v should become smaller with the number of servers (convex property in x) and it should not decrease if the number of requests gets larger. Therefore, for all $v \in V$ and all $x, y \in \mathbb{N}_0$, we have

$$\sigma_v(x,y) - \sigma_v(x+1,y) \quad \geq \quad \sigma_v(x+1,y) - \sigma_v(x+2,y) \tag{5.4}$$

$$\sigma_v(x,y) - \sigma_v(x+1,y) \quad \leq \quad \sigma_v(x,y+1) - \sigma_v(x+1,y+1) \tag{5.5}$$

In the following, whenever clear from the context, we omit the superscript ALG in the algorithm-dependent quantities defined above.

5.3 Algorithm Description

The goal of our algorithm is two-fold. On the one hand, we want to guarantee that the service cost of the algorithm is always within some fixed bounds of the optimal service cost. On the other hand, we want to achieve this while keeping the overall movement cost low. Specifically, for two parameters α and β, where

$$\alpha \geq 1 \quad \text{and} \quad \max\{\alpha - 1, \beta\} \geq 1. \tag{5.6}$$

we guarantee that at all times

$$S_t < \alpha \cdot S_t^* + \beta, \tag{5.7}$$

where S_t denotes the total service cost of the algorithm at time t. Condition (5.7) is maintained in the most straightforward greedy manner. Whenever after a new demand arrives, Condition (5.7) is not satisfied, the algorithm greedily move resources until Condition (5.7) holds again. Hence, as long as Condition (5.7) does not hold, the algorithm moves a resource that reduces the total service cost as much as possible. The algorithm stops moving any resources as soon as the validity of Condition (5.7) is restored.

Whenever the algorithm moves a server, it does a best possible move, i.e., a move that achieves the best possible service cost improvement. Thus, the algorithm always moves a server from a node where removing a server is as cheap as possible to a node where adding a server reduces the cost as much as possible. Therefore, for each movement m, we have

$$\begin{aligned} v_m^{src} &\in \arg\min_{v\in V} \left\{\sigma_{v,\tau_m}(f_{v,m-1}-1) - \sigma_{v,\tau_m}(f_{v,m-1})\right\} \text{ and} \quad (5.8)\\ v_m^{dst} &\in \arg\max_{v\in V} \left\{\sigma_{v,\tau_m}(f_{v,m-1}) - \sigma_{v,\tau_m}(f_{v,m-1}+1)\right\}, \quad (5.9)\end{aligned}$$

where $\arg\min_v$ and $\arg\max_v$ denote the sets of nodes minimizing and maximizing the respective terms.

5.4 Analysis Overview

While the algorithm itself is quite simple, its analysis turns out relatively technical. We thus first describe the key steps of the analysis by discussing a simple case. We assume that the service cost at any node is equal to 0 if there is at least one server at the node and the service cost is equal to the number of requests at the node, otherwise. Further, we assume that we run the algorithm of Section 5.3 with parameters $\alpha = 1$ and $\beta = 0$, i.e. after each request arrives, the algorithm moves to a configuration with optimal service cost. Note that these parameter settings violate Condition (5.7) and we will therefore get a weaker bound than the one promised by Theorem 5.1.

First, note that in the described simple scenario, the algorithm clearly never puts more than one server to the same node. Further, whenever the algorithm moves a server from a node u to a node v, the overall service cost has to strictly decrease and thus, the number of requests at node v is larger than the number of requests at node u. Consider some point in time t and let

$$r_{\min}(t) := \min_{v\in V: f_{v,t}=1} r_{v,t}$$

be the minimum number of requests among the nodes v with a server at time t. Hence, whenever at a time t, the algorithm moves a server from a node u to a node v, node u has at least $r_{\min}(t)$ requests and consequently, node v has at least $r_{\min(t)} + 1$ requests. Further, if at some later time $t' > t$, the server at node v is moved to some other node w, because the algorithm always removes a server from a node with as few requests as possible, we have $r_{\min}(t') \geq r_{\min}(t) + 1$. Consequently, if in some time interval $[t_1, t_2]$, there is some server that is moved more than once, we know that $r_{\min}(t_1) < r_{\min}(t_2)$. In our analysis, we partition time into phases, the first phase starts at time 0 and where phases are maximal time intervals in which each server is moved at most once (cf. Definition 5.2 in the formal analysis of the algorithm).

The above argument implies that after each phase $r_{\min}$ increases by at least one and therefore at any time t in Phase p, we have $r_{\min}(t) \geq p - 1$ and at the end of Phase p, we have

$r_{\min}(t) \geq p$. In Section 5.5, the more general form of this statement appears in Lemma 5.3. There, γ_p is defined to be the smallest service cost improvement of any movement in Phase p ($\gamma_p = 1$ in the simple case considered here), and Lemma 5.3 shows that $r_{\min}$ grows by at least γ_p in Phase p. Assume that at some time t in Phase p, a server is moved from a node u to a node v. Because node u already had its server at the end of Phase $p-1$, we have $r_{u,t} = r_{\min}(t) \geq p-1$. Consequently, at the end of Phase p, there is at least one node (the source of the last movement) that has no server and at least $p-1$ requests. The corresponding (more technical) statement in our general analysis appears in Lemma 5.5.

We will bound the total cost of the online algorithm and an optimal offline algorithm from above and below, respectively, as a function of the optimal service cost. Hence, the ratio between these two total costs provides the desired competitive factor. Our algorithm guarantees that at all times, the service cost is within fixed bounds of the optimal service cost (in the simple case here, the service cost is always equal to the optimal service cost). Knowing that there are nodes with many requests and no servers, therefore allows to lower bound the optimal service cost. In the general case, this is done by Lemma 5.8 and Lemma 5.9. In the simple case, considered here, as at the end of Phase p, there are k nodes with at least p requests (the nodes that have servers) and there is at least one additional node with at least $p-1$ requests, we know that at the end of Phase p, the optimal service cost is at least $p-1$. Consequently, the online algorithm (in the simple case) pays exactly the optimal service cost (as mentioned before, in the general case, the service cost is within fixed bounds of the optimal service cost) and at most $(p-1)k$ as movement cost. Hence, the total cost paid by the online algorithm is at most a factor $k+1$ times the optimal service cost since the optimal service cost is at least $p-1$. By choosing α which is slightly larger than 1 and a larger β ($\beta \geq k$), the algorithm becomes more lazy and one can show that the difference between the number of movements of ALG and the optimal service cost becomes significantly smaller. Also note that by construction, the service cost of ALG is always at most $\alpha S_t^* + \beta$.

When analyzing our algorithm, we mostly ignore to take into account the movement cost of an optimal offline algorithm. We only exploit the fact that by the time ALG decides to move a server for the first time, any other algorithm must also move at least one server and therefore the optimal offline cost becomes at least 1.

5.5 Upper Bound Analysis

In the following, we show that how to upper bound the total cost of our online algorithm ALG by a function of the total cost of an optimal offline algorithm OPT. Clearly, the algorithm at all times $t \geq 0$ guarantees that the service cost can be bounded as

$$S_t^{\mathcal{A}} < \alpha \cdot S_t^* + \beta \leq \alpha \cdot cost_t^{\mathcal{O}} + \beta. \tag{5.10}$$

In order to upper bound the total cost, it therefore suffices to study how the movement cost $M_t^{\mathcal{A}}$ of the online algorithm grows as a function of the optimal offline algorithm cost. Let OPT be an optimal offline algorithm and let $F^{\mathcal{O}}(t)$ be the configuration of OPT at time t. Recall that $\chi(F_0, F^{\mathcal{O}}(t))$ denotes the total number of movements required to move from the initial configuration to configuration $F^{\mathcal{O}}(t)$. We therefore have $cost_t^{\mathcal{O}} = S_t^{\mathcal{O}} + M_t^{\mathcal{O}} \geq S_t^* + \chi(F_0, F^{\mathcal{O}}(t))$. In order to upper bound $M_t^{\mathcal{A}}$ as a function of $cost_t^{\mathcal{O}}$, we will upper bound it as a function of $S_t^* + \chi(F_0, F^{\mathcal{O}}(t))$.

Instead of directly dealing with $\chi(F_0, F^{\mathcal{O}}(t))$, we will make use of the fact that our analysis works for a general cost function σ satisfying the conditions given in (5.2), (5.3), (5.4), and (5.5). Given a service cost function σ, consider a function σ' which is defined as follows:

$$\forall v \in V, \forall x \in \{0, \ldots, k\}, \forall y \in \mathbb{N}_0 : \sigma'_v(x, y) := \sigma_v(x, y) + \max\{0, f_v(0) - x\}$$

where $f_v(t)$ is the number of servers at time t on node v. Clearly, σ' also satisfies the conditions given in (5.2), (5.3), (5.4), and (5.5). In addition, for any time t and any configuration $F = \{(v, f_v) : v \in V\}$, we have

$$\begin{aligned} S'_t(F) &= \sum_{v \in V} \sigma'_v(f_v, r_{v,t}) \\ &= \sum_{v \in V} (\sigma_v(f_v, r_{v,t}) + \max\{0, f_v(0) - f_v\}) \\ &\overset{(5.1)}{=} S_t(F) + \chi(F_0, F) \end{aligned} \tag{5.11}$$

where $S'_t(F)$ refers to the total service cost w.r.t. the new cost function σ'. Hence, $S'_t(F)$ exactly measures the sum of service cost and movement cost of a configuration F. Of course now, in all our results, S_t^* corresponds to the combination of service and movement cost of an optimal configuration F^*.

We are now going to analyze the algorithm of Section 5.3. In the following, whenever we refer to the algorithm introduced in Section 5.3, we omit the superscript ALG. In our analysis, we will bound the total costs of optimal offline algorithm OPT and online algorithm ALG from below and above, respectively, as functions of optimal service cost and thus provide the upper bound (competitive factor) promised in Theorem 5.1. Hence we first go through calculating the optimal service cost.

For the analysis of the described online algorithm, we partition the movements into phases $p = 1, 2, \ldots$, where roughly speaking, a phase is a maximal consecutive sequence of movements in which no server is moved twice. We use m_p to denote the first movement of Phase p (for $p \in \mathbb{N}$). In addition, we define $v_m^{src,\mathcal{A}}$ and $v_m^{dst,\mathcal{A}}$ to be the nodes involved in the m-th server move, where we assume that ALG moves a server from node v_m^{src} to v_m^{dst}. Formally, the phases are defined as follows.

Definition 5.2 (Phases). *The movements are divided into phases $p = 1, 2, \ldots$, where Phase p starts with movement m_p and ends with movement $m_{p+1} - 1$. We have $m_1 = 1$, i.e., the first phase starts with the first movement. Further for every $p > 1$, we define*

$$m_p := \min\left\{m > m_{p-1} \ : \ \exists\, m' \in [m_{p-1}, m-1] \text{ s.t. } v_m^{src} = v_{m'}^{dst}\right\}. \tag{5.12}$$

For a Phase $p \geq 1$, let $\lambda_p := m_{p+1} - m_p$ be the number of movements of Phase p.

5.5.1 Optimal Service Cost Analysis

The algorithm moves servers in order to improve the service cost. Throughout the rest of this chapter, we use $\tau_m^{\mathcal{A}}$ to denote the time of the m^{th} movement. For a given movement m, we use $\gamma(m) > 0$ to denote service cost *improvement* of m. Further, we use F_0 to denote the initial configuration of the k servers and for a given (deterministic) algorithm ALG, for any $m \geq 1$, we let $F_m^{\mathcal{A}} = \{(v, f_{v,m}^{\mathcal{A}}) : v \in V\}$ be the configuration of the k servers for ALG after m server movements (i.e., after m server movements of ALG, node v has $f_{v,m}^{\mathcal{A}}$ servers).

$$\begin{aligned}\gamma(m) \;\; &:= \;\; S_{\tau_m}(F_{m-1}) - S_{\tau_m}(F_m) \\ &= \;\; \big(\sigma_{v_m^{dst},\tau_m}(f_{v_m^{dst},m-1}) - \sigma_{v_m^{dst},\tau_m}(f_{v_m^{dst},m})\big) \\ &\qquad - \big(\sigma_{v_m^{src},\tau_m}(f_{v_m^{src},m}) - \sigma_{v_m^{src},\tau_m}(f_{v_m^{src},m-1})\big). \end{aligned} \tag{5.13}$$

For each Phase p, we define the improvement γ_p of p and the *cumulative improvement* Γ_p by Phase p as follows

$$\gamma_p := \min_{m \in [m_p, m_{p+1}-1]} \gamma(m) \quad \text{and} \quad \Gamma_p := \sum_{i=1}^{p} \gamma_i, \quad \Gamma_0 := 0, \;\gamma_0 := 0. \tag{5.14}$$

We are now ready to prove our first technical lemma, which lower bounds the cost of removing servers from nodes with servers (for all $v \in V$ such that $f_v \geq 1$) at any point in the execution. The result of following lemma implies that removing any server of an optimal configuration during some Phase p increases the optimal service cost at least Γ_{p-1} (and Γ_p at end of Phase p) since the servers of an optimal configuration are located at places with maximum number of requests.

Lemma 5.3. *Let m be a movement and, $F = \{(v, f_v) : v \in V\}$ be the configuration of the algorithm at any point in the execution after movement m and let $t \geq \tau_m$ be the time at which the configuration F occurs. Then, for all times $t' \geq t$ and for all nodes $v \in V$, if $f_v > 0$ it holds that*

$$\sigma_{v,t'}(f_v - 1) - \sigma_{v,t'}(f_v) \geq \Gamma_{p-1},$$

where p is the phase in which movement m occurs.

Proof. We will show that for each server movement $m \in \mathbb{N}$ of the algorithm, it holds that

$$\forall v \in V \;:\; f_{v,m} > 0 \implies \sigma_{v,\tau_m}(f_{v,m} - 1) - \sigma_{v,\tau_m}(f_{v,m}) \geq \Gamma_{p-1}, \tag{5.15}$$

where p is the phase in which movement m occurs (i.e., the claim of the lemma holds immediately after movement m). The lemma then follows because (i) any configuration $\{(v, f_v) : v \in V\}$ occurring after movement m is the configuration $F_{m'}$ for some movement $m' \geq m$, (ii) the values Γ_{p-1} are monotonically increasing with p, and (iii) by (5.5), for all $v \in V$, the value $\sigma_{v,t}(f - 1) - \sigma_{v,t}(f)$ is monotonically non-decreasing with t.

It therefore remains to prove (5.15) for every m, where p is the phase of movement m. We prove a slightly stronger statement. Generally, for a movement m' and a Phase p', let $V_{p',m'}^{dst}$

be the set of nodes that have received a new server by some movement $m'' \leq m'$ of Phase p'. Hence,

$$V_{p',m'}^{dst} := \left\{ v \in V : \exists \text{ movement } m'' \leq m' \text{ of Phase } p' \text{ s.t. } v_{m''}^{dst} = v \right\}.$$

We show that in addition to (5.15), it also holds that

$$\forall v \in V_{p,m}^{dst} \;:\; f_{v,m} > 0 \implies \sigma_{v,\tau_m}(f_{v,m} - 1) - \sigma_{v,\tau_m}(f_{v,m}) \geq \Gamma_p. \tag{5.16}$$

We prove (5.15) and (5.16) together by using induction on m.

Induction Base ($m = 1$): The first movement occurs in Phase 1. By (5.14), $\Gamma_0 = 0$ and by (5.2), we also have $\sigma_{v,t}(f-1) - \sigma_{v,t}(f) \geq 0$ for all times $t \geq 0$, all nodes $v \in V$, and all $f \geq 1$. Inequality (5.15) therefore clearly holds for $m = 1$. It remains to show that also (5.16) holds for $m = 1$. We have $V_{1,1}^{dst} = \left\{v_1^{dst}\right\}$ and showing (5.16) for $m = 1$ therefore reduces to showing that $\sigma_{v_1^{dst},\tau_1}(f_{v_1^{dst},1} - 1) - \sigma_{v_1^{dst},\tau_1}(f_{v_1^{dst},1}) \geq \Gamma_1 = \gamma_1$, which follows directly from (5.13) and (5.14).

Induction Step ($m > 1$): We first show that Inequalities (5.15) and (5.16) hold immediately before movement m and thus,

$$\begin{aligned} \forall v \in V \quad &: \quad f_{v,m-1} > 0 \;\Rightarrow\; \sigma_{v,\tau_m}(f_{v,m-1} - 1) - \sigma_{v,\tau_m}(f_{v,m-1}) \geq \Gamma_{p-1}, & (5.17)\\ \forall v \in V_{p,m-1}^{dst} \quad &: \quad f_{v,m-1} > 0 \;\Rightarrow\; \sigma_{v,\tau_m}(f_{v,m-1} - 1) - \sigma_{v,\tau_m}(f_{v,m-1}) \geq \Gamma_p. & (5.18) \end{aligned}$$

If m is not the first movement of Phase p, Inequalities (5.17) and (5.18) follow directly from the induction hypothesis (for $m-1$) and from (5.5). Let us therefore assume that m is the first movement of Phase p. Note that in this case $V_{p,m-1}^{dst} = \emptyset$ and (5.18) therefore trivially holds. Because $m > 1$, we know that in this case $p \geq 2$. From the induction hypothesis and from (5.5), we can therefore conclude that for every node $v \in V_{p-1,m-1}^{dst}$ (every node v that is the destination of some server movement in Phase $p-1$), we have $\sigma_{v,\tau_m}(f_{v,m-1} - 1) - \sigma_{v,\tau_m}(f_{v,m-1}) \geq \Gamma_{p-1}$. Note that for all these nodes, we have $f_{v,m-1} > 0$. Because m is the first movement of Phase p, Definition 5.2 implies that $v_m^{src} \in V_{p-1,m-1}^{dst}$. Applying (5.8), we get that for all $v \in V$, $\sigma_{v,\tau_m}(f_{v,m-1}-1) - \sigma_{v,\tau_m}(f_{v,m-1}) \geq \sigma_{v_m^{src},\tau_m}(f_{v_m^{src},m-1}-1) - \sigma_{v_m^{src},\tau_m}(f_{v_m^{src},m-1}) \geq \Gamma_{p-1}$ and therefore (5.17) also holds if $m \geq 2$ is the first movement of some phase.

We can now prove (5.15) and (5.16). For all nodes $v \notin \left\{v_m^{src}, v_m^{dst}\right\}$, we have $f_{v,m} = f_{v,m-1}$ and we further have $V_{p,m}^{dst} = V_{p,m-1}^{dst} \cup \left\{v_m^{dst}\right\}$. For $v \notin \left\{v_m^{src}, v_m^{dst}\right\}$, (5.15) and (5.16) therefore directly follow from (5.17) and (5.18), respectively. For the two nodes involved in movement m, first note that $v_m^{src} \notin V_{p,m-1}^{dst}$. It therefore suffices to show that

$$\begin{aligned} f_{v_m^{src},m} = 0 \quad \text{or} \quad \sigma_{v_m^{src},\tau_m}(f_{v_m^{src},m} - 1) - \sigma_{v_m^{src},\tau_m}(f_{v_m^{src},m}) \;&\geq\; \Gamma_{p-1}, & (5.19)\\ \text{as well as} \quad \sigma_{v_m^{dst},\tau_m}(f_{v_m^{dst},m} - 1) - \sigma_{v_m^{dst},\tau_m}(f_{v_m^{dst},m}) \;&\geq\; \Gamma_p. & (5.20) \end{aligned}$$

We have $f_{v_m^{src},m} = f_{v_m^{src},m-1} - 1$ and $f_{v_m^{dst},m} = f_{v_m^{dst},m-1} + 1$. Inequality (5.19) therefore directly follows from (5.17) and (5.4). For (5.20), we have

$$\begin{aligned} &\sigma_{v_m^{dst},\tau_m}(f_{v_m^{dst},m} - 1) - \sigma_{v_m^{dst},\tau_m}(f_{v_m^{dst},m}) \\ &\quad\overset{(5.13)}{=} \quad \sigma_{v_m^{src},\tau_m}(f_{v_m^{src},m}) - \sigma_{v_m^{src},\tau_m}(f_{v_m^{src},m} - 1) + \gamma(m) \\ &\quad\overset{(5.17)}{\geq} \quad \Gamma_{p-1} + \gamma(m) \quad \overset{(5.14)}{\geq} \quad \Gamma_p. \end{aligned}$$

This completes the proof of (5.15) and (5.16) and thus the proof of the lemma. □

For each phase number p, let $\theta_p := \tau_{m_p}$ be the time of the the first movement m_p of Phase p. Before continuing, we give lower and upper bounds on γ_p, the improvement of Phase p. For all $p \geq 1$, we define

$$\eta_p := (\alpha - 1) \cdot S^*_{\theta_p} + \beta. \tag{5.21}$$

Lemma 5.4. *Let m be a movement of Phase p and let $F^* = \arg\min_F S_t(F)$ be the optimal configuration at time τ_m. We then have*

$$\frac{\eta_p}{\chi(F_{m-1}, F^*)} \leq \gamma(m) \leq \eta_{p+1}.$$

Proof. For the upper bound, observe that we have

$$\gamma(m) \leq S_{\tau_m}(F_{m-1}) - S^*_{\tau_m}$$

as clearly the service cost cannot be improved by a larger amount. Because at all times t, the algorithm keeps the service cost below $\alpha S^*_t + \beta$, we have $S_{\tau_m - 1}(F_{m-1}) < \alpha S^*_{\tau_m - 1} + \beta \leq \alpha S^*_{\tau_m} + \beta$. The upper bound on $\gamma(m)$ follows from (5.21) and because $S^*_{\tau_m} \leq S^*_{\theta_{p+1}}$.

For the lower bound on $\gamma(m)$, we need to prove that $\chi(F_{m-1}, F^*) \geq \eta_p/\gamma(m)$. Because the algorithm moves a server at time τ_m, we know that $S_{\tau_m}(F_{m-1}) \geq \alpha S_{\tau_m}(F^*) + \beta$ and applying the (5.21) of η_p, we thus have $S_{\tau_m}(F_{m-1}) - S_{\tau_m}(F^*) \geq \eta_p$. Intuitively, we have $\chi(F_{m-1}, F^*) \geq \eta_p/\gamma(m)$ because the algorithm always chooses the best possible movement and thus every possible movement improves the overall service cost by at most $\gamma(m)$. Thus, the number of movements needs to get from F_{m-1} to an optimal configuration F^* has to be at least $\eta_p/\gamma(m)$. For a formal argument, assume that we are given a sequence of $\ell := \chi(F_{m-1}, F^*)$ movements that transform configuration F_{m-1} into configuration F^*. For $i \in [\ell]$, assume that the i^{th} of these movements moves a server from node u_i to node v_i. Further, for any $i \in [\ell]$ let f_i be the number of servers at node u_i and let f'_i be the number of servers at node v_i before the i^{th} of these movements. Because the sequence of movements is minimal to get from F_{m-1} to F^*, we certainly have $f_i \leq f_{u_i,m-1}$ and $f'_i \geq f_{v_i,m-1}$. For the service cost improvement γ of the i^{th} of these movements, we therefore obtain

$$\begin{aligned}
\gamma &= \big(\sigma_{v_i,\tau_m}(f'_i) - \sigma_{v_i,\tau_m}(f'_i + 1)\big) - \big(\sigma_{u_i,\tau_m}(f_i - 1) - \sigma_{u_i,\tau_m}(f_i)\big) \\
&\overset{(5.4)}{\leq} \big(\sigma_{v_i,\tau_m}(f_{v_i,m-1}) - \sigma_{v_i,\tau_m}(f_{v_i,m-1} + 1)\big) \\
&\qquad - \big(\sigma_{u_i,\tau_m}(f_{u_i,m-1} - 1) - \sigma_{u_i,\tau_m}(f_{u_i,m-1})\big) \\
&\leq \gamma(m).
\end{aligned}$$

The last inequality follows from (5.8),(5.9), and (5.13). As the sum of the ℓ service cost improvements has to be at least η_p, we obtain $\ell = \chi(F_{m-1}, F^*) \geq \eta_p/\gamma(m)$ as claimed. □

We can now lower bound the distribution of requests at the time of each movement.

Lemma 5.5. *Let m be a movement of Phase p (for $p \geq 1$). Then, there are integers $\psi_v \geq 0$ for all nodes $v \in V$ such that*

$$\sum_{v \in V} \psi_v \geq k + \frac{\eta_p}{\gamma(m)} \quad \textit{and}$$
$$\forall t \geq \tau_m \; \forall v \in V : \psi_v > 0 \Longrightarrow \sigma_{v,t}(\psi_v - 1) - \sigma_{v,t}(\psi_v) \geq \Gamma_{p-1}.$$

Proof. It suffices to prove the statement for $t = \tau_m$. For larger t, the claim then follows from (5.5). Consider an optimal configuration

$$F^* = \{(v, f_v^*) : v \in V\}$$

at the time τ_m of movement m. Let us further consider the configuration F_{m-1} of the algorithm immediately before movement m. Consider a pair of nodes u and v such that $f_u^* > f_{u,m-1}$ and $f_{v,m-1} > f_v^*$. By the optimality of F^*, we have

$$\sigma_{u,\tau_m}(f_u^* - 1) - \sigma_{u,\tau_m}(f_u^*) \geq \sigma_{v,\tau_m}(f_{v,m-1} - 1) - \sigma_{v,\tau_m}(f_{v,m-1}). \tag{5.22}$$

Otherwise, moving a server from u to v would (strictly) improve the configuration F^*. By Lemma 5.3, we have $\sigma_{v,\tau_m}(f_{v,m-1} - 1) - \sigma_{v,\tau_m}(f_{v,m-1}) \geq \Gamma_{p-1}$ for all nodes v for which $f_{v,m-1} > 0$. Together with (5.22), for all $v \in V$ for which $\max\{f_{v,m-1}, f_v^*\} > 1$, we obtain

$$\sigma_{v,\tau_m}(\max\{f_{v,m-1}, f_v^*\} - 1) - \sigma_{v,\tau_m}(\max\{f_{v,m-1}, f_v^*\}) \geq \Gamma_{p-1}. \tag{5.23}$$

To prove the lemma, it therefore suffices to show that $\sum_{v \in V} \max\{f_{v,m-1}, f_v^*\} \geq k + \eta_p/\gamma(m)$, as we can then set $\psi_v := \max\{f_{v,m-1}, f_v^*\}$ and (5.23) implies the claim of the lemma. By (5.1), we have

$$\sum_{v \in V} \max\{f_{v,m-1}, f_v^*\} = k + \sum_{v \in V} \max\{0, f_v^* - f_{v,m-1}\} = k + \chi(F_{m-1}, F^*).$$

We therefore need that $\chi(F_{m-1}, F^*) \geq \eta_p/\gamma(m)$, which follows from Lemma 5.4. $\square$

In the next lemma, we derive a lower bound on $S^*_{\theta_p}$, the service cost of optimal configuration when Phase p starts. For each Phase $p \geq 1$, we first define $\overline{S}_p$ as follows.

$$\text{For } p \geq 3 : \overline{S}_p := \left(1 + (\alpha - 1)\frac{\gamma_{p-2}}{\gamma_{p-1}}\right) \cdot \overline{S}_{p-1} + \frac{\gamma_{p-2}}{\gamma_{p-1}}\beta, \text{ and } \overline{S}_1 := \overline{S}_2 := 1. \tag{5.24}$$

Lemma 5.6. *For all $p \geq 1$, we have $S^*_{\theta_p} \geq \overline{S}_p$.*

Proof. We prove the lemma by induction on p.

Induction Base ($p = 1, 2$): Using (5.11) we have $S^*_{\theta_1} \geq 1$ and since $\overline{S}_1 = \overline{S}_2 = 1$, we get $S^*_{\theta_2} \geq S^*_{\theta_1} \geq \overline{S}_2 = \overline{S}_1$.

Induction Step ($p > 2$): We use the induction hypothesis to assume that the claim of the lemma is true up to Phase p and we prove that it also holds for Phase $p+1$. Therefore by the induction hypothesis, for all $i \in [p]$,

$$S^*_{\theta_p} \geq \overline{S}_p. \tag{5.25}$$

For all $i \in [p]$, we define $\overline{\eta}_i := (\alpha - 1)\overline{S}_i + \beta$ and $\delta_i := \max\left\{\frac{\overline{\eta}_{i+1}}{\gamma_{i+1}}, \cdots, \frac{\overline{\eta}_p}{\gamma_p}\right\}$. As a consequence of (5.21) and (5.25), we get that $\eta_i \geq \overline{\eta}_i$ for all $i \in [p]$. In the following, let $p' \in [2, p]$ be some phase. Lemma 5.5 implies that after the last movement m of Phase p', there are non-negative integers ψ_v (for $v \in V$) such that $\sum_{v \in V} \psi_v \geq k + \eta_{p'}/\gamma_{p'} \geq \overline{\eta}_{p'}/\gamma_{p'}$ and for all times $t \geq \tau_m$, for all $v \in V$ for which $\psi_v > 0$, $\sigma_{v,t}(\psi_v - 1) - \sigma_{v,t}(\psi_v) \geq \Gamma_{p'-1}$. As there are only k servers for any feasible configuration $F = \{(v, f_v)\}$, we have $\sum_{v \in V} f_v = k$ and therefore $\sum_{v \in V}(\psi_v - f_v) \geq \overline{\eta}_{p'}/\gamma_{p'}$. For any $v \in V$ for which $\psi_v > f_v$, by using (5.4), we get $\sigma_{v,t}(f_v) \geq (\psi_v - f_v)\Gamma_{p'-1}$. Hence, after the last movement of Phase p', for any feasible configuration F, we have $S_t(F) \geq S^*_t \geq \frac{\overline{\eta}_{p'}}{\gamma_{p'}}\Gamma_{p'-1}$. At the beginning of Phase $p+1$ (for $p \geq 2$), the total optimal service cost therefore is

$$S^*_{\theta_{p+1}} \geq \max_{p' \in [2,p]} \frac{\overline{\eta}_{p'}}{\gamma_{p'}}\Gamma_{p'-1} \geq \delta_{p-1}\Gamma_{p-1} + \sum_{i=1}^{p-2}(\delta_i - \delta_{i+1}) \cdot \Gamma_i = \sum_{i=1}^{p-1} \gamma_i \cdot \delta_i. \tag{5.26}$$

We define ζ_i for all $i \in [3, p]$ as follows:

$$\zeta_i := \sum_{j=1}^{i-2} \gamma_j \cdot \delta_j. \tag{5.27}$$

Using the definition of δ_i, we thus have

$$\zeta_{p+1} = \zeta_p + \gamma_{p-1}\delta_{p-1} = \zeta_p + \overline{\eta}_p \frac{\gamma_{p-1}}{\gamma_p}.$$

Considering the definition of $\overline{\eta}_i$ we get

$$\zeta_{p+1} = \zeta_p \cdot \left(1 + (\alpha - 1)\frac{\gamma_{p-1}}{\gamma_p}\right) + \beta \cdot \frac{\gamma_{p-1}}{\gamma_p}.$$

We therefore have $\zeta_{p+1} = \overline{S}_{p+1}$ directly from (5.24) and thus the claim of the lemma follows. □

In order to explicitly lower bound the optimal service cost after p phases, we need the following technical statement.

Lemma 5.7. *Let $\ell \geq 2$ be an integer and consider a sequence $c_1, c_2, \ldots, c_\ell > 0$ of ℓ positive real numbers and let $c_{\max} = \max_{i \in [\ell]} c_i$ and $c_{\min} = \min_{i \in [\ell]} c_i$. Further, let $\lambda \geq 0$ be an arbitrary* *non-negative real number. We have*

$$\text{(I)} \quad \sum_{i=2}^{\ell} \frac{c_{i-1}}{c_i} \geq (\ell-1) \cdot \left(\frac{c_{\min}}{c_{\max}}\right)^{\frac{1}{\ell-1}},$$

$$\text{(II)} \quad \prod_{i=2}^{\ell} \left(1+\lambda\frac{c_{i-1}}{c_i}\right) \geq \left(1+\lambda\left(\frac{c_{\min}}{c_{\max}}\right)^{\frac{1}{\ell-1}}\right)^{\ell-1}.$$

Proof. The first part of the claim follows from the means inequality (the fact that the arithmetic mean is larger than or equal to the geometric mean). In the following, we nevertheless directly prove both parts together. We let $\boldsymbol{x} = (x_1, \dots, x_\ell) \in \mathbb{R}^\ell$ be a vector ℓ real variables and we define multivariate functions $f(\boldsymbol{x}) : \mathbb{R}^\ell \to \mathbb{R}$ and $g(\boldsymbol{x}) : \mathbb{R}^\ell \to \mathbb{R}$ as follows:

$$f(\boldsymbol{x}) := \sum_{i=2}^{\ell} \frac{x_{i-1}}{x_i} \quad \text{and} \quad g(\boldsymbol{x}) := \prod_{i=2}^{\ell}\left(1+\lambda\frac{x_{i-1}}{x_i}\right).$$

We further define $X \subset \mathbb{R}^\ell$ as $X := \left\{(z_1, \dots, z_\ell) \in \mathbb{R}^\ell \,|\, \forall i \in [\ell] : c_{\min} \leq z_i \leq c_{\max}\right\}$. We need to show that for $\boldsymbol{x} \in X$, $f(\boldsymbol{x})$ and $g(\boldsymbol{x})$ are lower bounded by the right-hand sides of Inequalities (I) and (II) above, respectively. Note that X is a closed subset of $\mathbb{R}^\ell$ and because $c_{\min} > 0$, both functions $f(\boldsymbol{x})$ and $g(\boldsymbol{x})$ are continuous when defined on X. The minimum for $\boldsymbol{x} \in X$ is therefore well-defined for both $f(\boldsymbol{x})$ and $g(\boldsymbol{x})$. We show that both $f(\boldsymbol{x})$ and $g(\boldsymbol{x})$ attain their minimum for

$$\boldsymbol{x}^* := (x_1^*, \dots, x_\ell^*), \quad \text{where } \forall i \in [\ell] : x_i^* = c_{\min} \cdot \left(\frac{c_{\max}}{c_{\min}}\right)^{\frac{i-1}{\ell-1}}.$$

Note that $\boldsymbol{x}^*$ is the unique configuration $\boldsymbol{x} \in X$ to the following system of equations

$$x_1 = c_{\min}, \quad x_\ell = c_{\max}, \quad \forall i \in \{2, \dots, \ell-1\} : x_i \in \frac{x_{i-1}}{x_i} = \frac{x_i}{x_{i+1}}. \tag{5.28}$$

Because we know that $\min_{\boldsymbol{x}\in X} f(\boldsymbol{x}) = f(\boldsymbol{x}^*)$ and $\min_{\boldsymbol{x}\in X} g(\boldsymbol{x}) = g(\boldsymbol{x}^*)$, it is therefore sufficient to show that for any $\boldsymbol{y} \in X$ that does not satisfy (5.28), $f(\boldsymbol{y})$ and $g(\boldsymbol{y})$ are not minimal. Let us therefore consider a vector $\boldsymbol{y} = (y_1, \dots, y_\ell) \in X$ that does not satisfy (5.28). First note that both $f(\boldsymbol{x})$ and $g(\boldsymbol{x})$ are strictly monotonically increasing in x_1 and strictly monotonically decreasing in x_ℓ. If either $y_1 > c_{\min}$ or $y_\ell < c_{\max}$, it is therefore clear that $f(\boldsymbol{y})$ and $g(\boldsymbol{y})$ are both not minimal (over X). Let us therefore assume that $y_1 = c_{\min}$ and $y_\ell = c_{\max}$. From the assumption that $\boldsymbol{y}$ does not satisfy (5.28), we then have an $i_0 \in \{2, \dots, \ell-1\}$ for which $\frac{y_{i_0-1}}{y_{i_0}} \neq \frac{y_{i_0}}{y_{i_0+1}}$ and thus $y_{i_0} \neq \sqrt{y_{i_0-1}y_{i_0+1}}$. We define a new vector $\boldsymbol{y'} = (y'_1, \dots, y'_\ell) \in X$ as follows. We have $y'_{i_0} = \sqrt{y_{i_0-1}y_{i_0+1}}$ and $y'_i = y_i$ for all $i \neq i_0$ and we will show that $f(\boldsymbol{y'}) < f(\boldsymbol{y})$ and $g(\boldsymbol{y'}) < g(\boldsymbol{y})$. Define

$$C := \prod_{i\in[2,\ell]\setminus\{i_0,i_0+1\}} \left(1+\lambda\frac{y_{i-1}}{y_i}\right).$$

We then have

$$\begin{aligned} f(\boldsymbol{y}) - f(\boldsymbol{y}') &= \left(\frac{y_{i_0-1}}{y_{i_0}} + \frac{y_{i_0}}{y_{i_0+1}}\right) - \left(\frac{y_{i_0-1}}{y'_{i_0}} + \frac{y'_{i_0}}{y_{i_0+1}}\right) \\ g(\boldsymbol{y}) - g(\boldsymbol{y}') &= \left[\left(1 + \lambda\frac{y_{i_0-1}}{y_{i_0}}\right) \cdot \left(1 + \lambda\frac{y_{i_0}}{y_{i_0+1}}\right) - \left(1 + \lambda\frac{y_{i_0-1}}{y'_{i_0}}\right) \cdot \left(1 + \lambda\frac{y'_{i_0}}{y_{i_0+1}}\right)\right] \cdot C \\ &= \left[\left(\frac{y_{i_0-1}}{y_{i_0}} + \frac{y_{i_0}}{y_{i_0+1}}\right) - \left(\frac{y_{i_0-1}}{y'_{i_0}} + \frac{y'_{i_0}}{y_{i_0+1}}\right)\right] \cdot \lambda C. \end{aligned}$$

Note that $\lambda \geq 0$ and $C > 0$. In both cases, we therefore need to show that

$$\forall y_{i_0} \in [c_{\min}, c_{\max}] \setminus \left\{\sqrt{y_{i_0-1}y_{i_0+1}}\right\} : \left(\frac{y_{i_0-1}}{y_{i_0}} + \frac{y_{i_0}}{y_{i_0+1}}\right) > \left(\frac{y_{i_0-1}}{y'_{i_0}} + \frac{y'_{i_0}}{y_{i_0+1}}\right). \tag{5.29}$$

This follows because the function $h : [c_{\min}, c_{\max}] \to \mathbb{R}$, $h(z) := \frac{y_{i_0-1}}{z} + \frac{z}{y_{i_0+1}}$ is strictly convex for $z \in [c_{\min}, c_{\max}]$ and it has a stationary point at $z = \sqrt{y_{i_0-1}y_{i_0+1}} \in [c_{\min}, c_{\max}]$. □

As long as $(\alpha - 1)S^*_{\theta_p} < \beta$, the effect of the $(\alpha - 1)S^*_{\theta_p}$-term on η_p (and thus of the αS^*_t term in (5.7) is relatively small. Let us therefore first analyze how the service cost grows by just considering terms that depends on β (and not on α).

Lemma 5.8. *For all $p \geq 3$, we have*

$$S^*_{\theta_p} \geq \min\left\{\frac{\beta}{\alpha - 1}, \ \beta \cdot (p-2) \cdot (2k)^{-\frac{1}{p-2}}\right\}.$$

Proof. Assume that $S^*_{\theta_p} < \beta/(\alpha - 1)$ as otherwise the claim of the lemma is trivially true. By Lemma 5.6, using $\alpha \geq 1$, for all $p \geq 3$, we get $\overline{S}_p \geq \overline{S}_{p-1} + \frac{\gamma_{p-2}}{\gamma_{p-1}}\beta$. Plugging in $\overline{S}_2 \geq 0$, induction on p therefore gives

$$S^*_{\theta_p} \geq \overline{S}_p \geq \beta \cdot \sum_{i=2}^{p-1} \frac{\gamma_{i-1}}{\gamma_i} \tag{5.30}$$

for all $p \geq 3$. We define $\gamma_{\min} = \min\{\gamma_1, \ldots, \gamma_{p-1}\}$ and $\gamma_{\max} = \max\{\gamma_1, \ldots, \gamma_{p-1}\}$. By Lemma 5.4 and because $\eta_1 \leq \cdots \leq \eta_{p-1}$, we have $\gamma_{\min} \geq \eta_1/k$ and $\gamma_{\max} \leq \eta_p$. From $\alpha \geq 1$ and (5.21), we have $\eta_1 \geq (\alpha - 1) + \beta$ since we know $S^*_{\theta_p} \geq 1$ for $p \geq 1$ regarding to (5.11). Further, we have $\eta_p = (\alpha - 1)S^*_{\theta_p} + \beta < 2\beta$. We therefore have $\gamma_{\min} \geq [(\alpha - 1) + \beta]/k$ and $\gamma_{\max} < 2\beta$ and thus

$$\frac{\gamma_{\min}}{\gamma_{\max}} \geq \frac{(\alpha - 1) + \beta}{2k\beta} \overset{(5.6)}{\geq} \frac{\max\{\beta, 1\}}{2k\beta} \geq \frac{1}{2k}.$$

The lemma now follows from (5.30) and from Inequality (I) of Lemma 5.7. □

On the other hand, as soon as $S^*_{\theta_p} > \max\left\{1, \frac{\beta}{\alpha-1}\right\}$, the effect of the β-term in (5.7) becomes relatively small. As a second case, therefore, we analyze how the service cost grows by just considering terms that depends on α (and not on β).

Lemma 5.9. *Let* $p_0 \geq 2$ *be a phase for which* $\overline{S}_{p_0} \geq \overline{S}_{p_0-1} \geq S_0 := \max\left\{1, \frac{\beta}{\alpha-1}\right\}$*. For any phase* $p > p_0$*, we have*

$$S^*_{\theta_p} \geq S_0 \cdot \left(1 + \frac{\sqrt{\alpha}-1}{(2k)^{\frac{1}{p-p_0}}}\right)^{p-p_0} \geq \frac{S_0}{2k} \cdot \alpha^{\frac{p-p_0}{2}}.$$

Proof. By Lemma 5.6, using $\beta \geq 0$, for all $p > p_0$, we get $\overline{S}_p \geq \left(1 + (\alpha-1)\frac{\gamma_{p-2}}{\gamma_{p-1}}\right) \cdot \overline{S}_{p-1}$. Induction on p therefore gives

$$S^*_{\theta_p} \geq \overline{S}_p \geq \overline{S}_{p_0} \cdot \prod_{i=p_0}^{p-1} \left(1 + (\alpha-1)\frac{\gamma_{i-1}}{\gamma_i}\right) \tag{5.31}$$

for all $p \geq p_0$. Similarly to before, we define $\gamma_{\min} = \min\{\gamma_{p_0-1}, \ldots, \gamma_{p-1}\}$ and $\gamma_{\max} = \max\{\gamma_{p_0-1}, \ldots, \gamma_{p-1}\}$. By Lemma 5.4, the assumptions regarding p_0, and because the values η_i are non-decreasing in i, we have

$$\begin{aligned} \gamma_{\min} &\geq \frac{\eta_{p_0-1}}{k} \geq \frac{\max\{(\alpha-1)+\beta, 2\beta\}}{k} \quad \text{and} \\ \gamma_{\max} &\leq \eta_p \leq (\alpha-1)S^*_{\theta_p} + \beta \leq 2(\alpha-1)S^*_{\theta_p}. \end{aligned}$$

The last inequality follows because $S^*_{\theta_p} \geq \overline{S}_p \geq \overline{S}_{p_0} \geq \max\left\{1, \frac{\beta}{\alpha-1}\right\}$ and by applying (5.6). We can now apply Inequality (II) from Lemma 5.7 to obtain

$$\begin{aligned} S^*_{\theta_p} \geq \overline{S}_p &\geq \overline{S}_{p_0} \cdot \left(1 + (\alpha-1)\left(\frac{\gamma_{\min}}{\gamma_{\max}}\right)^{\frac{1}{p-p_0}}\right)^{p-p_0} \\ &\geq \overline{S}_{p_0} \cdot \left(1 + (\alpha-1)\left(\frac{\max\{(\alpha-1)+\beta, 2\beta\}}{2k(\alpha-1)S^*_{\theta_p}}\right)^{\frac{1}{p-p_0}}\right)^{p-p_0}. \end{aligned} \tag{5.32}$$

In the following, assume that

$$S^*_{\theta_p} \leq \max\left\{1, \frac{\beta}{\alpha-1}\right\} \alpha^{\frac{p-p_0}{2}}. \tag{5.33}$$

Note that if (5.33) does not hold, the claim of the lemma is trivially true. By replacing $S^*_{\theta_p}$ on the right-hand side of (5.32) with the upper bound of (5.33), we obtain

$$\begin{aligned} S^*_{\theta_p} \geq \overline{S}_p &\geq \overline{S}_{p_0} \cdot \left(1 + (\alpha-1) \cdot \left(\frac{(\alpha-1)+\beta}{2k(\alpha-1)\max\left\{1, \frac{\beta}{\alpha-1}\right\}\alpha^{\frac{p-p_0}{2}}}\right)^{\frac{1}{p-p_0}}\right)^{p-p_0} \\ &\geq \overline{S}_{p_0} \cdot \left(1 + \frac{\alpha-1}{(2k)^{\frac{1}{p-p_0}}\sqrt{\alpha}}\right)^{p-p_0} \\ &\geq \overline{S}_{p_0} \cdot \left(1 + \frac{\sqrt{\alpha}-1}{(2k)^{\frac{1}{p-p_0}}}\right)^{p-p_0} \geq \frac{\overline{S}_{p_0}}{2k} \cdot \alpha^{\frac{p-p_0}{2}}. \end{aligned}$$

The lemma then follows because we assumed that $\overline{S}_{p_0} \geq \max\left\{1, \frac{\beta}{\alpha-1}\right\}$. □

5.5.2 Optimal Offline Algorithm Total Cost

Service Cost: In order to minimize the service cost, we can simply bound the service cost of OPT as follows

$$S^{\mathcal{O}}_{\theta_p} \geq S^*_{\theta_p}.$$

Movement Cost: To simplify our analysis, we take no notice of movement cost by optimal offline algorithm since it has no substantial effect on the competitive factor we will provide since OPT has to pay at least the optimal service cost which we show it is large enough. The total cost of optimal offline algorithm, therefore, is bounded as follows

$$cost^{\mathcal{O}}_{\theta_p} = M^{\mathcal{O}}_{\theta_p} + S^{\mathcal{O}}_{\theta_p} \geq S^*_{\theta_p}. \tag{5.34}$$

5.5.3 Online Algorithm Total Cost

Service Cost: The online algorithm has to keep the service cost smaller than a linear function of optimal service cost as mentioned in (5.7). In other words, the configuration of servers at any time has to be feasible as defined in Section 5.2. Thus

$$S^{\mathcal{A}}_{\theta_p} < \alpha S^*_{\theta_p} + \beta. \tag{5.35}$$

Movement Cost: First, using Definition 5.2 we bound the number of movement in each phase.

Observation 5.10. *For each Phase $p \geq 1$, we have $\lambda_p \leq k$.*

Proof. As an immediate consequence of Definition 5.2, we obtain that the maximum number of movements in each phase is at most k. Let $m > m_p$ and consider the movements $[m_p, m]$. We prove that if $m < m_{p+1}$, no two the movements in $[m_p, m]$ move the same server. The claim then follows because there are only k servers. For the sake of contradiction, assume that there is some server i that is moved more than once and let m' and m'' ($m', m'' \in [m_p, m]$, $m' < m''$) be the first two movements in $[m_p, m]$, where server i is moved. We clearly have $v^{dst}_{m'} = v^{src}_{m''}$ and Definition 5.2 thus leads to a contradiction to the assumption that $m < m_{p+1}$. □

As a result of above observation and Lemma 5.8 and Lemma 5.9, it is possible to prove the following lemma to bound the number of online algorithm movements by means of optimal service cost.

Lemma 5.11. *For any $\alpha \geq 1$ and β satisfying (5.6), there is a deterministic online algorithm $\mathcal{A}$ such that for all times $t \geq 0$, the total movement cost $M^{\mathcal{A}}_t$ is bounded as follows.*

- *If $\alpha = 1$, for any $\ell \geq 1$, $\varepsilon > 0$, and $\beta \geq k(2k)^{1/\ell}/\varepsilon$, we have*

$$M^{\mathcal{A}}_t \leq \varepsilon \cdot S^*_t + O(\ell k).$$

- *For $\alpha \geq 1+\varepsilon$ where $\varepsilon > 0$ is some constant and any β satisfying* (5.6)*, we have*

$$M_t^{\mathcal{A}} \leq k \cdot O\left(1 + \log_\alpha S_t^* + \min\left\{\frac{\log k}{\log\log k}, \log_\alpha k\right\} + \log_\alpha \frac{k}{1+\beta}\right).$$

Proof. First note that by Observation 5.10, the movement cost of our algorithm by time θ_p is at most

$$M_{\theta_p} \leq (p-1)k + 1 \leq pk. \tag{5.36}$$

Together with the lower bounds on $S^*_{\theta_p}$ of Lemma 5.8 and Lemma 5.9, this allows to derive an upper bound on the movement cost of our algorithm as a function of $S^*_{\theta_p}$. Note that as all upper bound claimed in the lemma have an additive term of $O(k)$ (with no specific constant), it is sufficient to prove that the lemma holds for all time $t = \theta_p$, where $p \geq 2$ is a phase number.

Let us first consider the case where $\alpha = 1$. Because in that case $\beta/(\alpha-1)$ is unbounded, we can only apply Lemma 5.8 to upper bound the movement cost as a function of S_t^*. We choose $\ell \geq 1$ and assume that $\beta \geq k(2k)^{1/\ell}/\varepsilon$ for $\varepsilon > 0$. Together with (5.36), for $p \geq \ell+2$, Lemma 5.8 then gives

$$S^*_{\theta_p} \geq \frac{k(2k)^{\frac{1}{\ell}}}{\varepsilon} \cdot (p-2) \cdot (2k)^{-\frac{1}{\ell}} = \frac{k}{\varepsilon}(p-2) \geq \frac{1}{\varepsilon}(M_{\theta_p} - 2k). \tag{5.37}$$

The first part of Lemma 5.11 then follows because the total movement cost for the first $\ell+2$ phases is at most $O(\ell k)$. The special cases are obtained as follows. For $\beta = \Omega\left(k + k/\varepsilon\right)$, we set $\ell = \Theta(\log k)$ and every $\varepsilon > 0$, whereas for $\beta = \Omega(k \log k/\log\log k)$, we set $\varepsilon = \Theta(\log\log k/\log^{1-\delta} k)$ and $\ell = \Theta\big(\frac{1}{\delta} \cdot \frac{\log k}{\log\log k}\big)$ for constant $0 < \delta \leq 1$.

Let us therefore move to the case where $\alpha > 1$. Let p_0 be the first Phase $p_0 \geq 2$ for which $S^*_{\theta_{p_0}} \geq S_0$, where $S_0 = \max\left\{1, \frac{\beta}{\alpha-1}\right\}$ as in Lemma 5.9. Further, we set $p_1 = p_0 + \lceil 2\log_\alpha(2k)\rceil$. Using Lemma 5.9, for $p \geq p_1$, we have

$$S_{\theta_p} \geq \frac{S_0}{2k} \cdot \alpha^{\frac{p-p_1}{2}} \alpha^{\frac{p_1-p_0}{2}} \geq S_0 \cdot \alpha^{\frac{p-p_1}{2}}.$$

We therefore get

$$M_{\theta_p} \leq k \cdot p \leq k\left(p_1 + 2\log_\alpha \frac{S^*_{\theta_p}}{S_0}\right) \leq k\left(p_0 + 1 + 2\log_\alpha S^*_{\theta_p} + \log_\alpha \frac{2k}{S_0}\right).$$

The second claim of Lemma 5.11 then follows by showing that

$$p_0 = O\big(\min\big\{\frac{\log k}{\log\log k}, \log_\alpha k\big\}\big).$$

If $S_0 = 1$, we have $p_0 = 2$. Otherwise, we can apply Lemma 5.8 to upper bound p_0 as the smallest value p_0 for which $\frac{\beta}{\alpha-1} = \beta(p-2)(2k)^{-1/(p-2)}$. For $\alpha = O\big(\frac{\log k}{\log\log k}\big)$, the assumption that α is at least $1+\varepsilon$ for some constant $\varepsilon > 0$ gives that $p_0 = \Theta\big(\frac{\log k}{\log\log k}\big)$. Otherwise, (i.e., for large α), we obtain $p_0 = \Theta(\log_{\alpha-1} k) = \Theta(\log_\alpha k)$. □

Note that by choosing $\alpha > 1$, the dependency of the movement cost $M_t^{\mathcal{A}}$ on the optimal service cost S_t^* is only logarithmic because terms $\min\left\{\frac{\log k}{\log\log k}, \log_\alpha k\right\}$ and $\log_\alpha \frac{k}{1+\beta}$ are dominated by $\log k$.

Proof of Theorem 5.1. Putting (5.34), (5.35), and Lemma 5.11 all together conclude the claim of theorem. $\square$

5.6 Lower Bound Analysis

The aim of this section is to formally prove our lower bound stated in Theorem 4.1 in Section 4.3. As discussed in Section 4.3, the lower bound even holds for the OMFL problem with uniform distances, that is it even holds for special case of the generalized uniform OMFL problem, where each node $v \in V$ can only have either 0 or 1 servers and where the cost for serving x requests at a server with 0 servers is linear in x. The lower bound thus holds for the generalized uniform OMFL problem and for the OMFL problem. An outline of the lower bound analysis was provided in Section 4.3.1. The formal proof consists of three parts. In Section 5.6.1, given some online algorithm ALG, we construct an explicit bad execution. In Section 5.6.2, we analyze the cost of the online algorithm ALG in the constructed execution and in Section 5.6.3, we bound the cost of an optimal offline algorithm OPT and we combine everything to complete the proof of Theorem 4.1.

5.6.1 Lower Bound Execution

We assume that ALG is the given online algorithm and OPT is an optimal offline algorithm. Further recall that we assume that ALG guarantees that the difference between the service cost of ALG and the optimal service cost at all times is less than β for some given $\beta > 0$.

We need n to be sufficiently large and for simplicity, we assume that $n \geq 3k$. We denote a feasible configuration by a set $F \subset V$ of size $|F| = k$. Further, without loss of generality, we assume that all servers of ALG and OPT are at the same locations at the beginning (i.e. at time $t = 0$). At each point t in the execution, a configuration F_t^* with optimal service cost (note that F_t^* does not necessarily equal the configuration of OPT) places servers at the k nodes with the most requests (breaking ties arbitrarily if there are several nodes with the same number of requests). Also, at a time t the optimal service cost is equal to the total number of requests at nodes in $V \setminus F_t^*$ for an arbitrary optimal configuration F_t^*.

Time is divided into phases. We construct the execution such that it lasts for at least k phases. As described in the outline, we define integers $\Gamma_1 < \Gamma_2 < \dots$ such that at the end of phase i, there are exactly k nodes with Γ_i requests (and all other nodes have fewer requests). For each phase i, we define V_i to be this set of k nodes with Γ_i requests. We also fix integers $k/3 \geq n_1 \geq n_2 \geq \dots \geq 1$ and at the beginning of each phase i, we pick a set N_i of n_i nodes to which we directly add requests so that all of them have exactly Γ_i requests. For $i = 1$, we pick N_1 as an arbitrary subset of $V \setminus F_0$. We define $V_0 := F_0$. For $i \geq 2$, we choose N_i as an arbitrary subset of $V_{i-2} \setminus V_{i-1}$. Clearly, at the end of phase i, we have $N_i \subseteq V_i$ as otherwise there would be more than k nodes with exactly Γ_i requests. Note that because $N_{i-1} \subseteq V_{i-1}$ and because $N_{i-1} \cap V_{i-2} = \emptyset$, $V_{i-2} \setminus V_{i-1}$ contains $n_{i-1} \geq n_i$ nodes and it is therefore possible to choose N_i as described. Note also that because $N_i \subseteq V_{i-2} \setminus V_{i-1}$, at the beginning of phase i all nodes in N_i have exactly Γ_{i-2} requests. The remaining ones of the k nodes that end up in V_i (and thus have Γ_i requests at the end of phase i) are chosen among the nodes in V_{i-1}. Consequently, at the end of phase $i-1$ and thus at the beginning of phase i, there are exactly k nodes V_{i-1} with Γ_{i-1} requests, n_{i-1} nodes $V_{i-2} \setminus V_{i-1}$ with Γ_{i-2} requests, $n_{i-2} - n_{i-1}$ requests $V_{i-3} \setminus (V_{i-2} \cup N_{i-1})$ with Γ_{i-3} requests, $n_{i-3} - n_{i-2}$ nodes with Γ_{i-4} requests, and so on. Now, n_i of the nodes in $V_{i-2} \setminus V_{i-1}$ are chosen as set N_i and we increase their number of requests to Γ_i. From now on, throughout phase i, there are $k + n_i$ nodes with at least Γ_{i-1} requests such that at most k of these nodes have Γ_i requests. The number of nodes with less than Γ_{i-1} requests is the same as at the end of phase $i-1$. In fact nodes that are not in $V_{i-1} \cup N_i$ do not change their number of requests after phase $i-1$. As a consequence of the execution, after increasing the number of requests in N_i to Γ_i, the optimal service cost remains constant throughout phase $i \geq 1$ and it can be evaluated to

$$\Sigma_i^* := n_i \cdot \Gamma_{i-1} + \sum_{j=2}^{i-1} (n_j - n_{j+1}) \Gamma_{j-1}.$$

For convenience, we also define $\Sigma_0^* := 0$ and moreover $\Sigma_1^* = 0$ since there are at most k nodes with Γ_1 requests at the end of phase 1.

In the following, let v be a free node at some point in the execution, if the algorithm currently has no server at node v. We now fix a phase $p \geq 1$ and assume that we are at a time t, when we have already picked the set N_p and increased the number of requests of nodes in N_p to Γ_p. By the above observation, we have $S_t^* = \Sigma_p^*$ and therefore $\mathcal{A}$ is forced to move if there are n_p free nodes with Γ_p requests and if we choose Γ_p such that

$$\gamma_p := \Gamma_p - \Gamma_{p-1} = \frac{(\alpha - 1)\Sigma_p^* + \beta}{n_p}. \tag{5.38}$$

We can now describe how and when the remaining $k - n_p$ nodes of V_p are chosen after picking the nodes in N_p. As described above, the nodes are chosen from V_{p-1}. We choose the nodes sequentially. Whenever we choose a new node from V_p, we pick some free node $v \in V_{p-1}$ with less than Γ_p requests and increase the number of requests of v to Γ_p. As described above, Γ_p is chosen large enough (as given in (5.38)) such that throughout phase p there are never more than $n_p - 1$ free nodes with Γ_p requests. Because $|N_p \cup V_{p-1}| = k + n_p$, as long as there are at most k nodes with Γ_p requests there always needs to be a free node $v \in V_{p-1}$ that we can pick and we actually manage to add k nodes to V_p.

5.6.2 Online Algorithm Total Cost

The service cost paid by ALG at any time t could be simply lower bounded by S_t^*. Hence, it remains to compute a lower bound for $M_t^{\mathcal{A}}$ as a function of optimal service cost. The following lemma computes such a lower bound.

Lemma 5.12. *For any $\alpha \geq 1$ and β, assume $\mathcal{A}$ be any deterministic online algorithm that can solve the problem. There exists a time $t > 0$ such that the execution of Section 5.6.1 guarantees the total movement cost $M_t^{\mathcal{A}}$ can be bounded as follows.*

- *If $\alpha = 1$, for any $\ell \geq 1$, $\varepsilon > 0$, and $\beta \leq k(2k)^{1/\ell}/\varepsilon$, we have*

$$M_t^{\mathcal{A}} \geq \varepsilon \cdot S_t^* + \Omega(\ell k).$$

  Specifically, for $\beta = O(k/\varepsilon)$ we get $M_t^{\mathcal{A}} \geq \varepsilon \cdot S_t^* + \Omega(k \log k)$ and for $\beta = O\big(\frac{k \log k}{\log \log k}\big)$ we have $M_t^{\mathcal{A}} \geq \varepsilon \cdot S_t^* + \Omega\big(\frac{k \log k}{\log \log k}\big)$.

- *For $\alpha \geq 1 + \varepsilon$ where $\varepsilon > 0$ is some constant and any β, we have*

$$M_t^{\mathcal{A}} \geq k \cdot \Omega\left(1 + \log_\alpha S_t^*\right).$$

Proof. Let us count the number of movements of ALG in a given Phase p. At each point in time t during the phase, let Φ_t be the number of free nodes with Γ_p requests (possibly including a node v that we already chose to be added to V_p). We know that for all t, $\Phi_t < n_p$. Whenever we decide to add a new node v to V_p, Φ_t increases by 1 (as v is a free node). The value of Φ_t can only decrease when ALG moves a server and each server movement reduces the value of Φ_t by at most 1. As after fixing N_p, we add $k - n_p$ nodes to V_p, we need at least $k - 2n_p \geq k/3$ movements to keep Φ_t below n_p throughout the phase. Consequently, every online algorithm ALG has to do at least $k/3$ movements in each phase.

Now we upper bound the optimal service cost Σ_p^* as a function of α, β, and p. Using (5.38), for all $p \geq 0$, we have

$$\Sigma_p^* = \sum_{i=1}^{p} n_p \cdot \gamma_{p-1}$$

For $p \geq 1$, we then get

$$\begin{aligned} \Sigma_p^* &= \frac{n_p}{n_{p-1}}\big((\alpha-1)\Sigma_{p-1}^* + \beta\big) + \Sigma_{p-1}^* \\ &= \left(1 + (\alpha-1)\frac{n_p}{n_{p-1}}\right) \cdot \Sigma_{p-1}^* + \beta \cdot \frac{n_p}{n_{p-1}}. \end{aligned} \tag{5.39}$$

In the following, we for simplicity assume that for $i = 1, 2, \ldots, p$, values n_i do not have to be integers. For integer n_i, the proof works in the same way, but becomes more technical and harder to read. We fix the values of n_i as

$$n_i := (k/3)^{\frac{p-i}{p-1}}$$

such that $n_1 = k/3$ and $n_p = 1$. For all $i \geq 1$, we then have $\frac{n_i}{n_i - 1} = \big(\frac{k}{3}\big)^{-\frac{1}{p-1}}$. Equation (5.39) now be simplified as

$$\Sigma_p^* = \left(1 + \frac{\alpha - 1}{(k/3)^{1/(p-1)}}\right) \cdot \Sigma_{p-1}^* + \beta \cdot \frac{1}{(k/3)^{1/(p-1)}}. \tag{5.40}$$

We have already seen that $S_t^* = \Sigma_p^*$. Using (5.40) and (5.39), the claim of the first part of the lemma follows analogously from Lemma 5.6 and Lemma 5.8 and the claim of the second part of the lemma follows analogously from Lemma 5.6 and Lemma 5.9 in the upper bound analysis section. □

5.6.3 Optimal Offline Algorithm Total Cost

An optimal offline algorithm, say OPT, knows the request sequence in advance. In other words, it can wait until all requests have arrived and just perform all the necessary server movements at the very end. Therefore, an upper bound for the total cost of OPT at any time t is

$$cost_t^{\mathcal{O}} \leq k + S_t^*. \tag{5.41}$$

We now have everything we need to prove Theorem 4.1.

Proof of Theorem 4.1. The proof of Theorem 4.1 now directly follows from Lemma 5.12 and from Equation (5.41). □

5.7 Chapter Notes

As discussed in Section 4.4, combined with the general cost functions studied in this chapter, it could be possible to solve the OMFL problem on HSTs. On each level of the hierarchical decomposition, the cost of each subtree can potentially be modeled using a cost function similar to what we use in this chapter. The online algorithm for the generalized uniform OMFL that was described in Section 5.3 can be used as a building block to devise an online algorithm which recursively solves the OMFL on an HST. Roughly speaking, each internal node of the HST runs an instance of the generalized uniform OMFL that determines how to distribute the available servers among its children nodes. Starting from the root, which has k servers, the recursive calls to the generalized uniform OMFL determine the number of servers at each leaf of the HST, giving a feasible OMFL solution.

Chapter 6

Minimizing Movement in Generalized Uniform OMFL: A Lower Bound

6.1 Introduction

In Chapter 4, we mentioned that the mobile facility location (MFL) problem in general metrics was introduced in [28] as a movement problem. In this chapter, we introduce and study a movement version of the generalized uniform OMFL problem that was studied in Chapter 5. In the movement version of the problem, when measuring the cost of an algorithm, we only take the number of movements of the servers into account. Clearly, if there is no requirement on the service cost of a solution, an optimal would not move any servers at all. It is therefore reasonable to define a threshold for the overall service cost as defined in Chapter 5. Every algorithm that solves the problem must at all times keep its overall service cost below that threshold, which is defined as a function of the optimal service cost. In the following, we formally provide a definition of the problem. Note that the following problem statement appears to be almost the same as the one provided in Section 5.2. However, there are some essential differences.

6.1.1 Problem Statement

We are given a set V of n nodes and there is a set of k servers. Further, there are requests $1, 2, \dots$ that adversarially arrive one at a time in an online fashion. Moreover two parameters α and β are given such that

$$\alpha \geq 1 \quad \text{and} \quad \max\{\alpha - 1, \beta\} \geq 1. \tag{6.1}$$

We assume that at time $t \geq 1$, request t arrives at node $v(t) \in V$. For a node $v \in V$, let $r_{v,t}$ be the number of requests at node v after t requests have arrived, i.e., $r_{v,t} := |\{i \leq t : v(i) = v\}|$. The *service cost* of node v at time t denoted by $\sigma_{v,t}$ is 0 if there is a server at v at time t and is $r_{v,t}$, otherwise. The *total service cost* at each time is the sum of the service costs of all the nodes in V at that time. In order to keep the total service cost small, an algorithm can move the servers between the nodes (if necessary, for answering one new request, we allow an algorithm to also move more than one server). However throughout the execution, each of the k servers is always placed at one of the nodes $v \in V$. We define a *configuration* of servers by integers $f_v \in \{0, 1\}$ for each $v \in V$ such that $\sum_{v \in V} f_v = k$. We describe such a configuration by a set of pairs as $F := \{(v, f_v) : v \in V\}$. The initial configuration is denoted by F_0. For some configuration F, we denote the total service cost at time t by $S_t(F) := \sum_{v \in V} \sigma_{v,t}$.

Feasible Configuration: We define a configuration F to be feasible at time t iff

$$S_t(F) < \alpha \cdot S_t^* + \beta \tag{6.2}$$

where S_t^* is the *optimal total service cost* at time t, i.e. $S_t^* := \min_F S_t(F)$. Note that S_t^* is not necessarily the same as the total service cost $S_t^{\mathcal{O}}$ of an optimal offline algorithm OPT at time t. We say that a configuration F^* is an *optimal configuration* at time t if $S_t(F^*) = S_t^*$.

Feasible Solution: For a given algorithm ALG, we denote the solution at time t by $\mathcal{F}_t^{\mathcal{A}} := \left\{F^{\mathcal{A}}(i) : i \in [0, t]\right\}$, where $F^{\mathcal{A}}(t)$ is the feasible configuration after reacting to the arrival of request t and where $F^{\mathcal{A}}(0) = F_0$. The service cost of an algorithm ALG at time t is denoted by $S_t^{\mathcal{A}} := S_t(F^{\mathcal{A}}(t))$.

Movement Cost: We define the movement cost $M_t^{\mathcal{A}}$ of given algorithm ALG to be the total number of server movements by time t. Generally, for two feasible configurations, $F = \{(v, f_v) : v \in V\}$ and $F' = \{(v, f'_v) : v \in V\}$, we define the distance $\chi(F, F')$ between the two configurations as follows:

$$\chi(F, F') := \sum_{v \in V} \max\{0, f_v - f'_v\} = \frac{1}{2} \cdot \sum_{v \in V} |f_v - f'_v| \,. \tag{6.3}$$

The distance $\chi(F, F')$ is equal to the number of movements that are needed to get from configuration F to configuration F' (or vice versa). Based on the definition of χ, we can express the movement cost of an algorithm ALG with solution $\mathcal{F}_t^{\mathcal{A}} = \left\{F^{\mathcal{A}}(i) : i \in [0, t]\right\}$ as $M_t^{\mathcal{A}} = \sum_{i=1}^{t} \chi\left(F^{\mathcal{A}}(i-1), F^{\mathcal{A}}(i)\right)$.

The total cost of an algorithm including an optimal offline algorithm, say OPT, is the total number of movements made by the algorithm. Note that an optimal offline algorithm cannot wait until all requests have arrived and just perform all the necessary server movements at the very end. The OPT algorithm must change the configuration of its servers as soon as Equation (6.2) is violated. An online algorithm has to be competitive with an optimal offline algorithm.

6.1.2 Further Related Work

The movement version of the generalized uniform OMFL problem generally falls into a class of movement problems introduced in [28]. In this version, the most similar of the classic problems is the paging problem [68] (equivalent to the k-server problem [54] with uniform distances). In the k-server problem, every new request has to be served by moving some server to the location of the request and the only cost considered is the total movement cost.

An interesting special case of the work studied in this chapter is when the service cost at a node u with 0 servers equals the number of requests at u and u's service cost is 0, otherwise. This case is closely related to the paging problem. Naturally, our problem is also related to metrical task systems, which can be seen as a generalization of the k-server problem [20].

Organization of the Chapter: In the rest of this chapter, we provide a lower bound for movement problem defined above in Section 6.2. In the last section, we discuss two open questions based on the movement problem defined in this chapter.

6.2 A Lower Bound

In this section, we provide a lower bound for the problem stated in Section 6.1.1. The following theorem claims that for every k, the competitive ratio of every deterministic online algorithm needs to be at least $\Omega(n)$. We remark that this lower bound even holds for the simple (and natural) scenario, where the service cost at a node with at least 1 server is 0 and the service cost at a node with 0 servers is equal to the number of requests at that node.

Theorem 6.1. *Assume that we are given parameters α and β which satisfy (6.1). Then, for any online algorithm $\mathcal{A}$ and for every $1 \leq k < n$, there exists an execution and a time $t > 0$ such that the competitive ratio between the number of movements by $\mathcal{A}$ and the number of movements of an optimal offline algorithm $\mathcal{O}$ is at least $n/2$. More precisely for all $M_t^{\mathcal{O}} > 0$ there is an execution such that $M_t^{\mathcal{A}} \geq \frac{n}{2} \cdot M_t^{\mathcal{O}}$.*

We provide our lower bound execution in the following. As we can assume that each node either has 0 or 1 servers, we slightly overload notation and simply denote a feasible configuration by a set $F \subset V$ of size $|F| = k$.

6.2.1 Lower Bound Analysis

We first fix ALG to be any deterministic online algorithm denoted by $\mathcal{A}$ and OPT to be any optimal offline algorithm denoted by $\mathcal{O}$. For proving the statement of Theorem 6.1, we distinguish two cases, depending on the number of servers k. In both cases, we define *iterations* to be subsequences of requests such that ALG needs to move at least once per iteration. The number of movements by ALG is therefore at least the number of iterations of a given execution.

Case $k \le \lfloor n/2 \rfloor$: At the beginning, we place a large number of requests on any $k-1$ nodes that initially have servers. We choose this number of requests sufficiently large such that no algorithm can ever move any of these $k-1$ servers. This essentially reduces the problem to $k = 1$ and $n - k + 1$ nodes.

To bound the number of movements by OPT, we then consider intervals of $n-k$ iterations such that ALG is forced to move in each iteration. During each interval, the requests are distributed in such a way that at the beginning of the i-th iteration of the interval there are at least $n-k-i+1$ nodes such that if any offline algorithm places a server on one of these nodes, (6.2) remains satisfied throughout the whole interval. Hence, there exists an offline algorithm that moves at most once in each interval and therefore the number of movements by OPT is upper bounded by the number of intervals.

Case $k > \lfloor n/2 \rfloor$: In this case, there is some resemblance between the constructed execution and the lower bound constructions for the paging problem. For simplicity assume that there are $n = k + 1$ nodes (we let requests arrive at only $k + 1$ nodes). At the beginning of each iteration we locate a sufficiently large number of requests on the node without any server of ALG such that (6.2) is violated. Thus, ALG has to move at least one server to keep (6.2) satisfied. By contrast, OPT does not need to move in each iteration. There is always a node which will not get new requests for the next k iterations and therefore OPT only needs to move at most once every k iterations to keep (6.2) satisfied.

Proof of Theorem 6.1. Consider any request sequence. First we provide a partitioning of the request sequence as follows. The request sequence is partitioned into *iterations*. Iteration 0 is the empty sequence and for every $i \ge 1$, iteration i consists of a request sequence of a length dependent on α, β, and the iteration number i. The request sequence of an iteration i is chosen dependent on a given online algorithm ALG such that ALG must move at least once in iteration i. We will see that while ALG needs to move at least once per iteration, there is an offline algorithm which only moves once every at least $n/2$ iterations.

In the proof, we reduce all the cases to two extreme cases. In the first case, we reduce the original metric on a set of n nodes with $k \le \lfloor n/2 \rfloor$ servers to the case where there is only 1 server. To do this, we first place sufficiently many requests on $k-1$ nodes that have servers at the beginning of execution (for simplicity, assume that we place an unbounded number of requests on these nodes). This prevents any algorithm from moving its servers from these $k-1$ nodes during the execution and hence we can ignore these $k-1$ nodes an servers in our analysis. In contrast, for the second case where $k > \lfloor n/2 \rfloor$, we assume that w.l.o.g., $k = n-1$ by simply only placing requests on the k nodes which have servers at the beginning and on one additional node.

In the following, we let t_i denote the end of an iteration i. Moreover suppose $\mathcal{I}$ is the total number of iterations, where we assume that $\mathcal{I} \equiv 0 \pmod{\max\{k, n-k\}}$.

Case $k \le \lfloor n/2 \rfloor$: The idea behind the execution is to uniformly increase the number of requests on the $n-k$ nodes that do not have the server at the beginning of an iteration i (i.e., at time t_{i-1}) in such a way that ALG has to move at least once to satisfy (6.2) at the end of iteration i. Moreover the distribution of requests guarantees that any node without the server at time t_{i-1} is a candidate to have the (free) server of ALG at time t_i. Let $v_t^{\mathcal{A}}$ denote the node on which ALG locates its server at time t and let $U(t)$ be the set of all nodes without server at time t. Moreover, let v_t^* be a node which has the largest number of requests among all nodes at time t. The node with the largest number of requests at the end if an iteration i, i.e. $v_{t_i}^*$, is chosen such that $v_{t_i}^* \neq v_{t_{i-1}}^{\mathcal{A}}$. At time 0, we have $r_u = 0$ for all nodes u. The distribution of requests at the end of iteration i is as follows:

$$\begin{aligned} \forall u \in U(t_{i-1}) \setminus \left\{v_{t_i}^*\right\} : r_u &= r_{v_{t_{i-1}}^*} + \max\{\beta, 1\}, &(6.4)\\ r_{v_{t_{i-1}}^{\mathcal{A}}} &= r_{v_{t_{i-1}}^*}, &(6.5)\\ r_{v_{t_i}^*} &= (\alpha - 1) \cdot S_{t_i}^* + r_{v_{t_{i-1}}^*} + \beta. &(6.6) \end{aligned}$$

Note that since it is clear from the context, we skip the second subscript (i.e., t) when referring to the number of requests at a node (cf. Section 6.1.1).

Claim 6.2. *The above execution guarantees that* ALG *has to move at least once per iteration. Further, there exists an offline algorithm $\mathscr{A}$ that moves its servers at most $\mathcal{I}/(n-k)$ times.*

Proof. Consider any interval of $n-k$ iterations such that the first iteration of this interval has ending time τ_1 and the finishing time of the last iteration (or the finishing time of the interval) is τ_{n-k}. Further, suppose the previous interval has finished at $\hat{t}$. Obviously, if this is the first interval, $\hat{t} = 0$. Let $U := U(\hat{t}) \setminus \bigcup_{t=\tau_1}^{\tau_{n-k}} \left\{v_t^{\mathcal{A}}\right\}$ denote the set of nodes which have not had the server of ALG during this interval. The offline algorithm for all iterations of this interval, locates its server either on node $v_{\tau_{n-k}}^{\mathcal{A}}$ if set U is empty or on some node in U, otherwise. The case in which U is empty indicates that every node in $U(\hat{t})$ has had the server of ALG exactly once within the interval. Whenever the offline algorithm needs to move, it locates its server at a node in $U \cup \left\{v_{\tau_{n-k}}^{\mathcal{A}}\right\}$. On the one hand and according to (6.4), node $v_{\tau_{n-k}}^{\mathcal{A}}$ or any node in U (in the case this set is not empty) has at least $r_{v_{t_{i-1}}^*} + \max\{\beta, 1\}$ requests at the end of each iteration i that is in this interval. Therefore, the offline service cost at t_i is

$$S_{t_i}^{\mathscr{A}} \leq (\alpha - 1) \cdot S_{t_i}^* + 2r_{v_{t_{i-1}}^*} + \beta + (n-k-2) \cdot \left(\max\{\beta, 1\} + r_{v_{t_{i-1}}^*}\right) \tag{6.7}$$

On the other hand, the optimal service cost is

$$S_{t_i}^* = (n-k-1) \cdot \left(\max\{\beta, 1\} + r_{v_{t_{i-1}}^*}\right) + r_{v_{t_{i-1}}^*} \tag{6.8}$$

using (6.4), (6.5), and (6.6). Hence (6.7) and (6.8) imply that

$$S_{t_i}^{\mathscr{A}} < \alpha S_{t_i}^* + \beta. \tag{6.9}$$

This guarantees that offline algorithm does not need to move more than once during any interval of $n-k$ iterations. In other words, at the beginning of the interval, the offline algorithm decides to locate its server to a node in $U \cup \left\{ v^{\mathcal{A}}_{\tau_{n-k}} \right\}$ if it needs because it knows the behavior of the online algorithm in advance as well as the request sequence. According to (6.9), this one movement by $\mathscr{A}$ is sufficient to keep (6.2) satisfied within the interval. Therefore, the offline algorithm moves at most $\mathcal{I}/(n-k)$ times.

At the end of each iteration i, if the online algorithm has not moved yet within the iteration i then we have $v^{\mathcal{A}}_{t_{i-1}} = v^{\mathcal{A}}_{t_i}$. Thus,

$$S^{\mathcal{A}}_{t_i} = (\alpha - 1) \cdot S^*_{t_i} + r_{v^*_{t_{i-1}}} + \beta + (n-k-1) \cdot \left(\max\{\beta, 1\} + r_{v^*_{t_{i-1}}} \right) \tag{6.10}$$

with respect to (6.4), (6.5), and (6.6). Therefore due to (6.8) and (6.10) we have $S^{\mathcal{A}}_{t_i} = \alpha S^*_{t_i} + \beta$. This implies that the online algorithm must had moved at least once to guarantee

$$\forall i : v^{\mathcal{A}}_{t_{i-1}} \neq v^{\mathcal{A}}_{t_i}.$$

Thus ALG has to move once per iteration and then the claim holds. □

Corollary 6.3. *The Claim 6.2 implies that*

$$M^{\mathcal{O}}_t \leq \frac{M^{\mathcal{A}}_t}{n-k}.$$

where t be the ending time of $(c \cdot (n-k))$-th iteration for any integer $c \geq 1$.

Proof. It follows by the fact that $M^{\mathcal{O}}_t \leq M^{\mathscr{A}}_t$. □

Case $k > \lfloor n/2 \rfloor$: Here when we have more servers than half of the nodes, we assume, w.l.o.g. $n = k + 1$. This is doable by letting the requests arrive at a fix set of nodes of size $k+1$ including k servers. Therefore, at each time there is only one node without a server in which this situation holds for any algorithm. Let $\bar{v}^{\mathcal{A}}_t$ denote the node without any server of ALG at time t. We force ALG to move in each iteration i by putting large enough number of requests on $\bar{v}^{\mathcal{A}}_{t_{i-1}}$ while any optimal offline algorithm only moves one of its servers after at least k iterations. Consider an interval of k iterations starting from the first iteration of this interval with ending time τ_1 and ending at the last iteration at time τ_k. For any iteration i of this interval the distribution of the requests at the end of the iteration is as follows.

$$r_{\bar{v}^{\mathcal{A}}_{t_{i-1}}} = \alpha S^*_{t_i} + \max\{\beta, 1\}. \tag{6.11}$$

According to (6.11) the optimal service cost does not change during the interval, i.e. $S^*_{\tau_i} = S^*_{\tau_i+1}$ for all $i \in [k-1]$ of the current interval.

Claim 6.4. *The above execution guarantees that* ALG *has to move at least once per iteration while the number of movements by any optimal offline algorithm is at most $\mathcal{I}/k$.*

Proof. At the end of iteration i, assume $\bar{v}^{\mathcal{A}}_{t_{i-1}} = \bar{v}^{\mathcal{A}}_{t_i}$, then we have

$$S^{\mathcal{A}}_{t_i} = \alpha S^*_{t_i} + \max\{\beta, 1\} \geq \alpha S^*_{t_i} + \beta \tag{6.12}$$

using (6.11). It implies that the online algorithm must had moved at least once to guarantee

$$\forall i : \bar{v}^{\mathcal{A}}_{t_{i-1}} \neq \bar{v}^{\mathcal{A}}_{t_i}.$$

The optimal offline algorithm, by contrast, need to move a server from $\bar{v}^{\mathcal{A}}_{\tau_k}$ to $\bar{v}^{\mathcal{A}}_{\tau_1}$ during the interval with respect to the request distribution in (6.11). The node $\bar{v}^{\mathcal{A}}_{\tau_k}$ is the node has $\alpha S^*_{\hat{t}} + \max\{\beta, 1\}$ requests within the interval due to (6.11) where $\hat{t}$ is the ending time of any iteration of the previous interval. Hence, at the end of any iteration i in the interval, the optimal offline service cost equals the optimal service cost and thus (6.2) remains satisfied. Consequently it implies that at most one movement by optimal offline algorithm is sufficient during the interval. This concludes that the number of movements by any optimal offline algorithm is at most $\mathcal{I}/k$ in this case. □

Let t be the ending time of $(c \cdot \max\{k, n-k\})$-th iteration for any integer $c \geq 1$. Using Corollary 6.3 and Claim 6.4

$$M^{\mathcal{A}}_t \geq \max\{n-k, k\} \cdot M^{\mathcal{O}}_t \geq \frac{n}{2} \cdot M^{\mathcal{O}}_t.$$

Thus the claim of the theorem holds. □

6.3 Chapter Notes

The first question related to the movement problem studied in this chapter is that whether the result provided by Theorem 6.1 is tight.

Inspired by the resemblances between the movement problem introduced in this chapter and the paging problem (i.e., the k-server problem with uniform distances), a natural direction would be to study randomized online algorithms for the problem against oblivious adversaries.

Bibliography

[1] I. Abraham, D. Dolev, and D. Malkhi. Lls: a locality aware location service for mobile ad hoc networks. In *Proceedings of the International Workshop on Foundations of Mobile Computing*, pages 75–84, 2004.

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

[3] N. Alon, G. Kalai, M. Ricklin, and L. Stockmeyer. Lower bounds on the competitive ratio for mobile user tracking and distributed job scheduling. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 334–343, 1992.

[4] S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

[5] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating systems: Three easy pieces (Chapter: Paging)*. Arpaci-Dusseau Books, 2015.

[6] V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Munagala, and V. Pandit. Local search heuristics for k-median and facility location problems. *SIAM Journal on Computing*, 33(3):544–562, 2004.

[7] H. Attiya, V. Gramoli, and A. Milani. A provably starvation-free distributed directory protocol. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 405–419, 2010.

[8] H. Attiya and J. Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

[9] G. Ausiello, E. Feuerstein, S. Leonardi, L. Stougie, and M. Talamo. Algorithms for the on-line travelling salesman. *Algorithmica*, 29(4):560–581, 2001.

[10] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive distributed file allocation. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 164–173, 1993.

[11] B. Awerbuch, Y. Bartal, and A. Fiat. Distributed paging for general networks. *Journal of Algorithms*, 28(1):67–104, 1998.

[12] B. Awerbuch, S. Kutten, and D. Peleg. Competitive distributed job scheduling. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 571–580, 1992.

[13] B. Awerbuch and D. Peleg. Online tracking of mobile users. *Journal of the ACM (JACM)*, 42(5):1021–1058, 1995.

[14] B. Awerbuch and D. Peleg. Sparse partitions. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 503–513, 1990.

[15] Y. Azar, A. Ganesh, R. Ge, and D. Panigrahi. Online service with delay. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 551–563, 2017.

[16] N. Bansal, N. Buchbinder, A. Madry, and J. S. Naor. A polylogarithmic-competitive algorithm for the k-server problem. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 267–276, 2011.

[17] Y. Bartal. Probabilistic approximations of metric spaces and its algorithmic applications. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 184–193, 1996.

[18] Y. Bartal and A. Rosen. The distributed k-server problem-a competitive distributed translator for k-server algorithms. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 344–353, 1992.

[19] A. Bjelde, Y. Disser, J. Hackfeld, C. Hansknecht, M. Lipmann, J. Meißner, K. Schewior, M. Schlöter, and L. Stougie. Tight bounds for online tsp on the line. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 994–1005. Society for Industrial and Applied Mathematics, 2017.

[20] A. Borodin, N. Linial, and M. E. Saks. An optimal on-line algorithm for metrical task system. *Journal of the ACM (JACM)*, 39(4):745–763, 1992.

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

[22] J. Byrka and K. Aardal. An optimal bifactor approximation algorithm for the metric uncapacitated facility location problem. *SIAM Journal on Computing*, 39(6):2212–2231, 2010.

[23] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, 1990.

[24] M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental clustering and dynamic information retrieval. *SIAM Journal on Computing*, 33(6):1417–1440, 2004.

[25] M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental clustering and dynamic information retrieval. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 626–635, 1997.

[26] A. Coté, A. Meyerson, and L. Poplawski. Randomized k-server on hierarchical binary trees. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 227–234, 2008.

[27] J. Current, M. Daskin, D. Schilling, et al. Discrete network location models. *Facility location: applications and theory*, 1:81–118, 2002.

[28] E. D. Demaine, M. Hajiaghayi, H. Mahini, A. S. Sayedi-Roshkhar, S. Oveisgharan, and M. Zadimoghaddam. Minimizing movement. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 258–267, 2007.

[29] M. Demirbas, A. Arora, T. Nolte, and N. Lynch. A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, pages 299–315, 2004.

[30] M. J. Demmer and M. P. Herlihy. The arrow distributed directory protocol. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 119–133, 1998.

[31] E. Dijxstra. Solution of a problem in concurrent programming control. *FACULTAD DE INGENIERÍAS FUNDACI Ó N UNIVERSITARIA LUISAMIG Ó*:50, 1965.

[32] G. Divéki and C. Imreh. Online facility location with facility movements. *Central European Journal of Operations Research*, 19(2):191–200, 2011.

[33] Z. Drezner and H. W. Hamacher. *Facility Location: Applications and Theory*. Springer Science & Business Media, 2004.

[34] J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 448–455, 2003.

[35] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[36] D. Fotakis. Incremental algorithms for facility location and k-median. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 347–358, 2004.

[37] D. Fotakis. Memoryless facility location in one pass. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 608–620, 2006.

[38] D. Fotakis. Online and incremental algorithms for facility location. *ACM SIGACT News*, 42(1):97–131, 2011.

[39] Z. Friggstad and M. R. Salavatipour. Minimizing movement in mobile facility location problems. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 357–366, 2008.

[40] M. Ghaffari and C. Lenzen. Near-optimal distributed tree embedding. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 197–211, 2014.

[41] D. Ginat, D. D. Sleator, and R. E. Tarjan. A tight amortized bound for path reversal. *Information Processing Letters*, 31(1):3–5, 1989.

[42] S. Guha and S. Khuller. Greedy strikes back: improved facility location algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 649–657, 1998.

[43] A. Gupta. Steiner points in tree metrics don't (really) help. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 220–227, 2001.

[44] M. T. Hajiaghayi, M. Mahdian, and V. S. Mirrokni. The facility location problem with general cost functions. *Networks*, 42(1):42–47, 2003.

[45] M. Herlihy. The Aleph toolkit: support for scalable distributed shared objects. In *International Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 137–149, 1999.

[46] M. Herlihy, F. Kuhn, S. Tirthapura, and R. Wattenhofer. Dynamic analysis of the arrow distributed protocol. *Theory of Computing Systems (TCS)*, 39(6):875–901, 2006.

[47] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.

[48] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Competitive concurrent distributed queuing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 127–133, 2001.

[49] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Ordered multicast and distributed swap. *ACM SIGOPS Operating Systems Review*, 35(1):85–96, 2001.

[50] M. Herlihy and M. P. Warres. A tale of two directories: implementing distributed shared objects in java. In *Proceedings of the ACM Conference on Java Grande*, pages 99–108, 1999.

[51] K. Jain, M. Mahdian, E. Markakis, A. Saberi, and V. V. Vazirani. Greedy facility location algorithms analyzed using dual fitting with factor-revealing LP. *Journal of the ACM (JACM)*, 50(6):795–824, 2003.

[52] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

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

[54] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11(2):208–230, 1990.

[55] M. Manasse, L. McGeoch, and D. Sleator. Competitive algorithms for on-line problems. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 322–333, 1988.

[56] M. T. Melo, S. Nickel, and F. Saldanha-Da-Gama. Facility location and supply chain management–a review. *European journal of operational research*, 196(2):401–412, 2009.

[57] A. Meyerson. Online facility location. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, page 426, 2001.

[58] T. Moscibroda and R. Wattenhofer. Facility location: distributed approximation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 108–117, 2005.

[59] M. Naimi and M. Trehel. An improvement of the $\log n$ distributed algorithm for mutual exclusion. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 371–377, 1987.

[60] D. Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, 2000.

[61] D. Peleg and E. Reshef. A variant of the arrow distributed directory with low average complexity. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 615–624, 1999.

[62] Y. Rabinovich and R. Raz. Lower bounds on the distortion of embedding finite metric spaces in graphs. *Discrete and Computational Geometry*, 19(1):79–94, 1998.

[63] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)*, 7(1):61–77, 1989.

[64] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis I. An analysis of several heuristics for the traveling salesman problem. *SIAM journal on computing*, 6(3):563–581, 1977.

[65] S. Shalev-Shwartz. Online learning and online convex optimization. *Foundations and Trends in Machine Learning*, 4(2):107–194, 2011.

[66] G. Sharma and C. Busch. An analysis framework for distributed hierarchical directories. *Algorithmica*, 71(2):377–408, 2015.

[67] G. Sharma and C. Busch. Distributed transactional memory for general networks. *Distributed Computing*, 27(5):329–362, 2014.

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

[69] S. Tirthapura and M. Herlihy. Self-stabilizing distributed queuing. *IEEE Transaction on Parallel and Distributed System (PDS)*, 17(7):646–655, 2006.

[70] J. L. A. van de Snepscheut. Fair mutual exclusion on a graph of processes. *Distributed Computing*, 2(2):113–115, 1987.

[71] B. Zhang and B. Ravindran. Dynamic analysis of the relay cache-coherence protocol for distributed transactional memory. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–11, 2010.

My Publications

This list contains the publications I made while I was a Ph.D. candidate under the supervision of Prof. Dr. Fabian Kuhn.

[1] Ahmadi, Mohamad and Ghodselahi, Abdolhamid and Kuhn, Fabian and Molla, Anisur Rahaman. The cost of global broadcast in dynamic radio networks. In *Proceedings of the 19th International Conference on Principles of Distributed Systems (OPODIS)*, 2016.

[2] A. Ghodselahi and F. Kuhn. Dynamic analysis of the arrow distributed directory protocol in general networks. In *Proceedings of the 31st International Symposium on Distributed Computing (DISC)*, 22:1–22:16, 2017.

[3] A. Ghodselahi and F. Kuhn. Serving online requests with mobile servers. In *Proceedings of the 26th International Symposium on Algorithms and Computation (ISAAC)*, pages 740–751, 2015.