Online and Multi-Agent Approximations for the Traveling Salesperson Problem

vom Fachbereich Mathematik
der Technischen Universität Darmstadt
zur Erlangung des Grades eines

Doktors der Naturwissenschaften
(Dr. rer. nat.)

genehmigte

Dissertation

von

Júlia Baligács

Erstgutachter: Prof. Dr. Yann Disser
Zweitgutachter: Prof. Dr. Yossi Azar

Darmstadt 2025

Online and multi-agent approximations for the traveling salesperson problem

Dissertation von
Júlia Baligács
Darmstadt, Technische Universität Darmstadt

Tag der Einreichung: 15.07.2025
Tag der mündlichen Prüfung: 03.09.2025

Jahr der Veröffentlichung der Dissertation auf TUprints: 2025
URN: urn:nbn:de:tuda-tuprints-314001
URL: https://tuprints.ulb.tu-darmstadt.de/id/eprint/31400

Acknowledgements

First and foremost, I am deeply grateful to my supervisor, Yann Disser, for giving me the opportunity to pursue my PhD in his group. Over the past four years, I have had the pleasure of learning immensely from his expertise and have received valuable support and guidance.

My sincere gratitude goes to Yossi Azar for agreeing to read and evaluate my dissertation, and to Kord Eickmeyer and Ulrich Reif for being part of the examination committee.

I would like to express my appreciation to my collaborators, Andreas Emil Feldmann, Irene Heinrich, Pascal Schweitzer, and Anna Zych-Pawlewicz, for their mathematical guidance and our insightful discussions. It has been a great pleasure to work with them.

I am profoundly grateful to Sofia Brenner, Maximilian Gläser, Annika Jäger, Annette Lutz, Nils Mosis, Linda Thelen, and Lena Volk for proofreading parts of my thesis, for their numerous suggestions for improving the presentation, and for their overall support throughout the final weeks. I would also like to thank my colleagues from the optimization group for creating such a welcoming and enjoyable working atmosphere over the past four years.

My research was made possible through the generous support of Studienstiftung des Deutschen Volkes. I highly appreciate their commitment to supporting PhD students, which goes far beyond financial assistance.

Finally, I wish to thank everyone, within or beyond the world of mathematics, who stood by me throughout this journey. Most importantly, I am deeply grateful to my parents and sisters for their unconditional love, support, and encouragement in pursuing my interest in mathematics.

Abstract

In the classical traveling salesperson problem (TSP), we need to find a shortest possible tour that visits all points of a finite metric space. This constitutes one of the most extensively studied problems in combinatorial optimization. In this thesis, we study the TSP under additional restrictions such as incomplete information or restricted computational complexity. We consider several such settings and study online and offline approximation algorithms for them.

In the online graph exploration problem, a single agent needs to find a TSP tour in an initially unknown weighted graph, which is gradually learned over time. We prove a constant competitive ratio for this problem when restricted to minor-free graphs. This result encompasses and significantly extends the graph classes that were previously known to admit a constant competitive ratio. The main ingredient of our proof is that we find a connection between the particular exploration algorithm Blocking and the existence of light spanners. We also exploit this connection in the opposite direction to construct light spanners of bounded-genus graphs. Moreover, we prove that the competitive ratio of the online graph exploration problem is at least 4, even when restricted to subcubic planar graphs. This improves on the previously best known lower bound of 10/3. Additionally, we study the collaborative tree exploration problem, which is a variant of the exploration problem with multiple agents. We give a slightly improved bound on the competitive ratio of the classical algorithm Yo*.

In the tricolored Euclidean traveling salesperson problem, we are given three sets of points in the Euclidean plane and need to find three non-crossing tours, each covering one of the sets. In this problem, we address restrictions in computational complexity rather than on information. Our main contribution is a polynomial-time $(5/3+\varepsilon)$-approximation algorithm. For this, we generalize Arora's famous PTAS for the Euclidean TSP. One of its key ingredients is the "Patching Lemma", which is known to generalize to two non-crossing tours but not to three or more. We circumvent this issue by combining a conditional patching scheme for three tours and an alternative approach based on a weighted solution for two colors.

In the open online dial-a-ride problem, a single agent is tasked to serve transportation requests arriving online, subject to minimizing the completion time. We introduce the algorithm Lazy and give a tight analysis, proving that its competitive ratio is 2.457 on general metric spaces, and 2.366 on the half-line. This improves on the previously best known bound of 2.696 in these metric spaces.

Zusammenfassung

Im klassischen Problem des Handlungsreisenden (TSP) soll eine kürzestmögliche Rundtour gefunden werden, die alle Punkte eines endlichen metrischen Raums besucht. Es zählt zu den am intensivsten untersuchten Problemen der kombinatorischen Optimierung. In dieser Arbeit betrachten wir Varianten des TSP unter zusätzlichen Einschränkungen, etwa unvollständiger Information oder begrenzter Rechenressourcen. Wir betrachten verschiedene solcher Szenarien und analysieren Online- und Offline-Approximationsalgorithmen.

Im Online-Graphenexplorationsproblem muss ein Agent eine TSP-Tour in einem anfangs unbekannten, gewichteten Graphen finden, der während der Exploration sukzessive erlernt wird. Wir beweisen, dass der kompetitive Faktor des Problems konstant ist, wenn es auf minorenfreie Graphen eingeschränkt wird. Dieses Resultat umfasst und erweitert die Graphenklassen, für die bisher ein konstanter kompetitiver Faktor bekannt war, signifikant. Die Kernidee des Beweises ist ein Zusammenhang zwischen dem Explorationsalgorithmus BLOCKING und der Existenz leichter Spanngraphen. Diese Verbindung nutzen wir auch in umgekehrter Richtung, um leichte Spanngraphen für Graphen mit beschränktem Genus zu konstruieren. Darüber hinaus zeigen wir, dass der kompetitive Faktor des Online-Graphenexplorationsproblems mindestens 4 beträgt, auch wenn das Problem auf subkubische planare Graphen eingeschränkt wird. Dies verbessert die bisherige untere Schranke von $10/3$. Zudem untersuchen wir das kollaborative Explorationsproblem, eine Variante mit mehreren Agenten, und geben hierbei eine leicht verbesserte Schranke für den kompetitiven Faktor des klassischen Algorithmus Yo* an.

Im dreifarbigen Euklidischen TSP sind drei Punktmengen in der Euklidischen Ebene gegeben, und es müssen drei sich nicht kreuzende Rundtouren gefunden werden, die jeweils eine der Mengen abdecken. Anstelle der Informationsverfügbarkeit betrachten wir hier Einschränkungen bezüglich der Berechnungskomplexität. Unser Hauptresultat ist ein $(5/3+\varepsilon)$-Approximationsalgorithmus mit polynomialer Laufzeit. Hierfür verallgemeinern wir Aroras berühmtes PTAS für das Euklidische TSP. Eine seiner Kernideen ist das sogenannte "Patching Lemma", das sich auf zwei nicht kreuzende Touren verallgemeinern lässt, jedoch nicht auf drei oder mehr. Wir umgehen dieses Problem, indem wir entweder ein bedingtes Patching-Schema für drei Touren anwenden oder einen alternativen Ansatz auf Basis einer gewichteten Lösung für zwei Farben verwenden.

Im offenen Online-Dial-a-Ride-Problem müssen online eintreffende Transportanfragen so bedient werden, dass die Gesamtabschlusszeit minimiert wird. Wir führen

dafür den Algorithmus LAZY ein und geben eine scharfe Schranke für seinen kompetitiven Faktor an: 2.457 in allgemeinen metrischen Räumen sowie 2.366 auf der Halbgeraden. Damit verbessern wir die bisher beste bekannte Schranke von 2.696 in diesen metrischen Räumen.

Contents

Chapter 1

Introduction

In the classical *traveling salesperson problem* (TSP), we are given a finite set of cities along with the distances between each pair. The task is to find a shortest route that visits all of them and returns to the starting city. The TSP is one of the most famous and extensively studied problems in combinatorial optimization [5, 17, 70, 76, 79, 111], and the strategies developed to solve it are crucial in logistics, planning, and vehicle routing. Importantly, the problem is NP-hard [78], that is, no polynomial-time algorithm exists that computes an optimal tour in general, unless P=NP. This connection to foundational questions such as P vs NP, together with its elegant formulation and wide applicability, makes the TSP a central and particularly intriguing problem in discrete mathematics and theoretical computer science.

In practice, one often faces additional challenges such as incomplete information or limited computational resources. This can make finding an optimum solution impossible. For example, one might not have access to a complete or correct map of the cities. Even when full information is available, the inherent computational hardness of problems like the TSP often makes finding optimum solutions within an acceptable time infeasible. In such scenarios, it is reasonable to seek an *approximation* instead, that is, a route that visits all cities and may not be optimal, but whose length is close to that of an optimal tour.

In this work, we study approximation guarantees for the TSP when optimal solutions cannot be found due to various restrictions. Specifically, we study two types of additional challenges for the TSP. On the one hand, we consider limitations on the computational power, specifically on time complexity. For this, we investigate a variant of the TSP with multiple agents in the Euclidean plane, that is, the cities lie in $\mathbb{R}^2$ and their distances are given by the corresponding Euclidean distance. We analyze approximation guarantees for this problem achievable by polynomial-time algorithms. On the other hand, we study restrictions in information, considering scenarios where the problem instance is revealed piece by piece and decisions must be made without knowledge of the full instance. Specifically, we consider two such settings: one where no complete map is available and only local information about the problem's structure can be accessed, and another involving navigation strategies

for a taxi driver responding to spontaneous transportation requests.

We measure the quality of an approximation as follows. Consider an optimization problem where the objective is to minimize some cost function. In the case of TSP, this is the length of a tour. For an instance I, we denote by $\textsc{Opt}(I)$ the cost of an optimum solution. Given $\alpha \in \mathbb{R}_{\geq 1}$, a valid solution for I is an *α-approximation* if its cost is at most $\alpha \cdot \textsc{Opt}(I)$. An *$\alpha$-approximation algorithm* is an algorithm that, for every possible input instance, produces an α-approximation. When considering problems with full information but restricted time complexity, we are typically looking for the smallest possible value for $\alpha \geq 1$ such that there exists a polynomial-time α-approximation algorithm.

Optimization problems with incomplete information fall into the field of *online optimization*. More precisely, in an *online (minimization) problem*, the input instance is revealed piece by piece, and typically, after each new piece is revealed, we have to make an irrevocable decision, incurring some cost. The objective is to minimize the total incurred cost. The challenge in developing a good online algorithm is to efficiently handle the revealed instance while also anticipating future scenarios for the unrevealed part.

We measure the quality of a deterministic online algorithm by the worst-case approximation factor that it achieves, using a framework called *competitive analysis*. More formally, we denote by $\textsc{Opt}(I)$, similarly as before, the cost of an optimum solution for instance I, when full information is available. In the context of online algorithms, this is referred to as the *offline optimum cost*. When we refer to *the offline optimum*, we mean an offline algorithm that, given an instance I with complete information, computes a valid solution of cost $\textsc{Opt}(I)$. In contrast, an online algorithm Alg receives information gradually over time, and we denote its incurred cost on instance I by $\textsc{Alg}(I)$. For $\rho \geq 1$, we say that Alg is *ρ-competitive* if there exists a constant $C \in \mathbb{R}_{\geq 0}$ such that

$$\textsc{Alg}(I) \leq \rho \cdot \textsc{Opt}(I) + C$$

for every instance I. If $C = 0$ is possible, we say that Alg is *strictly ρ-competitive*. The *(strict) competitive ratio* of the algorithm is defined as

$$\inf\{\rho \geq 1 : \textsc{Alg} \text{ is (strictly) } \rho\text{-competitive}\},$$

and the *(strict) competitive ratio* of the problem is defined as

$$\inf\{\rho \geq 1 : \text{there exists a (strictly) } \rho\text{-competitive algorithm}\}.$$

While the strict competitive ratio of a problem can, in general, be larger than its competitive ratio, in all online problems considered in this work, these values coincide. Therefore, we often refer simply to the competitive ratio.

1.1 Variants of TSP and our results

We now introduce the online and offline variants of the traveling salesperson problem that we study in this work and give an overview of the main results.

Single-agent online graph exploration

Consider a cartographer tasked with creating a map of all towns in a rural state. For better orientation, each town's residents have placed a road sign on every road leaving their town. The signs indicate the name and the distance to the next town down the road. This is the only information available, and the cartographer can read a road sign only when visiting the corresponding town. We assume they can take notes and never forget learned information. The cartographer wants to complete the task as efficiently as possible, minimizing the total traveled distance, including the distance to return to the starting position. In a more formal way, we think of the state as a graph, with the vertices representing the towns and the edges representing the roads.

This is called the *online graph exploration problem* and is formally defined in Chapter 2. Apart from cartography, natural applications include navigation of robots or searching data structures. The key question that we study is the following: Is there a constant-competitive algorithm for the online graph exploration problem? In other words, is there a strategy for the cartographer such that their total traveled distance is at most a constant factor times the length of a shortest TSP tour? This question was proposed in 1994 [75] and despite extensive research [23, 44, 47, 58, 68, 81, 92, 96, 107], it is still open.

In this work, we make progress on this problem in the following respects. First, we prove that on a large class of graphs, the so-called minor-free graphs, the problem admits a constant-competitive algorithm (Section 2.2). This result subsumes and significantly extends other graph classes previously known to admit a constant-competitive algorithm, such as planar [75] and bounded-genus graphs [92]. The main ingredient is a newfound connection between the performance of the particular exploration algorithm Blocking and the existence of so-called light graph spanners, which, roughly speaking, provide a strategy for approximating the distances between the towns using fewer roads. We complement this by giving lower bounds for the algorithm Blocking (Section 2.4) and use the connection in the opposite direction to give improved results for the existence of light graph spanners (Section 2.3).

Second, we give a lower bound of 4 on the competitive ratio of the online graph exploration problem (Section 2.5), improving on a previously known lower bound of $10/3$ [23]. An important ingredient for this is that we identify several restrictions on the agent's behavior that do not affect the competitive ratio (Section 2.1.3). As a byproduct, we also identify several graph properties that can be assumed without loss of generality.

Collaborative tree exploration

Consider the same scenario as before, but with a team of cartographers collaborating to create the map. We assume that each cartographer carries a walkie-talkie, allowing them to share all gathered information. For simplicity, we further assume that there is a central coordinator to whom all cartographers report their findings, and this coordinator determines the movements of each team member. We assume

that each cartographer can move with unit speed, and the objective in this setting is to minimize the overall completion time, rather than the total traveled distance. Thus, having team members remain stationary provides no benefit in decreasing the cost.

Finding a good strategy to coordinate the team is a highly non-trivial problem, even when the underlying graph is as simple as an unweighted tree. For instance, consider a situation where there are two roads leading out of the starting town, and the team consists of ten cartographers. A straightforward approach would be to send five cartographers down each road. However, if one of these roads turns out to lead to a dead end, in hindsight, it would have been preferable to send only a single team member there. More generally, the objective is to allocate more cartographers to regions with more towns, but determining which areas have more towns is challenging when information is only revealed during the course of exploration. For this reason, research typically revolves around the case of unweighted trees [37, 38, 41, 43, 56, 67, 98], which is called the *collaborative tree exploration problem* and is formally defined in Chapter 3.

In this work, we study a classical algorithm for the problem, called Yo* [98]. The main idea in this algorithm is to partition the tree into layers of smaller depth and then recursively apply an algorithm that performs well on trees of small depth. We give a refined version of this approach called RecYoYo and prove that it achieves a slightly better competitive ratio (Section 3.2.2), where our bound is independent of the number of vertices.

Colored Euclidean TSP

Next, we consider an offline variant of the TSP, which is a fundamental problem in geometric network optimization [94]. Given a set of factories, each requiring one of k available goods, we need to establish k roundtrip supply routes to serve them. To avoid constructing bridges, we adopt the restriction that these routes must be non-crossing. The goal is to minimize the total length of the routes.

More formally, in the *k-colored Euclidean traveling salesperson problem*, we are given a finite set of points in the Euclidean plane, each assigned one of k colors. For each color, we have to find a tour that covers all points of that color, with the objective of minimizing the sum of the tour lengths. Importantly, we require these tours to be non-crossing. Note that, without the non-crossing requirement, the problem would decompose into k instances of the Euclidean TSP that can be solved independently. However, the non-crossing constraint introduces significant complexity: Even small modifications to one tour can necessitate major changes in others, creating strong interdependencies across tours.

In this setting, we have full information about all point locations and colors beforehand, but are restricted to solutions computable in polynomial time. Note that the problem is NP-hard, which follows directly from the fact that the Euclidean TSP with a single agent is a special case. In this work, we examine a well-known approximation scheme by Arora for the single-agent Euclidean TSP [5] and study to

what extent it generalizes to the multi-agent setting. While it has been recently shown that Arora's approach can be extended to two colors [45], we will see that it does not generalize to three or more colors. For the specific case of three colors, we propose an alternative method building upon Arora's framework, resulting in a (5/3)-approximation algorithm (Chapter 4).

Online dial-a-ride

Last, we consider the problem of navigating a taxi that receives transportation requests over time in an online fashion. In contrast to the other online problems studied here, we assume time to be continuous, and at any point in time, a transportation request can appear, consisting of a starting point and a destination. We assume that the driver travels at unit speed and the objective is to minimize the completion time. This is a classical and well-studied problem in online optimization [21, 22, 24, 86, 83, 90, 89] and is called *online dial-a-ride*. A formal definition is given in Chapter 5.

The challenge for the taxi driver is not only to decide in which order to serve requests, but also when to serve them. It can often be beneficial for the driver to wait some amount of time before starting a journey. For example, assume that the underlying metric space is the real line, the driver is located at 0, and they receive a transportation request from 1 to 2. If the driver decides to immediately serve this request, then upon reaching point 1, a new request from 0 to 1 may appear. In hindsight, it would have been better to wait at position 0 until the second request appears.

In this work, we introduce an algorithm called Lazy for the online dial-a-ride problem that anticipates situations as the one described above and may choose to wait, even when unserved requests are available. We give a tight analysis of this algorithm, proving that its competitive ratio is 2.457 on general metric spaces (Chapter 5). This improves on the previously best known bound of 2.696 [21] on the competitive ratio of the problem. In addition, we also obtain improvements when the metric space is the half-line.

We remark that the results presented in Sections 2.2–2.4 were published in [12], the results in Chapter 4 were published in [11], and the results in Chapter 5 were published in [13, 14]. The results presented in Section 2.5 are currently under review.

Chapter 2

Single-agent online graph exploration

In this chapter, we study the *online graph exploration problem* proposed by Kalyanasundaram and Pruhs [75]. In this setting, a single agent has to traverse an initially unknown, undirected, connected graph $G = (V, E, w)$ with non-negative edge weights $w\colon E \to \mathbb{R}_{\geq 0}$. We assume that every vertex and every edge has a unique identifier. Upon visiting a vertex for the first time, the agent learns the identifiers of the adjacent vertices as well as the identifiers and weights of the corresponding edges. The *cost* incurred when traversing an edge is simply its weight. The objective is to visit all vertices and return to the starting position, while minimizing the total cost. An example is illustrated in Figure 2.1.

The arguably most important question for this problem is the following.

Problem 2.1 (Kalyanasundaram and Pruhs 1994)**.** *Is there a deterministic constant-competitive algorithm for online graph exploration?*

Several algorithms were proposed with a competitive ratio of $\mathcal{O}(\log(n))$ [92, 107], where n is the number of vertices, but better competitive ratios are only known for restricted classes of graphs, such as so-called bounded-genus graphs and graphs only using a constant number of different weights [75, 92]. Prior to our work, the best known lower bound on the competitive ratio was 10/3 [23]. In particular, the question of Kalyanasundaram and Pruhs remains open.

In this chapter, we make progress on the online graph exploration problem in two respects. First, we extend the classes of graphs admitting a constant-competitive algorithm to so-called minor-free graphs (formally defined in Section 2.1.1). More precisely, we prove the following result.

Theorem 2.2. *For every graph H, there is a constant c such that there exists a c-competitive algorithm for online graph exploration on H-minor-free graphs.*

Second, we improve on the lower bound on the competitive ratio as follows. Here, a graph is *subcubic* if all vertices have degree at most 3.

Theorem 2.3. *The competitive ratio of the online graph exploration problem is at least 4, even when restricted to subcubic planar graphs.*

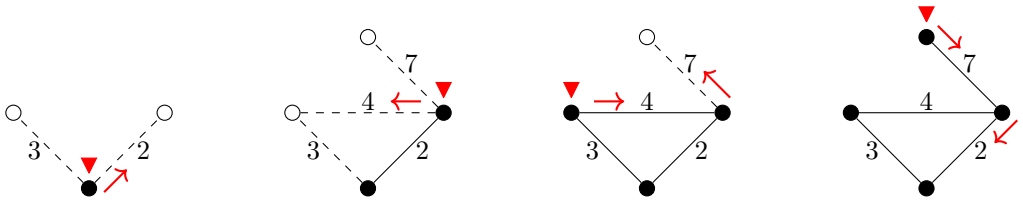

Figure 2.1: Example of an online graph exploration instance with four vertices. Filled vertices are explored, unfilled vertices are learned but not yet explored, and the dashed edges indicate boundary edges (see Section 2.1 for a definition). The red triangle marks the agent's current position, and red arrows indicate the edges it will traverse next. In the third step, two edges need to be traversed to reach the unexplored vertex. In the final step, all vertices are explored, and the agent returns to the starting position. Vertex and edge identifiers are omitted in the figure. In this example, the online agent incurs a cost of 26, whereas the offline optimum cost is $3 + 4 + 2 \cdot 7 + 2 = 23$.

Chapter outline. In Section 2.1, we establish all preliminaries necessary for our further study of the online graph exploration problem. Specifically, we introduce the required notation, define the graph classes on which we consider the problem, provide an overview of existing algorithms for online graph exploration, and show that several further assumptions can be made on the setting without loss of generality. Next, in Section 2.2, we prove our main result (Theorem 2.2). The key ingredient is a new-found connection between the exploration algorithm BLOCKING and the existence of so-called light spanners. In Section 2.3, we demonstrate how this connection can also be used in the opposite direction to derive improved bounds for the existence of graph spanners. In Section 2.4, we establish lower bounds for the algorithm BLOCKING, which show that Theorem 2.2 cannot be extended to the class of all graphs. Finally, in Section 2.5, we study lower bounds for general algorithms and prove Theorem 2.3.

2.1 Preliminaries

Let us introduce the notation used throughout this chapter. We consider weighted graphs of the form $G = (V, E, w)$, where $w\colon E \to \mathbb{R}_{\geq 0}$ assigns every edge a non-negative weight. We write $n := |G| := |V|$, $V(G) := V$, and $E(G) := E$. By slight abuse of notation, we often identify subgraphs of G with their sets of edges. For example, this allows us to work with intersections and unions of subgraphs.

During the course of exploration, we say that a vertex is *explored* if it was visited by the agent and it is *learned* if one of its neighbors or the vertex itself was visited, and thus the agent knows of its existence. An edge is a *boundary edge* if one of its endpoints is explored and the other endpoint is unexplored. By convention, we denote boundary edges by $e = (u, v)$ with u explored and v unexplored. Otherwise, we denote edges by $e = \{u, v\}$ as customary for undirected graphs. For an algorithm ALG, we denote by $\textsc{Alg}(G, v)$ the total cost of the algorithm incurred on

graph G when starting on vertex v. For two learned vertices u and v, we denote by $d(u, v)$ the length of a shortest path from u to v whose internal vertices are explored. In other words, this is the smallest upper bound on the distance between u and v in G that the agent is aware of. In particular, $d(u, v)$ may decrease over time, but never increase.

By $\textsc{Opt}(G)$, we denote the length of a shortest TSP tour, i.e., the cost incurred by the offline optimum. Note that the length of a shortest TSP tour is independent of the starting position, but this does not hold for an online algorithm in general. By $\mathrm{MST}(G)$, we denote the total weight of a minimum spanning tree of G. Note that, for every graph G, we have

$$\mathrm{MST}(G) \leq \textsc{Opt}(G) \leq 2 \cdot \mathrm{MST}(G).$$

In particular, if we have $\textsc{Alg}(G, v) \leq \rho \cdot \mathrm{MST}(G)$ for every graph G and starting vertex v, we obtain that the algorithm is strictly ρ-competitive. Conversely, if we have $\textsc{Alg}(G, v) > \rho \cdot \mathrm{MST}(G)$ for some choice of G and v, the strict competitive ratio of the algorithm is at least $\rho/2$. Therefore, we often estimate an algorithm's cost using a minimum spanning tree instead of the offline optimum.

2.1.1 Restrictions to graph classes

To make the question by Kalyanasundaram and Pruhs more tractable, a classical approach is to restrict the exploration problem to some *graph class* $\mathcal{G}$, where a graph class is simply a (typically infinite) set of graphs. For example, it is known that the competitive ratio of online graph exploration is 2 when restricted to unweighted graphs (i.e., all edges have the same weight, which we can assume to be 1) and $(1+\sqrt{3})/2$ on cycles [96]. While the first property provides a restriction on the edge weights, the second property provides a restriction on the underlying graph independent of the edge weights. Another way of formulating the latter result is as follows: There is an algorithm that is $(1+\sqrt{3})/2$-competitive on every cycle with every possible choice for the edge weights.

Interestingly, it is possible to give strong approximation guarantees for online graph exploration by only assuming some very general properties on the underlying graph and allowing every possible choice for the edge weights. It is a non-trivial problem to identify such properties. For examble, we will see later that restricting exploration to graphs of maximum degree 3 does not simplify the problem (Corollary 2.6) but restricting to planar graphs does. In this subsection, we introduce some graph classes that we study later in this respect. Note that the metric space induced by a weighted graph can be modeled by a complete graph by simply setting edge weights of non-present edges to be sufficiently large. Therefore, we consider graph classes that exclude complete graphs, as otherwise, the problem remains as hard as on general graphs. For example, it makes sense to assume some sparsity properties.

A *planar* graph is a graph that can be embedded into the plane. Less formally speaking, this means that the graph can be drawn in the plane without crossing

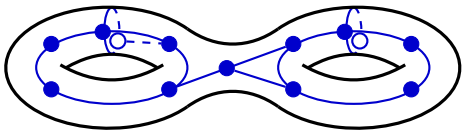

Figure 2.2: A graph embedded on an orientable surface of genus 2.

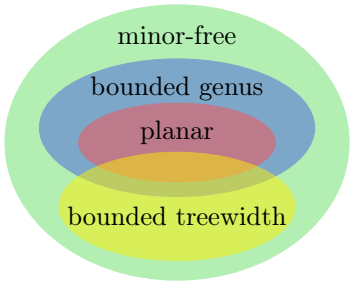

Figure 2.3: Overview of graph classes.

edges. For example, trees and cycles are planar. The smallest non-planar graph is the complete graph on five vertices K_5.

A generalization of planar graphs are *bounded-genus graphs*. Here, the *genus* of a graph G is the smallest integer $g \geq 0$ such that G can be embedded on an orientable surface of genus g, i.e., can be drawn on a torus with g handles without crossing edges (see Figure 2.2). It can be shown that, for every graph, such a value for g exists and we have $g \leq \lceil (n-3)(n-4)/12 \rceil$ for $n \geq 3$, where the bound is tight for complete graphs [104]. Since the plane has genus 0, we obtain that planar graphs are precisely the graphs of genus 0, or in other words, the graphs of genus at most 0. The term *bounded-genus graphs* should be understood here as a family of graph classes, i.e., it collectively refers to the graph classes of genus at most k for every $k \in \mathbb{N}_0$. For example, when we say that an algorithm is constant-competitive on bounded-genus graphs, we mean that, for every k, there exists a constant c (depending on k) such that the algorithm is c-competitive on graphs of genus at most k.

A further generalization is provided by *minor-free graphs*. A *minor* of a graph G is obtained by a sequence of edge deletions, vertex deletions, and edge contractions. Here, a *contraction of an edge* $\{u, v\}$ is obtained by introducing a new vertex w, letting all vertices adjacent to u or v be adjacent to w, and deleting u and v. For a fixed graph H, the class of *H-minor-free* graphs is the set of all graphs that do not contain H as a minor. For example, it is easy to see that the K_3-minor-free graphs are precisely the forests: If a graph contains a cycle, we can delete every vertex outside of the cycle and then contract edges until we obtain a cycle of length 3, i.e., K_3 is a minor of the graph. Further, it is easy to see that applying a vertex deletion, edge deletion, or edge contraction does not increase the genus of a graph. Therefore, if H is a graph of genus g, every graph of genus at most $g-1$ is H-minor-free. Thus, the class of H-minor-free graphs contains the class of graphs of genus at most $g-1$. Similarly, note that our main result (Theorem 2.2) can be reformulated as follows: Let $\mathcal{G}$ be a non-trivial graph class (i.e., not the class of all graphs) that is closed under taking minors. Then there exists a constant c such that there exists a c-competitive algorithm for online graph exploration on graphs in $\mathcal{G}$.

More generally, one can precisely describe every graph class $\mathcal{G}$ that is closed under taking minors by giving a set of excluded minors $\mathcal{H}$. This means that $G \in \mathcal{G}$ if and only if G does not contain any of the graphs in $\mathcal{H}$ as a minor. For example,

one can simply set $\mathcal{H}$ to be the set of all graphs that are not contained in $\mathcal{G}$. A groundbreaking result by Robertson and Seymour states that $\mathcal{H}$ can always be chosen to be finite [105]. A prominent special case is Kuratowski's theorem [87], which states that the planar graphs are precisely the graphs that exclude the minors K_5 and $K_{3,3}$.

The term *minor-free graphs* should again be understood as a family of graph classes, i.e., it collectively refers to the H-minor-free graph classes for every graph H. When we say that an algorithm is constant-competitive on minor-free graphs, we mean that, for every graph H, there exists a constant c (depending on H) such that the algorithm is c-competitive on H-minor-free graphs.

Finally, we remark that there are numerous other graph classes one could consider that lie outside the scope of this work. For instance, one could consider graphs with some bounded width parameter such as treewidth, pathwidth or tree-depth. However, these classes are all contained in minor-free graph classes (Figure 2.3). An example of a more general class of graphs that still excludes complete graphs is given by so-called *graphs of bounded expansion.*

2.1.2 Algorithms for online graph exploration

In this subsection, we survey algorithms for the online graph exploration problem from the literature, providing an overview of the state-of-the-art. We put a particular emphasis on their performance guarantees on different graph classes. An overview is given in Table 2.1.

Nearest Neighbor (NN). A natural strategy for online graph exploration is the Nearest Neighbor algorithm (NN), sometimes also referred to as the *greedy algorithm.* In this algorithm, in each step, the agent greedily selects a closest unexplored vertex to its current location. Note that even though distances may decrease during the course of exploration, for NN it makes no difference whether distances are measured in the partially explored graph or in the final graph: If there are no edges of weight 0, all vertices closest to the agent's position are indeed learned because, if a vertex v is not yet learned, the shortest path from v to the agent's location contains an even closer unexplored vertex. If there are edges of weight 0, we obtain that at least one such vertex is learned.

This algorithm has been extensively studied as a simple approach for approximating TSP, long before the online graph exploration problem was introduced. Rosenkranz, Stearns, and Lewis showed that its approximation ratio, i.e., its competitive ratio, is in $\Theta(\log n)$ [107]. Interestingly, the lower bound of $\Omega(\log n)$ is already achieved on trees [58], and on unweighted ladder graphs, assuming that ties are broken adversarially [68]. In particular, the competitive ratio of NN is at least $\Omega(\log n)$ on planar graphs.

Depth-First Search (DFS). Depth-first-search is a well-studied classical offline algorithm for traversing a connected unweighted graph. Here, the agent chooses in

	NN	DFS	hDFS [92]	BLOCKING$_\delta$
general	$\Theta(\log n)$ [107]	∞	$\Theta(\log n)$	$\Omega(n^{1/(\delta+4)})$
unweighted	$\Theta(\log n)$ [68]	2 [96]	2	2
k weights	$\Theta(\log n)$	∞	$\leq 2k$	$\Omega(n^{1/(\delta+4)})$ [92]
trees	$\Theta(\log n)$ [58]	1	$\Theta(\log n)$	$\Theta(1)$
planar/minor-free	$\Theta(\log n)$	∞	$\Theta(\log n)$	$\Theta(1)$ [75, 92] (Thm. 2.7)

Table 2.1: Overview of the performance of different algorithms on different graph classes. Results without a citation are either inherited (e.g., planar for NN) or folklore (e.g., DFS on trees). For hDFS, all results were proven in the same paper, which is cited in the header. For planar, bounded-genus, and minor-free graphs, the same bounds are known, which are summarized in the last line.

every step a neighboring unexplored vertex (ignoring the edge weights) or backtracks if none exists. This strategy is obviously also applicable in the online setting, as its execution only requires information available to the agent.

Note that DFS is 1-competitive on trees. Moreover, it is 2-competitive on unweighted graphs: To see this, observe that the cost of DFS is $2(n-1)$ and the weight of an MST is $n-1$. In [96], it was shown that no algorithm with a better competitive ratio on unweighted graphs exists. In particular, DFS outperforms NN on the class of unweighted graphs since the competitive ratio of NN on this class is $\Omega(\log n)$ [68]. More generally, the *aspect ratio* of a graph G is $\max\{w(e)/w(e') : e, e' \in E(G)\}$ and it is ∞ if G contains an edge of weight 0. Note that DFS is $2a$-competitive on graphs where the aspect ratio is bounded by $a \geq 1$: This follows from the fact that the cost of DFS is at most $2a$ times the cost of an MST.

However, note that DFS does not outperform NN in general. For the general problem, the competitive ratio of DFS cannot even be bounded by a graph parameter independent of the edge weights: To see this, consider a cycle with a single heavy edge e and all other edges having weight 1. Choosing a suitable starting position, DFS will always traverse the heavy edge. Letting $w(e) \to \infty$, this shows that the competitive ratio of DFS cannot be bounded by any function independent of the edge weights.

Hierarchical Depth-First Search (hDFS). In the graph that is to be explored, let $\text{comp}(w, u)$ denote the connected component of vertex u in the subgraph only containing edges of weight at most w. The key idea of the algorithm hDFS is to choose, for current location u, the smallest weight w such that $\text{comp}(w, u)$ contains a boundary edge and explore $\text{comp}(w, u)$ using DFS. Whenever a smaller weight with that property is found, the execution of DFS is interrupted and we begin a new execution of the procedure using the smaller weight. This algorithm was introduced by Megow, Mehlhorn, and Schweitzer [92], and the authors prove that hDFS is $2k$-competitive on graphs that only use k distinct edge weights.

Note that rounding edge weights to powers of 2 only distorts every edge weight by at most a factor of 2, and therefore, the cost of an algorithm and the cost of

the offline optimum by a factor of at most 2. For hDFS, this is beneficital because it might reduce the number of distinct edge weights. In [92], the authors prove that hDFS applied on a graph with rounded edge weights is $\Theta(\log n)$-competitive. In this, the lower bound is already achieved on a path.

We remark that, for the development of exploration algorithms, it is often sensible to assume that edge weights are rounded to powers of 2. This distorts the competitive ratio by a factor of at most 4, which is irrelevant for Problem 2.1.

Blocking. Kalyanasundaram and Pruhs introduced the algorithm ShortCut and showed that it is 16-competitive on planar graphs [75]. Later, Megow, Mehlhorn and Schweitzer [92] revisited the algorithm, addressed some technical intricacies and proposed their reinterpretation Blocking$_\delta$, which we also consider in this work. The main idea is to execute DFS ignoring all edges considered too heavy, where the threshold for an edge to be considered too heavy depends on the parameter δ of the algorithm. A full description of Blocking$_\delta$ is given in Section 2.2.2. In [92], the authors expand the result in [75] and show that the algorithm is constant-competitive on bounded-genus graphs. Moreover, they prove that its competitive ratio is at least $\Omega(n^{1/(\delta+4)})$. This implies that, for constant choices of the parameter δ, the performance of Blocking$_\delta$ on general graphs is worse than that of NN or hDFS. Interestingly, their lower bound construction uses only two different edge weights.

In Section 2.2, we further study the algorithm Blocking$_\delta$. We extend the result in [92] and prove that the algorithm is constant-competitive on graphs excluding some fixed minor (Theorem 2.2). We also prove that, if the parameter δ is adapted to the number of vertices using a doubling strategy, the algorithm can be made $\mathcal{O}(\log n)$-competitive on general graphs (Theorem 2.8).

Lower bounds. Prior to our work, the best known lower bound for the graph exploration problem was 10/3 which was shown by Birx, Disser, Hopp, and Karousatou [23]. Their construction builds on a previously known lower bound of 2.5 shown by Dobrev, Královič, and Markou [44]. Since the construction by Birx et al. is planar, the lower bound of 10/3 even holds when the problem is restricted to planar graphs. In this chapter, we further build on the work in [23, 44] and give a planar lower bound construction implying that the competitive ratio is at least 4 (Theorem 2.3). We also prove that the construction can be made subcubic.

Other related work. In addition to the above, the online graph exploration problem has been studied on a variety of specific graph classes. These include tadpole graphs [31], unicyclic graphs [58, 81], and cactus graphs [58]. Another approach to tackle the exploration problem was given in [47], where the authors revisited NN with learning augmentation.

Several other settings of exploration with a single agent have been studied, such as exploration of directed graphs [1, 40, 54, 55] or when the agent has limited memory [57, 103] but can use pebbles [42].

2.1.3 Equivalent problems

In this subsection, we prove that we can impose several restrictions on the agent's behavior without affecting the competitive ratio of the graph exploration problem. As a byproduct, we identify certain graph classes for which restricting the exploration problem to those classes does not make the problem easier. As a first step, we elaborate on the setting of online graph exploration by studying its "adversary", that is, the problem of creating bad instances for algorithms. The model that we describe here will serve as the foundation for proving our equivalence results and our lower bounds on the competitive ratio (Theorem 2.3).

The adversary model

It is often useful to view an online optimization problem as a game: One player selects an instance and reveals it piece by piece to the other player, who must make online decisions. In this setting, the latter player is the online algorithm, and the first player is referred to as the *adversary*. For deterministic online algorithms, it does not matter whether the adversary chooses the entire instance beforehand and gradually reveals it, or whether the adversary adapts the unrevealed parts based on the algorithm's decisions. By definition, if a deterministic algorithm is presented with the same instance I twice, it will make the same decisions in both runs. If there exists another instance I' that differs only in parts not yet revealed, the algorithm cannot distinguish between I and I', and thus behaves identically on both. Since a ρ-competitive algorithm must be ρ-competitive on every instance, the adversary can choose whether the algorithm is operating on I or I' once the distinction becomes relevant.

Finding strategies for the adversary can be seen as a "dual" online problem, called the *adversary problem*: Here, we have to construct an instance for the given online problem, while receiving information about the algorithm's decisions over time in an online fashion, without being able to modify the parts of the instance already revealed. The objective is to maximize the ratio $\textsc{Alg}(I)/\textsc{Opt}(I)$, where I is the instance construced by the adversary. It is immediate that the optimal value achievable in the adversary problem coincides with the strict competitive ratio of the online problem. Importantly, note that this model of an adaptive adversary is only applicable when considering deterministic algorithms.

In the context of online graph exploration, the adversary problem can be stated as follows: We have to construct a weighted graph G adaptively and the counterplayer is an agent that, in each step, moves to a new unexplored vertex. Once a vertex v is explored, we have to irrevocably determine the identifiers of its adjacent vertices as well as the identifiers and weights of the corresponding edges. The objective is to maximize the ratio of the total distance traveled by the agent and the length of a shortest TSP tour.

Throughout this chapter, we often use this adversarial perspective to prove lower bounds and equivalences between online problems.

Competitive ratio and strict competitive ratio

As a warm-up, we show, using the adversarial perspective, that there is no difference between the competitive ratio and the strict competitive ratio of the online graph exploration problem.

Observation 2.4. *Let $\rho \in \mathbb{R}_{\geq 1}$. There is a ρ-competitive algorithm for online graph exploration if and only if there is a strictly ρ-competitive algorithm.*

Proof. By definition, every strictly ρ-competitive algorithm is also ρ-competitive. For the other direction, assume that there is no strictly ρ-competitive algorithm. Then there exists an adversarial strategy $\mathcal{A}$ that constructs, for every exploration algorithm ALG, a graph G with starting vertex v, such that $\text{ALG}(G, v) \geq \rho \cdot \text{OPT}(G) + \varepsilon$ for some $\varepsilon > 0$. Consider the following adversarial strategy: Let the starting vertex v be adjacent to M vertices $v_1, \ldots, v_M$ via edges of weight 0. Treat each v_i as the starting vertex of a separate exploration instance and, for each, apply strategy $\mathcal{A}$. Let $G_1, \ldots, G_M$ denote the resulting graphs. Note that they are not necessarily equal because the algorithm might choose to follow a different strategy when facing the same situation multiple times and, in our adversarial strategy, we adapt the graph according to the agent's decisions. However, the adversary is able to build a suitable graph for every possible change of strategy.

Observe that we can treat these as M instances of the exploration problem because, during exploration of G_i, the agent does not gain any information about G_j for $j \neq i$. We charge the cost incurred in graph G_i on the i-th problem and let ALG_i denote the strategy followed in graph G_i. We obtain a graph G with

$$\text{ALG}(G, v) = \sum_{i=1}^{M} \text{ALG}_i(G_i, v_i) \geq \rho \cdot \sum_{i=1}^{M} \text{OPT}(G_i) + M\varepsilon = \rho \cdot \text{OPT}(G) + M\varepsilon.$$

Letting $M \to \infty$, this shows that no ρ-competitive algorithm exists. □

In the remainder of the chapter, we therefore only refer to the *competitive ratio* for simplicity.

The restricted online graph exploration problem

We prove that we can impose several restrictions on the agent's behavior without affecting the competitive ratio of the graph exploration problem. This will later be useful for our lower bound construction. In addition, these restrictions can serve as a "sanity check" when developing new algorithms: A sensible algorithm should not rely on any information or capabilities that we restrict in the following.

More precisely, we define the *restricted online graph exploration problem* as the online graph exploration problem, where the agent is additionally restricted, and the adversary is relaxed, as follows.

R1) Upon visiting a new vertex, the agent only learns the unique identifiers and weights of incident edges (but not the identifiers of neighboring vertices).

R2) The graph may have parallel edges.

R3) When the agent traverses a boundary edge $e = (u, v)$, a new boundary edge e' of weight $w(e)$ may appear incident to u.

R4) If there are two boundary edges $e = (u, v), e' = (u', v')$ with $d(u', v) < w(e')$, the agent must not traverse edge e'.

R5) If there are two boundary edges $e = (u, v), e' = (u, v')$ incident to the same vertex u with $w(e) = w(e')$, the adversary may decide which of the two edges the agent traverses first.

Note in the above that none of these restrictions applies to the agent in the offline optimum, that is, the offline optimum is allowed to traverse edge e' in the situation of R4) and, if the adversary decides in situation R5) that the online agent must traverse e' before e, the offline optimum may traverse e before e'.

Although one might expect that these restrictions increase the competitive ratio of the problem, we actually prove that these can be assumed without loss of generality. While some of the next result is folklore (R1 has already been noted in other works [92]), we provide a proof for the sake of completeness. In the following, if no $\rho \in \mathbb{R}$ exists such that the problem is ρ-competitive, we say for convenience that the competitive ratio is ∞ and the result still applies.

Observation 2.5. *If the competitive ratio of online graph exploration is $\rho \in \mathbb{R} \cup \{\infty\}$, the competitive ratio of restricted online graph exploration is also ρ.*

Proof. It is immediate that a ρ-competitive algorithm for the restricted online graph exploration problem is also ρ-competitive for online graph exploration. For the other direction, assume there exists an adversarial strategy $\mathcal{A}$, constructing, for every algorithm for the restricted problem, a multigraph G with starting vertex v such that $\textsc{Alg}(G, v) > \rho \cdot \textsc{Opt}(G)$. We give an adversarial strategy that achieves the same lower bound of ρ in the classical online graph exploration setting. For this, we construct a graph that mimics the multigraph constructed by $\mathcal{A}$ and argue that, in this graph, any movement of an agent for exploration mimics a movement of some agent for restricted exploration.

First, we argue that R1) and R2) can be assumed without loss of generality. The main idea is to replace each vertex v that $\mathcal{A}$ introduces by a suitable graph G_v called a *gadget*, in which we set all edge weights to 0, and connect all edges incident to v to different vertices of G_v (see Figure 2.4). For instance, we can set G_v to be a binary tree with at least d_v leaves, where d_v is the degree of v, and connect the edges to its leaves.

From the perspective of the induced metric space of the resulting graph, all vertices in G_v represent essentially the same vertex v because their pairwise distances are 0. In particular, we can assume that, if the agent explores any vertex of G_v, it immediately explores all vertices of G_v and thus, learns about all edges incident to vertices of G_v. Moreover, this construction does not affect the offline optimum cost.

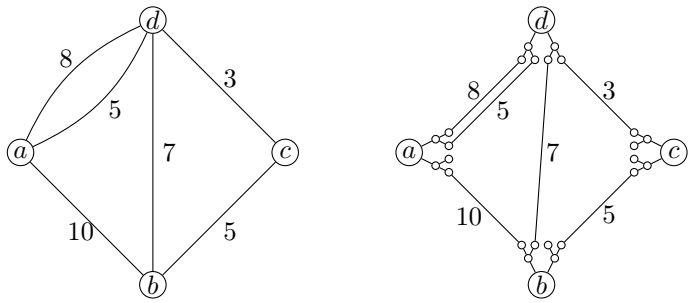

Figure 2.4: Construction for proving that we can assume R1) and R2). All edges without label have weight 0. We have replaced every vertex by a binary tree. If, in the left graph, vertices b and c are explored and vertex d is unexplored, the agent of the classical exploration problem knows that the boundary edges of weight 3 and 7 lead to the same vertex d. In the corresponding situation in the right graph, the agent does not have this information.

The key difference is the following: In the resulting graph, the agent cannot tell whether two boundary edges lead to the same gadget G_v. In other words, learning the identifiers of the endpoints of boundary edges effectively reduces to learning only the identifiers of the boundary edges themselves. Moreover, parallel edges between vertices u and v in the graph constructed by $\mathcal{A}$ can be modeled as multiple edges between leaves of G_u and G_v. This shows that we can assume without loss of generality R1) and R2).

In the following, we argue that the remaining properties can also be assumed without loss of generality using that we have already established this for R1) and R2). Since we can assume R1), we denote boundary edges in the remainder of the proof by $e = (u, \cdot)$. The adversary only needs to decide the identifier of the second vertex once the agent traverses e.

The statement that we can assume R5) now immediately follows from R1): Both edges have the form $e = (u, \cdot)$ of the same weight, so the agent cannot distinguish between them.

For R3), we use the following construction: For every boundary edge $e = (u, \cdot)$ introduced by $\mathcal{A}$, we instead introduce a sufficiently large number of boundary edges of weight $w(e)$, each incident to u, which we call a *bundle*. If the agent traverses e, this corresponds to traversing one of the edges of the corresponding bundle. Upon the traversal of an edge $e = (u, v)$ in the bundle, the adversary can decide whether either, all edges of the bundle lead lead to vertex v, or only e leads to vertex v and all other edges of the bundle lead to other vertices with not yet known endpoints, i.e., they remain boundary edges. Note that the latter corresponds to the additional capability R3) of $\mathcal{A}$.

For property R4), if the agent traverses a boundary edge $e' = (u', \cdot)$ even though a boundary edge $e = (u, \cdot)$ exists with $d(u', u) + w(e) < w(e')$, the adversary lets e' and e have the same endpoint v and it introduces a new boundary edge incident to u' of weight $w(e')$ (using R3). This corresponds to forcing the agent not to traverse e':

If the agent decides otherwise, the outcome remains the same as if it had chosen to traverse e, but it incurs a higher cost. □

The main idea for the proof of Observation 2.5 was to replace each vertex by a binary tree and connect the edges to its leaves (see Figure 2.4). If a graph class is closed under this operation, we say that it is *closed under tree replacement*. This property holds, for example, for the classes of all graphs, planar graphs, bounded-genus graphs, graphs excluding a non-planar minor, and graphs only using a bounded number of different edge weights including 0. Building on the last observation, we obtain equivalence for further variants of the problem, which we summarize in the following result. While one might expect that conditions R1)-R5) increase the competitive ratio, we now introduce additional properties that might be expected to decrease it. However, we show that all of the following properties can be assumed without loss of generality.

Corollary 2.6. *The competitive ratios ($\in \mathbb{R} \cup \{\infty\}$) of the following problems coincide.*

a) online graph exploration

b) restricted online graph exploration,

c) restricted and classical online graph exploration on subcubic graphs,

d) restricted and classical online graph exploration on graphs fulfilling the triangle inequality, that is, $w(\{u,v\}) \leq d(u,v)$ for all $\{u,v\} \in E$,

e) each of the above problems with the relaxation that the agent is given the total number of vertices at the beginning of exploration.

If $\mathcal{G}$ is a graph class closed under tree replacements, the equivalence still holds when restricting all of the problems to $\mathcal{G}$.

Proof. Equivalence of a) and b) is precisely Observation 2.5 and c) follows from the fact that the graphs constructed in the proof of Observation 2.5 are subcubic. For d), assume that an input graph contains an edge $e = \{u,v\}$ not fulfilling the triangle inequality, i.e., $w(e) > d(u,v)$, where the distance is measured here in the final graph. As long as e is a boundary edge, there exists another boundary edge on the shortest path from u to v that is cheaper for the agent to traverse. Due to property R4), e is never traversed so we can assume that the graph does not contain such edges.

For part e), let $\textsc{Alg}_N$ be the strategy that the agent follows when given the information that the final graph consists of N vertices and assume that $\textsc{Alg}_N$ is ρ-competitive on every graph on N vertices. Then $\textsc{Alg}_N$ is also ρ-competitive on every graph G on $n \leq N$ vertices: The adversary can attach to the last explored vertex of G a path of $N-n$ vertices using edges of weight 0 so that the agent technically operates in a graph on N vertices, however, its incurred cost on G equals the incurred

cost in the modified graph. This shows that ALG_N is also ρ-competitive on every graph G on $n \leq N$ vertices. For this reason, we can assume that $\text{ALG}_{N-1} = \text{ALG}_N$ for every $N \in \mathbb{N}_{\geq 2}$. We obtain that we can assume that the agent's strategy is independent on the number of vertices, even when this information is given beforehand.

The last statement follows immediately from the fact that such a graph class is closed under the constructions in Observation 2.5 and under attaching paths. □

Another way of stating the last result is that we can assume (any subset of) the listed properties (i.e., triangle inequality, subcubic, agent learns n, agent fulfills any subset of R1)-R5)) without changing the competitive ratio.

We note that the results in this subsection are not applicable when proving superconstant bounds on the competitive ratio. For example, if one aims to prove that an algorithm has competitive ratio $f(n)$ on graphs on n vertices for some function f depending on n, one loses generality when assuming the above restrictions. This is because the constructions in the proof increase the number of vertices.

2.2 A constant-competitive algorithm on minor-free graphs

In this section, we significantly expand the class of graphs on which the online graph exploration problem is known to admit a constant-competitive algorithm and prove Theorem 2.2. We start by reformulating the statement tailored to the algorithm BLOCKING_δ.

Theorem 2.7. *For every graph H and constant $\delta > 0$, there is a constant c (depending on H and δ) such that BLOCKING_δ is c-competitive on H-minor-free graphs.*

Prior to our work, the largest class of graphs which was known to admit a constant-competitive algorithm was the class of bounded-genus graphs [92].

Previous works only studied BLOCKING_δ for constant choices of the parameter δ, i.e., independent of the number of vertices n. It is known that competitive ratio of the algorithm is at least $\Omega(n^{1/(\delta+4)})$ if δ is a constant [92]. This naturally raises the question of whether improvement is possible if the agent is given n in advance and δ may depend on n. We also obtain the following result for BLOCKING_δ.

Theorem 2.8. *If the agent is given the number of vertices n in advance, the algorithm $\text{BLOCKING}_{\log(n)}$ is $\mathcal{O}(\log(n))$-competitive.*

This shows that $\text{BLOCKING}_{\log(n)}$ achieves the best previously known competitiveness.

The key ingredient in our analysis of BLOCKING_δ is a new-found connection to the existence of so-called light spanners. We begin by introducing this concept.

2.2.1 Graph spanners

Spanners were introduced in 1989 by Peleg and Schäffer [100] and have been instrumental in the development of approximation algorithms, particularly for TSP [5, 29, 30]. Here, a subgraph $H = (V, E')$ of a connected, undirected graph $G = (V, E)$ with edge weights $w\colon E \to \mathbb{R}_{\geq 0}$ is called a $(1+\varepsilon)$-*spanner* of G if $d_H(u, v) \leq (1+\varepsilon)\, d_G(u, v)$ for all $u, v \in V$, where d_H and d_G denote the shortest-path distance in H and G, respectively. Then H has *stretch* at most $(1+\varepsilon)$ and its *lightness* is $w(H)/w(\text{MST})$, where MST denotes a minimum spanning tree of G and, by slight abuse of notation, we write $w(\tilde{H}) := \sum_{e \in E(\tilde{H})} w(e)$ for a graph $\tilde{H}$.

We show that the online graph exploration algorithm Blocking$_\delta$ has a constant competitive ratio on every class of graphs that admits spanners of constant lightness for a fixed stretch. Examples of graph classes where the worst-case lightness does not depend on the number of vertices include planar graphs [4], bounded-genus graphs [60], apex graphs [62], bounded pathwidth graphs [61], bounded treewidth graphs [39], and, encompassing all of these results, minor-free graphs [30]. Our results rely on the existence of light spanners for minor-free graphs [30] and improve on the lightness for bounded-genus graphs (Theorem 2.19). There are also strong bounds for general graphs. Given a graph G with n vertices, an integer $k \geq 1$ and $\varepsilon \in (0, 1)$, G contains a $(2k-1)(1+\varepsilon)$-spanner of lightness $\mathcal{O}_\varepsilon\big(n^{1/k}\big)$ [34], where the notation $\mathcal{O}_\varepsilon$ indicates that the constant factor hidden in the $\mathcal{O}$-notation depends on ε. This will allow us to give an improved bound on general graphs.

In terms of lower bounds on the lightness of spanners, most constructions in the literature are unweighted graphs of high girth, where the girth of a graph is the length of its shortest cycle. For example, it is known that, for every $k \geq 3$, there exists a graph on n vertices with $\Omega(n^{1+1/k})$ edges and girth at least k [95][Theorem 6.6]. In such a graph, no proper subgraph is a $(k-1-\varepsilon)$-spanner for any $\varepsilon > 0$ because the removal of an edge $\{u, v\}$ distorts the distance between u and v by a factor of at least $(k-1)$. Since a (minimum) spanning tree has weight $\Theta(n)$, this implies that the minimum lightness of a $(k-1-\varepsilon)$-spanner in this graph is $\Omega(n^{1/k})$. Even further, a famous conjecture by Erdős asserts that there exist graph of girth at least $2k+1$ with $\Omega(n^{1+1/k})$ edges [49]. This is equivalent to a lower bound of $\Omega(n^{1/k})$ on the best lightness of a $(2k-\varepsilon)$-spanner in unweighted graphs. While this conjecture remains unresolved, a nearly matching upper bound on the spanner lightness was proven in [34].

2.2.2 The algorithm Blocking

In this subsection, we introduce the algorithm Blocking that we will later show to be constant-competitive on minor-free graphs.

Recall that, during the execution of an online graph exploration algorithm, $d(x, y)$ denotes the length of a shortest internally explored path from x to y. In particular, the distance may decrease during execution.

Definition 2.9 (Kalyanasundaram and Pruhs [75])**.** Given some $\delta > 0$, we say that a boundary edge $e = (u, v)$ is *δ-blocked* if there is another boundary edge $e' = (u', v')$ such that $w(e') < w(e)$ and $d(u, v') \leq (1 + \delta)w(e)$.

It is worth noting that not allowing the agent to traverse 0-blocked edges corresponds to condition R4) of restricted online graph exploration.

The rough idea of Blocking$_\delta$ is to perform a depth-first-traversal while ignoring all δ-blocked edges. Whenever a previously blocked edge turns unblocked, the agent moves to and explores one such edge, and initiates a DFS-traversal from its new position. Blocking$_\delta$ is formally specified in Algorithm 1. It is executed on an undirected, weighted, connected, and initially unexplored graph $G = (V, E, w)$ and takes as input a vertex v of G, denoting the current position of the agent. The parameter δ is fixed and not part of the input. The algorithm follows a recursive DFS-like structure and the input of the initial invocation is the start vertex.

Algorithm 1: Blocking$_\delta[v]$ [75, 92]

```
1 while there is a boundary edge e = (y, x) that is not δ-blocked and such
    that y = v or e was previously blocked by some edge (u, v) do
2     traverse a shortest internally explored path from v to y
3     traverse e
4     Blocking_δ[x]
5     traverse a shortest internally explored path from x to v
```

Observe that the algorithm is correct, i.e., every vertex is explored: Assume, for the sake of contradiction, that some vertex remains unexplored when the algorithm terminates, i.e., there are still boundary edges. Let $e = (u, v)$ be a boundary edge of minimum weight. Then e is not δ-blocked. Therefore, either the exploration of u should have triggered the exploration of v, or v should have been explored at the last point in time the edge turned unblocked.

Key properties of Blocking

Throughout the remainder of this section, let $G = (V, E, w)$ be a weighted graph, n be its number of vertices, v the given start vertex of G, and $\delta > 0$. We analyze the performance of Blocking$_\delta$ on G, i.e., we estimate its total cost Blocking$_\delta(G, v)$. To this end, let B be the set of boundary edges taken by Blocking$_\delta$, i.e., the edges traversed during the execution of line 3.

Observation 2.10 (Megow et al. [92])**.** *We have*

$$\textit{Blocking}_\delta(G, v, \delta) \leq 2(\delta + 2)w(B).$$

Proof. We charge all cost incurred in lines 2, 3, and 5 to the corresponding boundary edge $e \in B$. Note that the cost in line 2 is at most $(1 + \delta)w(e)$, because either we have $y = v$ such that $d_G(v, y) = 0$, or e was blocked by an edge (u, v), which

implies $d_G(y, v) \leq (1+\delta)w(e)$. The cost in line 3 is $w(e)$ and the cost in line 5 is at most the sum of the cost in lines 2 and 3. Therefore, each edge e in B is charged at most $2(\delta+2)w(e)$. □

In our subsequent analysis, we will frequently use a minimum spanning tree with a particular property. For this, in what follows, let MST_B be a minimum spanning tree of G that maximizes the number of edges shared with B, i.e.,

$$\mathrm{MST}_B \in \operatorname{argmax}\{|\mathrm{MST} \cap B| : \mathrm{MST} \text{ is a minimum spanning tree of } G\}.$$

As pointed out in [92], cycles in $B \cup \mathrm{MST}_B$ are long relative to the weight of the edges they contain. Specifically, the following holds.[1]

Lemma 2.11. *Let C be a cycle in $B \cup \mathrm{MST}_B$ and e be an edge of C. Then*

$$w(C \setminus \{e\}) > (1+\delta)w(e).$$

Proof. It suffices to show the assertion for an edge of maximum weight in the cycle C. As a first step, we show that such an edge must be in B, i.e., we show that $\operatorname{argmax}\{w(e)\colon e \in C\} \subseteq B$: For the sake of contradiction, assume otherwise and let $e = (u, v) \in \operatorname{argmax}\{w(e)\colon e \in C\} \cap (\mathrm{MST}_B \setminus B)$. Removing e from MST_B separates MST_B into two connected components. In particular, u and v are in different components. Start walking in $C \setminus \{e\}$ from u to v and let e' be the first edge that leads from u's connected component in $\mathrm{MST}_B \setminus \{e\}$ to v's connected component. Then $e' \in B \setminus \mathrm{MST}_B$ and by maximality of e, we have $w(e') \leq w(e)$. Therefore, replacing e by e' in MST_B gives another spanning tree of weight at most $w(\mathrm{MST}_B)$. This new spanning tree has one more edge in common with B than MST_B has. This contradicts the choice of MST_B, so that we can assume from now on $\operatorname{argmax}\{w(e)\colon e \in C\} \subseteq B$. This means that every edge in $\operatorname{argmax}\{w(e)\colon e \in C\}$ is *charged*, i.e., is traversed in some execution of line 3 of the algorithm.

Let $e = (u, v)$ be the edge in $\operatorname{argmax}\{w(e)\colon e \in C\}$ that is charged last. At the time e is traversed, it is a boundary edge, so that u is explored but v is not yet explored. Start walking in $C \setminus \{e\}$ from u to v and let $e' = (u', v')$ be the first edge leading from an explored vertex u' to an unexplored vertex v', i.e., e' is another boundary edge in C (cf. Figure 2.5).

Next, we show that $w(e') < w(e)$: Assume otherwise. By maximality of $w(e)$, this means $w(e') = w(e)$ so that $e' \in \operatorname{argmax}\{w(e)\colon e \in C\}$. But then we also have $e' \in B$ (i.e., e' is charged) because we have already shown that $\operatorname{argmax}\{w(e)\colon e \in C\} \subseteq B$. This contradicts the fact that e is the edge in $\operatorname{argmax}\{w(e)\colon e \in C\}$ that is charged last.

[1] Lemma 2.11 is closely related to and directly implies the assertion of Claim 1 in [92] (which is only stated for $e \in C \setminus \mathrm{MST}_B$). We give here a new version because, on the one hand, we need this more general statement later on, and on the other hand, there is a subtle flaw in the proof of Claim 1 in [92]: In the notation of that proof, when we modify MST by replacing an edge from $C \cap \mathrm{MST}$ by the edge e', it is not clear that we again obtain a tree.

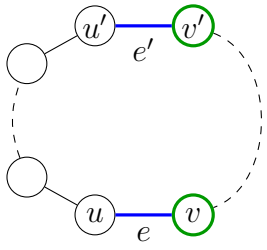

Figure 2.5: Illustration of Lemma 2.11. The green vertices (v and v') are unexplored and the black vertices are explored. The blue edges (e and e') are boundary edges.

Summing up, we have shown the following facts: Upon exploration of $e = (u, v)$, there is another boundary edge $e' = (u', v')$ in C with $w(e') < w(e)$. Since e is not blocked, this implies

$$w(C \setminus \{e\}) \geq d(u, v') > (1 + \delta)w(e). \qquad \square$$

2.2.3 Connection to spanners

Next, we investigate how the performance of $\textsc{Blocking}_\delta$ is related to graph spanners. For this, note that Lemma 2.11 can be reformulated as follows.

Lemma 2.12. *No proper subgraph of $B \cup \mathrm{MST}_B$ is a $(1+\delta)$-spanner of $B \cup \mathrm{MST}_B$.*

The lemma relates spanners to the behavior of $\textsc{Blocking}_\delta$. However, we need to take note that the lemma applies to $B \cup \mathrm{MST}_B$ rather than the original graph G.

A *monotone graph class* is a class of graphs $\mathcal{G}$ closed under taking subgraphs, i.e., if $G \in \mathcal{G}$ and H is a subgraph of G, then also $H \in \mathcal{G}$. Given a graph G, we define $\textsc{OptSpan}_\delta(G)$ as the minimum lightness of a $(1+\delta)$-spanner of G. Moreover, given a graph class $\mathcal{G}$, we set $\textsc{OptSpan}_\delta(\mathcal{G}) := \sup\{\textsc{OptSpan}_\delta(G) : G \in \mathcal{G}\}$ to be the supremum over all graphs in $\mathcal{G}$. By slight abuse of notation, we also allow δ to be a function $\delta \colon \mathbb{N} \to \mathbb{R}_{>0}$ and write $\textsc{OptSpan}_\delta(\mathcal{G}) := \sup\{\textsc{OptSpan}_{\delta(|G|)}(G) : G \in \mathcal{G}\}$.

In the following result, when we allow δ to depend on n, this requires that the agent is given the number of vertices beforehand. However, the theorem also applies to the case where $\delta(n)$ is a constant independent of n. In that case, the agent does not need to know the number of vertices beforehand.

Theorem 2.13. *For every monotone graph class $\mathcal{G}$ and every $\delta = \delta(n) > 0$, the algorithm $\textsc{Blocking}_\delta$ is $(2(\delta+2) \cdot \textsc{OptSpan}_\delta(\mathcal{G}))$-competitive on $\mathcal{G}$.*

Proof. Let $G \in \mathcal{G}$. We have

$$\textsc{Blocking}_\delta(G, v, \delta) \overset{\text{Obs 2.10}}{\leq} 2(\delta+2)w(B) \leq 2(\delta+2)w(B \cup \mathrm{MST}_B). \tag{2.1}$$

Since $B \cup \mathrm{MST}_B$ is a subgraph of G, we have $B \cup \mathrm{MST}_B \in \mathcal{G}$. By Lemma 2.12, the only $(1+\delta)$-spanner of $B \cup \mathrm{MST}_B$ is $B \cup \mathrm{MST}_B$ itself. Therefore,

$$\begin{aligned} w(B \cup \mathrm{MST}_B) &\leq \textsc{OptSpan}_\delta(B \cup \mathrm{MST}_B) \cdot w(\mathrm{MST}_B) \\ &\leq \textsc{OptSpan}_\delta(\mathcal{G}) \cdot w(\mathrm{MST}_B), \end{aligned} \tag{2.2}$$

where we have used that MST_B is also a minimum spanning tree of $\mathrm{MST}_B \cup B$. Combined, we obtain

$$\begin{aligned}\textsc{Blocking}_\delta(G, v, \delta) &\overset{(2.1)}{\leq} 2(\delta+2)w(B \cup \mathrm{MST}_B) \\ &\overset{(2.2)}{\leq} 2(\delta+2) \cdot \textsc{OptSpan}_\delta(\mathcal{G}) \cdot w(\mathrm{MST}_B).\end{aligned}$$

Recall that the total cost of the offline optimum is bounded from below by the weight of a minimum spanning tree. Therefore, this completes the proof. □

The theorem puts us in a position to leverage results on the lightness of spanners in order to draw conclusions regarding the competitive ratio of $\textsc{Blocking}_\delta$. For example, it has been shown that every planar graph contains a $(1+\delta)$-spanner of lightness at most $1+\frac{2}{\delta}$ [4]. Feeding this into Theorem 5.1, we conclude that $\textsc{Blocking}_\delta$ is $2(\delta+2)(1+2/\delta)$-competitive on planar graphs. This agrees with the bound proven in [75]. More generally, bounded-genus graphs have light spanners. In fact, in Section 2.3.3, we show that every graph of genus at most g contains a $(1+\delta)$-spanner of lightness at most $\big(1+\frac{2}{\delta}\big)\big(1+\frac{2g}{1+\delta}\big)$ (Theorem 2.19). From this, we obtain the following.

Corollary 2.14. *$\textsc{Blocking}_\delta$ is $2(\delta+2)\big(1+\frac{2}{\delta}\big)\big(1+\frac{2g}{1+\delta}\big)$-competitive on graphs of genus at most g.*

Even more generally, it is known that H-minor-free graphs have light spanners [30]. Specifically, every H-minor-free graph contains a $(1+\delta)$-spanner of lightness $\mathcal{O}\Big(\frac{\sigma_H}{\delta^3}\log\big(\frac{1}{\delta}\big)\Big)$ where $\sigma_H := |V(H)|\sqrt{\log|V(H)|}$. This yields a constant competitive ratio for $\textsc{Blocking}_\delta$ on H-minor-free graphs as follows.

Corollary 2.15. *$\textsc{Blocking}_\delta$ is $2(\delta+2)\cdot\mathcal{O}\big(\frac{\sigma_H}{\delta^3}\log\big(\frac{1}{\delta}\big)\big)$-competitive on H-minor-free graphs.*

For the case of general graphs, it is known that every graph G contains, for every $k \geq 1$ and $\varepsilon \in (0,1)$, a $(2k-1)(1+\varepsilon)$-spanner of lightness $\mathcal{O}_\varepsilon\big(n^{1/k}\big)$ [34]. This gives us the following.

Corollary 2.16. *Given $k = k(n) \in \mathbb{N}_{\geq 1}$ and $\varepsilon \in (0,1)$, $\textsc{Blocking}_{(2k-1)(1+\varepsilon)}$ is $2((2k-1)(1+\varepsilon)+2)\cdot\mathcal{O}_\varepsilon\big(n^{1/k}\big)$-competitive on every graph on n vertices.*

By suitably choosing δ in the Corollaries above, we obtain the following improved bounds.

Corollary 2.17.

a) $\textsc{Blocking}_2$ is $16(1+\frac{2}{3}g)$-competitive on graphs of genus at most g.

b) For every constant $\delta > 0$ and every graph H, $\textsc{Blocking}_\delta$ is constant-competitive on H-minor-free graphs.

c) $\textsc{Blocking}_{\log(n)}$ is $\mathcal{O}(\log(n))$-competitive on every graph on at most n vertices.

Proof. Part a) follows from Corollary 2.14 by setting $\delta = 2$. For part b), observe that the bound in Corollary 2.15 only depends on H and δ so that we obtain a constant competitive ratio when H and δ are constant.

For part c), consider the cost incurred by $\text{BLOCKING}_{\log(n)}$ on a graph on $n' \leq n$ vertices. Note that one can choose an integer $k \in \Theta(\log(n))$ and $\varepsilon \in (0.5, 1)$ such that we have $(2k-1)(1+\varepsilon) = \log(n)$. For the competitive ratio from Corollary 2.16, we then obtain

$$2((2k-1)(1+\varepsilon)+2) \cdot \mathcal{O}_\varepsilon((n')^{1/k}) = \mathcal{O}\big(\log(n) \cdot (n')^{1/\log(n)}\big) = \mathcal{O}(\log(n)),$$

where we have used $(n')^{1/\log(n)} \leq n^{1/\log(n)} = 2^{\log(n)\cdot(1/\log(n))} = 2$ in the last equality. □

In particular, this completes the proof of Theorem 2.7 and Theorem 2.8. For the case of planar graphs, part a) matches the best known bounds on planar graphs [75, 92]. For general surfaces, it slightly improves on the best known bound of $16(1+2g)$ on bounded-genus graphs [92]. Part b) is the first constant bound on minor-free graphs, and part c) is the first $\mathcal{O}(\log(n))$ bound for BLOCKING.

Note that Cololary 2.17 c) gives an $\mathcal{O}(\log n)$-competitive algorithm if we assume that n is given beforehand. Since this bound depends on n, the equivalence proven in Corollary 2.6 cannot be applied here. Next, we demonstrate how one can use a doubling-strategy to obtain an $\mathcal{O}(\log n)$-competitive algorithm relying on BLOCKING, even if n is not known beforehand. The strategy is to "guess" the number of vertices and whenever we learn that our guess was not high enough, we restart the algorithm and increase our guess.

More precisely, we define BLOCKING′ as follows: The algorithm is executed in *phases*. In each phase i, we are given a number n_i (the guess for the number of vertices) and, at the beginning of a phase, the agent is located in the starting vertex. Recall that we call a vertex *learned* when one of its neighbors or the vertex itself was visited. In phase i, we execute $\text{BLOCKING}_{\log(n_i)}$ until the number of learned vertices exceeds n_i or the entire graph is explored. Then, we return to the starting position. If the guess n_i was exceeded, we start the next phase with $n_{i+1} = n_i^2$ and the first phase is started with $n_1 = 2$. In other words, we have $n_i = 2^{2^{i-1}}$.

Lemma 2.18. *BLOCKING′ is $\mathcal{O}(\log n)$-competitive.*

Proof. Let G be a graph on $n \geq 2$ vertices and we analyze the performance of BLOCKING′ on G. Let ALG_i denote the cost incurred during phase i and let k be the total number of phases.

Note that, when a vertex v is visited for the first time, the total number of learned vertices is increased by the number of *new neighbors* of v, i.e., by the number of neighbors of v that are not neighboring any of the other visited vertices. If phase i ends upon the visit of v (i.e., the total number of visited and learned vertices exceeds n_i), we call the new neighbors of v the *threshold vertices of phase i*. Now, we define the graph G_i as follows: Take the subgraph of G that is induced by all vertices that were visited or learned during phase i, remove the threshold vertices of

phase i, remove all edges between unexplored vertices, and add an edge of weight 0 between every pair of unexplored vertices.

Note that G_i has the following properties: First, we have $|V(G_i)| \leq n_i$. Second, observe that the weights of the minimum spanning trees fulfill $\mathrm{MST}(G_i) \leq \mathrm{MST}(G)$ (note that we used here that edges of weight 0 were added between unexplored vertices). And third, the cost incurred during phase i is precisely the cost of $\textsc{Blocking}_{\log(n_i)}$ executed on G_i. Using Corollary 2.17 c), we obtain that the cost incurred during phase i is at most

$$\begin{aligned}\mathcal{O}(\log(n_i)) \cdot \textsc{Opt}(G_i) &\leq \mathcal{O}(\log(n_i)) \cdot 2\mathrm{MST}(G_i) \leq \mathcal{O}(\log(n_i)) \cdot 2\mathrm{MST}(G) \\ &\leq \mathcal{O}(\log(n_i)) \cdot 2\textsc{Opt}(G) = \mathcal{O}(\log(n_i)) \cdot \textsc{Opt}(G).\end{aligned}$$

Note that

$$\begin{aligned}\sum_{i=1}^{k} \log(n_i) = \sum_{i=0}^{k-1} \log(2^{2^i}) = \sum_{i=0}^{k-1} 2^i = \mathcal{O}(2^k) = \mathcal{O}(\log(n_k)) = \mathcal{O}(\log(n^2)) \\ = \mathcal{O}(\log n),\end{aligned}$$

where we have used that $n_k \leq n^2$. Together, we obtain that the total cost of $\textsc{Blocking}'$ is at most

$$\sum_{i=1}^{k} \mathcal{O}(\log(n_i)) \cdot \textsc{Opt}(G) = \mathcal{O}\left(\sum_{i=1}^{k} \log(n_i)\right) \cdot \textsc{Opt}(G) = \mathcal{O}(\log(n)) \cdot \textsc{Opt}(G).$$

□

2.3 Interlude: Graph spanners in bounded-genus graphs

Next, we exploit the connection between spanners and exploration in the opposite direction to derive the existence of good spanners in bounded-genus graphs. More precisely, we show the following result about the so-called greedy spanner and our proof for this will use ideas from the exploration literature [92]. This will also allow us to derive better approximation guarantees for $\textsc{Blocking}_\delta$, which we used for Corollary 2.14.

Theorem 2.19. *For every $\varepsilon > 0$, the greedy $(1+\varepsilon)$-spanner of a graph of genus g has lightness at most $\left(1+\frac{2}{\varepsilon}\right)\left(1+\frac{2g}{1+\varepsilon}\right)$.*

Prior to our work, the best known bound was due to Grigni [60] who showed that every graph of genus $g \geq 1$ contains a $(1+\varepsilon)$-spanner of lightness $1+\frac{12g-4}{\varepsilon}$. Let us briefly comment on how our bound compares to this. Observe that, for $g \geq 1$,

$$\left(1+\frac{2}{\varepsilon}\right)\left(1+\frac{2g}{1+\varepsilon}\right) = 1+\frac{2}{\varepsilon}+\frac{2g}{1+\varepsilon}+\frac{4g}{\varepsilon(1+\varepsilon)} < 1+\frac{2g}{\varepsilon}+\frac{2g}{\varepsilon}+\frac{4g}{\varepsilon} = 1+\frac{8g}{\varepsilon}.$$

Therefore, our bound is stronger than Grigni's bound for every $g \geq 1$. Moreover, in the planar case (i.e., $g = 0$), we obtain a lightness of $1 + \frac{2}{\varepsilon}$. It was shown by Althöfer et al. [4, Theorem 5] that this is best possible, i.e., our bound is tight for planar graphs. This means that Theorem 2.19 gives a tight bound in the case $g = 0$ and extrapolates this bound to graphs of larger genus.

Note that the worst-case lightness for spanners of graphs of genus g has to increase in g, since not every graph admits a light spanner [95, Theorem 6.6].

2.3.1 The greedy spanner

The *greedy* $(1+\varepsilon)$*-spanner* was suggested by Althöfer et al. [4] and is formally defined as the output of Algorithm 2. After ordering the edges by weight, it iteratively adds edges if they are short in comparison to the distance of their endpoints in the graph constructed so far. It was shown by Filtser and Solomon [53] that this spanner construction is existentially optimal for every monotone graph class, which means that the optimal lightness guarantee on any such class is achieved by the greedy spanner.

Algorithm 2: GREEDYSPANNER$(G = (V, E, w), \varepsilon)$

```
1 sort E = {e_1, ..., e_m} such that w(e_1) ≤ w(e_2) ≤ ··· ≤ w(e_m)
2 H ← (V, ∅)
3 for i ← 1, ..., m do
4     if d_H(u_i, v_i) > (1 + ε)w(e_i), where e_i = (u_i, v_i) then
5         H ← H ∪ {e_i}
6 return H
```

Note that the resulting graph H is indeed a $(1 + \varepsilon)$-spanner of G. The output of the algorithm actually depends on the chosen order of the edges. In particular, when edge weights appear multiple times, there may be several possible outputs. However, this will not be important in our context. When we refer to *the* greedy spanner, we mean that we arbitrarily fix some output of the algorithm.

The greedy spanner fulfills the following two key properties: First, the algorithm implicitly executes Kruskal's algorithm for finding a minimum spanning tree, i.e., it adds all edges to H that Kruskal's algorithm adds. With this, we obtain the following.

Observation 2.20. *There exists a minimum spanning tree of the input graph that is contained in the greedy spanner.*

The second key property, in fact, resembles the property of BLOCKING$_\delta$ in Lemma 2.11.

Observation 2.21 (Althöfer et al. [4])**.** *For every cycle C in the greedy spanner H and every edge e of C, we have $w(C \setminus \{e\}) > (1+\varepsilon)w(e)$. In other words, no proper subgraph of H is a $(1 + \varepsilon)$-spanner of H.*

Proof. Let C be a cycle in the greedy spanner. Let $e = \{u, v\}$ be the edge in C that is added last. At the time it is added, we have $(1+\varepsilon)w(e) < d_H(u, v) \leq w(C \setminus \{e\})$ by definition of the algorithm. Since all other edges in C have lower or equal weight than e, the property is fulfilled for them as well. □

2.3.2 Spanners in planar graphs

Before investigating spanners in bounded-genus graphs, we illustrate our techniques for the special case of planar graphs, giving an alternative proof of the following result.

Theorem 2.22 (Althöfer et al. [4])**.** *For every planar graph G and $\varepsilon > 0$, the greedy $(1+\varepsilon)$-spanner of G has lightness at most $1 + \frac{2}{\varepsilon}$.*

Our proof uses similar ideas as in [92, Theorem 1], i.e., we use techniques that were used to show that BLOCKING$_\delta$ is constant-competitive on planar graphs. The outline of our proof is roughly as follows: Fix an embedding of the greedy spanner in the plane and, in a suitable way, partition the greedy spanner into facial cycles, i.e., cycles that form the boundary of a face. Then use the fact that none of these cycles are short (cf. Observation 2.21).

Lemma 2.23. *Let G be a planar graph, H be the greedy $(1+\varepsilon)$-spanner of G and* MST *be a minimum spanning tree of H. Fix an embedding of H in the plane. Then we can associate with every edge $e \in H \setminus$ MST *a facial cycle C_e containing e, so that $C_e \neq C_{e'}$ for $e \neq e'$.*

Proof. We show that it is possible to iteratively choose an edge e in $H \setminus$ MST that closes a facial cycle C_e (of the fixed embedding of H) together with the edges of MST and the edges chosen in previous iterations: We define a partial order on the edges in $H \setminus$ MST. Every edge $e \in H \setminus$ MST closes a cycle C together with the edges of MST. We let another edge e' precede e in the partial order if it lies on the inside of this cycle in the considered embedding (see Figure 2.6). In each iteration, we can choose an edge which is minimal in this partial order amongst the edges to which no cycle has yet been assigned. Note that, in this construction, no two edges are assigned the same cycle. □

Next, we combine this with the fact that the greedy spanner does not contain short cycles (cf. Observation 2.21).

Lemma 2.24. *Let G be a graph and H be the greedy $(1+\varepsilon)$-spanner of G. Let D be a subgraph of H such that we can associate with every edge $e \in H \setminus D$ a cycle C_e of H containing e, with the property that $\sum_{e \in H \setminus D} w(C_e) \leq 2w(H)$. Then we have*

$$w(H) \leq \left(1 + \frac{2}{\varepsilon}\right) w(D).$$

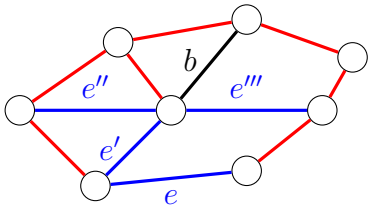

Figure 2.6: Illustration of the partial order in Lemma 2.23: The red edges (without label) denote MST, the black edge b has already been chosen in a previous iteration and the blue edges e, e', e'', e''' have not yet been assigned a facial cycle. In this example, e', e'', and e''' precede e. Moreover, e'' precedes e'. The edges e'' and e''' are both minimal. Note that e''' was not minimal before the black edge was chosen. If e''' is chosen in this step, the facial cycle assigned to e''' consists of the black edge b, the blue edge e''', and two red edges.

Proof. We obtain

$$\begin{aligned} w(H \setminus D) &= \sum_{e \in H \setminus D} w(e) \overset{\text{Obs 2.21}}{<} \frac{1}{1+\varepsilon} \sum_{e \in H \setminus D} w(C_e \setminus \{e\}) \\ &= \frac{1}{1+\varepsilon} \left(\sum_{e \in H \setminus D} w(C_e) - \sum_{e \in H \setminus D} w(e) \right) \\ &\leq \frac{1}{1+\varepsilon} (2w(H) - w(H \setminus D)) = \frac{1}{1+\varepsilon} (2w(D) + w(H \setminus D)). \end{aligned}$$

Rearranging yields

$$w(H \setminus D) \leq \frac{2}{\varepsilon} \cdot w(D) \tag{2.3}$$

and thus

$$w(H) = w(D) + w(H \setminus D) \overset{(2.3)}{\leq} \left(1 + \frac{2}{\varepsilon}\right) w(D). \qquad \square$$

Next, we show how this implies Theorem 2.22.

Proof of Theorem 2.22. Let G be a planar graph, let H be the greedy $(1+\varepsilon)$-spanner of G, and let MST denote a minimum spanning tree of H. By Observation 2.20, MST is also a minimum spanning tree of G, so that it suffices to show $w(H) \leq \left(1 + \frac{2}{\varepsilon}\right) w(\text{MST})$. Since G is planar, its subgraph H is planar as well. Let us fix an embedding of H on the plane such that no two edges cross. By Lemma 2.23, there is a facial cycle C_e for every edge $e \in H \setminus \text{MST}$ such that $C_e \neq C_{e'}$ for $e \neq e'$. As every edge of H is contained in at most two facial cycles, we have $\sum_{e \in H \setminus \text{MST}} w(C_e) \leq 2w(H)$. Therefore, we can apply Lemma 2.24 with $D = \text{MST}$ and obtain $w(H) \leq \left(1 + \frac{2}{\varepsilon}\right) w(\text{MST})$. $\square$

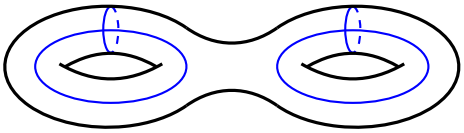

Figure 2.7: Surface of genus 2 with 4 non-separating cycles bounding a topological disk.

2.3.3 Generalization to bounded-genus graphs

In this subsection, we study light spanners for the class of bounded-genus graphs and prove Theorem 2.19.

Our proof is based on similar arguments as in [92, Theorem 2] for bounding the cost of BLOCKING$_\delta$ on bounded-genus graphs and the main idea is roughly as follows: Given an embedding of the greedy spanner on a surface of genus g, first cut the surface along several edges such that we obtain a disk. Then we can proceed along similar lines as we did for planar graphs (cf. Theorem 2.22). In this work, we estimate more carefully the weight of the edges along which we cut so that we obtain a slightly improved bound than in [92]. We will use the following topological lemma for the first step.

Lemma 2.25. *Let G be an unweighted connected graph of genus (exactly) $g \geq 1$. Fix an embedding of G on an orientable surface of genus g and let T be a spanning tree of G. Then there is a subgraph D of G with $T \subseteq D$ and $|E(D) \setminus E(T)| \leq 2g$ such that, in the inherited embedding of D, there is only a single face and the edges in D bound a topological disk.*[2]

Proof. It is a standard fact from topology that, on a surface of genus g, one can embed precisely $2g$ closed curves that are non-separating, i.e., it is possible to draw $2g$ cycles on the surface such that cutting along all of them does not disconnect the surface. Every collection of $2g$ curves that are non-separating bounds a topological disk (see Figure 2.7).[3]

We construct the set D greedily as follows (see Figure 2.8): Initially, let $D := T$. Ignoring all edges in $G \setminus D$, we have only a single face. Note that every edge in $G \setminus D$ closes a cycle with D. If we find an edge which only closes non-separating cycles, i.e, does not separate the surface into two faces, we add it to D. After this, the edges of D still only bound a single face. We repeat this step until we cannot find further edges whose addition would separate the surface into multiple faces.

We show that the resulting set of edges D is as desired. First, observe that we have $|E(D) \setminus E(T)| \leq 2g$, since there are at most $2g$ cycles on a surface of genus g that are non-separating.

[2] A topological disk is a surface homeomorphic to a 2-dimensional disk. Intuitively, a topological disk is a continuous deformation of a 2-dimensional disk.

[3] This can be proven as follows: The Euler characteristic of a surface of genus g is $2 - 2g$ [64, Section 2.2] and cutting along a non-separating closed curve increases the Euler characteristic by 1.

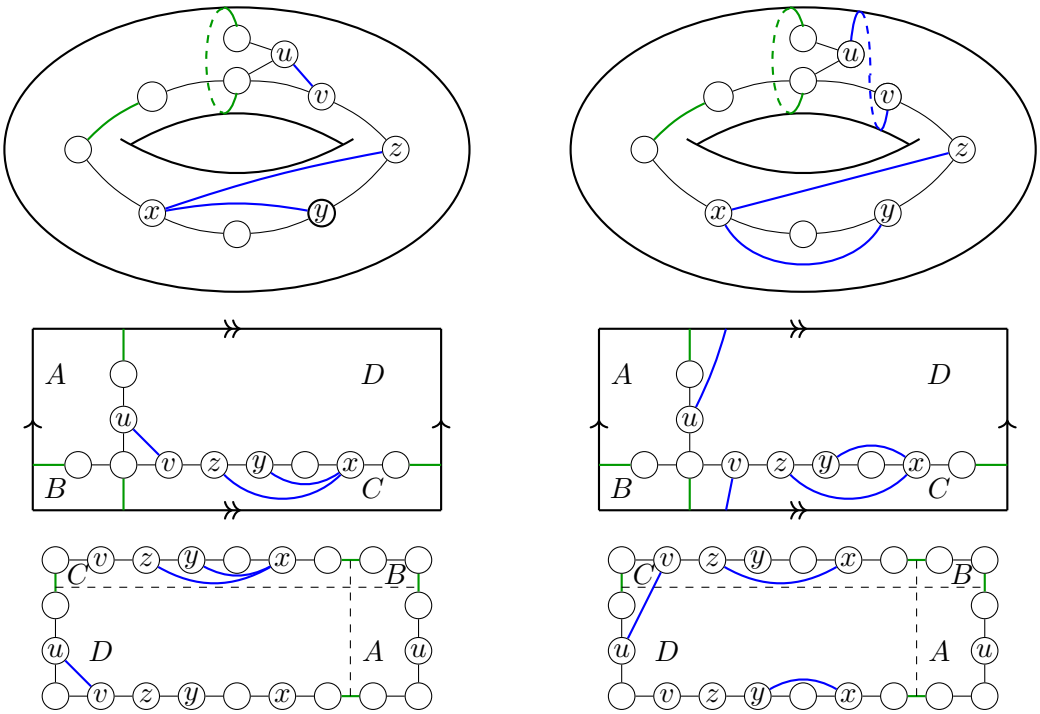

Figure 2.8: The two columns show the construction of D for the same graph with two different embeddings. The black edges belong to T, the green edges to $D \setminus T$, and the blue edges $(\{u, v\}, \{x, y\}, \{x, z\})$ to $G \setminus D$. In each column, the first subfigure shows the embedding on the torus. The second subfigure shows a different representation: The torus is obtained by gluing together the opposite sides of the rectangle. The last subfigure shows the disk obtained by cutting the surface along D. Note that it contains every edge of D twice and therefore, every vertex up to 4 times. However, note that the embedding specifies between which copies of the vertices the blue edges have to be drawn. The capital letters A, B, C, D denote areas of the torus and are included for better orientation: Leaving area A to the left leads to area D, leaving A to the top leads to B and so on.

It is left to show that D bounds a disk. By maximality of D, every edge $e \in G \setminus D$ is separating when added to D, i.e., in the inherited embedding of $D \cup \{e\}$, the edge e is incident to two faces. In particular, e is incident to two faces in the inherited embedding of every supergraph of D. Consider again the embedding of the entire graph G. It is known from topological graph theory that a minimal genus embedding of a connected graph is cellular, i.e., every face of the embedding of G is a topological disk [113] (see [97, Proposition 3.4.1]). Since every edge $e \in G \setminus D$ is incident to two distinct faces, its removal merges the two corresponding disks along a connected part of their common boundary, which yields another disk. Iteratively removing all edges in $G \setminus D$ in this way, we thus obtain a cellular embedding of D. Since, by construction, D induces only a single face, we obtain that D bounds a topological disk.

For an illustration of the construction, consider Figure 2.8. In the example in the left column, the two green edges enclose non-separating cycles, whereas all blue

edges close separating cycles. In the example in the right column, the half-dotted green edge in D could be replaced by the blue edge between u and v. □

Now, we have all the prerequisites in place to prove Theorem 2.19. The main idea is to give a similar construction as in Lemma 2.23 to partition the greedy spanner into facial cycles. Before delving into the proof, let us briefly comment on why Lemma 2.25 is not a reduction to the planar case, i.e., we cannot use the same construction as in Lemma 2.23.

Recall that the key ingredient of Lemma 2.23 was to define a partial order in which an edge e' precedes another edge e if e' is embedded on the inside of the cycle that e closes with MST. In the bounded-genus case, if the cycle closed by e is non-separating, there is no such thing as "the inside" of the cycle. For example, consider the edge $\{u, v\}$ in the right column of Figure 2.8 and the cycle it closes with MST. This cycle does not have an "inside" and cannot be decomposed into multiple faces. However, it separates the disk bounded by D into two parts. Therefore, we have to consider cycles that include edges of $D \setminus \text{MST}$ and we will use that these cycles partition the disk bounded by D.

Proof of Theorem 2.19. Let G be some graph of genus g. Let H be the greedy $(1+\varepsilon)$-spanner of G and let MST denote a minimum spanning tree of H. By Observation 2.20, we know that MST is also a minimum spanning tree of G, so that it suffices to show $w(H) \leq \big(1+\frac{2}{\varepsilon}\big)\big(1+\frac{2g}{1+\varepsilon}\big)w(\text{MST})$.

Let g' be the genus of H. If $g' = 0$, the assertion follows directly by Theorem 2.22. Therefore, we assume from now on $g' \geq 1$. Note that $g' \leq g$ because H is a subgraph of G. Fix an embedding of H on an orientable closed surface of genus g' such that no two edges cross. By Lemma 2.25, there is a subgraph D of H with $\text{MST} \subseteq D$ such that

$$|E(D) \setminus E(\text{MST})| \leq 2g' \leq 2g \tag{2.4}$$

and such that the edges of D induce only one face and bound a topological disk.

Next, observe that, for every edge e in $H \setminus \text{MST}$, we have $w(e) \leq w(\text{MST})/(1+\varepsilon)$: Every edge e in $H \setminus \text{MST}$ closes a cycle C together with the edges of MST. Using Observation 2.21, we obtain

$$w(e) < \frac{w(C \setminus \{e\})}{1+\varepsilon} \leq \frac{w(\text{MST})}{1+\varepsilon}.$$

In particular, this is fulfilled for edges in $D \setminus \text{MST}$. Combining this with (2.4), we obtain

$$w(D) = w(\text{MST}) + w(D \setminus \text{MST}) \leq \left(1 + \frac{2g}{1+\varepsilon}\right) w(\text{MST}). \tag{2.5}$$

The next step is to bound the weight of H by $(1+2/\varepsilon)w(D)$. For this, we use a similar construction as in Lemma 2.23 and show that it is possible to iteratively choose an edge e in $H \setminus D$ which, together with the edges of D and the edges chosen in previous iterations, closes a facial cycle C_e in the embedding of H. In each iteration, we find a suitable edge as follows: Pick an arbitrary edge e of $H \setminus D$.

If it defines a facial cycle together with D and edges chosen in previous iterations, we can simply choose e. Assume this is not the case. Note that e cuts the disk bounded by D in two parts and both contain edges in $H \setminus D$ to which no cycles have been assigned yet (otherwise e would close a suitable facial cycle). Pick the part whose boundary with D contains fewer edges (breaking ties arbitrarily) and pick a new edge e' in $H \setminus D$ which lies inside this half and has not yet been chosen in a previous iteration. Note that e' again cuts the disk in two parts and the boundary of the smaller part contains fewer edges of D than in the step before. Therefore, by repeating the steps above, we will end up with a suitable edge after finitely many steps. For example, on the left side of Figure 2.8, if we pick $e = (x, z)$, we will set $e' = (x, y)$ and this edge is suitable. After this, we can assign a facial cycle to (x, z) and then to (u, v). In the instance on the right, we can assign the cycles to the blue edges in any order.

Note that, in this construction, no two edges are assigned the same facial cycle. As every edge is contained in at most two facial cycles, we have

$$\sum_{e \in H \setminus D} w(C_e) \leq 2w(H).$$

Therefore, we can now apply Lemma 2.24 and obtain

$$w(H) \overset{\text{Lem 2.24}}{\leq} \left(1 + \frac{2}{\varepsilon}\right) w(D) \overset{(2.5)}{\leq} \left(1 + \frac{2}{\varepsilon}\right)\left(1 + \frac{2g}{1+\varepsilon}\right) w(\text{MST}). \qquad \square$$

2.4 Lower bounds for BLOCKING

We turn back to studying the performance of BLOCKING$_\delta$. Prior to our work, the algorithm was only studied for constant choices of the parameter δ. We have already established that a better competitive ratio of $\mathcal{O}(\log n)$ can be achieved when δ is adapted according to the number of vertices (Theorem 2.8). In this section, we complement this by giving lower bounds for the algorithm when δ is a function depending on n. Recall that the case where δ is a constant was already covered in [92], where a lower bound of $\Omega(n^{1/(4+\delta)})$ was proven. In this section, we extend this as follows.

Theorem 2.26. *The competitive ratio of BLOCKING$_\delta$, where $\delta = \delta(n) > 0$, is at least*

a) $\Omega(\log(n)/\log(\log(n)))$,

b) $\Omega(\log(n))$ *for* $\delta \in o(\log(n)/\log(\log(n)))$ *as well as for* $\delta \in \Omega(\log(n))$.

In particular, this shows that there is no δ such that BLOCKING$_\delta$ is constant-competitive, but it remains open, whether there is a choice of δ for which the algorithm is $o(\log(n))$-competitive. Our lower bounds also suggest that one cannot achieve a better competitive ratio for the classical setting where n is not known beforehand by using a doubling strategy for BLOCKING as in Lemma 2.18.

We begin by observing that, with slight adaptations to the construction given in [92], the lower bound of $\Omega(n^{1/(\delta+4)})$ carries over to non-constant δ that grow sufficiently slowly.

Observation 2.27. *Let $\delta = \delta(n) > 0$ such that $\delta = o(\log(n))$. Then the competitive ratio of* BLOCKING$_\delta$ *is lower bounded by $\Omega\big(n^{\frac{1}{\delta+4}}\big)$.*

Proof. We use precisely the same graph as in the proof of the lower bound for BLOCKING$_\delta$ for constant δ in [92, Section 4] with slight modifications of some parameters. Therefore, we do not repeat the construction, but only give an overview and present the calculations to verify the claim.

We begin by choosing $\bar{d} = \bar{d}(n)$ such that $\delta \cdot \bar{d}^{\delta+4} = \Theta(n)$. Note that this is possible because $\delta = o(\log(n))$ and we obtain $\bar{d} = \omega(1)$. For the following computation, we additionally assume that $\delta = \omega(1)$, as the case $\delta = \mathcal{O}(1)$ is already covered in [92, Theorem 3].

The outline of the construction in [92, Section 4] is roughly as follows: First, we choose an unweighted graph H with $\Theta(\bar{d}^{\delta+2})$ vertices and $\Theta(\bar{d}^{\delta+3})$ edges (that is bipartite and satisfies some additional conditions on the girth and the degree). From this, we build a new graph G consisting of edges of weight 1 and edges of a larger weight w, called *heavy edges*. Here, we set $w = \bar{d}^2$. This new graph G is obtained by replacing the vertices of H by gadgets, each consisting of $\mathcal{O}(\bar{d}^2 + \delta w) = \mathcal{O}(\delta\bar{d}^2)$ vertices. Therefore, the size of G is $\mathcal{O}(\delta\bar{d}^2 \cdot |H|) = \mathcal{O}(\delta\bar{d}^{\delta+4}) = \mathcal{O}(n)$ as desired. The edges within a gadget are all of weight 1 and the edges originating from the graph H (i.e., connecting different gadgets) are all of weight w, i.e., the number of heavy edges of weight w is $\Theta(|E(H)|) = \Theta(\bar{d}^{\delta+3})$.

In [92], it is shown that, when BLOCKING$_\delta$ is performed on G, each of the edges of weight w is first blocked and turns unblocked when the agent is at distance $\Theta(\delta w)$ from the edge, and only one of the heavy edges is unblocked at a time. Therefore, the agent traverses all of the heavy edges and the incurred cost for each of these edges is $\Theta((\delta+1)w) = \Theta(\delta w)$. It follows that the overall incurred cost of BLOCKING$_\delta$ on G is at least[4] $\Omega(\delta w \cdot |(E(H)|) = \Omega(\delta\bar{d}^{\delta+5})$.

The graph G is chosen such that there is a spanning tree only using edges of weight 1. Hence, the cost of the offline optimum on G is $\mathcal{O}(|G|) = \mathcal{O}(\delta\bar{d}^{\delta+4})$. Together, we obtain that the competitive ratio of BLOCKING$_\delta$ on G is at least

$$\frac{\Omega(\delta\bar{d}^{\delta+5})}{\mathcal{O}(\delta\bar{d}^{\delta+4})} = \Omega(\bar{d}) = \Omega\left(\left(\frac{n}{\delta}\right)^{\frac{1}{\delta+4}}\right) = \Omega\Big(n^{\frac{1}{\delta+4}}\Big),$$

where the last equality follows from the fact that $\delta^{1/(\delta+4)} \to 1$ as $\delta \to \infty$. □

Note that if $\delta + 4 \leq \log(n)/\log(\log(n))$, then $n^{\frac{1}{\delta+4}} \geq e^{\log n \cdot \frac{\log\log n}{\log n}} = \log n$. This means that Observation 2.27 implies that BLOCKING$_\delta$ has competitive ratio in $\Omega(\log(n))$ whenever $\delta = o(\log(n)/\log\log(n))$. In particular, this shows the first part of Theorem 2.26 b).

[4] In the proof of [92, Theorem 3], the factor of δ is omitted because δ is treated as a constant.

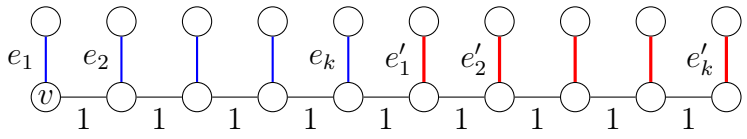

Figure 2.9: Illustration of the lower bound construction for BLOCKING$_\delta$ (Lemma 2.28). The light edges $e_1, \dots, e_k$ (depicted in blue) are of weight 1 and the heavy edges $e'_1, \dots, e'_k$ (depicted in red) are of weight $\frac{k+1}{\delta+1}$.

The next two results show that the parameter δ cannot be chosen too large either (the second part of Theorem 2.26 b).

Lemma 2.28. *Suppose $\delta = \delta(n) \in (0, \frac{n}{4})$. The competitive ratio of BLOCKING$_\delta$ is lower bounded by $\Omega(\delta)$, even on trees.*

Proof. Given a positive and large enough integer k, we construct a graph G as follows (see Figure 2.9): We begin with a path of $2k$ vertices and edges of weight 1. The first vertex v of this path is the start vertex. To the first k vertices, we attach leaves by edges $e_1, \dots, e_k$ of weight 1 and call these *light edges*. To the last k vertices, we attach leaves by edges $e'_1, \dots, e'_k$ of weight $h := \frac{k+1}{\delta+1}$ and call these *heavy edges*. Note that $n = 4k$ and $h > 1$ as k is chosen large enough.

Next, observe that BLOCKING$_\delta$ explores the graph in the following way: We can adversarially assume that BLOCKING$_\delta$ begins by exploring the path of $n/2$ vertices (and none of the light edges). Then, the heavy edge $e'_i = (u'_i, v'_i)$ is δ-blocked by the light edges $e_i \dots, e_k$, because $w(e'_i) > 1$ and the distance from u'_i to the unexplored end vertex of e_i is

$$k + 1 = (1 + \delta)\frac{k+1}{\delta+1} \leq (1 + \delta)w(e'_i).$$

But e'_i is not blocked by the light edges $e_1, \dots, e_{i-1}$, because the distance from u'_i to the end point of e_{i-1} is

$$k + 2 = (1 + \delta)\frac{k+2}{\delta+1} > (1 + \delta)w(e'_i).$$

Therefore, the agent explores, for $i = 0, \dots, k-1$, the $(k-i)$-th light edge and then the $(k-i)$-th heavy edge, forcing it to travel a distance of more than k at least k times. Hence,

$$\text{BLOCKING}_\delta(G, v, \delta) \geq k^2.$$

Since G is a tree, the length of the optimal tour is twice the total weight of the edges, i.e.,

$$2w(G) = 2\left((2k-1) + k + k \cdot \frac{k+1}{\delta+1}\right) \leq 2\left(\frac{(k+1)^2}{\delta+1} + 3k\right).$$

Therefore, the competitive ratio of BLOCKING$_\delta$ is at least

$$\frac{\text{BLOCKING}_\delta(G, v, \delta)}{2w(G)} \geq \frac{k^2}{2\left(\frac{(k+1)^2}{\delta+1} + 3k\right)} \to \frac{\delta+1}{2} \quad (k \to \infty).$$

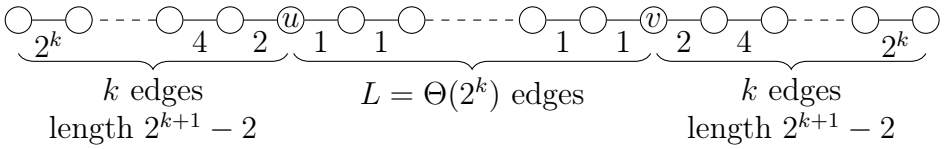

Figure 2.10: Illustration of the lower bound construction for hDFS from [92]. In Observation 2.29, we use this construction to give a lower bound of $\Omega(\log(\delta))$ for $\textsc{Blocking}_\delta$.

Since $k = n/4$, we have that $k \to \infty$ is equivalent to $n \to \infty$. Therefore, this proves the lower bound of $\Omega(\delta)$. □

To conclude our lower bound arguments for $\textsc{Blocking}_\delta$, observe that, for δ large enough, the behavior of $\textsc{Blocking}_\delta$ closely resembles the behavior of the algorithm hDFS [92], which always explores the lightest boundary edge. With this, we obtain the following.

Observation 2.29. *For every $\delta = \delta(n)$, the competitive ratio of $\textsc{Blocking}_\delta$ is lower bounded by $\Omega(\min\{\log(n), \log(\delta)\})$, even on paths.*

Proof. To prove the observation, we use the same graph as in the lower bound proof for the algorithm hDFS in [92, Theorem 5]. We begin by describing this construction (cf. Figure 2.10). Given $n \in \mathbb{N}$, set $k := \lceil \log(n) \rceil$ and $L := n - 2k - 1$. Note that $L \leq 2^k$ and $L = \Theta(n) = \Theta(2^k)$. We build a graph G as follows: Let P be a path of L edges of weight 1 and let u and v denote its endpoints. To both of these vertices, we attach a path of k edges of weights 2^i for $i = 1, \ldots, k$ (ordered increasingly in weight such that the edges of weight 2 are incident to u, respectively v). The graph G that we obtain is a path containing $L + 2k + 1 = n$ vertices as desired and has total weight $L + 2 \cdot (2^{k+1} - 2) \leq 5 \cdot 2^k$. We define the starting position to be u.

Now, we turn to analyzing the behavior of $\textsc{Blocking}_\delta$ on G. We claim that, for each $i \in \{1, \ldots, k\}$ with $i \geq k - \log((1+\delta)/5)$, no edge of weight 2^i is traversed before all of the lighter edges were traversed. To see this, let e be an edge of weight 2^i for such an i and observe that

$$\begin{aligned}(1+\delta)w(e) = (1+\delta)2^i &\geq (1+\delta)2^{k-\log((1+\delta)/5)} = (1+\delta) \cdot \frac{2^k}{(1+\delta)/5} \\ &= 5 \cdot 2^k \geq w(G).\end{aligned}$$

This implies that e is blocked as long as there is some lighter boundary edge.

The claim implies that, for each $i \in \{1, \ldots, k\}$ with $i \geq k - \log((1+\delta)/5)$, $\textsc{Blocking}_\delta$ explores both edges of weight 2^i before proceeding to heavier edges. Thus, the path P is traversed at least $\min\{\log((1+\delta)/5), k\} = \Omega(\min\{\log(\delta), \log(n)\})$ times. Therefore, the total cost of $\textsc{Blocking}_\delta$ on G is at least

$$\Omega(\min\{\log(\delta), \log(n)\}) \cdot w(P) = \Omega(\min\{\log(\delta), \log(n)\} \cdot n).$$

On the other hand, the cost of the offline optimum is $2w(G) = \Theta(n)$. With this, we obtain that the competitive ratio of $\textsc{Blocking}_\delta$ is at least $\Omega(\min\{\log(\delta), \log(n))$. □

We can now combine the lower bound constructions of this section to prove Theorem 2.26.

Proof of Theorem 2.26. We begin by proving part b), i.e., a lower bound of $\Omega(\log(n))$ for $\delta \in o(\log(n)/\log(\log(n))) \cup \Omega(\log(n))$. In Observation 2.27 (and the short comment after), we have covered the case $\delta \in o(\log(n)/\log\log(n))$. By Lemma 2.28, we obtain the lower bound of $\Omega(\log n)$ for every δ in the range from $\Omega(\log(n))$ to $n/4$, and by Observation 2.29, we obtain the lower bound for $\delta \geq n/4$. Together, this completes the proof of Theorem 2.26 b).

We now turn to part a), i.e., a lower bound of $\Omega(\log(n)/\log(\log(n)))$ for every δ. Note that part b) implies that a competitive ratio of $o(\log(n))$ is only possible for $\delta \in \Omega(\log(n)/\log\log(n)) \cap o(\log(n))$. Using Lemma 2.28 in this range implies the assertion of Theorem 2.26 a). □

2.5 A general lower bound of 4

In this section, we give a general lower bound construction for the online graph exploration problem and prove Theorem 2.3. We begin by restating it.

Theorem 2.3. *The competitive ratio of the online graph exploration problem is at least 4, even when restricted to subcubic planar graphs.*

Prior to our work, the best known lower bound was 10/3 [23], where the construction was also planar.

Overview of the proof of Theorem 2.3

To prove Theorem 2.3, we have to do the following: We give an adversarial strategy that, given an algorithm Alg fulfilling properties R1)-R5) of the restricted online graph exploration problem and $\varepsilon > 0$, constructs a planar graph G with starting vertex v such that $\textsc{Alg}(G, v)/\textsc{Opt}(G) \geq 4 - \varepsilon$. This implies that the strict competitive ratio of the restricted exploration problem on planar graphs is at least 4. Using the fact that the competitive ratio and strict competitive ratio coincide (Observation 2.4) and that assuming properties R1)-R5) and restriction to subcubic graphs is without loss of generality (Corollary 2.6), this implies the statement of Theorem 2.3.

Before giving the adversarial strategy in detail, let us introduce the main ideas behind it. Fix an algorithm Alg for the restricted online graph exploration problem. When we refer to *the agent*, we mean the agent of this online algorithm. In contrast, the *offline agent* denotes the agent in an offline optimum solution. The graph G that we construct consists of subgraphs serving as building blocks, which we refer to as *blocks*. The agent will be forced to traverse almost every block in order to reach

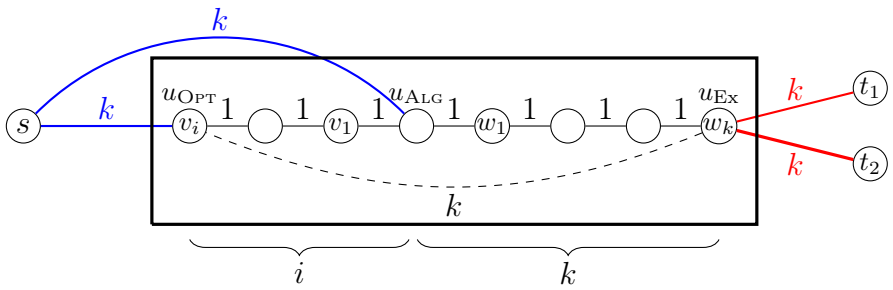

Figure 2.11: An adversarially constructed instance for the block traversal problem. The blue edges are entry edges and the red edges are exit edges. The dashed edge is only present if the agent does not travel back to s. The figure shows a block construced by $\mathcal{A}_1$ (defined in Lemma 2.32) when identifying vertices at distance 0 with each other and omitting parallel edges.

every vertex of the graph. Note that the cost incurred by the agent to traverse a block for the first time might differ from the cost of a second traversal, as the agent gains information during the first traversal that can be used later. Roughly speaking, we will arrange blocks such that every block is traversed once by the offline agent and twice by the online agent. Moreover, blocks will be built such that the agent pays three times as much as the offline agent for the first traversal.

We want to emphasize that the strategy of a block structure was already used in previous works on lower bounds for online graph exploration [23, 44]. Here, we use the same strategy for arranging the constructed blocks in the graph G as in [23]. The blocks that we define are based on the same ideas as in [23, 44], but we give a refined structure that allows for a stronger, and arguably simpler, analysis.

We informally describe the idea for the construction of a (single) block: Consider Figure 2.11. The agent starts at vertex s and we estimate its incurred cost until it traverses a red edge (without requiring it to visit all vertices). We can assume that the agent starts by exploring u_{ALG} (using R5), where it finds itself somewhere along a path of edges of weight 1. It then explores some of this path until the adversary decides to stop the process and we let w_k be the last explored vertex of the path. The other endpoint is then v_i, which is not yet explored, but v_{i-1} is. The value of i depends on the algorithm's behavior. Note that $i \neq 1$ only occurs if the agent explores the path in "zigzag movements", that is, if the agent changes direction before reaching an endpoint of the path. By R5), we can assume that the agent has to explore v_i before it can traverse a red edge. Therefore, the agent has to travel along the edge of weight k to v_i and back before traversing a red edge. In contrast, the offline agent can visit all vertices by following the path $(s, v_i, \dots, v_1, u_{\text{ALG}}, w_1, \dots, w_k, t_1)$.

When focusing on the red edges only, we observe the following behavior: Once the agent is located at w_k, it incurs a cost of $3k$ to traverse a red edge, whereas the offline agent only incurs a cost of k because it has explored v_i before w_k. Intuitively speaking, this means that the agent pays three times as much for traversing a red edge as the offline optimum. Next, we recursively replace all edges of weight 1 in

this graph by blocks of the same structure (with edge weights scaled down). This increases the fraction of edges for which the agent pays three times as much as the offline agent. In the limit, the agent incurs three times the cost of the offline agent for traversing the block.

Importantly, note that, during a second traversal, the agent does not have to use vertices $v_1, \ldots, v_i$, which makes the second traversal shorter. However, the cost incurred during the first traversal increases with i, i.e., if the agent explores the path in "zigzag movements". Therefore, in the analysis, we will carefully track these values. This is one of the key improvements over the work in [23]. In Section 2.5.1, we formally establish the notion of a block, the problem of traversing it, and the key values that need to be tracked. In Section 2.5.2, we then construct the blocks described above and prove that the agent incurs three times the cost of the offline optimum when traversing them.

Once we have established blocks, we arrange them in a graph as illustrated in Figure 2.15, which is already used in [23]. The structure of this graph is motivated by a lower bound construction for the exploration problem on unweighted graphs, which can be used to prove a lower bound of 2 on this class. In this graph, the agent travels on some walk from v_s to v_1 (Figure 2.15), where it encounters three paths of blocks. Since it cannot distinguish between these paths, we may assume that the first path that is completely explored is the one that leads back to v_s. The agent then has to backtrack to v_1 to make further progress. This behavior occurs in every cycle of the graph. The agent completes visiting all vertices at some position in the last cycle C^*. After this, it has to return to the starting position, incurring additional cost. In total, we obtain that it traverses (almost) every block twice. By contrast, the offline agent traverses the upper half of each cycle first and then, starting from the last cycle, traverses the lower half of each cycle. It thus traverses every block precisely once. This construction is formalized in Section 2.5.3.

2.5.1 The block traversal problem

In this subsection, we formalize the problem of traversing a block, for which we introduce a new online problem, called the *block traversal problem*. Intuitively, it is a variant of online graph exploration, where the task is only to find the exit of a maze instead of exploring all of it.

An instance of the *block traversal problem* (see Figure 2.11 for an example) consists of a weighted connected graph B, called a *block*, together with a starting vertex s and two target vertices t_1, t_2 (where s, t_1, t_2 are not considered to be part of B). The vertex s is connected to two different vertices of B, called its *algorithmic entry vertex* u_{Alg} and its *optimal entry vertex* u_{Opt}, via two edges of the same weight and we call these edges the *algorithmic entry edge* and *optimal entry edge*. The target vertices t_1 and t_2 are connected to the same vertex of B, called its *exit vertex* u_{Ex} and we call the corresponding edges the *exit edges*. By slight abuse of notation, by B we sometimes refer to the instance of the block traversal problem, instead of only the graph. In the block traversal problem, an exploration agent

obeying restrictions R1)-R5) is initially located in s and is tasked to reach t_1 or t_2 minimizing the total traveled distance.

Later, a block will be part of our graph in the lower bound construction and we will ensure that the agent needs to traverse almost all blocks in order to reach every vertex. Note that with the relaxed requirement that blocks only need to be traversed, instead of the requirement that all vertices are explored, we obtain a safe lower bound on the cost incurred by the agent. However, for the offline optimum, we still require to visit all vertices to obtain an upper bound on the cost incurred by the offline agent.

As already hinted at in the previous section, we keep track of multiple values for an instance B of the block traversal problem that turn out to be useful later on. In our constructions, we will often connect blocks where the exit edges of one block B correspond to the entry edges of the subsequent block B'. We will charge the cost of traversing these on the block B'. For this reason, when studying B in isolation, we never charge the cost of traversing an exit edge in the following values.

- The *algorithm's cost* $\textsc{Alg}_B$ is the cost incurred by the agent for the block traversal problem on B, excluding the cost of traversing an exit edge.
- The *block length* is $l_B := d(s, u_{\text{Ex}})$. In other words, this is the cost incurred by the online or offline agent in a second traversal.
- The *optimal exploration cost* $\textsc{Opt}_B$ is the length of a shortest walk from s to u_{Ex} that visits all vertices in B. In other words, this is the cost incurred by the offline agent.
- The *zigzag length* is $z_B := \textsc{Opt}_B - l_B$. Intuitively, this is the "discount" that the offline or online agent obtains for a second traversal. The name is inspired by the fact that, in the instances that we construct, this value is larger if the agent prefers to travel in "zigzag movements".
- The *algorithm's core cost* $\textsc{AlgC}_B$ is defined by $\textsc{AlgC}_B := \textsc{Alg}_B - z_B$.

The definition of the last value is motivated as follows: In our final construction, blocks will be traversed twice by the agent and we will be able to show that during the first traversal, the incurred cost is $3\textsc{Opt}_B + z_B$, and during the second traversal, the incurred cost is $\textsc{Opt}_B - z_B$. In a similar spirit to potential function arguments, the adversary "saves" z_B in the first traversal and charges it only in the second traversal, i.e., the core cost is the cost charged during the first traversal. As explained in Section 2.5, our goal is to prove the following result, which we will do in the next subsection. While this is closely related to [23, Theorem 3.1], the key difference is that we estimate the core cost instead of the total algorithm cost.

Theorem 2.30. *There exists an adversarial strategy that, for every $\varepsilon > 0$ and every block traversal algorithm Alg, constructs an instance B of the block traversal problem with $\textsc{Alg}C_B/\textsc{Opt}_B \geq 3 - \varepsilon$.*

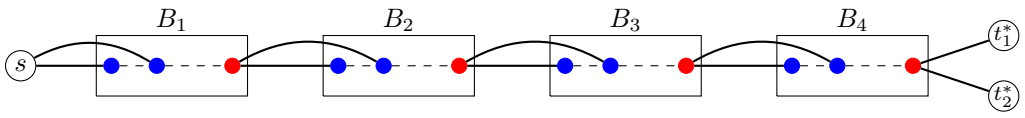

Figure 2.12: A chain of 4 blocks. The construction is obtained by identifying the exit vertex of B_j with the starting vertex of B_{j+1}. The algorithmic and optimal entry vertices are depicted in blue, the exit vertices are depicted in red. All other vertices inside the blocks are omitted. The dashed lines indicate that blocks are connected.

Before proving this result, we need to introduce the concept of connecting blocks to each other, which will be crucial for our proof. We define a *chain of blocks* (see Figure 2.12) as a sequence of blocks $B_1, \ldots, B_i$ where all entry and exit edges have the same weight by identifying the exit vertex of block B_j with the starting vertex of block B_{j+1} ($j \in \{1, \ldots, i-1\}$). In particular, we identify the target vertices of block j with the algorithmic and optimal entry vertex of block $j+1$. Let s be the starting vertex of B_1 and t_1^*, t_2^* be the target vertices of the last block B_i. Consider the block traversal problem on the chain, which we call in the following P, with s as the starting vertex and t_1^*, t_2^* as the target vertices. Note that, while ALG does not learn anything during the traversal of a block B_j about another block $B_{j'}$, the traversal of B_j can still affect the strategy followed in $B_{j'}$: Indeed, the agent may change strategy when facing the same situation multiple times. Let ALG_{B_j} denote the strategy followed in block B_j.

Observation 2.31. *In the block traversal problem on chain P, we have*

a) $\text{ALG}_P = \sum_{j=1}^{i} \text{ALG}_{B_j}$,

b) $\text{OPT}_P = \sum_{j=1}^{i} \text{OPT}_{B_j}$,

c) $l_P = \sum_{j=1}^{i} l_{B_j}$.

Proof. When the online or offline agent traverses an edge leading from B_j to B_{j+1}, we charge this cost on B_{j+1}. The statements now follow from the fact that every path from s to t_1^* or t_2^* contains the exit vertices of all blocks (where the exit vertex of one block is the starting vertex of the subsequent block). □

2.5.2 Recursive block construction

We now turn to proving Theorem 2.30. As explained in Section 2.5, we construct the blocks for this recursively using similar ideas as in [23], but estimate the core cost instead of the total algorithm's cost. One of the key differences is that our blocks have a fixed block length and variable optimum costs, whereas in [23], the block length is variable and the optimum cost is fixed. In our final construction, when the agent has to backtrack, it will have the choice along two different chains containing the same number of blocks. It will be beneficial for us that these have

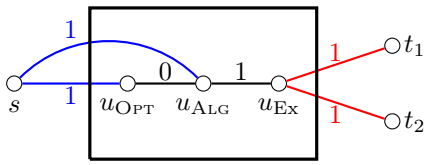

Figure 2.13: Block for the adversary strategy $\mathcal{A}_0$ of level 0.

the same length. Another key difference is that we construct only one type of block, where in [23], the beginnings and ends of chains of blocks have a special form so that three types of blocks are needed. This is not necessary for our construction, which makes it simpler.

Fix a sufficiently large and even integer k. For every $d \in \mathbb{N}_0$, we recursively define an adversarial strategy $\mathcal{A}_d$ that constructs an instance B of the block traversal problem with the following properties.

B1) The block length is $l_B = 2k^d$.

B2) The optimal cost is bounded by $\text{OPT}_B \leq (d+2)k^d$.

B3) The weight of the entry edges and exit edges is k^d.

We say that d is the *level* of the adversarial strategy, which corresponds to its recursion depth.

Lemma 2.32. *For every $d \in \mathbb{N}_0$, there exists an adversarial strategy $\mathcal{A}_d$ of level d that, for every block traversal algorithm ALG, constructs a planar instance B of the block traversal problem fulfilling B1)-B3) with the following property: Let r_d be the infimum of $\text{ALG}_B/\text{OPT}_B$ over all algorithms ALG, where B is the block constructed by $\mathcal{A}_d$ for ALG. Letting $k \in \Omega(d^2)$ and $d \to \infty$, we have $r_d \to 3$.*

Proof. For $d = 0$, we define $\mathcal{A}_0$ to output the block traversal instance illustrated in Figure 2.13 for every algorithm ALG. It is straightforward that $l_B = \text{OPT}_B = \text{ALG}_B = 2$ so that properties B1)-B3) are fulfilled and we have $r_0 = 1$.

Next, we define the strategy $\mathcal{A}_d$ for $d \geq 1$ assuming that $\mathcal{A}_{d-1}$ is already given.

Construction. The construction is illustrated in Figure 2.14. The starting vertex s is incident to the two entry edges of weight k^d and we may assume that the agent traverses the algorithmic entry edge (using property R5)). Then it is at position u_{ALG}, which we identify with the starting vertices of two blocks in which we use $\mathcal{A}_{d-1}$, i.e., u_{ALG} is incident to four edges of weight k^{d-1}. From here, we build two chains of blocks starting from u_{ALG}, i.e., whenever the agent traverses for the first time an exit edge of a block, we present it with a new block, where we identify the exit vertex of the previous block with the starting vertex of the new block. In each of these blocks, we apply strategy $\mathcal{A}_{d-1}$. Note that we use here property B3) to be able to build these chains. In the chain, we say that a block is explored if the agent has traversed its exit edge. This procedure stops if either (1) the agent travels back to s or (2) one of the two chains consists of $(k/2) - 1$ explored blocks.

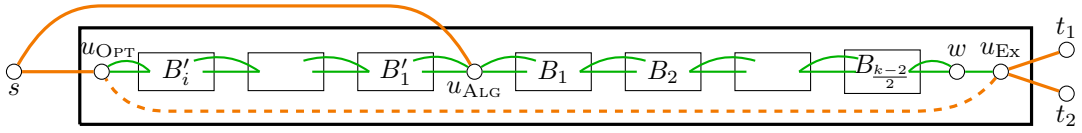

Figure 2.14: Recursive block construction (of level d). $B_1, \dots, B_{(k/2)-1}, B'_1, \dots, B'_i$ are blocks constructed by $\mathcal{A}_{d-1}$. The thick edges (depicted in orange) have weight k^d and the edges depicted in green have weight k^{d-1}. The dashed edge between u_{OPT} and u_{Ex} is only present in case (2), i.e., if the agent does not travel back to s. From this drawing, we obtain that the block is planar.

In case (1), let $B'_i, \dots, B'_1, u_{\text{ALG}}, B_1, \dots, B_j$ be the chains of blocks, where we have $1 \leq i, j \leq (k/2) - 1$ and all blocks except for B'_i, B_j are explored (but B'_i, B_j have been entered at some point). When the case is triggered, the agent is located at vertex s. Then we complete block B'_i to some valid block constructed by $\mathcal{A}_{d-1}$ and connect its two exit edges both to the same vertex u_{OPT}, which we define to be the optimal entry vertex. The chain of blocks $u_{\text{ALG}}, B_1, \dots, B_j$ is extended into a chain of $(k/2) - 1$ blocks. Importantly, we only fix here the number of these blocks but do not reveal the structure of the new blocks to the agent. This will be determined when the agent traverses them, where we will again use strategy $\mathcal{A}_{d-1}$. The two exit edges of block $B_{(k/2)-1}$ are then connected to a vertex w, and w is connected to a vertex u_{Ex} by an edge of weight k^{d-1}, which we define to be the exit vertex.

In case (2), let $B'_i, \dots, B'_1, u_{\text{ALG}}, B_1, \dots, B_{(k/2)-1}$ be the chains of blocks, where we have $i \leq (k/2) - 1$ and all blocks but B'_i are explored. The case is triggered when the agent traverses an exit edge of block $B_{(k/2)-1}$. Then we let both exit edges of $B_{(k/2)-1}$ lead to the same vertex w. This is connected by an edge of weight k^{d-1} to the exit vertex u_{Ex}. Block B'_i is completed to some valid block construced by $\mathcal{A}_{d-1}$ and both of its exit edges lead to the same vertex u_{OPT}, which we define to be the optimal entry vertex. Additionally, we introduce an edge of weight k^d between u_{OPT} and u_{Ex}. The case $d = 1$ is illustrated in Figure 2.11.

Block properties and analysis. Note that B3) is obviously fulfilled in the construction and one can observe from Figure 2.14 that the constructed block is planar, using that, by induction, the blocks constructed by $\mathcal{A}_{d-1}$ are planar as well. Using Observation 2.31 and that blocks constructed by $\mathcal{A}_{d-1}$ fulfill properties B1)-B3), we obtain that a shortest path from s to u_{Ex} is given by first traversing the algorithmic entry edge, then blocks $B_1, \dots, B_{(k/2)-1}$ and then the edges to w and u_{Ex}. We obtain

$$l_B = k^d + \sum_{j=1}^{(k/2)-1} l_{B_j} + 2k^{d-1} = k^d + (k/2) \cdot 2k^{d-1} = 2k^d, \tag{2.6}$$

so that property B1) is fulfilled (note that, in case (2), the existence of the additional edge $\{u_{\text{OPT}}, u_{\text{Ex}}\}$ does not lead to a shorter path). In both cases (1) and (2), observe that

$$\text{OPT}_B = k^d + \sum_{j=1}^{i} \text{OPT}_{B'_j} + k^{d-1} + \sum_{j=1}^{(k/2)-1} \text{OPT}_{B_j} + 2k^{d-1}. \tag{2.7}$$

Using that $\textsc{Opt}_{B_j} \leq (d+1)k^{d-1}$, we obtain

$$\begin{aligned}\textsc{Opt}_B &\leq k^d + \left(i + \frac{k}{2} - 1\right) \cdot (d+1)k^{d-1} + 3k^{d-1} \\ &\overset{3\leq 2(d+1)}{\leq} k^d + \left(i + \frac{k}{2} + 1\right) \cdot (d+1)k^{d-1} \overset{i\leq \frac{k}{2}-1}{\leq} k^d + (d+1)k^d = (d+2)k^d,\end{aligned}$$

so that property B2) is fulfilled as well.

By definition, we have $z_B = \textsc{Opt}_B - l_B$ and plugging in our findings from (2.6) and (2.7), we obtain

$$\begin{aligned}z_B &= \sum_{j=1}^{(k/2)-1} (\textsc{Opt}_{B_j} - l_{B_j}) + \sum_{j=1}^{i} \textsc{Opt}_{B'_j} + k^{d-1} \\ &= \sum_{j=1}^{(k/2)-1} z_{B_j} + \sum_{j=1}^{i} (z_{B'_j} + l_{B'_j}) + k^{d-1}. \end{aligned} \tag{2.8}$$

Next, we analyze the cost incurred by the agent. We say that a block is *backtracked* if the agent traverses it — either from s to u_{Ex} or in the other direction — after the block was explored. In case (1), the agent traverses 3 times an entry edge of weight k^d and, before the case was triggered, it explores and backtracks the blocks $B'_1, \dots, B'_{i-1}$. To reach u_{Ex}, Alg also has to traverse the blocks $B_1, \dots, B_{(k/2)-1}$, where we use that the edge $\{u_{\textsc{Opt}}, u_{\text{Ex}}\}$ is not present in this case. Traversing the exit edge of $B_{(k/2)-1}$ and $\{w, u_{\text{Ex}}\}$, the agent incurs another cost of $2k^{d-1}$. Using our findings for chains (Observation 2.31), we obtain

$$\textsc{Alg}_B \geq 3k^d + \sum_{j=1}^{(k/2)-1} \textsc{Alg}_{B_j} + \sum_{j=1}^{i-1} (\textsc{Alg}_{B'_j} + l_{B'_j}) + 2k^{d-1}. \tag{2.9}$$

In case (2), the agent traverses one entry edge, explores and backtracks the blocks $B'_1, \dots, B'_{i-1}$ and explores blocks $B_1, \dots, B_{(k/2)-1}$ before the case was triggered. Then, the agent is located at w and, by property R5), the agent is not able to traverse an exit edge before exploring $u_{\textsc{Opt}}$. Traveling from w to $u_{\textsc{Opt}}$ and then to u_{Ex} costs at least $2k^d + k^{d-1}$. Therefore, we obtain the same bound (2.9) for case (2) as in case (1). By (2.8) and (2.9), we obtain a core cost of

$$\begin{aligned}\textsc{AlgC}_B &= \textsc{Alg}_B - z_B \\ &\geq 3k^d + \sum_{j=1}^{(k/2)-1} (\textsc{Alg}_{B_j} - z_{B_j}) + \sum_{j=1}^{i} (\textsc{Alg}_{B'_j} - z_{B'_j}) - \textsc{Alg}_{B'_i} - \underbrace{l_{B'_i}}_{=2k^{d-1}} + k^{d-1} \\ &= 3k^d + \sum_{j=1}^{(k/2)-1} \textsc{AlgC}_{B_j} + \sum_{j=1}^{i} \textsc{AlgC}_{B'_j} - \textsc{Alg}_{B'_i} - k^{d-1} \\ &\geq 3k^d + \sum_{j=1}^{(k/2)-1} \textsc{AlgC}_{B_j} + \sum_{j=1}^{i} \textsc{AlgC}_{B'_j} - (3d+4)k^{d-1}, \end{aligned} \tag{2.10}$$

where we have used for the last inequality that $\textsc{Alg}_{B_i'} \leq 3\textsc{Opt}_{B_i'} \leq 3(d+1)k^{d-1}$ because otherwise, the statement of the Lemma is clear.

Ratio analysis. For $d > 1$, we obtain for a suitable choice of $i \in \{1, \dots, (k/2)-1\}$ using (2.7) and (2.10) that

$$\begin{aligned} r_d &\geq \frac{3k^d + \sum_{j=1}^{(k/2)-1} \textsc{AlgC}_{B_j} + \sum_{j=1}^{i} \textsc{AlgC}_{B_j'} - (3d+4)k^{d-1}}{k^d + \sum_{j=1}^{(k/2)-1} \textsc{Opt}_{B_j} + \sum_{j=1}^{i} \textsc{Opt}_{B_j'} + 3k^{d-1}} \\ &\geq \frac{3k^d + r_{d-1}\left(\sum_{j=1}^{(k/2)-1} \textsc{Opt}_{B_j} + \sum_{j=1}^{i} \textsc{Opt}_{B_j'}\right) - (3d+4)k^{d-1}}{k^d + \sum_{j=1}^{(k/2)-1} \textsc{Opt}_{B_j} + \sum_{j=1}^{i} \textsc{Opt}_{B_j'} + 3k^{d-1}} \\ &\geq \frac{3k^d + r_{d-1}k(d+1)k^{d-1} - (3d+4)k^{d-1}}{k^d + k(d+1)k^{d-1} + 3k^{d-1}} = \frac{3 + r_{d-1}(d+1) - (3d+1)/k}{1 + (d+1) + 3/k}, \end{aligned}$$

where we have used for the inequality in the last line that $\textsc{Opt}_{B_j} \leq (d+1)k^{d-1}$ by B2), $i < k/2$, and the fact that $\frac{x+z}{y+z} \leq \frac{x}{y}$ for $x \geq y \geq 0$ and $z \geq 0$. From this recursive expression, it is easy to see that, if $k \in \Omega(d^2)$ and we let $d \to \infty$ (and therefore also $k \to \infty$), we have $r_d \to 3$. (This can be seen from that fact that the terms with k in the denominator can be neglected and $r = 3$ is the only solution of $r = (3 + r(d+1))/(1 + d + 1)$.) □

Choosing d large enough in the Lemma, we obtain Theorem 2.30.

2.5.3 Block arrangement

Now that we have established the necessary preliminaries on block traversal, we turn to proving our main result (Theorem 2.3) in this section. For this, we use the same block arrangement as in [23], but we give a refined analysis using our stronger results on the block traversal problem.

Proof of Theorem 2.3. As a first step, we give the construction for the block arrangement as in [23].

Construction. Let $\varepsilon > 0$ be arbitrary and choose d, k large enough (depending on ε) with $k \in \Omega(d^2)$. Fix an online graph exploration algorithm Alg. Whenever, we refer to a *block* in this proof, we mean a block construced by $\mathcal{A}_d$. Moreover, let M and N be a large enough integers.

We construct a graph G (adaptively depending on the behavior of Alg) as follows (see Figure 2.15). Let v_s denote the starting position and identify it with the starting vertices of two blocks, i.e., v_s is incident to four boundary edges of weight k^d and the first two such traversed boundary edges are the algorithmic entry edges of two different blocks. Whenever the agent succesfully explores and leaves a block, its exit vertex is identified with the starting vertex of a new block, so that we obtain two chains of blocks P_1 and P_1' and, in each block, we employ strategy $\mathcal{A}_d$. This procedure ends when one of the chains consists of M blocks construced by $\mathcal{A}_d$ and we assume w.l.o.g. that this occurs for chain P_1. We let the exit edges of the last

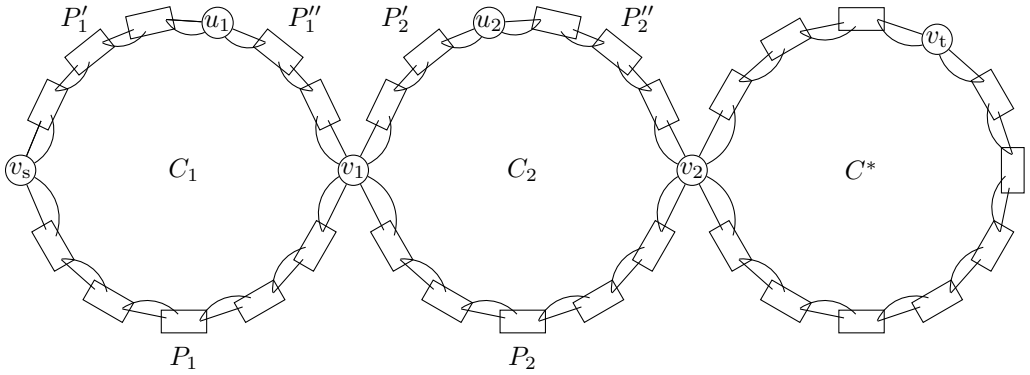

Figure 2.15: The block arrangement from [23] used here in the proof of Theorem 2.3. Rectangle shapes represent blocks and circle shapes represent single nodes. All drawn edges have weight k^d, where we have omitted edges inside the blocks. Pairs of edges beginning at the same point inside a block are its exit edges and a pair of edges leading to two different points in a block are its entry edges. Observe that the construction is planar, using that the blocks are planar.

explored block lead to a vertex v_1. This vertex is identified with the starting vertex of three further blocks, i.e., v_1 is incident to 8 edges of weight k^d (two exit edges and 6 entry edges). The agent starts exploring these and, similarly as before, we build chains of blocks. Let M' denote the number of blocks in P_1', i.e., P_1' consists of $M'-1$ explored blocks and one partially explored block. Let M'' be the maximum number of blocks contained in any of the three chains starting from v_1. We stop the process when $M' + M'' = M$ and the agent traverses a new exit edge. Let P_1'' be the chain starting from v_1 of length M''. When the process stops, there is exactly one block that was entered, but not explored: If the last block that was traversed is in P_1', this partially explored block is in P_1'', and if the last block that was explored is in P_1'', the partically explored block is in P_1'. We complete it to some valid block that $\mathcal{A}_d$ constructs (for some algorithm) and call this block B_1^*. We let the edges of the last blocks in P_1' and P_1'' lead to the same vertex u_1.

We obtain a cycle of blocks consisting of P_1, P_1', P_1'', which we refer to as C_1 and, from vertex v_1, there are two chains of blocks with unexplored ends of length less than M. We iterate the procedure by ignoring the explored cycle C_1 and interpreting v_1 as the new starting vertex v_s. This is repeated until we obtain N such cycles of blocks. Then, we have two chains of blocks with unexplored ends beginning from v_N. As soon as their total numbers of blocks sum up to $2M$, we let the exit edges of the last blocks in these chains be connected to a closing vertex v_t. We call this last cycle of blocks between v_N and v_t cycle C^*. Observe that the construction is planar using that the blocks constructed by $\mathcal{A}_d$ are planar.

Analysis. First, note that every block, except for the blocks B_i^* for each i, was only entered via an entry edge before it was explored. For each of these blocks B, the agent incurs a cost of $\textsc{Alg}_B$ during the first exploration. Next, we will argue

that almost every block will be traversed a second time. More precisely, we claim the following, where we use that all blocks have the same block length $2k^d$.

Claim 2.33. *The agent incurs a total cost of at least $N(2M-1)\cdot 2k^d$ for backtracking blocks.*

Proof. We only argue that the algorithm incurs a cost of $(2M-1)\cdot 2k^d$ for backtracking in C_1 and it will be clear that this holds similarly for all other cycles $C_2, \dots, C_N$ (but not for C^*). We distinguish two cases: $M' + M'' = M$ was triggered by (1) exploration of a block in P_1' or (2) by exploration of a block in P_1''.

In case (1), observe that the agent already backtracked all M blocks in P_1 at the time when $M' + M'' = M$ was triggered, so that it incurred a cost of at least $M \cdot k^d$. Then, the agent has to travel to v_1 to reach cycle C_2. Since it has to return to the starting position when all vertices were explored, it needs to traverse a path from v_1 to v_s at the end. The shortest such path is to again traverse all blocks in P_1 so that the agent incurs another cost of $M \cdot 2k^d$ for backtracking in C_1.

In case (2), observe that the agent backtracked all blocks in $P_1' \setminus \{B_1^*\}$ at the time when $M' + M'' = M$ was triggered. To proceed with the exploration, the agent has to travel to v_1 and the shortest way for this is by backtracking all blocks in chain P_1'', so we can charge this cost by assuming that these blocks were backtracked. At the end, the agent has to return to the starting position, for which it needs to travel from v_1 to v_s and the shortest way for this is by backtracking all blocks in chain P_1. Together, we obtain that, in either case, we can assume that the agent backtracks all blocks in $C_1 \setminus \{B_1^*\}$ and we have $|C_1 \setminus \{B_1^*\}| = (2M-1)$. □

The claim implies that, for every block, except for B_i^* ($j \in \{1, \dots, N\}$) and the blocks in C^*, the agent incurs a cost of $\textsc{Alg}_B$ for the first traversal, and a cost of $k^d = l_B$ for backtracking. We obtain

$$\begin{aligned}\textsc{Alg}(G) &\geq \sum_{j=1}^{N} \sum_{B \in C_j \setminus \{B_j^*\}} \textsc{Alg}_B + l_B = \sum_{j=1}^{N} \sum_{B \in C_j \setminus \{B_j^*\}} \textsc{AlgC}_B + \textsc{Opt}_B \\ &\geq (4-\varepsilon) \cdot \sum_{j=1}^{N} \sum_{B \in C_j \setminus \{B_j^*\}} \textsc{Opt}_B, \end{aligned} \tag{2.11}$$

where we have use Lemma 2.32 and that k and d are large enough (depending on ε).

At the same time, in the offline optimum, the agent only needs to explore every block once and never backtracks a block. We obtain

$$\begin{aligned}\textsc{Opt}(G) &= \sum_{j=1}^{N} \left(3k^d + \sum_{B \in C_j} \textsc{Opt}_B \right) + \sum_{B \in C^*} \textsc{Opt}_B + 2k^d \\ &\leq \sum_{j=1}^{N} \left(\sum_{B \in C_j \setminus \{B_j^*\}} \textsc{Opt}_B \right) + (2N + 2M + 1)(d+2)k^{d-1}, \end{aligned} \tag{2.12}$$

where, for the last inequality, we have estimated each of $\text{OPT}_B, 3k^d, 2k^d$ by $(d+2)k^d$ (for $B = B_j^*$ and $B \in C^*$) and used that C^* consists of $2M$ blocks.

Note that

$$\frac{(2N+2M+1)(d+2)k^d}{\text{ALG}(G)} \leq \frac{(2N+2M+1)(d+2)k^d}{N \cdot (2M-1)2k^d} \to 0 \ \ (M, N \to \infty),$$

where we have used the safe lower bound $\text{ALG}(G) \geq N(2M-1)l_B = N(2M-1)2k^d$.

By (2.11) and (2.12), we obtain together with the last inequality for the competitive ratio that $\text{ALG}(G)/\text{OPT}(G) \to (4-\varepsilon)$ as $M, N \to \infty$. Since $\varepsilon > 0$ was chosen arbitrary, we obtain that the competitive ratio is at least 4 on planar graphs. Combining with the fact that any planar construction can be made subcubic (Corollary 2.6), this completes the proof of Theorem 2.3. □

Remark 2.34. It is easy to see that any algorithm that only backtracks when there is no boundary edge incident to the agent's position, is 4-competitive on the constructed graph. By careful observation, one can see that, for such algorithms, we have $z_B = \Theta(k^{d-1})$, in particular, $\text{OPT}_B = 2k^d + \Theta(k^{d-1})$, and we have $\text{ALG}_B \leq 6k^d$ for every block construced by $\mathcal{A}_d$. In G, then every block is backtracked at most once, so that we obtain in total an upper bound of $4 \cdot \text{OPT}(G)$.

Note that, for example, DFS fulfills this property on the constructed graphs. This shows that our analysis for this construction is tight.

2.6 Outlook

The key question in online graph exploration is whether the problem admits a constant-competitive algorithm [75]. While this problem remains open, our results suggest steps towards resolving this question.

First, we have shown that several assumptions can be made about the problem without affecting its competitive ratio (Corollary 2.6). These include restrictions on the agent, such as learning only the identifiers of boundary edges rather than their endpoints, as well as restrictions to subcubic graphs and to graphs satisfying the triangle inequality. This understanding can serve as a "sanity check" in future research: If one aims to develop a new algorithm, the agent should not rely on information or abilities that it does not possess under these restrictions. Conversely, if one aims to establish a lower bound, one can impose these restrictions on the agent to simplify the construction. However, for example, relying on violations of the triangle inequality is not a promising approach.

Second, we have established a general lower bound of 4 on the competitive ratio of the online graph exploration problem (Theorem 2.3). Our construction primarily builds on ideas already present in the literature [23, 44]. We believe that we have pushed these ideas to their limits, and that overcoming the barrier of 4 requires new approaches. Roughly speaking, the approach in [23] and in our work is to take the blocks constructed in [44], reuse them recursively, and arrange these blocks within an unweighted graph. The unweighted graph used for the block arrangement is

structured such that (almost) every block must be traversed twice by the online agent, but only once by the offline agent. Since the competitive ratio of exploration on unweighted graphs is at most 2, there is no further room for improvement in this part of the construction. Moreover, because the blocks themselves are already constructed using recursion, with the recursion depth taken to infinity, there is likewise no further room for improvement within the block construction. Also, it is important to note that our construction is planar, and it is known that the competitive ratio on planar graphs is at most 16 [75], so a substantially different construction would be needed for a non-constant lower bound.

If one aims to prove a non-constant lower bound on the competitive ratio of online graph exploration, our results for minor-free graphs (Theorem 2.2) imply that it is necessary to consider graph classes that include all minors. This suggests that studying dense high-girth graphs or expanders [82] could be a promising direction.

In terms of upper bounds for the online graph exploration problem on general graphs, not even a competitive ratio of $o(\log n)$ has been achieved so far, and our results eliminate $\textsc{Blocking}_\delta$ as a candidate for this for most values of δ (Theorem 2.26). It only remains to close the gap between $\delta \in o(\log(n)/\log\log(n))$ and $\delta \in \Omega(\log(n))$. Moreover, the lower bound constructions for $\textsc{Blocking}_\delta$ in our work and in [92] rely heavily on unfavorable tie-breaking by the algorithm: Whenever an edge is unblocked, we assume that the agent explores it immediately, even if lighter boundary edges are available at its current position. An interesting question for future research is whether handling unblocked edges more effectively could lead to an algorithm with a competitive ratio of $o(\log n)$.

However, we believe that proving such a result requires ideas beyond those presented in this work. Our main approach for establishing a constant competitive ratio on minor-free graphs relied on the fact that these graphs contain light spanners. Since it is known that general graphs and graphs of bounded expansion do not admit light spanners [95, Theorem 6.6], new techniques are necessary to generalize this result.

Regarding spanners, we gave an improved upper bound on the lightness of spanners in bounded-genus graphs. It is a natural question whether our bound is already tight for $g \geq 1$ or can further be improved. In particular, it is unclear whether the worst-case lightness for a fixed stretch must depend linearly on g.

Chapter 3

Collaborative tree exploration

We now turn to studying the online graph exploration problem with multiple agents. More precisely, in the *collaborative graph exploration problem*, we are coordinating a team of k agents tasked with traversing an initially unknown, undirected, connected graph $G = (V, E, w)$ with non-negative edge weights $w\colon E \to \mathbb{R}_{\geq 0}$. Similarly as in the single-agent setting, we assume that every vertex and every edge has a unique identifier. Upon visiting a vertex for the first time, the identifiers of the adjacent vertices as well as the identifiers and weights of the corresponding edges are revealed. Importantly, we assume here that agents can communicate globally, i.e., all agents have the same information. For simplicity, we simply say that there exists a central coordinator that gathers all information and determines each agent's movements. Initially, they are all located at the same starting vertex and we say that a vertex is *explored* if it has been visited by at least one of them. We assume that each agent can move at unit speed and the time needed to traverse an edge is simply its weight. The task is to explore all vertices, subject to minimizing the completion time. An example of an instance where all edges have weight 1 is illustrated in Figure 3.1. Note that, in the case $k = 1$, the completion time equals the total traveled distance since waiting does not serve any benefit in this case. Therefore, collaborative exploration is indeed a generalization of the single-agent graph exploration problem.

In the collaborative exploration problem, we usually denote competitive ratios using asymptotic notation. Therefore, we do not make a difference between the strict and non-strict competitive ratio. For simplicity, we do not require the agents to return to the starting position after exploring the tree. Note that this does not make a difference for the competitive ratio either when using asymptotic notation because returning to the starting position takes at most $\textsc{Opt}(G)$ time units.

It turns out that finding a good strategy in the collaborative case is highly non-trivial. Recall that, for the single-agent online graph exploration problem, DFS is 1-competitive on trees and 2-competitive on general graphs, where no algorithm with a better competitive ratio exists [96]. By contrast, for the collaborative case, it is already unclear how to coordinate the agents on a graph as simple as an unweighted tree. Already our small example in Figure 3.1 shows that the strict competitive ratio on unweighted trees is at least 2. In fact, a lower bound

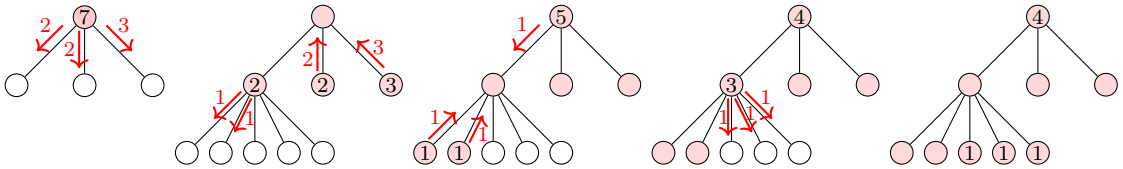

Figure 3.1: Example of an instance for collaborative tree exploration with $k = 7$ agents. The filled vertices are explored and the unfilled vertices are learned but not yet explored. The numbers in the vertices indicate the number of agents located in that vertex. The red arrows together with the numbers indicate where the agents move in the next step. The illustrated online algorithm requires 4 time steps and it is easy to see that the offline optimum requires only 2 time steps.

of $\Omega(\log(k)/\log\log(k))$ is known for the competitive ratio of collaborative exploration on an unweighted tree on n vertices with $k \leq n\log^c n$ agents for any constant c [43, 46].

For this reason, research typically revolves around finding good strategies for unweighted trees. Recall that *unweighted* means here that all edges have weight 1. This special case is called the *collaborative tree exploration problem*. In this chapter, we study a classical algorithm for the problem called Yo* introduced by Ortolf and Schindelhauer [98]. We give a refined version called RecYoYo and give a slightly improved bound on its competitive ratio.

Theorem 3.1. *The algorithm RecYoYo is* $\log(k) \cdot 2^{\mathcal{O}(\sqrt{\log(D)\cdot\log\log(k)})}$*-competitive for collaborative tree exploration on trees of depth* D*.*

Note that, for $k \geq D^\varepsilon$ for some $\varepsilon > 0$, we have $\log(D) = \mathcal{O}(\log(k))$ so that we obtain $\mathcal{O}(\sqrt{\log(D)\cdot\log\log(k)}) = \mathcal{O}(\sqrt{\log(k)\cdot\log\log(k)}) = o\big(\sqrt{\log^2(k)}\big) = o(\log(k))$. Therefore, for $k \geq D^\varepsilon$, the given bound in the Theorem is in $2^{o(\log(k))} = k^{o(1)}$.

Our result improves on the bound of $2^{\mathcal{O}(\sqrt{\log(D)\cdot\log\log(k)})}\cdot\log(k)\cdot(\log(k)+\log(n))$ for Yo* proven in [98]. The strength of our result is that we eliminate the dependency on n in the competitive ratio. However, we remark that this should be understood as an improvement on Yo*, but not on the best known competitive ratio for the problem in general. More precisely, observe that, for $k \geq n^\varepsilon$, we have $\log(n) = \mathcal{O}(\log(k))$, so in this range for k, our improvement is a factor of $\log(k)$. If k is polylogarithmic in n, i.e., $k \leq \log^C(n)$ for some constant C, consider the following two cases: First, if $D \geq n^\varepsilon$ for some $\varepsilon > 0$, we obtain that the exponent in our bound is $\sqrt{\log(D)\log\log(k)} \geq \Omega(\sqrt{\log(n)}) \geq \omega(\log(k))$, so that our bound is $2^{\omega(\log k)} = k^{\omega(1)}$. Thus, the bound is weaker than the trivial $\mathcal{O}(k)$ bound on the competitive ratio (we explain in the following section how the $\mathcal{O}(k)$ bound is obtained). Second, assume that D is also polylogarithmic in n. An algorithm is known that explores trees in $\mathcal{O}((n/k) + kD)$ rounds [38]. When both k and D are at most polylogarithmic in n, this algorithm is even constant-competitive.

3.1 Preliminaries

When considering collaborative graph exploration on unweighted graphs, we consider time to be discrete and, in every time step, each agent may traverse an edge. We also refer to a time step as a *round*. Throughout this chapter, T denotes a tree that is to be explored, n denotes its number of vertices, and k denotes the number of agents. We consider T to be rooted in the starting position of the agents and we denote its depth by D.

First, note that we have $\textsc{Opt}(G) \geq D$ because no agent can reach a vertex of depth D in less than D rounds. In addition, we have $\textsc{Opt}(G) \geq n/k$ because, in every round, at most k new vertices are explored. Next, note that the following offline algorithm explores T in at most $D + \lceil 2n/k \rceil$ rounds: Consider a depth-first-search traversal on T. The length of such a traversal is $2(n-1) \leq 2n$. We equally partition this tour into k paths and assign an agent to each of them. For each agent, traveling to the starting vertex of the path takes at most D rounds and exploring the path takes at most $\lceil 2n/k \rceil$ rounds. Together, we obtain that

$$D + \lceil 2n/k \rceil \geq \textsc{Opt}(G) \geq \max\{D, n/k\} \geq \frac{1}{2} \cdot (D + n/k).$$

In particular, we have $\textsc{Opt}(G) = \Theta(D + n/k)$.

Next, observe that the online algorithm that performs a DFS traversal with a single agent and leaves all other agents idle in the root takes at most $2n$ rounds. Using that $\textsc{Opt}(G) \geq n/k$, we obtain that its competitive ratio is at most $2k = \mathcal{O}(k)$. Considerable efforts have been made to improve upon this bound [37, 38, 41, 43, 46, 56, 67], which we summarize in the following.

State-of-the-art. It is important to note that, when considering the competitive ratio of the collaborative tree exploration problem for some fixed n and different values for k, it turns out that the competitive ratio first increases in k and then decreases again. In fact, it is known that, for $k \geq Dn^{1+\varepsilon}$, a constant-competitive algorithm exists [41]. On the other hand, it is known that the competitive ratio is at least $\Omega(\log(k)/\log\log(k))$ for $k \leq n \log^c n$ for any constant c [43, 46].

It is in general challenging to derive upper bounds on the competitive ratio without additional constraints on the number of agents. The first improvement over the trivial bound of $\mathcal{O}(k)$ was achieved in [56], where a $\mathcal{O}(k/\log(k))$-competitive algorithm was given. In [37], an algorithm with competitive ratio $\mathcal{O}(k/\exp(\sqrt{\log(k)}))$ was given and this was later improved to a $\mathcal{O}(\sqrt{k})$-competitive algorithm [38].

When allowing additional constraints on the number of agents, the already mentioned results give the following. We obtain a constant-competitive algorithm when $k \geq Dn^{1+\varepsilon}$ [41]. For $k \geq n^{\varepsilon}$ for some fixed $\varepsilon > 0$, the bound in [98] gives a competitive ratio of $2^{\mathcal{O}(\sqrt{\log(D) \cdot \log\log(k)})} \cdot \log(k) \cdot (\log(k) + \log(n)) = 2^{o(\log(k))} = k^{o(1)}$.

To summarize, the domain where no upper bound of $k^{o(1)}$ on the competitive ratio is known yet, is given when $k \leq n^{o(1)}$ and $D \geq n^{1-o(1)}$.

3.2 A slightly improved bound for Yo*

We give an improved version of the algorithm Yo* introduced in [98] and we call our reinterpretation RecYoYo. Similarly to [98], the main idea is to do a recursion over algorithms. This means that, given an algorithm Alg, called the *base algorithm*, we construct a new algorithm Alg′ with a better competitive ratio. We then apply this repeatedly until we obtain no further improvement on the competitive ratio. The "base case" can be, for example, the simple algorithm Yo-yo introduced in [98]: Let m_i denote the number of nodes at depth i. In the Yo-yo algorithm, we execute, for each $i = 1, \dots, D$, $\lceil m_i/k \rceil$ *phases*, where in a *phase* for depth i, every agent travels from the root to an unexplored vertex at depth i and then back to the root. In [98], it was proven that this algorithm is $4D$-competitive.

Before diving into the algorithm RecYoYo, let us introduce the notation needed for this. Recall that we consider T to be rooted at the starting vertex. For a vertex v of T, we denote by $d(v)$ the depth of v in T, that is, the length of the unique path from v to the root. By $T^{(v)}$, we denote the subtree of T rooted at v, and for $d \in \mathbb{N}$, we denote by T_d the subtree of T only containing vertices of depth at most d.

Similarly as in the case for single-agent graph exploration, we say that a vertex is *learned* if its parent was explored. Note that, in the case of trees, the distance of two learned vertices u and v does not decrease during the course of exploration because the unique path connecting them is internally explored if u and v are learned.

3.2.1 The algorithm RecYoYo

We turn to defining our procedure of obtaining the algorithm Alg′ given a base algorithm Alg. Before we give the formal description, we outline the main ideas for the algorithm.

Intuitive description of Alg′

Given a base algorithm Alg with competitive ratio depending on the depth (such as Yo-yo), we aim to decrease this dependence on D by dividing the tree into several layers (see Figure 3.2) and apply the base algorithm on several trees of smaller depth. We divide a tree of depth D into b layers, each of depth a, where the last layer may have depth less than a, i.e., we have $\lceil D/b \rceil = a$. In the first layer, we apply the base algorithm. Then, we explore the tree layer by layer. Importantly, we handle every new layer together with the previously explored one. These two layers build a forest of partially explored subtrees of depth at most $2a$ (cf. Figure 3.2).

If there are more such subtrees than agents, we assign every agent to a subtree and they explore these in parallel using DFS, which is optimal for exploration with a single agent. Since the size of the subtrees may substantially differ, it does not make sense that agents who were assigned a smaller subtree, and therefore finish exploration earlier, wait for too long for the other agents to complete exploring their subtree. Therefore, when half of the agents are done, the idle ones are redistributed to subtrees to which no agent has been assigned yet. For simplicity of the analysis,

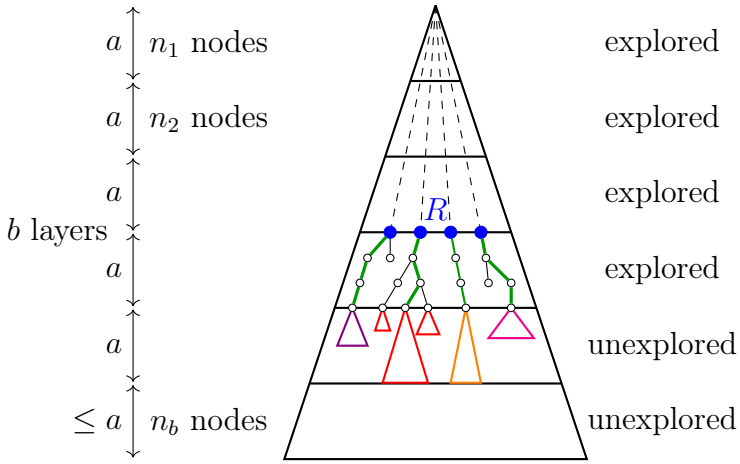

Figure 3.2: Illustration of the algorithm RecYoYo. The figure shows the set R with $j = 4$. The set R consists of vertices at depth $(j-1) \cdot a$ that have unexplored descendents. The triangles in the fifth layer indicate the unexplored subtrees. Those drawn in the same color are part of the same subtree $T_{2a}^{(v)}$ for some $v \in R$.

we assume that the non-idle agents do not move during the redistribution of the agents that have finished their DFS traversal.

If there are less subtrees than agents, some subtrees are assigned multiple agents and then it is non-straightforward how to explore these. In this case, we use the base algorithm. Similarly as before, we redistribute agents when a large enough portion of them have finished exploration. The difference of our algorithm to Yo* in [98] lies in this step: We choose a slightly different rule for when to redistribute, which allows us to spend less rounds in this step.

Note that, since each new layer is handled together with the previously explored one, some parts of each layer (except for the last one) will actually be explored twice. In Figure 3.2, the green paths are already explored but need to be traversed again for the exploration of the subsequent layer. However, the strength of handling multiple layers together is that this decreases the numer of connected components in the forest of unexplored subtrees. This property is captured later in Observation 3.3. The smaller number of trees allows to redistribute the agents less often.

Formal definition of Alg′

The algorithm Alg′ given a base algorithm Alg is formally defined in Algorithm 3. Note that, in this formulation of the algorithm, we require that only trees of a given depth D are used as input, which we think of as a parameter of the algorithm. We remark that one can eliminate the assumption that the depth is known beforehand using a doubling-strategy. That is, one starts executing the algorithm setting D to some constant and whenever we learn that the actual depth of the tree exceeds D, we double the value and restart the algorithm. Note that the algorithm is then restarted at most $\log(D)$ times (where D refers here to the actual depth of the tree)

and that an additional factor of $\log(D) = 2^{\log\log(D)}$ in the bound $2^{\mathcal{O}(\sqrt{\log(D)\log\log(k)})}$ can be hidden in the constant factor of the $\mathcal{O}$-notation.

In addition, the algorithm is parameterized by a and b, which are chosen such that $\lceil D/b \rceil = a$. The algorithm uses the subroutine DistributeAgents(A, S), where A is a set of agents and S is a set of learned vertices. This routine distributes the agents evenly on S, i.e., every agent in A is assigned precisely one vertex of S such that each vertex $v \in S$ is assigned $\lfloor |A|/|S| \rfloor$ or $\lceil |A|/|S| \rceil$ many agents. Then the routine moves the agents to their assigned vertices (on the unique path in the tree). The invocation DistributeAgents(S) only giving a set of vertices S corresponds to DistributeAgents(A, S) with A being the set of all agents. The subroutine ParallelDFS$_{2a}$ will only be invoked when no agent is located in a descendant of another agent's position. Then the agents perform in parallel depth-first-search on the subtrees $T_{2a}^{(v_i)}$, where v_i is the position of the i-th agent when this routine is invoked. We say that agent i is *done* when it has explored $T_{2a}^{(v_i)}$.

Algorithm 3: Alg'(D, a, b)

input: unexplored tree T of depth D

explore T_a using base algorithm Alg
for $j = 1, \ldots, b-1$ **do**
  $R \leftarrow \{v \in T : d(v) = (j-1)a, v \text{ has an unexplored descendant}\}$
  $A \leftarrow$ set of all agents *// set of idle agents*
  $R' \leftarrow R$ *// set of vertices in R that were not yet assigned an agent*
  while $|R| > k$ **do**
    DistributeAgents(A, R')
    perform ParallelDFS$_{2a}$ until $\lceil k/2 \rceil$ of the agents are done
    $A \leftarrow$ done agents
    $R \leftarrow R \setminus \{v \in R : T_{2a}^{(v)} \text{ is explored}\}$
    $R' \leftarrow R \setminus \{v \in R : \text{ an agent is located in } T^{(v)}\}$
  while $R \neq \emptyset$ **do**
    DistributeAgents(R)
    perform in parallel base algorithm Alg on $T_{2a}^{(v)}$ for each $v \in R$ until $\lceil |R|/2 \rceil$ of these subtrees are explored
    $R \leftarrow R \setminus \{v \in R : T_{2a}^{(v)} \text{ is explored}\}$

First, we observe that the algorithm indeed explores all vertices. The fact that it terminates will become clear from the runtime analysis.

Observation 3.2. *After iteration j of the for-loop ($j \in \{1, \ldots, b-1\}$), all vertices of depth at most $(j+1) \cdot a$ are explored. All vertices of larger depth are unexplored. In particular, Alg' is correct (if it terminates).*

Proof. We inductively show the statement. First, note that by correctness of the base algorithm, every node of depth at most a was explored before the first iteration

of the for-loop. Now, assume that, given some $j \in \{1, \dots, b-1\}$, all vertices until depth at most $j \cdot a$ are explored and let v be a vertex with $d(v) \in [j \cdot a + 1, (j+1) \cdot a]$. Let v' be its unique ancestor at depth $(j-1) \cdot a$. Then $v \in T_{2a}^{(v')}$. The vertex v' has an unexplored descendent and, by induction hypothesis, v' is explored. Therefore, we have $v' \in R$ after the execution of line 3. A vertex u is only removed from R if $T_{2a}^{(u)}$ is explored. Since an iteration of the for-loop only ends when $R = \emptyset$, we obtain that $T_{2a}^{(v')}$, and therefore also v, is explored at the end of the iteration of the for-loop. □

3.2.2 Analysis of RecYoYo

Throughout this section, fix a tree T of depth D and we analyze the number of rounds that Alg' takes on T. Later, we conclude our analysis of RecYoYo, which we obtain by recursively applying the procedure.

For this, let n_j denote the number of vertices in the *j-th layer*, that is, the number of vertices of depth $d(v) \in [(j-1) \cdot a, j \cdot a - 1]$ (see Figure 3.2). Similarly, we say that an edge is *in the j-th layer* if its parent vertex is in the j-th layer, where the parent vertex of an edge is the endpoint that is closer to the root. Moreover, let

$$R_j := \{v \in T : d(v) = (j-1) \cdot a, \text{the depth of } T^{(v)} \text{ is at least } a+1\}.$$

Note that R_j is precisely the set R initialized in line 3 of the j-th iteration of the for-loop. To see this, note that, by Observation 3.2, at the beginning of the j-th iteration when R is initialized, no vertex of depth more than $j \cdot a$ is explored, but all vertices of depth at most $j \cdot a$ are. Therefore, a vertex v of depth $(j-1) \cdot a$ has an unexplored descendant if and only if it has a descendent at depth more than $j \cdot a$. This is the case if and only if the depth of $T^{(v)}$ is at least $a+1$.

We can link the sizes of the R_j to n and a as follows.

Observation 3.3. *We have $\sum_{j=1}^{b-1} |R_j| \leq n/a$.*

Proof. Every vertex $v \in \bigcup_j R_j$ has, by definition, a descendant at depth $d(v) + a$. Let P_v be a path leading from v to such a descendant (these are depicted in green in Figure 3.2). Note that these paths are edge-disjoint and have length at least a. Using that T contains $n-1$ edges, the statement follows. □

Next, we estimate the number of rounds that Alg$'$ takes depending on the competitive ratio of the base algorithm Alg.

Theorem 3.4. *Assume that the base algorithm Alg is $f(D,k)$-competitive on trees of depth D using k agents, where f is monotonically increasing in D and k. Then Alg$'$ is $\mathcal{O}(\log(k) \cdot (b + f(2a, k)))$-competitive. Importantly, the constant factor hidden in the $\mathcal{O}$-notation is independent of n, k, D, a, b.*

Proof. For each line of the algorithm, we estimate the number of rounds that are spent during execution of that line. Note that lines 3–5, 9–11, and 15 are each

carried out in zero time steps, so we only estimate the number of rounds spent during execution of the other lines. The invocation in the first line takes, by assumption, at most $f(a,k)\cdot(a+2|T_a|/k) \le f(a,k)\cdot(a+2n/k)$ rounds, where we have used that the offline optimum for T_a is at most $a+2|T_a|/k$.

Next, we estimate for $j \in \{1,\dots,b\}$, the number of rounds spent in the first while-loop, i.e., in lines 6–11. After each iteration of the while-loop, the size of R is decreased by at least $k/2$ so that the loop is executed at most $|R_j|/(k/2) = 2|R_j|/k$ times. In each of these iterations of the while-loop, the cost incurred in line 7 is at most $2D$. Therefore, the number of rounds spent in line 7 is at most $4D\cdot|R_j|/k$.

During the execution of line 8, only edges in layers j and $j+1$ are traversed. Note that none of the edges is traversed by two different agents because, in this while-loop, no two agents are assigned to explore the same subtree. Since in a DFS traversal, an edge is traversed twice, we obtain that at most $2(n_j+n_{j+1})$ edge traversals occur, where we have used that there are at most n_j edges in each layer j. Moreover, during each round spent in line 8, at least $k/2$ agents make progress in the sense that they traverse an edge. Therefore, the number of rounds spent in line 8 (over all iterations of the while-loop) is at most $2(n_j+n_{j+1})/(k/2) = 4(n_j+n_{j+1})/k$.

Summing up over all j, we obtain that the overall time steps spent in the first while-loop (lines 6–11) is at most

$$\begin{aligned}&\sum_{j=1}^{b-1} 4\frac{n_j+n_{j+1}}{k} + 4D\cdot\frac{|R_j|}{k} \le 8\frac{n}{k} + 4\frac{D}{k}\cdot\sum_{j=1}^{b-1}|R_j| \overset{\text{Obs.3.3}}{\le} 8\frac{n}{k} + 4\frac{D}{k}\cdot\frac{n}{a}\\ &= \frac{n}{k}\cdot\left(8+4\frac{D}{a}\right) \le \frac{n}{k}\cdot(8+4b),\end{aligned} \tag{3.1}$$

where we have used for the last inequality that $D/a = D/\lceil D/b\rceil \le D/(D/b) = b$.

We turn to estimating the number of rounds spent in the second while-loop, i.e., in lines 12–15. When the while-loop is entered, we have $|R| \le k$ and, after each iteration, the size of R is halved. Therefore, it is executed at most $\log(k)+1$ times. In each iteration, line 13 is executed in at most $2D$ time steps. Therefore, we spend at most $2D\cdot(\log(k)+1)$ rounds during the execution of line 13.

Consider a single iteration of the second while-loop. To estimate the cost in line 14, let $T_1,\dots,T_{|R|}$ be the subtrees of depth $2a$ on which the base algorithm is executed. On each of these subtrees, the base algorithm can explore the entire subtree in at most $f(2a,\lfloor k/|R|\rfloor)\cdot(2a+2|T_i|/\lfloor k/|R|\rfloor)$ rounds. In the following, let $\text{median}_i(|T_i|)$ denote the median of $|T_1|,\dots,|T_{|R|}|$. The number of rounds until half of the trees $T_1,\dots,T_{|R|}$ are explored is at most

$$\begin{aligned}&f(2a,\lfloor k/|R|\rfloor)\cdot\left(2a+\frac{2\cdot\text{median}_i(|T_i|)}{\lfloor k/|R|\rfloor}\right)\\ &\le f(2a,k)\cdot\left(2a+\frac{4|R|\cdot\text{median}_i(|T_i|)}{k}\right),\end{aligned} \tag{3.2}$$

where we have use that $\lfloor k/|R|\rfloor \ge k/(2|R|)$ and that f is increasing in the second

variable. Further, observe that

$$\frac{|R|}{2} \cdot \text{median}_i(|T_i|) \leq \sum_{i=1}^{|R|} |T_i| \leq n_j + n_{j+1}. \tag{3.3}$$

Plugging this into (3.2), we obtain that the number of rounds spent in line 14 in each iteration of the while-loop is at most

$$f(2a,k) \cdot \left(2a + 8\frac{n_j + n_{j+1}}{k}\right). \tag{3.4}$$

We obtain that the total cost incurred in lines 13 and 14 is at most

$$\begin{aligned}
&\sum_{j=1}^{b-1} (\log(k)+1) \cdot \left(2D + f(2a,k) \cdot \left(2a + \frac{8}{k}(n_j + n_{j+1})\right)\right) \\
&\leq (\log(k)+1) \cdot \left(Db + f(2a,k) \cdot \left(2ab + \frac{8}{k}\sum_{j=1}^{b-1} n_j + n_{j+1}\right)\right) \\
&\leq \underbrace{(\log(k)+1)}_{\leq 2\log(k)} \cdot \left(Db + f(2a,k) \cdot \left(2D + \frac{16n}{k}\right)\right) \\
&\leq 16 \cdot \log(k) \cdot (b + f(2a,k)) \cdot (D + 2n/k)
\end{aligned}$$

Together with (3.1) and the fact that the first line takes at most $f(a,k)\cdot(a+2n/k)$ time steps, we obtain that ALG$'$ terminates in the following number of time steps

$$\begin{aligned}
&(f(a,k) + 16\log(k) \cdot (b + f(2a,k))) \cdot (D + 2n/k) + (8 + 4b) \cdot \frac{n}{k} \\
&\leq \mathcal{O}(\log(k) \cdot (b + f(2a,k))) \cdot \frac{1}{2}(D + n/k).
\end{aligned}$$

Since the offline optimum is at least $(D+n/k)/2$ and the constant in the $\mathcal{O}$-notation is independent of n, k, D, a, b, this completes the proof of the Theorem. □

Next, note that $a = \lceil D/b \rceil \leq 2D/b$ as long as $b \leq D$. In particular, in the algorithm ALG$'$, the base algorithm is invoced on trees of depth at most $4D/b$ and the bound in the Lemma can be reformulated as $\mathcal{O}(\log(k) \cdot (b + f(2a,k))) \leq \mathcal{O}(\log(k) \cdot (b + f(4D/b,k)))$.

We now define RECYOYO as follows: Recall that we assume that D is known beforehand. We set $b = 2^{\Theta(\sqrt{\log(D)\log\log(k)})}$ and apply the procedure recursively. Importantly, in every recursion, we use the same value for b (i.e., b is not adapted for recursive invocation with lower depth). Note that, in every recursion, we decrease the depth of the trees on which the base algorithm is invoced by at least a factor of $b/4$. We let the recursion end when we reach trees of some constant depth and then simply use Yo-yo as the base algorithm. Note that we obtain a recursion depth of $\Theta(\log_{(b/4)}(D)) = \Theta(\log(D)/\log(b))$.

In the analysis, we will use an arbitrary fixed value for b and then see that the optimal choice is $b = 2^{\Theta(\sqrt{\log(D)\log\log(k)})}$.

Proof of Theorem 3.1. Let $f(D,k)$ denote the competitive ratio of RecYoYo on trees of depth D with k agents. By Theorem 3.4, there exists a constant $C \geq 1$ independent of n, k, D, a, b such that $f(D,k) \leq C\log(k)(b + f(4D/b, k))$. Let C' denote the constant threshold such that RecYoYo on trees of depth at most C' is simply Yo-yo. Note that we have $f(C',k) = \mathcal{O}(1)$ because Yo-yo is $4D$-competitive on trees of depth D. In the following, let $r = \Theta(\log(D)/\log(b))$ denote the obtained recursion depth. We have

$$\begin{aligned} f(D,k) &\leq C\log(k)\left(b + f\left(\frac{D}{(b/2)}, k\right)\right) \leq C\log(k)\cdot b + C^2\log^2 k\left(b + f\left(\frac{D}{(b/2)^2}, k\right)\right) \\ &\leq \mathcal{O}(1)\cdot\sum_{i=1}^{r} b(C\log(k))^i \leq \mathcal{O}(1)\cdot b\cdot(C\log(k))^{r+1} \\ &= \log(k)\cdot 2^{\mathcal{O}(\log(b)+\log\log(k)\cdot\log(D)/\log(b))} \end{aligned}$$

Setting $b = 2^{\Theta(\sqrt{\log(D)\log\log(k)})}$, the assertion follows. □

Chapter 4

Colored Euclidean TSP

We turn to studying an offline mutli-agent variant of the traveling salesperson problem. In short, in the *k-colored Euclidean traveling salesperson problem (k*-ETSP*)*, k sets of points have to be covered by k disjoint curves in the plane (see Figure 4.1 for an example). This is a fundamental problem in geometric network optimization [94] and generalizes the well-known *Euclidean traveling salesperson problem (ETSP)*. It captures applications ranging from VLSI design [50, 88, 110, 109] to set visualisation of spatial data [3, 32, 48, 69, 102].

Formally, an instance of the k-ETSP is a partition $(T_c)_{c\in C}$ of a finite set of *terminals* $T \subseteq \mathbb{R}^2$ in the Euclidean plane, where $|C| = k$. We consider every $c \in C$ to be a *color* and every point in T_c to be of color c. A solution to the instance is a k-tuple $\Pi = (\pi_c)_{c\in C}$ of closed curves in $\mathbb{R}^2$, also referred to as *tours*, such that every curve π_c visits all terminals of color c, i.e., $T_c \subseteq \pi_c$, and the curves are pairwise disjoint, i.e., $\pi_c \cap \pi_{c'} = \emptyset$ for $c \neq c'$. In the notation used here, we identify curves with their images. We will explain later why a solution always exists (Section 4.1). The objective of the k-ETSP is to minimize the total length of the tours, i.e., to minimize $l(\Pi) := \sum_{c\in C} l(\pi_c)$, where $l(\pi)$ denotes the Euclidean length of π.

Note that the ETSP corresponds to the k-ETSP with $k = 1$ and is thus a special case of the k-ETSP for any $k \geq 1$. Since the ETSP is known to be NP-hard [99], this also holds for the k-ETSP. In this chapter, we study polynomial-time approximations for the k-ETSP and prove the following result.

Theorem 4.1. *For every $\varepsilon > 0$, there exists an algorithm that computes a $\left(\frac{5}{3}+\varepsilon\right)$-approximation for the* 3-ETSP *in time $\left(\frac{n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)}$.*

Since this is the first work to explicitly study the 3-ETSP, all previously known bounds on the best possible approximation ratio are inherited from work on other problems. The best such bound is 10/3 [18], which is inherited from the colored Steiner tree problem (see Section 4.1 for a definition).

We believe that our approach is applicable for a wider range of non-crossing problems, for instance for the *red-blue-green-yellow separation problem* (cf. [45]), where the task is to draw curves that separate the points of different colors.

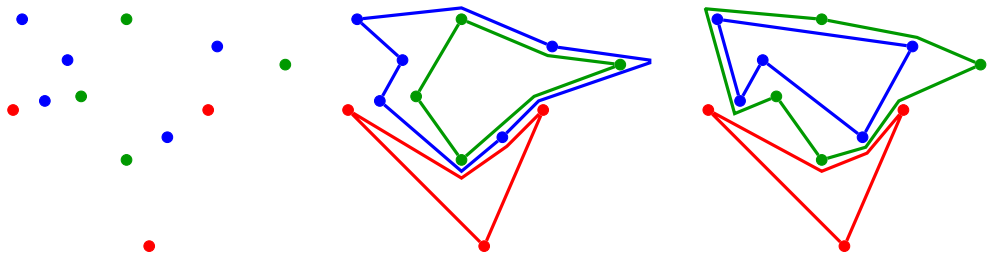

Figure 4.1: An instance of the 3-ETSP together with two possible solutions. An optimum solution does not exist: The curves can get arbitrarily close but must not touch. Observe that, in the middle subfigure, the red and green tour are not δ-close for any $\delta > 0$, as the blue tour lies in between, but in the right subfigure, they are.

4.1 Preliminaries

The *non-crossing Euclidean Steiner forest problem*, introduced in [48], is closely related to the k-ETSP. In this setting, an instance is of the same form as in the k-ETSP, and the goal is to find k non-crossing trees embedded in the Euclidean plane, each covering all points of a color. The objective is to minimize the total weight of the trees, that is, the sum of the lengths of the line segments in the forest. The case $k = 1$ is simply referred to as the *Euclidean Steiner tree problem*.

It is easy to see that every instance of the Euclidean Steiner forest problem admits a solution. This follows from the fact that a tree embedded in the Euclidean plane cannot separate any two points, as it contains no cycles. Moreover, any solution to this problem can be transformed into a solution to the k-ETSP by having each agent perform a DFS traversal of its corresponding tree. Since each edge is traversed twice, the total length of the resulting tours is twice the weight of the forest. Conceptually, instead of traversing each edge twice, one can think of replacing each edge with two infinitesimally close edges leading to the same terminal, thereby creating a non-self-intersecting cycle for each color. We refer to this transformation as "doubling" the Steiner trees. In particular, this implies that every instance of the k-ETSP admits a feasible solution.

Importantly, the Euclidean Steiner forest problem is NP-hard, and the best known bound on the approximation factor achievable in polynomial time is $5/3$ [18]. Since the optimum cost of the k-ETSP is lower bounded by the optimum cost of the Steiner forest problem, the "doubling" approach of the Steiner forests yields a $10/3$-approximation algorithm for k-ETSP. In this work, we improve on this by providing a $(5/3 + \varepsilon)$-approximation algorithm.

Next, we remark on the issue of the existence of an optimum for k-ETSP. It is important to note that, for $k > 1$, an optimum does not always exist (cf. Figure 4.1). This is because the tours of different colors are not allowed to touch, that is, in any valid solution, the tours need to keep some distance $\varepsilon > 0$, which can be further decreased to still obtain a valid solution of potentially shorter length. In order to still define an approximation, we follow the approach of [45] by defining the value $\textsc{Opt} := \inf\{l(\Pi) : \Pi \text{ is a solution}\}$ and saying that a solution Π is

an α-approximation if $l(\Pi) \leq \alpha \textsc{Opt}$. In this notation, we omit the dependence of Opt on the given instance for brevity, as the specific instance will always be clear from context.

In the study of approximation algorithms, a classical question is what the best-possible approximation ratio is that can be achieved with a polynomial-time algorithm. When any ratio can be achieved, we say that there exists a *polynomial-time approximation scheme (PTAS)*. More formally, this is an algorithm that receives as input $\varepsilon > 0$ and an instance, and outputs a $(1+\varepsilon)$-approximation in time $n^{\mathcal{O}_\varepsilon(1)}$, where the notation $\mathcal{O}_\varepsilon$ indicates that the constant factor hidden in the $\mathcal{O}$-notation may depend on ε. In other words, for every fixed $\varepsilon > 0$, a $(1+\varepsilon)$-approximation algorithm exists. If, in the running time, ε does not appear in the exponent of n, that is, the running time is of the form $f(\varepsilon) \cdot n^{\mathcal{O}(1)}$, we say that we even have an *efficient polynomial-time approximation scheme (EPTAS)*.

It is well-known that the ETSP admits a PTAS [5], which was gradually improved [101, 15] to an EPTAS [80] with the running time proven tight under the gap-ETH, that is, the *exponential time hypothesis*, which is the conjecture that NP-hard problems cannot be solved in time $2^{o(1)}$. The result was recently extended to an EPTAS for 2-ETSP [45]. However, for $k \geq 3$, the situation turns out to be substantially more complicated and it is wide open what the optimal achievable approximation ratio is. In particular, it is unclear whether a PTAS exists, which is arguably the most important question for the k-ETSP.

Problem 4.2. *Is there a PTAS for the k-ETSP for $k \geq 3$?*

Similarly, it is known that for the 2-colored Euclidean Steiner forest problem, a PTAS exists [18], but the case for 3 or more colors is open.

Next, we give an overview of the key ideas and intermediate steps that we take to prove Theorem 4.1.

Outline for the proof of Theorem 4.1

To prove Theorem 4.1, we adapt Arora's algorithm for Euclidean TSP [5]. One of the key ingredients of that algorithm is the so-called *Patching Lemma*, which allows to locally modify any tour such that the number of crossings with a line segment is bounded, without increasing the length of the tour too much. It was shown in [45] that this is still possible for two tours, but it does not seem to be possible for more than two non-crossing tours (see Figure 4.2, [18, 45]). We show how to circumvent this issue by imposing an additional condition on the tours to be patched. For this, we say that two tours are *δ-close* if they can be connected by a straight line segment of length at most δ that is disjoint from the third tour (cf. Figure 4.1). We will only modify the tours in a δ'-neighborhood of the segment, which is defined as the set of all points that have distance at most δ' to some point on the segment.

Lemma 4.3. *Let s be a straight line segment and $\delta = l(s)$ be its length. Let a solution to the* 3-ETSP *be given in which two of the three tours are not δ-close. For every $\delta' > 0$, the solution can be modified inside a δ'-neighborhood of s such that it intersects s in at most* 18 *points and its cost is increased by at most $\mathcal{O}(\delta)$.*

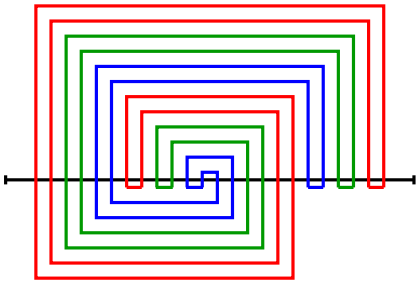

Figure 4.2: A modified example from [18] that is presumably non-patchable.

For the case where patching is not possible, we take a different approach. For this, we define a *two-tour presolution* to be a pair of disjoint tours $(\pi_{cc'}, \pi_{c''})$ such that $\pi_{cc'}$ visits all terminals colored c and c', and $\pi_{c''}$ visits all terminals colored c''. Such tours can easily be transformed into a feasible solution to 3-ETSP by "doubling" $\pi_{cc'}$ (cf. Figure 4.3, Observation 4.7). We call the resulting solution an *induced two-tour solution*. The following Lemma implies that, if the additional condition in our patching lemma for three colors is not fulfilled, finding induced two-tour presolutions is a valid approach.

Lemma 4.4. *For every $\varepsilon > 0$, there exists $\delta > 0$ such that, for every $(1+\varepsilon)$-approximate solution to* 3-ETSP *in which the two shorter tours are δ-close, we can find a two-tour presolution (π_1, π_2) with $2l(\pi_1) + l(\pi_2) \leq \left(\frac{5}{3} + 2\varepsilon\right) \cdot \textsc{Opt}$.*

Next, similarly as in [5], we place a suitable grid on the plane and place so-called *portals* on the grid lines (cf. Section 4.3.1). Roughly speaking, a solution to the k-ETSP is *portal-respecting* if it only intersects grid lines at portals and intersects every portal at most a constant number of times (see Section 4.3.1 for a formal definition). Combining the ideas in [5] with Lemmas 4.3 and 4.4, we obtain the following result.

Theorem 4.5. *For every instance of* 3-ETSP *and $\varepsilon > 0$, either, there is a solution that is a $(1+\varepsilon)$-approximation and portal-respecting with respect to a suitable grid, or there is a portal-respecting two-tour presolution that induces a $\left(\frac{5}{3} + \varepsilon\right)$-approximation.*

Theorem 4.5 allows us to restrict ourselves to finding portal-respecting solutions. The last step is to show that such solutions can be computed in polynomial time using dynamic programming. For this, we generalize the approach in [45] to any number of colors k while simultaneously allowing for weighted tours. We denote this version of the problem by k-ETSP$'$ (see Section 4.4 for a formal definition). In the following, Opt denotes the minimum cost of a solution to k-ETSP$'$, that is, the shortest possible weighted length of a portal-respecting solution.

Theorem 4.6. *For every $k \in \mathbb{N}$, there is a polynomial-time algorithm that computes a parametric solution $\Pi(\lambda)$ to k-ETSP$'$ such that $\lim_{\lambda \to 0} l(\Pi(\lambda)) = \textsc{Opt}$.*

Here, we work with parametric solutions because, as explained above, a solution of length Opt does not necessarily exist. More precisely, a *parametric solution* is a

function $\Pi\colon (0,\infty) \to \{\Pi' : \Pi' \text{ is a solution}\}$ that continuously interpolates between solutions. In this, *continuously* can be interpreted, e.g., with respect to the Fréchet-distance on the space of curves, which is defined as follows: For $\pi_1, \pi_2\colon [0,1] \to \mathbb{R}^2$, the Fréchet distance between π_1 and π_2 is $d_{\mathrm{Fr}}(\pi_1,\pi_2) := \sup_{t\in[0,1]} \|\pi_1(t) - \pi_2(t)\|$. Intuitively, the algorithm of Theorem 4.6 computes the optimal combinatorial structure of a solution, i.e., the optimal order in which portals and terminals are visited or bypassed, and the parameter $\lambda > 0$ sets the minimal spacing between the tours, which we let converge to 0. Importantly, our solution allows to efficiently recover the (non-parametric) solution $\Pi(\lambda)$ for given $\lambda > 0$ and to compute Opt.

Further related work

Other problems in geometric network optimization include the following: In the *k-traveling repairperson problem*, we can use k tours (that are allowed to intersect) to cover the terminals, subject to minimizing the latency, i.e., the sum of the times at which a terminal is visited [33, 35, 51]. In the *traveling salesperson problem with neighborhoods*, the task is to find a shortest tour that visits at least one point in each of a set of neighborhoods [63, 91, 93, 108].

Moreover, the TSP has been extensively studied for other metric spaces, apart from the Euclidean case considered in this work. For example, it is known that there is a PTAS in the case of a metric space of bounded doubling dimension [16, 59]. On the other hand, it is known that a PTAS for general metric spaces does not exist with the best known lower bound on the achievable approximation factor being 123/122 [79]. Currently, the best approximation algorithm known in general metric spaces was suggested by Karlin et al. [76, 77], achieving an approximation factor of $1.5 - 10^{-36}$.

4.2 Two-tour presolutions

Recall that a *two-tour presolution* for 3-ETSP is a pair of non-crossing tours $(\pi_{cc'}, \pi_{c''})$ such that $\pi_{cc'}$ visits all terminals in $T_c \cup T_{c'}$ and $\pi_{c''}$ visits all terminals in $T_{c''}$ for some $\{c, c', c''\} = \{\mathrm{R}, \mathrm{G}, \mathrm{B}\}$, where $\{\mathrm{R}, \mathrm{G}, \mathrm{B}\}$ denotes throughout this chapter the color set C in the case of 3-ETSP, standing for red, green, and blue. In this section, let $c'' = \mathrm{B}$ without loss of generality. We first investigate how two-tour presolutions can be transformed into solutions to the 3-ETSP.

For this, note that, if we are given a single tour π_{RG} that visits all red and green terminals, it is possible to replace it by two parametrized disjoint tours $\pi_{\mathrm{R}}(\lambda)$ and $\pi_{\mathrm{G}}(\lambda)$ that fulfill the following: They have Fréchet-distance at most λ from π_{RG}, the tour $\pi_{\mathrm{R}}(\lambda)$ visits all red terminals and $\pi_{\mathrm{G}}(\lambda)$ visits all green terminals, and we have $\lim_{\lambda\to 0} l(\pi_{\mathrm{R}}(\lambda)) = \lim_{\lambda\to 0} l(\pi_{\mathrm{G}}(\lambda)) = l(\pi_{\mathrm{RG}})$ (cf. Figure 4.3). Choosing $\lambda > 0$ small enough and considering the constructed tours $\pi_{\mathrm{R}}(\lambda), \pi_{\mathrm{G}}(\lambda)$, we obtain the following.

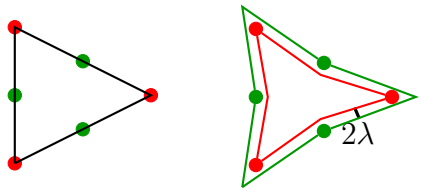

Figure 4.3: On the left, we have a single curve π_{RG} visiting all red and green points. On the right, we have replaced π_{RG} by two parametrized disjoint curves $\pi_{\mathrm{R}}(\lambda)$ and $\pi_{\mathrm{G}}(\lambda)$ with Fréchet-distance at most λ to π_{RG}, visiting the terminals of the corresponding color. In particular, the Fréchet-distance between π_{R} and π_{G} is at most 2λ.

Observation 4.7. *Fix an instance of the* 3-ETSP *and let $\pi_{\mathrm{RG}}, \pi_{\mathrm{B}}$ be a two-tour presolution. For every $\delta > 0$, there exists a solution to the* 3-ETSP *of cost at most $2 \cdot l(\pi_{\mathrm{RG}}) + l(\pi_{\mathrm{B}}) + \delta$, called an* induced two-tour solution.

Next, we show that, if the two shorter tours of a $(1+\varepsilon)$-approximation for 3-ETSP are in some sense close to each other, then there is a good two-tour presolution. However, note that this is not a reduction to 2-ETSP: In 2-ETSP, the objective is to minimize $l(\pi_{\mathrm{B}}) + l(\pi_{\mathrm{RG}})$. In our case, we need to minimize $l(\pi_{\mathrm{B}}) + 2 \cdot l(\pi_{\mathrm{RG}})$, i.e., we need to solve a weighted variant of 2-ETSP. We will see later that we can compute a $(1+\varepsilon)$-approximation for this weighted variant of 2-ETSP in polynomial time (cf. Theorem 4.17).

Recall that two tours are δ-close if they can be connected by a straight line segment of length at most δ that does not intersect the third tour.

Lemma 4.8. *Let $\Pi = (\pi_{\mathrm{R}}, \pi_{\mathrm{G}}, \pi_{\mathrm{B}})$ be a solution to a given instance of the* 3-ETSP *and let $\delta > 0$. W.l.o.g., let π_{B} be the longest tour, i.e., $l(\pi_{\mathrm{B}}) \geq l(\pi_{\mathrm{R}}), l(\pi_{\mathrm{G}})$. Assume that π_{R} and π_{G} are δ-close. Then there is a two-tour presolution $(\pi_{\mathrm{RG}}, \pi_{\mathrm{B}})$ with*

$$2 \cdot l(\pi_{\mathrm{RG}}) + l(\pi_{\mathrm{B}}) \leq \frac{5}{3} \cdot l(\Pi) + 8\delta.$$

Proof. The construction is illustrated in Figure 4.4. Since π_{R} and π_{G} are δ-close and terminals are finitely many distinct points, we can pick points $\boldsymbol{x} \in \pi_{\mathrm{R}}$ and $\boldsymbol{y} \in \pi_{\mathrm{G}}$ such that they do not equal any terminal, i.e., $\boldsymbol{x}, \boldsymbol{y} \notin T$, and the straight line segment $s := \overline{\boldsymbol{xy}}$ has length at most $1.5 \cdot \delta$, and s does not intersect the tours in any other points than its endpoints, i.e., $s \cap \pi_{\mathrm{B}} = \emptyset$, $s \cap \pi_{\mathrm{R}} = \{\boldsymbol{x}\}$, and $s \cap \pi_{\mathrm{G}} = \{\boldsymbol{y}\}$. (Observe that because of the additional condition that the endpoints of s do not lie on terminals, there is not necessarily such a segment of length at most δ.)

In the following, given a closed curve π and two points $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \pi$, by $\pi[\boldsymbol{x}_1, \boldsymbol{x}_2]$, we denote the shorter of the two subcurves of π that connects $\boldsymbol{x}_1$ with $\boldsymbol{x}_2$ (ties can be broken arbitrarily). Observe that it is possible to pick two points $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \pi_{\mathrm{R}}$ close enough to $\boldsymbol{x}$ and $\boldsymbol{y}_1, \boldsymbol{y}_2 \in \pi_{\mathrm{G}}$ close enough to $\boldsymbol{y}$ such that $\pi_{\mathrm{R}}[\boldsymbol{x}_1, \boldsymbol{x}_2]$ and $\pi_{\mathrm{G}}[\boldsymbol{y}_1, \boldsymbol{y}_2]$ do not contain any of the terminals, i.e., $T \cap (\pi_{\mathrm{R}}[\boldsymbol{x}_1, \boldsymbol{x}_2] \cup \pi_{\mathrm{G}}[\boldsymbol{y}_1, \boldsymbol{y}_2]) = \emptyset$, and the straight line segments $\overline{\boldsymbol{x}_1\boldsymbol{y}_1}, \overline{\boldsymbol{x}_2\boldsymbol{y}_2}$ have length at most $2 \cdot \delta$, are nonintersecting, do not intersect π_{B}, and only intersect $\pi_{\mathrm{R}}, \pi_{\mathrm{G}}$ in $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{y}_1, \boldsymbol{y}_2$ (cf. left side of Figure 4.4).

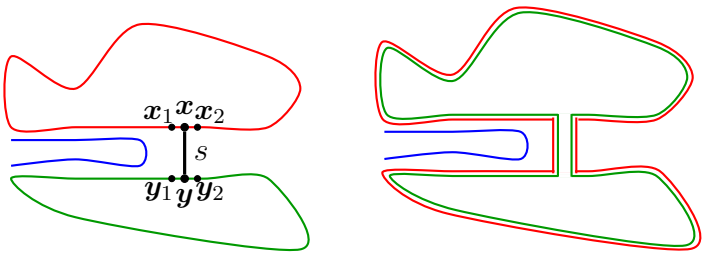

Figure 4.4: On the left, we are given a solution to 3-ETSP where the red and green tour are δ-close. On the right, we see how the solution can be transformed into an induced two-tour solution.

Therefore, the curve $\pi_{\mathrm{RG}} := (\pi_{\mathrm{R}} \setminus \pi_{\mathrm{R}}[\boldsymbol{x}_1, \boldsymbol{x}_2]) \cup \overline{\boldsymbol{x}_1\boldsymbol{y}_1} \cup (\pi_{\mathrm{G}} \setminus \pi_{\mathrm{G}}[\boldsymbol{y}_1, \boldsymbol{y}_2]) \cup \overline{\boldsymbol{x}_2\boldsymbol{y}_2}$, which is illustrated on the right side of Figure 4.4, forms a closed curve that visits all red and green points and does not intersect π_{B}. Moreover, we have

$$\begin{aligned} l(\pi_{\mathrm{B}}) + 2l(\pi_{\mathrm{RG}}) &\leq l(\pi_{\mathrm{B}}) + 2l(\pi_{\mathrm{R}}) + 2l(\pi_{\mathrm{G}}) + 8\delta \\ &= (l(\pi_{\mathrm{R}}) + l(\pi_{\mathrm{G}}) + l(\pi_{\mathrm{B}})) + (l(\pi_{\mathrm{R}}) + l(\pi_{\mathrm{G}})) + 8\delta \\ &\leq \frac{5}{3} \cdot (l(\pi_{\mathrm{R}}) + l(\pi_{\mathrm{G}}) + l(\pi_{\mathrm{B}})) + 8\delta, \end{aligned}$$

where we have used in the last inequality that $l(\pi_{\mathrm{B}}) \geq l(\pi_{\mathrm{R}}), l(\pi_{\mathrm{G}})$. □

Note that applying Lemma 4.8 to a $(1+\varepsilon)$-approximation with $\delta \leq \varepsilon \mathrm{OPT}/24$ gives a two-tour presolution $(\pi_{\mathrm{RG}}, \pi_{\mathrm{B}})$ with

$$2 \cdot l(\pi_{\mathrm{RG}}) + l(\pi_{\mathrm{B}}) \leq \frac{5}{3} \cdot (1+\varepsilon) \cdot \mathrm{OPT} + 8\delta = \left(\frac{5}{3} + \frac{5}{3}\varepsilon\right) \cdot \mathrm{OPT} + 8\delta \leq \left(\frac{5}{3} + 2\varepsilon\right) \cdot \mathrm{OPT},$$

which completes the proof of Lemma 4.4.

4.3 Our structure theorem

In this section, we prove our structure theorem for 3-ETSP (cf. Theorem 4.5) which, roughly speaking, states the following: For every $\varepsilon > 0$, either, there is a two-tour presolution that induces a $\left(\frac{5}{3} + \varepsilon\right)$-approximation, or there is a $(1+\varepsilon)$-approximate solution that fulfills some additional constraints (or both). Later, we will see that it is possible to find a good solution that fulfills these additional constraints and a good two-tour presolution in polynomial time.

Our final algorithm for the 3-ETSP will preprocess the input such that the terminals remain distinct points and have integer coordinates. This allows us to assume throughout this section that terminals lie in $\{0, \ldots, L\}^2$ for some integer L that is a power of 2. As the problem is not interesting for small L, we also assume $L \geq 4$ whenever necessary. In Section 4.5, we explain in more detail how we preprocess

the input and show that a near-optimal solution to the preprocessed input can be transformed in polynomial time to a near-optimal solution to the original input.

Following [45], we assume without loss of generality that, for every $\varepsilon > 0$ and $\delta > 0$, there is a $(1+\varepsilon)$-approximate solution to the 3-ETSP whose tours consist of straight line segments, where each segment connects two points that each are at distance at most δ from a terminal. To see this intuitively, interpret each tour of a solution as a sequence of terminals to visit or bypass. The cheapest way to realize such a sequence is by straight line segments with endpoints arbitrarily close to terminals (cf. Figure 4.1). For this reason, we will assume from now on that all tours that we work with consist of such straight line segments. Since we assume in this section that terminals lie in $\{0,\dots,L\}^2$, we have in particular, that the straight line segments have endpoints in $N_\delta(\{0,\dots,L\}^2)$, where $N_\delta(A)$ denotes the δ-neighborhood of a set A, that is, $N_\delta(A) := \{\boldsymbol{x} : \|\boldsymbol{x}-\boldsymbol{a}\| < \delta \text{ for some } \boldsymbol{a} \in A\}$.

4.3.1 Dissection and portals

In this subsection, we place a suitable grid on the Euclidean plane and place some portals on it through which the tours will later be allowed to cross the grid lines. For this, we follow the definitions as in [5].

Fix an instance of k-ETSP with $T \subseteq \{0,\dots,L\}^2$ where L is a power of two. We pick a *shift vector* $\boldsymbol{a} = (a_1, a_2) \in \{0,\dots,L-1\}^2$ and consider the square

$$C(\boldsymbol{a}) := \left[-a_1 - \frac{1}{2}, 2L - a_1 - \frac{1}{2}\right] \times \left[-a_2 - \frac{1}{2}, 2L - a_2 - \frac{1}{2}\right],$$

i.e., $C(\boldsymbol{a})$ is the square $[0,\dots,2L]^2$ shifted by $-\boldsymbol{a} - (0.5, 0.5)$. Note that $C(\boldsymbol{a})$ contains every terminal.

The *dissection* $D(\boldsymbol{a})$ is a full 4-ary tree defined as follows (illustrated in Figure 4.5): Each node is a square in $\mathbb{R}^2$. The root of $D(\boldsymbol{a})$ is $C(\boldsymbol{a})$. Given a node S of the tree of side length more than one, we partition S into four smaller equal-sized squares and these define the four children of S. If S has side length one, it is a leaf. Note that this is well-defined because L is a power of two.[1]

Given a square S, we define its *border edges* to be the unique four straight line segments bounding it and we define its *border* ∂S to be the union of the border edges.

A *grid line* is either a horizontal line containing $(0, -a_2 - 0.5 + k)$ or a vertical line containing $(-a_1 - 0.5 + k, 0)$ for some $k \in \{1,\dots,2L-1\}$. Note that every border edge of a square in $D(\boldsymbol{a})$ is either contained in a grid line or contained in a border edge of $C(\boldsymbol{a})$ (which is not on a grid line). Since terminals have coordinates in $\mathbb{Z}$ and grid lines have coordinates in $0.5+\mathbb{Z}$, no terminal lies on a grid line. More precisely, every terminal lies exactly in the center of a leaf of $D(\boldsymbol{a})$.

A *boundary* is a border edge of a non-root node in $D(\boldsymbol{a})$ not contained in another border edge (see Figure 4.5 for an example). Observe that its length is $\frac{2L}{2^i}$ for

[1] In previous work, the well-known *quad-trees* are defined as a subtree of the dissection, on which the dynamic program of [5] is based, which however we do not rely on in this work.

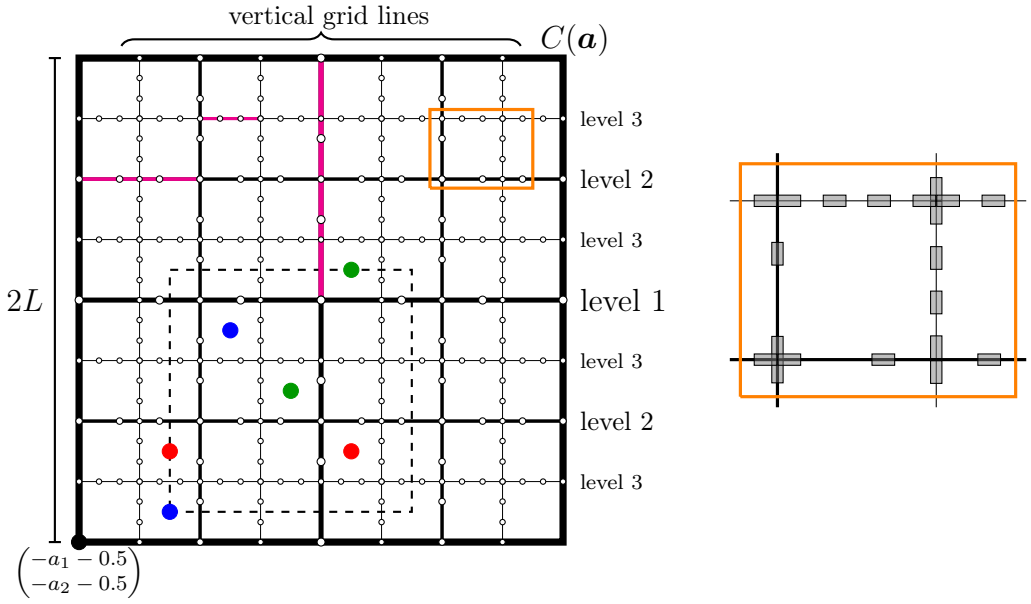

Figure 4.5: The figure on the left illustrates the dissection $D(\boldsymbol{a})$ with $L = 4$ and $\boldsymbol{a} = (1, 0)$. The dashed lines denote $\partial[0, L]^2$. The three pink lines are examples of boundaries of levels one, two, and three. The levels of all horizontal grid lines are indicated. We have placed 4 portals on every boundary, represented as circles. For better overview, we have drawn only one portal at endpoints of boundaries. Note that the endpoint of a boundary is actually contained in up to four portals. This is illustrated on the right hand side, where the exact portal placement of the marked orange area is given.

some $i \in \{1, \dots, \log(2L)\}$. Then we define its *level* as i. Note that a grid line only contains boundaries of the same level so we can define the level of a grid line as the level of the boundaries that it contains (cf. Figure 4.5). If two boundaries (or grid lines) of level i and j are given with $i < j$, we say that level j is *deeper* than level i (resembling the property that the corresponding boundary belongs to a node that is deeper in the dissection), and level i is *shallower* than level j.

Observe that there is precisely one vertical (respectively horizontal) grid line of level one and, for every $i \in \{1, \dots, \log(2L) - 1\}$, there are twice as many grid lines of level $i + 1$ as grid lines of level i. In total, there are $2L - 1$ horizontal and $2L - 1$ vertical grid lines. With this, we immediately obtain the following property.

Observation 4.9. *Let g be either a vertical line containing point $(k - 0.5, 0)$ or a horizontal line containing point $(0, k - 0.5)$ for some $k \in \{1, \dots, L\}$. Consider the dissection $D(\boldsymbol{a})$ for a vector $\boldsymbol{a} \in \{0, \dots, L - 1\}^2$ chosen uniformly at random. Then, g is a grid line with respect to $D(\boldsymbol{a})$, and, for every $i \in \{1, \dots, \log(2L)\}$, we have*

$$\Pr_{\boldsymbol{a}}(\textit{the level of } g \textit{ is } i) = \frac{2^{i-1}}{2L - 1}.$$

The next observation follows immediately from the fact that a square in $D(\boldsymbol{a})$

only contains smaller squares of $D(\boldsymbol{a})$ so that, in particular, it only contains boundaries of deeper levels (cf. Figure 4.5).

Observation 4.10. *Let $b = \overline{\boldsymbol{xy}}$ be a boundary of level i. If a grid line g crosses its interior $b^\circ := b \setminus \{\boldsymbol{x}, \boldsymbol{y}\}$, the level of g is at least $i + 1$. The levels of the grid lines crossing b through $\boldsymbol{x}$ and $\boldsymbol{y}$ are at most i.*

A *δ-portal* (or, in short, *portal*) on a straight line segment is a subsegment of length $\delta \in (0, 1)$. Given a segment s, we define $\mathrm{grid}(s, k, \delta)$ as the set of k equispaced δ-portals on s such that the endpoints of s are contained in the first and last δ-portal respectively.

We place portals on $D(\boldsymbol{a})$ as follows: We will choose a large enough integer $r \in \mathbb{N}$ (called the *portal density factor*) and $\delta > 0$ (the *portal length*) small enough. Then, for every boundary b, we place the portals $\mathrm{grid}(b, r \log(L), \delta)$ on b (cf. Figure 4.5). Note hereby that $\log(L) \in \mathbb{N}$ because L is a power of two. Observe that, on a deeper level boundary, portals are placed more densely, which will turn out to be a key property.

4.3.2 Snapping non-crossing curves to portals

In this subsection, we show that disjoint tours can be modified so that they only intersect grid lines in portals, without increasing their lengths too much. To prove this, we follow the same ideas as in [5, Section 2.2]. Nevertheless, the snapping technique in [5, Section 2.2] needs some adaptation to work in the setting of non-crossing curves. This technique was used in previous work for pairs of non-crossing tours [45] and for Steiner trees [18]. Here, we provide a unified framework for this technique, which may be of wider interest and can be applied to a variety of non-crossing Euclidean problems.

In the following, if $s = \overline{\boldsymbol{x}_1\boldsymbol{x}_2}$ is a straight line segment, we let $s^\circ := s \setminus \{\boldsymbol{x}_1, \boldsymbol{x}_2\}$. This allows us to specify more precisely where the segments are allowed to intersect. In particular, if we require that s_1° and s_2° are disjoint for two segments s_1 and s_2, they are allowed to share an endpoint.

Lemma 4.11. *Let $\mathcal{S} = \{s_1, \dots, s_m\}$ be a finite set of straight line segments in the Euclidean plane such that each s_i connects two points in $N_{\frac{1}{4}}(\{0, \dots, L\}^2)$ and assume $L \geq 4$. Choose a vector $\boldsymbol{a} \in \{0, \dots, L-1\}^2$ uniformly at random and consider the dissection $D(\boldsymbol{a})$. For every portal density factor $r \in \mathbb{N} \setminus \{0\}$, portal length $\delta \in (0, 1)$, and $\delta' > 0$, there is a set of curves $\mathcal{S}' = \{s_1', \dots, s_m'\}$ (not necessarily straight line segments) such that*

a) s_i' differs from s_i only in $N_{\delta'}(G)$ where G denotes the union of all grid lines in $D(\boldsymbol{a})$,

b) if the segments s_i° are pairwise disjoint, then the curves $(s_i')^\circ$ are pairwise disjoint as well,

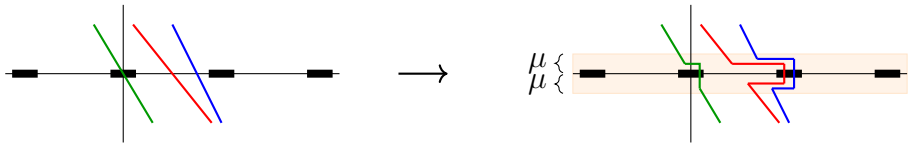

Figure 4.6: On the left, the red and blue tour cross a boundary outside of a portal and the green tour crosses in an intersection of grid lines. On the right, the tours are modified in a μ-Neighborhood of the grid line such that they are still non-crossing, only cross boundaries at portals and do not cross in intersections of grid lines.

c) every s_i' intersects every boundary b of $D(\boldsymbol{a})$ only in the portals $\text{grid}(b, r\log(L), \delta)$*, i.e., $s_i' \cap b \subseteq \text{grid}(b, r\log(L), \delta)$,*

d) no s_i' contains an intersection point of two grid lines, i.e., $g_1 \cap g_2 \cap s_i' = \emptyset$ for every $i \in \{1, \dots, m\}$ and grid lines $g_1 \neq g_2$,

e) the curves of $\mathcal{S}'$ intersect the grid lines in finitely many points,

f) $\mathbb{E}_{\boldsymbol{a}}[l(\mathcal{S}') - l(\mathcal{S})] \leq 7\sqrt{2} \cdot \frac{l(S)}{r}$, where $l(\mathcal{S}) := \sum_{s\in\mathcal{S}} l(s)$.

Proof. Throughout this proof, the segments of $\mathcal{S}$ are gradually modified to obtain the set of curves $\mathcal{S}'$ in the end. All the modifications satisfy the invariant that the current set of curves only intersects grid lines in finitely many points that we refer to as *crossings*. We use a parameter $\mu > 0$ throughout the proof, that will be carefully set later, in a way that the condition $\mu < \min\{\delta, \delta', 0.25\}$ is satisfied. We will modify the segments inside $N_\mu(G)$. This implies then condition a) and that the segments are not modified in $N_{\frac{1}{4}}(\{0, \dots, L\}^2)$, i.e., the segments are not modified in some neighborhood around their endpoints. This makes it a bit easier to prove part d) later because we can assume than that the segments are disjoint in the considered area where we make modifications.

We obtain the set $\mathcal{S}'$ from $\mathcal{S}$ as follows: Consider every boundary b of $D(\boldsymbol{a})$ one by one (in arbitrary order) and apply the following modifications: (Type 1) Move every crossing on b with $\mathcal{S}$ to the nearest portal on b as illustrated by the red and blue segments in Figure 4.6. (Type 2) If a crossing on b with $\mathcal{S}$ lies precisely on the intersection of two grid lines, move it inside the portal by at most μ as illustrated by the green segment in Figure 4.6.

Observe that we only modify the tours in $N_\mu(b)$ and that the modifications can be done such that we do not create additional crossings between the segments. Therefore, it is immediate that conditions a), b), and e) are fulfilled after this procedure.

To prove parts c) and d), we have to show that, once a boundary has been considered, we neither create additional crossings on this boundary outside of portals nor do we move existing crossings out of portals. Note that, when we apply a modification of Type 1 on a boundary b, the new lines created for redirecting a

crossing to a portal can create new crossings with grid lines perpendicular to b°. We argue that these new crossings lie in a portal: We have seen in Observation 4.10 that these lines are of deeper levels. In particular, if b' is a boundary perpendicular to b°, $b \cap b'$ is an endpoint of b' and, therefore, lies in a portal of b'. The new crossings created on b' are at distance at most $\mu < \delta$ from b, in particular from an endpoint of b', so that they are placed in a portal of b'.

In contrast, a modification of Type 2 on b can relocate a crossing on a boundary b' perpendicular to b (not only b°) but it does not increase the total number of crossings on b' (cf. green curve in Figure 4.6). We show that a boundary on which a crossing is relocated, has not been considered yet: Let $\boldsymbol{p}$ be a crossing on b that needs to be moved by a modification of Type 2. Then, $\boldsymbol{p}$ lies in the intersection of two or more (at most four) boundaries. Let b' be the last boundary considered before b that contains $\boldsymbol{p} = b \cap b'$. By construction of the modification procedure, after b' was considered, there were no crossings lying in the intersection of b' with any grid line. In particular, there was no crossing at $b \cap b'$. Since b' was the last boundary considered before b that contains $b \cap b'$, no crossing can be moved to $b \cap b'$ in the meantime. This gives a contradiction so that we obtain that no other boundary containing the crossing $\boldsymbol{p}$ was considered yet. To summarize, when a boundary b is considered, modifications of Type 1 and Type 2 do not create additional crossings outside of portals or relocate crossings on boundaries that were already considered. With this, we obtain that conditions c) and d) are fulfilled as well.

It is left to show that condition f) is satisfied. First, note that the cost of a modification of Type 2 is at most μ. Since we apply finitely many modifications of Type 2 (at most L^2), μ can be chosen small enough such that the total cost of modifications of Type 2 is at most $l(\mathcal{S})/r$. Next, as already explained, the new crossings created by a modification of Type 1 already lie in portals so that they do not need to be moved. Therefore, the total number of modifications of Type 1 we need to apply is at most the number of crossings of the unmodified set $\mathcal{S}$ with grid lines, i.e., $|G \cap \mathcal{S}|$ where $G \cap \mathcal{S} := \bigcup_{i=1}^{m} G \cap s_i$.

Consider some grid line g. Let i denote its level in $D(\boldsymbol{a})$. Recall that the length of a boundary lying on g is $\frac{2L}{2^i}$ and we place $r \log(L)$ portals on it. Therefore, the distance between two portals on g is at most $2L/2^i(r\log(L)-1)$. Hence, the cost of moving a crossing on g to the closest portal is also at most $\frac{2L}{2^i(r\log(L)-1)}$. Overall, the total cost of modifications of Type 1 on boundaries on g is in expectation at most

$$\begin{aligned}
&\sum_{i=1}^{\log 2L} \Pr(g \text{ has level } i) \cdot |g \cap \mathcal{S}| \cdot \frac{2L}{2^i(r\log(L)-1)} \\
&\overset{\text{Obs. 4.9}}{=} \sum_{i=1}^{\log 2L} \frac{2^{i-1}}{2L-1} \cdot |g \cap \mathcal{S}| \cdot \frac{2L}{2^i(r\log(L)-1)} \\
&= \frac{|g \cap \mathcal{S}|}{r} \cdot \frac{1}{2} \cdot \underbrace{\frac{2L}{2L-1}}_{\leq 2} \cdot \underbrace{\frac{\log 2L}{(\log(L) - \frac{1}{r})}}_{\leq 3} \leq 3 \cdot \frac{|g \cap \mathcal{S}|}{r},
\end{aligned}$$

where we have used in the last inequality that $L \geq 4$. Summing up over all grid

lines gives the following estimate on the total cost of modifications of Type 1

$$\sum_{g \text{ is a grid line}} 3 \cdot \frac{|g \cap \mathcal{S}|}{r} = 3 \cdot \frac{|G \cap \mathcal{S}|}{r}. \tag{4.1}$$

As the last step, we relate the number of crossings $|\mathcal{S} \cap G|$ to the total length of $\mathcal{S}$. For this, consider a straight line segment $s_i \in \mathcal{S}$ between two points in $N_{\frac{1}{4}}(\{0, \dots, L\}^2)$. Let $(x_1, y_1), (x_2, y_2) \in \{0, \dots, L\}^2$ be the closest points to the endpoints of s_i and consider the straight line segment $s_i'' := \overline{(x_1, y_1)(x_2, y_2)}$. Note that $|G \cap s_i''| = |G \cap s_i|$. If $(x_1, y_1) = (x_2, y_2)$, we have $G \cap s_i = G \cap s_i'' = \emptyset$, so assume that $(x_1, y_1) \neq (x_2, y_2)$ and, in particular, $l(s_i'') \geq 1$. By choice of $(x_1, y_1), (x_2, y_2)$, we have that the distance between the endpoints of s and s' is at most $1/4$ so that we obtain with triangle inequality

$$l(s_i'') \leq l(s_i) + 2 \cdot \frac{1}{4} \overset{l(s'') \geq 1}{\leq} l(s_i) + \frac{1}{2} \cdot l(s_i'')$$

so that $l(s_i'') \leq 2l(s_i)$. Therefore, it suffices to relate the number of crossings of s_i'' with grid lines to $l(s_i'')$.

Note that the length of s_i'' is $\sqrt{|x_1 - x_2|^2 + |y_1 - y_2|^2}$. Moreover, s_i'' crosses precisely $|x_1 - x_2|$ vertical grid lines and $|y_1 - y_2|$ horizontal grid lines. Hence,

$$\begin{aligned} |s_i'' \cap G|^2 &= \left(|x_1 - x_2| + |y_1 - y_2|\right)^2 \\ &= 2\left(|x_1 - x_2|^2 + |y_1 - y_2|^2\right) - \left(|x_1 - x_2| - |y_1 - y_2|\right)^2 \\ &\leq 2 \cdot \left(|x_1 - x_2|^2 + |y_1 - y_2|^2\right) = 2 \cdot l(s_i'')^2, \end{aligned}$$

so that $|s_i \cap G| = |s_i'' \cap G| \leq \sqrt{2} \cdot l(s_i'') \leq 2\sqrt{2} \cdot l(s_i)$. Combining this with (4.1), the total cost of modifications of Type 1 is at most

$$3 \cdot \frac{|G \cap \mathcal{S}|}{r} = \frac{3}{r} \sum_{i=1}^{m} |s_i \cap G| \leq \frac{3}{r} \sum_{i=1}^{m} 2\sqrt{2} \cdot l(s_i) = 6\sqrt{2} \cdot \frac{l(\mathcal{S})}{r}. \tag{4.2}$$

Recall that we have seen that the total cost of modifications of Type 2 is at most $l(\mathcal{S})/r$. Together with (4.2), we obtain

$$\mathbb{E}_{\boldsymbol{a}}[l(S') - l(S)] \leq 6\sqrt{2} \cdot \frac{l(\mathcal{S})}{r} + \frac{l(\mathcal{S})}{r} \leq 7\sqrt{2} \cdot \frac{l(\mathcal{S})}{r}. \qquad \square$$

4.3.3 The patching technique for three disjoint tours

In the previous section, we have seen that a (reasonable) solution to k-ETSP can be modified such that it only intersects grid lines in portals. In this section, we investigate how the tours can further be modified to reduce the number of intersection points per portal. This will be important for our algorithm because it considers all possible ways that a solution can cross the squares of $D(\boldsymbol{a})$ through the portals. To obtain a reasonable running time, we need a constant bound on the number of crossings. As briefly explained in the introduction, we cannot hope to show this for

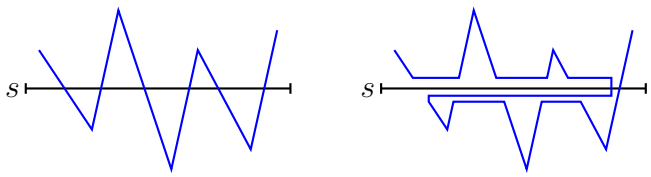

Figure 4.7: Illustration of the patching scheme in [5] for a single tour: On the left, we are given a tour π intersecting the segment s in six points. On the right, we see how the tour can be modified such that the number of crossings is at most two and the length is increased at most by $3 \cdot l(s)$.

every solution, even for $k = 3$ (cf. Figure 4.2). Therefore, we restrict ourselves to the 3-ETSP and show the desired properties for this problem under some additional assumptions.

Before delving into the proof, let us introduce some useful notation and establish the prerequisites. Let $\pi_{\mathrm{R}}, \pi_{\mathrm{G}}, \pi_{\mathrm{B}}$ be a solution to an instance of 3-ETSP and s be a straight line segment such that $\pi_c \cap s$ consists of finitely many distinct points for all $c \in \{\mathrm{R}, \mathrm{G}, \mathrm{B}\}$. Then we say that s is *non-aligned* to the solution and we call the points in $\bigcup_{c \in C} \pi_c \cap s$ *crossings*. We define an order on the crossings by rotating the plane such that s is parallel to the x-axis and then ordering them by their x-coordinates. This allows us to speak of a crossing to be "next to" or "in between" other ones. The *color* of a crossing $\boldsymbol{x}$, denoted $c(\boldsymbol{x})$, is d if $\boldsymbol{x} \subseteq s \cap \pi_d$. With this, we can classify the occurring patterns by sequences of the three colors and we call this a *crossing pattern*. For example, if $x_1, \ldots, x_6$ denote the ordered crossings, by the crossing pattern RRGGBB, we mean that $c(x_1) = c(x_2) = \mathrm{R}$, $c(x_3) = c(x_4) = \mathrm{G}$ and $c(x_5) = c(x_6) = \mathrm{B}$.

Our work builds on existing results for one and two tours. Arora [5] showed that the number of crossings of a single tour with a line segment can be reduced as follows (cf. Figure 4.7).

Lemma 4.12 (Arora [5])**.** *Let π be a closed curve and s be a non-aligned straight line segment. For every $\delta > 0$, there is a curve π' differing from π only inside $N_\delta(s)$ such that $|s \cap \pi'| \leq 2$ and $l(\pi') \leq l(\pi) + 3 \cdot l(s)$.*

Dross et al. [45] proved that the number of crossings between two disjoint tours and a straight line segment s can be reduced to a constant number, at additional cost $\mathcal{O}(l(s))$. For our purposes, we only need the two-color patching scheme for the special crossing pattern given in the following result. To see why the following lemma holds, one can carefully investigate the proof in [45] or observe that the scheme illustrated in Figure 4.8 works as desired.

Lemma 4.13 (Dross et al. [45])**.** *Let π_c, π_d be disjoint closed curves and s be a non-aligned straight line segment. Assume the crossing pattern is given by cddccdd. For every $\delta > 0$, there are disjoint closed curves π'_c, π'_d differing from π_c and π_d only inside $N_\delta(s)$ such that the new crossing pattern is cdd, and the total length of the modified tours fulfills $l(\pi'_c) + l(\pi'_d) \leq l(\pi_c) + l(\pi_d) + 4 \cdot l(s)$.*

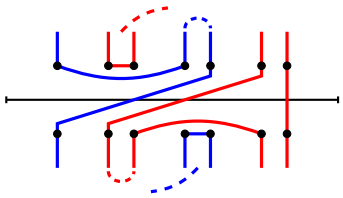

Figure 4.8: Illustration of the patching scheme for the crossing pattern BRRBBRR, see also [45]. Observe that the connections illustrated by dashed lines must exist (up to symmetry).

Now, we have all the prerequisites in place to give a patching procedure in the case that no red crossing is next to a green crossing on the considered segment (or for any other other choice of two colors), i.e., to prove Lemma 4.3.

We remark that our work is closely related to [18] and [45]. The main result of [18] is that Arora's patching lemma can be adapted to two Steiner trees, and three Steiner trees if one of them may use parts of another. The main contribution of [45] is that Arora's patching lemma can be adapted to two TSP tours. In contrast to these two results, our patching procedure needs to be more involved: In our setting, we have to deal with more complex crossing patterns whose mono-colored groups cannot be reduced to a single size (they were reduced to size 1 in [18] and 2 in [45]), and we have to ensure that the modified tours remain connected (which is more immediate in [18] and [45]) and that they remain Jordan curves (which is more immediate in [45] and not needed in [18]).

First, we give a more precise formulation of Lemma 4.3.

Lemma 4.14 (Tricolored Patching)**.** *Let $\pi_{\mathrm{R}}, \pi_{\mathrm{G}}, \pi_{\mathrm{B}}$ be disjoint closed curves and s be a non-aligned straight line segment. Assume that, in the crossing pattern on s, there is no red crossing next to a green crossing. Then for every $\delta > 0$, there are disjoint closed curves $\pi'_{\mathrm{R}}, \pi'_{\mathrm{B}}, \pi'_{\mathrm{G}}$ such that*

a) for all $c \in \{\mathrm{R}, \mathrm{G}, \mathrm{B}\}$, π_c differs from π'_c only inside $N_\delta(s)$,

b) $|(s \cap \pi'_{\mathrm{R}}) \cup (s \cap \pi'_{\mathrm{B}}) \cup (s \cap \pi'_{\mathrm{G}})| \leq 18$,

c) $l(\pi'_{\mathrm{R}}) + l(\pi'_{\mathrm{G}}) + l(\pi'_{\mathrm{B}}) \leq l(\pi_{\mathrm{R}}) + l(\pi_{\mathrm{G}}) + l(\pi_{\mathrm{B}}) + 75 \cdot l(s)$.

Proof. We modify the three curves in several steps to reduce the number of crossings with s. For this, we arrange the crossings into *groups*, where a group is a maximal set of consecutive monochromatic crossings. The *color of a group* $\mathcal{G}$, denoted $c(\mathcal{G})$, is then the color of the crossings in this group.

The road map for our proof is the following: First, we reduce the number of crossings inside a group by applying Lemma 4.12. Next, we further simplify the occurring patterns by applying Lemma 4.13. Then, we give a new patching scheme for the remaining occurring patterns for three colors.

Reducing the number of crossings per group. For every group $\mathcal{G}$ that contains more than two crossings, we apply the following modifications: Let $s'(\mathcal{G})$ be the subsegment of s beginning in the first crossing of $\mathcal{G}$ and ending in the last crossing of $\mathcal{G}$. Apply Lemma 4.12 to the tour of color $c(\mathcal{G})$ and the subsegment $s'(\mathcal{G})$. After this, we obtain three tours $\overline{\pi}_\mathrm{R}, \overline{\pi}_\mathrm{G}, \overline{\pi}_\mathrm{B}$ differing from $\pi_\mathrm{R}, \pi_\mathrm{B}, \pi_\mathrm{G}$ only in a δ-neighborhood of s such that each group contains at most two crossings and the total cost of the modifications is at most $\sum_\mathcal{G} 3 \cdot l(s'(\mathcal{G})) \leq 3 \cdot l(s)$. Therefore,

$$l(\overline{\pi}_\mathrm{R}) + l(\overline{\pi}_\mathcal{G}) + l(\overline{\pi}_\mathrm{B}) \leq l(\pi_\mathrm{R}) + l(\pi_\mathcal{G}) + l(\pi_\mathrm{B}) + 3 \cdot l(s). \tag{4.3}$$

Bounding the number of red and green groups with only one crossing. After the previous step, every group contains now either one or two crossings. Our next goal is to bound the number of red and green groups with only one crossing.

For this, note that, for every $c, c' \in \{\mathrm{R}, \mathrm{G}, \mathrm{B}\}, c \neq c'$ and every choice of two crossings $\boldsymbol{x}, \boldsymbol{y}$ of color c, the number of crossings of color c' in between them is even: The union of the subcurves $\overline{\pi}_c[\boldsymbol{x}, \boldsymbol{y}] \cup s[\boldsymbol{x}, \boldsymbol{y}]$ together form a closed curve, where $\overline{\pi}_c[\boldsymbol{x}, \boldsymbol{y}]$ denotes one of the two subcurves of $\overline{\pi}_c$ that connect $\boldsymbol{x}$ and $\boldsymbol{y}$. Since $\overline{\pi}_{c'}$ is also a closed curve, the number of intersection points in $(\overline{\pi}_c[\boldsymbol{x}, \boldsymbol{y}] \cup s[\boldsymbol{x}, \boldsymbol{y}]) \cap \overline{\pi}_{c'}$ must be even. Because $\overline{\pi}_c$ and $\overline{\pi}_{c'}$ are disjoint, the number of crossings in $\overline{\pi}_{c'} \cap s[\boldsymbol{x}, \boldsymbol{y}]$ must be even.

Next, note that, if we are given a red or green group $\mathcal{G}$, it can only have blue groups as neighbors because we assumed that no red crossing is next to a green crossing. If it is neighbored by two blue groups, the number of crossings in $\mathcal{G}$ must be even, i.e., two, because otherwise there is an odd number of crossings of color $c(\mathcal{G})$ in between these two blue groups. Therefore, if $\mathcal{G}$ has only one crossing, it has only one neighboring group, which means that it is either the first or the last group on the segment s. In particular, there are at most two red or green groups consisting of only one crossing.

Patching bichromatic patterns. In the next step, we eliminate alternating sequences of red and blue, respectively green and blue groups. As already explained, a bichromatic red-blue or green-blue crossing pattern cannot contain RBR or GBG, so it suffices to be able to patch the patterns $BRRBBRR$ and $BGGBBGG$. For this, consider an arbitrary red-blue (or green-blue) bichromatic pattern and apply the following modifications. While the crossing pattern contains one the above patterns ($BRRBBRR$ or $BGGBBGG$), pick the leftmost starting such pattern. Let s' be the shortest subsegment containing the chosen pattern (i.e., beginning in the first crossing and ending in the last crossing of the chosen pattern) and apply Lemma 4.13 to s'. The resulting pattern is then BRR, respectively BGG, and the cost of this step is at most $4l(s')$. Additionally, move the at most three resulting crossings sufficiently close to the rightmost endpoint of s', which gives an additional cost of at most $6l(s')$. The cost of modifications for a single choice of the pattern is then at most $10 \cdot l(s')$.

Let $\boldsymbol{x}$ be the leftmost crossing in s' after the modifications and $\boldsymbol{y}$ be the right endpoint of s', i.e., $\boldsymbol{x}$ is sufficiently close to $\boldsymbol{y}$. Note that, after the modifications,

none of the two possible patterns can start left of $\boldsymbol{x}$ and none of the patterns can entirely lie left of $\boldsymbol{y}$. This is because the chosen pattern was leftmost. Therefore, if a new subsegment s_1' is chosen in the next step, it starts right of $\boldsymbol{x}$ or in $\boldsymbol{x}$ and ends right of $\boldsymbol{y}$. In particular, when applying the procedure on s_1', the crossings can be moved close enough to the right endpoint of s_1' so that they do not lie in s'. Then the subsegment s_2' chosen after s_1' does not intersect s'. This means that the sequence s_i' of the s' chosen in the modifications above is such that each point on s is contained in at most two of the s_i'. Therefore, the total cost of the modifications is at most $\sum_i 10 \cdot l(s_i') \leq 20 \cdot l(s)$.

With this, we obtain three curves $\tilde{\pi}_\mathrm{R}, \tilde{\pi}_\mathrm{G}, \tilde{\pi}_\mathrm{B}$ differing from $\pi_\mathrm{R}, \pi_\mathrm{G}, \pi_\mathrm{B}$ only inside $N_\delta(s)$ and having total length at most

$$\begin{aligned} l(\tilde{\pi}_\mathrm{R}) + l(\tilde{\pi}_\mathrm{G}) + l(\tilde{\pi}_\mathrm{B}) &\leq l(\overline{\pi}_\mathrm{R}) + l(\overline{\pi}_\mathrm{G}) + l(\overline{\pi}_\mathrm{B}) + 20 \cdot l(S) \\ &\overset{(4.3)}{\leq} l(\pi_\mathrm{R}) + l(\pi_\mathrm{G}) + l(\pi_\mathrm{B}) + 23 \cdot l(S) \end{aligned} \tag{4.4}$$

such that the crossing pattern fulfills the following:

i) no red group is next to a green group,

ii) every group contains at most two crossings,

iii) every red or green group which is not the first or last group along s contains precisely two crossings,

iv) the subpatterns $BRRBBRR$ and $BGGBBGG$ are not contained,

v) between any two crossings of a color c, the number of crossings of color $c' \neq c$ is even, in particular, the patterns $RRBRR$ and $GGBGG$ are not contained.

To sum up, the possible crossing patterns are as follows: First, i) and ii) imply that the sequence alternates between a blue group and a red or green group where every group has one or two crossings. Second, combining iii) with iv) and v), we obtain that there is no subsequence of blue group, red group, blue group, red group, blue group. And similarly, there is no subsequence of blue group, green group, blue group, green group, blue group.

Therefore, the crossing pattern of $\tilde{\pi}_\mathrm{R}, \tilde{\pi}_\mathrm{G}, \tilde{\pi}_\mathrm{B}$ with s has the following form: $\mathcal{G}_1 P \mathcal{G}_2$ where $\mathcal{G}_1, \mathcal{G}_2 \in \{\emptyset, R, G, RR, GG\}$, and P is a subsequence of the infinite sequence $B^*RRB^*GGB^*RRB^*GG\ldots$, where each B^* can be replaced independently by B or BB. The next step will be to modify the sequence P such that it has bounded length.

Patching trichromatic patterns In this step, we show that a pattern of the form $B^*RRB^*GGB^*RRB^*GG\ldots$ can be patched in a way that the resulting pattern contains either at most one green or at most one red group. For this, assume that we are given a pattern where we have at least two groups of each color. Then it contains the subpattern $GGB^*RRB^*GGB^*RR$, where the roles of R and G can

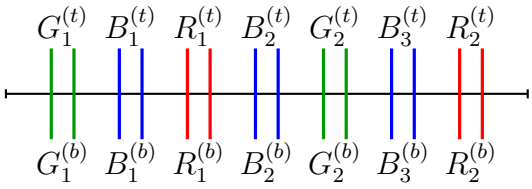

Figure 4.9: The crossing pattern $GGBBRRBBGGBBRR$ in a δ-neighborhood of s.

be exchanged. We number the groups in that pattern by $G_1, B_1, R_1, B_2, G_2, B_3, R_2$ (cf. Figure 4.9). Our goal is to reduce this pattern to GGB^*RR.

As a first step, note that, in this pattern, every B^* needs to be replaced by the same choice in $\{B, BB\}$: If B_1 and B_2 do not have the same number of crossings, the number of blue crossings between G_1 and G_2 is odd, which gives a contradiction. Similarly, B_2 and B_3 contain the same number of crossings. Therefore, we investigate the two patterns $GGBBRRBBGGBBRR$ and $GGBRRBGGBRR$.

Even though we make adjustments only in a δ-neighborhood of s, it is important to study how the crossings are connected via the entire tours because we need to ensure that our patching procedure does not disconnect a tour.

For this, we say that the top of a crossing $\boldsymbol{x}$ on s is *connected to the top* of another crossing $\boldsymbol{y}$ of the same color c if, when traveling along the curve π_c from $\boldsymbol{x}$ in the direction upwards from s, the first other crossing in $s \cap \pi_c$ encountered is $\boldsymbol{y}$ and we enter $\boldsymbol{y}$ from the top. This is similarly defined for *bottoms* of crossings and combinations of tops and bottoms of crossings. Moreover, we say that the top of a group $\mathcal{G}$ is connected to the top of another group $\mathcal{G}'$ if one of the tops of the crossings in $\mathcal{G}$ is connected to one of the tops of the crossings in $\mathcal{G}'$. We denote the tops and bottoms of the groups by $G_1^{(t)}, G_1^{(b)}, \ldots$ (cf. Figure 4.9).

We start by investigating the connections of the middle blue group B_2. First, note that there cannot be a connection from B_2 to any blue group on the opposite side, i.e., from $B_2^{(t)}$ to $B_i^{(b)}$ or from $B_2^{(b)}$ to $B_i^{(t)}$ for any $i \in \{1, 2, 3\}$: To see this, observe that, in Figure 4.10 (a), including any of the dotted lines would lead to the green, respectively red, groups being disconnected. Therefore, B_2 can only be connected to a blue group on the same side, i.e., $B_2^{(t)}$ can be connected to $B_i^{(t)}$ and $B_2^{(b)}$ can be connected to $B_j^{(b)}$ for some $i, j \in \{1, 2, 3\}$.

Next, observe that it is impossible to have the connections with $i = j = 1$ or $i = j = 3$: As illustrated in Figure 4.10 (b), this would lead to the red (in case $i = j = 1$), respectively green (in case $i = j = 3$), groups to be disconnected. Also, it is not possible that both, $B_2^{(t)}$ and $B_2^{(b)}$, are only connected to themselves, i.e., $i = j = 2$, because then B_2 would not be connected to any other blue group. Therefore, B_2 is connected to either B_1 or B_2 on the same side and assume w.l.o.g. that the connection between $B_2^{(t)}$ and $B_1^{(t)}$ exists. Next, we to show that then the connection between $B_2^{(b)}$ and $B_3^{(b)}$ exists as well. Observe in Figure 4.10 (c) that the existence of the connection between $B_2^{(t)}$ and $B_1^{(t)}$ implies the existence of a connection between $R_1^{(b)}$ and R_3, i.e., one of the red dashed connections must

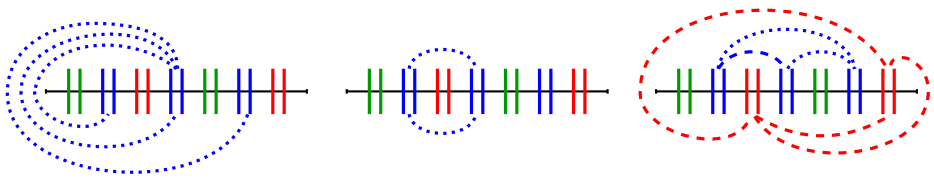

Figure 4.10: Impossible connections of the blue groups. Dashed lines represent existing connections and dotted lines represent impossible connections.

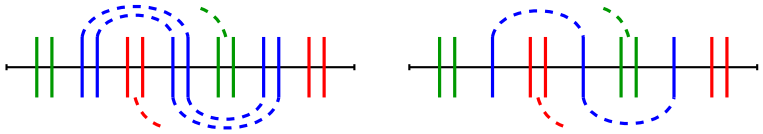

Figure 4.11: We show that, in the two considered crossing patterns, the connections illustrated by dashed lines must exist.

exist. With this, observe that any of the dotted blue connections (together with any of the red dashed connections) would lead to G_2 being disconnected from G_1. There must be a connection from B_3 to some other blue group and we can now see from Figure 4.10 (c) that the only remaining possibility is a connection between $B_2^{(b)}$ and $B_3^{(b)}$.

To summarize, we have seen that there is a connection between $B_2^{(t)}$ and $B_i^{(t)}$, and between $B_2^{(b)}$ and $B_j^{(b)}$ for some $\{i, j\} = \{1, 3\}$ and there are no other connections of B_2. In particular, given the case that each blue group consists of two crossings, the tops and bottoms of both crossings in B_2 are connected to the same other blue group. Assume from now on w.l.o.g. that $i = 1$ and $j = 3$. The two possible crossing patterns are illustrated in Figure 4.11 together with the connections of B_2. By the dashed red and green line, we indicate that $R_1^{(b)}$ must be connected to R_2 and $G_2^{(t)}$ must be connected to G_1.

From this, one can observe that cutting the tours open at points at distance at most δ from s and reconnecting as illustrated in Figure 4.12, the three curves are still connected and therefore, still form closed non-crossing curves. Note that the new intersection pattern is GGB^*RR as desired. One can also see in the figure that every line included for the reconnection has length at most $l(s)$. Since each line connects two points where the tours were cut open and there were 14, respectively 11, crossings, the total cost of the reconnection is at most $14 \cdot l(s)$. Moreover, the modified tours have at most 6 crossings so that moving them sufficiently close to the right endpoint of s gives an additional cost of at most $12 \cdot l(s)$. Overall, this shows that we can reduce the intersection pattern $GGB^*RRB^*GGB^*RR$

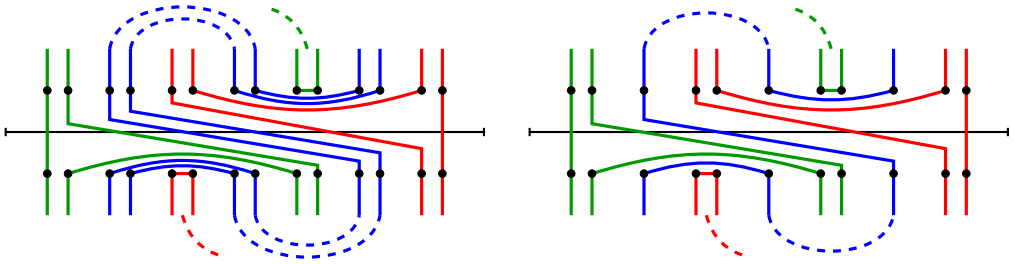

Figure 4.12: Patching scheme for three non-crossing tours in the two considered crossing patterns.

to GGB^*RR and move the crossings sufficiently close to the right endpoint of s at cost $(14+12)\cdot l(s) = 26\cdot l(s)$.

Given this, we proceed similarly as in the paragraph on bichromatic patterns to reduce the number of crossings with s: While the intersection pattern contains the subsequence $GGB^*RRB^*GGB^*RR$ (or with exchanged roles of G and R), choose the leftmost such pattern and apply the described patching scheme to the shortest subsegment s' of s containing the subpattern and move the new crossings sufficiently close to the right endpoint of s. We have seen that the cost of a single application is $26\cdot l(s')$ and, similarly as for bichromatic patterns, one can show that every point in s is contained in at most two of the chosen s'. Therefore, the total cost of the modifications is at most $52\cdot l(s)$. Let $\pi'_{\mathrm{R}}, \pi'_{\mathrm{G}}, \pi'_{\mathrm{B}}$ denote the resulting tours. Then we have

$$\begin{aligned} l(\pi'_{\mathrm{R}}) + l(\pi'_{\mathrm{G}}) + l(\pi'_{\mathrm{B}}) &\leq l(\tilde{\pi}_{\mathrm{R}}) + l(\tilde{\pi}_{\mathrm{G}}) + l(\tilde{\pi}_{\mathrm{B}}) + 52\cdot l(S) \\ &\overset{(4.4)}{\leq} l(\pi_{\mathrm{R}}) + l(\pi_{\mathrm{G}}) + l(\pi_{\mathrm{B}}) + 75\cdot l(S). \end{aligned} \tag{4.5}$$

To summarize, we have shown that the intersection pattern of the modified tours $\pi'_{\mathrm{R}}, \pi'_{\mathrm{G}}, \pi'_{\mathrm{B}}$ with S is of the form $\mathcal{G}_1\mathcal{P}\mathcal{G}_2$ where $\mathcal{G}_1, \mathcal{G}_2 \in \{\emptyset, R, G, RR, GG\}$ and $\mathcal{P}$ is a subsequence of $B^*RRB^*GGB^*RRB^*GG\ldots$ that does not contain two red and two green groups. A longest possible such subsequence is $BBRRBBGGBBRRBB$. Therefore, the lengths of $\mathcal{G}_1$ and $\mathcal{G}_2$ are at most two and the length of $\mathcal{P}$ is at most 14. This implies that the total number of crossings is at most 18.

Moreover, observe that we have only modified the tours inside $N_\delta(s)$ and that by equation (4.5) the total length of $\pi'_{\mathrm{R}}, \pi'_{\mathrm{G}}, \pi'_{\mathrm{B}}$ is as desired. □

4.3.4 Structure theorem for three non-crossing tours

Now, we have all the prerequisites in place to state and prove our structure theorem (cf. Theorem 4.5) for three non-crossing tours.

For this, given an instance of k-ETSP with terminals in $\{0,\ldots,L\}^2$ and a shift vector $\boldsymbol{a} \in \{0,\ldots,L-1\}^2$, we say that a solution $\Pi = (\pi_c)_{c\in C}$ is (r,m,δ)-*portal respecting* if, for every boundary b in $D(\boldsymbol{a})$, the intersection points $b \cap \bigcup_{c\in C} \pi_c$ are

contained in the the portals grid$(b, r\log(L), \delta)$ and every portal is intersected in at most m points in total (in particular, since grid lines consist of boundaries, grid lines are only crossed in portals).

Theorem 4.15 (Structure Theorem for 3-ETSP)**.** *Consider an instance of* 3-ETSP *with* $T \subseteq \{0, \dots, L\}^2$ *and let* $\varepsilon > 0$ *be given. Then there exists a shift vector* $\boldsymbol{a} \in \{0, \dots, L-1\}^2$*, such that, for every* $\delta > 0$ *small enough, there exists one of the following (or both)*

- *a* $(\lceil \frac{14}{\varepsilon} \rceil, 18, \delta)$*-portal-respecting solution of cost at most* $(1+\varepsilon) \cdot \textsc{Opt}$
- *a* $(\lceil \frac{14}{\varepsilon} \rceil, 18, \delta)$*-portal-respecting two-tour presolution* (π_1, π_2) *of weighted cost at most* $2l(\pi_1) + l(\pi_2) \leq \left(\frac{5}{3} + \frac{\varepsilon}{2}\right) \cdot \textsc{Opt}$*.*

Proof. Let $\eta > 0$ be a small real number that will be appropriately set later on in the proof. Let $\Pi = (\pi_{\mathrm{R}}, \pi_{\mathrm{G}}, \pi_{\mathrm{B}})$ be a solution of cost at most $(1+\eta)\cdot\textsc{Opt}$. Recall that we can assume w.l.o.g. that each tour in Π consists of straight line segments connecting points in $N_{\frac{1}{4}}(\{0, \dots, L\}^2)$. Choose a shift vector $\boldsymbol{a} \in \{0, \dots, L-1\}^2$ uniformly at random. Let $\delta > 0$ be a small enough real number that will be appropriately upper bounded later on in the proof. Set $r := \lceil (14)/\varepsilon \rceil$, and consider the dissection $D(\boldsymbol{a})$ with the portals grid$(b, r\log(L), \delta)$ placed on every boundary b. Throughout this proof, a *crossing* is a point in the intersection of the (partially modified) tours and the grid lines.

As the first step, we move all crossings to portals by applying the modifications of Lemma 4.11 to $\pi_{\mathrm{R}}, \pi_{\mathrm{G}}, \pi_{\mathrm{B}}$ where we set the value δ' to be less than 0.25 and let $\hat{\Pi} = (\hat{\pi}_{\mathrm{R}}, \hat{\pi}_{\mathrm{G}}, \hat{\pi}_{\mathrm{B}})$ be the resulting tours. The tours remain disjoint and, since they were only modified in a δ'-neighborhood around the grid lines and terminals have distance at least 0.25 from any grid line, they still visit all the terminals so that $\hat{\pi}_{\mathrm{R}}, \hat{\pi}_{\mathrm{G}}, \hat{\pi}_{\mathrm{B}}$ is still a solution to the given instance of 3-ETSP. By Lemma 4.11, we have

$$\begin{aligned}\mathbb{E}\Big[l(\hat{\Pi})\Big] &\leq \left(1 + \frac{7\sqrt{2}}{r}\right) \cdot (l(\pi_{\mathrm{R}}) + l(\pi_{\mathrm{G}}) + l(\pi_{\mathrm{B}})) \\ &\leq \left(1 + \eta + \frac{7\sqrt{2}\cdot(1+\eta)}{r}\right) \cdot \textsc{Opt}. \end{aligned} \tag{4.6}$$

Moreover, no crossing of $\hat{\pi}_{\mathrm{R}}, \hat{\pi}_{\mathrm{G}}, \hat{\pi}_{\mathrm{B}}$ with grid lines lies in the intersection of two grid lines. Note that this implies the following: Given a boundary b, consider the first portal s on b. Recall that it is placed such that its left endpoint $\boldsymbol{x}$ lies on the left endpoint of b and $\boldsymbol{x}$ lies in the intersection of b with a grid line g perpendicular to b. Since no crossing lies in the intersection of two grid lines, there is no crossing that equals $\boldsymbol{x}$. Therefore, if we are about to apply patching (i.e., Lemma 4.14) on s, we can choose a subsegment s' of s not containing $\boldsymbol{x}$ and apply patching on s' so that the the tours remain unchanged in some neighborhood of g. The last portal on b can be patched analogously. This means that we can patch portals in a way that grid lines of shallower levels are not affected.

W.l.o.g., assume $\hat{\pi}_{\mathrm{R}}, \hat{\pi}_{\mathrm{G}} \leq \hat{\pi}_{\mathrm{B}}$. We further modify the tours in $\hat{\Pi}$ by applying the following procedure: Consider every boundary b one by one in non-decreasing order of their levels and consider every portal s on b. If there is no red crossing next to a green crossing on s, we apply patching (i.e., Lemma 4.14) on s (as described in the previous paragraph). If there is a red crossing next to a green crossing, we connect the red and green tour along s and, for the remainder of the procedure, we identify the colors red and green with each other (i.e., we are left with only two tours). Note that this enables us to apply patching.

We argue that the resulting tours are $(r, 18, \delta)$-portal respecting. Similarly as in the proof of Lemma 4.11, note that patching can create additional crossings on a grid line perpendicular to the portal under consideration. However, we have argued above that only grid lines crossing s° are affected and, by Observation 4.10, these grid lines are of deeper levels so that the corresponding boundaries have not been considered yet. Moreover, the additionally created crossings on a boundary b' perpendicular to b° lie inside a portal because we only change the tours in a sufficiently small neighborhood (smaller than the portal length) of b and $b \cap b'$ is an endpoint of b' so it lies in a portal on b'. With this, we obtain that the obtained tours are $(r, 18, \delta)$-portal respecting for every choice of $\boldsymbol{a}$.

Next, we bound the cost of the resulting tours. It follows from Lemma 4.14 that the cost of patching in a single portal is $\mathcal{O}(\delta)$. Since there are $\mathcal{O}(L^2 r \log(L))$ portals, the total cost of patching is $\mathcal{O}(\delta L^2 r \log(L))$, so δ can be chosen small enough to make the total cost of patching at most $\textsc{Opt}/r$. To summarize, if there was never a red crossing next to a green crossing in a portal, we obtain a $(r, 18, \delta)$-portal respecting solution of cost at most

$$\mathbb{E}_{\boldsymbol{a}}\Big[l(\hat{\Pi})\Big] + \frac{\textsc{Opt}}{r} \overset{(4.6)}{\leq} \left(1 + \eta + \frac{7\sqrt{2} \cdot (1+\eta) + 1}{r}\right) \cdot \textsc{Opt} \leq (1+\varepsilon) \cdot \textsc{Opt},$$

where we have used in the last inequality that $7 \cdot \sqrt{2} \approx 9.899$, $r \geq 14/\varepsilon$ and η can be chosen small enough. In particular, at any time during the procedure in which the red and green tour are not connected, the total length of any subset of the three tours is increased at most by $\textsc{Opt}/r$.

Now, assume that, at some point in the procedure where a portal s is considered, there is a red crossing next to a green crossing. The cost of connecting the red and green tour is then at most 2δ, which is at most $\textsc{Opt}/r$ for δ small enough. Therefore, the total cost of patching and connecting the tours is at most $2\textsc{Opt}/r$. Therefore, the result of the procedure is a two-tour presolution (π_1, π_2) that is $(r, 18, \delta)$-portal respecting with

$$\begin{aligned} 2l(\pi_1) + l(\pi_2) &\leq 2(l(\hat{\pi}_{\mathrm{R}}) + l(\hat{\pi}_{\mathrm{G}})) + l(\hat{\pi}_{\mathrm{B}}) + 2 \cdot \frac{2\textsc{Opt}}{r} \\ &\overset{l(\hat{\pi}_{\mathrm{R}}), l(\hat{\pi}_{\mathrm{G}}) \leq l(\hat{\pi}_{\mathrm{B}})}{\leq} \frac{5}{3} \cdot (l(\hat{\pi}_{\mathrm{R}}) + l(\hat{\pi}_{\mathrm{G}}) + l(\hat{\pi}_{\mathrm{B}})) + \frac{4\textsc{Opt}}{r} \\ &\overset{(4.6)}{\leq} \frac{5}{3} \cdot \left(1 + \eta + \frac{7\sqrt{2} \cdot (1+\eta) + 4}{r}\right) \cdot \textsc{Opt} \leq \left(\frac{5}{3} + \frac{\varepsilon}{2}\right) \cdot \textsc{Opt}, \end{aligned}$$

where we have used in the last inequality that $7 \cdot \sqrt{2} \approx 9.899$, $r \geq 14/\varepsilon$ and η can be chosen small enough.

Last, recall that $\boldsymbol{a}$ was chosen uniformly at random from $\{0, \dots, L-1\}^2$. We have seen that we obtain a portal-respecting solution for every choice of $\boldsymbol{a}$ and the resulting cost is in expectation as desired. Using the probabilistic method, this implies that there is a vector $\boldsymbol{a} \in \{0, \dots, L-1\}^2$ such that the obtained tours have the cost stated in the Theorem. □

Recall that, given a two-tour presolution, one can "double" one of the tours to obtain an induced two-tour solution for 3-ETSP (cf. Observation 4.7). Applying this to a $(\lceil 14/\varepsilon \rceil, 18, \delta)$-portal respecting two-tour presolution obtained from Theorem 4.15, this gives a $(\lceil 14/\varepsilon \rceil, 36, \delta)$-portal respecting solution for 3-ETSP. Combining this with Theorem 4.15, we obtain the following.

Corollary 4.16. *For every instance of* 3-ETSP *with terminals in* $\{0, \dots, L\}^2$ *and every* $\varepsilon > 0$*, there is a shift vector* $\boldsymbol{a} \in \{0, \dots, L-1\}^2$ *such that, for every* $\delta > 0$ *small enough, there exists a* $(\lceil 14/\varepsilon \rceil, 36, \delta)$*-portal respecting solution of cost at most* $\left(\frac{5}{3} + \varepsilon\right) \cdot \textsc{Opt}$.

4.4 Computing portal-respecting solutions

In the previous section, we have seen that there is a $\left(\frac{5}{3} + \varepsilon\right)$-approximate portal-respecting solution. In this section, we give a polynomial-time algorithm that computes an "optimal" (in the sense of Theorem 4.6) portal-respecting solution.

Our algorithm is based on the same ideas as Arora's dynamic programming algorithm for Euclidean TSP [5] and the algorithm by Dross et al. for 2-ETSP [45]. The difference to our work is that we need to solve a more general problem: First, we allow for any fixed number of colors of terminals and search for non-crossing tours. Second, we have weighted colors, i.e., the tours of different colors contribute differently to the total cost.

More precisely, by k-ETSP$'$ we denote the following problem: A set C of k colors is given together with an integer L that is a power of two. The input consists of a set of terminals $T_c \subseteq \{0, \dots, L\}^2$ for each color $c \in C$, a color weight $w_c \geq 0$ for each $c \in C$, a shift vector $\boldsymbol{a} \in \{0, \dots, L-1\}^2$, $\delta > 0$ (sufficiently small) and integers $r, m \in \mathbb{N}$. We consider the dissection $D(\boldsymbol{a})$ and, as before, we place $r \log(L)$ portals on every boundary. A *solution* to k-ETSP$'$ is a k-tuple of tours $\Pi = (\pi_c)_{c \in C}$ such that every terminal is visited by the tour of the same color (i.e., $T_c \subseteq \pi_c$ for every $c \in C$), the tours are pairwise disjoint, and (r, m, δ)-portal respecting. The *cost of a solution* for k-ETSP$'$ is then $l(\Pi) := \sum_{c \in C} w_c \cdot l(\pi_c)$. Similarly as for k-ETSP (cf. Figure 4.1), a solution minimizing the cost does not necessarily exist. By $\textsc{Opt} := \inf\{l(\Pi)) : \Pi \text{ is a solution}\}$, we denote the value that we want to approximate.

The aim of this section is to prove Theorem 4.6. We begin by giving a refined formulation.

Theorem 4.17. *There is an algorithm that computes a parametric solution* $\Pi(\lambda)$ *to* k-ETSP$'$ *in time* $L^{\mathcal{O}(mr\log(k))}$ *such that* $\lim_{\lambda\to 0} l(\Pi(\lambda)) = OPT$.

The main idea is to use dynamic programming. More precisely, we consider the nodes of $D(\boldsymbol{a})$ (i.e., squares in $\mathbb{R}^2$) one by one from leaves to the root and compute all possible ways that a solution can cross the border of the square. For a non-leaf node, we will find these possibilities by combining the solutions for the four subsquares.

4.4.1 The multipath problem and the lookup-table

In this subsection, we investigate how the problem of finding a portal-respecting solution can be decomposed into subproblems. Throughout this subsection, fix an instance $(T_c)_{c\in C}, (w_c)_{c\in C}, \boldsymbol{a}, \delta, r, m$ of k-ETSP$'$.

Recall that portals are subsegments of boundaries of $D(\boldsymbol{a})$ of positive length δ. For every portal in $D(\boldsymbol{a})$, we place m distinct points in it, called *subportals*, and we will consider solutions that only intersect boundaries in subportals. We place the subportals in a way that no subportal lies at the intersection of the gridlines. Note that each subportal is crossed at most once because the tours are pairwise disjoint and a subportal is a point. Observe that every (r, m, δ)-portal-respecting solution can be transformed into a subportal-respecting solution at cost at most 2δ for every portal. Since δ will be chosen arbitrarily small, this cost is negligible. Therefore, the requirement that a solution only intersects portals in the subportals is not a restriction.

Let a node (i.e., a square) S of $D(\boldsymbol{a})$ be given. We consider all possible ways that the tours of a solution can leave and enter S through the subportals. For this, we color each subportal with one of the colors in C or leave it uncolored. This encodes which of the tours in a solution crosses through the subportal where uncolored means that none of them crosses. We denote a coloring on all the subportals associated with ∂S by a k-tuple $(P_c)_{c\in C}$ of pairwise disjoint subsets of the subportals. Given such a coloring, we also have to specify which subportals are crossed consecutively by a tour. For this, we consider matchings between the colored subportals (cf. Figure 4.13). The fact that tours of different colors are non-crossing and that we can restrict ourselves to constructing simple tours[2], significantly reduces the number of matchings that need to be considered.

For this, we define a *non-crossing matching* of a coloring $(P_c)_{c\in C}$ as a k-tuple denoted $(M_c)_{c\in C}$ such that the following holds: For every $c \in C$, M_c is a partition of P_c into sets of size two, and there are pairwise disjoint curves $(\pi_{\boldsymbol{pq}})$ for every $\{\boldsymbol{p}, \boldsymbol{q}\} \in \bigcup_{c\in C} M_c$ such that each $(\pi_{\boldsymbol{pq}})$ connects the subportals $\boldsymbol{p}$ and $\boldsymbol{q}$ (cf. Figure 4.13). Observe that the latter condition is fulfilled if and only if the the set of straight line segments $\{\overline{\boldsymbol{pq}} : \{\boldsymbol{pq}\} \in \bigcup_{c\in C} M_c\}$ is pairwise disjoint. We will see later that the number of non-crossing matchings is bounded by $L^{\mathcal{O}(mr)}$ (cf. Lemma 4.18).

[2]Assume that one of the tours π_c is not simple, i.e., there is a point $\boldsymbol{x}$ in which π_c crosses itself. Then, π_c can be modified in a sufficiently small neighborhood of $\boldsymbol{x}$ such that π_c is simple in that neighborhood, still does not intersect any of the other tours, and its length is decreased.

If S contains all terminals of a color c or none of the terminals of color c, the tour π_c of a solution does not necessarily intersect the border of S (but note that it might intersect it to bypass a terminal of another color). For this reason, we also allow $P_c = \emptyset$. In this case, we need the constructed curves to form a single cycle. However, if S contains at least one but not all terminals of color c, a solution must intersect the border of S so we only allow for $P_c \neq \emptyset$. We also need to ensure that the tours π_c that we construct in the end are connected (i.e., do not consist of several disconnected cycles). For this reason, in the case $P_c \neq \emptyset$, we will require $\pi_c \cap S$ to consist of curves starting and ending at the border of S.

Now, we have all the prerequisites in place to define the *multipath problem* (cf. Figure 4.13): Fix an instance of k-ETSP$'$ and a node S of $D(\boldsymbol{a})$. The input consists of a coloring $(P_c)_{c\in C}$ of the subportals of ∂S and a non-crossing matching $(M_c)_{c\in C}$ of $(P_c)_{c\in C}$, where $P_c = \emptyset$ is only a valid input if $S \cap T_c \in \{T_c, \emptyset\}$. A solution to the multipath problem for S is a k-tuple $(\Pi_c)_{c\in C}$ of sets of simple curves in S such that:

a) every terminal of color c in S is visited by a curve of color c, i.e., $T_c \cap S \subseteq \bigcup_{\pi\in\Pi_c} \pi$,

b) the curves are disjoint,

c) for every $c \in C$ and every $\{\boldsymbol{p}, \boldsymbol{q}\} \in M_c$, there is a curve $\pi \in \Pi_c$ connecting the two subportals and not crossing the border of S otherwise, i.e., $\{\boldsymbol{p}, \boldsymbol{q}\} = \pi \cap \partial S$,

d) the border of S is only crossed at colored subportals, i.e., $\partial S \cap \left(\bigcup_{c\in C,\pi\in\Pi_c} \pi\right) \subseteq \bigcup_{c\in C} P_c$, and boundaries inside S are only crossed at subportals,

e) for every $c \in C$, if $P_c \neq \emptyset$, each curve of color c connects two subportals of color c on ∂S,

f) for every $c \in C$, if $P_c = \emptyset$, then $\bigcup_{\pi\in\Pi_c} \pi$ is a single closed curve (or $\emptyset$).

Observe that conditions c) and d) together imply that every colored subportal is crossed exactly once by a curve of the same color and there are no other crossings on the border of S. The *cost* of a solution to the multipath problem is $\sum_{c\in C} w_c \cdot l(\Pi_c)$, where $l(\Pi_c) := \sum_{\pi\in\Pi_c} l(\pi)$ denotes the total length of the set of curves. Note that, similarly as for k-ETSP and k-ETSP$'$, a solution minimizing the cost does not necessarily exist.

We define the *lookup-table* LT_S of S as follows:

$$\mathrm{LT}_S[(P_c)_{c\in C}, (M_c)_{c\in C}] := \inf\Big\{\sum_{c\in C} w_c \cdot l(\Pi_c) : (\Pi_c)_{c\in C} \text{ is a solution to the multipath problem with inputs } (P_c)_{c\in C} \text{ and } (M_c)_{c\in C}\Big\},$$

where $(P_c)_{c\in C}$ is a coloring of the subportals of S and $(M_c)_{c\in C}$ is a non-crossing matching for $(P_c)_{c\in C}$, where $P_c = \emptyset$ is only allowed if $T_c \cap S \in \{\emptyset, S\}$.

As we will calculate every entry of the lookup-tables in a dynamic programming fashion, it is important to bound the number of entries of LT_S. First, we bound the number of non-crossing matchings. The key idea for this is to upper bound them by the Catalan numbers.

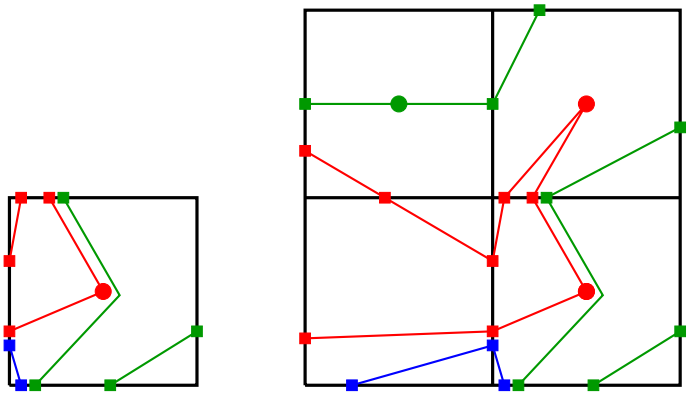

Figure 4.13: Illustration of the multipath problem: The cycles denote terminals and the rectangles denote subportals. On the left, a solution of a base case is given. On the right, we are given a combination of compatible entries and the resulting solution.

Lemma 4.18. *For a node S of $D(\boldsymbol{a})$ and a coloring $(P_c)_{c\in C}$ of the subportals in ∂S, the number of non-crossing matchings is at most $L^{\mathcal{O}(mr)}$ and we can list all the non-crossing matchings in time $L^{\mathcal{O}(mr)}$.*

Proof. Let a coloring $(P_c)_{c\in C}$ of the subportals of S be given. We count the number of non-crossing matchings of $\bigcup_{c\in C} P_c$, i.e., we ignore the colors and allow for all colored subportals to be matched as long as the straight line segments $\{\overline{p_1p_2} : \{\boldsymbol{p}_1, \boldsymbol{p}_2\}$ is in the matching$\}$ are non-crossing. It is immediate that this is an upper bound on the number of non-crossing matchings of $(P_c)_{c\in C}$.

Let $i := |\bigcup_{c\in C} P_c|$ denote the number of colored subportals and let $f(i)$ denote the number of non-crossing matchings between them. Assume a non-crossing matching contains the pair of subportals $\{\boldsymbol{p}, \boldsymbol{p}'\}$. Note that $\overline{\boldsymbol{pp}'}$ separates the square into two faces. Therefore, the matching cannot contain a pair $\{\boldsymbol{q}, \boldsymbol{q}'\}$ where $\boldsymbol{q}$ and $\boldsymbol{q}'$ lie in different faces. Hence, if there are j subportals different from $\boldsymbol{p}, \boldsymbol{p}'$ in one of the faces, there are $i-j-2$ subportals different from $\boldsymbol{p}, \boldsymbol{p}'$ in the other face, and the number of non-crossing matchings that contain the pair $\{\boldsymbol{p}, \boldsymbol{p}'\}$ is $f(i)\cdot f(i-j-2) \leq f(i)\cdot f(i-j)$. Summing up over all choices for $\boldsymbol{p}'$, we obtain

$$f(i) \leq \sum_{j=0}^{i-1} f(j) \cdot f(i-j). \tag{4.7}$$

Note that this sum is precisely the recursive definition of the well-known *Catalan numbers* (see, e.g., [106]). Additionally using that $f(0) = 1$ and $f(1) = 0$, we obtain that $f(i)$ is upper bounded by the i-th Catalan number C_i. Since $C_i = \mathcal{O}(4^i)$ (cf. [106, Theorem 3.1]), we have

$$f(i) \leq C_i = \mathcal{O}(4^i).$$

As i is the number of colored portals, we have $i = \mathcal{O}(mr\log(L))$. Together, this gives that the number of non-crossing matchings is upper bounded by

$$\mathcal{O}(4^{\mathcal{O}(mr\log(L))}) = 2^{\mathcal{O}(mr\log(L))} = L^{\mathcal{O}(mr)}.$$

Note that the above recursion also gives an efficient recursive procedure for listing all the non-crossing matchings. □

Next, note that the number of subportals on the border of S is $\mathcal{O}(mr\log(L))$ so there are $(k+1)^{\mathcal{O}(mr\log(L))} = L^{\mathcal{O}(mr\log(k))}$ possible colorings of the subportals of S, and we can easily list all colorings in $L^{\mathcal{O}(mr\log(k))}$ time as well. Combining this with Lemma 4.18, we obtain the following result.

Observation 4.19. *For every node S of $D(\boldsymbol{a})$, the number of entries of the lookup-table LT_S is bounded by $L^{\mathcal{O}(mr\log(k))}$.*

4.4.2 A dynamic programming algorithm

Next, we show that the entries of the lookup-table can be computed in polynomial time. We begin with computing the tables of leaves.

Lemma 4.20. *For every leaf S of $D(\boldsymbol{a})$, the lookup-table LT_S can be computed in time $L^{\mathcal{O}(mr\log(k))}$.*

Proof. Consider a leaf of $D(\boldsymbol{a})$, i.e., a square S of side length 1, and an instance of the multipath problem, i.e., a coloring of the subportals $(P_c)_{c\in C}$ and a non-crossing matching $(M_c)_{c\in C}$. Recall that terminals are distinct points in $\{0,\dots,L\}^2$ so that S contains at most one terminal and, if S contains a terminal, it lies in the center of S.

If it contains no terminal, observe that an optimal solution is simply given by $\Pi_c = \{\overline{\boldsymbol{p}_1\boldsymbol{p}_2} : \{\boldsymbol{p}_1,\boldsymbol{p}_2\} \in M_c\}$ $(c \in C)$. Therefore, assume from now on that $D(\boldsymbol{a})$ does contain a terminal $\boldsymbol{t}$, say of color $c^* \in C$. If there is no subportal colored c^* (note that this is only possible if $\boldsymbol{t}$ is the only portal of color c), consider the solution given by $\Pi_c = \{\overline{\boldsymbol{p}_1\boldsymbol{p}_2} : \{\boldsymbol{p}_1,\boldsymbol{p}_2\} \in M_c\}$ $(c \neq c^*)$ and additionally letting Π_{c^*} consist of a single small cycle of length at most λ that visits $\boldsymbol{t}$ and does not intersect any of the tours in $\Pi_c, c \neq c^*$. In case one of the straight line segments in Π_c intersects $\boldsymbol{t}$, it can be bended by a sufficiently small amount at cost at most λ (the parameter that our solution depends on) so that it does not intersect $\boldsymbol{t}$ or any other of the straight line segments. For $\lambda \to 0$, the cost of this solution, i.e., $\sum_{\{\boldsymbol{p}_1,\boldsymbol{p}_2\}\in M_c}\|\boldsymbol{p}_1-\boldsymbol{p}_2\|$, gives the desired entry of the lookup-table. Therefore, assume from now on additionally that there are subportals colored c^*.

For one of the matched pairs in M_{c^*}, the curve connecting the two subportals has to visit the terminal $\boldsymbol{t}$. Say this pair is given by $\{\boldsymbol{p}_1,\boldsymbol{p}_2\}$. Then we obtain a solution as follows: First, include the curve $\overline{\boldsymbol{p}_1\boldsymbol{t}} \cup \overline{\boldsymbol{t}\boldsymbol{p}_2}$ in Π_{c^*}. Then, add the straight line segments $\overline{\boldsymbol{q}_1\boldsymbol{q}_2}$ to Π_c for all $\{\boldsymbol{q}_1,\boldsymbol{q}_2\} \in M_c$ where $\overline{\boldsymbol{q}_1\boldsymbol{q}_2}$ does not intersect $\overline{\boldsymbol{p}_1\boldsymbol{t}} \cup \overline{\boldsymbol{t}\boldsymbol{p}_2}$ $(c \in C$ and $\{\boldsymbol{q}_1,\boldsymbol{q}_2\} \neq \{\boldsymbol{p}_1,\boldsymbol{p}_2\})$. For each remaining pair $\{\boldsymbol{q}_1,\boldsymbol{q}_2\} \in M_c$ that intersects $\overline{\boldsymbol{p}_1\boldsymbol{t}} \cup \overline{\boldsymbol{t}\boldsymbol{p}_2}$, there is a suitable point $\boldsymbol{t}'$ at distance at most λ from $\boldsymbol{t}$ such

that $\overline{\boldsymbol{q}_1\boldsymbol{t}'} \cup \overline{\boldsymbol{t}'\boldsymbol{q}_2}$ does not cross any other curves. This is illustrated by one of the green pairs on the left side in Figure 4.13. We add $\pi := \overline{\boldsymbol{q}_1\boldsymbol{t}'} \cup \overline{\boldsymbol{t}'\boldsymbol{q}_2}$ to Π_c. Note that $l(\pi) = \|\boldsymbol{q}_1 - \boldsymbol{t}'\| + \|\boldsymbol{q}_2 - \boldsymbol{t}'\| = \|\boldsymbol{q}_1 - \boldsymbol{t}\| + \|\boldsymbol{q}_2 - \boldsymbol{t}\|$ for $\lambda \to 0$. Therefore, given a choice of the pair $\{\boldsymbol{p}_1, \boldsymbol{p}_2\}$, we can efficiently compute the optimal cost of a solution. To find the entry of the lookup-table, we compute the cost for every choice of the pair $\{\boldsymbol{p}_1, \boldsymbol{p}_2\} \in \bigcup_{c \in C} M_c$ and check which choice minimizes the cost.

To summarize, there are $\mathcal{O}(mr \log(L))$ subportals so that the number of possible choices of the pair $\{\boldsymbol{p}_1, \boldsymbol{p}_2\}$ is at most $\mathcal{O}(m^2 r^2 \log^2 L)$. Once a pair is chosen, computing the cost of a solution (as $\lambda \to 0$) can be done in time $\mathcal{O}(mr \log(L))$ because the number of curves is bounded by the number of subportals. Therefore, a single entry of the lookup-table of a leaf can be computed in time $\mathcal{O}(m^2 r^2 \log^2 L) \leq L^{\mathcal{O}(mr)}$. Combining this with the fact that the size of every lookup-table is at most $L^{\mathcal{O}(mr \log(k))}$ (cf. Observation 4.19), we obtain that LT_S can be computed in time $L^{\mathcal{O}(mr \log(k))}$. □

Next, we investigate how the lookup-table of a non-leaf node of $D(\boldsymbol{a})$ can be computed.

Lemma 4.21. *For every non-leaf node S of $D(\boldsymbol{a})$, its lookup-table LT_S can be computed in time $L^{\mathcal{O}(mr \log(k))}$ if the lookup-tables of its four children are given.*

Proof. Let a square S of side length > 1, a coloring of the subportals $(P_c)_{c \in C}$ and a non-crossing matching $(M_c)_{c \in C}$ be given. Let $S^{(1)}, S^{(2)}, S^{(3)}, S^{(4)}$ denote its four children in $D(\boldsymbol{a})$ and assume that the lookup-tables of the children are already computed.

Note that each child $S^{(i)}$ ($i \in \{1, \ldots, 4\}$) shares two border edges with siblings and the other two of its border edges are contained in border edges of S (cf. right side of Figure 4.13). In particular, each $S^{(i)}$ shares subportals with siblings and the parent. More formally, let P denote the set of subportals on the border of S and $P^{(i)}$ denote the set of subportals on the border of $S^{(i)}$ ($i \in \{1, 2, 3, 4\}$). Then we have $P^{(i)} \subseteq \bigcup_{j \neq i} P^{(j)} \cup P$. In order to be able to combine the curves of solutions of the four children, we also need the following notion: a *matched sequence of subportals* is a sequence of subportals $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_N$ such that $\{\boldsymbol{p}_j, \boldsymbol{p}_{j+1}\} \in \bigcup_{c \in C, i \in \{1,\ldots,4\}} M_c^{(i)}$ for every $j \in \{1, \ldots, N-1\}$. Note that such a sequence exists if and only if combining the four subsolutions gives a curve that connects $\boldsymbol{p}_1$ with $\boldsymbol{p}_N$. Also note that, in a matched sequence, $\boldsymbol{p}_2, \ldots, \boldsymbol{p}_{N-1}$ cannot lie in P.

Given a combination of an entry of each lookup-table of the four children $(P_c^{(i)})_{c \in C}$, $(M_c^{(i)})_{c \in C}$, ($i \in \{1, 2, 3, 4\}$), we say that it is *compatible* with $(P_c)_{c \in C}, (M_c)_{c \in C}$ if

a) the colorings of the shared subportals coincide, i.e., for $i, j \in \{1, 2, 3, 4\}, c \in C$ and $p \in P^{(i)} \cap P^{(j)}$, we have $\boldsymbol{p} \in P_c^{(i)}$ if and only if $\boldsymbol{p} \in P_c^{(j)}$ and, similarly, for $\boldsymbol{p} \in P^{(i)} \cap P$, we have $\boldsymbol{p} \in P_c^{(i)}$ if and only if $\boldsymbol{p} \in P_c$,

b) For every $c \in C$ and every $\{\boldsymbol{p}, \boldsymbol{q}\} \in M_c$, there is a matched sequence of portals $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_N$ with $\boldsymbol{p}_1 = \boldsymbol{p}$ and $\boldsymbol{p}_N = \boldsymbol{q}$,

c) if $P_c \neq \emptyset$, for every maximal matched sequence of portals $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_N$, we have $\boldsymbol{p}_1, \boldsymbol{p}_N \in P$,

d) if $P_c = \emptyset$, then $\bigcup_{i\in\{1,2,3,4\},\{\boldsymbol{p},\boldsymbol{q}\}\in M_c^{(i)}} \overline{\boldsymbol{pq}}$ is a single cycle.

Note that condition c) implies that the combination of subsolutions is acyclic and only consists of curves touching ∂S. An example of a compatible combination and the resulting solution is illustrated on the right hand side of Figure 4.13.

We obtain that the desired entry of LT_S is given by the following equation:

$$\mathrm{LT}_S[(P_c)_{c\in C},(M_c)_{c\in C}] = \min\Bigg\{\sum_{i=1}^{4}\mathrm{LT}_{S^{(i)}}\big[(P_c^{(i)})_{c\in C},(M_c^{(i)})_{c\in C}\big] : (P_c^{(i)})_{c\in C},(M_c^{(i)})_{c\in C} \\ (i\in\{1,2,3,4\}) \text{ is compatible with } (P_c)_{c\in C},(M_c)_{c\in C}\Bigg\}.$$

We investigate the complexity of computing this minimum: The number of possibilities for combining entries of the lookup-table of the four children is at most $\big(L^{\mathcal{O}(mr\log(k))}\big)^4 = L^{\mathcal{O}(mr\log(k))}$. Since the number of subportals on the boundary of S and on the boundaries of its four children is at most $\mathcal{O}(mr\log(L))$, checking whether a combination is compatible, takes time at most $\mathcal{O}(m^2r^2\log^2(L))$: For each portal, we need to check whether its color coincides with the corresponding portal of its parent, respectively its sibling, which takes constant time. Also, for each portal, we have to compute the longest matched sequence in which it is contained, and the length of this sequence is upper bounded by the total number of subportals. We obtain that LT_S can be computed in time

$$L^{\mathcal{O}(mr\log(k))}\cdot\mathcal{O}(m^2r^2\log^2(L)) = L^{\mathcal{O}(mr\log(k))}.$$

□

Now, we have all the prerequisites in place to prove Theorem 4.17, i.e., we show that Opt can be computed in time $L^{\mathcal{O}(mr\log(k))}$.

Proof of Theorem 4.17. Observe that the value Opt for the given instance of k-ETSP′ is given by the solution to the multipath problem of $C(\boldsymbol{a})$ with $P_c = \emptyset$ for all $c \in C$. Recall that $D(\boldsymbol{a})$ is a full 4-ary tree with L^2 leafs. Therefore, $D(\boldsymbol{a})$ contains in total at most $\mathcal{O}(L^2)$ nodes. As we have seen in Lemmas 4.20 and 4.21, the lookup-tables of the nodes of $D(\boldsymbol{a})$ can be computed using dynamic programming and computing a single lookup-table takes time at most $L^{\mathcal{O}(mr\log(k))}$. Together, this gives that our algorithm has running time $\mathcal{O}\big(L^2\cdot L^{\mathcal{O}(mr\log(k))}\big) = L^{\mathcal{O}(mr\log(k))}$. As usual, a parametrized solution $\Pi(\lambda)$ with $\lim_{\lambda\to 0} l(\Pi(\lambda)) = \mathrm{Opt}$ can be found by reverse-engineering the computation of the lookup-tables. □

4.5 Perturbation

In the previous sections, we have focused on solving 3-ETSP when the terminals have integer coordinates. In this section, we show how an input for general 3-ETSP can be preprocessed to obtain an instance of the desired form, and how a solution

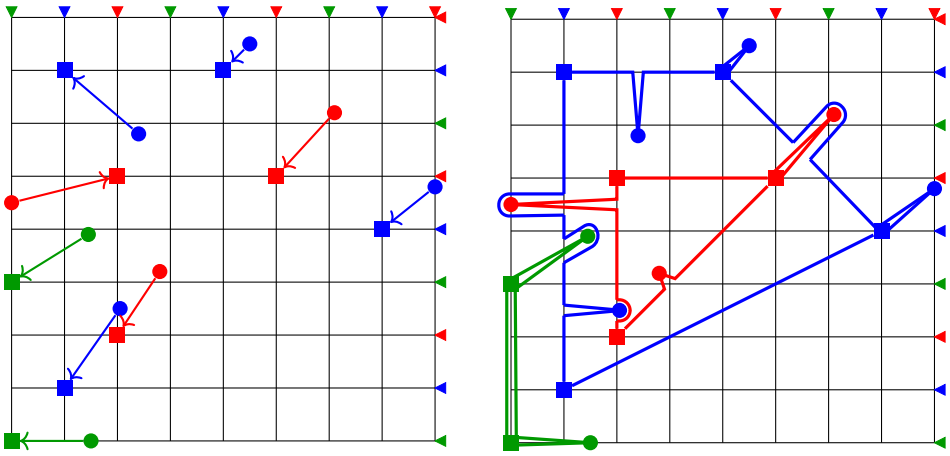

Figure 4.14: Illustration of the algorithms PERTURBATION (left side) and BACK-PERTURBATION (right side): The terminals in instance $\mathcal{I}$ are illustrated as circles and the terminals of the constructed instance $\mathcal{I}'$ are illustrated as rectangles. The triangles at the grid boundary indicate on which gridlines the terminals of each color can be snapped on.

to the preprocessed instance can be transformed back into a solution for the original input. For this, we use similar ideas as in [5].

Note that, if the terminal sets of different colors are, in some sense, sufficiently far apart, we can find solutions for each color independently, resulting in disjoint tours. More precisely, we call an instance of the 3-ETSP *ε-reducible* if there exists a partition of the three colors $\{c, c'\} \cup \{c''\} = \{\mathrm{R}, \mathrm{G}, \mathrm{B}\}$ such that applying the known PTAS for 2-ETSP [45] on T_c and $T_{c'}$, and independently applying Arora's PTAS for 1-ETSP [5] on $T_{c''}$, yields disjoint tours. In this case, these tours together form a $(1+\varepsilon)$-approximation for 3-ETSP. Therefore, in what follows, we restrict our attention to non-reducible instances.

Theorem 4.22. *Let $\varepsilon > 0$ and $\mathcal{I}$ be a non-ε-reducible instance of* 3-ETSP.

a) There is an algorithm PERTURBATION *that has running time $\mathcal{O}(n)$ and returns an instance $\mathcal{I}'$ of* 3-ETSP *with terminals in $\{0, \dots, L\}^2$ where $L = \mathcal{O}(n/\varepsilon)$ is a power of two.*

b) There is an algorithm BACK-PERTURBATION *that has running time $\mathcal{O}(n^2)$ and, given a $(1+\varepsilon')$-approximate solution to the instance $\mathcal{I}'$, returns a $(1+\varepsilon'+\mathcal{O}(\varepsilon))$-approximate solution to the instance $\mathcal{I}$.*

Proof. We begin with part a). For this, consider the algorithm PERTURBATION formally defined in Algorithm 4, which is illustrated on the left of Figure 4.14. Intuitively speaking, we first choose a smallest-possible square S in the plane containing all terminals. For this, we only consider aligned squares, that is, with borders parallel to the x-axis and y-axis. Next, we put a suitable number of $L + 1 = \mathcal{O}(n/\varepsilon)$ equispaced horizontal and vertical lines in the square, forming a lattice. Then, we

Algorithm 4: PERTURBATION

input: $\varepsilon > 0$, $T_{\mathrm{R}}, T_{\mathrm{G}}, T_{\mathrm{B}} \subseteq \mathbb{R}^2$ with $\bigcup_{c\in\{\mathrm{R,G,B}\}} T_c =: T$ and $|T| =: n$

1 $(S, f) \leftarrow \operatorname{argmin}\{f' : S' \text{ is an aligned square of side length } f' \text{ with } T \subseteq S'\}$

2 $L \leftarrow \min\{2^i : 2^i \geq \lceil n/\varepsilon \rceil, i \in \mathbb{N}\}$

3 let $v_0, \dots, v_L$ and $h_0, \dots, h_L$ denote $L+1$ vertical and $L+1$ horizontal equispaced straight line segments of length f in S with $h_0, v_0, h_L, v_L \subseteq \partial S$

4 $T'_{\mathrm{R}}, T'_{\mathrm{G}}, T'_{\mathrm{B}} \leftarrow \emptyset$

5 **forall** $c \in \{\mathrm{R}, \mathrm{G}, \mathrm{B}\}$ *and* $\boldsymbol{t} \in T_c$ **do**

6 $\quad z \leftarrow \begin{cases} 0, & \text{if } c = R \\ 1, & \text{if } c = G \\ 2, & \text{if } c = B \end{cases}$

7 $\quad \boldsymbol{t'} \leftarrow \operatorname{argmin}\{\|\boldsymbol{t''} - \boldsymbol{t}\| : \boldsymbol{t''} = v_i \cap h_j \text{ with } i \bmod 3 = j \bmod 3 = z\}$

8 $\quad$ add $\boldsymbol{t'}$ to T'_c

9 identify the points $\{v_i \cap h_j : i, j \in \{0, \dots, L\}\}$ with $\{0, \dots, L\}^2$

10 **return** $L, T'_{\mathrm{R}}, T'_{\mathrm{G}}, T'_{\mathrm{B}}$

snap each terminal to a closest possible lattice point, allowing each color to use only every third line of the lattice to ensure that no two terminals of different colors are snapped to the same point. Note that terminals of the same color might be snapped to the same point, however, this does not cause any issues. Last, we identify the lattice with $\{0, \dots, L\}^2$.

We begin by investigating the running time of PERTURBATION. For this, observe that line 1 can be executed in time $\mathcal{O}(n)$ as we only have to find the smallest and largest x-coordinates and y-coordinates of the terminals. The for-loop in line 5 is executed n times, and all other steps can be executed in time $\mathcal{O}(1)$. Therefore, its total running time is $\mathcal{O}(n)$.

Next, note that the sets $(T'_c)_{c\in\{\mathrm{R,G,B}\}}$ are disjoint because, due to lines 6 and 7, terminals of different colors cannot be placed on the same line h_i or v_j. Therefore, the output is a valid instance of k-ETSP. It is immediate from the construction that its terminals lie in $\{0, \dots, L\}^2$ and that $L = \mathcal{O}(n/\varepsilon)$. This completes the proof of part a).

Before moving on to BACK-PERTURBATION, we note some more useful properties of the algorithm PERTURBATION. First, observe that the assumption that the instance is non-ε-reducible implies that, for every terminal, there exists a terminal of another color at distance at most $(1+\varepsilon)\mathrm{OPT}$. This implies that $f \leq \mathcal{O}(\mathrm{OPT})$. Therefore, the distance between two consecutive horizontal lines h_i and h_{i+1} (or consecutive vertical lines v_i and v_{i+1}) is at most $f/L \leq \mathcal{O}(\mathrm{OPT}/L)$. It follows that, for every terminal $\boldsymbol{t}$ in T_c, the distance to the closest terminal in T'_c is at most

$$\sqrt{2} \cdot 3 \cdot \mathcal{O}(\mathrm{OPT}/L) \leq \mathcal{O}(\mathrm{OPT}/L), \tag{4.8}$$

due to line 7 (cf. left side of Figure 4.14).

Algorithm 5: BACK-PERTURBATION

input: $T_{\mathrm{R}}, T_{\mathrm{G}}, T_{\mathrm{B}} \subseteq [0, L]^2$, $(1+\varepsilon')$-approximate solution $\pi'_{\mathrm{R}}, \pi'_{\mathrm{G}}, \pi'_{\mathrm{B}}$ to an instance $T'_{\mathrm{R}}, T'_{\mathrm{G}}, T'_{\mathrm{B}} \subseteq \{0, \dots, L\}^2$

1 choose $\delta > 0$ small enough
2 **forall** $c \in \{\mathrm{R}, \mathrm{G}, \mathrm{B}\}$ *and* $\boldsymbol{t} \in T_c$ **do**
3 $\quad s \leftarrow \operatorname{argmin}\{l(s') : s'$ is a straight line segment between $\boldsymbol{t}$ and $\pi'_c\}$
4 $\quad$ **if** *s contains a terminal other than $\boldsymbol{t}$ (in T_d or T'_d for any color d)* **then**
5 $\quad\quad$ replace s by a segment of length $\leq 2l(s)$ connecting $\boldsymbol{t}$ and π_c not containing other terminals
6 $\quad$ apply patching (Lemma 4.14) to $\pi_{c'}, \pi_{c''}$ along s where $c', c'' \neq c$
7 $\quad$ redirect the remaining crossings around s
8 $\quad$ choose $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \pi'_c$ close enough at distance at most δ to $s \cap \pi_c$ such that $\pi'_c[\boldsymbol{x}_1, \boldsymbol{x}_2]$ does not contain terminals
9 $\quad$ replace π'_c by $(\pi'_c \setminus \pi'_c[\boldsymbol{x}_1, \boldsymbol{x}_2]) \cup \overline{\boldsymbol{x_1 t}} \cup \overline{\boldsymbol{x_2 t}}$
10 **return** $\pi'_{\mathrm{R}}, \pi'_{\mathrm{G}}, \pi'_{\mathrm{B}}$

Now, we turn to proving part b). For this, investigate the algorithm BACK-PERTURBATION given in Algorithm 5 and illustrated on the right side of Figure 4.14. In short, it connects the terminals of $\mathcal{I}$ to the given solutions for $\mathcal{I}'$ by shortest possible straight line segments, chosen such that these segments do not interfere with any other terminal. These segments may create additional crossings of the tours so that the other tours need to be redirected around the segments (cf. Figure 4.14 right side). To ensure we only have to redirect a finite number of crossings, we first apply patching along the segments for the two tours of the other colors.

First, note that every single line of the algorithm can be executed in time $\mathcal{O}(n)$. Since the loop beginning in line 2 is executed n times, this gives a running time of $\mathcal{O}(n^2)$. Next, note that, by construction, the resulting tours are indeed a solution to the instance of k-ETSP with terminal sets $T_{\mathrm{R}}, T_{\mathrm{G}}, T_{\mathrm{B}}$.

It remains to estimate the cost of the resulting tours. For this, we assume that $L, T'_{\mathrm{R}}, T'_{\mathrm{G}}, T'_{\mathrm{B}}$ are the output of PERTURBATION on $T_{\mathrm{R}}, T_{\mathrm{G}}, T_{\mathrm{B}}$. By OPT, we refer to the infimum cost of solutions to $\mathcal{I}$ and, by OPT_p, we refer to the infimum cost of solutions to the perturbed instance $\mathcal{I}'$. Note that the length of a segment s chosen in the algorithm is at most $\mathcal{O}(\mathrm{OPT}/L)$ because we have already seen that, for every terminal t in T_c, the distance to the closest terminal in T'_c is at most $\mathcal{O}(\mathrm{OPT}/L)$ (see (4.8)). Therefore, in each executiong of line 6, the total length of the tours is increased at most by $\mathcal{O}(\mathrm{OPT}/L)$ (cf. Lemma 4.14). Then the remaining number of crossings is at most 18 so that the additional cost in line 7 is $\mathcal{O}(\mathrm{OPT}/L)$ as well. Last, in line 9, since δ can be chosen small enough, the length of the considered tour is also increased by at most $\mathcal{O}(\mathrm{OPT}/L)$. Since the loop in line 2 is executed at most n times, the total length of given solution to $\mathcal{I}'$ is increased by at most

$$\mathcal{O}(n \cdot \mathrm{OPT}/L) = \mathcal{O}(\varepsilon)\mathrm{OPT},$$

where we have used that $L = \mathcal{O}(n/\varepsilon)$. It remains to relate OPT to OPT_p. Note

Algorithm 6: $\left(\frac{5}{3}+\varepsilon\right)$-approximation for 3-ETSP

parameters: large enough constant M
input : $\varepsilon > 0$, disjoint terminal sets $T_{\mathrm{R}}, T_{\mathrm{G}}, T_{\mathrm{B}} \subseteq \mathbb{R}^2$
$(L, T'_{\mathrm{R}}, T'_{\mathrm{G}}, T'_{\mathrm{B}}) \leftarrow$ PERTURBATION$(\varepsilon/M, T_{\mathrm{R}}, T_{\mathrm{G}}, T_{\mathrm{B}})$
choose $\delta, \mu > 0$ small enough
Sol $\leftarrow \emptyset$
forall $\boldsymbol{a} \in \{0, \dots, L-1\}^2$ **do**
  compute a $(1+\mu)$-approximate solution $\Pi = (\pi_{\mathrm{R}}, \pi_{\mathrm{G}}, \pi_{\mathrm{B}})$ for 3-ETSP$'$ with terminals $T'_{\mathrm{R}}, T'_{\mathrm{G}}, T'_{\mathrm{B}}$ that is $(\lceil 14M/\varepsilon \rceil, 36, \delta)$-portal-respecting in the dissection $D(\boldsymbol{a})$
  Sol $\leftarrow$ Sol $\cup \{\Pi\}$
$\Pi' \leftarrow \operatorname{argmin}\{l(\Pi) : \Pi \in \mathrm{Sol}\}$
return BACK-PERTURBATION(Π')

that the algorithm can also be applied the other way around, i.e., given a solution to instance $\mathcal{I}$, creating a solution to $\mathcal{I}'$ of length increased by at most $\mathcal{O}(\varepsilon)\mathrm{OPT}$. This implies that $\mathrm{OPT}_p \leq (1+\mathcal{O}(\varepsilon))\mathrm{OPT}$.

Coming back to applying the algorithm on a $(1+\varepsilon')$-approximation for $\mathcal{I}'$ that is obtained by applying the algorithm PERTURBATION on $\mathcal{I}$, we obtain that the cost of the resulting tour is at most

$$(1+\varepsilon')\mathrm{OPT}_p + \mathcal{O}(\varepsilon)\mathrm{OPT} \leq ((1+\varepsilon')(1+\mathcal{O}(\varepsilon)) + \mathcal{O}(\varepsilon))\mathrm{OPT} = (1+\varepsilon'+\mathcal{O}(\varepsilon))\mathrm{OPT},$$

which completes the proof.

□

4.6 A $(5/3+\varepsilon)$-approximation for 3-ETSP

We have all the prerequisites in place to prove our main result. We begin by recalling the theorem.

Theorem 4.1. *For every $\varepsilon > 0$, there exists an algorithm that computes a $\left(\frac{5}{3}+\varepsilon\right)$-approximation for the* 3-ETSP *in time $\left(\frac{n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)}$.*

Proof. We can check whether a given instance is ε-reducible in time $(\frac{n}{\varepsilon})^{\mathcal{O}(1/\varepsilon)}$ by simply checking for all three choices of a partition of the three colors $\{c, c'\} \cup \{c''\} = \{\mathrm{R}, \mathrm{G}, \mathrm{B}\}$, whether applying the known PTAS for 2-ETSP [45] on T_c and $T_{c'}$, and independently applying Arora's PTAS for 1-ETSP [5] on $T_{c''}$, yields disjoint tours. In that case, we have found a solution as desired. Therefore, assume that the given instance is not ε-reducible. In that case, we show that Algorithm 6 computes a $\left(\frac{5}{3}+\varepsilon\right)$-approximation for 3-ETSP in time $(\frac{n}{\varepsilon})^{\mathcal{O}(1/\varepsilon)}$.

In Theorem 4.22 a), we have seen that the algorithm PERTURBATION outputs $L = \mathcal{O}(nM/\varepsilon)$ and an instance $(T'_{\mathrm{R}}, T'_{\mathrm{G}}, T'_{\mathrm{B}})$ of 3-ETSP with terminals lying

in $\{0,\dots,L\}^2$. Let OPT_p denote the optimal cost of this instance. By Corollary 4.16, there exists a shift vector $\boldsymbol{a} \in \{0,\dots,L-1\}^2$ and $\delta > 0$ small enough such that there exists a $(\lceil 14M/\varepsilon \rceil, 36, \delta)$-portal respecting solution of cost at most $\left(\frac{5}{3} + \frac{\varepsilon}{M}\right) \cdot \text{OPT}_p$ to the perturbed instance. Since, in the for-loop, we check all choices for $\boldsymbol{a}$ and, in line 7, output the shortest such solution, we obtain that Π' has cost upper bounded by $\left(\frac{5}{3} + \frac{\varepsilon}{M}\right) \cdot (1+\mu) \cdot \text{OPT}_p$.

By Theorem 4.22 b), Back-Perturbation(Π') is a solution to the original instance of cost at most $\left((\frac{5}{3} + \frac{\varepsilon}{M})(1+\mu) + \mathcal{O}\left(\frac{\varepsilon}{M}\right)\right) \cdot \text{OPT}$. Therefore, it is possible to choose the constant M large enough and $\mu > 0$ small enough such that the cost of the resulting solution is at most $(5/3+\varepsilon) \cdot \text{OPT}$ as desired.

It remains to estimate the running time of Algorithm 6. By Theorem 4.22, the steps in line 1 and 8 have running time $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$. By Theorem 4.17, the step in line 5 has running time $L^{\mathcal{O}((M/\varepsilon)\cdot 36\cdot \log(3))} = \left(\frac{n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)}$. The for-loop is executed $L^2 = \mathcal{O}\left(\left(\frac{n}{\varepsilon}\right)^2\right)$ times, so that the overall running time is $\left(\frac{n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)}$, which completes the proof of the Theorem. □

4.7 Outlook

In this chapter, we initiated the study of the k-ETSP for $k \geq 3$ and have seen that the problem is substantially more challenging than the cases $k \leq 2$. This is because classical techniques that succeed for one and two colors fail when extended to three or more. The central open question remains whether a PTAS exists for $k \geq 3$ (Problem 4.2).

In terms of generalizing Arora's algorithm, we have seen that, for any number of colors, an optimum portal-respecting solution can be computed in polynomial time. Furthermore, for every $\varepsilon > 0$, there exists a $(1+\varepsilon)$-approximation that only crosses the dissection at portals (where the portal placement depends on ε). However, the crucial patching lemma does not generalize to more than 2 colors. In other words, we can locally modify tours so that they only cross at portals, but we cannot modify them such that they cross each portal at most a constant number of times.

This indicates that new ideas are needed to resolve Problem 4.2. First, note that it is unclear whether the non-patchable tours depicted in Figure 4.2 actually appear as optimal solutions of some instance. To rule out the possibility of obtaining a PTAS via optimal portal-respecting solutions, one would need to construct an example in which every $(1+\varepsilon)$-approximation crosses some portal more than a constant number of times, for any shift vector $\boldsymbol{a}$ (which was a parameter of the dissection).

We observe that, even if one cannot bound the number of crossings per portal by a constant, as long as the number of possible crossing patterns is at most a polynomial, this would yield a *quasi-polynomial-time approximation scheme (QPTAS)*, that is, an algorithm that computes a $(1+\varepsilon)$-approximation in time $n^{\mathcal{O}_\varepsilon(\log n)}$. For instance, for the case of spirals as in Figure 4.1, even though the number of crossings may be polynomial, the number of crossing patterns is also only polynomial (even though one might expect 3^m possible crossing patterns for m crossings). This is because the

crossing pattern is fully desribed by an order of the colors and the number of windings of the spiral. A natural direction for further investigation is to analyze spirals where the winding direction alternates and study whether, for such configurations, a super-polynomial number of crossing patterns is possible.

Chapter 5

Online dial-a-ride

In the open online dial-a-ride problem, we are given a metric space (M, d), typically $\mathbb{R}^n$ or $\mathbb{R}^n_{\geq 0}$, and have control of a server that can move at unit speed. Over time, *requests* of the form $(a, b; t)$ arrive. Here, $a \in M$ is the *starting position* of the request, $b \in M$ is its *destination*, and $t \in \mathbb{R}_{\geq 0}$ is the *release time* of the request. We consider the online variant of the problem, meaning that the server does not get to know all requests at time 0, but rather at the respective release times. Our task is to control the server such that it serves all requests, i.e., we have to move the server to position a, load the request $(a, b; t)$ there after its release time t, and then move to position b where we unload the request. The objective is to minimize the *completion time*, i.e., the time when all requests are served. Importantly, the server is not told which request is the last one. This means that we have to minimize the completion time of the currently revealed requests at all times.

We assume that the server always starts at time 0 in some fixed point, which we call the *origin* $O \in M$. The server has a *capacity* $c \in (\mathbb{N} \cup \{\infty\})$ and is not allowed to have more than c requests loaded at the same time. Furthermore, our main focus is the *non-preemptive* version of the problem, that is, the server may not unload a request preemptively at a point that is not the request's destination. In the dial-a-ride problem, a distinction is made between the open and the closed variant. In the *closed* dial-a-ride problem, the server has to return to the origin after serving all requests. In contrast, in the *open* dial-a-ride problem, the server may finish anywhere in the metric space. In this work, we focus on the open variant of the problem. An example for an instance of the problem is illustrated in Figure 5.1.

In this chapter, we present a deterministic parametrized algorithm called $\textsc{Lazy}_\alpha$ for open online dial-a-ride. Our main result is a tight analysis in general metric spaces.

Theorem 5.1. *For $\alpha = \frac{1}{2} + \sqrt{11/12}$, the algorithm $\textsc{Lazy}_\alpha$ has a competitive ratio of $\alpha + 1 \approx 2.457$ for open online dial-a-ride on general metric spaces for every capacity $c \in \mathbb{N} \cup \{\infty\}$. In this, the choice of the parameter α is optimal.*

Prior to our work, the best known upper bound on the competitive ratio of the problem was 2.696 [21]. In addition to the study on general metric spaces, we analyze Lazy for open online dial-a-ride on the half-line, i.e., where $M = \mathbb{R}_{\geq 0}$.

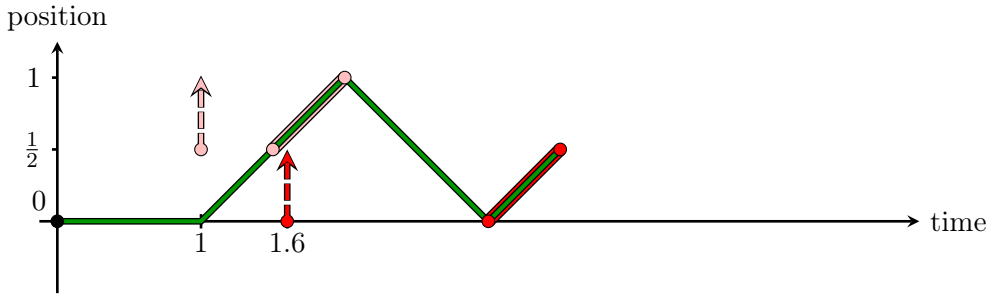

Figure 5.1: Example of an instance for open online dial-a-ride with capacity 1. The first request revealed is $r_1 = (0.5, 1; 1)$ (depicted in pink). The server travels immediately towards the starting position of r_1 and picks it up at time 1.5. At time 1.6, the second request $r_2 = (0, 0.5; 1)$ is revealed (depicted in red). Since the capacity is only 1 and we study the non-preemptive case, the server has to deliver r_1 before it can pick up r_2. We have $\text{ALG}(\sigma) = 3.5$ and $\text{OPT}(\sigma) = 2.6$ because the offline optimum waits in 0 until r_2 is revealed, then serves r_2, and then serves r_1.

We show that, in this special metric space, even better bounds on the competitive ratio are possible by setting the value of α differently. More precisely, we show the following.

Theorem 5.2. *For $\alpha = \frac{1+\sqrt{3}}{2}$, LAZY_α has a competitive ratio of $\alpha + 1 \approx 2.366$ for open online dial-a-ride on the half-line for every capacity $c \in \mathbb{N} \cup \{\infty\}$. In this, the choice of the parameter α is optimal.*

Prior to our work, the best known upper bound for the problem on the half-line was the one inherited from general metric spaces, i.e., 2.696 [21].

While we mainly work with the non-preemptive version of the problem, we will show that our bounds also hold in the preemptive case (Corollary 5.17). Consequently, we also obtain improved bounds for this version of the problem. An overview of the different variants of open online dial-a-ride and our results is given in Table 5.1.

5.1 Preliminaries

First, we observe, similarly to Section 2.1.3, that for the open online dial-a-ride problem there is no difference between the competitive ratio and the strict competitive ratio. This allows us to use these terms interchangeably.

Observation 5.3. *Consider the open online dial-a-ride problem on $\mathbb{R}^n$ or $\mathbb{R}^n_{\geq 0}$ with $n \geq 1$. The competitive ratio and the strict competitive ratio of the problem coincide.*

Proof. It is clear that any strictly ρ-competitive algorithm is also ρ-competitive. For the other direction, we argue similarly as we did for exploration using the adversary

metric space		old bounds		new bounds
		lower	upper	upper
general	non-preemptive	2.05	**2.696** [21]	**2.457** (Thm 5.1)
	preemptive	2.04	2.696	2.457 (Cor 5.17)
line	non-preemptive	**2.05** [24]	2.696	2.457
	preemptive	**2.04** [25]	**2.41** [25]	—
half-line	non-preemptive	**1.9** [90]	2.696	**2.366** (Thm 5.2)
	preemptive	**1.62** [90]	2.41	2.366 (Cor 5.17)

Table 5.1: State of the art of the open online dial-a-ride problem and overview of our results: Bold bounds are original results, other bounds are inherited.

model (see Section 2.1.3). Assume that there is no strictly ρ-competitive algorithm. Then there exists an adversarial strategy that constructs, for every deterministic algorithm ALG, an instance σ with $\text{ALG}(\sigma) \geq \rho \cdot \text{OPT}(\sigma) + \varepsilon$ for some $\varepsilon > 0$. Note that we can scale the instance σ by any non-negative $M \in \mathbb{R}_{\geq 0}$, meaning that we multiply all release times, starting positions, and destinations with M. This results in an adversarial strategy that constructs for every algorithm an instance σ with $\text{ALG}(\sigma) \geq \rho \cdot \text{OPT}(\sigma) + M\varepsilon$. Letting $M \to \infty$, this shows that there is no ρ-competitive algorithm. □

Next, we give an overview of the algorithms for online dial-a-ride from the literature.

5.1.1 State of the art

Ignore and Replan. Two of the most natural algorithms for the online dial-a-ride problem are IGNORE and REPLAN. The basic idea of IGNORE is to repeatedly follow an optimum schedule over the currently unserved requests and ignoring all requests released during its execution. The competitive ratio of this algorithm is known to be exactly 4 [21, 83]. By contrast, the main idea of REPLAN is to start a new schedule over all unserved requests whenever a new request is released. While this algorithm is very natural and may be the first algorithm studied for the online dial-a-ride problem, it has turned out to be notoriously difficult to analyze and eluded tight analysis up to this day. However, it is known that its competitive ratio is at least 2.5 [10] and at most 4 [21]. When $c = 1$, even an upper bound of 3 is known [83]. So far, REPLAN has been a canonical candidate for a best-possible algorithm. We finally rule it out as our algorithm LAZY has a competitive ratio of 2.457, which beats the known lower bound of 2.5 for REPLAN.

Adaptions of Ignore and Replan. Several variants of the algorithms IGNORE and REPLAN have been proposed, such as SMARTSTART [83], SMARTERSTART [24] or WAITORIGNORE [90], which lead to improvements on the best known bounds on the competitive ratio of the dial-a-ride problem. Prior to this work, the best

known upper bound for the open online dial-a-ride problem, in both the preemptive and non-preemptive version, was 2.696 and was achieved by the algorithm SmarterStart [21]. Bjelde et al. [25] proved a stronger bound of $1 + \sqrt{2} \approx 2.41$ for the preemptive version when the metric space is the line.

Schedule-based algorithms. Crucially, our upper bound of 2.457 for Lazy (Theorem 5.1) beats, for the first time, a known lower bound of 2.5 for the class of *schedule-based* algorithms [21], that is, algorithms that divide the execution into subschedules, such that, when such a subschedule is started, it serves all requests that are at that time revealed but not yet served, and these subschedules are never interrupted. Note that interruption of a schedule is a quite different notion than preemption: Preemption means that a request can be unloaded without being served, i.e., unloaded at another position than its destination. Conversely, interruption of a schedule means that the server changes its plan that it has made to serve several requests (defined in more detail in Section 5.2.1). In particular, we can speak of interruption of a schedule in the non-preemptive version of the problem. Historically, all upper bounds, prior to those via Lazy, were based on schedule-based algorithms [22, 24]. Our results imply that online algorithms cannot afford to irrevocably commit to serving some subset of requests if they hope to attain the best possible competitive ratio.

Lower bounds. First, observe that, if the metric space is the real line, we easily obtain a lower bound of 2 on the competitive ratio of the problem. To see this, consider the following adversarial strategy: We construct a request sequence σ consisting of a single request r that is revealed at time 1. If, at that time, the server is located in a non-positive point, we let $r = (1, 1; 1)$. Otherwise, we let $r = (-1, -1; 1)$. In either case, at time 1, the server needs at least one additional time unit to serve r. Therefore, we have $\text{Alg}(\sigma) \geq 2$. Conversely, the server in the offline optimum solution starts moving at time 0 towards the starting position of r and reaches it at time 1 so that $\text{Opt}(\sigma) = 1$.

Beating this simple lower bound of 2 has turned out to be very challenging. However, Birx et al. [24] were able to give a slight improvement and proved that every algorithm for the open online dial-a-ride problem has a competitive ratio of at least 2.05, even if the metric space is the line. For open online dial-a-ride on the half-line, Lipmann [90] established a lower bound of 1.9 for the non-preemptive version and a lower bound of 1.62 for the preemptive version.

Online TSP and closed dial-a-ride. An important special case of dial-a-ride is the *online traveling salesperson problem*, which is obtained by letting $a = b$ for all requests $(a, b; t)$. For open online TSP, it is known that the competitive ratio is tigh +ötly $\approx$ 2.04 on the real line [25] and at most 2.41 on general metric spaces [28]. The closed variant of online TSP is tightly analyzed with a competitive ratio of 2 on general metric spaces [6, 10, 52], of 1.64 on the line [10, 25], and of 1.5 on the half-line [26].

For the closed variant of the online dial-a-ride problem, the competitive ratio is known to be exactly 2 on general metric spaces [6, 10, 52] and between 1.76 and 2 on the line [21, 25]. On the half-line the best known lower bound is 1.71 [6] and the best known upper bound is 2 [6, 52].

5.1.2 Factor-revealing approach

Many of our upper bound results for LAZY were informed by a *factor-revealing technique*, inspired by a similar approach of Bienkowski et al. [19]. For this, note that the adversary problem (cf. Section 2.1) for LAZY can be formulated as the following optimization problem

$$\max\left\{\frac{\textsc{Alg}(\sigma)}{\textsc{Opt}(\sigma)} \;\middle|\; \sigma \text{ describes a dial-a-ride instance}\right\}. \tag{5.1}$$

An optimum solution to this problem immediately yields the competitive ratio of ALG. Of course, we cannot hope to solve this optimization problem or even describe it with a finite number of variables. The main idea in the factor-revealing approach is to relax (5.1) to a practically solvable problem over a finite number of variables.

The key is to select a set of variables that captures the structure of the problem well enough to allow for meaningful bounds. The algorithm LAZY will consist of so-called *schedules* and we can, for example, introduce variables for the starting positions and durations of the second-to-last schedule and the last schedule. We then need to relate those variables via constraints that ensure that an optimum solution to the relaxed problem actually has a realization as a dial-a-ride instance. For example, we might add the constraint that the distance between the starting positions of the last two schedules is upper bounded by the duration of the second-to-last schedule.

The power of the factor-revealing approach is that it allows to follow an iterative process for deriving structurally crucial inequalities: When solving the relaxed optimization problem, we generally have to expect an optimum solution that is not realizable and overestimates the competitive ratio. We can then focus our efforts on understanding why the corresponding variable assignment cannot be realized by a dial-a-ride instance. Then, we can introduce additional variables and constraints to exclude such solutions. In this way, the unrealizable solutions inform our analysis in the sense that we obtain bounds on the competitive ratio that can be proven analytically by only using the current set of variables and inequalities. Once we obtain a realizable lower bound, we thus have found the exact competitive ratio of the algorithm under investigation.

In order to practically solve the relaxed optimization problems, we limit ourselves to linear programs (LPs). Note that the objective of (5.1) is linear if we normalize to $\textsc{Opt}(\sigma) = 1$. For open online dial-a-ride on a scalable metric space such as $\mathbb{R}^n$ or $R^n_{\geq 0}$, this is without loss of generality because the competitive ratio is invariant with respect to rescaling the release times, starting positions, and destinations of

the requests. Another advantage of using linear programs is that we immediately obtain a formal proof from an optimum solution to the dual program. Of course, the correctness of the involved inequalities still needs to be established.

In this chapter, we present a purely analytic proof of our results. Many of the inequalities we derive in lemmas were informed by a factor-revealing approach via a linear program with a small number of binary variables. As a consequence, we cannot easily obtain a proof by simply providing an optimal dual solution. One possibility would be to branch on all binary variables, that is, to solve the linear program for every possible assignment of the binary variables. However, since we require four binary variables, this would entail presenting $2^4 = 16$ linear programs along with their dual solutions. For this reason, we opt for an analytic proof, which is more tractable. Nevertheless, solving the MILP informed our analysis in the sense that it helped us identify the crucial properties and inequalities. We refer to Section 5.2.4 for more details of the binary program that informed our results for the half-line.

5.2 Improved upper bounds for open online dial-a-ride

In this section, we introduce the algorithm Lazy and prove that it improves on the best known competitive ratio of the open online dial-a-ride problem on general metric spaces, and on the half-line. At the end of this section, we demonstrate how the factor-revealing approach was used for our findings.

5.2.1 The algorithm Lazy

The rough idea of the algorithm Lazy is to wait until several requests are revealed and then start a schedule serving them. Whenever a new request arrives, we check whether we can deliver all currently loaded requests and return to the origin in a reasonable time. If this is possible, we do so and, after possibly waiting for some time, begin a new schedule including the new requests starting from the origin. If this is not possible, we keep following the current schedule and consider the new request later.

More formally, we say that a request is *loaded* when it has been picked up by the server, but it has not yet reached the request's destination. We will also say that a request has been *loaded at time* t and *unloaded at time* t', if t is the first point in time when it is loaded and t' is the last point in time when it is loaded. A *schedule* is a sequence of actions specifying the server's behavior, including its movement and where requests are loaded or unloaded. Given a set of requests R and some point $x \in M$, we denote by $S(R, x)$ a shortest schedule serving all requests in R beginning from point x at some time after all requests in R are released. In other words, we can ignore the release times of the requests when computing $S(R, x)$. As waiting is not beneficial for the server if there are no release times, the *length of*

Algorithm 7: LAZY$_\alpha$

initialize: $i \leftarrow 0$

upon receiving a request:
if *server can serve all loaded requests and return to O until time $\alpha \cdot \textsc{Opt}(t)$* **then**
  execute DELIVER_AND_RETURN

upon becoming idle:
if $t < \alpha \cdot \textsc{Opt}(t)$ **then**
  execute WAIT_UNTIL($\alpha \cdot \textsc{Opt}(t)$)
else if $R_t \neq \emptyset$ **then**
  $i \leftarrow i + 1$, $R^{(i)} \leftarrow R_t$, $t^{(i)} \leftarrow t$, $p^{(i)} \leftarrow p_t$
  $S^{(i)} \leftarrow S(R^{(i)}, p^{(i)})$
  execute FOLLOW_SCHEDULE($S^{(i)}$)

the schedule, i.e., the distance the server travels, is the same as the time needed to complete it and we denote this by $|S(R, x)|$. By $\textsc{Opt}[t]$, we denote an optimal schedule beginning in O at time 0 and serving all requests (taking into account their release times) that are released not later than time t. By $\textsc{Opt}(t)$, we denote its completion time.[1]

Now that we have established the notation needed, we can describe the algorithm (cf. Algorithm 7). By t, we denote the current time. By p_t, we denote the position of the server at time t, and by R_t, we denote the set of requests that have been released but not served until time t. The variable i is a counter over the schedules started by the algorithm. By $S^{(i)}$, we denote the i-th schedule started by the algorithm. By $t^{(i)}$, we denote its starting time, by $p^{(i)}$, its starting position, and by $R^{(i)}$, the unserved requests until time $t^{(i)}$. The waiting parameter $\alpha \geq 1$ specifies how long we wait before starting a schedule. The algorithm uses the following commands: DELIVER_AND_RETURN orders the server to finish serving all currently loaded requests and return to O in the fastest possible way, WAIT_UNTIL(t) orders the server to remain at its current location until time t, and FOLLOW_SCHEDULE(S) orders the server to execute the actions defined by schedule S. When any of these commands is invoked, the server aborts what it is doing and executes the new command. Whenever the server has completed a command, we say that it becomes *idle*.

We make a few comments for illustration of the algorithm. First, note that when none of the if-statements are fulfilled, the server simply continues what it was doing before, i.e., it either continues its execution of one of the commands, or it keeps being idle. If, upon receiving a request, the server returns to the origin before

[1] By slight abuse of notation, in this chapter, we allow Opt to take either a request sequence σ or a time t as its argument. Evaluating Opt at time t corresponds to evaluating on the restriction of σ to the requests revealed no later than t. The sequence σ is omitted in this notation as it will always be clear from context.

completing its current schedule, i.e., the if-statement in line 3 of the algorithm holds and line 4 is executed, we say that this schedule is *interrupted*. Observe that, due to interruption, the sets $R^{(i)}$ are not necessarily disjoint. Also, observe that $p^{(1)} = O$, and if schedule $S^{(i)}$ was interrupted, we have $p^{(i+1)} = O$ and $t^{(i+1)} = \alpha \cdot \textsc{Opt}(t^{(i+1)})$. If $S^{(i)}$ was not interrupted, $p^{(i+1)}$ is the ending position of $S^{(i)}$.

The following observations follow directly from the definitions above and the fact that requests in $R^{(i)} \setminus R^{(i-1)}$ were released after time $t^{(i-1)}$.

Observation 5.4. *For every request sequence, the following hold.*

a) For every $i > 1$, $\textsc{Opt}(t^{(i)}) \geq t^{(i-1)} \geq \alpha \cdot \textsc{Opt}(t^{(i-1)})$.

b) For every $x, y \in M$ and every subset of requests R, we have

$$|S(R, x)| \leq d(x, y) + |S(R, y)|.$$

c) Let $i > 1$ and assume that $S^{(i-1)}$ was not interrupted. Let x be the starting position of the request in $R^{(i)}$ that is picked up first by $\textsc{Opt}(t^{(i)})$. Then

$$\textsc{Opt}(t^{(i)}) \geq t^{(i-1)} + |S(R^{(i)}, x)| \geq \alpha \cdot \textsc{Opt}(t^{(i-1)}) + |S(R^{(i)}, x)|.$$

5.2.2 Upper bound for Lazy on general metric spaces

This section is concerned with the proof of the upper bound in Theorem 5.1. For the remainder of this section, let $(r_1, \ldots, r_n)$ be some fixed request sequence. Let k be the number of schedules started by $\textsc{Lazy}_\alpha$, and let $S^{(i)}$, $t^{(i)}$, $p^{(i)}$, $R^{(i)}$ $(1 \leq i \leq k)$ be defined as in the algorithm. Note that we slightly abuse notation here because k, $S^{(i)}$, $t^{(i)}$, $p^{(i)}$, and $R^{(i)}$ depend on α. As it will always be clear from the context what α is, we allow this implicit dependency in the notation.

As it will be crucial for the proof in which order Opt and Lazy serve requests, we introduce the following notation. Let

- $r^{(i)}_{f,\textsc{Opt}} = (a^{(i)}_{f,\textsc{Opt}}, b^{(i)}_{f,\textsc{Opt}}; t^{(i)}_{f,\textsc{Opt}})$ be the first request in $R^{(i)}$ picked up by $\textsc{Opt}[t^{(i)}]$,
- $r^{(i)}_{l,\textsc{Opt}} = (a^{(i)}_{l,\textsc{Opt}}, b^{(i)}_{l,\textsc{Opt}}; t^{(i)}_{l,\textsc{Opt}})$ be the last request in $R^{(i)}$ delivered by $\textsc{Opt}[t^{(i+1)}]$,
- $r^{(i)}_{f,\textsc{Lazy}} = (a^{(i)}_{f,\textsc{Lazy}}, b^{(i)}_{f,\textsc{Lazy}}; t^{(i)}_{f,\textsc{Lazy}})$ be the first request in $R^{(i)}$ picked up by $\textsc{Lazy}_\alpha$,
- $r^{(i)}_{l,\textsc{Lazy}} = (a^{(i)}_{l,\textsc{Lazy}}, b^{(i)}_{l,\textsc{Lazy}}; t^{(i)}_{l,\textsc{Lazy}})$ be the last request in $R^{(i)}$ delivered by $\textsc{Lazy}_\alpha$.

Note that, if $S^{(i)}$ was not interrupted, the ending position of the server in that schedule is also the starting position of the next schedule, i.e., $b^{(i)}_{l,\textsc{Lazy}} = p^{(i+1)}$.

Definition 5.5. We say that the i-th schedule is *α-good* if

a) $|S^{(i)}| \leq \textsc{Opt}(t^{(i)})$ and

b) $t^{(i)} + |S^{(i)}| \leq (1 + \alpha) \cdot \textsc{Opt}(t^{(i)})$.

In this section, we prove by induction on i that, for $\alpha \geq \frac{1}{2} + \sqrt{11/12}$, every schedule is α-good. Note that this immediately implies the upper bound in Theorem 5.1.

We begin with proving the base case.

Observation 5.6 (Base case)**.** *For every $\alpha \geq 1$, the first schedule is α-good.*

Proof. Recall that $S^{(1)}$ begins in O and is the shortest tour serving all requests in $R^{(1)}$. $\textsc{Opt}[t^{(1)}]$ begins in O and serves all requests in $R^{(1)}$, too. This implies that $|S^{(1)}| \leq \textsc{Opt}(t^{(1)})$. From the fact that we have $t^{(1)} = \alpha \cdot \textsc{Opt}(t^{(1)})$, it follows that $t^{(1)} + |S^{(1)}| \leq (1+\alpha) \cdot \textsc{Opt}(t^{(1)})$. □

Next, we observe briefly that the induction step is not too difficult when the last schedule was interrupted.

Observation 5.7 (Interruption case)**.** *Let $\alpha \geq 1$. Assume that schedule $S^{(i)}$ was interrupted. Then $S^{(i+1)}$ is α-good.*

Proof. If schedule $S^{(i)}$ was interrupted, we have $p^{(i+1)} = O$ and $t^{(i+1)} = \alpha \cdot \textsc{Opt}(t^{(i+1)})$. Note that $\textsc{Opt}(t^{(i+1)})$ begins in O and serves, amongst others, all requests in $R^{(i+1)}$ so that we have

$$|S^{(i+1)}| = |S(R^{(i+1)}, O)| \leq \textsc{Opt}(t^{(i+1)}).$$

Using that $t^{(i+1)} = \alpha \cdot \textsc{Opt}(t^{(i+1)})$, this also implies condition (b) of being α-good, i.e., that $t^{(i+1)} + |S^{(i+1)}| \leq (1+\alpha) \cdot \textsc{Opt}(t^{(i+1)})$. □

For this reason, we will assume in many of the following statements that the schedule $S^{(i)}$ was not interrupted.

The following lemma shows that the first requirement for schedule $S^{(i+1)}$ to be α-good is satisfied.

Lemma 5.8. *Let $\alpha \geq \frac{1+\sqrt{17}}{4} \approx 1.281$ and $i \in \{1, \ldots, k-1\}$. If $S^{(i)}$ is α-good, then we have $|S^{(i+1)}| \leq \textsc{Opt}(t^{(i+1)})$.*

Proof. First, note that if $S^{(i)}$ was interrupted, the statement follows by Observation 5.7. Therefore, assume from now on that $S^{(i)}$ was not interrupted. Also, if $\textsc{Opt}[t^{(i+1)}]$ serves $r^{(i)}_{l,\textsc{Lazy}}$ at $p^{(i+1)}$ before collecting any request from $R^{(i+1)}$, we trivially have

$$|S^{(i+1)}| = |S(R^{(i+1)}, p^{(i+1)})| \leq \textsc{Opt}(t^{(i+1)}).$$

Therefore, assume additionally that $\textsc{Opt}[t^{(i+1)}]$ collects $r^{(i+1)}_{f,\textsc{Opt}}$ before serving $r^{(i)}_{l,\textsc{Lazy}}$. Next, we prove the following assertion.

Claim 5.9. *In the setting described above, we have*

$$d(a^{(i+1)}_{f,\textsc{Opt}}, p^{(i+1)}) \leq \left(1 + \frac{2}{\alpha} - \alpha\right) \textsc{Opt}(t^{(i)}). \tag{5.2}$$

Proof. Note that $r^{(i+1)}_{f,\textsc{Opt}}$ is released not earlier than $t^{(i)} \geq \alpha \textsc{Opt}(t^{(i)})$. Since we assume that $\textsc{Opt}(t^{(i+1)})$ collects $r^{(i+1)}_{f,\textsc{Opt}}$ before serving $r^{(i)}_{l,\textsc{Lazy}}$ at $p^{(i+1)}$, we obtain

$$\textsc{Opt}(t^{(i+1)}) \geq \alpha \cdot \textsc{Opt}(t^{(i)}) + d(a^{(i+1)}_{f,\textsc{Opt}}, p^{(i+1)}). \tag{5.3}$$

Upon the arrival of the last-revealed request in $R^{(i+1)}$, we have $\textsc{Opt}(t) = \textsc{Opt}(t^{(i+1)})$ and, at that time, the server can finish its current schedule and return to the origin

by time $t^{(i)} + |S^{(i)}| + d(p^{(i+1)}, O)$. As we assume that $S^{(i)}$ was not interrupted, this yields

$$t^{(i)} + |S^{(i)}| + d(p^{(i+1)}, O) > \alpha \cdot \textsc{Opt}(t^{(i+1)}). \tag{5.4}$$

Combined, we obtain that

$$\begin{aligned} d(a_{f,\textsc{Opt}}^{(i+1)}, p^{(i+1)}) &\overset{(5.3)}{\leq} \textsc{Opt}(t^{(i+1)}) - \alpha \cdot \textsc{Opt}(t^{(i)}) \\ &\overset{(5.4)}{\leq} \frac{1}{\alpha} \cdot \big(t^{(i)} + |S^{(i)}| + d(p^{(i+1)}, O)\big) - \alpha \cdot \textsc{Opt}(t^{(i)}) \\ &\overset{S^{(i)}\alpha\text{-good}}{\leq} \frac{1}{\alpha} \cdot \big((1+\alpha) \cdot \textsc{Opt}(t^{(i)}) + d(p^{(i+1)}, O)\big) - \alpha \cdot \textsc{Opt}(t^{(i)}) \\ &\leq \left(1 + \frac{2}{\alpha} - \alpha\right) \textsc{Opt}(t^{(i)}), \end{aligned}$$

where we have used in the last inequality that $d(p^{(i+1)}, O) \leq \textsc{Opt}(t^{(i)})$ because $\textsc{Opt}(t^{(i)})$ begins in O and has to serve $r_{l,\textsc{Lazy}}^{(i)}$ at $p^{(i+1)}$. This completes the proof of the claim. □

Now, we turn back to proving Lemma 5.8. We obtain

$$\begin{aligned} |S^{(i+1)}| &\leq d(p^{(i+1)}, a_{f,\textsc{Opt}}^{(i+1)}) + |S(R^{(i+1)}, a_{f,\textsc{Opt}}^{(i+1)})| \\ &\overset{\text{Obs 5.4c)}}{\leq} d(p^{(i+1)}, a_{f,\textsc{Opt}}^{(i+1)}) + \textsc{Opt}(t^{(i+1)}) - \alpha \cdot \textsc{Opt}(t^{(i)}) \\ &\overset{(5.2)}{\leq} \left(1 + \frac{2}{\alpha} - 2\alpha\right) \textsc{Opt}(t^{(i)}) + \textsc{Opt}(t^{(i+1)}) \\ &\leq \textsc{Opt}(t^{(i+1)}), \end{aligned}$$

where the last inequality follows from the fact that $1 + \frac{2}{\alpha} - 2\alpha \leq 0$ if and only if $\alpha \geq \frac{1+\sqrt{17}}{4} \approx 1.2808$. □

Recall that the goal of this section is to prove that every schedule is α-good. So far, we have proven the base case (Observation 5.6) and $|S^{(i+1)}| \leq \textsc{Opt}(t^{(i+1)})$ (Lemma 5.8) in the induction step. To complete the induction step, it remains to show that $t^{(i+1)} + |S^{(i+1)}| \leq (1+\alpha) \cdot \textsc{Opt}(t^{(i+1)})$ assuming $S^{(1)}, \ldots, S^{(i)}$ are α-good. In Observation 5.7, we have already seen that this holds if $S^{(i)}$ was interrupted. To show that the induction step also holds if $S^{(i)}$ was not interrupted, we distinguish several cases for the order in which Opt serves the requests. In the following proofs, when we talk about "the server's behavior", we always refer to the behavior of Lazy. By contrast, we use "the behavior of $\textsc{Opt}[t]$" to specify, e.g., the order in which an optimum solution handles the requests (that are revealed by time t). We begin with the case that $\textsc{Opt}[t^{(i+1)}]$ picks up some request in $R^{(i+1)}$ before serving $r_{l,\textsc{Lazy}}^{(i)}$, i.e., that $\textsc{Opt}[t^{(i+1)}]$ does not follow the order of the $S^{(i)}$.

Lemma 5.10. *Let $\alpha \geq 1$. Assume that $S^{(i)}$ is α-good and was not interrupted, and that $\textsc{Opt}[t^{(i+1)}]$ picks up the request $r_{f,\textsc{Opt}}^{(i+1)}$ before serving $r_{l,\textsc{Lazy}}^{(i)}$. Then we have $t^{(i+1)} + |S^{(i+1)}| \leq (1+\alpha) \cdot \textsc{Opt}(t^{(i+1)})$.*

Proof. Using the order in which OPT handles the requests, we obtain the following. After picking up $r^{(i+1)}_{f,\text{OPT}}$ at $a^{(i+1)}_{f,\text{OPT}}$ after time $t^{(i)}$, $\text{OPT}[t^{(i+1)}]$ has to serve $r^{(i)}_{l,\text{LAZY}}$ at $p^{(i+1)}$ so that

$$\text{OPT}(t^{(i+1)}) \geq t^{(i)} + d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}). \tag{5.5}$$

After finishing schedule $S^{(i)}$, the server either waits until time $\alpha \cdot \text{OPT}(t^{(i+1)})$ or immediately starts the next schedule, i.e., we have

$$t^{(i+1)} = \max\{\alpha \cdot \text{OPT}(t^{(i+1)}), t^{(i)} + |S^{(i)}|\}.$$

If $t^{(i+1)} = \alpha \cdot \text{OPT}(t^{(i+1)})$, the assertion follows immediately from Lemma 5.8. Thus, assume $t^{(i+1)} = t^{(i)} + |S^{(i)}|$. This yields

$$\begin{aligned}
t^{(i+1)} + |S^{(i+1)}| &\overset{\text{Obs 5.4b)}}{\leq} t^{(i)} + |S^{(i)}| + d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}) + |S(R^{(i+1)}, a^{(i+1)}_{f,\text{OPT}})| \\
&\overset{S^{(i)}\alpha\text{-good}}{\leq} (1+\alpha) \cdot \text{OPT}(t^{(i)}) + d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}) \\
&\qquad + |S(R^{(i+1)}, a^{(i+1)}_{f,\text{OPT}})| \\
&\overset{\text{Obs 5.4a)}}{\leq} \frac{1+\alpha}{\alpha} t^{(i)} + d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}) + |S(R^{(i+1)}, a^{(i+1)}_{f,\text{OPT}})| \\
&\overset{\text{Obs 5.4c)}}{\leq} \frac{1}{\alpha} t^{(i)} + d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}) + \text{OPT}(t^{(i+1)}) \\
&\overset{(5.5)}{\leq} \frac{1}{\alpha}\Big(\text{OPT}(t^{(i+1)}) - d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}})\Big) + d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}) \\
&\qquad + \text{OPT}(t^{(i+1)}) \\
&= \left(1 + \frac{1}{\alpha}\right)\text{OPT}(t^{(i+1)}) + \left(1 - \frac{1}{\alpha}\right) d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}) \\
&\leq 2 \cdot \text{OPT}(t^{(i+1)}),
\end{aligned}$$

where we have used in the last inequality that $d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}) \leq \text{OPT}(t^{(i+1)})$ as $\text{OPT}[t^{(i+1)}]$ has to visit both points. Since $2 \leq 1+\alpha$, this completes the proof. □

Next, we consider the case where OPT handles $r^{(i)}_{l,\text{LAZY}}$ and $r^{(i+1)}_{f,\text{OPT}}$ in the same order as LAZY.

Lemma 5.11. *Let $\alpha \geq 1$. Assume that schedules $S^{(1)}, \ldots, S^{(i)}$ are α-good, $S^{(i)}$ was not interrupted, and $\text{OPT}[t^{(i+1)}]$ serves $r^{(i)}_{l,\text{LAZY}}$ before collecting $r^{(i+1)}_{f,\text{OPT}}$. If we have*

$$d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}) + \text{OPT}(t^{(i)}) \leq \alpha \cdot \text{OPT}(t^{(i+1)}), \tag{5.6}$$

then $t^{(i+1)} + |S^{(i+1)}| \leq (1+\alpha) \cdot \text{OPT}(t^{(i+1)})$.

Proof. Similarly as in the proof of Lemma 5.10, we can assume

$$t^{(i+1)} = t^{(i)} + |S^{(i)}|. \tag{5.7}$$

We have

$$\begin{aligned} t^{(i+1)} + |S^{(i+1)}| &\overset{\text{Obs 5.4b)}}{\leq} t^{(i)} + |S^{(i)}| + d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}) + |S(R^{(i+1)}, a^{(i+1)}_{f,\text{OPT}})| \\ &\overset{S^{(i)}\alpha\text{-good}}{\leq} (1+\alpha)\cdot \text{OPT}(t^{(i)}) + d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}) \\ &\qquad + |S(R^{(i+1)}, a^{(i+1)}_{f,\text{OPT}})| \\ &\overset{\text{Obs 5.4c)}}{\leq} (1+\alpha)\cdot \text{OPT}(t^{(i)}) + d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}) \\ &\qquad + \text{OPT}(t^{(i+1)}) - \alpha \cdot \text{OPT}(t^{(i)}) \\ &= \text{OPT}(t^{(i+1)}) + d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}) + \text{OPT}(t^{(i)}) \\ &\leq (1+\alpha)\cdot \text{OPT}(t^{(i+1)}), \end{aligned}$$

where the last inequality follows from assumption (5.6). □

Now that we have proven the case described in Lemma 5.11, i.e., where (5.6) holds, we assume in the following the opposite case, i.e.,

$$d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}) > \alpha \cdot \text{OPT}(t^{(i+1)}) - \text{OPT}(t^{(i)}). \tag{5.8}$$

The following lemma states that, in this case, the $(i-1)$-th schedule (if it exists) was interrupted, i.e., the i-th schedule starts in the origin at time $\alpha \cdot \text{OPT}(t^{(i)})$.

Lemma 5.12. *Let $\alpha \geq \frac{1+\sqrt{3}}{2} \approx 1.366$. Assume that the i-th schedule is α-good and was not interrupted, and $\text{OPT}[t^{(i+1)}]$ serves $r^{(i)}_{l,\text{LAZY}}$ before collecting $r^{(i+1)}_{f,\text{OPT}}$. If (5.8) holds, then $p^{(i)} = O$ and $t^{(i)} = \alpha \cdot \text{OPT}(t^{(i)})$.*

Proof. If $i = 1$, we obviously have $p^{(i)} = O$ and $t^{(i)} = \alpha \cdot \text{OPT}(t^{(i)})$. Thus, assume that $i \geq 2$. If $r^{(i)}_{l,\text{LAZY}} \in (R^{(i-1)} \cap R^{(i)})$, schedule $S^{(i-1)}$ was interrupted and, thus, the statement holds. Otherwise, request $r^{(i)}_{l,\text{LAZY}}$ is released while schedule $S^{(i-1)}$ is running, i.e., $t^{(i)}_{l,\text{LAZY}} \geq t^{(i-1)} \geq \alpha \cdot \text{OPT}(t^{(i-1)})$. Combining this with the assumption that $\text{OPT}[t^{(i+1)}]$ serves $r^{(i)}_{l,\text{LAZY}}$ before collecting $r^{(i+1)}_{f,\text{OPT}}$, we obtain

$$\begin{aligned} \text{OPT}(t^{(i+1)}) &\geq t^{(i)}_{l,\text{LAZY}} + d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}) \\ &\geq \alpha \cdot \text{OPT}(t^{(i-1)}) + d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}). \end{aligned} \tag{5.9}$$

Rearranging yields

$$\begin{aligned} \alpha \cdot \text{OPT}(t^{(i-1)}) &\overset{(5.9)}{\leq} \text{OPT}(t^{(i+1)}) - d(p^{(i+1)}, a^{(i+1)}_{f,\text{OPT}}) \\ &\overset{(5.8)}{<} \text{OPT}(t^{(i)}) - (\alpha - 1)\text{OPT}(t^{(i+1)}) \\ &\overset{\text{Obs 5.4a)}}{\leq} (1 + \alpha - \alpha^2)\text{OPT}(t^{(i)}), \end{aligned}$$

which is equivalent to

$$\textsc{Opt}(t^{(i-1)}) < \big(1 + \frac{1}{\alpha} - \alpha\big)\textsc{Opt}(t^{(i)}). \tag{5.10}$$

By the assumption that $S^{(i-1)}$ is α-good, the server finishes schedule $S^{(i-1)}$ not later than time $(\alpha+1)\cdot\textsc{Opt}(t^{(i-1)})$. Thus, at the time when request $r^{(i)}_{l,\textsc{Lazy}}$ is released, the server can serve all loaded requests and return to the origin by time

$$\max\{(\alpha+1)\textsc{Opt}(t^{(i-1)}), t^{(i)}_{l,\textsc{Lazy}}\} + \textsc{Opt}(t^{(i-1)}). \tag{5.11}$$

In the case where $\max\{(\alpha+1)\textsc{Opt}(t^{(i-1)}), t^{(i)}_{l,\textsc{Lazy}}\} = (\alpha+1)\textsc{Opt}(t^{(i-1)})$, we can bound (5.11) by

$$\begin{aligned}(\alpha+2)\cdot\textsc{Opt}(t^{(i-1)}) &\overset{(5.10)}{<} (\alpha+2)\left(1+\frac{1}{\alpha}-\alpha\right)\textsc{Opt}(t^{(i)})\\ &= \left(3+\frac{2}{\alpha}-\alpha^2-\alpha\right)\textsc{Opt}(t^{(i)})\\ &\le \alpha\cdot\textsc{Opt}(t^{(i)}),\end{aligned}$$

where the last inequality holds for $\alpha \ge 1.343$.

In the case where $\max\{(\alpha+1)\textsc{Opt}(t^{(i-1)}), t^{(i)}_{l,\textsc{Lazy}}\} = t^{(i)}_{l,\textsc{Lazy}}$, we can bound (5.11) as follows. Since $r^{(i)}_{l,\textsc{Lazy}} \in R^{(i)}$, it holds that

$$\begin{aligned}t^{(i)}_{l,\textsc{Lazy}} + \textsc{Opt}(t^{(i-1)}) &\le \textsc{Opt}(t^{(i)}) + \textsc{Opt}(t^{(i-1)})\\ &\overset{(5.10)}{\le} \left(2+\frac{1}{\alpha}-\alpha\right)\textsc{Opt}(t^{(i)}) \le \alpha\cdot\textsc{Opt}(t^{(i)}),\end{aligned}$$

where the last inequality holds for $\alpha \ge \frac{1+\sqrt{3}}{2} \approx 1.366$. This implies that (5.11) can be bounded in either case by $\alpha\cdot\textsc{Opt}(t^{(i)})$, which means that the server can return to the origin by time $\alpha\cdot\textsc{Opt}(t^{(i)})$. Therefore, we have $p^{(i)} = O$. □

We now come to the technically most involved case.

Lemma 5.13. *Let $\alpha \ge \frac{1}{2} + \sqrt{11/12} \approx 1.457$. Assume that the i-th schedule is α-good and was not interrupted, and $\textsc{Opt}[t^{(i+1)}]$ serves $r^{(i)}_{l,\textsc{Lazy}}$ before collecting $r^{(i+1)}_{f,\textsc{Opt}}$. If (5.8) holds, then $t^{(i+1)} + |S^{(i+1)}| \le (1+\alpha)\cdot\textsc{Opt}(t^{(i+1)})$.*

Proof. We begin by proving the following assertion.

Claim 5.14. *$\textsc{Opt}[t^{(i+1)}]$ serves all requests in $R^{(i)}$ before picking up $r^{(i+1)}_{f,\textsc{Opt}}$ in $a^{(i+1)}_{f,\textsc{Opt}}$.*

Proof. To prove the claim, assume otherwise, i.e., that $\textsc{Opt}[t^{(i+1)}]$ serves $r^{(i)}_{l,\textsc{Opt}}$ after collecting $r^{(i+1)}_{f,\textsc{Opt}}$. The request $r^{(i+1)}_{f,\textsc{Opt}}$ is released after schedule $S^{(i)}$ is started, i.e.,

after time $\alpha \cdot \textsc{Opt}(t^{(i)})$. Thus,

$$\begin{aligned}\textsc{Opt}(t^{(i+1)}) &\geq \alpha \cdot \textsc{Opt}(t^{(i)}) + d(a^{(i+1)}_{f,\textsc{Opt}}, b^{(i)}_{l,\textsc{Opt}}) \\ &\overset{\triangle\text{-ineq}}{\geq} \alpha \cdot \textsc{Opt}(t^{(i)}) + d(a^{(i+1)}_{f,\textsc{Opt}}, p^{(i+1)}) \\ &\quad - d(b^{(i)}_{l,\textsc{Opt}}, O) - d(O, p^{(i+1)}). \end{aligned} \tag{5.12}$$

Since $S^{(i)}$ starts in O, ends in $p^{(i+1)}$ and serves $r^{(i)}_{l,\textsc{Opt}}$, we obtain

$$\begin{aligned} d(O, b^{(i)}_{l,\textsc{Opt}}) + d(b^{(i)}_{l,\textsc{Opt}}, O) &\leq |S^{(i)}| + d(p^{(i+1)}, O) \\ &\overset{\text{Lem 5.8}}{\leq} \textsc{Opt}(t^{(i)}) + d(p^{(i+1)}, O). \end{aligned} \tag{5.13}$$

Furthermore, because $\textsc{Opt}[t^{(i+1)}]$ serves $r^{(i)}_{l,\textsc{Lazy}}$ at $p^{(i+1)}$ before picking up $r^{(i+1)}_{f,\textsc{Opt}}$ at $a^{(i+1)}_{f,\textsc{Opt}}$, we have

$$\begin{aligned}\textsc{Opt}(t^{(i+1)}) &\geq d(O, p^{(i+1)}) + d(p^{(i+1)}, a^{(i+1)}_{f,\textsc{Opt}}) \\ &\overset{(5.8)}{>} d(O, p^{(i+1)}) + \alpha \cdot \textsc{Opt}(t^{(i+1)}) - \textsc{Opt}(t^{(i)}). \end{aligned} \tag{5.14}$$

Combining all of the above yields

$$\begin{aligned}\textsc{Opt}(t^{(i+1)}) &\overset{(5.12),(5.13)}{\geq} \alpha \cdot \textsc{Opt}(t^{(i)}) + d(a^{(i+1)}_{f,\textsc{Opt}}, p^{(i+1)}) \\ &\quad - \frac{\textsc{Opt}(t^{(i)}) + d(p^{(i+1)}, O)}{2} - d(O, p^{(i+1)}) \\ &\overset{(5.14)}{>} \alpha \cdot \textsc{Opt}(t^{(i)}) + d(a^{(i+1)}_{f,\textsc{Opt}}, p^{(i+1)}) - \frac{\textsc{Opt}(t^{(i)})}{2} \\ &\quad - \frac{3}{2}\big(\textsc{Opt}(t^{(i)}) - (\alpha - 1)\textsc{Opt}(t^{(i+1)})\big) \\ &= \left(\frac{3}{2}\alpha - \frac{3}{2}\right)\textsc{Opt}(t^{(i+1)}) - (2 - \alpha)\textsc{Opt}(t^{(i)}) + d(a^{(i+1)}_{f,\textsc{Opt}}, p^{(i+1)}) \\ &\overset{(5.8)}{>} \left(\frac{3}{2}\alpha - \frac{3}{2}\right)\textsc{Opt}(t^{(i+1)}) - (2 - \alpha)\textsc{Opt}(t^{(i)}) \\ &\quad + \alpha \cdot \textsc{Opt}(t^{(i+1)}) - \textsc{Opt}(t^{(i)}) \\ &= \left(\frac{5}{2}\alpha - \frac{3}{2}\right)\textsc{Opt}(t^{(i+1)}) - (3 - \alpha)\textsc{Opt}(t^{(i)}) \\ &\overset{\text{Obs 5.4a)}}{\geq} \left(\frac{5}{2}\alpha - \frac{3}{2}\right)\textsc{Opt}(t^{(i+1)}) - \left(\frac{3}{\alpha} - 1\right)\textsc{Opt}(t^{(i+1)}) \\ &= \left(\frac{5}{2}\alpha - \frac{1}{2} - \frac{3}{\alpha}\right)\textsc{Opt}(t^{(i+1)}) \\ &\geq \textsc{Opt}(t^{(i+1)}) \end{aligned}$$

where the last inequality holds if and only if $\alpha \geq \frac{1}{10} \cdot (3 + \sqrt{129}) \approx 1.436$. As this is a contradiction, we have that $\textsc{Opt}[t^{(i+1)}]$ serves all requests in $R^{(i)}$ before picking up $r^{(i+1)}_{f,\textsc{Opt}}$ in $a^{(i+1)}_{f,\textsc{Opt}}$. This completes the proof of the claim. □

Now that we have established the claim, we turn back to the proof of Lemma 5.13. Let $T \geq 0$ denote the time it takes $\textsc{Opt}[t^{(i+1)}]$ until it has served $r^{(i)}_{l,\textsc{Opt}}$, i.e., all requests from $R^{(i)}$. First, observe that

$$T \geq \textsc{Opt}(t^{(i)}). \tag{5.15}$$

By the claim, we have

$$\textsc{Opt}(t^{(i+1)}) \geq T + d(b^{(i)}_{l,\textsc{Opt}}, a^{(i+1)}_{f,\textsc{Opt}}) + |S(R^{(i+1)}, a^{(i+1)}_{f,\textsc{Opt}})|. \tag{5.16}$$

The algorithm $\textsc{Lazy}_\alpha$ finishes $R^{(i+1)}$ by time

$$\begin{aligned}
t^{(i+1)} + S^{(i+1)} &\overset{\text{Lem 5.12}}{=} \alpha \cdot \textsc{Opt}(t^{(i)}) + |S^{(i)}| + |S^{(i+1)}| \\
&\leq \alpha \cdot \textsc{Opt}(t^{(i)}) + |S^{(i)}| + d(p^{(i+1)}, a^{(i+1)}_{f,\textsc{Opt}}) + |S(R^{(i+1)}, a^{(i+1)}_{f,\textsc{Opt}})| \\
&\leq \alpha \cdot \textsc{Opt}(t^{(i)}) + |S^{(i)}| + d(p^{(i+1)}, b^{(i)}_{l,\textsc{Opt}}) + d(b^{(i)}_{l,\textsc{Opt}}, a^{(i+1)}_{f,\textsc{Opt}}) \\
&\qquad + |S(R^{(i+1)}, a^{(i+1)}_{f,\textsc{Opt}})| \\
&\overset{(5.16)}{\leq} \alpha \cdot \textsc{Opt}(t^{(i)}) + |S^{(i)}| + d(p^{(i+1)}, b^{(i)}_{l,\textsc{Opt}}) \\
&\qquad + \textsc{Opt}(t^{(i+1)}) - T.
\end{aligned} \tag{5.17}$$

As $S^{(i)}$ visits $b^{(i)}_{l,\textsc{Opt}}$ before $p^{(i+1)}$ and $\textsc{Opt}[t^{(i+1)}]$ visits $p^{(i+1)}$ before $b^{(i)}_{l,\textsc{Opt}}$,

$$\begin{aligned}
|S^{(i)}| + T &\geq \Big(d(O, b^{(i)}_{l,\textsc{Opt}}) + d(b^{(i)}_{l,\textsc{Opt}}, p^{(i+1)})\Big) \\
&\qquad + \Big(d(O, p^{(i+1)}) + d(p^{(i+1)}, b^{(i)}_{l,\textsc{Opt}})\Big) \\
&= 2 \cdot d(p^{(i+1)}, b^{(i)}_{l,\textsc{Opt}}) + d(O, b^{(i)}_{l,\textsc{Opt}}) + d(O, p^{(i+1)}) \\
&\geq 3 \cdot d(p^{(i+1)}, b^{(i)}_{l,\textsc{Opt}}).
\end{aligned} \tag{5.18}$$

Combined, we obtain that the algorithm finishes not later than

$$\begin{aligned}
t^{(i+1)} + |S^{(i+1)}| &\overset{(5.17)}{\leq} \alpha \cdot \textsc{Opt}(t^{(i)}) + |S^{(i)}| + d(p^{(i+1)}, b^{(i)}_{l,\textsc{Opt}}) + \textsc{Opt}(t^{(i+1)}) - T \\
&\overset{(5.18)}{\leq} \alpha \cdot \textsc{Opt}(t^{(i)}) + |S^{(i)}| + \frac{|S^{(i)}| + T}{3} + \textsc{Opt}(t^{(i+1)}) - T \\
&\overset{\text{Obs 5.4a)}}{\leq} 2 \cdot \textsc{Opt}(t^{(i+1)}) + \frac{4}{3}|S^{(i)}| - \frac{2}{3}T \\
&\overset{\text{Lem 5.8,(5.15)}}{\leq} 2 \cdot \textsc{Opt}(t^{(i+1)}) + \frac{2}{3}\textsc{Opt}(t^{(i)}) \\
&\overset{\text{Obs 5.4a)}}{\leq} \left(2 + \frac{2}{3\alpha}\right) \cdot \textsc{Opt}(t^{(i+1)}) \\
&\leq (1 + \alpha) \cdot \textsc{Opt}(t^{(i+1)})
\end{aligned}$$

where the last inequality holds if and only if $\alpha \geq \frac{1}{2} + \sqrt{11/12}$. □

The above results enable us to prove the upper bound in Theorem 5.1.

Proof of upper bound in Theorem 5.1. Our goal was to prove by induction that every schedule is α-good for $\alpha \geq \frac{1}{2} + \sqrt{11/12}$. In Observation 5.6, we have proven the base case. In the induction step, we have distinguished several cases. First, we have seen in Observation 5.7 that the induction step holds if the previous schedule was interrupted. Next, we have seen in Lemma 5.8 that the induction hypothesis implies $|S^{(i+1)}| \leq \textsc{Opt}(t^{(i+1)})$. If the previous schedule was not interrupted, we have first seen in Lemma 5.10 that the induction step holds if $\textsc{Opt}[t^{(i+1)}]$ loads $r^{(i+1)}_{f,\textsc{Opt}}$ before serving $r^{(i)}_{l,\textsc{Opt}}$. If $\textsc{Opt}[t^{(i+1)}]$ serves $r^{(i)}_{l,\textsc{Opt}}$ before loading $r^{(i+1)}_{f,\textsc{Opt}}$, the induction step holds by Lemma 5.11 and Lemma 5.13. □

5.2.3 Upper bound for Lazy on the half-line

In this subsection, we prove that Lazy achieves a better competitive ratio if the metric space considered is the half-line. In particular, we prove Theorem 5.2, i.e., that $\textsc{Lazy}_\alpha$ is $(1+\alpha)$-competitive for $\alpha = \frac{1+\sqrt{3}}{2} \approx 1.366$. Later, we will show that our bound is tight.

Note that all of the results in the previous subsection, except for Lemma 5.13, hold for all $\alpha \geq \frac{1+\sqrt{3}}{2} \approx 1.366$. Thus, it only remains to show a counterpart to Lemma 5.13 for $\alpha = \frac{1+\sqrt{3}}{2}$ on the half-line. Then the proof of Theorem 5.2 is analogous to the proof of Theorem 5.1, where we only have to replace Lemma 5.13 by the following result.

Lemma 5.15. *Let $\frac{1+\sqrt{3}}{2} \leq \alpha \leq 2$, and let $M = \mathbb{R}_{\geq 0}$. Assume that the i-th schedule is α-good and was not interrupted, and that $\textsc{Opt}[t^{(i+1)}]$ serves $r^{(i)}_{l,\textsc{Lazy}}$ before collecting $r^{(i+1)}_{f,\textsc{Opt}}$. If (5.8) holds, then $t^{(i+1)} + |S^{(i+1)}| \leq (1+\alpha) \cdot \textsc{Opt}(t^{(i+1)})$.*

Proof. First, observe that, in Lemma 5.13, we have the same assumptions except that we worked on general metric spaces. Therefore, all the inequalities shown in Lemma 5.13 hold in this setting, too, so that we can use them for our proof. Next, note that on the half-line, we have for any $x, y \in M$

$$d(x,y) \leq \max\{d(x,O), d(y,O)\}. \tag{5.19}$$

We show that this implies that a similar claim as in Lemma 5.13 holds.

Claim 5.16. *$\textsc{Opt}[t^{(i+1)}]$ serves all requests in $R^{(i)}$ before picking up $r^{(i+1)}_{f,\textsc{Opt}}$ in $a^{(i+1)}_{f,\textsc{Opt}}$.*

Proof. To prove the claim, assume otherwise, i.e., that $\textsc{Opt}[t^{(i+1)}]$ serves $r^{(i)}_{l,\textsc{Opt}}$ after collecting $r^{(i+1)}_{f,\textsc{Opt}}$. The request $r^{(i+1)}_{f,\textsc{Opt}}$ is released after schedule $S^{(i)}$ is started, i.e., after time $\alpha \cdot \textsc{Opt}(t^{(i)})$. Thus,

$$\begin{aligned}
\textsc{Opt}(t^{(i+1)}) &\geq \alpha \cdot \textsc{Opt}(t^{(i)}) + d(a^{(i+1)}_{f,\textsc{Opt}}, b^{(i)}_{l,\textsc{Opt}}) \\
&\geq \alpha \cdot \textsc{Opt}(t^{(i)}) + d(a^{(i+1)}_{f,\textsc{Opt}}, p^{(i+1)}) - d(b^{(i)}_{l,\textsc{Opt}}, p^{(i+1)}) \\
&\overset{(5.19)}{\geq} \alpha \cdot \textsc{Opt}(t^{(i)}) + d(a^{(i+1)}_{f,\textsc{Opt}}, p^{(i+1)}) \\
&\qquad - \max\{d(b^{(i)}_{l,\textsc{Opt}}, O), d(O, p^{(i+1)})\}.
\end{aligned} \tag{5.20}$$

Combining the above with the results from the proof of Lemma 5.13 yields

$$\begin{aligned}
\textsc{Opt}(t^{(i+1)}) &\overset{(5.20),(5.13)}{\geq} \alpha \cdot \textsc{Opt}(t^{(i)}) + d(a^{(i+1)}_{f,\textsc{Opt}}, p^{(i+1)}) \\
&\quad - \max\Big\{\frac{\textsc{Opt}(t^{(i)}) + d(p^{(i+1)}, O)}{2}, d(O, p^{(i+1)})\Big\} \\
&\overset{(5.14)}{>} \alpha \cdot \textsc{Opt}(t^{(i)}) + d(a^{(i+1)}_{f,\textsc{Opt}}, p^{(i+1)}) \\
&\quad - \max\Big\{\frac{2\textsc{Opt}(t^{(i)}) - (\alpha-1)\textsc{Opt}(t^{(i+1)})}{2}, \\
&\qquad\qquad \textsc{Opt}(t^{(i)}) - (\alpha-1)\textsc{Opt}(t^{(i+1)})\Big\} \\
&= \frac{\alpha-1}{2}\textsc{Opt}(t^{(i+1)}) + (\alpha-1)\textsc{Opt}(t^{(i)}) + d(a^{(i+1)}_{f,\textsc{Opt}}, p^{(i+1)}) \\
&\overset{(5.8)}{>} \Big(\frac{\alpha-1}{2} + \alpha\Big)\textsc{Opt}(t^{(i+1)}) + (\alpha-2)\textsc{Opt}(t^{(i)}) \\
&\overset{\text{Obs 5.4a)}}{\geq} \Big(\frac{\alpha-1}{2} + \alpha + \frac{\alpha-2}{\alpha}\Big)\textsc{Opt}(t^{(i+1)}) \\
&\geq \textsc{Opt}(t^{(i+1)})
\end{aligned}$$

where the last inequality holds for all $\alpha \geq \frac{4}{3}$. As this is a contradiction, we have that $\textsc{Opt}[t^{(i+1)}]$ serves all requests in $R^{(i)}$ before picking up $r^{(i+1)}_{f,\textsc{Opt}}$ in $a^{(i+1)}_{f,\textsc{Opt}}$. This completes the proof of the claim. □

Now that we have established the claim, we turn back to the proof of Lemma 5.15. Let $T \geq 0$ denote the time it takes $\textsc{Opt}[t^{(i+1)}]$ until it has served $r^{(i)}_{l,\textsc{Opt}}$, i.e., all requests from $R^{(i)}$. First, observe that

$$T \geq \textsc{Opt}(t^{(i)}). \tag{5.21}$$

If $p^{(i+1)} \geq b^{(i)}_{l,\textsc{Opt}}$, as $\textsc{Opt}[t^{(i+1)}]$ visits $p^{(i+1)}$ before $b^{(i)}_{l,\textsc{Opt}}$, we have

$$T \geq d(O, p^{(i+1)}) + d(p^{(i+1)}, b^{(i)}_{l,\textsc{Opt}}) \overset{(5.19)}{\geq} 2 \cdot d(p^{(i+1)}, b^{(i)}_{l,\textsc{Opt}}).$$

Otherwise, if $p^{(i+1)} < b^{(i)}_{l,\textsc{Opt}}$, as $S^{(i)}$ visits $b^{(i)}_{l,\textsc{Opt}}$ before $p^{(i+1)}$, we have

$$T \overset{(5.21)}{\geq} \textsc{Opt}(t^{(i)}) \geq |S^{(i)}| \geq d(O, b^{(i)}_{l,\textsc{Opt}}) + d(b^{(i)}_{l,\textsc{Opt}}, p^{(i+1)}) \overset{(5.19)}{\geq} 2 \cdot d(b^{(i)}_{l,\textsc{Opt}}, p^{(i+1)}).$$

Thus, in either case, we have

$$d(b^{(i)}_{l,\textsc{Opt}}, p^{(i+1)}) \leq \frac{T}{2}. \tag{5.22}$$

Combined, we obtain that the algorithm finishes not later than

$$t^{(i+1)} + |S^{(i+1)}| \overset{(5.17)}{\leq} \alpha \cdot \textsc{Opt}(t^{(i)}) + |S^{(i)}| + d(p^{(i+1)}, b^{(i)}_{l,\textsc{Opt}}) + \textsc{Opt}(t^{(i+1)}) - T$$

$$\overset{(5.22)}{\leq} \alpha \cdot \textsc{Opt}(t^{(i)}) + |S^{(i)}| + \frac{T}{2} + \textsc{Opt}(t^{(i+1)}) - T$$
$$\overset{\text{Obs 5.4a)}}{\leq} 2 \cdot \textsc{Opt}(t^{(i+1)}) + |S^{(i)}| - \frac{T}{2}$$
$$\overset{\text{Lem 5.8, (5.21)}}{\leq} 2 \cdot \textsc{Opt}(t^{(i+1)}) + \frac{1}{2}\textsc{Opt}(t^{(i)})$$
$$\overset{\text{Obs 5.4a)}}{\leq} \left(2 + \frac{1}{2\alpha}\right) \cdot \textsc{Opt}(t^{(i+1)}) \leq (1+\alpha) \cdot \textsc{Opt}(t^{(i+1)})$$

where the last inequality holds if and only if $\alpha \geq \frac{1+\sqrt{3}}{2} \approx 1.366$. □

To complete our upper bounds, we comment on the performance of the Lazy algorithm in the preemptive vesion of the open online dial-a-ride problem. Recall that, in the preemptive setting, the server is allowed to unload requests anywhere and pick them up later again. In this version, prior to our work, the best known upper bound on general metric spaces was 2.696 [21] and the best known upper bound on the line and the half-line was 2.41 [25]. Clearly, every non-preemptive algorithm can also be applied in the preemptive setting, however, its competitive ratio may be degraded since the optimum might use preemption. Our algorithm Lazy repeatedly executes optimal solutions for subsets of requests and can be turned preemptive by using preemptive solutions. With this change, our analysis of Lazy still carries through in the preemptive case and improves the state of the art for general metric spaces and the half-line.

Corollary 5.17. *The competitive ratio of the open preemptive online dial-a-ride problem with any capacity $c \in \mathbb{N} \cup \{\infty\}$ is upper bounded by*

a) $\frac{3}{2} + \sqrt{\frac{11}{12}} \approx 2.457$ *and this bound is achieved by* $\textsc{Lazy}\left(\frac{1}{2} + \sqrt{\frac{11}{12}}\right)$,

b) $1 + \frac{1+\sqrt{3}}{2} \approx 2.366$ *on the half-line and this bound is achieved by* $\textsc{Lazy}(\frac{1+\sqrt{3}}{2})$.

5.2.4 Factor-revealing approach for the half-line

We illustrate how we made use of the factor revealing approach described in Section 5.1.2 for the dial-a-ride problem on the half-line. Consider the following variables (recall that $k \in \mathbb{N}$ is the number of schedules started by $\textsc{Lazy}_\alpha$).

- $t_1 = t^{(k-1)}$, the start time of the second to last schedule
- $t_2 = r^{(k)}$, the start time of the last schedule
- $s_1 = |S^{(k-1)}|$, the duration of the second to last schedule
- $s_2 = |S^{(k)}|$, the duration of the last schedule
- $\textsc{Opt}_1 = \textsc{Opt}(t^{(k-1)})$, duration of the optimal tour serving requests released until $t^{(k-1)}$

- $\textsc{Opt}_2 = \textsc{Opt}(t^{(k)})$, duration of the optimal tour
- $p_1 = p^{(k)}$, the position where $\textsc{Lazy}_\alpha$ ends the second to last schedule
- $p_2 = a^{(k)}_{f,\textsc{Opt}}$, the position of the first request in $R^{(k)}$ picked up first by the optimal tour
- $s_2^a = |S(R^{(k)}, a^{(k)}_{f,\textsc{Opt}})|$, duration of the schedule serving $R^{(k)}$ starting in p_2
- $d = d(p^{(k)}, a^{(k)}_{f,\textsc{Opt}})$, the distance between p_1 and p_2

With these variables $x = \big(t_1, t_2, s_1, s_2, \textsc{Opt}_1, \textsc{Opt}_2, p_1, p_2, s_2^a, d\big)$, we can create the following valid optimization problem.

$$
\begin{aligned}
\max \quad & t_2 + s_2 \\
\text{s.t.} \quad & \textsc{Opt}_2 = 1 && (5.23)\\
& d = |p_1 - p_2| && (5.24)\\
& t_2 = \max\{t_1 + s_1, \alpha\textsc{Opt}_2\} && (5.25)\\
& t_1 \geq \alpha\textsc{Opt}_1 && (5.26)\\
& \textsc{Opt}_1 \geq p_1 && (5.27)\\
& s_2 \leq d + s_2^a && (5.28)\\
& \textsc{Opt}_2 \geq t_1 + s_2^a && (5.29)\\
& t_1 + s_1 \leq (1+\alpha)\textsc{Opt}_1 && (5.30)\\
& \textsc{Opt}_2 \geq p_1 + d \quad \text{or} \quad \textsc{Opt}_2 \geq t_1 + d && (5.31)\\
& d \geq \alpha\textsc{Opt}_2 - \textsc{Opt}_1 \quad \text{or} \quad s_1 - p_1 \leq 2(\textsc{Opt}_2 - p_2) && (5.32)\\
& x \geq 0 && (5.33)
\end{aligned}
$$

The notation in (5.31) and (5.32) means that at least one of the two inequalities has to be satisfied. Note that all constraints except for (5.24), (5.25), (5.31), and (5.32) are linear. In order to make the problem linear, we introduce a binary variable for each of these constraints $b_1, \ldots, b_4 \in \{0, 1\}$, let $M > 0$ be a large enough constant and we proceed as follows: For equality (5.24), note that we have $|p_1 - p_2| = \max\{p_1 - p_2, p_2 - p_1\}$ and we replace the constraint by the inequalities

$$
\begin{aligned}
d &\geq p_1 - p_2,\\
d &\geq p_2 - p_1,\\
d &\leq p_1 - p_2 + b_1 \cdot M,\\
d &\leq p_2 - p_1 + (1 - b_1) \cdot M.
\end{aligned}
$$

In the above, $b_1 = 0$ indicates the case that $\max\{p_1 - p_2, p_2 - p_1\} = p_1 - p_2$. In equality (5.25), we handle the maximum analogously, using the binary variable b_2. Constraint (5.31) can be replaced by the inequalities

$$
\begin{aligned}
\textsc{Opt}_2 &\geq p_1 + d - b_3 \cdot M,\\
\textsc{Opt}_2 &\geq t_1 + d - (1 - b_3) \cdot M,
\end{aligned}
$$

and (5.32) is again handled similarly, using the binary variable b_4. This results in a *mixed-integer linear program (MILP)*, that is, a linear program with certain variables required to take integer values. Using a MILP solver, one finds that the following is an optimal solution.

$$\begin{aligned}&\big(t_1, t_2, s_1, s_2, \textsc{Opt}_1, \textsc{Opt}_2, p_1, p_2, s_2^a, d, b_1, b_2, b_3, b_4\big)\\ &= \Big(1, \frac{\alpha+1}{\alpha}, \frac{1}{\alpha}, 2-\alpha, \frac{1}{\alpha}, 1, 0, 2-\alpha, 0, 2-\alpha, 0, 1, 1, 1\Big)\end{aligned}$$

The objective function value of this solution is $\max\{3+\frac{1}{\alpha}-\alpha, 1+\alpha\}$. This expression is minimized for $\alpha = \frac{1+\sqrt{3}}{2} > 1.366$.

5.3 Lower bounds on the competitive ratio of Lazy

In this section, we provide lower bounds on the competitive ratio of $\textsc{Lazy}_\alpha$. More precisely, we show the following.

Theorem 5.18. *For all $\alpha \geq 0$, the algorithm $\textsc{Lazy}_\alpha$ has a competitive ratio of at least $\frac{3}{2} + \sqrt{11/12} \approx 2.457$ for open online dial-a-ride on general metric spaces for every capacity $c \in \mathbb{N} \cup \{\infty\}$.*

Note that this implies that our parameter choice in Section 5.2.2 is optimal, i.e., this completes the proof of Theorem 5.1.

Our proof will consist of a lower bound construction for $\alpha \geq 1$ and a separate construction for $\alpha < 1$. In the following constructions, we let the metric space (M, d) be the real line, i.e., $M = \mathbb{R}$, $O = 0$, and $d(a, b) = |a-b|$. Note that lower bounds on the line trivially carry over to general metric spaces. Moreover, our constructions work for any given server capacity $c \in \mathbb{N} \cup \{\infty\}$ because, in our construction, a larger server capacity does neither change the behavior of the optimum solution nor the behavior of Lazy.

First, observe that, for any $\alpha \geq 0$, the competitive ratio of $\textsc{Lazy}_\alpha$ is lower bounded by $1 + \alpha$. This can be easily seen by observing the request sequence consisting of the single request $r_1 = (1, 1; 1)$. In this case, the offline optimum has completed the sequence by time 1, whereas $\textsc{Lazy}_\alpha$ waits in O until time $\max(\alpha, 1)$ and then moves to 1 and serves r_1 not earlier than $1 + \alpha$.

Observation 5.19. *For any $\alpha \geq 0$, $\textsc{Lazy}_\alpha$ has a competitive ratio of at least $1 + \alpha$ for the open online dial-a-ride problem on the line for any capacity $c \in \mathbb{N} \cup \{\infty\}$.*

This gives the desired lower bound of $(3/2 + \sqrt{11/12})$ for all $\alpha \geq 1/2 + \sqrt{11/12}$. Now, we give a lower bound construction for the case $1 \leq \alpha < 1/2 + \sqrt{11/12}$.

Proposition 5.20. *For $\alpha \geq 1$, $\textsc{Lazy}_\alpha$ has a competitive ratio of at least $2 + \frac{2}{3\alpha}$ for the open online dial-a-ride problem on the line.*

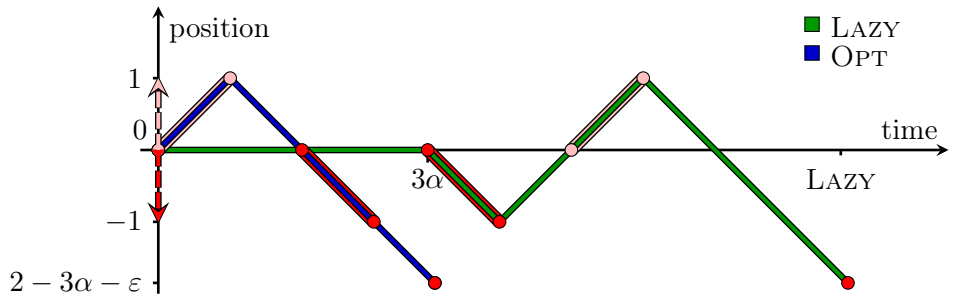

Figure 5.2: Instance of the open online dial-a-ride problem on the line where LAZY_α has a competitive ratio of at least $2 + \frac{2}{3\alpha}$ for all $\alpha \geq 1$.

Proof. First, observe that, for $\alpha \geq 1/2 + \sqrt{11/12}$, we have $2 + \frac{2}{3\alpha} \leq 1 + \alpha$ so that the assertion follows from Lemma 5.19. Therefore, let $\alpha \in [1, 1/2 + \sqrt{11/12})$ and let $\varepsilon > 0$ be small enough such that $3\alpha + 2 > 3\alpha^2 + \alpha\varepsilon$. Note that this is possible because $3\alpha + 2 > 3\alpha^2$ for $\alpha \in [1, (1/2) + \sqrt{11/12})$.

We construct an instance of the open online dial-a-ride problem, where the competitive ratio of LAZY_α converges to $2 + \frac{2}{3\alpha}$ for $\varepsilon \to 0$ (cf. Figure 5.2). We define the instance by giving the requests

$$r_1 = (0, 1; 0),\ r_2 = (0, -1; 0), \text{ and } r_3 = (2 - 3\alpha - \varepsilon, 2 - 3\alpha - \varepsilon; 3\alpha + \varepsilon).$$

One solution is to first serve r_1 and then r_2. This is possible in 3 time units and, after this, the server is in position -1. Then, the server can reach point $2 - 3\alpha - \varepsilon$ by time $3 + (-2 + 3\alpha + \varepsilon - 1) = 3\alpha + \varepsilon$. At this point in time, r_3 is released and can immediately be served. Thus, we have

$$\text{OPT} := \text{OPT}(3\alpha + \varepsilon) = 3\alpha + \varepsilon.$$

We now analyze what LAZY_α does on this request sequence. Note that we have $\text{OPT}(0) = 3$. Thus, the server waits in O until time 3α. Since no new request arrives until this time, the server starts an optimal schedule serving r_1 and r_2. Without loss of generality, we can assume that LAZY_α starts by serving r_2, because the starting positions and destinations of r_1 and r_2 are symmetrical. At time $3\alpha + \varepsilon$, request r_3 is released, and the server has currently loaded r_2. Delivering r_2 and returning to the origin takes the server until time $3\alpha + 2$. By definition of α and ε, we have

$$3\alpha + 2 > 3\alpha^2 + \alpha\varepsilon = \alpha\text{OPT}. \tag{5.34}$$

This implies that the server is not interrupted in its current schedule. It continues serving r_2 and then serves r_1 at time $3\alpha + 3$. Together with (5.34), it follows that, after serving r_1, the server immediately starts serving the remaining request r_3. Moving from 1 to $2 - 3\alpha - \varepsilon$ takes $3\alpha - 1 + \varepsilon$ time units, i.e., the server serves r_3 at time $(3\alpha + 3) + (3\alpha - 1 + \varepsilon) = 6\alpha + 2 + \varepsilon$. Thus, the competitive ratio is at least

$$\frac{6\alpha + 2 + \varepsilon}{\text{OPT}} = \frac{6\alpha + 2 + \varepsilon}{3\alpha + \varepsilon} = 2 + \frac{2 - \varepsilon}{3\alpha + \varepsilon}.$$

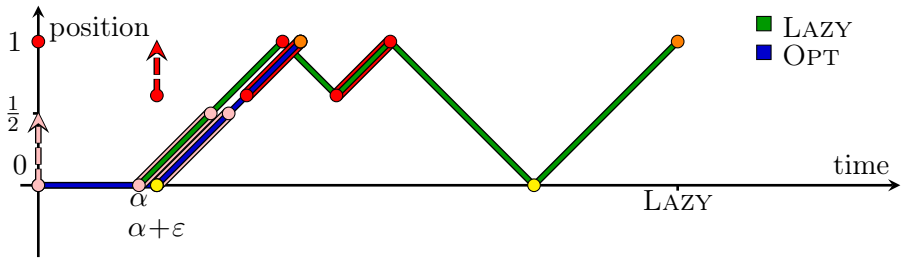

Figure 5.3: Instance of the open online dial-a-ride problem on the line where LAZY_α has a competitive ratio of at least $1 + \frac{3}{\alpha+1}$ for all $\alpha \in [0, 1)$.

The statement follows by taking the limit $\varepsilon \to 0$. □

Next, we give a lower bound construction for $\alpha < 1$.

Proposition 5.21. *For $\alpha \in [0, 1)$, the algorithm LAZY_α has a competitive ratio of at least $1 + \frac{3}{\alpha+1}$ for the open dial-a-ride problem on the half-line.*

Proof. Let $\alpha \in [0, 1)$ and $\varepsilon \in (0, \min\{\frac{\alpha}{2}, \frac{1}{\alpha} - \alpha, 1 - \alpha\})$. We construct an instance of the open dial-a-ride problem, where the competitive ratio of LAZY_α converges to $1 + \frac{3}{\alpha+1}$ for $\varepsilon \to 0$ (cf. Figure 5.3). We define the instance by giving the requests

$$r_1 = \left(0, \frac{1}{2}; 0\right),\ r_2 = (1, 1; 0),\ r_3 = (0, 0; \alpha + \varepsilon),$$
$$r_4 = \left(\frac{1}{2} + \varepsilon, 1; \alpha + \varepsilon\right), \text{ and } r_5 = (1, 1; \alpha + 1 + \varepsilon).$$

One solution is to first wait in O until time $\alpha + \varepsilon$ and serve r_3. Then, the server can pick up and deliver r_1, move to $\frac{1}{2} + \varepsilon$, and immediately do the same with r_4. This can be done by time $\alpha + 1 + \varepsilon$. Now, the server is in position 1 and can thus immediately serve r_2 and r_5. It finishes serving all request in time $\alpha + 1 + \varepsilon$. Since the last request is released at time $\alpha + 1 + \varepsilon$, we have

$$\text{OPT} := \text{OPT}(\alpha + 1 + \varepsilon) = \alpha + 1 + \varepsilon. \tag{5.35}$$

We now analyze what LAZY_α does on this request sequence. Note that we have $\text{OPT}(0) = 1$. Hence, the server waits in O until time α. Since no new requests arrive until this time, the server starts an optimal schedule serving r_1 and r_2, i.e., picks up r_1 and starts moving towards position $\frac{1}{2}$. At time $\alpha + \varepsilon$, r_3 and r_4 are released. We have $\text{OPT}(\alpha + \varepsilon) = \alpha + 1 + \varepsilon$. Serving the loaded request r_1 and returning to 0 would take the server until time

$$\alpha + 1 \overset{\varepsilon < \frac{1}{\alpha} - \alpha}{>} \alpha + (\alpha^2 + \alpha\varepsilon) = \alpha(\alpha + 1 + \varepsilon) = \alpha \cdot \text{OPT}(\alpha + \varepsilon). \tag{5.36}$$

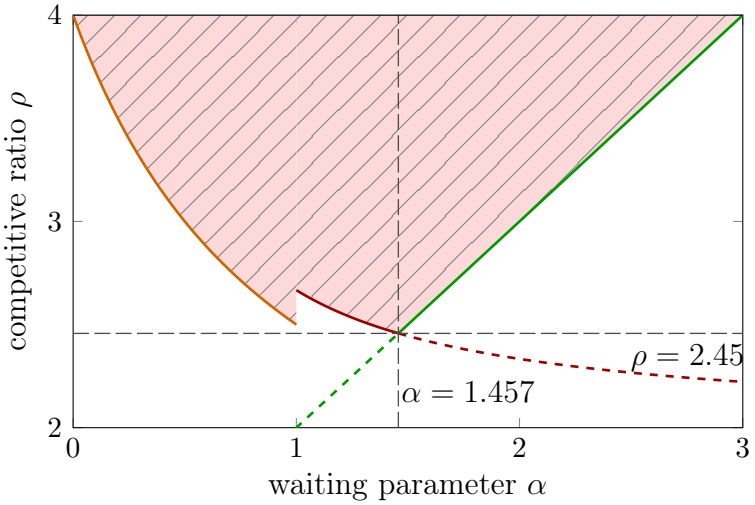

Figure 5.4: Lower bounds on the competitive ratio of LAZY_α depending on α. The lower bound of Observation 5.19 is depicted in green, the lower bound of Proposition 5.20 in red, and the lower bound of Proposition 5.21 in orange.

Thus, the server keeps following its tour, i.e., it serves r_1 and then r_2 at time $\alpha + 1$. By (5.36) and since $\text{OPT}(\alpha + 1) = \text{OPT}(\alpha + \varepsilon)$, the server immediately starts serving r_3 and r_4. The shortest tour is serving r_4 first, i.e., the server starts moving towards $\frac{1}{2} + \varepsilon$. At time $\alpha + 1 + \varepsilon$, request r_5 is released. Since

$$\alpha + 1 + \varepsilon \overset{(5.35)}{=} \text{OPT}(\alpha + 1 + \varepsilon) > \alpha \cdot \text{OPT}(\alpha + 1 + \varepsilon),$$

the server keeps following its schedule. It completes that schedule in position O at time $(\alpha + 1) + (1 - 2\varepsilon) + 1 = 3 + \alpha - 2\varepsilon$. Then, the server starts its last tour in order to serve r_5. It moves to 1 and finishes serving the last request at time $4+\alpha-2\varepsilon$. Thus, the competitive ratio is

$$\frac{4 + \alpha - 2\varepsilon}{\text{OPT}} = \frac{4 + \alpha - 2\varepsilon}{\alpha + 1 + \varepsilon} = 1 + \frac{3 - 3\varepsilon}{\alpha + 1 + \varepsilon}.$$

The statement follows by taking the limit $\varepsilon \to 0$. □

Now, we combine our results for the lower bounds (cf. Figure 5.4). Combining Observation 5.19 and Proposition 5.20, we obtain that, for $\alpha \geq 1$, LAZY_α has a competitive ratio of at least $\max\{1 + \alpha, 2 + 2/(3\alpha)\}$. Minimizing over $\alpha \geq 1$ yields a competitive ratio of at least $3/2 + \sqrt{11/12} > 2.457$. For the case $\alpha < 1$, we have seen in Proposition 5.21 that the algorithm LAZY_α has a competitive ratio of at least $1 + 3/(\alpha + 1) > 5/2$. Together, this proves the lower bound in Theorem 5.18.

5.3.1 Lower bound on the half-line

We go on to show lower bounds on the performance of LAZY on the half-line. More precisely, we prove the following.

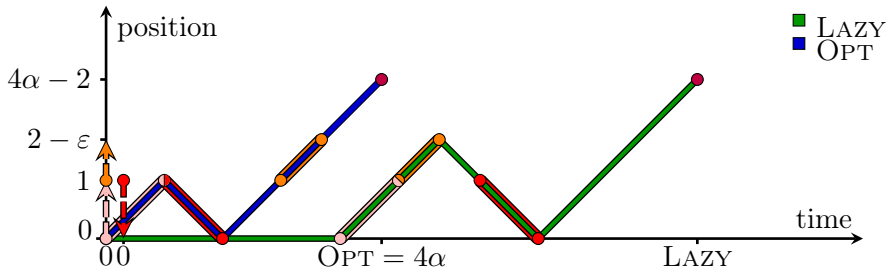

Figure 5.5: Instance of the open online dial-a-ride problem on the half-line where LAZY_α has a competitive ratio of at least $2 + \frac{1}{2\alpha}$ for all $\alpha \in [1, 1.366)$.

Theorem 5.22. *For all $\alpha \geq 0$, the competitive ratio of LAZY_α for open online dial a ride on the half-line for every capacity $c \in \mathbb{N} \cup \{\infty\}$ is at least $\frac{3+\sqrt{3}}{2} \approx 2.366$.*

This shows that our parameter choice in Section 5.2.3 is optimal, i.e., together with the upper bounds from that section, this result implies Theorem 5.2.

Note that the lower bound constructions in the proofs of Observation 5.19 and Proposition 5.21 are instances on the half-line. Thus, the lower bounds also hold on the half-line. Combining these two results gives that, for $\alpha \in [0, 1) \cup [\frac{1+\sqrt{3}}{2}, \infty)$, the competitive ratio of LAZY_α is at least $\frac{3+\sqrt{3}}{2} \approx 2.366$.

The next result closes the gap between $\alpha < 1$ and $\alpha \geq 1.366$.

Proposition 5.23. *For every $\alpha \in [1, 1.366)$, the algorithm LAZY_α has a competitive ratio of at least $2 + \frac{1}{2\alpha}$ for the open online dial-a-ride problem on the half-line for every capacity $c \in \mathbb{N} \cup \{\infty\}$.*

Proof. Let $\alpha \in [1, 1.366)$ and let $\varepsilon > 0$ be sufficiently small. We define an instance (cf. Figure 5.5) by giving the request sequence

$$r_1 = (0, 1; 0),\ r_2 = (1, 0; 0),\ r_3 = (1, 2 - \varepsilon; 0), \text{ and } r_4 = (4\alpha - 2, 4\alpha - 2; 4\alpha).$$

The offline optimum delivers the requests in the order (r_1, r_2, r_3, r_4) with no waiting times. This takes 4α time units.

On the other hand, because $\text{OPT}(0) = 4 - 2\varepsilon$, LAZY_α waits in the origin until time $\alpha(4 - 2\varepsilon)$ and starts serving requests r_1, r_2, r_3 in the order (r_1, r_3, r_2). At time 4α, request r_4 is released. At this time, serving the loaded request r_1 and returning to the origin takes time

$$\alpha(4 - 2\varepsilon) + 2 = 4\alpha + 2 - 2\alpha\varepsilon \overset{\alpha \in [1, 1.366), \varepsilon \ll 1}{>} 4\alpha^2 = \alpha \cdot \text{OPT}(4\alpha).$$

Thus, LAZY_α continues its schedule and afterwards serves r_4. Overall, this takes time $\alpha(4 - 2\varepsilon) + (4 - 2\varepsilon) + 4\alpha - 2 = 8\alpha + 2 - (2\alpha + 2)\varepsilon$. We obtain that the competitive ratio of LAZY_α is at least

$$\lim_{\varepsilon \to 0} \frac{8\alpha + 2 - (2\alpha + 2)\varepsilon}{4\alpha} = 2 + \frac{1}{2\alpha}.$$

□

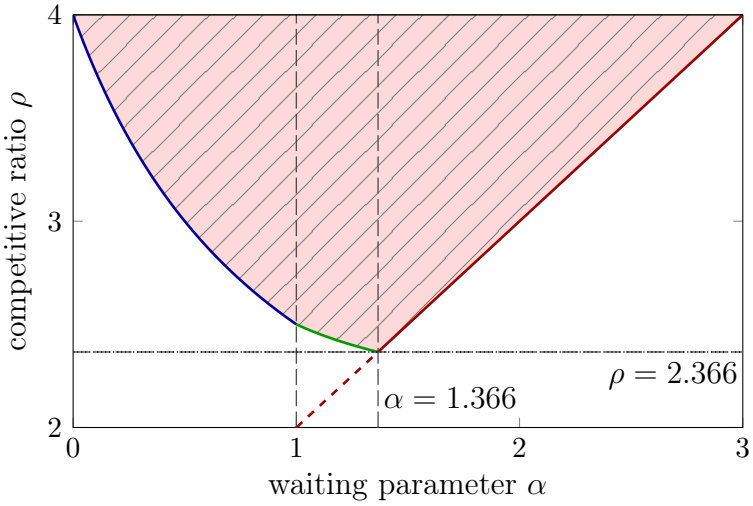

Figure 5.6: Lower bounds on the competitive ratio of LAZY_α depending on α. The lower bound of Observation 5.19 is depicted in red, the lower bound of Proposition 5.21 in blue, and the lower bound of Proposition 5.23 in green.

By combining the results from Observation 5.19, Proposition 5.21, and Proposition 5.23 (cf. Figure 5.6), we obtain Theorem 5.22, i.e., that the competitive ratio of LAZY on the half-line cannot be better than $\frac{3+\sqrt{3}}{2} \approx 2.366$.

5.4 Outlook

In this chapter, we presented an improved algorithm, LAZY, for the open online dial-a-ride problem and proved that it achieves a competitive ratio of 2.457 on general metric spaces and 2.366 on the half-line. The best known lower bounds for this problem are 2.05 on general metric spaces [23] and 1.9 on the half-line [90]. Determining the exact competitive ratio of the open online dial-a-ride problem thus remains an open question.

Importantly, our results show that the natural candidate REPLAN is not optimal, as a lower bound of 2.5 is known for this algorithm [7]. Furthermore, the same lower bound of 2.5 is known for schedule-based algorithms, i.e., algorithms that divide execution into subschedules such that, when a subschedule starts, it serves all requests revealed but not yet served at that time, and these subschedules are never interrupted. Our results imply that schedule-based algorithms cannot be optimal either. Moreover, we have established lower bounds for LAZY, showing that our analysis is tight.

Thus, to obtain an algorithm with a better competitive ratio, a new approach is needed. For example, instead of simply executing a shortest schedule for the unserved requests, one might want to take into account other aspects when planning.

For instance, to anticipate that schedules can be interrupted, one could minimize the sum of completion times rather than the overall completion time. Another objective could be to choose schedules that end at a position closer to the origin.

Last, we remark that there are numerous variants of the dial-a-ride problem studied in the literature. This includes settings where the request sequence has to fulfill some reasonable additional properties [26, 66, 86], where the server is presented with additional [2, 7] or less [89] information, where the server has some additional abilities [27, 73], where the server has to handle requests in a given order [65, 73], where the requests have to be served before some deadline [112], where the distances between points are not necessarily symmetric [8], or where we consider different objectives than the completion time [9, 19, 20, 66, 71, 72, 84, 85, 86]. Other examples include the study of randomized algorithms [83], or other metric spaces, such as a circle [74]. We believe that applying LAZY, or an adaptation of it, to some of these variants could serve as an interesting starting point for future research.

Chapter 6

Conclusion

We studied approximation algorithms for the traveling salesperson problem under additional constraints, including incomplete information and restrictions on the running time.

The central focus of our work was the online graph exploration problem, in which an agent is tasked to find a TSP tour in a graph that is learned gradually over time. The key open question for this problem is whether a constant-competitive algorithm exists [75]. We made progress on this problem in the following sense: We proved that a constant-competitive algorithm exists on minor-free graphs (Theorem 2.2). Prior to our work, the bounded-genus graphs [92] were the largest class known to admit a constant competitive ratio. This indicates that difficult instance for online graph exploration could be expanders or graphs fulfilling some density properties. Investigating these types of graphs could serve as a starting point for developing improved lower bound constructions.

In addition, we proved that the competitive ratio of the problem is at least 4 (Theorem 2.3), improving on the previously best known lower bound of 10/3 [23]. However, since our construction is planar, it cannot be used to obtain a non-constant lower bound, as the competitive ratio for online graph exploration on planar graphs is at most 16 [75]. As discussed in Section 2.6, we believe that existing techniques for proving lower bounds [23, 44] have been pushed to their limits, and that breaking the barrier of 4 will require new approaches.

Next, we considered the exploration problem using a team of k agents. Finding an optimal strategy for this is a wide-open problem, even on graphs as simple as unweighted trees. Consequently, most research on collaborative exploration has focused on this special case, which we also addressed in our work. We gave a slightly improved bound on the competitive ratio of the algorithm Yo* [98]. The most difficult instances for collaborative tree exploration seem to be exploration of trees of large depth $D \geq n^{1-o(1)}$ using a small team of $k \leq n^{o(1)}$ agents. When it comes to collaborative exploration of general weighted graphs, essentially nothing is known yet, and the problem remains wide open. Building on our findings in Chapter 2, an interesting direction for future research could be the following: Can better approximation guarantees be achieved for collaborative exploration on minor-

free graphs compared to general graphs?

We also investigated the open online dial-a-ride problem. To this end, we introduced the algorithm Lazy and proved a competitive ratio of 2.457 on general metric spaces (Theorem 4.1) and 2.366 on the half-line (Theorem 5.2). Prior to our work, the best known upper bound in both metric spaces was 2.696 [21]. The best known lower bounds are 2.05 [24] on general metric spaces and 1.9 on the half-line [90]. Closing these gaps remains an open problem. Note that the problem assumes the underlying metric space to be continuous, and it is typically considered on $\mathbb{R}^n$ or $\mathbb{R}^n_{\geq 0}$. While the special case of online TSP has been studied on a circle [74], we believe it would be highly interesting to investigate dial-a-ride on other types of metric spaces. For instance, any weighted graph can be transformed into a continuous metric space by allowing the agent (and possibly the requests) to occupy any point along an edge. Investigating which graph structures lead to higher or lower competitive ratios could provide an interesting direction for further research.

While the problems discussed above fall in the field of online optimization, we also investigated a different type of restriction for the TSP, where full information is available but algorithms are required to run in polynomial time. Specifically, we studied the k-colored Euclidean TSP under this perspective. For the cases $k = 1$ [5] and $k = 2$ [45], it is known that an EPTAS exists. However, we have seen that the case $k \geq 3$ is substantially more challenging, as the Patching Lemma, which is one of the key ingredients for Arora's algorithm, does not generalize to more than two non-crossing tours. In our approach, we circumvented this issue by giving a conditional patching scheme for three tours and an alternative approach for the case where patching is not possible, based on a weighted solution with two colors. The most pressing open question for k-ETSP remains whether a PTAS exists for $k \geq 3$.

We note that a very intriguing question for the TSP, which lies beyond the scope of this work, is the following: What is the best possible approximation ratio achievable by a polynomial-time algorithm for the classical TSP? Here, we specifically refer to the *metric TSP*, where the problem is defined on a weighted graph and the agent is allowed to visit vertices multiple times. In contrast, if the agent is required to visit each vertex exactly once, there is no polynomial-time constant-factor approximation algorithm, since this case reduces to the Hamiltonian cycle problem, which is NP-complete. The best known upper bound for the approximation factor for metric TSP is $1.5 - 10^{-36}$ [76, 77], which slightly improves on the bound of 1.5 achieved by Christofide's algorithm [36]. The best known lower bound is 123/122 [79].

We have studied two different types of restrictions, on information and on computational complexity, and we believe it would be an interesting direction to impose both types of restrictions simultaneously. While online optimization typically does not focus on the running time of algorithms, incorporating computational complexity considerations could yield valuable insights. For example, all known algorithms for the online graph exploration problem can be implemented to run in polynomial time, meaning that the decision of which unexplored vertex to visit next can be computed efficiently. A natural question is whether stronger lower bounds on the competitive ratio can be established when restricting to polynomial-time algorithms,

possibly under complexity-theoretic assumptions such as P $\neq$ NP or the Exponential Time Hypothesis. We believe that investigating such problems could serve as a bridge between online optimization and computational complexity.

Bibliography

[1] Susanne Albers and Monika R. Henzinger. Exploring unknown environments. *SIAM Journal on Computing*, 29(4):1164–1188, 2000.

[2] Luca Allulli, Giorgio Ausiello, and Luigi Laura. On the power of lookahead in on-line vehicle routing problems. In *Proceedings of the 11th Annual International Computing and Combinatorics Conference (COCOON)*, pages 728–736, 2005.

[3] Basak Alper, Nathalie Henry Riche, Gonzalo Ramos, and Mary Czerwinski. Design study of LineSets, a novel set visualization technique. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2259–2267, 2011.

[4] Ingo Althöfer, Gautam Das, David P. Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9:81–100, 1993.

[5] Sanjeev Arora. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *Journal of the ACM*, 45(5):753–782, 1998.

[6] Norbert Ascheuer, Sven Oliver Krumke, and Jörg Rambau. Online dial-a-ride problems: Minimizing the completion time. In *Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 639–650, 2000.

[7] Giorgio Ausiello, Luca Allulli, Vincenzo Bonifaci, and Luigi Laura. On-line algorithms, real time, the virtue of laziness, and the power of clairvoyance. In *Proceedings of the 3rd International Conference on Theory and Applications of Models of Computation (TAMC)*, pages 1–20, 2006.

[8] Giorgio Ausiello, Vincenzo Bonifaci, and Luigi Laura. The on-line asymmetric traveling salesman problem. *Journal of Discrete Algorithms*, 6(2):290–298, 2008.

[9] Giorgio Ausiello, Marc Demange, Luigi Laura, and Vangelis Th. Paschos. Algorithms for the on-line quota traveling salesman problem. *Information Processing Letters*, 92(2):89–94, 2004.

[10] Giorgio Ausiello, Esteban Feuerstein, Stefano Leonardi, Leen Stougie, and Maurizio Talamo. Algorithms for the on-line travelling salesman. *Algorithmica*, 29(4):560–581, 2001.

[11] Júlia Baligács, Yann Disser, Andreas Emil Feldmann, and Anna Zych-Pawlewicz. A $(5/3+\varepsilon)$-approximation for tricolored non-crossing euclidean TSP. In *Proceedings of the 32nd Annual European Symposium on Algorithms (ESA)*, pages 15:1–15:15, 2024.

[12] Julia Baligács, Yann Disser, Irene Heinrich, and Pascal Schweitzer. Exploration of graphs with excluded minors. In *Proceedings of the 31st European Symposium on Algorithms (ESA)*, pages 11:1–11:15, 2023.

[13] Júlia Baligács, Yann Disser, Nils Mosis, and David Weckbecker. An improved algorithm for open online dial-a-ride. In *Proceedings of the 20th Workshop on Approximation and Online Algorithms (WAOA)*, pages 154–171, 2022.

[14] Júlia Baligács, Yann Disser, Farehe Soheil, and David Weckbecker. Tight analysis of the lazy algorithm for open online dial-a-ride. In *Proceedings of the 18th International Algorithms and Data Structures Symposium (WADS)*, pages 43–64, 2023.

[15] Yair Bartal and Lee-Ad Gottlieb. A linear time approximation scheme for Euclidean TSP. In *Proceedings of the 54th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 698–706, 2013.

[16] Yair Bartal, Lee-Ad Gottlieb, and Robert Krauthgamer. The traveling salesman problem: Low-dimensionality implies a polynomial time approximation scheme. *SIAM Journal on Computing*, 45(4):1563–1581, 2016.

[17] Mandell Bellmore and George L. Nemhauser. The traveling salesman problem: A survey. *Operations Research*, 16(3):538–558, 1968.

[18] Sergey Bereg, Krzysztof Fleszar, Philipp Kindermann, Sergey Pupyrev, Joachim Spoerhase, and Alexander Wolff. Colored non-crossing euclidean steiner forest. In *Proceedings of the 26th International Symposium on Algorithms and Computation (ISAAC)*, pages 429–441, 2015.

[19] Marcin Bienkowski, Artur Kraska, and Hsiang-Hsuan Liu. Traveling repairperson, unrelated machines, and other stories about average completion times. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 28:1–28:20, 2021.

[20] Marcin Bienkowski and Hsiang-Hsuan Liu. An improved online algorithm for the traveling repairperson problem on a line. In *Proceedings of the 44th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 6:1–6:12, 2019.

[21] Alexander Birx. *Competitive analysis of the online dial-a-ride problem*. PhD thesis, TU Darmstadt, 2020.

[22] Alexander Birx and Yann Disser. Tight analysis of the smartstart algorithm for online dial-a-ride on the line. *SIAM Journal on Discrete Mathematics*, 34(2):1409–1443, 2020.

[23] Alexander Birx, Yann Disser, Alexander V. Hopp, and Christina Karousatou. An improved lower bound for competitive graph exploration. *Theoretical Computer Science*, 868:65–86, 2021.

[24] Alexander Birx, Yann Disser, and Kevin Schewior. Improved bounds for open online dial-a-ride on the line. *Algorithmica*, 2022.

[25] Antje Bjelde, Jan Hackfeld, Yann Disser, Christoph Hansknecht, Maarten Lipmann, Julie Meißner, Miriam Schlöter, Kevin Schewior, and Leen Stougie. Tight bounds for online tsp on the line. *ACM Transactions on Algorithms*, 17(1), 2020.

[26] Michiel Blom, Sven O. Krumke, Willem E. de Paepe, and Leen Stougie. The online TSP against fair adversaries. *INFORMS Journal on Computing*, 13(2):138–148, 2001.

[27] Vincenzo Bonifaci and Leen Stougie. Online k-server routing problems. *Theory of Computing Systems*, 45(3):470–485, 2008.

[28] Vincenzo Bonifaci and Leen Stougie. Online k-server routing problems. *Theory of Computing Systems*, 45(3):470–485, 2009.

[29] Glencora Borradaile, Erik D. Demaine, and Siamak Tazari. Polynomial-time approximation schemes for subset-connectivity problems in bounded-genus graphs. *Algorithmica*, 68(2):287–311, 2014.

[30] Glencora Borradaile, Hung Le, and Christian Wulff-Nilsen. Minor-free graphs have light spanners. In *Proceedings of the 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 767–778, 2017.

[31] Sebastian Brandt, Klaus-Tycho Foerster, Jonathan Maurer, and Roger Wattenhofer. Online graph exploration on a restricted graph class: Optimal solutions for tadpole graphs. *Theoretical Computer Science*, 839:176–185, 2020.

[32] Thom Castermans, Mereke van Garderen, Wouter Meulemans, Martin Nöllenburg, and Xiaoru Yuan. Short plane supports for spatial hypergraphs. *Journal of Graph Algorithms and Applications*, 23(3):463–498, 2019.

[33] Kamalika Chaudhuri, Brighten Godfrey, Satish Rao, and Kunal Talwar. Paths, trees, and minimum latency tours. In *Proceedings of the 44th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 36–45, 2003.

[34] Shiri Chechik and Christian Wulff-Nilsen. Near-optimal light spanners. *ACM Transactions on Algorithms*, 14(3):1–15, 2018.

[35] Chandra Chekuri and Amit Kumar. Maximum coverage problem with group budget constraints and applications. In *Proceedings of the 7th International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX)*, volume 3122, pages 72–83, 2004.

[36] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. In *Operations Research Forum*, volume 3, page 20. Springer, 2022.

[37] Romain Cosson. Breaking the $k/\log k$ barrier in collective tree exploration via tree-mining. In *Proceedings of the 35th Annual Symposium on Discrete Algorithms (SODA)*, pages 4264–4282, 2024.

[38] Romain Cosson and Laurent Massoulié. Collective tree exploration via potential function method. In *15th Innovations in Theoretical Computer Science Conference (ITCS)*, pages 35:1–35:22, 2024.

[39] Erik D. Demaine, MohammadTaghi HajiAghayi, and Bojan Mohar. Approximation algorithms via contraction decomposition. *Combinatorica*, 30(5):533–552, 2010.

[40] Xiaotie Deng and Christos H. Papadimitriou. Exploring an unknown graph. *Journal of Graph Theory*, 32(3):265–297, 1999.

[41] Dariusz Dereniowski, Yann Disser, Adrian Kosowski, Dominik Paj, and Przemyslaw Uznański. Fast collaborative graph exploration. *Information and Computation*, 243:37–49, 2015.

[42] Yann Disser, Jan Hackfeld, and Max Klimm. Tight bounds for undirected graph exploration with pebbles and multiple agents. *Journal of the ACM*, 66(6):40(41), 2019.

[43] Yann Disser, Frank Mousset, Andreas Noever, Nemanja Skoric, and Angelika Steger. A general lower bound for collaborative tree exploration. *Theoretical Computer Science*, 811:70–78, 2020.

[44] Stefan Dobrev, Rastislav Královič, and Euripides Markou. Online graph exploration with advice. In *Proceedings of the 19th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 267–278, 2012.

[45] François Dross, Krzysztof Fleszar, Karol Wegrzycki, and Anna Zych-Pawlewicz. Gap-ETH-tight approximation schemes for red-green-blue separation and bicolored noncrossing Euclidean travelling salesman tours. In *Proceedings of the 34nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1433–1463, 2023.

[46] Miroslaw Dynia, Jakub Lopuszanski, and Christian Schindelhauer. Why robots need maps. In *Proceedings of the 14th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 41–50. Springer, 2007.

[47] Franziska Eberle, Alexander Lindermayr, Nicole Megow, Lukas Nölke, and Jens Schlöter. Robustification of online graph exploration methods. In *Proceedings of the 36th Conference on Artificial Intelligence (AAAI)*, pages 9732–9740, 2022.

[48] Alon Efrat, Yifan Hu, Stephen Kobourov, and Sergey Pupyrev. Mapsets: Visualizing embedded and clustered graphs. *Journal on Graph Algorithms and Applications*, 19(2):571–593, 2015.

[49] Paul Erdős. Extremal problems in graph theory. In *Proceedings of the Symposium on Theory of Graphs and its Applications*, pages 29–36, 1963.

[50] Jeff Erickson and Amir Nayyeri. Shortest non-crossing walks in the plane. In *Proceedings of the 22nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 297–308, 2011.

[51] Jittat Fakcharoenphol, Chris Harrelson, and Satish Rao. The k-traveling repairmen problem. *ACM Transactions on Algorithms*, 3(4):40, 2007.

[52] Esteban Feuerstein and Leen Stougie. On-line single-server dial-a-ride problems. *Theoretical Computer Science*, 268(1):91–105, 2001.

[53] Arnold Filtser and Shay Solomon. The greedy spanner is existentially optimal. *SIAM Journal on Computing*, 49(2):429–447, 2020.

[54] Rudolf Fleischer and Gerhard Trippen. Exploring an unknown graph efficiently. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA)*, pages 11–22, 2005.

[55] Klaus-Tycho Foerster and Roger Wattenhofer. Lower and upper competitive bounds for online directed graph exploration. *Theoretical Computer Science*, 655:15–29, 2016.

[56] Pierre Fraigniaud, Leszek Gasieniec, Dariusz R. Kowalski, and Andrzej Pelc. Collective tree exploration. *Networks*, 48(3):166–177, 2006.

[57] Pierre Fraigniaud, David Ilcinkas, Guy Peer, Andrzej Pelc, and David Peleg. Graph exploration by a finite automaton. *Theoretical Computer Science*, 345(2-3):331–344, 2005.

[58] Robin Fritsch. Online graph exploration on trees, unicyclic graphs and cactus graphs. *Information Processing Letters*, 168:106096, 2021.

[59] Lee-Ad Gottlieb. A light metric spanner. In *Proceedings of the 56th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 759–772, 2015.

[60] Michelangelo Grigni. Approximate TSP in graphs with forbidden minors. In *Proceedings of the 27th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1853, pages 869–877, 2000.

[61] Michelangelo Grigni and Hao-Hsiang Hung. Light spanners in bounded path-width graphs. In *Proceedings of the 37th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 467–477, 2012.

[62] Michelangelo Grigni and Papa Sissokho. Light spanners and approximate TSP in weighted graphs with forbidden minors. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 852–857, 2002.

[63] Joachim Gudmundsson and Christos Levcopoulos. A fast approximation algorithm for TSP with neighborhoods. *Nordic Journal of Computing*, 6(4):469, 1999.

[64] Allen Hatcher. *Algebraic topology*. Cambridge University Press, 2000.

[65] D. Hauptmeier, S.O. Krumke, J. Rambau, and H.-C. Wirth. Euler is standing in line dial-a-ride problems with precedence-constraints. *Discrete Applied Mathematics*, 113(1):87–107, 2001.

[66] Dietrich Hauptmeier, Sven Oliver Krumke, and Jörg Rambau. The online dial-a-ride problem under reasonable load. In *Proceedings of the 4th Italian Conference on Algorithms and Complexity (CIAC)*, pages 125–136, 2000.

[67] Yuya Higashikawa, Naoki Katoh, Stefan Langerman, and Shin-ichi Tanigawa. Online graph exploration algorithms for cycles and trees by multiple searchers. *Journal of Combinatorial Optimization*, 28(2):480–495, 2014.

[68] Stefan Hougardy and Mirko Wilde. On the nearest neighbor rule for the metric traveling salesman problem. *Discrete Applied Mathematics*, 195:101–103, 2015.

[69] Ferran Hurtado, Matias Korman, Marc J. van Kreveld, Maarten Löffler, Vera Sacristán, Akiyoshi Shioura, Rodrigo I. Silveira, Bettina Speckmann, and Takeshi Tokuyama. Colored spanning graphs for set visualization. *Computational Geometry*, 68:262–276, 2018.

[70] K. Ilavarasi and K. Suresh Joseph. Variants of travelling salesman problem: A survey. In *International Conference on Information Communication and Embedded Systems (ICICES2014)*, pages 1–7, 2014.

[71] Patrick Jaillet and Xin Lu. Online traveling salesman problems with service flexibility. *Networks*, 58(2):137–146, 2011.

[72] Patrick Jaillet and Xin Lu. Online traveling salesman problems with rejection options. *Networks*, 64(2):84–95, 2014.

[73] Patrick Jaillet and Michael R. Wagner. Generalized online routing: New competitive ratios, resource augmentation, and asymptotic analyses. *Operations Research*, 56(3):745–757, 2008.

[74] Vinay A. Jawgal, V. N. Muralidhara, and P. S. Srinivasan. Online travelling salesman problem on a circle. In *In Proceedings of the 15th International Conference on Theory and Applications of Models of Computation (TAMC)*, pages 325–336, 2019.

[75] Bala Kalyanasundaram and Kirk R. Pruhs. Constructing competitive tours from local information. *Theoretical Computer Science*, 130(1):125–138, 1994.

[76] Anna R. Karlin, Nathan Klein, and Shayan Oveis Gharan. A (slightly) improved approximation algorithm for metric TSP. In *Proceedings of the 53rd Annual ACM Symposium on the Theory of Computing (STOC)*, pages 32–45, 2021.

[77] Anna R. Karlin, Nathan Klein, and Shayan Oveis Gharan. A deterministic better-than-3/2 approximation algorithm for metric TSP. In *Proceedings of the 24th Conference on Integer Programming and Combinatorial Optimization (IPCO)*, pages 261–274, 2023.

[78] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[79] Marek Karpinski, Michael Lampis, and Richard Schmied. New inapproximability bounds for TSP. *Journal of Computer and System Sciences*, 81(8):1665–1677, 2015.

[80] Sándor Kisfaludi-Bak, Jesper Nederlof, and Karol Wegrzycki. A Gap-ETH-Tight Approximation Scheme for Euclidean TSP. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 351–362. IEEE, 2022.

[81] Koji M. Kobayashi and Ying Li. An improved upper bound for the online graph exploration problem on unicyclic graphs. *Journal of Combinatorial Optimization*, 48(1):1–38, 2024.

[82] Michael Krivelevich and Benjamin Sudakov. Minors in expanding graphs. *Geometric and Functional Analysis*, 19(1):294–331, 2009.

[83] Sven O. Krumke. Online optimization competitive analysis and beyond. Habilitation thesis, Zuse Institute Berlin, 2001.

[84] Sven O. Krumke, Willem E. de Paepe, Diana Poensgen, Maarten Lipmann, Alberto Marchetti-Spaccamela, and Leen Stougie. On minimizing the maximum flow time in the online dial-a-ride problem. In *Proceedings of the 3rd International Conference on Approximation and Online Algorithms (WAOA)*, pages 258–269, 2006.

[85] Sven O. Krumke, Willem E. de Paepe, Diana Poensgen, and Leen Stougie. News from the online traveling repairman. *Theoretical Computer Science*, 295(1-3):279–294, 2003.

[86] Sven Oliver Krumke, Luigi Laura, Maarten Lipmann, Alberto Marchetti-Spaccamela, Willem de Paepe, Diana Poensgen, and Leen Stougie. Non-abusiveness helps: An O(1)-competitive algorithm for minimizing the maximum flow time in the online traveling salesman problem. In *Proceedings of the 5th International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX)*, pages 200–214, 2002.

[87] Casimir Kuratowski. Sur le probleme des courbes gauches en topologie. *Fundamenta mathematicae*, 15(1):271–283, 1930.

[88] Yoshiyuki Kusakari, Daisuke Masubuchi, and Takao Nishizeki. Finding a non-crossing Steiner forest in plane graphs under a 2-face condition. *Journal of Combinatorial Optimization*, 5(2):249–266, 2001.

[89] Maarten Lipmann, Xiwen Lu, Willem E. de Paepe, Rene A. Sitters, and Leen Stougie. On-line dial-a-ride problems under a restricted information model. *Algorithmica*, 40(4):319–329, 2004.

[90] Marteen Lipmann. *On-line routing*. PhD thesis, Technische Universiteit Eindhoven, 2003.

[91] Cristian S. Mata and Joseph S. B. Mitchell. Approximation algorithms for geometric tour and network design problems. In *Proceedings of the 11th Annual Symposium on Computational Geometry (SCG)*, pages 360–369, 1995.

[92] Nicole Megow, Kurt Mehlhorn, and Pascal Schweitzer. Online graph exploration: New results on old and new algorithms. *Theoretical Computer Science*, 463:62–72, 2012.

[93] Joseph S. B. Mitchell. A constant-factor approximation algorithm for TSP with pairwise-disjoint connected neighborhoods in the plane. In *Proceedings of the 26th Annual Symposium on Computational Geometry (SCG)*, pages 183–191, 2010.

[94] Joseph S. B. Mitchell. Shortest paths and networks. In *Handbook of Discrete and Computational Geometry, Third Edition*, chapter 31. CRC Press LLC, 3 edition, 2017.

[95] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge University Press, 2017.

[96] Shuichi Miyazaki, Naoyuki Morimoto, and Yasuo Okabe. The online graph exploration problem on restricted graphs. *IEICE transactions on information and systems*, 92(9):1620–1627, 2009.

[97] Bojan Mohar and Carsten Thomassen. *Graphs on Surfaces*. Johns Hopkins University Press, 2001.

[98] Christian Ortolf and Christian Schindelhauer. A recursive approach to multi-robot exploration of trees. In *Proceedings of the 21st International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 8576, pages 343–354, 2014.

[99] Christos H. Papadimitriou. The Euclidean Traveling Salesman Problem is NP-Complete. *Theoretical Computer Science*, 4(3):237–244, 1977.

[100] David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of graph theory*, 13(1):99–116, 1989.

[101] Satish Rao and Warren D. Smith. Approximating Geometrical Graphs via "Spanners" and "Banyans". In *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 540–550, 1998.

[102] Iris Reinbacher, Marc Benkert, Marc van Kreveld, Joseph S. B. Mitchell, Jack Snoeyink, and Alexander Wolff. Delineating boundaries for imprecise regions. *Algorithmica*, 50(3):386–414, 2008.

[103] Omer Reingold. Undirected connectivity in log-space. *Journal of the ACM (JACM)*, 55(4):1–24, 2008.

[104] Gerhard Ringel and John WT Youngs. Solution of the heawood map-coloring problem. *Proceedings of the National Academy of Sciences*, 60(2):438–445, 1968.

[105] Neil Robertson and Paul D. Seymour. Graph minors. XX. wagner's conjecture. *Journal of Combinatorial Theory, Series B*, 92(2):325–357, 2004.

[106] Steven Roman. *An Introduction to Catalan Numbers*. Compact Textbooks in Mathematics. Springer International Publishing, 2015.

[107] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. An analysis of several heuristics for the traveling salesman problem. *SIAM journal on computing*, 6(3):563–581, 1977.

[108] Shmuel Safra and Oded Schwartz. On the complexity of approximating TSP with neighborhoods and related problems. *Computational Complexity*, 14(4):281–307, 2006.

[109] Jun-Ya Takahashi, Hitoshi Suzuki, and Takao Nishizeki. Algorithms for finding non-crossing paths with minimum total length in plane graphs. In *Proceedings of the 3rd International Symposium on Algorithms and Computation (ISAAC)*, pages 400–409. Springer, 1992.

[110] Jun-Ya Takahashi, Hitoshi Suzuki, and Takao Nishizeki. Finding shortest non-crossing rectilinear paths in plane regions. In *International Symposium on Algorithms and Computation*, pages 98–107. Springer, 1993.

[111] Vera Traub and Jens Vygen. *Approximation Algorithms for Traveling Salesman Problems*. Cambridge University Press, 2024.

[112] Fanglei Yi and Lei Tian. On the online dial-a-ride problem with time-windows. In *Proceedings of the 1st International Conference on Algorithmic Applications in Management (AAIM)*, pages 85–94, 2005.

[113] John William Theodore Youngs. Minimal imbeddings and the genus of a graph. *Journal of Mathematics and Mechanics*, pages 303–315, 1963.

Academic Curriculum Vitae

10/2021 – 09/2025	**PhD in Mathematics** Technische Universität Darmstadt
10/2020 – 09/2021	**MSc in Mathematics and Foundations of Computer Science** University of Oxford
10/2017 – 09/2020	**BSc in Mathematics** Technische Universität Darmstadt