Graph Exploration with Advice and Online Crossing Minimization

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

JANOSCH FUCHS

Master of Science

aus Dormagen

Berichter Privatdozent Dr. rer. nat. Walter Unger
Universitätsprofessor Dr. sc. ETH Zürich Dennis Komm
Universitätsprofessor Dr. rer. nat. Peter Rossmanith

Tag der mündlichen Prüfung: 26.08.2022

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek verfügbar.

Abstract

Online problems reveal their input instance piece by piece and require an irrevocable decision each time a new piece is revealed. This scenario models critical applications where a decision, once it is made, cannot be changed afterwards which also influences future decisions. Like for classical offline problems, it is common to make a worst-case analysis to give a guarantee on the performance of algorithms in this setting, which are called online algorithms. This is achieved by assuming that a malicious adversary, knowing the behavior of the online algorithm, creates the input. Due to the nature of uncertainty online algorithms rarely compute an optimal solution. Thus, their performance is measured in terms of the *competitive ratio*, which compares the solution computed by the online algorithm to the optimal solution achievable when the whole instance is known beforehand.

To measure the impact of the missing knowledge about the future the classical online setting was extended with a helpful oracle providing information about the future input. The online algorithm receives the information by accessing a tape containing a binary bit string prepared by the oracle. The number of bits that the algorithm reads during its computation is called its *advice complexity*. Bounds on the advice complexity of an online algorithm that optimally solves an online problem show how much information about the future is needed to avoid any mistake. An obvious upper bound for the advice complexity of all online problems is the size of the binary encoding of the input instance or the optimal solution using self-delimiting encoding. But most of the time it is possible to derive better bounds by encoding only critical decisions which then reveals the core difficulty of an online problem. Thus, the advice complexity strongly correlates to the difficulty of an online problem and it can be used to measure the difficulty of different online problems.

The first technical part of this thesis considers an online graph drawing problem on degree-bounded bipartite graphs, the so-called *online slotted One-Sided Crossing Minimization Problem* (online slotted OSCM-k). One set of vertices together with a total ordering is fixed on a line and known to the online algorithm. In each step a vertex, which is incident to k vertices of the already known set, needs to be placed on a free placeholder, which we call a *slot*. The slots are also arranged on a line parallel to the known vertex set.

The task is to place the vertices such that the number of edge crossings is minimized, when all edges are drawn as straight lines. Note that the slots make the problem harder to solve for an online algorithm but in the offline case it is equivalent to the version without slots. We prove that the online slotted OSCM-k has no constant competitive ratio. Thus, we look at the problem on 2-regular graphs and derive constant upper and lower bounds for this restricted graph class.

The second part analyzes the advice complexity of a graph exploration problem. Here, an algorithm has to move an agent through a graph from vertex to vertex using the edges such that the agent traverses the cheapest or shortest tour that visits every vertex at least once. The algorithm does not know the graph beforehand and each time the agent is located at a new vertex the reachable neighborhood is revealed together with the cost to reach all neighboring vertices from the current location. Already seen or visited vertices are recognizable. Due to the already known tight upper bound of $n \log(n)$ for the advice complexity of the graph exploration problem on dense graphs, we focus on sparse graphs. We show that the number of advice bits which is sufficient and also necessary to solve the graph exploration problem on directed and undirected graphs is linear in the number of edges. We also investigate how much advice can be saved when the graph structure is known beforehand and only the traveling cost of the edges is unknown.

Zusammenfassung

Die Eingabe einer Instanz eines Online-Problems wird nur Stück für Stück aufgedeckt und erfordert jedes Mal eine unwiderrufliche Entscheidung, wenn ein neues Teilstück aufgedeckt wird. Dieses Szenario modelliert kritische Anwendungen bei denen eine Entscheidung, sobald sie einmal getroffen wurde, nicht mehr rückgängig gemacht werden kann und auch zukünftige Entscheidungen beeinflusst. Wie bei klassischen Offline-Problemen, ist es üblich eine worst-case Analyse zu machen um eine Garantie für die Leistung eines Online-Algorithmus zu erhalten. Für die Worst-Case-Analyse wird angenommen, dass ein böswilliger Gegenspieler, der das Verhalten des Online-Algorithmus kennt, die Eingabe erstellt. Online-Algorithmen schaffen es nur selten eine optimale Lösung zu bestimmen. Deshalb wird ihre Leistung mit der *competitive ratio* gemessen, die die Lösung des online Algorithmus mit der optimalen Lösung vergleicht, die erreicht werden kann, wenn die komplette Eingabe im Vorhinein bekannt ist.

Um die Auswirkungen des fehlenden Wissens über die zukünftigen Anfragen zu messen, wurde das klassische Online-Szenario um ein hilfreiches Orakel erweitert, das Auskunft über die noch kommenden Eingabeteile gibt. Der Online-Algorithmus kann über ein Band auf die Information zugreifen. Dieses Band ist mit einer Sequenz von Bits beschriftet. Die Anzahl der während der Berechnung gelesenen Bits ist dann die so genannte *advice complexity*. Durch möglichst scharfe Schranken an die advice complexity eines Algorithmus, der ein Online-Problem optimal löst, zeigt man wie viele Informationen über die Zukunft wirklich notwendig sind, um alle Fehler zu vermeiden. Eine offensichtliche obere Schranke für die advice complexity eines jeden Online-Problems ist die Größe einer binären Codierung der kompletten Eingabe oder der kompletten optimalen Lösung. Aber meistens ist es möglich, bessere Schranken zu entwickeln indem man nur kritische Entscheidungen codiert, was den schweren Kern eines Problems offenbart. Daher besteht ein starker Zusammenhang zwischen der advice complexity und der Schwere eines Online-Problems. Somit ist es möglich, mit Hilfe der advice complexity die Komplexität verschiedener Online-Probleme miteinander zu vergleichen.

Der erste technische Teil dieser Dissertation betrachtet ein Online-Graphzeichnungsproblem auf bipartiten Graphen mit beschränktem Grad. Das

Problem wird *online slotted One-Sided Crossing Minimzation* genannt (online slotted OSCM-k). Eine der bipartiten Mengen von Knoten und eine dazugehörige Anordnung auf einer Geraden sind zu Beginn bekannt. In jedem Schritt muss ein Knoten, der mit k Knoten aus der bereits bekannten Menge verbunden ist, einem leerem Platzhalter zugeordnet werden. Diese Platzhalter werden *slots* genannt und sind ebenfalls auf einer Linie angeordnet, die parallel zu der Linie verläuft auf der die bereits bekannten Knoten angeordnet sind. Die Aufgabe besteht darin, die Knoten so zu platzieren, dass die Anzahl der Kreuzungen zwischen den Kanten minimiert wird, wenn alle Kanten als gerade Linien gezeichnet werden. Die slots machen das Problem schwer für einen Online-Algorithmus, aber für die Betrachtung im Offline-Fall machen sie keinen Unterschied. Wir zeigen, dass die competitive ratio des online slotted OSCM-k nicht durch eine konstante beschränkt ist und entwickeln obere und untere Schranken für das Problem, wenn die Eingabe auf 2-reguläre Graphen beschränkt ist.

Der zweite Teil analysiert die advice complexity eines Graph Exploration Problems. Bei diesem Problem muss ein Algorithmus einen Agenten über die Kanten eines Graphen von Knoten zu Knoten bewegen, sodass der Agent die günstigste oder kürzeste Tour abläuft, die alle Knoten des Graphen mindestens einmal besucht. Der Algorithmus kennt dabei den Graphen zu Beginn nicht, sieht aber jedes Mal, wenn der Agent auf einem neuen Knoten ist, die mit einer Kante erreichbare Nachbarschaft und die dazugehörigen Kosten der Kante. Bereits gesehene oder besuchte Knoten werden dabei wiedererkannt. Bedingt durch die bereits bekannte obere Schranke von $n \log(n)$, die asymptotisch identisch mit der unteren Schranke ist, konzentrieren wir uns auf Graphen, die im Verhältnis zu den Knoten nur linear viele Kanten haben. Wir zeigen, dass eine in der Anzahl der Kanten lineare Menge von Advice-Bits ausreicht und auch notwendig ist, um das graph exploration Problem auf gerichteten und ungerichteten Graphen zu lösen. Wir untersuchen zusätzlich noch, wie viel Advice eingespart werden kann, wenn die Struktur des Graphen (die Kanten und Knoten) im Vorhinein bekannt ist und nur die Information über die Kosten der Kanten fehlt.

Acknowledgment

I am very thankful for my time at the I1 in Aachen and for the wonderful people that I met and that supported me during that time. First of all, I thank Gerhard Woeginger. In his time at our department he contributed a great amount of deep academic knowledge. Every time I needed a broader overview of a topic, encountered an unknown problem or needed help to place a result in an academic context, I went to Gerhard and got his support. I always admired his kindness, patience and knowledge. Unfortunately he passed away far too early.

Walter Unger and Hans-Joachim Böckenhauer helped me throughout the entire time as a PhD student. They shaped me as a researcher, taught me how to think about theoretical problems and how to write up scientific results. I admire their patience with me and cannot express how thankful I am for all the time they invested in my education. However, it is not only the professional relationship, which I am thankful for. Over the last seven years, we also became friends and after long working sessions they taught me how to play Skat. I will always remember the entertaining and funny discussions we had and hope there are many more to come in the future.

Additionally, I want to thank Peter Rossmanith for inviting me to the research visit in Taiwan and all the interesting discussions we had, Björn Tauer for the amazing time we had together during research visits, conferences and in the gym, Henri Lotze for his help and being an awesome colleague although we are not working at the same chair and Tom Janßen who proof-read this thesis very quickly due to the difficult situation that occurred at our department.

Last but not least, I thank several persons from my personal life. My girlfriend Eva Fluck is an important part of my life and I am very thankful that she accepts me the way I am and stands at my side during hard times. I could always talk to my good friends Martin Bellgardt and Benjamin Hohlmann about every topic that bothered me and they supported me in several difficult situations in my life. Special thanks to my parents, who would give everything for me. I admire them and without their support during my whole life, I would not have come so far.

Contents

Chapter 1

Introduction

Many real-world problems are representable by a combinatorial optimization problem and inherit an online nature. Requests arrive over time and need to be resolved as fast as possible. A new request should not influence the decision that was already made on a previous request. Also waiting until all requests arrived and computing the optimal solution at the end is often not a feasible approach. Thus, a decision for a request is irrevocable and is only based on the previous requests and the corresponding decisions. Computing an optimal solution in such a scenario is often impossible due to the lack of knowledge about the future. However, the question arises whether there is an approach to solve problems in such an environment that still guarantees a certain quality of the online computed solution. This is at its core the problem of online algorithms computing a solution to an online optimization problem.

The terminology of online algorithms was introduced by Sleator and Tarjan [55]. For online algorithms, the running time and the memory usage during computation is not limited. Similar to the classical offline setting, online algorithms are analyzed in a worst-case scenario. A malicious adversary creates the input, knowing the behavior of the online algorithm. The performance of an online algorithm for optimization problems is measured with the competitive ratio. It is the worst case ratio between the solution cost of an online algorithm and the optimal solution that would be achievable when the instance were known beforehand. For a deeper introduction to online algorithms and competitive analysis we refer to the book "Online computation and competitive analysis" by Borodin and El-Yaniv [13].

The online problems considered in this thesis are all online minimization problems on graphs. A graph $G = (V, E)$ is an abstract structure of objects, called vertices, and the relations between them. If two vertices $u, w \in V$ are related, the set of edges E contains the edge (u, v). Throughout the thesis, we denote for any graph $G = (V, E)$ with vertex set V and edge set E, the number of vertices by $n = |V|$ and the number of edges by $m = |E|$.

1.1 Online Problems with Advice

In several scenarios an autonomous agent, which is in contact with a more powerful system, perceives some local information and has to react in an online manner. The main system knows everything the agent knows but cannot make the decision, although it would be capable of making the optimal decision, because it needs to be physically present. To prevent the agent from making decisions with strong negative impact, the main system can communicate with the agent, e.g., sending more information that is unknown to the agent. However, the communication is the cost intensive bottleneck of this setup and the main system wants to send as little information as possible. Thus, it needs to send the crucial information extending the limited knowledge of the agent such that it can make a best possible response. So the question arises what the minimal amount of information that is required by the agent to always give the best possible response is.

This scenario captures the main idea of online problems with advice. An online algorithm with advice has access to a binary string containing information created by a helpful oracle that knows the complete input and the behavior of the algorithm. The algorithm can access the information on the so-called advice tape at any time during its computation. The number of bits that the algorithm reads until it finishes its computation is then called its *advice complexity*. Thus, minimizing the advice complexity of an optimal online algorithm is equivalent to finding the crucial information that is missing in the online setting. The length of the advice tape is unbounded and the algorithm does not know where the prepared string of information ends. Otherwise it would be possible to encode some information only by the length of the advice string and not its content. This additional cannot be measured, which is the main concern of the advice model. Dobrev et al. [21] introduced the online model with advice, which was later refined by Hromkovič et al. [35] and Böckenhauer et al. [12] as well as by Emek et al. [27]. In this thesis we use the model of Hromkovič et al. and Böckenhauer et al. .

Analyzing the advice complexity is not only an approach to resolve situations as described above, it also helps to make online problems more comparable and allows a more fine-grained analysis of their complexity. In the classical online analysis the only parameter measuring the performance of an online algorithm is the competitive ratio. It seems intuitive to compare the achievable competitive ratio of different online problems. But many problems possess a difficulty such that every online algorithm has an arbitrary large competitive ratio. However, by encoding the complete input, every online problem with advice can be solved optimally. Encoding the complete input often contains redundant information and not only the crucial one. Analyzing lower and upper bounds for the advice complexity of online problems instead allows a more differentiated classification.

For a deeper introduction to online algorithms with advice we refer the reader to the book An Introduction to Online Computation by Komm [38] and the survey by Boyar et al. [14].

1.2 Problems and Results

This work is split into two parts, which consider very different online problems. Therefore, an additional introduction and the related work can be found at the beginning of the respective chapters. To be precise, Sections 2.1, 3.1 and 3.7.1 contain related work for the different topics considered in this thesis.

In the first part, we look at an online graph drawing problem, called the one-sided crossing minimization problem. We analyze its behavior in the classical online scenario without any advice. The input graph is 2-regular and bipartite and the algorithm has a priori knowledge about the vertices of one set, their ordering on a straight line and a set of free slots positioned on a line parallel to the first. Each request is a vertex with its neighbors and the task is to place it on one of the free slots such that the total number of edge crossings at the end is minimized. The slots make the problem more difficult for an online algorithm and capture the idea of irrevocable decisions in online problems because once two consecutive slots are filled it is forbidden to place another request in between. An offline algorithm just computes the ordering for the unknown set and assigns the vertices to the slots afterwards. We present a deterministic online algorithm and prove that it achieves a constant competitive ratio.

Secondly, we look at the graph exploration problem, where the task is to move an agent through an unknown graph along its edges such that it traverses the shortest tour visiting every vertex at least once. Depending on the desired form of the solution, we distinguish between the cyclic and the non-cyclic variant of the problem. The graph is revealed during exploration and each time the agent visits a new vertex, its incident edges, their costs and the adjacent vertices are revealed. Vertices that have been revealed or already visited are recognized when the agent is placed on a vertex that would reveal the vertex again. In this work, we only consider the advice complexity of the problem. Thus, we focus on the number of advice bits that are necessary and sufficient to compute an optimal solution on directed or undirected graphs. Due to an already existing tight upper bound for general graphs, we focus on sparse graphs, i.e., graphs where the number of edges is linear in the number of vertices. We provide upper and lower bounds that are linear in the number of vertices. In the end, we also consider a scenario where the algorithm knows the graph structure and only the edge costs have to be learned during exploration.

1.3 Bibliographic Note

The results discussed in this thesis are currently in three different stages of publishing. Some parts have already been accepted at a conference, but not yet published. Others extend previously published results and are currently under review in a peer-reviewed journal. There are also completely new results that will be published later.

The results presented in Chapter 2 have been accepted at the 33rd International Workshop on Combinatorial Algorithms (IWOCA 2022). A full version containing all proofs is available as preprint on arxiv.org:

- [16] Elisabet Burjons, Janosch Fuchs and Henri Lotze. The Slotted Online One-Sided Crossing Minimization Problem on 2-Regular Graphs. abs/2201.04061, 2022.

Preliminary results that form the foundation of Chapter 3 have previously been published on a peer-reviewed conference and an extended version containing all proofs is available at arxiv.org. An improved version with new results and tighter bounds has been submitted to a journal and is currently under review:

- [11] Hans-Joachim Böckenhauer, Janosch Fuchs and Walter Unger. Exploring Sparse Graphs with Advice (Extended Abstract). *In Proceedings of the 16th International Workshop on Approximation and Online Algorithms*, WAOA 2018.

- [10] Hans-Joachim Böckenhauer, Janosch Fuchs and Walter Unger. The Graph Exploration Problem with Advice. abs/1804.06675, 2018.

During his studies as a PhD student, the author also contributed to the following publications that are not covered in this thesis:

- [56] Bjoern Tauer, Dennis Fischer, Janosch Fuchs, Laura Vargas Koch and Stephan Zieger. Waiting for Trains: Complexity Results. *In the 6th International Conference on Algorithms and Discrete Applied Mathematics*, CALDAM 2020.

- [22] Jan Dreier, Janosch Fuchs, Tim A. Hartmann, Philipp Kuinke, Peter Rossmanith, Bjoern Tauer and Hung-Lung Wang. The Complexity of Packing Edge-Disjoint Paths. *In the 14th International Symposium on Parameterized and Exact Computation*, IPEC 2019.

- [4] Lali Barrière, Xavier Muñoz, Janosch Fuchs and Walter Unger. Online Matching in Regular Bipartite Graphs. *In Parallel Processing Letters 28(2)*, 2018.

Chapter 2

Online Crossing Minimization

In graph drawing problems, given a graph, one usually wants to embed the graph into some space with limited dimension. The most common and practical examples are on the Euclidean plane. It is also common to try and embed such graphs in a way that minimizes the number of edges that cross each other. If a graph can be embedded in the Euclidean plane without any crossings, we say the graph is planar. A survey on graph drawing and crossing minimization can be found in [5, 52].

One common way to depict bipartite graphs is by arranging the vertices in each partition on a straight (horizontal) line, making the lines for the two partition sides parallel. In this scenario, the edges are drawn vertically from one side of the partition to the other as straight lines. Thus, the problem of minimizing the crossings in this scenario is reduced to properly ordering the vertices in each partition. However, in some practical applications it is enough to restrict ourselves to ordering one set of the partition (the free side), while the other set remains fixed (the fixed side). It is also common to restrict the degree of the vertices in the free side [44, 42]. This (one-sided) problem is formally defined as follows.

Definition 1. *Given a bipartite Graph $G = (S \dot{\cup} V, E)$, let the nodes of S and V be aligned in some ordering on straight lines parallel to each other, where S is on the top line and V on the bottom line. Let the edges E be drawn as straight lines only. Let the degree of the nodes of S be bounded by some $k \in \mathbb{N}$. The* One-Sided Crossing Minimization Problem (OSCM-k) *is defined as the problem of finding a total ordering of the nodes of S such that the number of resulting edge crossings in the graph is minimized.*

We will assume that the ordering of V is part of the instance and fixed, such that we can label and reference the nodes of V with ascending natural numbers, starting from the "left". If $|S| = |V|$, we sometimes speak of nodes "above" and "below" one another, by assuming that the nodes on both lines are drawn equidistantly.

2.1 Related Work

The OSCM problem has already been extensively studied in the past under different names, such as *bipartite crossing number* [33, 52], *crossing problem* [25], *fixed-layer bipartite crossing minimization* [42] and others. Eades and Wormald [25] showed that the OSCM problem is NP-complete for dense graphs, while Muñoz et al. [44] showed NP-completeness for sparse graphs. Muñoz et al. also introduced the OSCM-k and showed that the OSCM-2 can be solved optimally using the barycenter heuristic.

Li and Stallmann [42] showed that the barycenter heuristic has an approximation ratio of $\Omega(\sqrt{n})$ on general bipartite graphs and also proved that OSCM-k admits a tight $k-1$ approximation. Nagamochi presented a randomized approximation algorithm for general graphs [45] and another approximation algorithm for bipartite graphs of large degree [46].

Further researching the complexity, Dujmović and Whitesides [23] first showed that OSCM is fixed parameter tractable, i.e., it can be solved in $f(k)n^{\mathcal{O}(1)}$, where the parameter k is the number of crossings. The currently best known FPT running time is in $\mathcal{O}(k2^{\sqrt{2k}} + n)$ and was given by Kobayashi and Tamaki [37].

To the best of our knowledge, the field of online analysis of crossing minimization is hardly researched. A closely related problem arises in the field of graph drawing, called *dynamic graph drawing*. Here, the task is to visually arrange a graph that is iteratively expanded over time. The visualization follows certain empirical criteria to make the data comprehensible, where crossing minimization is one of these criteria. For a survey regarding dynamic graph drawing, see [53]. Dynamic graph drawing has many applications, for instance Frishman and Tal [32] presented an algorithm to compute online layouts for a sequence of graphs and its application in discussion thread visualization and social network visualization. In another example North and Woodhull [49] focus on hierarchical graph drawing, defined on a more restricted graph class that needs to be visualized in a tree-like fashion, which overlaps with our topic regarding applications. While one of the most mentioned applications of the offline OSCM is wire crossing minimization in VLSI, this is arguably less applicable when looking at an online version of the problem. However, the results of an online analysis can be helpful for the application fields of graph drawing, e.g., software visualization, decision support systems and interactive graph editors.

While dynamic graph drawing and online graph problems are similar in that parts of the graph are revealed in an iterative fashion and not previously known, a central difference is that in dynamic graph drawing the manipulation of previous decisions is usually allowed. This is not the case in the classical online model. Thus, while theory and practice are looking at similar problems, and are following the same goal of aesthetic graph drawings, the methods to achieve this goal are different.

2.1.1 Overview of Contribution

In this chapter we analyze the online version of the OSCM-k problem. Observe that it can be defined in two different ways. The first version is the online *free* OSCM-k: Given a bipartite graph $(S \dot{\cup} V, E)$, an algorithm initially sees a fixed set of vertices V, and then in each step a request appears for a subset of vertices $R_i \subseteq V$, such that every vertex must be made adjacent to a vertex in S. Thus, after the arrival of the request R_i, one has to place a vertex $s_i \in S$ on the top line and adjacent to the vertices in R_i. In this version, one chooses the partial ordering of s_i with respect to the other vertices already present in S.

The online free OSCM-k problem is solvable with a competitive ratio of at most $k-1$, using the same barycenter algorithm as in the offline case [42].

Consequently we focus on a different version of this problem, which we call the online *slotted* OSCM-k, which is formally defined as follows.

Definition 2. *Given a vertex set V, a request sequence for online slotted OSCM-k is a sequence $R_1, \ldots, R_n$ of subsets of V, each of size k. These requests are not necessarily disjunct. The second set of vertices S is initiated as $S = \{s_1, \ldots, s_n\}$. Initially there are no edges between S and V. Once a request $R_j \subseteq V$ arrives, an online algorithm solving online slotted OSCM-k chooses a vertex s_i without any edges, and places an edge between s_i and every vertex in R_j. The goal is to minimize the total number of crossings.*

The slotted OSCM-k is a model that follows the aesthetic paradigms of the area of dynamic graph drawing, where the so-called *mental map* and human readability is sustained. The term *mental map* describes the goal to make current visualizations of the graph recognizable in later iterations of the graph. Unlike for the free OSCM-k, no upper or lower bound on the competitive ratio is known.

We call the vertices $s_i \in S$ *slots*. If a request $R_j \subseteq V$ is satisfied by adding edges between every vertex in R_j and slot s_i we say that request R_j is *assigned* to slot s_i. Moreover, we call a slot s_i *unsatisfied* or *free* if no request has been satisfied using this slot, thus the slot has no edges yet. Analogously, a *satisfied* slot s_i is a slot in S with edges to a subset $R_j \subseteq V$.

Online slotted OSCM-k, gives the algorithm the advantage of knowing the number of requests in advance. However, one has the distinct constraint that, once two consecutive slots are satisfied, the algorithm will not be able to assign any request to a vertex between the satisfied slots, as such a vertex does not exist.

We prove that online slotted OSCM-k is not competitive for any $k \geq 2$ in general graph classes. However, if we focus on 2-regular graphs, we prove that this problem has a constant competitive ratio. In particular, we prove a lower bound of $4/3$ in this case, and then present an algorithm with a competitive ratio of at most 5 as an upper bound.

2.2 Lower Bounds on General Graphs

We begin by looking at online slotted OSCM-k on general graphs, and show that for every non-trivial value of k, i.e., $k \geq 2$, there is no algorithm with a constant competitive ratio.

Theorem 1. *There is no online algorithm with a constant competitive ratio for online slotted OSCM-k, for any $k \geq 2$.*

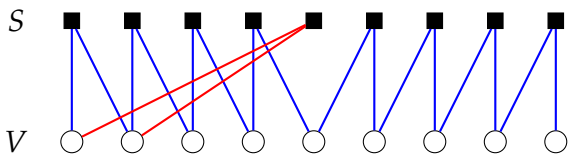

Figure 2.1: Theorem 1: At first the requests $\{v_i, v_{i+1}\}$ are presented, for $1 \leq i \leq n-1$. These requests can be placed such that there are no edge crossings, like shown in blue. Some slot has to be left open for which the request associated with the red edges is given.

Proof. Let us consider an algorithm A solving online slotted OSCM-k. Given the initial sets of vertices $V = \{v_1, \ldots, v_n\}$ and slots $S = \{s_1, \ldots, s_n\}$, A is presented the following request sequence: $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}$. Assume without loss of generality that A has assigned these requests to slots in S without producing a single crossing. Since we have n requests to fill n slots with, and A has only one unsatisfied slot s_i for some $i \in \{1, ..., n\}$, the last request will be assigned to s_i. We assume without loss of generality that $i \geq \lfloor \frac{n}{2} \rfloor$. The adversary now presents the request $\{v_1, v_2\}$ as the last request of the input. This results in at least $2 \cdot 2 \cdot (\frac{n}{2} - 1)$ crossings as opposed to the optimal solution, which only results in a single crossing as depicted in Figure 2.1. The competitive ratio is thus at least $\frac{2 \cdot 2 \cdot (\frac{n}{2}-1)}{1} = 2n - 4$ and therefore not bounded by any fixed constant c. □

If we look closely at the proof, we see that the arguments rely on the adversary, constructing the instance, being able to freely choose the degrees of the vertices in V. If we required the degree of the vertices in V to be defined in advance, the same strategy would not work. Thus, it makes sense to look at graph classes where the degree of the vertices in the graph is fixed, in particular regular graphs.

In the following we focus on online slotted OSCM-2 on 2-regular graphs, as this particular case is already hard to analyze, and we prove that the competitive ratio is within the range between $\frac{4}{3}$ and 5.

We conjecture that for any higher degree, online slotted OSCM-k on k-regular graphs would also have a constant competitive ratio with the constant depending on k. One can observe that a higher vertex degree means

that even optimal solutions must have a lot of crossings. Thus, even when an online algorithm makes a sub-optimal choice, the number of crossings of the optimal solution it is compared to should to some extend compensate for the mistakes.

2.3 Lower Bound for 2-Regular Graphs

We begin by proving a lower bound for the competitive ratio of online slotted OSCM-2 on 2-regular graphs.

It is important to note that an offline algorithm can find an optimal solution in a greedy fashion, as we will see in Lemma 1. In the following lower bound, we prove that online algorithms cannot find an optimal solution, greedily or otherwise. The difficulty is that a request cannot be assigned in between two consecutive satisfied slots. Thus, an online algorithm has to fulfill a request by assigning it to a sub-optimal slot. An example of such a situation is depicted in Figure 2.2. We can use this fact to construct a lower bound for online slotted OSCM-2 on 2-regular graphs as follows.

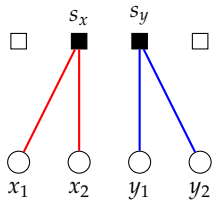

Figure 2.2: In this graph, a new request $R_i = \{x_2, y_1\}$ appears. This request cannot be satisfied optimally. An assignment between s_x and s_y would be optimal, but there is no free slot between them.

Theorem 2. *Every deterministic online algorithm solving the slotted OSCM-2 on 2-regular graphs has a competitive ratio of at least* $4/3 - \varepsilon$*, for any* $\varepsilon > 0$*.*

Proof. For any node, every algorithm only has a finite number of slots to insert it into. Given an empty graph of size $n > 6$, the adversary will repeat its strategy on the set of the five leftmost free nodes, filling up the graph from left to right until 6 or fewer free nodes are left in the graph. Given five free slots, the adversary will repeat the strategy depicted in Figure 2.3 which we will now describe in detail. For ease of notation, we will denote the five leftmost free slots as $s_1, \ldots, s_5$ and the five leftmost edge-free vertices as $v_1, \ldots, v_5$.

The adversary starts by presenting the request $\{v_3, v_4\}$. We consider two possibilities: Either an algorithm places this request at s_3 or smaller, or s_4 or higher.

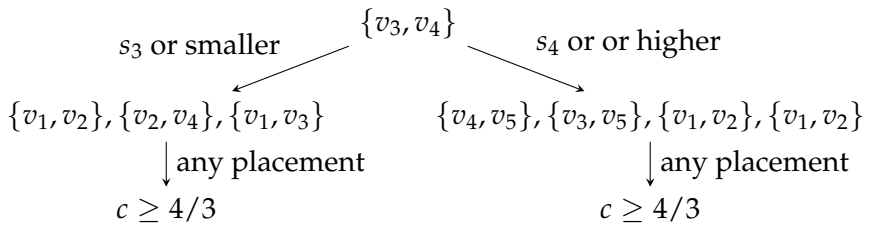

Figure 2.3: Theorem 2: All possible behaviors of any algorithm, when presented with request $\{v_3, v_4\}$, and the resulting bound on the competitive ratio of each decision branch, when confronted with these adversarial requests.

Case 1 (Algorithm assigns $\{v_3, v_4\}$ to s_3 or smaller): The adversary presents the request $\{v_1, v_2\}$. We assume that any reasonable algorithm places the second request to the left of the first one (on a smaller slot). If an algorithm places the second request to the right, it would directly incur 4 crossings instead of none.

We branch on three possibilities depending on the free slots after the first two placements. The free slots are either $\{s_1, s_4, s_5\}$, $\{s_2, s_4, s_5\}$, or $\{s_3, s_4, s_5\}$. The adversary presents the request $\{v_2, v_4\}$ and subsequently the request $\{v_1, v_3\}$.

- If the free slots are $\{s_3, s_4, s_5\}$, any assignment of $\{v_2, v_4\}$ and $\{v_1, v_3\}$ results in at least 7 crossings.
- If the free slots are $\{s_2, s_4, s_5\}$, any assignment of $\{v_2, v_4\}$ and $\{v_1, v_3\}$ results in at least 4 crossings.
- If the free slots are $\{s_1, s_4, s_5\}$, any assignment of $\{v_2, v_4\}$ and $\{v_1, v_3\}$ results in at least 5 crossings.

However, an assignment of $\{v_3, v_4\}$ to s_4, $\{v_1, v_2\}$ to s_1, $\{v_2, v_4\}$ to s_3 and $\{v_1, v_3\}$ to s_2 results in only 3 crossings, where the considered algorithms all have at least four crossings. Thus, the competitive ratio of any algorithm assigning request $\{v_3, v_4\}$ to s_3 or lower is at least $4/3$ on these four requests.

Case 2 (Algorithm assigns $\{v_3, v_4\}$ to s_4 or larger): The adversary gives the request $\{v_4, v_5\}$ followed by $\{v_3, v_5\}$ and the two identical requests $\{v_1, v_2\}$.

The assignment of the last two requests to slots s_1 and s_2 is optimal and generates one crossing.

The optimal assignment places $\{v_3, v_4\}$ to s_3, $\{v_4, v_5\}$ to s_5 and $\{v_3, v_5\}$ to s_4, resulting in two crossings; 3 in total with the last two requests.

However, the algorithms we consider cannot assign $\{v_3, v_4\}$ to s_3, so they assign it to s_4 or higher. These algorithms incur at least 3 crossings for the first three requests, and 4 crossings in total for the five requests. Thus, they also have a competitive ratio of at least $4/3$ on these five requests.

Note that in Case 1, the adversary presents only four nodes in total, while in Case 2 five nodes are used. Independent of which case is used, the adversary can now use the five leftmost free slots and edge-free vertices to repeat this tactic. Once $r \leq 6$ slots are left, the remaining slots are filled up as follows. The adversary presents the following r requests: $\{v_{n-r}, v_{n-r+1}\}, \ldots, \{v_{n-1}, v_n\}$. One slot is still free after presenting these requests, and the last request is $\{v_{n-r}, v_n\}$. This results in $r - 1 \leq 5$ additional, unavoidable crossings.

From the case distinction above and the argument to fill up the rest of the graph, one can easily verify that the lower bound on the competitive ratio of every algorithm tends to $4/3$ for growing n. □

This lower bound proves that no online algorithm for online slotted OSCM-2 on 2-regular graphs can perform optimally on all instances. In the following we introduce some notions used to prove an upper bound for the competitive ratio in the same setting.

2.4 Preliminaries and Notation

In order to prove upper bounds for online slotted OSCM-2 on 2-regular graphs, we need to first extract some structural properties of this problem. We introduce the notion of *propagation arrows*, which helps us to lower bound the total number of crossings of the remaining graph if we only have a partial request sequence. Then we observe that finding an optimal placement involves only looking at the placement of every pair of requests relative to each other.

The number of crossings of an optimal assignment for a request sequence is the *number of unavoidable crossings* of the request sequence. The difference between the number of crossings incurred by an algorithm A and the number of unavoidable crossings is consequently the *number of avoidable crossings* of A on that request sequence.

Consider a 2-regular instance for online slotted OSCM-k with slots $S = \{s_1, \ldots, s_n\}$, vertices $V = \{v_1, \ldots, v_n\}$ and a request sequence $R_1, \ldots, R_n$. Let us assume that at some point after the k-th request has been satisfied by algorithm A, there are satisfied slots and the vertices in V have degree 2, 1 or 0, depending on how many times these vertices have appeared in requests. Since we know that the final graph will be 2-regular, for those vertices in V with degree less than two we are still expecting a request that contains each of them, and for any unsatisfied slot there will be a request which will be satisfied using this slot.

Figure 2.4: Propagation arrows before the first request and after part of the instance is satisfied.

Intuitively, we use propagation arrows to greedily match unsatisfied vertices to available slots in a way that minimizes the number of crossings. We can see this in Figure 2.4. For instance, in an empty graph every vertex v_i in V will have two propagation arrows to the slot s_i, but once some slots are occupied we take the leftmost vertex with degree less than two and assign a propagation arrow to the leftmost unsatisfied slot. We know that the instance is 2-regular, so for two missing edges of vertices in V there must be an empty slot. We can define the propagation arrows formally as follows.

First, we know that after k requests for a 2-regular graph, there are $n - k$ unsatisfied requests, which corresponds to $2(n - k)$ missing edges. We will double count the missing edges with the following two lists.

The *list of unsatisfied vertices* L_V of an instance after the k-th request is an ordered list that contains every vertex $v_i \in V$ from smallest to largest at most twice. L_V will contain no copies of a vertex $v_i \in V$ if it already has appeared twice in the request sequence $R_1, \ldots, R_k$, i.e., if v_i has degree 2, L_V will contain $v_i \in V$ once if v_i has appeared only once in $R_1, \ldots, R_k$, i.e., if v_i has degree 1 in the partially satisfied graph, finally, L_V contains a vertex v_i twice if v_i does not appear in $R_1, \ldots, R_k$, and thus has degree 0 at that point.

Thus, we can analogously consider the *list of unsatisfied slots* L_S as an ordered list that contains each unsatisfied slot twice, again from smallest to largest. From the previous observation it should be clear that $|L_V| = |L_S|$.

Definition 3. *Consider a* 2*-regular instance for online slotted OSCM-k with slots* $S = \{s_1, \ldots, s_n\}$*, vertices* $V = \{v_1, \ldots, v_n\}$ *and a request sequence* $R_1, \ldots, R_n$*. Let A be an algorithm that has satisfied k requests. Let us consider the corresponding* L_V *and* L_S *for this request. There is a* propagation arrow *from vertex v to slot s if both occupy the same place in the ordered lists* L_V *and* L_S*, i.e., if v is the i-th element of* L_V *and s is the i-th element of* L_S *for some* $i \leq 2(n - k)$*.*

Observe that propagation arrows do not cross one another by construction. So if we count the crossings of a partial graph including the crossings between graph edges and propagation arrows, we have a lower bound on

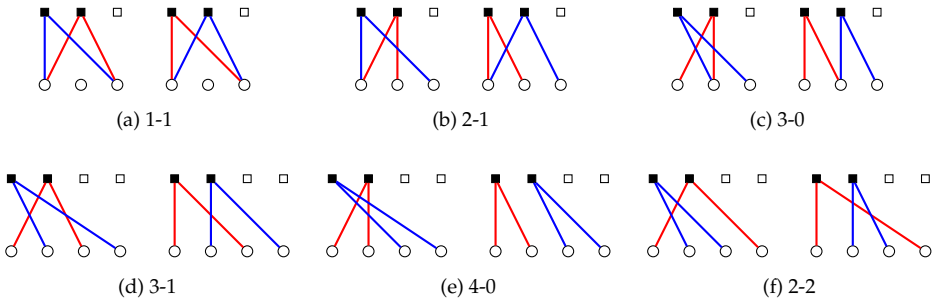

Figure 2.5: Case distinction for step one of Lemma 1. Each case is depicted before and after the untangling. The request s_x is drawn in red and s_y in blue.

the number of crossings that the graph will have after the request sequence is completely satisfied.

In the following, we observe that an instance is optimally solved if and only if for every pair of requests, the relative order of their slot assignments is optimal, i.e., if the placement of these two requests is such that there are fewer crossings between them than otherwise. This basically means that a crossing is unavoidable if and only if the relative order of the two requests involved in this crossing is optimal, regardless of any other placement of any other request within the graph. This provides us with a very powerful tool to analyze the performance of online algorithms solving online slotted OSCM-2 on 2-regular graphs.

In order to prove the aforementioned statement, we first need the following lemma.

Lemma 1. *Let two requests $R_x = \{x_1, x_2\}$ and $R_y = \{y_1, y_2\}$ be assigned to slots s_x and s_y. Without loss of generality assume that $x_1 \leq y_1$ and $x_2 \leq y_2$. An assignment where $s_x < s_y$ generates fewer or equally many crossings in the final graph than an assignment where $s_y < s_x$ if every other assigned slot remains unchanged.*

Proof. We separate this proof into two steps. In the first step, we show that the number of crossings between the two requests is always the same or smaller if $s_x < s_y$. This can be done with a case distinction as depicted in Figure 2.5.

Now for the second step, we need to show that the number of crossings in an overall graph is still smaller or equal if $s_x < s_y$. We prove it by means of a contradiction. Let us consider a 2-regular graph G for which we have two such requests $\{x_1, x_2\}$ and $\{y_1, y_2\}$ and an assignment where $s_y < s_x$, with a total number of crossings c_G. Now let us consider the graph G' which is the same as G except that the placement of $\{x_1, x_2\}$ and $\{y_1, y_2\}$

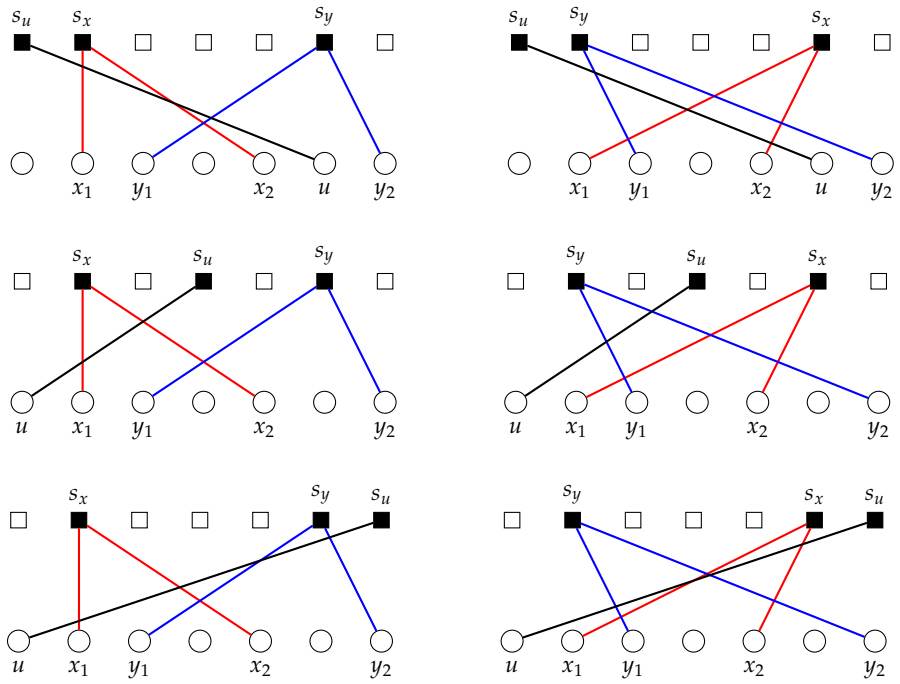

Figure 2.6: As shown in Lemma 1, there cannot be a crossing between R_x and an edge (u, s_u) that makes the ordering $s_y < s_x$ better than $s_x < s_y$.

is exchanged, making $s_x < s_y$, with a number of crossings $c_{G'}$. Assume that $c_G < c_{G'}$. We already know that this is not due to the number of crossings between edges to s_x and s_y, as such a case would be covered by Figure 2.5. Without loss of generality assume therefore that (one of) the extra crossing(s) in G' is between some edge (u, s_u) and one of the modified edges (x_t, s_i) with $t \in \{1, 2\}$.

In order for this pair of edges to produce an extra crossing in G' compared to G at all, we know that $u \notin [x_1, x_2]$, as otherwise this crossing is unavoidable and thus the same in G and G'. We make a case distinction over the remaining cases, which we depict in Figure 2.6.

Assume now that $x_2 < u$. Then, $s_u < s_x$ in order for the edges to cross at all. This positioning produces two crossings with R_x in G and possibly some crossings with R_y. However, since R_x is only assigned further to the right in G', we get the exact same number of crossings between (u, s_u) and the edges of R_x and R_y in G'.

Assume finally that $u < x_2$. Then, $s_x < s_u$ in order for the edges to cross at all. This positioning produces two crossings with R_x in G and possibly some crossings with R_y. We do a case distinction on whether $s_u < s_y$ or $s_y < s_u$.

If $s_u < s_y$, then R_y and R_x simply "change roles" in G' compared to

G and the number of crossings remains the same. If $s_y < s_u$, then (u, s_u) crosses the edges of R_x and of R_y completely in both G and G'.

Thus, by swapping the slot assignment in this way one cannot reduce the number of crossings in the overall graph. □

Lemma 1 plainly states that for each pair of requests, the optimal ordering gives the leftmost request a slot that is to the left of the slot assigned to the rightmost request. In this instance the notion of left and right requests only means that if the requests are not for identical pairs of vertices, the left request contains the left-most distinguished vertex.

In order to find an upper bound on the competitive ratio, we only have to observe that any pair of requests is either placed optimally or otherwise bound the number of crossings generated by that pair with the number of unavoidable crossings in the optimal solution.

2.5 Upper Bound for 2-Regular Graphs

With these structural properties we are ready to present the algorithm that will provide us with an upper bound of 5 for the competitive ratio.

Neglecting to take the state of the graph into account when making decisions regarding the insertion of requests seems to result in relatively bad upper bounds. As an example, we take the simple barycenter algorithm (Algorithm 1) proposed in [44], which optimally solves the offline OSCM-2. This algorithm computes the average between the two requested vertices and assigns it to this particular point. In the case of the slotted version of the problem, we have to adjust it to take the closest free slot.

Algorithm 1 is no better than 8-competitive, as the following simple example illustrated in Figure 2.7 shows. If we request the sequence $\{x_{n-1}, x_n\}$, $\{x_{n-1}, x_n\}, \{x_{n-3}, x_{n-2}\}, \{x_{n-3}, x_{n-2}\}, \ldots, \{x_1, x_2\}, \{x_1, x_2\}$, the two first requests are placed on slots s_{n-1}, s_{n-2}, and then the following requests consecutively occupy slots to the left of those until the last request, which is assigned the only available slot s_n. The last pair of edges crosses all others, resulting in roughly $4n$ crossings compared to $\frac{n}{2}$ crossings in the optimal case.

Algorithm 1 Barycenter algorithm from [44], adjusted for the slotted case.

1: *free_slots* = $\{1, \ldots, n\}$;
2: **for** $\{x_1, x_2\}$ in *input* **do**
3: $s := \lfloor \frac{x_1 + x_2}{2} \rfloor$;
4: **while** *s.isUsed()* **do**
5: $s := \{ t \mid argmin_{\forall t \in S. \neg t.isUsed()}(t - s) \mid \}$ // take leftmost on tie
6: Assign $\{x_1, x_2\}$ to s;

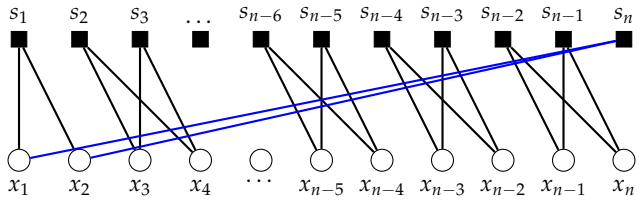

Figure 2.7: The request sequence is $\{x_{n-1}, x_n\}, \{x_{n-1}, x_n\}, \{x_{n-3}, x_{n-2}\}, \{x_{n-3}, x_{n-2}\}, \ldots, \{x_1, x_2\}, \{x_1, x_2\}$. The edges of the last request cross all others, resulting in roughly $4n$ crossings compared to $\frac{n}{2}$ crossings in the optimal case.

Algorithm 2 Chooses in each step the insertion with the lowest number of additional edge-edge and edge-propagation arrow crossings.

```
1:  free_slots = {1, ..., n};
2:  for element in input do
3:      least_crossings := ∞;
4:      best_slot := 0;
5:      for slot in free_slots do
6:          G.simulate_node_insertion(slot, element);
7:          new_crossings=G.edge_edge_crossings() + G.edge_prop_crossings();
8:          if new_crossings < least_crossings then
9:              least_crossings = new_crossings;
10:             best_slot = slot;
11:         G.revert_simulated_insertion(slot, element);
12:     G.insert_node(best_slot, element);
13:     free_slots := free_slots \ best_slot;
```

In order to achieve a good upper bound for the OSCM-2, we present Algorithm 2 that given a request selects the slot which minimizes the total number of crossings – including crossings between edges and propagation arrows – among all available slots.

Note that analyzing an algorithm in this setting is not completely trivial. Our approach is to show that the types of crossings between two requests produced by our algorithm are "good-natured". Specifically, we look at pairs of requests for which the crossings can be avoided entirely if they are appropriately ordered, i.e., 3-0 or 4-0 crossings as depicted in Figure 2.5 (c) and (e) respectively. This type of crossing is then either not produced by Algorithm 2 or we can show that a number of unavoidable crossings is necessary to produce this configuration. With this, we can then upper bound the competitive ratio. Note that this relatively rough estimate is most likely an overestimation of the actual competitive ratio of the algorithm, but even such an estimate already requires a deep structural analysis.

First, we present some lemmata outlining some relevant structural properties of assignments made by Algorithm 2, then we consider each type of critical crossing, 4-0 crossings and then 3-0 crossings and finally we show that the competitive ratio is still bounded when these types of crossings appear.

2.5.1 Structural Properties

To start the analysis of Algorithm 2 we first make a few observations on the changes of the propagation arrows after a request is satisfied.

Consider a request $\{x_1, x_2\}$, which is assigned to slot s_x by some algorithm. Before this request arrived, there were two propagation arrows from vertices y_1 and y_2 going to slot s_x (note that it is possible that $y_1 = y_2$). After the request is assigned to s_x the propagation arrows pointing to s_x have to be shifted, because the slot is not available anymore. Simultaneously, one propagation arrow of each x_1 and x_2 disappears as the request is satisfied. The remaining propagation arrows have to reflect this movement out of s_x and into the two empty positions left by x_1 and x_2, and they do so in the following way.

Observation 1. *Let $R = \{x_1, x_2\}$ be a request assigned to slot s_x. And let $y_1 \leq y_2$ be the vertices (or vertex) whose propagation arrows point to s_x before this request arrived. Only propagation arrows connected to nodes between the leftmost vertex of x_1 and y_1 and the rightmost vertex of x_2 and y_2 will be shifted.*

Observe that there are no propagation arrows connected to vertices between y_1 and y_2, since these would be connected to slots other than s_x and produce crossings between propagation arrows, which is impossible by definition. Figure 2.8 demonstrates this observation.

Proof. Let t be the number of propagation arrows attached to vertices in the interval between the leftmost vertex of x_1 and y_1 and the rightmost vertex of x_2 and y_2 before R is assigned to s_x. After the placement, the number of

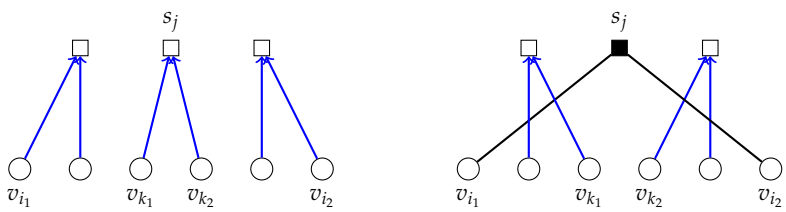

Figure 2.8: Schematic diagram showing how propagation arrows shift after a placement.

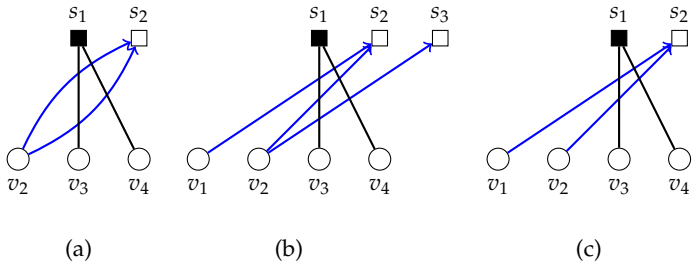

Figure 2.9: Types of crossings avoided by Algorithm 2, as mentioned in Lemma 2. The propagation arrows are drawn in blue and the edges already present in the graph are drawn in black.

propagation arrows in x_1 and x_2 is reduced by 1. The number of slots that require two arrows has been reduced by 1 in s_x. If $t = 2$, then $x_1 = y_1$, $x_2 = y_2$ and the interval has no remaining propagation arrows, after R is placed. Otherwise, each available slot has to be matched to each available vertex and no additional propagation arrows from outside the interval are required because the placement of R removes two propagation arrows and one slot. □

While Observation 1 is not specific to Algorithm 2, we can use it in the subsequent proofs. We continue with a lemma that allows us to shorten many case distinctions in the following proofs.

Lemma 2. *There is no instance such that after some request two propagation arrows connected to a slot s_2 cross both edges adjacent to a satisfied slot s_1 when using Algorithm 2.*

Alternatively, the situations depicted in Figure 2.9 will never occur if one uses Algorithm 2.

Proof. We prove the lemma by contradiction and assume that after Algorithm 2 satisfied some arbitrary request $R_x = \{x_1, x_2\}$ there are two propagation arrows crossing edges (v_3, s_2) and (v_4, s_2). Figure 2.9 shows three different situations how these crossings can occur: (a) Either both propagation arrows are connected to a single node $v_2 < v_3$ that cross the edges of s_1, (b) there is a propagation arrow from two nodes $v_1 < v_2$ crossing the nodes of s_1 to a slot s_2 with $s_1 < s_2$ and another edge from v_2 to a slot s_3 with $s_1 < s_2 < s_3$ or (c) there are the edges of (b) without the additional edge (v_2, s_3). Cases (a) and (b) are very similar, but in case (a) both propagation arrows from v_2 go to the same slot, whereas in case (b) they are split between s_2 and s_3. In case (c) the vertex v_2 is already assigned an edge, thus it only has one remaining propagation arrow. We ignore this already

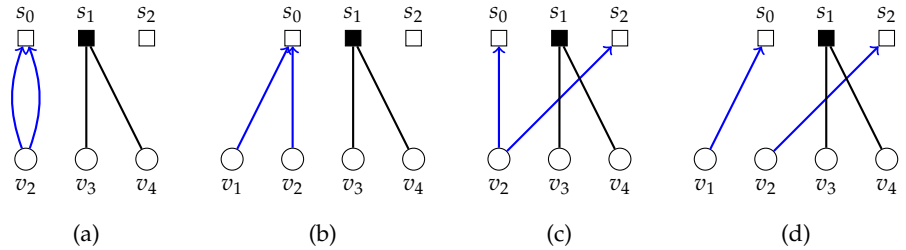

Figure 2.10: Possible configuration before the request R_x is added and the propagation arrows are shifted to s_2. The propagation arrows are drawn in blue and the edges already present in the graph are drawn in black.

present edge, as its precise nature makes no difference for the following argumentation.

We assume that the propagation arrows from v_2 (and possibly v_1) are the first ones that cross the edges of s_1 as described in the lemma after request R_x has been satisfied. It is possible that there are vertices between v_1 and v_2 or between v_2 and v_3. However, if these vertices exist, they cannot have propagation arrows. Otherwise, v_2 (and possibly v_1) would not be responsible for the first two propagation arrows that cross s_1, but the propagation arrows of these other nodes. We look at the first request R_x whose assignment results in such a structure and how the graph looked like before serving R_x.

Note that every slot has two propagation arrows pointing to it and after assigning a request to this slot, the propagation arrows pointing to that slot move to a neighboring free slot. Thus, there are four different configurations possible before the request R_x is satisfied, presented in Figure 2.10: (a) Both arrows are connected to a single node $v_2 < v_3$ and a slot $s_0 < s_1$, (b) the two arrows are from different nodes v_2 and v_1 and are connected to a slot $s_0 < s_1$, (c) both arrows are connected to v_2, one of them pointing to s_0 and one to s_2 with $s_0 < s_1 < s_2$ (d) the two arrows are from different nodes v_1 and v_2, one of them points to s_0 and one to s_2 with $s_0 < s_1 < s_2$.

We know also by using Observation 1 that the assignment of R_x will only shift the propagation arrows around s_1 if these arrows are part of the affected interval between the vertices of R_x and the propagation arrows pointing to the slot assigned to R_x.

Cases (a) and (b) have no previous arrows crossing with edges of s_1, thus, they require that two propagation arrows are shifted to the right-hand side. As we saw in Observation 1, this can only happen if R_x is assigned to s_0 or to the left hand side of it and the vertices x_1 and x_2 are both to the right of v_2. The propagation arrows of x_1 and x_2 pointed previously to a slot to the right of s_0, thus, assigning R_x to s_0 will result in the two propagation

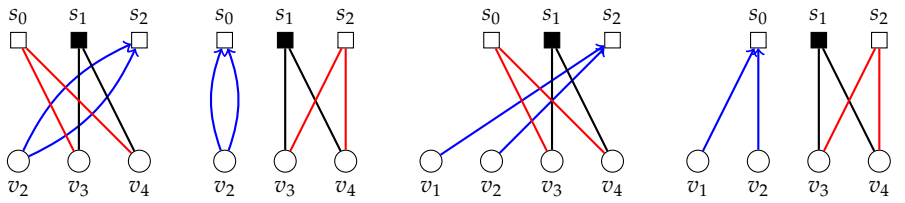

Figure 2.11: Comparing the crossings of assigning R_x to the left or right hand side of s_1 for the cases (a) and (b) from Figure 2.10. The propagation arrows are blue, already present edges are black and the newly introduced edges, adjacent to the recently fulfilled request R_x, are red.

arrows previously pointing to s_0 shifting to the right to fill up the slots left by the missing propagation arrows of x_1 and x_2. These gaps are filled from the left hand side, which results in the two crossing propagation arrows shown in Figure 2.9.

Assume that Algorithm 2, given case 2.10(a) or (b) and a request R_x with $v_2 < x_1 < x_2$, assigns R_x to a free slot to the left-hand side of s_1. Figure 2.11 shows that assigning R_x further to the right results in fewer crossings, which is a contradiction to the procedure of the algorithm itself. If the vertices x_1 and x_2 do not coincide with v_3 and v_4, they are even further to the right-hand side. If this is the case, we get even more crossings if R_x is assigned to the left-hand side of s_1.

For the cases (c) and (d) from Figure 2.10 only one propagation arrow needs to be pushed to the right-hand side. Thus, without loss of generality only x_2 has to be to the right-hand side of v_2 and the position of x_1 is arbitrary. Either x_1 is a vertex to the left-hand side of v_2 (it is even possible that $x_1 = v_1$) or it is to the right. The latter case is equivalent to the cases (a) and (b) in the sense that two more propagation arrows will cross over s_1 and a placement to the right of s_1 will result in fewer crossings as we saw in Figure 2.11.

In the first case x_1 is to the left of v_2, and we push only one more propagation arrow to the right-hand side of s_1. Figure 2.12 shows that choosing the position s_2 to the right of s_1 results in fewer crossings. Just as with cases (a) and (b) we can assume that x_2 is the leftmost possible vertex to the right of v_2 and otherwise the number of avoided crossings with the placement to the left of s_1 only grows.

Thus, Algorithm 2 will not place a request such that two edges of a slot are crossed by two propagation arrows. □

Lemma 2 forbids specific configurations of the propagation arrows during the course of applying Algorithm 2 to a request sequence. The following lemma uses a counting argument to guarantee that a specific request

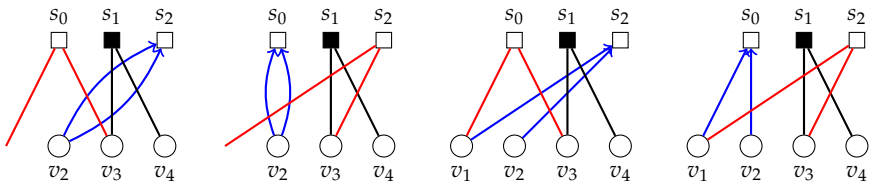

Figure 2.12: Comparing the crossings for assigning R_x to the left or right-hand side of s_1, for the cases (c) and (d) from Figure 2.10. The propagation arrows are blue, already present edges are black and the newly introduced edges, adjacent to R_x, are red.

between two (far apart) vertices must eventually appear in a specific setting. Such requests (from vertices that are far apart) always guarantee the appearance of unavoidable crossings as depicted in Figure 2.5 (f). The appearance of such requests guarantees in later proofs the existence of such unavoidable crossings, which can be counted in a way that bounds the competitive ratio.

Lemma 3. *Let there be two requests $\{x_1, x_2\}$ and $\{y_1, y_2\}$ that are assigned to slots s_x and s_y, with $x_1 < x_2 < y_1 < y_2$ and no free slot between s_x and s_y. If there are two neighboring vertices u, v, with $x_2 \leq u < v \leq y_1$ and propagation arrows pointing to two different slots s_l, s_r, with $s_l < s_x < s_y < s_r$, and the request $\{u, v\}$ appears, then there must be a future request $\{a, b\}$, with $a \leq x_2$ and $y_1 \leq b$, which unavoidably crosses all edges of u and v. This future request $\{a, b\}$ is denoted as a* housing *request.*

Figure 2.13 depicts the situation described in the statement of Lemma 3.

Proof. Our proof is a simple counting argument. The request $\{u, v\}$ removes two propagation arrows. One points to the left of the filled block between s_x and s_y and the other points to its right. Depending on its placement the request pushes one propagation arrow from one side of the satisfied block between s_x and s_y to the other.

Without loss of generality we assume that $\{u, v\}$ is placed on s_l. The second propagation arrow pointing to s_l comes from x_2 (if $u \neq x_2$) or a vertex even further left. It is not possible that it comes from a vertex between x_2 and v due to Lemma 2. When the request $\{u, v\}$ is placed, it pushes this second propagation arrow to the slot s_r. This propagation arrow represents a mismatch between open slots and remaining edges. The number of remaining edges to the left of u and to the right of v is odd, but the slots always consume two of these remaining edges. This has to be compensated by some request $\{a, b\}$ that is placed right of s_y, where a is to the left hand

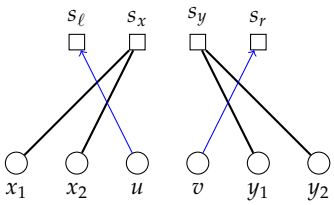

Figure 2.13: Sketch of the situation described in the statement of Lemma 3.

side of u and b is to the right-hand side of v. This request crosses all edges of u and v. □

Where Lemmas 2 and 3 are applicable for specific configurations, the following lemma provides a tool giving a set of edges or propagation arrows which are necessary to make a local configuration (e.g., a crossing of two requests) feasible in the context of the remaining graph.

Lemma 4. *For every edge or propagation arrow, starting at a vertex v_i of V and pointing to a slot s_j with $i < j$ (analogously $j < i$), there is one edge or propagation arrow pointing from a vertex v_k to a slot s_l with $i < k$ and $l \leq i$ (analogously $k < i$ and $i \leq l$).*

Proof. We use a simple handshake argument and count the already present edges and the propagation arrows in the graph to prove the statement.

At first, we separate the vertices into four sets as depicted in Figure 2.14. The set A contains the vertex s_l and all vertices from S that are to the left-hand side of s_l. The set B contains the vertices from S that are to the right-hand side of s_l. The set C contains the vertex v_i and all vertices from V that are to the left-hand side of v_i. The last set, called D, contains the vertices from V that are to the right-hand side of v_l.

The vertices in the set A have two incident edges or two incident propagation arrows. These edges or propagation arrows start at a vertex in C or

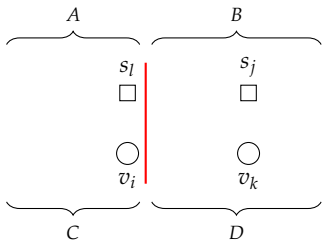

Figure 2.14: Depending on the vertex v_i, the vertices are split into four sets, A, B, C and D.

in D. We denote the set of edges that connect a vertex from A with a vertex from C as E_{AC}. Analogously, we define the edge set E_{AD}. We also split the propagation arrows starting at the vertices from V and ending at a vertex in A, into two sets P_{AC} and P_{AD}. We observe that

$$2|A| = E_{AC} + E_{AD} + P_{AC} + P_{AD} \tag{2.1}$$

must always be true.

Additionally, the sum of the edges and propagation arrows starting at a vertex in C must be $2i$. More formally,

$$2i = E_{AC} + E_{BC} + P_{AC} + P_{BC} \, . \tag{2.2}$$

The number of vertices in the set A must be i, because we choose the vertex v_i at position i as a reference point to define the set A. Thus, we can combine Equation (2.1) and Equation (2.2) to obtain

$$0 = E_{AD} + P_{AD} - E_{BC} - P_{BC} \, .$$

Note that, since propagation arrows never cross each other, either P_{AD} or P_{BC} is empty (it is also possible that both are empty). Thus, every edge or propagation arrow crossing from one side to the other is compensated by an edge crossing into the other direction. □

With our structural properties and observations regarding the propagation arrows we can now start to analyze the critical crossings depicted in Figure 2.5 (e) and (c). These crossings are critical in the sense that they contain only avoidable crossings and no unavoidable ones. So, they decrease the performance of our algorithm and do not guarantee a constant competitive ratio like the other crossings depicted in Figure 2.5,which have at least one unavoidable crossing. In the following sections we overcome this problem by showing that for each of these critical crossings there must exist some other request that unavoidably crosses one of the requests involved in the critical crossing.

2.5.2 The 4-0 Crossings

Recall that due to Lemma 1 the optimal solution for a 2-regular instance of the online OSCM-2 problem consists on minimizing crossings between every pair of requests. Thus, we can look at a pair of requests and exhaustively classify them as depicted in Figure 2.5 and then analyze the competitive ratio of an algorithm depending on how many of these types of crossings appear. In particular, if no 3-0 crossings (Figure 2.5(c)) or 4-0 crossings (Figure 2.5(e)) were produced by an algorithm, the algorithm would be 3-competitive at worst, as any sub-optimal placement would be trivially compensated by at least one unavoidable crossing. Thus, in order

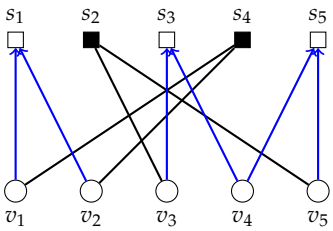

Figure 2.15: A 4-0 crossing with a slot in between. These types of crossings are forbidden by Lemma 5.

to analyze the competitive ratio of Algorithm 2, we only have to look at 3-0 and 4-0 crossings.

Using Lemma 2 we can now prove that Algorithm 2 will not make too many mistakes when producing 4-0 crossings. First we prove that Algorithm 2 will never produce 4-0 crossings with gaps, i.e., unsatisfied slots between the 2 slots generating the 4-0 crossing as depicted for instance in Figure 2.15.

Lemma 5. *Algorithm 2 never generates 4-0 crossings with gaps in between. More precisely, for each pair s_i, s_j with $i < j$ assigned by Algorithm 2 that generate a 4-0 crossing, every s_k with $i < k < j$ is already taken.*

Proof. Let us assume that there are no 4-0 crossings with gaps in the graph. We prove this lemma by means of a contradiction.

Let $\{v_1, v_2\}$ be the request assigned to slot s_i by Algorithm 2, and a new request $R = \{v_3, v_4\}$ is made where $v_1 < v_2 < v_3 < v_4$ without loss of generality.

Let s_j be a slot to the left of s_i with the smallest crossing values for R and let s_k be the leftmost empty slot between s_j and s_i.

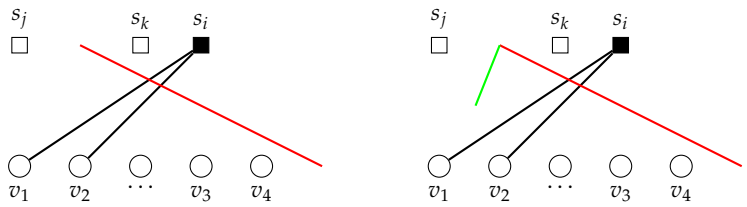

Figure 2.16: Counting crossings in a 4-0 crossing with a gap in Lemma 5. Only edges like the one depicted in red will make a placement in s_k more favorable than a placement in s_j. However for every red edge a green edge must exist or the graph would not be free of 4-0 crossings.

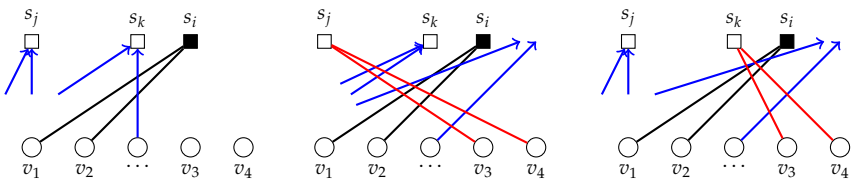

Figure 2.17: At most one propagation arrow crosses from the left of v_2 to s_k by Lemma 2, as discussed in the proof of Lemma 5.

The only crossings that would make a placement in s_k more unfavorable than a placement in s_j are edges coming from the right of v_4 to a slot between s_j and s_k as depicted in the left of Figure 2.16. There cannot be any propagation arrows of this kind as we assume that all the slots between s_j and s_k are full.

For any edge coming from a vertex v_{t_1} to the right of v_4 into slot s_t with $j < t < k$ there must be another edge coming from a vertex v_{t_2} to the left of (or directly from) the vertex v_2. Otherwise we would have a 4-0 crossing with an empty slot, namely $v_1 < v_2 < v_{t_2} < v_{t_1}$ and the slots $s_t < s_k < s_i$, which would be a contradiction to the assumption that this is the first occurrence, as depicted in Figure 2.16. This means that $v_{t_2} \leq v_2$. However, in this case the edge v_{t_2} generates crossings only for the assignment of R to s_j and not for the assignment to s_k, which means that for every crossing counting for s_k there is at least one crossing counting for s_j.

Finally we are only left with counting the crossings for the propagation arrows going to s_j with the placing in s_k and vice-versa as depicted in the three drawings of Figure 2.17.

Before we assign the request $\{v_3, v_4\}$, we know that due to Lemma 2 only one propagation arrow can cross from the left of (or directly from) v_2 to the slot s_k. Thus, when assigning the request to the slot s_k there are no extra crossings for the propagation arrows going to s_j. However, if we assign the request to slot s_j, the propagation arrows assigned to s_j will now be transferred to s_k as we saw in Observation 1, creating four new crossings between these propagation arrows and the new edges. We have just seen that the placement in s_j generates more crossings than the placement in s_k which contradicts our assumption that s_j has the smallest crossing values. □

Next we prove that Algorithm 2 only generates 4-0 crossings when they are forced or in a very specific configuration. We do so with two different lemmas. If we have a request for a pair of vertices, such that every available slot generates at least one 4-0 crossing, we call this crossing a forced 4-0 crossing. Observe that it is possible that more than one 4-0 crossing is forced by the same request (see Figure 2.18).

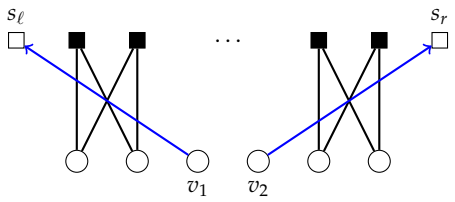

Figure 2.18: More than one 4-0 crossing might be forced by the same request

Lemma 6. *If Algorithm 2 is used, for every forced 4-0 crossing there is at least one uniquely identifiable and unavoidable crossing.*

Proof. We will prove this using Lemma 3. If a request $\{v_1, v_2\}$ arrives in time step t and every possible placement generates a 4-0 crossing, the propagation arrows of v_1 and v_2 must point to two different slots before the request is served due to Lemma 2. We assume $v_1 < v_2$ and call the slot on the left-hand side s_ℓ and the other one s_r, as sketched in Figure 2.18.

We denote the set of edges, which are crossed by the request $\{v_1, v_2\}$ when it is placed in s_ℓ, with L. Analogously, we define the set of crossed edges R for the slot s_r. To be precise, the set L contains the edges (s_j, v_i) with $s_\ell < s_j < s_r$ and $v_i < v_1$ (see Figure 2.19). Our algorithm will always choose the slot which results in the least amount of crossings. Therefore, if our algorithm chooses (without loss of generality) the slot s_ℓ, we know that positioning the request in slot s_r results in at least the same number of crossings. It follows that $|L| \leq |R|$ (or $|L| \leq |R| + 1$ if the other edge connected to v_2 is placed between s_ℓ and s_r but the edge connected to v_1 is not).

Since we are in the situation of a forced 4-0 crossing, there are at least two edges in each set L and R that belong to the same request. We look at

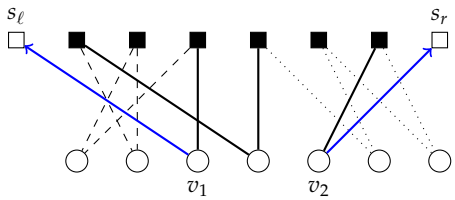

Figure 2.19: The set of edges L is dashed, and the set of edges R is dotted. The request $\{v_1, v_2\}$ will generate two crossings per dashed edge if positioned in s_ℓ and two crossings per dotted edge respectively if positioned in s_r. There is one unavoidable crossing for the edges going to vertices between v_1 and v_2 no matter the positioning.

the pair of edges (s_i, v_{i_1}) and (s_i, v_{i_2}) in L, with $v_{i_1} < v_{i_2}$ with the smallest possible v_{i_2}, respectively the pair of edges (s_j, v_{j_1}) and (s_j, v_{j_2}) in R, with $v_{j_1} < v_{j_2}$ and the largest possible v_{j_1}. Applying Lemma 3, it follows that there will be a future request between at least v_{i_2} and v_{j_1}, meaning that a future *housing* request will cross at least one of the edges – in L and R respectively – of every pair generating a 4-0 crossing except at most one. Observe that due to Lemma 2 there cannot be any available edge slot between v_{i_2} and v_{j_1} other than v_1 and v_2, which will matter later in the proof.

Note that every request where only one edge is in L (or R) unavoidably crosses the request $\{v_1, v_2\}$ anyway. Thus, at least $\frac{|L|}{2} + \frac{|R|}{2}(+1)$ edges are unavoidably crossed after the request in Lemma 3. In this case, the unavoidable crossing between the *housing* request and v_1 and v_2 compensate for the potentially missing crossings with the edges from v_{i_2} and v_{j_1}. In other words, for every slot between s_ℓ and s_r there is at least one edge which is unavoidably crossed, apart from the aforementioned exceptions.

The request $\{v_1, v_2\}$ crosses all edges in L twice if it is placed in s_ℓ. Thus, the number of avoidable crossings is $2|L|$, which is at most twice as large as the number of unavoidable crossings $\frac{|L|}{2} + \frac{|R|}{2}(+1) \geq |L|$.

The argument above works if there is only one forced 4-0 crossing for a set of requests before the *housing* request from Lemma 3 appears. In the following we discuss why we can assign a uniquely identifiable unavoidable crossing for each set of potentially overlapping forced 4-0 crossings. Overlapping 4-0 crossings appear when both involved requests in a 4-0 crossing are again completely crossed by another third request (see Figure 2.20).

We call the request that generates the first forced 4-0 crossing after its placement $\{v_1, v_2\}$. The algorithm had the decision to place it in the left slot s_ℓ, crossing $|L|$ edges, or in the right slot s_r, crossing $|R|$ edges.

Without loss of generality assume that the request $\{v_1, v_2\}$ was placed in s_ℓ. In order to have an overlapping 4-0 request, we assume that the request of the second forced 4-0 crossing $\{v_3, v_4\}$ is to the left of the request $\{v_1, v_2\}$. Because this request is also a forced 4-0 crossing, it can be placed in

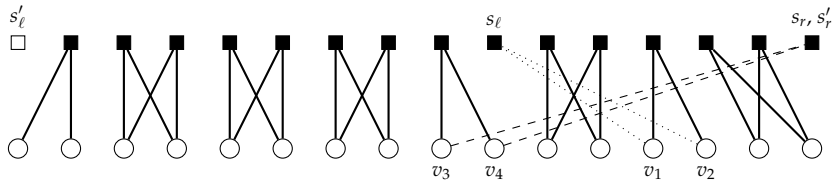

Figure 2.20: Two sets of 4-0 crossings overlap each other before a *housing* request appears. The first one is drawn with a dotted pair of edges and the second one is dashed. This particular scenario is not completely realistic for Algorithm 2 but could happen if the overall graph is larger.

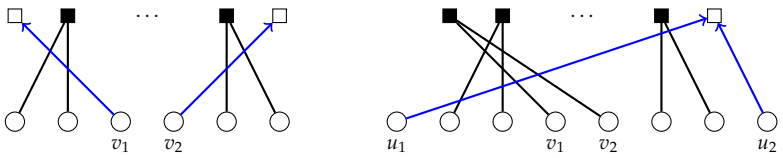

Figure 2.21: If there is no available slot without a 4-0 crossing the propagation arrows point to different sides and a request u_1, u_2 must eventually exist.

a slot to the left $s'_\ell < s_\ell$ or a slot to the right $s'_r \geq s_r$. The slots must be further left (respectively, further right) because when the request $\{v_1, v_2\}$ arrives, all of the other vertices between v_{i_2} and v_{j_1} must be filled (as we already argued), thus v_3 and v_4 are to the left of v_{i_2}. Combined with the assumption that the second 4-0 crossing is forced, we get the restricted position for v_3 and v_4.

Analogous to the previous case, let L' be the set of edges (v, s) with $v < v_3$ and $s'_\ell < s < s'_r$ and let R' be the set of edges (v, s) with $v > v_4$ and $s'_\ell < s < s'_r$. Note that the edges of the first 4-0 request are now part of R'. As already explained above, if the algorithm decides to place the request in s'_r, this implies that $|R'| \leq |L'|(+1)$ holds. Moreover, $|R'| \geq \frac{|L|}{2} + |L|$ holds because $v_4 < v_{i_2}$, which by definition means that half of the edges of R are to the right of v_4 and thus part of R'. Applying Lemma 3 again, it follows that there will be a future request that unavoidably crosses at least

$$\frac{|L'|}{2}(+1) + \frac{|R'|}{2} \geq \frac{|R'|}{2} + \frac{|R'|}{2} \geq \frac{|R'|}{2} + \frac{|R|}{4} + \frac{|L|}{2}$$

edges. The number of avoidable crossings is $2|L| + 2|R'| \leq |L| + |R| + 2|R'|$, which if divided by 4 for each possible 4-0 crossing, means that

$$\frac{|L|}{4} + \frac{|R|}{4} + \frac{|R'|}{2} \leq \frac{|R'|}{2} + \frac{|R|}{4} + \frac{|L|}{2} .$$

Thus, for each of the avoidable 4-0 crossings we have at least one uniquely identifiable unavoidable crossing. Observe that we can iterate this argument for every possible overlapping 4-0 crossing. Moreover, if in the second case the request $\{v_3, v_4\}$ was placed in s'_ℓ, the analogous counting argument still holds. □

We just proved that forced 4-0 crossings imply one additional unavoidable crossing when using Algorithm 2. This means that we can consider 4-0 crossings as if they were 5-1 crossings instead. However, this is insufficient, there can be 4-0 crossings produced by Algorithm 2 that are not forced. In the following lemma we prove that non-forced 4-0 crossings are

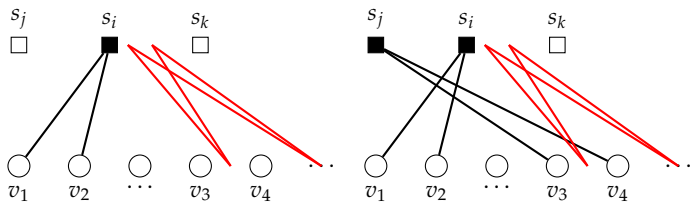

Figure 2.22: If there is more than one slot positioned like the red ones (between the slots s_i and s_k with one vertex between v_3 and v_4, and one to the right of v_4 each), Algorithm 2 may choose slot s_j generating a 4-0 crossing.

only produced by Algorithm 2 in a very specific configuration. Then we will proceed to look at the number of uniquely identifiable unavoidable crossings of that configuration.

Lemma 7. *Let there be an instance and the responses of our algorithm up to some time step i result in a graph, whose 4-0 crossings have either been forced (Figure 2.21) or were served because any alternative placement would result in two 3-1 crossings as sketched in Figure 2.22. If there is a request in time step i with an available slot, which will not generate any 4-0 crossings, it will be selected by Algorithm 2 over any slot which will generate a 4-0 crossing, unless there are two additional requests resulting in two 3-1 crossings for the alternative placement as depicted in Figure 2.22.*

Proof. Assume we have a graph in which any appearing 4-0 crossings are in the configurations of Figures 2.21 and 2.22. Let $\{v_1, v_2\}$ be a request assigned to slot s_i. Let $\{v_3, v_4\}$ be a new request with $v_2 < v_3$ without loss of generality. The new request can be assigned to a slot s_k to the right of s_i without generating new 4-0 crossings or to a slot s_j to the left of s_i as depicted in the first drawing of Figure 2.23. We can assume due to Lemma 5 that s_j is the rightmost available slot after s_i.

As we did in Lemma 5, we first count edge crossings and then the propagation arrow crossings.

In order to do so, we divide the relevant slots into two subsets. The subset X contains the slots between s_j and s_i. Recall that all the slots in this area are filled. The subset Y contains the slots between s_i and s_k, which are all filled as well. We also divide the vertices into three subsets. Any vertex to the left of v_3 belongs to subset A. Vertices between v_3 and v_4 belong to subset B and vertices to the right of v_4 belong to subset C. This division is depicted in Figure 2.23.

Only edges to slots in X or Y will generate crossings that count solely for one of the two placements. In particular any edge from C to X or Y will generate two additional crossings for the placement in s_k with respect to

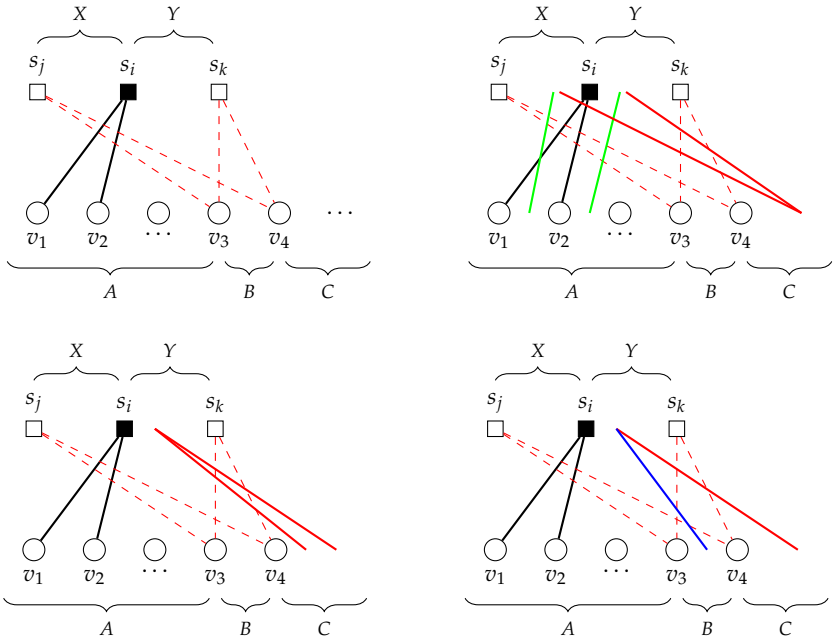

Figure 2.23: We have two possible placements for the request $\{v_3, v_4\}$. Red edges contribute extra crossings to the placement in s_k and green edges contribute extra crossings to the placement in s_j. If a slot in Y has both endpoints in C, it is a forced 4-0 crossing, but having one endpoint in B is possible.

the placement in s_j. These edges are depicted in red in the second drawing of Figure 2.23. On the other hand, any edge from A to a slot in X or Y generates two additional crossings for the placement in s_j with respect to the placement in s_k. Those edges are depicted in green in the second drawing of Figure 2.23. Finally, the edges from B to X or Y are neutral with respect to both placements. This means that we only need to analyze previously placed requests in X or Y with one endpoint in C, as these are the only ones that will make a placement in s_j more likely with respect to a placement in s_j.

We now analyze all possible requests in X with at least one endpoint in C. Recall that we assume that there is always a slot that does not force a 4-0 crossing, thus there cannot be a pair of edges from C connected to the same slot in X or Y. On the other hand, if a pair of edges from C and B respectively go to a slot X (we call this request CXB), we have a previous 4-0 crossing in the graph. Consequently, we distinguish two cases: Either the request s_i or the request CXB generated a 4-0 crossing.

If the placement in s_i generated the 4-0 crossing, we argue that there

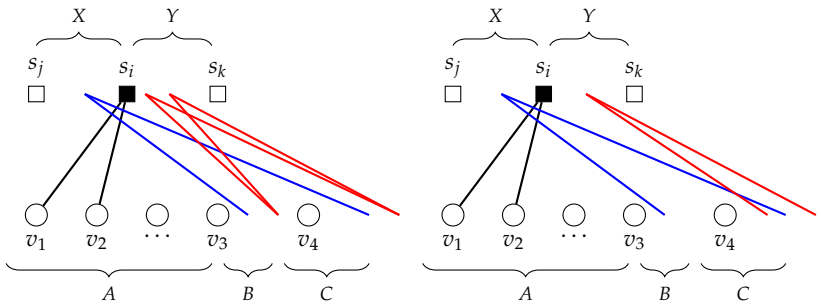

Figure 2.24: If we have a request placed in X with one endpoint in C and one in B, a 4-0 request was already present in the graph. This means that we had a situation like Figure 2.22 already with respect to that placement, and we either have a situation like Figure 2.22 with respect to v_3 and v_4 as well (left picture) or we have a forced 4-0 crossing (right picture), contradicting the assumption of Lemma 7.

was a situation comparable to Figure 2.23. If the 4-0 crossing was forced when s_i was placed, there must have been a request to the right of v_1, and it was satisfied by a slot in X, but between this request and CXB there were at least 3 propagation arrows, in particular from v_1, v_2 and v_3, so this situation is forbidden by Lemma 2. If there was a situation as in Figure 2.23 involving the request CXB and v_1, v_2, there must be two requests between v_1 and v_2 that are in region A and are satisfied in the region of X. These two requests will completely counteract the crossing contributions of the request CXB.

If the request CXB generated the 4-0 crossing, it also could not have been forced, as the propagation arrow from v_4 is between the two propagation arrows of the request and it also generates a situation forbidden by Lemma 2. Thus there must have been a situation like in Figure 2.22 involving the request CXB. In this case, as depicted in Figure 2.24, the two requests contributing to the situation in Figure 2.22 will already be present.

Finally, if a request has one endpoint in C and one in A, it follows that their crossings for the placements in s_j and s_k compensate.

We thus only consider slots in Y with at least one endpoint in C. If a slot in Y with one endpoint in C has the other endpoint in A, the number of edge crossings will be higher for the placement in s_j already. Moreover, there cannot be a slot in Y with two edges directed to C, as in the case for slots in X, this would contradict the assumption that we are not in the case of a forced 4-0 crossing, as depicted with two red edges in the third drawing of Figure 2.23. We are left with only one case: There is a fulfilled request in Y with a vertex in B and a vertex in C, as depicted in the fourth drawing of Figure 2.23. This type of request generates two extra crossings for the placement in s_k with respect to the placement in s_j. This is still not

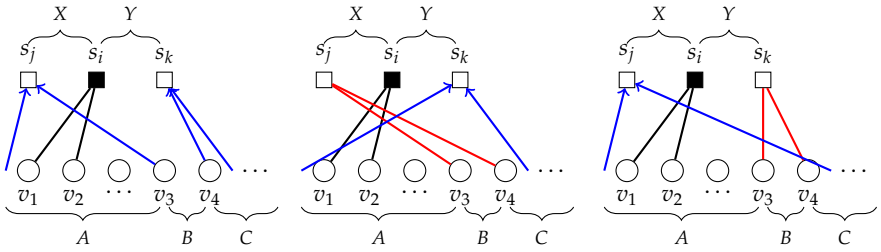

Figure 2.25: Only one propagation arrow might cross s_i due to Lemma 2.

a problem if there is only one such request, as these crossings would still be offset by the 4 extra crossings of the placement in s_j. Moreover, if there is more than one such request we have the case of Figure 2.22, where a 4-0 crossing is allowed.

Finally, we are left with counting the propagation arrow crossings. As depicted in Figure 2.25, the rightmost placement of the propagation arrows has the arrow from v_3 pointing to s_i and only the leftmost arrow from C pointing to s_k. In the second and third pictures we see what happens to these arrows after a possible s_j and s_k placement. The number of crossings due to the propagation arrows stays the same. If the propagation arrows were further left, the number of crossings in the 4-0 placement could only increase, and the number of crossings for the s_k placement could only decrease. This means that if there is at most one slot in Y with an endpoint in B and an endpoint in C, a placement in s_k is preferred. □

We now prove that the 4-0 crossings described in Lemma 7 also have uniquely identifiable unavoidable crossings, just as we did in Lemma 6 for the forced 4-0 crossings.

Lemma 8. *Any 4-0 crossing incurred by Algorithm 2 has two uniquely identifiable unavoidable crossings, because any alternative placement would result in two 3-1 crossings as sketched in Figure 2.22.*

Proof. Observe that if we only consider the crossings generated by the placement of the request generating the 4-0 crossings, we do not risk double counting unavoidable crossings. In a configuration like that depicted in Figure 2.22, where Algorithm 2 generates a 4-0 crossing, an optimal algorithm can place the same requests as depicted on the right side of Figure 2.26. The placement of the new request with a 4-0 crossing by Algorithm 2 has 6 crossings with previously placed requests (Figure 2.26 left), while the optimal placement for this request has only 2 crossings with previously placed requests (Figure 2.26 right). These crossings are unavoidable. □

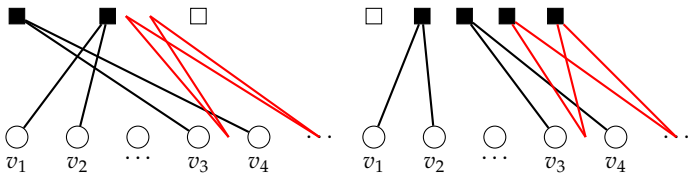

Figure 2.26: Algorithm 2 has generated a 4-0 crossing. We also depict the optimal configuration for such a situation

Using Lemmas 6 and 8, we can finally conclude that any 4-0 crossings incurred by Algorithm 2 have at least one unavoidable crossing, which leads us to the following theorem.

Theorem 3. *Forced and non-forced 4-0 crossings incurred by Algorithm 2 have at least one unavoidable crossing.*

2.5.3 The 3-0 Crossings

It remains to prove that Algorithm 2 only generates a 3-0 crossing – depicted in Figure 2.5 (c) – if there is at least one unavoidable crossing for one of the two requests that are responsible for the 3-0 crossing. In general the proofs use a case distinction in a way similar to the proofs from the previous section handling the 4-0 crossings.

As in the 4-0 case, we start by proving that Algorithm 2 never produces a 3-0 crossing with a gap.

Lemma 9. *Algorithm 2 never generates 3-0 crossings with gaps in between. More precisely, for each pair s_j, s_i assigned by Algorithm 2 with $j < i$ that generate a 3-0 crossing, every slot s_k with $j < k < i$ is already full.*

Proof. Let us assume that v_1 and v_2 is the pair of vertices adjacent to a filled slot s_i. Let v_2 and v_3 be a pair of vertices from a new request, with $v_1 < v_2 < v_3$. If Algorithm 2 creates a 3-0 crossing between the requests $\{v_1, v_2\}$ and $\{v_2, v_3\}$, it places the second request in a slot s_j with $s_j < s_i$.

We can assume that s_j is the rightmost available slot to the left of s_i, due to the following observations. If there exists a slot s'_i between s_j and s_i, we observe that the propagation arrow of v_2 cannot point to s_j, when the request $\{v_2, v_3\}$ arrives, because then the two propagation arrows pointing to s'_i have to start at vertices to the right of v_2 and cross the edges of the request $\{v_1, v_2\}$ which violates Lemma 2. This means that the propagation arrow of v_2 must point to s'_i or to a slot to the right of s_i. However, when the request $\{v_2, v_3\}$ is placed in s_j, it pushes the propagation arrows that pointed to s_j, which must come from vertices to the left of v_2, to a slot to the right of s_j, in this case s'_i, crossing the edges of the newly placed request

and violating Lemma 2 again. Thus, s_j must be the rightmost available slot to the left of s_i. □

In the following lemma we explore the situation that the 3-0 crossing happens on the edge of the graph, that is, a placement on any remaining slot causes a 3-0 crossing.

Lemma 10. *Let there be two requests $\{v_1, v_2\}$ and $\{v_2, v_3\}$ with $v_1 < v_2 < v_3$. Assume without loss of generality that Algorithm 2 creates a 3-0 crossing between these requests, with the first request for vertices $\{v_1, v_2\}$ being placed in slot s_i and during the placement of the second request there is no available slot $s_k > s_i$. Then there is at least one uniquely identifiable unavoidable crossing with the request $\{v_2, v_3\}$.*

Proof. Let us assume that v_1 and v_2 is the pair of vertices adjacent to a filled slot s_i. Let v_2 and v_3 be a pair of vertices from a new request, with $v_1 < v_2 < v_3$. If Algorithm 2 creates a 3-0 crossing between the requests $\{v_1, v_2\}$ and $\{v_2, v_3\}$, it places the second request in a slot s_j with $s_j < s_i$. Recall that due to Lemma 9, s_j must be the rightmost available slot to the left of s_i.

If there is no free slot s_k with $s_k > s_i$, then all slots to the right of s_j are filled. Moreover the propagation arrows from v_2 and v_3 are the two rightmost propagation arrows and both point to s_j when the request $\{v_2, v_3\}$ arrives. Let the number of satisfied slots to the right of s_j be t and the number of vertices that are to the right of v_2 be b. If the number of satisfied slots is larger than the number of vertices on the bottom line ($t > b$), there are at least two edges from vertices that are to the left of v_2 pointing to satisfied slots that are to the right of s_j. It is not possible that a satisfied slot to the right of s_j is adjacent to two vertices that are to the left of v_2 due to Lemma 2, because the propagation arrows of v_2 and v_3 would cross it. Thus, the second adjacent vertex must be to the right of v_2, resulting in at least one unavoidable crossing for the request $\{v_2, v_3\}$. If $t \leq b$, the slot s_j is above the vertex v_2 or to the right of it. In this case, the edge $\{v_1, s_i\}$ must be compensated for by an edge that starts to the right of v_2 and points to a slot to the left of s_j. It is not possible that the second edge of this slot also comes from a vertex to the right of v_2 (see Lemma 4). Thus, when no slot s_k exists, there must exist an edge that unavoidably crosses the request $\{v_2, v_3\}$. □

What remains is an exhaustive case distinction analogous to the analysis done for the 4-0 crossings.

Theorem 4. *If Algorithm 2 creates a 3-0 crossing between two requests $\{v_1, v_2\}$ and $\{v_2, v_3\}$, there is at least one uniquely identifiable unavoidable crossing for at least one of the two requests.*

Proof. Let us assume that v_1 and v_2 is the pair of vertices adjacent to a filled slot s_i. Let v_2 and v_3 be a pair of vertices from a new request, with $v_1 < v_2 < v_3$. If Algorithm 2 creates a 3-0 crossing between the requests $\{v_1, v_2\}$ and $\{v_2, v_3\}$, it places the second request in a slot s_j with $s_j < s_i$. Recall that due to Lemma 9, s_j must be the rightmost available slot to the left of s_i. Moreover, the case where there is no available slot $s_i < s_k$ is already covered by Lemma 10.

Thus, in the following we assume our algorithm can choose between the slot s_j, resulting in a 3-0 crossing, and the slot s_k, which does not generate a 3-0 crossing ($s_j < s_i < s_k$). This situation is depicted in Figure 2.27 (a). We look at the cases where Algorithm 2 prefers a placement on s_j. In the following we investigate which edges must exist to make the slot s_j more preferable.

To count the crossings we divide the relevant slots into two subsets. The subset X contains the slots between s_j and s_i and the subset Y contains the slots between s_i and s_k. We do not need to consider the area to the left of s_j or to the right of s_k, because the number of crossings with edges incident to a slot in one of these areas is independent of the position of the request $\{v_2, v_3\}$. We also divide the vertices on the bottom line into three different subsets. The vertices to the left of v_2 form the set A. The vertices between v_2 and v_3 are in the set B and the vertex v_3 and all vertices to its right form the set C.

Figure 2.27 (b) shows which edges or propagation arrows cross the new request only if placed in the slot s_j (in green) and the ones that cross the new request only if placed in slot s_k (in red). An edge that is incident to a vertex between v_2 and v_3 (area B) does not favor a particular slot, because it crosses the request $\{v_2, v_3\}$ once, independent of its placement.

There must be edges or propagation arrows in our graph such that, avoiding the 3-0 crossing results in at least three crossings in order to make a placement in s_j favorable. Thus, requests like in Figure 2.27 (c) or (d) must be present in our graph in order to make the 3-0 crossing a feasible choice.

However, if a vertex between v_2 and v_3 exists (area B is not empty), its edges unavoidably cross the request $\{v_2, v_3\}$ two times, fulfilling the statement of our lemma. Thus, we can assume that there is no vertex between v_2 and v_3. Therefore, the edges that make the 3-0 crossing more favorable must be incident to vertices from C. The corresponding slot for these edges can be either in X or Y.

We start with the case that there is a filled slot $s_x \in X$ with both edges incident to the set C and analyze it in more detail. Assume that the request $\{v_2, v_3\}$ arrives in the time step t. Thus, at the end of time step $t-1$ the slots s_i and s_x are filled. If the propagation arrows of v_2 and v_3 point to two different slots (s_j and s_k), we can apply Lemma 3 and know that eventually there will be two unavoidable crossings between the request $\{v_2, v_3\}$ and the overarching request. To see this more clearly, we point out explicitly

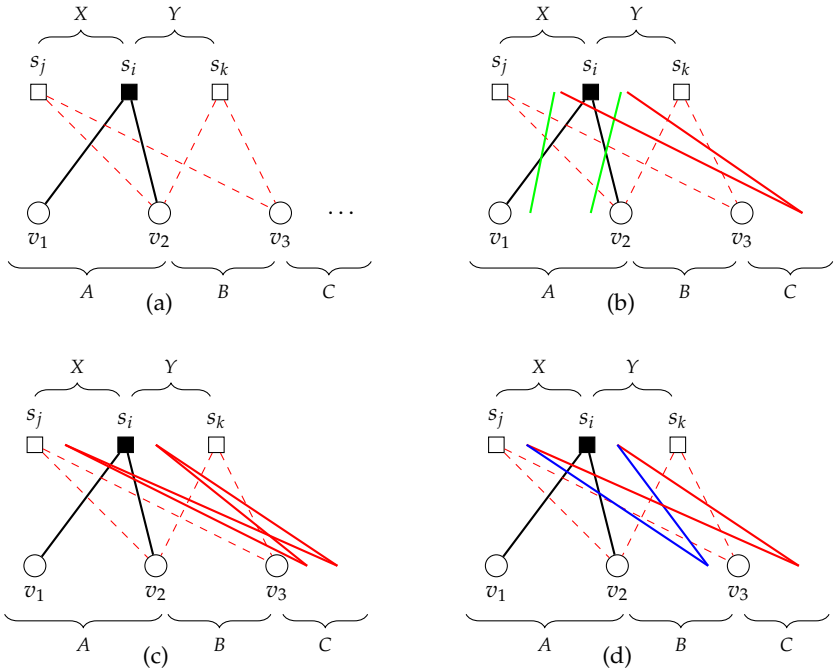

Figure 2.27: We have two possible placements for the request $\{v_2, v_3\}$. Red edges contribute extra crossings to the placement in s_k and green edges contribute extra crossings to the placement in s_j. The blue edges cross with one edge of the request independent of its placement.

how we can apply Lemma 3: The request $\{v_2, v_3\}$ is the request $\{u, v\}$, the free slots s_j and s_k are the slots s_l and s_r, the filled slots s_i and $s_x \in X$ correspond to the slots s_x and s_y.

If the propagation arrows of v_2 and v_3 point to the same slot, Lemma 2 prohibits that the propagation arrows point towards s_k. Therefore, the only possible configuration at the end of time step $t-1$ is that both propagation arrows already point towards s_j. This indicates that the placement of a previous request from a time step $t' < t$ lead to this situation.

In the following we look at the time step t', in which the request arrived that pushed the propagation arrows – starting at v_2 and v_3 – to the slot s_j for the last time. Note that the request $\{v_1, v_2\}$, placed in s_i, cannot push the propagation arrows to s_j, because positioning it onto s_j instead of s_i creates fewer crossings. To be precise, the three crossings between the propagation arrows of v_2 and v_3 and the edges of the request $\{v_1, v_2\}$ can be avoided by placing it in s_j instead of s_i. This means that when the request $\{v_1, v_2\}$ arrived, the propagation arrow configuration must have been different. Thus, the slot s_i must be filled and there must be a different request that is responsible for the last push of the propagation arrows in time step

t'.

Consequently at least one of the propagation arrows of v_2 and v_3 points to a free slot left of s_j or to a free slot right of s_j at the beginning of time step t'. However, if the propagation arrow of v_2 points to a free slot left of s_j, there must be another propagation arrow starting to the right of v_2 that points to s_j. Thus, there are two propagation arrows starting at a vertex to the right of v_2 that cross the edges incident to s_i violating Lemma 2. Therefore, the request in time step t' cannot push the propagation arrows from left to right; it must push the propagation arrows from a free slot to the right of s_j to the left, onto s_j. Note that this implies that all of the slots $s_x \in X$ are also already filled in time step t', because otherwise propagation arrows coming from the right of v_2 would point to $s_x \in X$ and cross both edges incident to s_i, violating Lemma 2 again.

To push the propagation arrows to the left, it is necessary that at least one vertex of the request is to the left of v_3. Since there is no vertex between v_2 and v_3, it must also be to the left of v_2. We differentiate between two cases: The other vertex of this request can be to the left of v_2 or to the right of v_3.

If the other vertex of the request is also to the left of v_2, the propagation arrows of v_2 and v_3 cross both edges of this request at the end of t', violating Lemma 2. If the second vertex is to the right of v_3 we have a request that unavoidably crosses all edges of v_2 and v_3. Thus, for every feasible configuration for time step t', we have unavoidable crossings.

Now we consider the case that there is a slot $s_y \in Y$ with two incident edges to vertices from C and analyze it in more detail. Assume that the request $\{v_2, v_3\}$ arrives in time step t. Thus, at the end of time step $t-1$ the slots s_i and s_y are filled. Again we differentiate between two cases: Either the propagation arrows of v_2 and v_3 point to different slots, s_j and s_k respectively, or they point to the same slot.

If they point to different slots, we can apply Lemma 3 just as in the previous case and know that there will be two unavoidable crossings between the edges from the request $\{v_2, v_3\}$ and the future overarching request.

In the following we assume that the propagation arrows from v_2 and v_3 point to the same slot, s_j or s_k. Note if both point to s_k, the vertex v_3 must be adjacent to s_y, otherwise Lemma 2 is violated. However in this case, the slots become symmetric. Thus, without loss of generality we look at the case that both propagation arrows point to s_j.

Just as in the previous case, we look at the last time step in which the propagation arrows are pushed to the slot s_j and call it t'. At the start of time step t', the request $\{v_1, v_2\}$ must already be placed in the slot s_i, due to the same argument as before. Again it is also not possible that the last time the propagation arrows are pushed is from left to right: If the propagation arrow of v_2 points to a slot left of s_j, the propagation arrows pointing to s_j start at a vertex to the right of v_2 and violate Lemma 2. Thus, at least one

propagation arrow from v_3 is pointing to a slot right of s_j at the start of t'. To push the propagation arrows to the left, it is necessary that at least one vertex of the request is to the left of v_3. The other vertex can be either to the left of v_2 or to the right of v_3 again.

If the other vertex of the request is also to the left of v_2, by the same argument as before, the propagation arrows of v_2 and v_3 cross both edges at the end of t', violating Lemma 2. If the second vertex is to the right of v_3 it unavoidably crosses two times with the request $\{v_2, v_3\}$.

Thus, we have finally proven that for every feasible configuration leading to a 3-0 crossing there must be two unavoidable crossings with the edges of the request $\{v_2, v_3\}$ by the end of the request sequence. □

Theorem 4 shows that 3-0 crossings incurred by Algorithm 2 only happen in conjunction with two extra unavoidable crossings with the request generating the 3-0 crossing. Therefore any 3-0 crossing is in effect a 5-2 crossing. When we assume that the adversary maximizes the number of these crossings, the ratio between the cost of the solution achieved by our algorithm and an optimal one would be 2.5.

2.5.4 The Upper Bound

We can finally put all results together to conclude with an upper bound for the competitive ratio of Algorithm 2 to solve online slotted OSCM-2 on 2-regular graphs.

Theorem 5. *Algorithm 2 solves the online slotted OSCM-2 on 2-regular graphs with a competitive ratio of at most 5.*

Proof. In order to calculate the competitive ratio of Algorithm 2 we simply compare the optimal placement to the placement chosen by Algorithm 2 for every pair of requests.

We exhaustively look at possible placements of pairs of requests, as depicted in Figure 2.5. Observe that except for the 3-0 crossings and 4-0 crossings, the remaining request pairs are no worse than 3-competitive regardless of the algorithm used. Moreover, Theorem 3 ensures that for every 4-0 crossing incurred by Algorithm 2, there is at least one uniquely identifiable unavoidable crossing, meaning that the number of crossings incurred by Algorithm 2 is 5, but optimally there must be at least 1 unavoidable crossing. Finally, Theorem 4 guarantees that there are also two uniquely identifiable unavoidable crossings for every occurrence of a 3-0 crossing. Thus, Algorithm 2 is at most 5-competitive. □

2.6 Conclusion

We have shown that the general slotted OSCM-k is not competitive for any $k \geq 2$, which led us to analyze the case of the slotted OSCM-k on 2-regular graphs. On this graph class we have given a construction to prove a lower bound of $\frac{4}{3}$ on the competitive ratio. Algorithm 2, which utilizes the information of the remaining space and unavoidable crossings in the graph in the form of our so-called *propagation arrows*, was proven to be at most 5-competitive. This was done by limiting the number of total crossings generated by pairs of requests that do not cross one another in an optimal solution.

There are several open questions. First, there is still a considerable gap between the lower and upper bound on the competitive ratio that we have given. We assume that Algorithm 2 performs better than our analysis suggests and that the upper bound can be made tighter.

While Theorem 1 proves non-competitiveness on general graphs for any $k \geq 2$, the case of regular graphs with degree 3 or higher is still open. We suggest to further analyze this graph class.

Chapter 3

Graph Exploration with Advice

Orientation and navigation in an unknown environment are among the basic tasks for autonomous agents. The environment can be physical or virtual like a network of computers. To send messages as fast as possible, it is helpful to know the structure of the network. Thus an explorer can be used to visit and test the connections for every computer in the network. A more physical example is the field of robotics. There are many applications for robots that explore unknown environments on their own [3, 57, 58]. For example, exploring caves or abandoned mines is often a dangerous task and autonomous robots can be used to create maps of such environments.

Since there are many different applications for exploring, there are also many different models for the environment and for the perception of the explorer. The survey of Berman [7] gives an overview of navigation problems and distinguishes the following main properties: The representation of the environment, the task that should be solved, and the senses of the agent. The environment can be a geometric space with obstacles [9] or, this is the case we analyze in this chapter, an abstract and discrete space, where the explorer can move from one point to a neighboring one, i.e., a graph. Starting from a vertex our task is to find a shortest path to a target vertex or to compute a shortest tour that visits every vertex at least once.

3.1 Related Work

The task to compute a shortest tour that visits every vertex at least once is related to the well-known *Traveling Salesman Problem* where the task is to compute the shortest tour that visits every vertex precisely once. Kalyanasundaram et al. introduced the graph exploration problem as an online version of the TSP [36] with the *fixed-graph scenario* which defines the senses of the explorer. In this model, the vertices have unique labels, and the explorer sees all vertices reachable within one step, their labels and their distances to the current position and has unlimited capacity to store the gathered in-

formation. Moving onto a new vertex reveals the adjacent vertices and the explorer recognizes if a vertex was already reachable from a previous step.

Obviously, not having complete knowledge of the graph beforehand makes it impossible in general for the explorer to find a tour – a sequence of vertices in which they can occur multiple times – of optimal length. Thus, algorithms achieving some provable approximation guarantees have been investigated. The best known lower bound on the approximation ratio for exploring general and undirected graphs in the fixed-graph scenario is $\frac{10}{3}$ [8]. For the special case of undirected and weighted graphs with bounded genus g, an upper bound of $16(1 + 2g)$ is known [43]. The case of directed graphs in the fixed-graph scenario is also well studied [1, 28, 29]. In [29], the authors give tight bounds for deterministic and randomized graph exploration in directed graphs with weighted or unweighted edges. Moreover, they look at a variation of the problem where the explorer has to search for a specific vertex in the graph. There are many slight variations of the graph exploration problem: The memory of the algorithm [19, 30], the number of explorers in the graph [15, 17, 18] or the abilities to set pebbles [6] are well studied variations.

For a more fine-grained analysis of how much information about the unknown graph is really needed by the explorer, we look at a variation of the graph exploration problem, where the algorithm has access to some information in the form of a bit string, provided by a helpful oracle that knows the network. The number of bits that the algorithm reads until it finishes its computation is then called its *advice complexity*. The first time that the graph exploration problem was analyzed using the advice complexity model was in [31], where Fraigniaud et al. were able to improve the classical upper bound of 2 on the competitive ratio for tree exploration by adding advice. They proved that $\log\log(D) - c$ bits of advice suffice for c-competitiveness, where D is the diameter of the input tree and $c < 2$. Moreover, they showed that every algorithm that uses fewer advice bits has to be at least 2-competitive.

Since then, there have been many results regarding graph searching problems with advice [20, 39, 34]. The search for a specific vertex in the graph stands in the focus of research in [39]. The authors present an algorithm that uses $\Theta(n/r)$ bits of advice to obtain a competitive ratio r, where n is the number od vertices. In [20], Dobrev et al. look at the trade-off between advice and competitiveness for the cyclic graph exploration problem. Moreover, they show that $\Omega(n \log n)$ bits of advice are necessary to optimally explore a weighted undirected graph. Gorain et al. [34] show bounds for a weaker oracle model, where the oracle does not know the starting position of the algorithm. Moreover, they show a lower bound stating that, with $\mathcal{O}(n)$ advice bits, the length of the exploration sequence is $\Omega(n^2)$.

3.1.1 Overview of Contribution

In this work's second part, we analyze the cyclic and the non-cyclic graph exploration problem in three settings. We start with directed graphs and prove that $\mathcal{O}(m)$ bits of advice suffice to compute an optimal solution, where m is the number of edges. Note that an upper bound of $\mathcal{O}(n \log n)$ advice bits can easily be achieved by sorting the vertices by their first visits and encoding this order. Encoding the label of one vertex costs $\log(n)$ bits of advice and there are at most $n-1$ vertices that are visited for the first time (the start vertex is always the first vertex that is visited for the first time). The algorithm then moves the agent from its current position to the next vertex in the encoded ordering, using the shortest path in the already explored part of the graph. Such a path always exists, because the oracle encoded the order of first visits.

A lower bound of $\Omega(n \log n)$ advice bits for general graphs was first observed by Královič [40]. For the sake of completeness, we include a proof for directed graphs in Section 3.5.2. These (almost) tight bounds motivate the investigation of special graph classes. We present an improvement over the general strategy for sparse graphs, i.e., for graphs with $o(n \log n)$ edges. Note that we can assume that $m \geq n-1$, as otherwise the graph would not be connected, making its exploration impossible. We first focus on the case where the algorithm has to compute a cyclic tour visiting all vertices of a directed graph that is unknown and has bounded outdegree. Then, we show how the problem for unbounded-degree graphs can be solved by some modification of the algorithm. Since our algorithm relies on the encoding of an optimal solution within the given advice and does not take the edge weights into account, we formulate our results for the more general case of arbitrary edge weights. We then complement these upper bounds with a lower bound for the case of bounded degree graphs and the already mentioned lower bound for general directed graphs.

Afterwards we discuss the graph exploration problem in the context of undirected graphs. The upper bound of $\mathcal{O}(n \log n)$ mentioned above also works for undirected graphs and matches the lower bound given in [20]. Thus, we look at degree-bounded graphs again and present a simple strategy that needs at most $(2n-4)\log(k)$ bits of advice where k bounds the degree. To complete the analysis we also present an asymptotically matching lower bound. The results for directed and undirected graphs are summarized in Table 3.1.

At the end we also take a look at a different scenario where the algorithm has partial information about the given instance without using advice. We distinguish between different levels of a priori knowledge and analyze how much advice is still necessary depending on the amount of available information.

The remaining chapter is organized as follows. In Section 3.2, we give

graph model	outdegree	upper bound	lower bound
directed	2	$5.25n - 1$	$1.25n$
directed	arbitrary	$5.25m - 1$	$\Omega(n \log(n))$ [20]
undirected	2	$1 + \log(n) + 2\log(\log(n))$	$\log(n-2)$
undirected	k	$(2n-4)\log(k)$	$\frac{n}{4}(\log(k+1) - 1)$

Table 3.1: Upper and lower bounds on the advice complexity of graph exploration. The bounds hold for the cyclic and the non-cyclic graph exploration problem.

the basic definitions for dealing with the graph exploration problem. Section 3.3 gives some basic observations and in Section 3.4 we start with the analysis of the directed graph exploration problem. We focus on the cyclic graph exploration problem and present an algorithm that uses at most $5.25n - 1$ bits of advice to compute the optimal exploration sequence on directed graphs with outdegree bounded by two. This result is extended to directed graphs of arbitrary degree resulting in an advice complexity of $5.25m - 1$. Afterwards we discuss how our algorithms can be adapted to also solve the non-cyclic graph exploration problem, without needing additional advice. To complete our analysis, in Section 3.5, we show that any algorithm needs at least $1.25n$ bits of advice to explore graphs with outdegree bounded by two. The graph exploration problem on undirected and bounded-degree graphs is handled in Section 3.6. After some structural observations exclusive to undirected graphs, we present an upper bound of $(2n-4)\log(k)$, where k bounds the degree, and a complementary lower bound. In Section 3.7 we analyze different a priori information about the given input and how it influences the advice complexity. We show that knowing the structure of the graph (the vertices and their edges) but not the weight function for the edges does not allow any online algorithm to compute the optimal solution. If the algorithm has the additional information, for every edge, whether it is used never, once or multiple times in an optimal solution, it can compute the optimal solution.

3.2 Preliminaries

We start with the definition of the basic variant of the graph exploration problem.

Definition 4. *Let $G = (V, E)$ be a directed graph. Every vertex $v \in V$ has a fixed unique identifier. There is an agent, called* explorer*, initially positioned on some start vertex $v_0 \in V$. The algorithm has to move this explorer along the directed edges of G to visit all vertices and return to v_0. The edges are weighted by a weight function $cost\colon E \to \mathbb{N}$, and the goal is to minimize the total weight along the cyclic tour traveled by the explorer. In every vertex the explorer is located, it sees the outgoing edges, their weights, and the vertex identifiers at the endpoints of these edges, but not the incoming edges. The explorer has unlimited capacity to store all information gathered during the exploration.*

We continue with some notations for the solutions or search sequences computed by online and offline algorithms.

Definition 5. *Let $G = (V, E)$ be a directed graph. The* out-neighborhood *of a vertex $v \in V$ is defined as $N_{out}(v) = \{w \mid (v, w) \in E\}$. Analogously, we define the* in-neighborhood *of a vertex v as $N_{in}(v) = \{w \mid (w, v) \in E\}$.*

We describe the tour followed by the explorer in terms of a *search sequence*.

Definition 6. *Let $G = (V, E)$ be a graph. A sequence $S = (v_0, v_1, \ldots, v_s)$ is called a* search sequence *if $(v_{i-1}, v_i) \in E$ for all $1 \leq i \leq s$. If $v_0 = v_s$, we call S a* cyclic search sequence.

For a search sequence $S = (v_0, \ldots, v_j)$ with $e_i = (v_{i-1}, v_i)$, for $1 \leq i \leq j$, we denote by $E(S) = (e_1, e_2, \ldots, e_j)$ the set of edges in S and by $V(S) = \{v_0, \ldots, v_j\}$ the set of vertices in S. With S_E, we refer to the sequence of edges, in which some edges might appear multiple times. The number describing how often an edge e appears in S_E is the number of traversals of e and is denoted by $\#_S(e)$. The cost of a search sequence $S = (v_0, \ldots, v_j)$ is defined by $cost(S) = \sum_{e \in E(S)} cost(e) \cdot \#_S(e)$.

The search sequence is determined by the algorithm as follows. In each step, the explorer is located at some vertex v and the algorithm chooses one of the vertices from $N_{out}(v)$ as the target and moves the explorer towards it. As soon as the explorer arrives at the target vertex, a new round starts and the algorithm again receives the unique identifiers for the out-neighborhood and has to make an irrevocable decision which next vertex to choose.[1] The goal is to compute a cyclic search sequence visiting each vertex at least once, which we call an *exploration sequence*.

Since the algorithm lacks global information about the structure of the graph, there is no deterministic algorithm that finds an optimal exploration

[1] Note that, with each decision, the algorithm influences the new input for the next decision. Thus, strictly speaking, the graph exploration problem is no classical online problem. But the adversary still knows the behavior of the deterministic algorithm and can, with this knowledge, prepare the input graph, the unique identifiers for the vertices, and thus the enumeration of the edges. Hence, we can analyze the graph exploration problem using the same methodology as used for online problems.

sequence for an arbitrary graph. We employ the model of *online algorithms with advice* as defined in [35, 12] for measuring the amount of missing information, which we can define in the framework of graph exploration as follows. An online algorithm with advice computes a search sequence $S = (v_0, v_1, \ldots, v_{end})$ for an unknown graph, where v_i is computed from the partial knowledge about the graph gathered in the first $i-1$ rounds and the content ϕ of the advice tape, i.e., an unbounded binary sequence of *advice bits* computed by an *oracle* that knows the complete input graph together with its edge weight function. An online algorithm with advice *solves* the graph exploration problem if there exists a computable advice ϕ such that S is an optimal exploration sequence, for any input $(G, cost)$. The algorithm has sequential access to the bits from the advice tape, and its *advice complexity* is the number of accessed advice bits. As usual, we measure the advice complexity with respect to the input size by considering a worst-case input of the respective size.

Now we explain how the algorithm makes a decision to extend a search sequence S, based on the advice from the oracle. The basic idea is that the oracle chooses a set of optimal exploration sequences $\mathcal{S}$ and communicates a sufficient amount of information such that the algorithm can compute one of these sequences without taking the weights of the edges into consideration. The advice needs to ensure that the algorithm does not use an edge more often than in any solution of $\mathcal{S}$ and that all vertices are visited. As a first step, we partition the edges into three sets according to their number of traversals (none, one, or multiple times) in an optimal exploration sequence.

To this end, we use the following notation.

Definition 7. *Let $G = (V, E)$ be a graph, let S^* be an optimal exploration sequence from $\mathcal{S}$ and let S be an arbitrary search sequence. Then $\#_S(e)$ is the number of traversals through an edge $e \in E$ in the search sequence S and E_0, E_1, and E_{multi} are the sets of edges in E which are visited* 0, 1, *or multiple times by S^*, respectively.*

We denote the set of edges in E which are visited at least once by S^* by $E_{used} = E_{multi} \cup E_1$. If $\#_S(e) = \#_{S^*}(e)$, e.g., an edge e is used as often in S as in the fixed optimal solution S^*, we say that e is exhausted *in S.*

The number of traversals for the edges could differ for different optimal solutions, but the oracle fixes a set of optimal solutions such that the number of traversals for the edges is the same in every solution from the fixed set. This allows the oracle to give advice that is consistent during the exploration. Figure 3.1 (a) shows a sample graph where the number of traversals for the edges in an optimal solution is non-unique. The five traversals through the vertex x needs to be split up between the two paths (y, v_1, x) and (y, v_2, x).

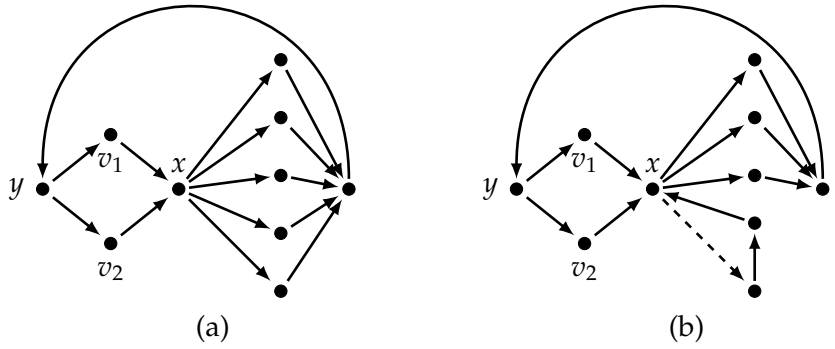

Figure 3.1: (a) The optimal number of traversals for the edges of this graph is non-unique. The five successors of x and the two possible paths to x require that the algorithm traverses (y, v_1, x) or (y, v_2, x) multiple times. (b) In an optimal solution, the dashed edge must be exhausted before the last of all other outgoing edges of the vertex x is exhausted.

Since the edges from E_{multi} need additional advice to avoid errors, the oracle is interested in fixing a solution set $\mathcal{S}$ that minimizes the number of edges from E_{multi}. For the graph presented in Figure 3.1(a), the oracle chooses $\mathcal{S}$ such that, for all solutions in $\mathcal{S}$, either the edges (y, v_1) and (v_1, x) or the edges (y, v_2) and (v_2, x) are used exactly once. In any solution from this set, the number of edges from E_{multi} is three. The solution set $\mathcal{S}$ contains at least $5! = 120$ different solutions because the order in which the out-neighborhood of the vertex x is visited is not fixed. In the following, when we speak of an optimal and fixed solution S^*, we mean $S^* \in \mathcal{S}$.

Definition 8. *We call an optimal exploration sequence with a minimum number of edges from E_{multi}* multi-edge-optimal.

To ease the analysis of our algorithm, we now explain in detail how the oracle fixes the solution set $\mathcal{S}$. The oracle constructs a graph $G' = (V, E)$ from the given graph $G = (V, E)$ such that all solutions in $\mathcal{S}$ are multi-edge-optimal and the number of traversals is the same for all solutions in $\mathcal{S}$. The oracle constructs G' by perturbing the weights on the edges by a small amount such that, for all pairs $y, x \in V$ that are connected by more than one path, at most one of them consists of multi-edges only. This minimizes the number of edges from E_{multi} and also fixes the traversal number for every edge. It is important that the perturbation of the edge traversal weights does not create a new optimal solution that is not optimal in the original graph G. The following lemma guarantees that we can choose a sufficiently small perturbation such that all $S^* \in \mathcal{S}$ are also optimal for G.

Lemma 11. *Let S^* be an optimal exploration sequence for a graph with n vertices. Then $\#_{S^*}(e) \leq n$, for all $e \in E(S^*)$.*

Proof. Assume that there exists an edge e with $\#_{S^*}(e) > n-1$. This would imply that the algorithm sends the explorer more often along the edge e than there are vertices in G. Thus, at least one of the traversals along e is not needed to explore a vertex and this contradicts the optimality of S^*. □

When the oracle adds an additional cost of $1/n^{2i}$ to every edge e_i, for $1 \leq i \leq m$, the difference between the cost of an optimal solution on G' and on G will be smaller than 1. Since

$$n \cdot \sum_{i=1}^{m} \frac{1}{n^{2i}} = \sum_{i=1}^{m} \frac{1}{n^{2i-1}} < 1\,,$$

the optimal solution for G' is also optimal for G, even when we assume that the optimal solution traverses every edge $n-1$ times.

From now on, we assume that the set of multi-edge-optimal solutions $\mathcal{S}$ contains only exploration sequences S^* that are fixed by the oracle. This means that the number of traversals for every edge is unique, but not the order in which the edges are traversed. For instance, in Figure 3.1(a), the order in which the five out-neighbors of x are visited does not matter. In contrast, Figure 3.1(b) shows that the order in which the edges are used can be important. The dashed edge must be traversed before the last of all other outgoing edges is exhausted.

We already know that the input graph must be strongly connected if it has a solution that visits every vertex at least once. Since the connectivity of the graph alone does not reflect the current position of the explorer and the already exhausted edges, we introduce the term of an *expandable search sequence* to formulate a stronger and more precise connectivity requirement.

Definition 9. *We call a search sequence $S = (v_0, \ldots, v_k)$* expandable *with respect to a fixed optimal exploration sequence $S^* = \{v_0, \ldots, v_{end}\}$ if there exists an outgoing edge $e = (v_k, w)$ that is not yet exhausted and there is a search sequence from v_k that uses only non-exhausted edges, visits all yet unvisited vertices and reaches the final vertex v_{end} such that $cost(S^*) = cost(S)$ and $V(S) = V$.*

For instance, for the graph from Figure 3.1 (b), if an algorithm uses all of the non-dashed edges outgoing from x before the dashed edge is traversed, its search sequence is not expandable anymore.

3.2.1 Self-Delimiting Encoding

Often, the oracle will encode just some binary decision on the advice tape, which will cost just one bit of advice. But if we want to encode a longer message, the algorithm might not know where one bit string of information ends and the next one begins. This happens, e.g., when the oracle wants to communicate a natural number x to the algorithm.

x	prefix	binary representation	$\lceil \log(x+1) \rceil$	$2\lceil \log(\lceil \log(x+1) \rceil) \rceil$
0	0	0	-	-
1	0	1	-	-
2	10	10	2	2
3	10	11	2	2
4	1100	100	3	4
5	1100	101	3	4
6	1100	110	3	4
7	1100	111	3	4
8	1110	1000	4	4
89	111100	1011001	7	6

Table 3.2: Examples for our self-delimiting encoding. The prefix together with the binary representation encode the number x on the advice tape.

To encode a number x on the advice tape, the oracle can first encode the length of the binary representation of x and then x itself as binary string. We use an encoding similar to self-delimiting codes from [38], which are strongly related to *Elias coding* [26]. We have to adapt our encoding to ensure that the number 0 can be communicated to the algorithm. Table 3.2 shows examples of our self-delimiting encoding. The binary string representing x has a length of $\lceil \log(x+1)) \rceil$. This length minus one will be encoded in a prefix with $2\lceil \log(\lceil \log(x+1) \rceil) \rceil$ bits. In the prefix, every bit on an odd position (starting with the first bit) is part of the binary number and the bits on the even positions are 1 if the number continues and 0 if the prefix ends. Thus, $2\lceil \log(\lceil \log(x+1) \rceil) \rceil + \lceil \log(x+1) \rceil$ bits of advice suffice to encode x on the advice tape. There are two exceptions, the number 0 is encoded as 00 and 1 is encoded as 01. This minimize the advice complexity because these two numbers are read often by our algorithm and consume only two bits with this encoding.

3.3 Structural Observations

In this section, we focus on some structural properties of any of the fixed optimal solutions $S^* \in \mathcal{S}$. We distinguish between two different graphs that can be induced by S^*. If we remove the edges from E_0 and just look at the graph $G_{S^*} = (V, E_{used})$, we have a graph for which all edges are needed in an optimal exploration sequence. If we are more interested in traversed subsequences and want to distinguish different traversals over the same edge from E_{multi}, we look at the multigraph $M_{S^*} = (V, E_{used}, \#_{S^*})$. In M_{S^*}, every edge $e = (v, u) \in E_{multi}$ is replaced by $\#_{S^*}(e)$ many edges that point from v to u. The multigraph M_{S^*} is an Eulerian graph and the solution S^* describes an Eulerian cycle. Let $d_{in}(v)$ denote the indegree of the vertex v in M_{S^*}.

Definition 10. *A directed graph G is called Eulerian if every vertex has as many incoming edges as it has outgoing edges. If G is also weakly connected, it has an Eulerian cycle, i.e., a tour that visits every edge in the graph once and ends at the same vertex as it starts.*

The following lemmas describe structural properties of optimal exploration sequences from $\mathcal{S}$, which will be helpful for reconstructing one of the fixed solutions from a rather small amount of advice.

Lemma 12. *Let $G = (V, E)$ be a connected graph, and let $S^* \in \mathcal{S}$ be a multi-edge-optimal exploration sequence for G. Then any two subsequences $S_i = (v, \dots, u)$ and $S_j = (v, \dots, u)$ from S^* are interchangeable, i.e., changing their order in S^* results in another optimal exploration sequence $\hat{S} \in \mathcal{S}$.*

Proof. Changing the order of two subsequences S_i and S_j, which start at a vertex $v \in V$ and end at $u \in V$, does not change the number of traversals for any $e \in E(\hat{S})$. Thus, the cost of $\hat{S}$ is still minimal when the two subsequences S_i and S_j are exchanged.

Moreover, the exploration sequence $\hat{S}$ is still feasible. If we assume, for a contradiction, that the exploration sequence $\hat{S}$ is not feasible, there must be a vertex u that is not visited or the sequence $\hat{S}$ is not continuous. By exchanging two subsequences that start and end at the same vertex, the exploration sequence stays continuous, i.e., two consecutive edges (v_1, v_2) and (v_2, v_3) in $\hat{S}$ are incident to the same vertex v_2. If we assume that there is some unvisited vertex u, then it must be part of the subsequence, because the remaining part of the exploration sequence $\hat{S}$ is equal to S^*. However the subsequences visit the same vertices as they do in S^*. Therefore, our assumption that $\hat{S}$ is not feasible, is false. □

When the interchangable sequences from Lemma 12 are cyclic sequences $C_i = (v, \dots, v)$ and $C_j = (v, \dots, v)$, we speak about interchangeable cycles. It still holds that changing their order in S^* results in an optimal exploration sequence $\hat{S} \in \mathcal{S}$. Note that all cycles and all sequences with the same end vertex that start at the starting vertex v_0 are interchangeable. For the remaining vertices, we make the following observation.

Corollary 1. *Let $G = (V, E)$ be a connected graph, and let $S^* \in \mathcal{S}$ be multi-edge-optimal. A vertex v is part of $d_{in}(v) - 1$ cyclic subsequences $C_i = (v, \dots, v) \in S^*$ with $1 \leq i \leq d_{in}(v)$ that are interchangeable.*

3.3.1 The Structure of Multi-Edges

Using Corollary 1 and Lemma 12, we get a better intuition for the fixed set of solutions $\mathcal{S}$. The solutions in $\mathcal{S}$ only differ in the order in which the interchangeable cycles and paths are traversed. The following lemmas

discuss the number of traversals for the edges in an arbitrary solution from $\mathcal{S}$, based on the property that the number of edges from E_{multi} is minimized.

Lemma 13. *Let $G = (V, E)$ be a connected graph, and let $S^* \in \mathcal{S}$ be multi-edge-optimal. If there is an edge $(w, v) \in E_{multi}$, there is no edge $(v, w) \in E_{used}$.*

Proof. The vertices u and v are visited at least two times, because $(w, v) \in E_{multi}$.

If the two edges (w, v) and (v, w) are consecutive in S^*, we can create a cheaper exploration sequence S' by removing the cycle (w, v, w). In S', the vertices u and v are still visited at least once and, for every other vertex, the number of visits does not change. Furthermore, S' is continuous because only a cyclic subsequence is removed. This contradicts the optimality of S^*.

If the two edges (w, v) and (v, w) are not consecutive, we create an exploration sequence $\hat{S}$ where the two edges are consecutive. From Corollary 1, we know that the edge (w, v) must be part of a cyclic subsequence. This cyclic subsequence is of the form $(w, v, z, \ldots, w)$. Now, we have two subsequences from the vertex v to the vertex w. One is the path $(v, z, \ldots, w)$ and the other one is the edge $(v, w) \in E_{used}$. Applying Lemma 12, we exchange these sequences and obtain an exploration sequence $\hat{S}$ with the same weight. Now, we are in the situation from above and this contradicts the optimality of S^*. □

Lemma 14. *Let $G = (V, E)$ be a graph with a multi-edge-optimal exploration sequence $S^* \in \mathcal{S}$. Let $C = (v, \ldots, v)$ be a simple cyclic subsequence of S^*. The multigraph $M_{S'} = (V, E, \#_{S'})$ induced by the number of traversals of S' with*

$$\#_{S'}(e) = \begin{cases} \#_{S^*}(e) - 1 & \text{for } e \in E(C), \\ \#_{S^*}(e) & \text{otherwise,} \end{cases}$$

contains at least two connected components that are Eulerian. Note that a single vertex without edges is an Eulerian component.

Proof. Assume $M_{S'} = (V, E, \#_{S'})$ is only one connected component. The graph M_{S^*} is Eulerian because, for every vertex in V, the number of incoming edges matches the number of outgoing edges. When we remove one incoming and one outgoing edge for every vertex in C, the number of incoming matches the number outgoing edges again, for all vertices in V. Thus, the graph $M_{S'}$ is also Eulerian.

When a directed graph is connected and Eulerian, it contains an Eulerian cycle (see Definition 10). Therefore, there exists a cheaper exploration sequence with the same number of traversals as in S'. This is a contradiction to the minimality of the cost of S^*. □

From Lemma 14, we can conclude that, in every cyclic subsequence C of an optimal solution $S^* \in \mathcal{S}$, there are at least two edges that are used at

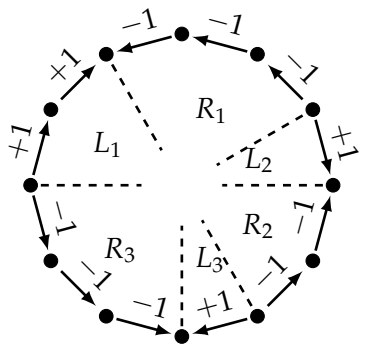

Figure 3.2: A cycle of multi-edges in the underlying undirected graph. Edges that are contiguous and equally directed are merged into subsequences R_i and L_i.

most once. If this was not the case, then the graph M_{S^*} without C would not fall apart into two components since every Eulerian graph is strongly connected.

Corollary 2. *Let $G = (V, E)$ be a graph with a multi-edge-optimal exploration sequence $S^* \in \mathcal{S}$. Let $C = (v, \ldots, v)$ be a simple cyclic subsequence of S^*. Then there are at least two edges $e_1, e_2 \in E(S)$ with $\#_{S^*}(e_1) = \#_{S^*}(e_2) = 1$.*

Our next observation is that also every cycle in the underlying undirected graph contains at least two edges from E_1. For a given optimal exploration sequence $S^* \in \mathcal{S}$, we look at the underlying undirected graph and assume there is a cycle C of multi-edges. For such a cycle of multi-edges, we merge equally directed edges into sequences L_i and R_i, as presented in Figure 3.2. If we increment the traversal numbers for the sequences L_i and decrease them for the sequences R_i, we obtain a cheaper solution, as proven in the following lemma.

Lemma 15. *Let $G = (V, E)$ be a graph with a multi-edge-optimal exploration sequence $S^* \in \mathcal{S}$. Let $L_i = (v^i, \ldots, u^i)$ and $R_i = (w^i, \ldots, u^i)$, with $1 \leq i \leq k$, be $2k$ simple subsequences of S^* with $w^i = v^{i+1}$ $(1 \leq i < k)$ and $v^1 = w^k$. Then there are at least two edges $e_1, e_2 \in L_i \cup R_i$ with $\#_{S^*}(e_1) = \#_{S^*}(e_2) = 1$.*

Proof. Assume that there is at most one edge $e_1 \in E_1$ and $\#_{S^*}(e) > 1$ holds for all edges $e \in \bigcup_{1 \leq i \leq k}(E(L_i) \cup E(R_i)) \setminus \{e_1\}$. Without loss of generality, we assume that

$$\sum_{1 \leq i \leq k} cost(L_i) < \sum_{1 \leq i \leq k} cost(R_i). \tag{3.1}$$

We create a cheaper exploration sequence S' by choosing the cheap sequences L_i to a vertex v more often than the expensive R_i. Thus, the number

of traversals in S' is defined with respect to the optimal solution S^*:

$$\#_{S'}(e) = \begin{cases} \#_{S^*}(e) + 1 & \text{for } e \in \bigcup_{1 \leq i \leq k} E(L_i)\,, \\ \#_{S^*}(e) - 1 & \text{for } e \in \bigcup_{1 \leq i \leq k} E(R_i)\,, \\ \#_{S^*}(e) & \text{otherwise.} \end{cases}$$

Due to Equation (3.1), the newly created solution must be cheaper. Note that $\#_{S^*}(e) - 1 > 0$ for $e \in \bigcup_{1 \leq i \leq k}(E(L_i) \cup E(R_i)) \setminus \{e_1\}$ and $\#_{S^*}(e_1) - 1 = 0$, if $e_1 \in E_1$. Thus, the graph $M_{S'}$ has at most one edge fewer than M_{S^*}, but it is at least weakly connected. Moreover, for all vertices in $M_{S'}$, the number of incoming edges is equal to the number of outgoing edges. From Definition 10, we know these two conditions imply that $M_{S'}$ is strongly connected. Therefore, the graph $M_{S'}$ has an Eulerian cycle which is a cheaper solution on G than S^*, contradicting its optimality. □

We showed in Lemmas 13 to 15 and Corollary 2 that every cyclic subsequence in an optimal solution $S^* \in \mathcal{S}$ has to connect a new vertex to the already visited part of the graph. Because the graph induced by E_{multi} is cycle-free, it is a forest and we will from now on call it F. This holds also for the underlying undirected graph as well.

3.3.2 Exploring the Last Vertex

During exploration the algorithm gathers more and more knowledge about the graph. With this knowledge it may be possible to make optimal decisions without the help of the oracle. This occurs at the latest when we have explored every vertex except for (the last) one, because this vertex is visited exactly once.

Lemma 16. *Let $G = (V, E)$ be a graph with a multi-edge-optimal exploration sequence $S^* \in \mathcal{S}$. The last vertex in S^* that is visited for the first time is visited exactly once.*

Proof. We prove the statement by contradiction. Assume there would exist a multi-edge-optimal exploration sequence $S^* \in \mathcal{S}$ in which the last vertex v in the order of the first visits is visited more than once. Then there exists a cyclic subsequence that does not visit any new vertex contradicting the optimality of S^*. □

Note that this implies that the last vertex that is explored has only one outgoing edge in the graph G_{S^*}. The lemma also holds true for the non-cyclic graph exploration problem, but for this problem the last vertex has no outgoing edges in G_{S^*}.

In the following we show how an algorithm can compute the optimal path to the last vertex only with the information gathered so far during exploration.

Lemma 17. *Let $G = (V, E)$ be a graph with a multi-edge-optimal exploration sequence $S^* \in \mathcal{S}$. When the algorithm computes the prefix of S^* until only one vertex is not visited, it can expand its search sequence with respect to S^* without additional information.*

Proof. The explorer is placed at a vertex u for the first time and only one vertex v is not yet visited. Thus, the algorithm knows all edges in the graph except for the outgoing edges that start at v. The shortest path from u to v contains only edges that are known to the algorithm. Therefore, the shortest path from u to v in the currently known network must also be part of the optimal exploration sequence.

When the explorer is then moved onto v, all edges (and their weights) are revealed. For the cyclic graph exploration problem, only one of these edges is part of S^* and it is used once to return to the starting vertex. The algorithm knows which edge must be used to leave v because it is part of a shortest path from v to the starting vertex. For the non-cyclic graph exploration problem none of the edges is part of S^* because the exploration is finished when the explorer reaches v. □

3.3.3 The Number of Edges in an Optimal Solution

In the following we present a tight upper bound for the number of distinct edges that are part of an optimal exploration sequence S^*. Note that these edges may be used multiple times in S^* and we only bound the size of the set E_{used}.

Our proof works inductively on the number of cycles in S^*. For the induction we need another lemma showing that removing a cyclic subsequence $C \subseteq S^*$ results in an optimal exploration sequence for the graph where we remove the vertices that are uniquely explored by C. From Lemmas 13 to 15 and Corollary 2 we know that every cyclic subsequence in an optimal solution $S^* \in \mathcal{S}$ has to explore at least one new vertex.

Lemma 18. *Let $G = (V, E)$ be a graph with an optimal exploration sequence $S^* \in \mathcal{S}$. Further let $C \subseteq S^*$ be a cyclic subsequence in S^* and the vertices visited only by C are $V_C = V(C) \setminus V(S^* \setminus C)$. Respectively, $E_C = E(C) \setminus E(S^* \setminus C)$ is the set of edges only used in C. The exploration sequence $S = S^* \setminus C$ is optimal for the graph $H = (V \setminus V_C, E \setminus E_C)$.*

Proof. We prove the optimality of S for H by contradiction and assume that there exists a cheaper exploration sequence S'. We call the vertex where the cycle C starts and ends v. If S' is cheaper than S, we are able to construct a exploration sequence for G that is cheaper than S^*. We use S' on H and on the first visit of v, we add the cycle C into the exploration sequence. This new sequence visits every vertex and must have lower costs than S^*, because $cost(S^*) = cost(S) + cost(C) > cost(S') + cost(C)$. This contradicts

the assumption that S^* is an optimal exploration sequence and therefore S is an optimal exploration sequence for H. □

Theorem 6. *Let $G = (V, E)$ be a graph (directed or undirected) with $|V| = n$ and an optimal exploration sequence S^*. The number of distinct edges in S^* is at most $|E_{used}| \leq 2n - 2$.*

Proof. We prove the statement by induction over the number of cyclic subsequences in S^*. It is important that the cyclic subsequences are ordered in the way they appear in S^*.

Our induction hypothesis is that graphs with a cyclic exploration sequence containing at most k cyclic subsequences and n vertices have at most $2n - 2$ edges. As the base case for our induction, assume that there is just one cyclic subsequence. Then S^* is a simple cycle and therefore the number of edges in S^* is $m = n \leq 2n - 2$, for $n > 1$.

For the induction step, we will start with a graph $G' = (V', E')$ with $k + 1$ cyclic subsequences, $|V'| = n'$ and $|E'| = m'$ and remove one cyclic subsequence and count the removed edges and vertices. Note that every cyclic subsequence that is traversed by the explorer has to contain at least one vertex that is visited exactly once. When we remove the first traversed cyclic subsequence C in S^*, $x \geq 1$ vertices become unvisited and $2x$ edges become unused. Thus, after deleting C, we obtain a graph $G = (V, E)$ with $|V| = n$ and $|E| = m$. The number of vertices and edges in the graphs G and G' are in the relation

$$n = n' - x \quad \text{and} \quad m = m' - 2x \,.$$

When we reformulate the number of vertices of G' in dependency to G and apply the induction hypothesis, we obtain

$$2n' - 2 = 2(n + x) - 2 = 2n + 2x - 2 \overset{\text{(IH)}}{\geq} m + 2x = m' - 2x + 2x = m'$$

as our bound for the number of edges in G'. Thus, the statement holds by induction. Note that we can apply the induction hypothesis because the exploration sequence $S = S^* \setminus C$ is optimal for G (see Lemma 18). □

There are graphs for which this bound is tight, as shown in Figure 3.3. The optimal exploration sequence on these graphs consists of $n - 1$ edge disjoint cyclic subsequences.

3.4 Exploring Directed Graphs with Advice

We start by introducing an online algorithm with advice for the cyclic graph exploration problem on directed graphs with outdegree two. First we explain how we use the advice from the oracle and introduce new notations.

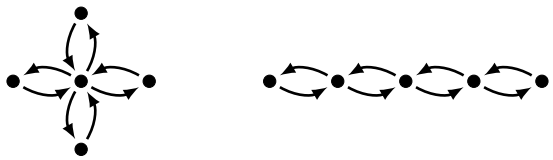

Figure 3.3: Examples where the number of edges in E_{used} is exactly $2n - 2$.

Then, we explain the algorithm, show its correctness and prove its advice complexity of $5.25n - 1$, where n is the number of vertices in the input graph.

Afterwards we explain how the algorithm can be used to solve the cyclic graph exploration problem on graphs with unbounded outdegree. This increases the required advice in dependence on the number of edges. At last we discuss why we can apply the same algorithm for the case of the non-cyclic graph exploration problem.

3.4.1 Directed Graphs with Outdegree Two

Before we informally explain our algorithm for directed graphs with outdegree at most two, we recall what the algorithm perceives in the *fixed-graph scenario*. As soon as the explorer is located in a vertex v, the algorithm gets to know the identifiers of v and of all out-neighbors $N_{out}(v)$ of v. If a neighbor $w \in N_{out}(v)$ was already the out-neighbor of a previously visited vertex, the algorithm recognizes this vertex. The still unvisited vertices $N_{in}(v)$ that lead to v and their corresponding edges stay hidden as long as the explorer is positioned at v.

Now we explain how the algorithm uses advice and makes its decisions. There are three different types of queries to the oracle:

- For every vertex v with two outgoing edges, we ask once which outgoing edge is used less often in $S^* \in \mathcal{S}$. This edge is called *light*.
- For every *light* edge e, we ask its number of traversals $\#_{S^*}(e)$, including 0.
- For every vertex v, we ask if on the last visit of v, the incident *light* edge is always used to leave v, in all solutions from $\mathcal{S}$.

Definition 11. *Let G be a graph where every vertex has at most two outgoing edges and let S^* be a multi-edge-optimal exploration sequence. For a vertex v with two outgoing edges e_1 and e_2 we call e_1* light *if $\#_{S^*}(e_1) < \#_{S^*}(e_2)$. The other edge e_2 will be called* heavy*. If both edges are used equally often, the oracle can decide arbitrarily which edge is* light*.*

Note that the number of traversals for the *light* edge is a lower bound for both edges because the *heavy* edge is used at least as often as the *light* edge.

Definition 12. *Let G_{S^*} be the graph induced by a multi-edge-optimal solution S^* where every vertex has at most two outgoing edges. For a vertex v with two outgoing edges, an edge is called* last *if it is used to leave v on its last visit, in all solutions from $\mathcal{S}$.*

Using a *last* edge too early prevents the algorithm from completing an optimal exploration sequence. Such an edge is important to sustain the expandability (see Definition 9) of the current exploration sequence.

The algorithm knows for every vertex, which of the incident edges is *light*, its number of traversals and if it is *last*. If the explorer stands on a vertex with only one outgoing edge, the algorithm does not need to make a decision because the only valid move is to use this edge. Therefore, we assume that the explorer is positioned at a previously unvisited vertex where the algorithm has to decide between two edges. When the explorer is positioned on a vertex v, we distinguish three different cases, based on the *light* edge e_1.

- If $\#_{S^*}(e_1) = 0$, the edge e_1 is ignored in the further process and we always traverse e_2 when we visit v.
- If $\#_{S^*}(e_1) \geq 1$ and e_1 is *last*, the algorithm uses the edge e_1 on the first $\#_{S^*}(e_1) - 1$ visits of v.
- If $\#_{S^*}(e_1) \geq 1$ and e_1 is not *last*, the algorithm uses the edge e_1 on the first $\#_{S^*}(e_1)$ visits of v and exhausts it.

Afterwards, the explorer is sent along the other edge e_2. We will prove that, with this strategy, the algorithm recognizes when the explorer visits v for the last time. On the last visit of v, the explorer is sent along e_1 if it is *last* or along e_2 otherwise.

The main idea of this strategy is to avoid any mistake by first using the interchangeable cycles and not exhausting the *last* edge. As long as the explorer does not use an edge for the last time, (the algorithm knows that) the incident vertices are visited again.

With the information about *light* and *last* edges, our algorithm will reconstruct a specific exploration sequence from $\mathcal{S}$. The exploration sequence uses the *light* edges as often as possible before using the *heavy edge*.

It remains to show that our algorithm uses every edge e as often as it is used in S^*, which implies the correctness of our algorithm.

Theorem 7. *Let G be a graph where every vertex has at most two outgoing edges and let S^* be a multi-edge-optimal exploration sequence. Knowing for every vertex*

v with two outgoing edges e_1, e_2 that e_1 is the light *edge, its number of traversals $\#_{S^*}(e_1)$ and whether e_1 or e_2 is a* last *edge suffices to extend a search sequence every time the explorer visits v such that e_1 and e_2 are used as often as they are used in S^*.*

Proof. For the first $\#_{S^*}(e_1) - 1$ visits of v, we can choose the edge e_1 using Corollary 1. Now we have to distinguish how to extend the search sequence depending on the *last* edge.

If we know that e_1 is a *last* edge, we have to exhaust e_2 without knowing $\#_{S^*}(e_2)$ before we use e_1 for the last time. We claim that we can recognize when the edge $e_2 = (v)$ is exhausted, or in other words, when we visit v for the last time.

We know that e_2 is part of at least $\#_{S^*}(e_1)$ interchangeable cycles. For every traversed interchangeable cycle $C = (v, \ldots, v)$, we know for all outgoing adjacent edges if they are used in S^*. If there is one non-exhausted edge $e = (x, y)$, reachable by a path starting at v using e_2, the edge e_2 cannot be exhausted. Due to the definition of *last* edges (see Definition 12) the last traversal of e_1 will lead to a vertex u that is not reachable via e_2. It is also not possible that the subsequence exploring u leads to x such that the edge e is part of this sequence, leaving v for the last time. This would imply that there are two interchangeable sequences s_1, s_2, both starting at v and ending at x, one using e_1 and containing u and the other one using e_2, as presented in Figure 3.4. Being part of such an interchangeable sequence violates the definition of a *last* edge and contradicts the assumption of e_1 being one. Thus, the edge e cannot be part of the last traversal of e_1 and its vertices must be explored using e_2. Therefore, we can recognize that e_2 is exhausted when all edges reachable by a sequence that starts at v and uses e_2 are exhausted as well.

On the other hand, if we know that e_1 is not a *last* edge, we can use it once more and exhaust it. We also know that, for all following visits of v, we have to use e_2. For the sake of contradiction, assume the edge e_2 is the first edge in our search sequence that is used more often than in S^*. The edge e_1 was already used $\#_{S^*}(e_1)$ times. When our explorer traverses e_2 as often as it is traversed in S^*, all incoming edges are exhausted. To use the edge e_2 once more, the explorer has to traverse a cycle, leading back to v. But because all incoming edges are already exhausted, we would need to use one edge on the way to v more often than it is used in S^*. This contradicts our assumption that e_2 was the first edge which is used more often than in S^*. □

Using Theorem 7, we are able to extend a sequence such that every edge is used as often as in the fixed optimal solution S^*. Since G_{S^*} is strongly connected, we will not leave any parts of the graph unexplored. For every vertex the explorer visits, the adjacent edges will be eventually exhausted.

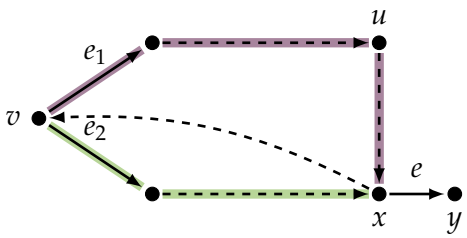

Figure 3.4: A situation violating the condition that e_1 is a *last* edge. The dashed edges represent some sequences of vertices and the colored paths from v to x are the two interchangeable sequences s_1 and s_2.

If an unvisited vertex v in our exploration sequence existed, no visited vertex would have an outgoing edge that leads to v. This violates the Eulerian property of G_{S^*}. Therefore, the algorithm will compute an optimal exploration sequence.

It remains to count how many bits of advice are required to acquire the knowledge of the *light* edge, its number of traversals, and the *last* edge, for every vertex in G, needed for Theorem 7. Note that it is possible that a vertex with two outgoing edges does not have a *last* edge because for both edges there exist solutions in $\mathcal{S}$ where the edge is used on the last visit. However, for our algorithm it makes no difference whether the *heavy* edge is a *last* edge as defined in Definition 12 or whether there is no *last* edge. In both cases, we can exhaust the *light* edge.

Lemma 19. *For a given graph $G_{S^*} = (V, E_{used})$ with an outdegree bounded by two and a multi-edge-optimal exploration sequence S^*, the algorithm needs at most $2n$ bits of advice to learn about the light edge and which edge is* last *in S^*, for all vertices.*

Proof. For two outgoing edges e_1 and e_2 which are used in an optimal solution S^*, the algorithm asks whether $\#_{S^*}(e_1) < \#_{S^*}(e_2)$ holds. Or in other words, the algorithm asks whether the edge e_1 is light. This can be determined with one bit of advice. Additionally, the algorithm asks for every vertex with two outgoing edges which of them should be used for the *last* time to leave the vertex. The algorithm will ask these two binary questions at most once per vertex in the graph. □

The upper bound on the number of advice bits required to communicate the number of traversals for all *light* edges is more technically involved. From Lemmas 13 to 15 and Corollary 2, we know that the graph induced by E_{multi} is a forest F. We call the edges from E_1 that are incident to a vertex $v \in V(F)$ *entry* edge if they are directed towards v and, analogously, *exit* edge otherwise. The trees in F can be degenerated, like in Figure 3.5(a),

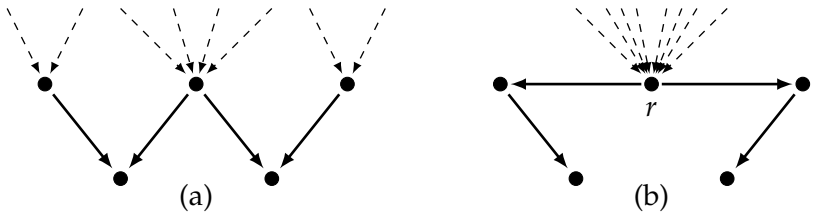

Figure 3.5: (a) shows a degenerate tree of multi-edges in F. (b) shows how the degenerated tree of multi-edges from (a) can become a rooted tree if the center vertex is chosen as the root. In both figures the entry edges are dashed.

i.e., the subsequences of S^* traversing the tree can be edge- or even vertex-disjoint. This lack of structure needs to be overcome.

To show that the sum of traversal numbers of all *light* edges in a tree of multi-edges $T \subseteq F$ is bounded, we construct a new tree T^r, which we call a *rooted tree* in the following, and a corresponding exploration sequence S'. In this *rooted tree*, all *entry* edges are collected at one root vertex r and, from this vertex, all leaves are reachable. An example is shown in Figure 3.5(b). The new exploration sequence ensures that the *rooted tree* T^r is traversed in the correct order and the remaining graph is traversed as it is traversed in S^*. This more structured tree allows us to count the required advice bits in a recursive way, starting at the root. We can use the bound on the sum of the number of traversals for the *light* edges in T^r also for T if there is a bijection $b\colon E(T) \to E(T^r)$ such that $\#_{S^*}(e) \leq \#_{S'}(b(e))$ holds, for all *light* edges $e \in E(T)$.

Lemma 20. *Let G be a graph where every vertex has at most two outgoing edges, let $S^* \in \mathcal{S}$ be a multi-edge-optimal exploration sequence, let F be the forest induced by the edges that are used multiple times in S^* and let $T \subseteq F$ be connected. We can create a* rooted tree *T^r and an exploration sequence S' such that there is a bijection $b\colon E(T) \to E(T^r)$ with $\#_{S^*}(e) \leq \#_{S'}(b(e))$, for all* light *edges $e \in E(T)$.*

Proof. As root vertex in T^r, we choose a vertex in T that has no incoming edges and the largest number of vertices in its subtree. Figure 3.6 visualizes our iterative approach where we replace one edge with another until all vertices are reachable from r. If there is a vertex v on a path $p = (r, \ldots, u, v)$ that has another incoming edge (w, v), the vertex w is not reachable by a path starting at r. We take the ancestor of w with no incoming edges, we call it w_{pre}, add an edge (u, w_{pre}), and delete the edge (u, v). We set the number of traversals of (u, w_{pre}) in S' to $\#_{S^*}((u, v))$. Thus, the new edge is traversed as often as the deleted edge. The numbers of traversals for all the edges on the path $w_{pre}, \ldots, w, v$ are increased by $\#_{S^*}((u, v))$.

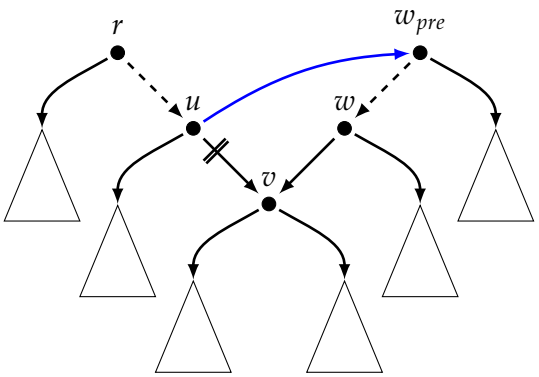

Figure 3.6: One step in the construction of T^r from Lemma 20. The dashed edges represent paths and the crossed out edge is removed and replaced by the blue edge (u, w_{pre}).

We repeat this procedure until there are no more vertices without incoming edges except for the root r. By construction, for every edge at a vertex u that we delete, we add at least one with a higher number of traversals in S'. Thus, the number of edges in both trees is the same. Additionally, for all edges e that occur in both trees, $\#_{S^*}(e) \leq \#_{S'}(e)$, which concludes the proof. □

The following two lemmas give upper bounds on how often a rooted tree is visited in S'. Since a rooted tree is visited as often as its original tree, we also obtain a bound on the number of visits of the original tree. At first, we look at a single rooted tree. Afterwards we extend the result to a forest of rooted trees and give a bound on how often the rooted trees are visited altogether.

Lemma 21. *In a rooted tree T^r constructed from a tree of multi-edges $T \subseteq F$, the root vertex is visited at most $\frac{n}{2} + 1$ times.*

Proof. In the construction used in the proof of Lemma 20, all incoming traversals into the tree T are shifted to the root vertex of T^r. Thus, it is visited as often as the tree T has *entry* edges. Since S^* is Eulerian, the number of *entry* edges is exactly as large as the number of *exit* edges. We count the number of *exit* edges in T^r.

An *exit* edge can be attached to an inner vertex or to a leaf vertex. We denote the number of *exit* edges incident to a leaf as e^l and the number of *exit* edges incident to an inner vertex as e^i. They depend on the number of leaves n^l and the number of inner vertices n^i. When we interpret the *exit* edges as leaves in a binary tree, we obtain

$$n^l + n^i \geq e^l + e^i - 1 \tag{3.2}$$

as an upper bound on the number of *exit* edges.

Every *exit* edge is part of a cycle that visits a new vertex for the first time, except for one edge, leaving the tree for the last time. Thus, the number of *exit* edges plus the number of vertices in the tree is at most the number of all vertices n. Using (3.2), we obtain

$$n \geq e^l + e^i + n^l + n^i - 1 \overset{(3.2)}{\geq} e^l + e^i + e^l + e^i - 1 - 1 = 2(e^l + e^i) - 2$$

and thus

$$\frac{n}{2} + 1 \geq e^l + e^i$$

as a relation between the number of vertices and the *exit* edges in the tree. Therefore, the root vertex in T^r is visited no more than $\frac{n}{2} + 1$ times. □

Lemma 22. *In a forest of multi-edges F, containing x rooted trees $T_1^r, \ldots, T_x^r$, where the root vertex r_j is visited k_j times, the sum of visits is bounded by $\sum_{j=1}^{x} k_j \leq \frac{n+3x}{2}$.*

Proof. As before, we count the number of outgoing edges, for each tree, to know how often the root vertices will be visited altogether. We use the same notation as in the proof of Lemma 21, and observe that (3.2) holds for every rooted tree $T_j^r \in F$.

Note that, in the case of multiple trees, there can be *exit* edges of one tree that are *entry* edges of a different tree. These edges do not visit a new vertex outside any tree and are similar to the edges that are used to leave some tree for the last time. Thus, their number must be subtracted when comparing the number of outgoing edges with the total number of vertices in the graph. For every tree, there can be at most two of these edges. If there are more than two, we can find a cycle containing only *entry*, *exit* and multi-edges that does not visit a new vertex (the cycle does not have to be directed, see Lemma 15). One of the two edges is an *exit* edge of a tree T_i^r and the *entry* edge of another tree T_j^r, which is used to visit T_j^r for the first time. The other edge is the *exit* edge of T_j^r, leaving it for the last time and maybe leading back to the first tree.

By applying Equation (3.2), we obtain the following dependency between the number of *exit* edges, the number of trees in F and the number of vertices in the whole graph:

$$\begin{aligned} n &\geq \left(\sum_{j=1}^{x} e_j^l + e_j^i + n_j^l + n_j^i \right) - 2x \overset{(3.2)}{\geq} \left(\sum_{j=1}^{x} 2(e_j^l + e_j^i) - 1 \right) - 2x \\ &= \left(2 \sum_{j=1}^{x} e_j^l + e_j^i \right) - 3x \,. \end{aligned}$$

This proves that we can use $\frac{n+3x}{2}$ as an upper bound on the number of *exit* edges in F. Thus, the rooted trees altogether are not visited more often than $\frac{n+3x}{2}$ times. □

We claim that the number of traversals for all *light* edges in a *rooted tree* can be communicated with linearly many bits in total, measured in the number of vertices.

The number of traversals of the root vertex r has to be distributed to the outgoing edges. Figure 3.7(a) shows how the number of traversals for a vertex that is visited y times is distributed to the next level. The algorithm asks, at the vertex v, for the number of traversals of the *light* edge x_1 and this number will strongly influence the number of traversals for the successors of v. When the algorithm moves the explorer onto one of them, it asks again for the number of traversals for the *light* edge which again influences the number of traversals for its successors, and so on. Figure 3.7(b) shows how the number of queries increases on each level and, on the other hand, how the number of required bits decreases with each iteration. Thus, we formulate a recursive function $g(y)$ that describes the advice costs of these queries.

For a number of traversals y at a vertex v, the algorithm asks for the number of traversals of the *light* edges and thus for a number x with $2 \leq x \leq \frac{y}{2}$. On the next level, we have to call this function two times. Once for $y - x$ and another time for x, because these are the traversal numbers for the successors. We want to know how many advice bits are required to encode the number of traversals for the *light* edges of T^r. The function $g(y)$ becomes zero for all values $y < 2$ because these vertices are not part of the tree T^r. For y with $2 \leq y \leq 3$, we need two bits of advice because the *light* edge is used at most once. Thus, we end up with the following recursive

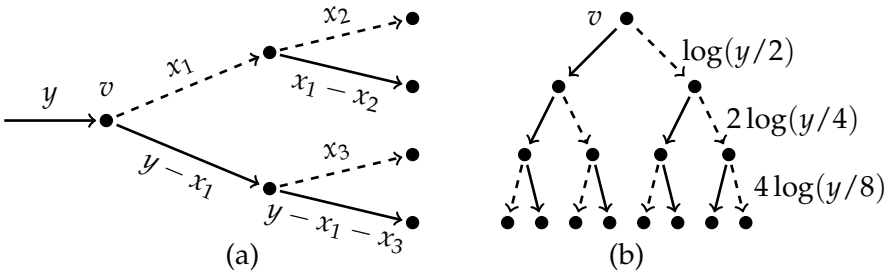

Figure 3.7: The incoming number of traversals y at the vertex v needs to be split up between the two outgoing multi-edges. This continues recursively downwards in the tree. An edge is drawn dashed if it is *light*.

function

$$g(y) = \max_{2 \leq x \leq \frac{y}{2}} \{2\lceil \log(\lceil \log(x+1) \rceil) \rceil + \lceil \log(x+1) \rceil + g(y-x) + g(x)\}$$

with $g(1) = 0, g(2) = 2, g(3) = 2$. Recall that the function $g(y)$ describes the number of advice bits needed to get the number of traversals for all *light* edges in a single rooted tree T^r.

Lemma 23. *The recursive function*

$$g(y) = \max_{2 \leq x \leq \frac{y}{2}} \{2\lceil \log(\lceil \log(x+1) \rceil) \rceil + \lceil \log(x+1) \rceil + g(y-x) + g(x)\}$$

with $g(1) = 0, g(2) = g(3) = 2$, *is bounded by* $g(y) \leq \frac{9}{2}y - 3\lceil \log(y+1) \rceil - 1$.

Proof. We prove the statement by induction. For $y = 4$, the only possible value for x is 2. In this case, both sides of the equation match:

$$2\lceil \log(\lceil \log(3) \rceil) \rceil + \lceil \log(3) \rceil + g(2) + g(2) = 8 \leq \tfrac{9}{2} \cdot 4 - 3 \cdot 3 - 1 = 8\,.$$

Now we assume that the statement holds for all natural numbers smaller than y. So, we can apply our induction hypothesis to bound the function $g(x)$ for all values $x < y$ and obtain

$$\begin{aligned} &\max_{2 \leq x \leq \frac{y}{2}} \{2\lceil \log(\lceil \log(x+1) \rceil) \rceil + \lceil \log(x+1) \rceil + g(y-x) + g(x)\} \\ \leq &\max_{2 \leq x \leq \frac{y}{2}} \{2\lceil \log(\lceil \log(x+1) \rceil) \rceil + \tfrac{9}{2}y - 3\lceil \log(y-x+1) \rceil - 2\lceil \log(x+1) \rceil - 2\}\,. \end{aligned}$$

We note that $f(x) = 2\lceil \log(\lceil \log(x+1) \rceil) \rceil - 2\lceil \log(x+1) \rceil \leq -2$ holds for $x \geq 2$ because the function $f(x)$ is strictly monotonically decreasing and, for $f(2) = 2 \cdot 1 - 2 \cdot 2 = -2$, equality holds. This implies

$$\begin{aligned} g(y) &\leq \max_{2 \leq x \leq \frac{y}{2}} \{\tfrac{9}{2}y - 3\lceil \log(y-x+1) \rceil - 4\} \\ &\leq \tfrac{9}{2}y - 3\lceil \log(\tfrac{y}{2}+1) \rceil - 4 \\ &\leq \tfrac{9}{2}y - 3\lceil \log(y+1) \rceil - 1\,. \end{aligned}$$

□

Thus, to encode the number of traversals of all *light* edges in one rooted tree T^r, with the largest amount of traversals, $g(\frac{n}{2}+1) \leq \frac{9n+14}{4} - 3\lceil \log(\frac{n}{2}+2) \rceil \leq 2.25n$ advice bits are sufficient. To get a bound on the number of traversals of all *light* edges in the forest of multi-edges from S^*, we need to apply Lemma 22.

Lemma 24. *For a given graph* $G_{S^*} = (V, E_{used})$ *with out-degree bounded by* 2 *and a multi-edge-optimal exploration sequence* S^*, $2.25n$ *advice bits suffice to learn the exact traversal numbers for the* light *edges in the forest of multi-edges.*

Proof. Using the construction from Lemma 20 for every tree of multi-edges, we get a forest of rooted trees. For each rooted tree, the number of traversals for the *light* edges is at least as large as in the original graph. Without loss of generality, the root vertex of any of these trees is visited at least four times $k_j \geq 4$, because otherwise we would not read more than two bits of advice per vertex. Together with the results from Lemmas 22 and 23, we obtain the following bound:

$$\begin{aligned}\sum_{j=1}^{x} g(k_j) &\overset{\text{Lemma 23}}{\leq} \sum_{j=1}^{x} (\tfrac{9}{2}k_j - 3\lceil \log(k_j+1)\rceil - 1) \\ &= \sum_{j=1}^{x} ((\tfrac{9}{2}k_j)) - \sum_{j=1}^{x} (3\lceil \log(k_j+1)\rceil) - x \\ &\overset{\text{Lemma 22}}{\leq} \tfrac{9}{2} \cdot \tfrac{n+3x}{2} - \sum_{j=1}^{x} (3 \cdot 2) - x \\ &= \frac{9n+27x}{4} - 7x = \frac{9n-x}{4} \leq 2.25n\,.\end{aligned}$$

□

Note that the $2.25n$ bits of advice suffice only for the vertices that are part of the forest of multi-edges. For every vertex not part of the forest, the algorithm will read two bits of advice to get the number of traversals for the adjacent *light* edge. The number of required advice bits is maximized when the tree of multi-edges is as large as possible. Lemma 21 implies that the number of remaining vertices in a maximized tree is at most $\frac{n}{2}$.

Theorem 8. *There exists an online algorithm which solves the cyclic graph exploration problem using* $5.25n - 1$ *bits of advice on a given unknown directed graph* $G = (V, E, cost)$ *with outdegree bounded by* 2*.*

Proof. For our algorithm, the knowledge about the *light* edge and its correct number of traversals and the knowledge which edge is *last* are sufficient to compute an optimal exploration sequence. The number of traversals can be determined independently of the cost function. We use the strategy described in the proof of Theorem 7 for our algorithm and sum up the number of required advice bits. Note that we do not need any advice when the explorer visits the last vertex for the first time, due to Lemma 17. The advice bits are used as follows:

- $2(n-1)$ advice bits to know which edges are *light* and if they are *last*.
- $2.25n$ advice bits to know the traversal number for the *light* edges in E_{multi}.
- $n+1$ advice bits to know the traversal number for the *light* edges in E_0 and E_1.

For every vertex in G_{S^*} with two outgoing edges, the algorithm reads one bit of advice to know which edge is marked as *light*. Additionally, it asks if the *light* edge is *last* (Definition 12). Due to Lemma 19, we know that this information consumes $2n$ bits of advice.

The remaining advice bits are used as the traversal numbers for the *light* edges. We count the advice in two steps, at first we look at the *light* edges that are also multi-edges, i.e., the *light* edges in the forest of multi-edges. Due to Lemma 20, we know that we can rearrange the edges in a tree T_j such that it becomes a rooted tree T_j^r. Every path to a leaf in T_j^r must contain the root and the number of traversals is at least as large as it was in T_j. Due to Lemmas 22 and 24, we know that the overall number of bits to encode the number of traversal, for all *light* edges in a forest of rooted trees is at most $2.25n$. This is also an upper bound for the number of required bits in the original graph without rooted trees.

For the vertices that are not adjacent to a multi-edge, the *light* edge is used once or never. Thus, the algorithm will read two bits of advice for these vertices. The adversary maximizes the required advice by maximizing the size of the multi-edge forest, but doing this decreases the number of vertices that are not adjacent to a multi-edge. In the proofs of Lemmas 21 and 22, we saw that there are at most $\frac{n+1}{2}$ vertices that are not adjacent to a multi-edge, when the multi-edge forest is maximized. Therefore, $n+1$ advice bits are needed to get the number of traversals for the *light* edges that are not part of the multi-edge tree. Thus, with $3.25n+1$ bits of advice all traversal numbers for the *light* edges can be communicated. Note that, if the graph does not contain a tree of multi-edges, $2n$ bits of advice suffice for this knowledge.

In total, our algorithm will never read more than $5.25n-1$ bits of advice. □

3.4.2 Adaption for Increasing Degree

We now explain how our algorithm can be adapted to solve the graph exploration problem on general unknown directed graphs. In order to count the number of advice bits more easily, we transform the given graph $G=(V,E)$ of unbounded degree into a graph $H=(V\cup V', E')$ with out-degree bounded by 2, which has more vertices and edges. To be more precise, H is implicitly constructed from G during the traversal. The algorithm and the oracle agree on this construction and the oracle provides the advice for H. One step in G will be represented by a sequence of steps in H.

To construct the graph H, we introduce *out-trees*, which are used to replace a large number of directed edges.

Definition 13. *An* out-tree T_v *is a directed binary tree, directed from the root* v *to the leaves, that minimizes the maximum distance between* v *and some leaf.*

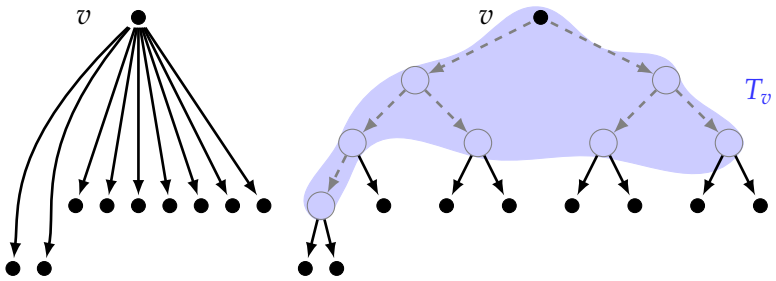

Figure 3.8: A vertex with nine outgoing edges and the resulting out-tree.

The algorithm replaces every vertex v with more than two outgoing edges by a tree T_v. Figure 3.8 shows a vertex with nine outgoing edges and the resulting out-tree T_v. The gray edges and vertices in Figure 3.8 are part of T_v, and are called virtual edges/vertices. The black edges represent the edges from G. Note that the virtual vertices are not visited when the outgoing edges of T_v, representing the edges from G, are not used in S^* on G. Thus, the exploration sequence on H does not need to visit every virtual vertex, only the vertices from V have to be visited at least once.

The construction of H itself does not need any advice, but the increased number of edges and vertices influences the number of advice bits used overall. Therefore, we now analyze how the number of vertices and edges change.

Lemma 25. *When a graph $G = (V, E)$ with an optimal exploration sequence S^* is transformed into a graph $H = (V \cup V', E')$, with $|V \cup V'| = n'$ and $|E'| = m'$, such that every vertex has an outgoing degree of at most 2 by replacing vertices v with an out-tree T_v, the number of vertices and edges of H are bounded by $n' \leq m$ and $m' \leq 2m$.*

Proof. As soon as the algorithm moves the explorer onto a vertex v with $x > 2$ outgoing edges for the first time, it separates the edges into two sets A and B with $|A| = \lceil \frac{x}{2} \rceil$ and $|B| = \lfloor \frac{x}{2} \rfloor$. The edges from A are shifted to a new virtual vertex v_A and the edges of B are shifted to another new virtual vertex v_B. Additionally, the virtual edges (v, v_A) and (v, v_B) are added. This procedure is repeated for every vertex, virtual or not, with more than two outgoing edges.

For a vertex v with $x > 2$ outgoing edges, we need $x - 2$ virtual vertices in the out-tree replacing v. This can be proven by induction over the degree of v in G: A vertex v with $x = 3$ outgoing edges $(v, v_1), (v, v_2), (v, v_3)$ gets only one additional virtual vertex u. The out-tree T_v contains the following edges $(v, v_1), (v, u), (u, v_2), (u, v_3)$. Assume that we have a vertex with x outgoing edges and an out-tree with $x - 2$ virtual vertices that replace the edges. For the induction step, we need to attach one new virtual vertex to a

leaf on the lowest possible level to create an out-tree for a vertex with $x+1$ outgoing edges. This replaces one outgoing edge of the tree with a new leaf with two additional exits.

We call the number of vertices with only one adjacent outgoing edge n_1 and the number of edges attached to these vertices m_1. Analogously, we call the number of vertices with two or more outgoing edges $n_{>1}$ and the number of edges adjacent to these vertices $m_{>1}$. The number of virtual vertices in all out-trees is at most

$$m_{>1} - 2n_{>1} = m - m_1 - 2n_{>1} = m - n_1 - 2n_{>1} = m - n - n_{>1} \leq m - n.$$

The root vertices of the out-trees are the old vertices of G and are also part of H. Therefore, the number of vertices in H is at most $n' \leq m - n + n = m$. Every vertex in H has at most two outgoing edges. Thus, the number of edges m' in H is bounded from above by $2m$. □

Although the construction increases the number of vertices and edges, the computed solution for H can be easily modified to obtain a solution for G.

Lemma 26. *Let G be an arbitrary directed graph and let H be the degree-bounded graph resulting from the construction described in Lemma 25 with an optimal exploration sequence S. For every $w_1, \ldots, w_k \in V(H) \setminus V(G)$, we replace every path $(v, w_1, \ldots, w_k, u)$ in S by an edge $(v, u) \in E(G) \setminus E(H)$ yields an optimal exploration sequence S^* for the graph G.*

Proof. From an optimal exploration sequence S for H, we can construct an optimal exploration sequence S^* for G by removing all virtual vertices in S. Every path $(v, \ldots, u)$ that leads from a root v in a constructed out-tree T_v to a leaf u can be replaced with the edge (v, u) which must be part of G by construction. Thus, if S is optimal for H, S' must be an exploration sequence of minimal cost for G. □

Note that the important information, which edge is used for the last traversal and which edge is light, can be communicated with two bits per (virtual or real) vertex, such that we can apply Theorem 8. Obviously, we also increased the number of vertices by transforming G into H and therefore also the number how often the algorithm asks for such information is increased. Therefore, we need to compute how transforming G into H increases the needed advice if we apply the same approach as in Theorem 8.

Theorem 9. *There exists an online algorithm which solves the graph exploration problem using $5.25m - 1$ bits of advice on a given unknown directed graph $G = (V, E, cost)$.*

Proof. We use the algorithm described in Theorem 8 and make one adjustment: The algorithm and the oracle use the construction from Lemma 25 to ensure that every vertex has at most two outgoing and at most two incoming edges.

This increases the advice complexity for the *last* and *light* edges from $2(n-1)$ to at most $2(m-1)$, for the number of traversals for the *light* edges that are part of a multi tree from $2.25n$ to $2.25m$, and the advice for the number of traversals for the *light* edges that are not part of a multi tree is increased from n to m. Thus, the algorithm needs $5.25m - 1$ bits of advice in total. The computed solution for the transformed graph can then be transferred to G by applying Lemma 26. □

3.4.3 Non-Cyclic Graph Exploration

In order to solve the non-cyclic graph exploration problem as well, it suffices to discuss the differences in the last steps of our algorithm. The last vertex that is visited is always visited only once (see Lemma 16). Thus, it has only one incoming edge in the graph induced by the exploration sequence, which must be used only once and *last*. Recall that our algorithm avoids to exhaust *last* edges as long as possible and will exhaust the other incident edge first. For the non-cyclic graph exploration problem, this last vertex has no outgoing edge that is used in the optimal exploration sequence. In this case, it is not defined, which outgoing edge is *light* or *last*, but such a definition is also not needed.

The algorithm is capable of detecting the last vertex in the optimal exploration sequence without additional advice and will not query the oracle regarding the outgoing edges of the last vertex (see Lemma 17). When the algorithm moves the explorer onto this vertex, it knows that there is no remaining unvisited vertex, because there is no edge incident to an already visited vertex that leads to a new vertex. Thus, our algorithm will terminate without reading advice for the last vertex.

Theorem 10. *There exists an online algorithm which solves the non-cyclic graph exploration problem using* $5.25n - 1$ *bits of advice on a given unknown directed graph* $G = (V, E, cost)$ *with outdegree bounded by* 2*.*

Theorem 11. *There exists an online algorithm which solves the non-cyclic graph exploration problem using* $5.25m - 1$ *bits of advice on a given unknown directed graph* $G = (V, E, cost)$*.*

3.5 Lower Bounds for the Advice Complexity on Directed Graphs

We contrast our previous results with lower bounds for the advice complexity of the graph exploration problem (cyclic and non-cyclic) on directed graphs. First we present a lower bound for graphs with outdegree bounded by 2, the same graph class for which we designed our first algorithm. For the sake of completeness we also present an already known lower bound for directed graphs with arbitrary outdegree.

3.5.1 Directed Graphs with Outdegree Two

In the following, we look at the number of advice bits that is required to solve the graph exploration problem optimally and show that, for directed graphs with outdegree two, the number of advice bits must be linear in the number of vertices n. More precisely, we show that there cannot exist an algorithm that solves every instance of the non-cyclic graph exploration problem with less than $1.25n$ bits of advice. Afterwards, we extend this result to the cyclic graph exploration problem.

As a preparation, we first show a weaker lower bound of $n - 2$ on the number of advice bits. For this, we construct a family of instances $\mathcal{G}$ such that, for every instance in $\mathcal{G}$, the algorithm has to choose between two indistinguishable vertices in every step (except the last two). Every instance in the family $\mathcal{G}$ requires a unique sequence of decisions to obtain an optimal solution. One wrong decision forces the algorithm to skip vertices and an optimal solution cannot be achieved anymore. So, in the end we will have at least 2^{n-2} different instances, each with a unique optimal solution.

Before we prove this intermediate result, we discuss the idea of the construction of $\mathcal{G}$. Every instance can be separated into k layers and the agent starts on the first vertex of layer k. In Figure 3.9, an intermediate layer i is shown. The green edges are part of the optimal exploration sequence and lead through one layer, visiting every vertex, and from the last vertex of the layer to the first vertex of the next layer. The red edges start in layer i and point to a vertex on a layer $j < i$, representing a wrong decision because they skip the remaining vertices on layer i. The dashed edges are from previous layers $j > i$ and are misleading edges, like the red ones, but for previous layers.

The last three layers are visualized in Figure 3.10. Here, the red, violet and orange edges indicate the misleading edges, the color depends on the layer on which the misleading edge starts. The green edges indicate the unique optimal solution.

The edges in our construction ensure that, for every every vertex v on a layer i (except for the last two vertices of the graph), the following properties are satisfied:

(a) For every layer j, with $k \geq j > i$, there exists exactly one vertex that has an outgoing edge pointing to v.

(b) The two vertices in the *out-neighborhood* of v have the same number of incoming edges that have already been visible when v is visited for the first time.

(c) One vertex from the *out-neighborhood* of v is on a lower level than v and the other one is on the same level i or it is the starting vertex of the next layer (if v is the last vertex of a layer).

To achieve property (a), the number of vertices on layer k must be half the total number of vertices. In the following, the number of vertices is halved on every layer, down to two. Layer 1 is an exception as it always contains three vertices, as shown in Figure 3.10.

For property (c), it is important that there is an instance in the family of graphs $G \in \mathcal{G}$ for every possible mapping between the misleading edges and the vertices on lower levels. Otherwise, the algorithm can identify the given instance after traversing the first layer due to the ordering in which the unvisited vertices (the endpoints of the misleading edges) reappear. Only if, for every possible decision, there is an instance where the same decision leads to a mistake, the algorithm cannot gain any knowledge about the given instance. We achieve this by adding an instance for every possible permutation between the misleading edges of a layer and its lower layers. This means that the family of instances contains

$$\prod_{i=1}^{k} \left\lceil \frac{n}{2^i} \right\rceil ! = \prod_{i=1}^{k} (2^{k-i} + 1)!$$

instances because every layer contains one vertex less than all lower levels together and the number of permutations for n elements is $n!$.

The main idea of the proof is that, if the algorithm does not ask for one bit of advice in every step (except the last two), there are at least two instances that are treated equally although their optimal solution differs in at least one step.

Lemma 27. *Every online algorithm with advice needs at least $n - 2$ bits of advice to solve the non-cyclic graph exploration problem on a graph with outdegree bounded by 2 and $n = 2^k + 1$ vertices.*

Proof. We construct a family of graphs $\mathcal{G}$ such that, for all instances in $\mathcal{G}$, the algorithm cannot know which instance the adversary chooses. So, during every step of exploration (except the last two) there are always multiple instances in $\mathcal{G}$ that differ only in the not yet explored part of the graph. Thus, for every step there exists at least one instance in $\mathcal{G}$ which has the same optimal solution up to this point, but differs in the next step. The

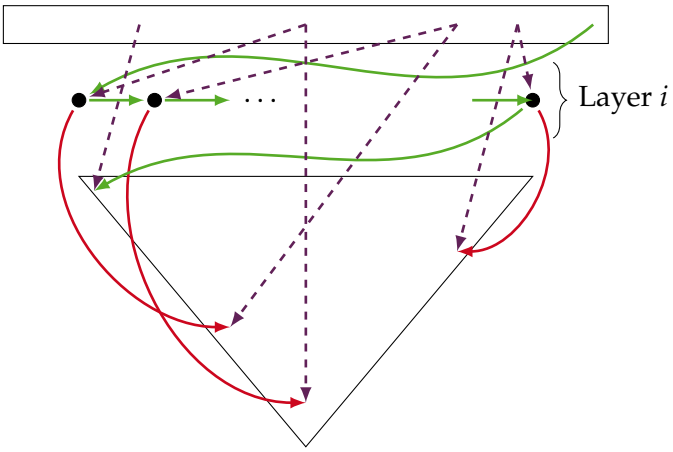

Figure 3.9: A schematic drawing of the construction from the proof of Lemma 27. The rectangle on top represents the already explored part of the graph. In the center, one layer i is shown. The triangle below represents the part of the graph that is not yet explored by our algorithm. Every vertex on layer i has the same number of incoming edges (except the first vertex on the layer) and exactly one outgoing edge to a vertex on a lower layer (except the last vertex on the layer). The green edges are part of the optimal exploration sequence. The dashed edges come from previous layers and the red edges are edges that should not be used when traversing layer i.

main idea why the algorithm cannot know which instance the adversary chooses is described in condition (b). Note that the optimal solution, for every graph in $\mathcal{G}$, visits every vertex only once and is unique.

In the following, we give a detailed explanation of our construction. Each graph in $\mathcal{G}$ is separated into k layers and has $n = 2^k + 1$ vertices. To compute the optimal solution, the algorithm needs to visit the vertices of one layer in the correct order and can only start with the next layer if every vertex in the current layer is visited. Figure 3.10 shows an example for $k = 3$. Each layer i contains 2^{i-1} vertices, for $2 \leq i \leq k$. The layer $i = 1$ is an exception and contains three vertices, as shown in Figure 3.10.

To describe the set of edges for an instance in $\mathcal{G}$, we name the vertices depending on their layer and their position in the ordering in which the optimal exploration sequence visits them. We call the vertices $v_{r,i}$, with $i \in \{1, \ldots, k\}$ being the layer a vertex is contained in and r enumerating the vertices in each layer in the same order in which the optimal solution visits them. Obviously, the algorithm does not know these names. A vertex $v_{r,i}$ has an edge to the vertex $v_{r+1,i}$ and one to a vertex $v_{x,j}$ with $j < i$ and x according to some permutation (see property (c)). The last vertex in each layer $v_{2^{i-1},i}$ does not have an edge to another vertex on the same layer. In-

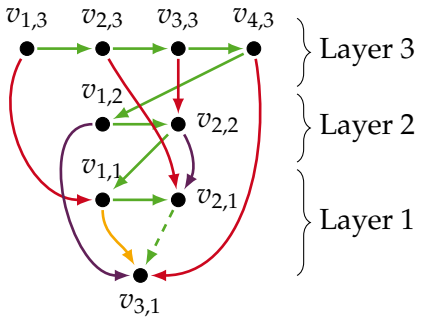

Figure 3.10: An example of the construction explained in the proof of Lemma 27, for $k = 3$ layers. The green edges are part of the unique optimal exploration sequence. The other edges are all misleading edges. The red edges have their starting point in layer three, the violet edges start in layer two and the orange edge starts in layer one.

stead, it has an edge to the vertex $v_{1,i-1}$ and one to another vertex $v_{x,j}$ with $j < i$ and x according to some permutation.

Thus, for every layer $i > 1$, every vertex on layer i points to a different vertex on a layer $j < i$, except for the last vertex of the layer, which points to two vertices on lower layers. Which vertex points to which is taken care of by a permutation which also ensures that the graph induced by the edges that start in one layer i to a lower one form a perfect matching (except for the last vertex of the layer, which is matched to the first vertex of the next layer $v_{1,i-1}$ and another vertex on a layer $j < i$). The number of vertices on a layer $i > 1$ is 2^{i-1}, which is one vertex less than there are on all lower levels together

$$2 + \sum_{j=1}^{i-1} 2^{j-1} = 2 + \sum_{j=0}^{i-2} 2^j = 2 + 2^{i-1} - 1 = 2^{i-1} + 1 \,.$$

This perfect matching between the vertices of one layer and the vertices on all lower levels also ensures that every vertex on a layer $j < i$ has exactly one incoming edge from a vertex on layer i (see property (a)).

Now we discuss the special case of the last layer $i = 1$. As presented in Figure 3.10, the vertex $v_{1,1}$ has edges to $v_{2,1}$ and $v_{3,1}$, and $v_{2,1}$ has only one edge pointing to $v_{3,1}$. When the algorithm does not make a wrong decision and the agent reaches the vertex $v_{2,1}$, only one vertex remains unexplored. Therefore, the algorithm can extend the current solution to an optimal one without any additional information. When the last vertex $v_{3,1}$ is reached, the computation is completed. Thus, for the last two vertices, no additional information is required and the algorithm can compute the optimal solution if there was no mistake made until the second to last vertex.

Now, we will explain the view of the algorithm and the optimal solution for any graph in $\mathcal{G}$. The explorer starts at the vertex $v_{1,k}$. To compute the optimal solution, all vertices of a layer i must be explored before traversing to the next layer, because there are no edges from a lower layer to a higher one. This is shown in Figure 3.9. Therefore, the algorithm needs to choose the correct edge (marked with green) that leads to the next vertex on the same layer or, if the last vertex in a layer is reached, the first vertex of the following layer. However, in each but the last two steps the algorithm chooses between vertices that were seen equally often. Thus, the algorithm cannot distinguish between the vertices and therefore does not know which is the next one in the optimal sequence. When the explorer is positioned on the second to last vertex of the optimal sequence, only one outgoing edge exists. However for all other vertices, the algorithm has to decide between two vertices with the same number of already seen incoming edges.

Thus, only on the first $n-2$ vertices a mistake is possible. Therefore, 2^{n-2} unique optimal solutions are possible. For each of these, there must be one corresponding instance in $\mathcal{G}$. Thus, for every outgoing edge, except for the ones incident to the last two vertices, there is a graph in $\mathcal{G}$ where the edge is part of the optimal solution. If the algorithm would now compute an optimal solution with less than $n-2$ bits of advice, there must be at least two instances where it makes the same decision in at least one time step. However this is a contradiction to the uniqueness of the solutions for the instances in $\mathcal{G}$. □

Now, we improve the construction from Lemma 27 to get a lower bound of $1.25n$ for the advice complexity. In order to increase the number of needed advice bits, it is necessary that some vertices are visited multiple times. Therefore, we replace every vertex by a cycle of length five, as shown in Figure 3.11. So, a layer i now consists of a set of cycles, each of length five. To compute the optimal exploration sequence, every vertex of a cycle needs to be explored before the explorer moves on to the next cycle. For the last layer in our construction, we reuse the construction from Lemma 27.

The main idea of the proof stays the same. We construct a family of instances $\mathcal{G}$ that can be separated into $k+1$ layers. Properties (a) and (b) still hold for every instance. Property (c) does not hold true anymore because, for each cycle, there is a vertex that has one outgoing edge to a vertex of the same cycle and one to a vertex of the next cycle in the same layer. However, we note that the following property holds true (for every layer except layer 1).

(d) For each cycle c, there are four edges that point to a layer lower than c and one that points to the next cycle on the same level or it points to the first cycle of the next layer (if c is the last cycle of a layer).

But again, we ensure that the algorithm is not able to recognize the

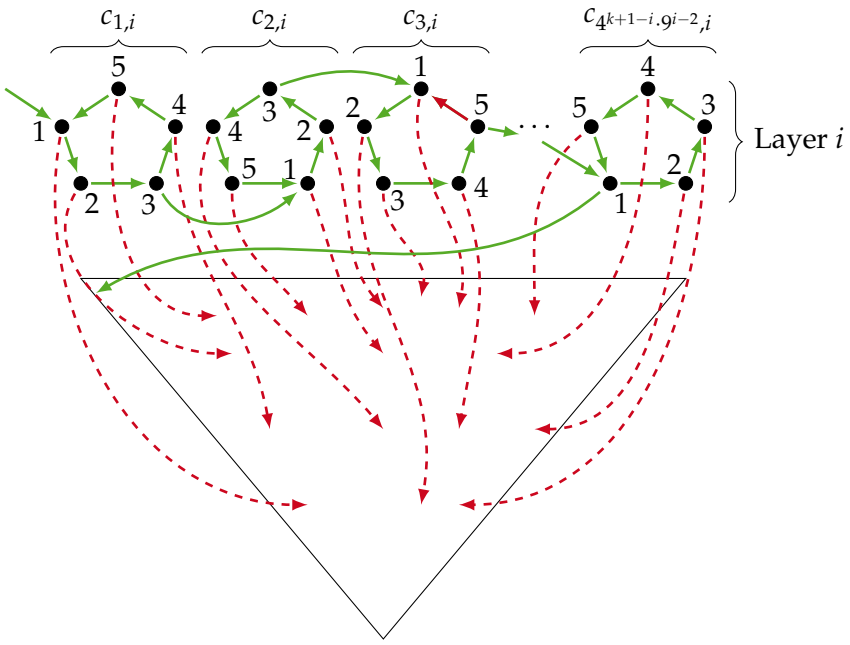

Figure 3.11: A schematic drawing of the construction from Theorem 12. The cycles on top represent one layer i. The numbers on every cycle correspond to the order the vertices are visited in the optimal solution. The triangle below represents the lower layers, the part of the graph that is not yet explored by our algorithm. Every vertex on layer i has one incoming edge from each previous layer $j > i$ and exactly one outgoing edge, marked with red, to a vertex on a lower layer, except for one vertex in each cycle. The other outgoing edges, marked in green, lead either to the next vertex in the cycle or to the next cycle. The green edges are part of the optimal exploration sequence and the red edges are wrong edges that should not be used when traversing layer i.

given instance by the graph structure. This can be achieved by having an instance $G \in \mathcal{G}$ for every possible permutation between the misleading edges in one layer and the vertices in lower levels.

Theorem 12. *Every online algorithm with advice needs at least* $1.25n$ *bits of advice to solve the non-cyclic graph exploration problem on a given unknown graph with* n *vertices and outdegree bounded by two.*

Proof. We construct a family of graphs $\mathcal{G}$ such that, for all instances in $\mathcal{G}$, the algorithm cannot know which instance the adversary chooses. So, during every step of exploration (except the last two) there are always multiple instances in $\mathcal{G}$ that differ only in the not yet explored part of the graph. Thus, for every step there exists at least one instance in $\mathcal{G}$ which has the

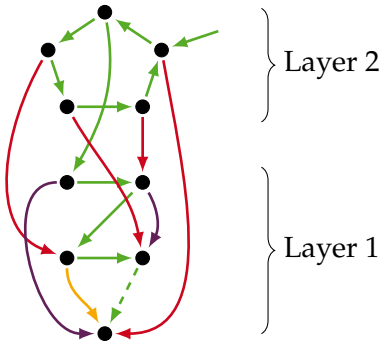

Figure 3.12: An example of the construction explained in Theorem 12, for the number of layers $k+1 = 2$. The green edges are part of the unique optimal exploration sequence. The other edges are all misleading edges.

same optimal solution up to this point and differs in the next step. The main idea why an algorithm cannot know which instance the adversary chooses is the combination of condition (b) and the fact that every instance has a different permutation between the misleading edges of one layer and the vertices on lower layers. Note that the optimal solution is unique, for every graph in $\mathcal{G}$.

In the following, we give a detailed explanation of our construction. Each graph in $\mathcal{G}$ is separated into $k+1$ layers and has $n = 9^k + 1$ vertices. The layers $i > 1$ consist of a set of vertex-disjoint cycles. To compute the optimal solution, the algorithm needs to visit the cycles of one layer in the correct order and can only start with the next layer after every cycle in the current layer is completely visited. Figure 3.12 shows an example for $k = 1$. Each layer i contains $4^{k+1-i} \cdot 9^{i-2}$ cycles of length five, for $2 \leq i \leq k$. The layer $i = 1$ is an exception. The remaining vertices are used to create a graph just like in the construction of the proof of Lemma 27 and this whole construction is then attached as the last layer.

To describe the set of edges for an instance in $\mathcal{G}$, we name the cycles depending on their layer and their position in the ordering in which the optimal exploration sequence visits them. We call the cycles $c_{r,i}$, with $i \in \{2, \ldots, k+1\}$ being the layer a cycle is contained in and r enumerating the cycles in each layer in the same order in which the optimal solution visits them. The vertices on a cycle $c_{r,i}$ will be enumerated with $v_{y,c_{r,i}}$ and $1 \leq y \leq 5$. Obviously, the algorithm does not know these names. One part of the edge set are the cycle edges $(v_{5,c_{r,i}}, v_{1,c_{r,i}})$ and $(v_{y,c_{r,i}}, v_{y+1,c_{r,i}})$, for $1 \leq y \leq 4$. Additionally, there is one vertex in every cycle $c_{r,i}$ that has an edge to the next cycle $v_{1,c_{r+1,i}}$ or, if $c_{r,i}$ is the last cycle of layer i, to the first cycle on the next layer $v_{1,c_{1,i-1}}$. The other four vertices of the cycle $v_{y,c_{r,i}}$,

have edges to vertices on a lower layer $v_{y',c_{r',j}}$ with $j < i$ (see property (d)).

Thus, for every layer $i > 1$, every cycle on layer i points to four different vertices on a layer $j < i$, except for the last cycle of the layer, which points to five vertices on lower layers. Which vertex points to which is handled by a permutation that also guarantees that every vertex on a layer $j < i$ has exactly one incoming edge from layer i (see property (a)). This implies that the number of vertices n_i on a layer i and the number of remaining vertices in lower layers satisfy

$$\frac{4}{5} \cdot n_i + 1 = \sum_{j=1}^{i-1} n_j \, .$$

An example with $k = 2$ contains 82 vertices. The first layer ($i = 3$) consists of 9 cycles (45 vertices), the second layer ($i = 2$) consists of 4 cycles (20 vertices) and the last layer ($i = 1$) contains 17 vertices. Every cycle on a layer, except for the last cycle, has four edges pointing to vertices on lower layer. In the last cycle, five edges have to point to vertices on lower layer. Thus, there have to be $\frac{4}{5} \cdot 45 + 1 = 37$ remaining vertices in the graph without the vertices from the first layer. The construction ensures that every edge has a feasible endpoint and the algorithm has to choose between indistinguishable vertices.

Now we will discuss the special case of the last layer $i = 1$. The last layer consists of

$$n - 5 \left(\sum_{2}^{k+1} 4^{k+1-i} \cdot 9^{i-2} \right) = 9^k + 1 - 5\frac{(9^k - 4^k)}{5} = 4^k + 1$$

vertices which allows us to use the construction from Lemma 27. We know that every algorithm needs at least $4^k - 1$ bits of advice on this part.

Now we will explain the view of the algorithm and the optimal solution for any graph in $\mathcal{G}$. The explorer starts at the vertex $v_{1,c_{1,k+1}}$. To compute the optimal solution, all vertices of the same cycle $c_{1,k+1}$ must be explored before traversing to the next cycle on the same layer. This is shown in Figure 3.11. Therefore, the algorithm needs to choose the correct edge (marked with green) that leads to the next vertex on the same cycle or the following cycle, if the current cycle is explored. For each step in the same cycle, the algorithm chooses between vertices that were seen equally often. Thus, the algorithm cannot distinguish the vertices and therefore does not know which the next one in the optimal sequence is. When one cycle is explored, there are five edges that could lead to the next cycle, but only one is the correct one. Again, these vertices cannot be distinguished (see property (b)).

After explaining the construction and pointing out why the algorithm cannot distinguish the vertices, we have to discuss the advice complexity of our construction. For every cycle, the algorithm has to choose the correct

edge four times. For every possible choice, $\mathcal{G}$ contains a graph, where the corresponding edge is part of the solution. This results in four bits of advice to choose the correct cycle edges. To find the edge that points to the next cycle in the ordering, the algorithm has to choose one out of five possible edges. Again, we ensure that $\mathcal{G}$ contains, for every possible edge, one graph where the edge is the correct choice. Thus, for each cycle, our algorithm needs $\log(5)$ bits of advice to find the correct successor cycle.

The total number of cycles in our graphs is

$$\sum_{i=2}^{k+1} 4^{k+1-i} \cdot 9^{i-2} = \frac{1}{5}(9^k - 4^k) .$$

For every cycle, the algorithm needs in total $(4 + \log 5)$ bits of advice. Together with the $4^k - 1$ advice bits for the last layer (due to Lemma 27), the inequality

$$\frac{1}{5}(9^k - 4^k)(4 + \log 5) + 4^k - 1 \geq 1.25n = 1.25(9^k + 1)$$

must be satisfied, and thus

$$9^k(\log 5 - 2.25) \geq 4^k(\log 5 - 1) + 11.25 .$$

For values $k \geq 4$, the equation holds true. Thus, we have a family of graphs such that every online algorithm with advice needs at least $1.25n$ bits of advice to compute the optimal solution. □

Note that the optimal exploration sequence for any graph in our construction visits a vertex at most twice. To further improve the lower bound, it would require a construction in which the vertices are visited more often. Only if the optimal exploration sequence becomes longer, the algorithm has to make more decisions. However, from previous chapters, we also know that only the decision where the last edge becomes exhausted is critical. Thus, the more often a vertex is visited, the more interchangeable decisions lead to an optimal solution.

To obtain the same lower bound for the cyclic graph exploration problem, we only need to add one edge to our construction, connecting the last vertex with the starting vertex. The optimal exploration sequence stays the same until every vertex was visited once. Then, for the last step, it is necessary that the agent visits the starting vertex again. However, the arguments why the vertices cannot be distinguished stays the same.

Theorem 13. *Every online algorithm with advice needs at least* $1.25n$ *bits of advice to solve the cyclic graph exploration problem on a given unknown graph with outdegree bounded by* 2 *and* n *vertices.*

3.5.2 Lower Bound for Directed Graphs with Arbitrary Outdegree

We show that the number of advice bits to solve the graph exploration problem (cyclic or non-cyclic) on directed graphs is at least in $\Omega(n \log(n))$ thus providing a proof of [40] for the sake of completeness. For this lower bound, we construct a family of graphs $\mathcal{G}_n$, of size $(n-1)!$, where every instance has a unique optimal solution. This is achieved by a permutation π of the vertex set (without the starting vertex), for every instance in our family of graphs.

The permutation is the ordering in which the vertices of the graph need to be explored in the optimal solution. If a vertex v_j is visited before all vertices v_i with $\pi(i) < \pi(j)$ are visited the algorithm must make a detour to visit the vertices v_i. The algorithm needs advice to decide which vertex is visited next because from a vertex v_i all vertices v_j with $\pi(i) < \pi(j)$ are reachable in one step. The algorithm does not learn anything from the graph structure because the unvisited vertices are indistinguishable and for every possible decision there exists an instance in which it is optimal.

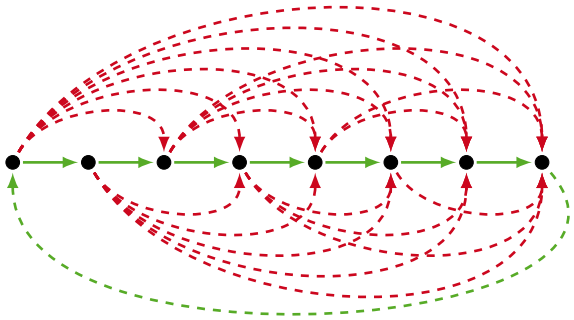

Figure 3.13: An example for the construction in Theorem 14 with $n = 8$. The green edges indicate the optimal solution and the red edges are the misleading edges. The green dashed edge is only part of the optimal solution for the cyclic graph exploration problem.

Theorem 14. *Every online algorithm with advice needs at least $\Omega(n \log(n))$ bits of advice to solve the cyclic or non-cyclic graph exploration problem on a general directed graph with n vertices.*

Proof. For every possible permutation $\pi\colon \{2,\dots,n\} \to \{2,\dots,n\}$ of the vertex set without the starting vertex v_1 and for $n = 2^k$, we construct an

instance. A vertex v_i has an edge to v_j if $\pi(i) < \pi(j)$ holds. The starting vertex has an edge to all other vertices. The last vertex in the ordering has only one edge leading back to the starting vertex. Thus, a vertex v_i has $n - \pi(i)$ outgoing edges, except for the last vertex, which has one outgoing edge.

Figure 3.13 shows an example for $n = 8$. The vertices are drawn from left to right following their ordering in π. The red edges are misleading edges that need to be avoided and the green edges are part of the optimal solution. The dashed green edge is only part of the optimal solution, for the cyclic graph exploration problem.

When the algorithm starts with its computation, it needs to know which of the $n-1$ outgoing edges is part of the optimal solution. For every possible decision, there exists an instance in which the choice is optimal. This pattern repeats on every vertex except for the last two vertices in the ordering. However the number of edges from which the algorithm has to choose one decreases. Thus, the algorithm needs at least

$$\sum_{i=1}^{n-2} \lfloor \log(n-i) \rfloor = \sum_{i=2}^{2^k-1} \lfloor \log(i) \rfloor = \sum_{i=1}^{\lfloor \log(2^k-1) \rfloor} i \cdot 2^i \geq \frac{n}{2} \log\left(\frac{n}{2}\right) \in \Omega(n \log(n))$$

bits of advice. Also enumerating all instances and encoding the given one, would need at least $\log((n-1)!) \in \Omega(n \log(n))$ bits of advice. □

3.6 Undirected and Sparse Graphs

In the following we focus on undirected graphs. With the possibility to traverse any edge in both directions the set of edge-optimal solutions increases because traversing any cyclic subsequence of an optimal exploration sequence in the opposite direction also yields in an optimal exploration sequence. Intuitively, the increased size of the solution set should make the problem easier. However, the lower bound of $\Omega(n \log(n))$ by Dobrev et al. [20] shows that the graph exploration problem on weighted undirected graphs is as hard as it is on directed graphs.

3.6.1 Structural Properties on Undirected Graphs

We already know from Lemma 13 that if an edge (u, v) is used multiple times in S^* there cannot be an edge $(v, u) \in S^*$. For undirected graphs we show even stronger restrictions on the edges' numbers of traversals.

Lemma 28. *No edge in an optimal exploration sequence $S^* \in \mathcal{S}$ for an undirected graph $G = (V, E)$ is used twice in the same direction.*

Proof. We prove the statement by contradiction. Assume there is an edge $e = (u, v)$ that is traversed two times in the same direction. Without loss

of generality we assume that the agent is moved twice from u to v. Then there must exist a subsequence of the form $(u, v, \ldots, u, v)$. We create an exploration sequence S that visits the same vertices but does not use the edge e at all, by removing the two traversals along e and reversing the path from v to u. This exploration sequence contradicts the optimality of S^*. □

Thus, on undirected graphs the edges from E_{multi} are used only once in each direction.

Lemma 29. *Every optimal exploration sequence $S^* \in \mathcal{S}$ for an undirected graph $G = (V, E, cost)$ contains at most $2n - 2$ vertices.*

Proof. An undirected graph $G = (V, E, cost)$ can be transformed into a directed one $G' = (V, E', cost')$ by replacing every undirected edge with two oppositely directed edges which have the same cost as the undirected edge. The optimal exploration sequence S^* is still optimal for G'. The graph G' has twice as many edges and Theorem 6 still holds. Due to Lemma 28 the edges in G' are traversed at most once.

Thus, every vertex is visited as often as it has incoming edges from S^* and from Theorem 6 we know that there are at most $2n - 2$ edges in S^*. Therefore, there are at most $2n - 2$ vertices in S^*. □

3.6.2 Algorithm for Undirected Graphs with Bounded Degree

If the degree is bounded by 2, the number of different graph classes that are connected is strongly limited. The only possible input is a cycle or a path. For the cyclic graph exploration, a path is trivial to solve, because every edge is traversed twice in the unique optimal solution. However, a cycle is already difficult to solve in that even a constant number of advice bits does not suffice, no matter if we want to solve the cyclic or the non-cyclic version of the problem.

The main idea to prove this is that during the exploration of the cycle each newly revealed edge is more expensive than the sum of all previous ones. This increasing sequence ends at some edge u, v which is then not part of the unique optimal exploration sequence because the path from u to v not using the edge is cheaper than using the edge. However, the algorithm needs advice to identify the edge where it is necessary to turn around and traverse the cycle in the other direction.

Lemma 30. *Every online algorithm with advice requires at least $\log(n-2)$ bits of advice to solve the cyclic or the non-cyclic graph exploration on a given unknown undirected graph $G = (V, E, cost)$ with degree bounded by 2.*

Proof. For every fixed n, we create a family of cycles that have length n and differ only in the weights for the edges. The edges are denoted by e_1 to e_n counterclockwise beginning at the starting vertex, as shown in Figure 3.14.

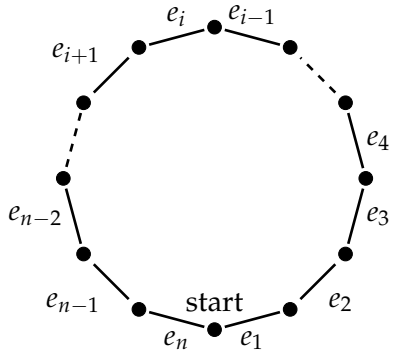

Figure 3.14: A schematic drawing of the construction from the proof of Lemma 30. The dashed edges represent a sequence of edges.

The weights of the edges are set such that in the unique optimal solution the explorer traverses the cycle until it reaches an edge e_i, which is not part of the solution, with $1 < i < n$. On this position it turns around and traverses the cycle in the opposite direction until it reaches the other vertex incident to e_i.

The weights of the edges e_1 to e_{i-1} are increasing such that each edge e_j with $1 < j < i$ has a weight that is larger than the sum of all previous edge weights. Analogously, the weights of the edges from e_{i+1} to e_n are decreasing such that each edge e_j with $i < j < n$ has a weight that is larger than the sum of the weights of all edges with a larger index.

The weight of the edge e_i is set to 10^x, where $x \gg n$ is not known to the algorithm. The sum of all other weights must be smaller than this value. However, the increasing (decreasing, respectively) weights cannot follow a fixed structure because an online algorithm could identify the edge e_i because it is the edge where the fixed structure of weights stops. Thus, for every fixed i, we add every instance to our family of cycles where the edge weights fulfill

$$\sum_{\substack{j=1 \\ j \neq i}}^{n} w(e_j) < w(e_i),$$

$$\sum_{j=1}^{k} w(e_j) < w(e_{k+1}) \text{ , for } 1 < k < i,$$

$$\sum_{j=k}^{n} w(e_j) < w(e_{k+1}) \text{ , for } i < k < n, \text{ and}$$

$$w(e_1) = w(e_n) = 1 \,.$$

Therefore, the online algorithm is unable to deduce the index i from the

weights of the edges $e_1, \ldots, e_i$ or $e_n, \ldots, e_i$. For each index i, there is one unique optimal solution. Note that if the first edges that the algorithm sees have different weights, it could traverse along the edge of lower weight because it knows this edge cannot be e_i, as it is the most expensive one. The algorithm can continue along this path until it reaches an edge that is more expensive than the first edges without using advice. Thus, there are $n-2$ different possible positions for the edge e_i. If an algorithm reads less than $\log(n-2)$ bits of advice, it has to use the same advice string for two instances that differ in their optimal solution. Thus, the algorithm either traverses the edge e_i or does not traverse the edge adjacent to e_i, for one of the instances. In both situations the algorithm is not able do compute the unique optimal solution. Consequently, there is no online algorithm with advice that can solve the graph exploration problem on undirected graphs with bounded degree by 2 with less than $\log(n-2)$ bits of advice. □

For the non-cyclic graph exploration we need one bit of advice to ensure that we solve the problem on a path. If the starting vertex is not an endpoint of the path, we split the edges in two sets A and B depending on whether they are to the right or left of the starting vertex. The optimal exploration sequence starts to explore the set with the lower total cost, because the side that is explored first is traversed twice. An online algorithm without advice cannot compute the cost of the two sets without exploring them first. However with one bit of advice we can communicate the optimal decision.

We also need to know in which direction the exploration has to start when we solve the non-cyclic graph exploration on a cycle, because all optimal solutions have one edge that is not traversed. Thus, the graph G_{S^*} (which is induced by the edges that are used in the optimal solution) is a path. As above, we split the edges in two sets A and B depending on whether they are to the right or left of the starting vertex in the graph G_{S^*}. An online algorithm has to know which of the sets has lower total cost to compute the optimal solution.

Lemma 31. *There exists an online algorithm which solves the non-cyclic graph exploration problem using* $1 + \log(n) + 2\log(\log(n))$ *bit advice on a given unknown undirected graph* $G = (V, E, cost)$ *with degree bounded by* 2.

Proof. Our algorithm reads the complete advice string at the beginning. The first bit defines the direction in which we start with our exploration sequence. With a self-delimiting encoding, the $\log(n) + 2\log(\log(n))$ bits of advice communicate a number i, with $1 < i < n$, which is the index of the edge that is not part of the optimal solution. Note that if the input is a path, the oracle encodes n on the advice tape. Thus, the algorithm traverses to an endpoint of the path, recognizes the input as a path and computes the optimal solution by traversing to the other endpoint.

On a cycle, our algorithm starts to traverse the graph in the starting direction which is defined by the oracle and stops when it reaches the edge e_i. Then it traverses the already used edges again and finally all remaining edges except e_i. □

When the degree is bounded by a constant k, the optimal exploration sequence becomes more complicated because there are edges in the graph that are not part of it. Thus, we need the algorithm to avoid choosing an edge that is not part of S^*.

Theorem 15. *There exists an online algorithm which solves the cyclic or the non-cyclic graph exploration problem using* $(2n-4)\log(k)$ *bits of advice on a given unknown undirected graph* $G = (V, E, cost)$ *with degree bounded by* k*.*

Proof. We can read $\log(k)$ bits of advice for every vertex (except the last two, (see Lemma 17) in the optimal exploration sequence to know which of the incident edges the algorithm has to choose. From Lemma 29 we know that there are at most $2n-2$ vertices in S^*. Thus, we need $(2n-4)\log(k)$ bits of advice to compute S^* in total. □

Note that this result improves the trivial upper bound of $n\log(n)$ for the case that $\log(k^2) \leq \log(n)$ holds.

3.6.3 Lower Bound for Undirected and Sparse Graphs

In order to prove a lower bound for the advice complexity, we use the same idea as presented in [20] and make only small changes so that the construction works for graphs with degree bounded by k. First we give a lower bound for the advice complexity for the cyclic graph exploration problem. Afterwards we make small adaptions to obtain a bound for the non-cyclic problem.

The idea for both lower bounds is the same. We construct cliques of size k and connect them in a cyclic way. Each clique must be traversed in a specific order due to some permutation chosen by the adversary. We make sure that it is non-optimal to traverse the vertices in a different order with the cost function for the edges.

Theorem 16. *Every online algorithm with advice needs at least* $\frac{n}{4}(\log(k+1)-1)$ *bits of advice to solve the cyclic graph exploration problem on a given unknown undirected graph with degree bounded by* k*.*

Proof. Our lower bound construction consists of cliques that need to be traversed in a specific order based on a permutation chosen by the adversary. At first we focus on the construction of a single clique and how it is traversed. Afterwards we explain how to enlarge the construction with multiple cliques.

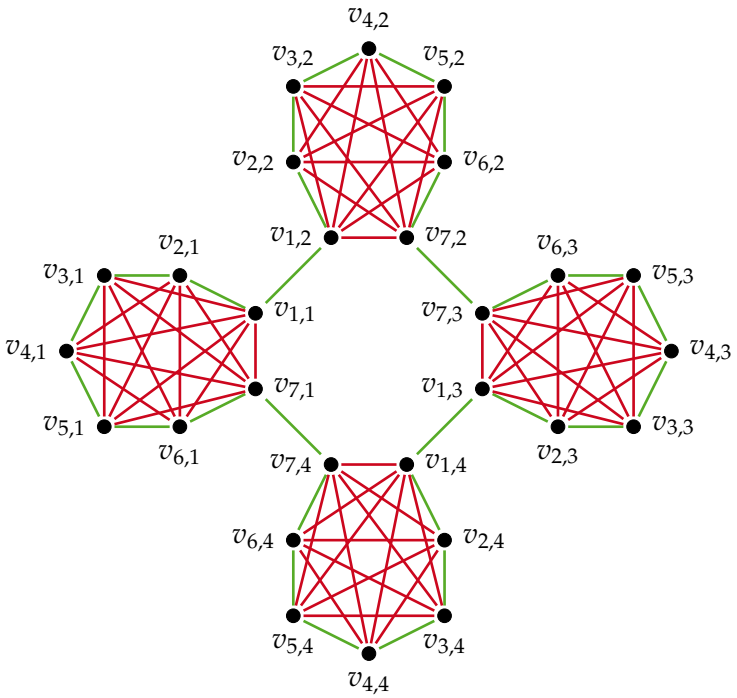

Figure 3.15: An example of the construction explained in Theorem 16, for clique size $k = 7$ and the number of vertices $n = 4k$. The green edges are part of the optimal exploration sequence and the red edges are misleading edges.

Each clique j contains k vertices $v_{i,j}$ with $1 \leq i \leq k$. The edge weight for an edge $(v_{a,j}, v_{b,j})$ is $\max(a, b)$. The algorithm starts at the vertex $v_{k,j}$ and sees $k-1$ edges with cost k. The optimal exploration sequence visits the vertices in descending order paying in total $\frac{k(k+1)}{2} - 1$. If the algorithm traverses the vertices in a different order there is a subsequence of the form $(v_{a,j}, v_{b,j}, v_{c,j})$ with $b < a$ and $b < c$. Thus, comparing the total cost with the cost for the optimal exploration sequence, the term b that is contained in the cost of the optimal exploration sequence is replaced with a or c, which are both larger than b.

To allow a construction of arbitrary size we connect an even number of cliques in a cyclic way. We add the edges $(v_{1,j}, v_{1,j+1})$ and $(v_{k,j-1}, v_{k,j})$ for j being odd. The vertex $v_{k,1}$ has an edge to the vertex $v_{k,2r}$ where $2r$ is the number of cliques. Each edge connecting two cliques has a cost of 1. Figure 3.15 shows an example for $2r = 4$ cliques, each of size $k = 7$. Note that, the optimal exploration sequence can be traversed in both directions on a undirected graph. Thus, the algorithm can decide to traverse the graph in a clockwise or counterclockwise direction. We need to avoid that one direc-

tion results in a trivial solution. With trivial, we mean that the vertices are traversed in an ascending order, regarding their permutation. The permutation is revealed by the edge costs if the clique is traversed in the reverse direction. Therefore, only every second clique is difficult to solve for the online algorithm. The other half of the cliques only becomes difficult if the algorithm traverses the graph in the opposite direction.

In total our construct has $n = 2kr$ vertices, each is denoted as $v_{i,j}$ with $1 \leq i \leq k$ and $1 \leq j \leq 2r$. Any online algorithm has to use advice for one half of the cliques to compute the ascending order of the vertices. Thus, it has to read for r cliques $\sum_{i=1}^{k-1} \log(i)$ bits of advice. To simplify our calculations we choose $k = 2^y - 1$ such that $\lfloor \log(k) \rfloor = \log(\frac{k+1}{2})$ holds. We can lower bound the total number of advice bits that is required to explore our construction with

$$\frac{n}{2k} \sum_{i=1}^{k-1} \log(i) \geq \frac{n}{2k} \sum_{i=1}^{k-1} \lfloor \log(i) \rfloor = \frac{n}{2k} \sum_{i=1}^{\lfloor \log(k) \rfloor} 2^i \cdot i \geq \frac{n}{4} \log\left(\frac{k+1}{2}\right) .$$

□

For the non-cyclic version we can remove one edge in our construction making it simpler because the optimal exploration sequence becomes unique.

Theorem 17. *Every online algorithm with advice needs at least* $\frac{n}{2}(\log(k+1) - 1)$ *bits of advice to solve the non-cyclic graph exploration problem on a given unknown undirected graph with degree bounded by k.*

Proof. We use the same construction as described in Theorem 16 but remove the edge $(v_{k,1}, v_{k,2r})$ that connects the first clique with the last one ($2r$ is the number of cliques). So, we remove the possibility to traverse the cliques in a counterclockwise direction. Therefore, we can always force the algorithm to traverse the vertices in a descending order for all cliques and not only for half of them. This doubles the number of required advice bits. □

3.7 Exploring with a Map

In our previous models where the online algorithm has no knowledge about the graph, all its decisions depend on the advice given by the oracle. Only with information gathered during exploration and together with the advice it was possible to make decisions such as determining if an edge is exhausted. Analyzing the advice complexity in such a model shows the amount of crucial information that is sufficient to explore a graph when all knowledge is missing. Obviously the same advice also allows an algorithm

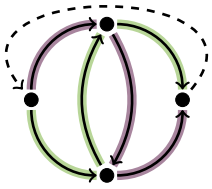

Figure 3.16: An example to show that the missing information about the edge costs prohibits any online algorithm to compute the optimal solution without additional advice. Both marked paths, purple and green, can be optimal but any online algorithm cannot decide which by knowing only the cost of the first edge.

to explore a graph if it has some a priori knowledge about the graph. However, in the following we show that even less advice is necessary to explore a graph when the structure of the graph is known a priori to the online algorithm. To be precise, the algorithm knows all vertices and edges, their labels but not the edge weights. We denote this by saying the algorithm has a map of the graph.

The amount of knowledge about the structure makes a huge difference. In the extreme case where the complete input is known beforehand, the problem degenerates to an offline problem. As online algorithms are not limited in their computational power this is the least interesting case. But if the knowledge about the graph is only partial or unsure, we are able to give more finely granulated bounds on the advice complexity. In the following we distinguish two settings.

In the first setting, additionally to the map of the graph, the algorithm has knowledge about the number of traversals for every edge. It can identify its current location on the map at any time and recognizes the seen vertices and edges on the map. Instead of knowing the precise number of traversals for every edge, it only knows if an edge is used once, never or more than once in the optimal solution. The cost for traversing an edge stays hidden until the explorer is placed at a vertex incident to the edge.

We show that this a priori knowledge already suffices to compute the optimal solution and the number of traversals for every edge of the graph.

In the second setting, the online algorithm knows the vertices and edges in the given graph but lacks the information about the cost function. It learns the cost of and edge if it is placed at an incident vertex in the case of undirected graphs. For directed graphs, the cost of an edge is revealed if the explorer is placed at the vertex where the edge starts. Thus, the algorithm knows the cost of an edge before it is used for the first time. Figure 3.16 shows an example why it is not possible to compute the optimal solution

in this settings without advice.

Such a scenario can arise when an autonomous agent has to explore an area with an old or incomplete map. The paths on the map still exits but they may be less passable than they were when the map was created. However, before the agent decides to drive along the path it sees the condition of the road or if the mine shaft that should be explored is blocked. The weight of an edge can be adjusted to represent different conditions. If the path is nearly impassable, the cost can be set such that it is never part of an optimal solution. But if it takes just a little bit more time or fuel to take the path, the cost can be chosen to be slightly higher than one. Thus, the a priori information given to the algorithm is not completely reliable, but it shows where structural dead ends or bottlenecks are.

We show that there exists an online algorithm that can explore any directed or undirected graph with $m \log(3)$ bits of advice, where m is the number of edges, if it has a map showing all edges and vertices.

3.7.1 Related Work

The idea of providing an online algorithm with knowledge about the graph is not new. In the Canadian Travelers problem, introduced in [50], the task is to move an agent from the starting vertex to the target vertex along the shortest tour, knowing the graph, its edges and their costs, but edges can fail which makes them unavailable. The algorithm only finds out that the chosen path is blocked, i.e., an edge fails, when the agent is placed on an incident edge. The authors also introduce the Double-Valued Graph problem where some edges have two possible costs from which an adversary has to choose one as soon as the agent, searching for a shortest path, is placed on an incident edge. Again, the algorithm has full knowledge about the graph, its edges and the possible costs. The authors provide a complexity result for each of the two problems by presenting a reduction from the quantified Boolean formula problem (QBF) showing that an online algorithm can only compute a solution with a certain competitive ratio if the QBF is satisfiable. Thus, an online algorithm needs to be able to solve a PSPACE-complete problem to compute a solution for the online problems that has the competitive ratio intended by the reduction. In [47] the authors assume that the edge costs are random variables and that they are always usable. Moreover, the probability distribution is known beforehand, but the value of an edge is only revealed when the agent is placed at an incident vertex. They investigate the connection between the Canadian travelers problem and Markov Decision Processes. If the edge cost is resampled when the edge is visited again or the input graph is a directed acyclic graph, the problem becomes polynomially solvable. For this setting, two strategies are presented and their competitiveness is analyzed in [60]. The simple greedy approach is $2^{k+1} - 1$ competitive, where k is an upper bound for the number of edges

that can fail. The second strategy involves backtracking to the start if a failed edge is revealed and it is $2k + 1$ competitive. The same upper bound was independently developed in [59]. Here the authors also present a lower bound of $k + 1$ for randomized online algorithms.

A very similar problem, called stochastic shortest route problem was introduced in [54]. Here the adversarial online worst case is not of interest, because the part of the input that is unknown is random. The graph is known a priori and the costs of the edges are unknown random variables. These graphs are called stochastic graphs. If the probability distribution is known beforehand a common approach is to use the expected value as cost, for every edge in the graph, and applying basic techniques to compute the shortest path. Later the problem became also known as the stochastic shortest path problem, but the model often differs in minor details. In [2] the costs are known beforehand and an edge can fail, similar to the CTP problem, but with an probabilistic model instead of an adversarial one. The algorithm knows that an edge fails only if an agent that traverses the graph is positioned on an incident vertex. The authors show that using stochastic programming methods the problem can be solved in polynomial time. In [51] different models regarding the time at which the cost of an edge is revealed are discussed. In their last model, the edge costs are learned progressively. An agent, deciding every time it is positioned at a vertex which edge should be traversed next, traverses the graph and learns the edge costs during its exploration. The authors provide policies from the currently available information of the algorithm. A more recent result [48] provides a $n^{\Theta(\log(n))}$ algorithm that maximizes the probability to find a path that does not exceed some given threshold in a stochastic graph where the edge costs follow a normal distribution. The authors also look at other distributions than the normal distribution and show that the problem is reduced to the deterministic shortest path problem if the distribution is additive or satisfies stochastic dominance.

In general, the approach to support an online algorithm with additional information about the input is well studied for basic online problems. These settings are called semi-online and [24] provides an overview of the different, so called, extra pieces of information (EPI) and their impact on the competitive ratio for the famous scheduling problem. The EPI are always a priori knowledge about the future input and can contain, e.g., the total size of all jobs or the optimum makespan. There are also results for semi-online graph problems. The authors of [41] analyze the semi-online Bipartite Matching problem in a graph $G(U, V_P \cup V_A, E_G)$, where U is one side of the bipartite set and the other side is split into V_P and V_A. The sets U and V_P and edges between these vertices are EPI, i.e., known beforehand. Only the vertices from V_A arrive in an online fashion. They introduce a parameter that represents the ratio of online arriving vertices to already known vertices and give lower and upper bounds depending on this ratio.

3.7.2 Algorithm with a Map and Advice

We show that the cost function makes a difference, even if the algorithm has a map of the graph, i.e, it knows all vertices and their edges. If all edges have the same weight (unit cost function) or the cost function is known beforehand, the algorithm is able to compute the total cost for every cyclic (or non-cyclic) exploration sequence without moving the agent through the graph and chooses the cheapest one as its solution.

Theorem 18. *There exists an online algorithm which solves the cyclic and non-cyclic graph exploration problem using no advice on known directed and known undirected graphs if the cost function is known beforehand.*

Proof. As the algorithm is not limited in its computing power, it can compute all possible exploration sequences by brute force. Obviously, this is possible for the directed and undirected case. It chooses the solution which results in the shortest exploration sequence. □

If the cost function is not known, the algorithm will not know which of the feasible exploration sequences will be the cheapest. The missing knowledge can be compensated if the algorithm has knowledge about the optimal solutions $S^* \in \mathcal{S}$. We show that we can compute the number of traversals for every edge when we know which edges are used never E_0, once E_1 and multiple times E_{multi}. Thus, knowing the number of traversals for every edge suffice to compute an optimal solution even if the cost function is unknown.

Lemma 32. *Let $G = (V, E)$ be a graph with an multi-edge optimal exploration sequence $S^* = (v_0, \ldots, v_{end})$. There is an algorithm $\mathcal{A}(V, E_0, E_1, E_{multi})$ which computes, for each edge $e \in E$, the number of visits by S^*.*

Proof. Note that the algorithm $\mathcal{A}$ a priori only knows the exact number of visits for an edge e if $\#_{S^*}(e) \leq 1$. Corollary 2 and Lemma 15 show that the graph $F = (V, E_{multi})$ is a forest, even if the edges are undirected. Furthermore, Lemma 13 shows that, if edges are used multiple times in one direction, they are not used at all in the reverse direction.

We now describe a recursive algorithm which computes the exact number of visits for the edges from E_{multi}. Every tree T in the forest F must have at least one leaf. A leaf v in F has exactly one edge that is used multiple times and the other adjacent edges in G are used exactly once or never. Let U be the set of those edges from E_{multi}, where the number of traversals is unknown. As long as $U \neq \emptyset$, the graph $F' = (V, U)$ has at least one leaf v. Let $e \in U$ be the only directed edge attached to v in F'. For all other edges $e' \in E(G)$ attached to v, we already know the number of visits, i.e., $e' \in E_0 \cup E_1 \cup (E_{multi} \setminus U)$. The algorithm $\mathcal{A}$ sums up the traversal numbers

of all other incoming and outgoing edges for v, i.e.,

$$\begin{aligned} c_{in} &= \sum_{(x,v)\in E\setminus e} \#_{S^*}(x,v), \\ c_{out} &= \sum_{(v,x)\in E\setminus e} \#_{S^*}(v,x). \end{aligned}$$

Thus, the usage of the edge e can be calculated, because $d_\Delta(v) = 0$ holds for $G_{S^*} = (V, E_{used})$. Therefore, the number of traversals is the difference between incoming and outgoing edges from E_1. Only two cases need to be considered. If $e = (w, v)$, then $c_{in} < c_{out}$ and $\mathcal{A}$ can compute its traversal number $\#_{S^*}(e) = c_{out} - c_{in}$. Analogously, for $e = (v, w)$, we have $\#_{S^*}(e) = c_{in} - c_{out}$.

The result can be used in the next level of the trees to calculate the number of traversals for multiply used edges analogously. If the number of traversals for all edges $e \in E_{multi}$ attached to a leaf of $F' = (V, U)$ are computed, we update $U' = U \setminus \{e\}$. Now, we have again new leaves in the forest $F'' = (V, U')$ for which exactly one adjacent edge is part of U'. For all other edges, the precise number of traversals is known because they are from E_0, E_1 or $E_{multi} \setminus U$.

Repeating this bottom-up approach for the forest F'' allows us to compute the exact usage of every edge from E_{multi}. □

Note that the algorithm requires knowledge about the graph to know the number of leaves in the forest of multi-edges.

Theorem 19. *There exists an online algorithm which solves the cyclic and the non-cyclic graph exploration problem using* $\log(3)m$ *bits of advice on known directed and undirected graphs.*

Proof. We use the advice bits to determine for every edge its membership in one of the three sets E_0, E_1, E_{multi}. Thus, only the numbers of traversals for the edges in E_{multi} remain unknown. Since the algorithm knows the vertices and edges of the graph, it can apply Lemma 32 to compute the exact usage for every edge. Due to the unlimited computing power, our algorithm can compute all possible exploration sequences such that no edge is used more often than in S^* by brute force. Note that, for undirected graphs we do not need to apply Lemma 32 because from Lemma 28 we know that the edges from E_{multi} are each used twice. □

Chapter 4

Conclusion and Future Work

We have seen two very different online problems. The first one was from the field of graph drawing and we introduced a deterministic online algorithm that achieves a constant competitive ratio. We proved its performance by excluding certain substructures and showing that other critical substructures can only appear when the optimal solution also incurs high costs. From there on the remaining part of the proof was an exhaustive case distinction.

Our results showed that the best possible competitive ratio is between $\frac{4}{3}$ and 5. Decreasing this gap is one of the future tasks that we propose. Another unsolved problem is to extend our results to k-regular graphs with $k > 2$. The problem should become easier for increasing k, because the number of unavoidable mistakes also increases with k. As an extreme example, for $k = n$ any ordering of the vertices is optimal.

The second technical part considered the advice complexity of the graph exploration problem on directed and undirected graphs. Due to previous results we focused on degree-bounded and sparse graphs. We derived upper and lower bounds for the cyclic and the non-cyclic version of the problem. Afterwards, we also discussed a variation where the online algorithm has a priori knowledge about the input and its influence on the advice complexity.

Although the presented bounds are asymptotically matching, there is still some room for improving the lower and upper bounds. A large part of the information provided by the oracle was used to encode traversal numbers for edges used often. However, in the lower bound construction all the edges were used only once, twice or never. Thus, it seems like that the information how often an edge is traversed can be communicated with less advice or it is possible to create a more complex lower bound construction where the number of traversals is from a greater range.

Moreover, the last topic concerning additional a priori knowledge needs further investigation. The previous observations regarding the number of

traversals of the edges in an undirected graph should make it possible to reduce the advice that is needed to compute an optimal solution when the graph is known beforehand.

Bibliography

[1] S. Albers and M. R. Henzinger. Exploring unknown environments. *SIAM J. Comput.*, 29(4):1164–1188, 2000.

[2] G. Andreatta and L. Romeo. Stochastic shortest paths with recourse. *Networks*, 18(3):193–204, 1988.

[3] A. M. Andrew. Artificial intelligence and mobile robots: Case studies of successful robot systems. *Robotica*, 17(2):229–235, 1999.

[4] L. Barrière, X. Muñoz, J. Fuchs, and W. Unger. Online matching in regular bipartite graphs. *Parallel Process. Lett.*, 28(2):1850008:1–1850008:13, 2018.

[5] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom.*, 4:235–282, 1994.

[6] M. A. Bender, A. Fernández, D. Ron, A. Sahai, and S. P. Vadhan. The power of a pebble: Exploring and mapping directed graphs. *Inf. Comput.*, 176(1):1–21, 2002.

[7] P. Berman. On-line searching and navigation. In A. Fiat and G. J. Woeginger, editors, *Proceedings of Online Algorithms, The State of the Art (the book grew out of a Dagstuhl Seminar, June 1996)*, volume 1442 of *Lecture Notes in Computer Science*, pages 232–241. Springer, 1996.

[8] A. Birx, Y. Disser, A. V. Hopp, and C. Karousatou. An improved lower bound for competitive graph exploration. *Theor. Comput. Sci.*, 868:65–86, 2021.

[9] A. Blum, P. Raghavan, and B. Schieber. Navigating in unfamiliar geometric terrain. *SIAM J. Comput.*, 26(1):110–137, 1997.

[10] H. Böckenhauer, J. Fuchs, and W. Unger. The graph exploration problem with advice. *CoRR*, abs/1804.06675, 2018.

[11] H.-J. Böckenhauer, J. Fuchs, and W. Unger. Exploring sparse graphs with advice (extended abstract). In L. Epstein and T. Erlebach, editors, *In Proceedings of the 16th International Workshop on Approximation and Online Algorithms, WAOA 2018, Helsinki, Finland, August 23-24, 2018*, volume 11312 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2018.

[12] H.-J. Böckenhauer, D. Komm, R. Královič, R. Královič, and T. Mömke. On the advice complexity of online problems. In Y. Dong, D. Du, and O. H. Ibarra, editors, *In Proceedings of the 20th International Symposium on Algorithms and Computation, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009*, volume 5878 of *Lecture Notes in Computer Science*, pages 331–340. Springer, 2009.

[13] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.

[14] J. Boyar, L. M. Favrholdt, C. Kudahl, K. S. Larsen, and J. W. Mikkelsen. Online algorithms with advice: A survey. *ACM Comput. Surv.*, 50(2):19:1–19:34, 2017.

[15] P. Brass, I. Vigan, and N. Xu. Improved analysis of a multirobot graph exploration strategy. In *In Proceedings of the 13th International Conference on Control Automation Robotics & Vision, ICARCV 2014, Singapore, December 10-12, 2014*, pages 1906–1910. IEEE, 2014.

[16] E. Burjons, J. Fuchs, and H. Lotze. The slotted online one-sided crossing minimization problem on 2-regular graphs, 2022.

[17] J. Chalopin, P. Flocchini, B. Mans, and N. Santoro. Network exploration by silent and oblivious robots. In D. M. Thilikos, editor, *In Proceedings of the 36th International Workshop on Graph Theoretic Concepts in Computer Science, WG 2010, Zarós, Crete, Greece, June 28-30, 2010 Revised Papers*, volume 6410 of *Lecture Notes in Computer Science*, pages 208–219, 2010.

[18] S. Das, P. Flocchini, S. Kutten, A. Nayak, and N. Santoro. Map construction of unknown graphs by multiple agents. *Theor. Comput. Sci.*, 385(1-3):34–48, 2007.

[19] K. Diks, P. Fraigniaud, E. Kranakis, and A. Pelc. Tree exploration with little memory. *J. Algorithms*, 51(1):38–63, 2004.

[20] S. Dobrev, R. Královič, and E. Markou. Online graph exploration with advice. In G. Even and M. M. Halldórsson, editors, *In Proceedings of the 19th International Colloquium on Structural Information and Communication Complexity, SIROCCO 2012, Reykjavik, Iceland, June 30-July 2,*

2012, Revised Selected Papers, volume 7355 of *Lecture Notes in Computer Science*, pages 267–278. Springer, 2012.

[21] S. Dobrev, R. Královič, and D. Pardubská. How much information about the future is needed? In V. Geffert, J. Karhumäki, A. Bertoni, B. Preneel, P. Návrat, and M. Bieliková, editors, *In Proceedings of the 34th Conference on Current Trends in Theory and Practice of Computer Science,SOFSEM 2008, Nový Smokovec, Slovakia, January 19-25, 2008*, volume 4910 of *Lecture Notes in Computer Science*, pages 247–258. Springer, 2008.

[22] J. Dreier, J. Fuchs, T. A. Hartmann, P. Kuinke, P. Rossmanith, B. Tauer, and H. Wang. The complexity of packing edge-disjoint paths. In B. M. P. Jansen and J. A. Telle, editors, *14th International Symposium on Parameterized and Exact Computation, IPEC 2019, September 11-13, 2019, Munich, Germany*, volume 148 of *LIPIcs*, pages 10:1–10:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[23] V. Dujmović and S. Whitesides. An efficient fixed parameter tractable algorithm for 1-sided crossing minimization. *Algorithmica*, 40(1):15–31, 2004.

[24] D. Dwibedy and R. Mohanty. Semi-online scheduling: A survey. *Comput. Oper. Res.*, 139:105646, 2022.

[25] P. Eades and N. C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994.

[26] P. Elias. Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theory*, 21(2):194–203, 1975.

[27] Y. Emek, P. Fraigniaud, A. Korman, and A. Rosén. Online computation with advice. *Theor. Comput. Sci.*, 412(24):2642–2656, 2011.

[28] R. Fleischer and G. Trippen. Exploring an unknown graph efficiently. In G. S. Brodal and S. Leonardi, editors, *In Proceedings of the 13th Annual European Symposium on Algorithms, ESA 2005, Palma de Mallorca, Spain, October 3-6, 2005*, volume 3669 of *Lecture Notes in Computer Science*, pages 11–22. Springer, 2005.

[29] K. Foerster and R. Wattenhofer. Lower and upper competitive bounds for online directed graph exploration. *Theor. Comput. Sci.*, 655:15–29, 2016.

[30] P. Fraigniaud and D. Ilcinkas. Digraphs exploration with little memory. In V. Diekert and M. Habib, editors, *In Proceedings of the 21st Annual Symposium on Theoretical Aspects of Computer Science, STACS 2004,*

Montpellier, France, March 25-27, 2004, volume 2996 of *Lecture Notes in Computer Science*, pages 246–257. Springer, 2004.

[31] P. Fraigniaud, D. Ilcinkas, and A. Pelc. Tree exploration with advice. *Information and Computation*, 206(11):1276–1287, 2008.

[32] Y. Frishman and A. Tal. Online dynamic graph drawing. *IEEE Trans. Vis. Comput. Graph.*, 14(4):727–740, 2008.

[33] M. R. Garey and D. S. Johnson. Crossing number is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 4(3):312–316, 1983.

[34] B. Gorain and A. Pelc. Deterministic graph exploration with advice. In I. Chatzigiannakis, P. Indyk, F. Kuhn, and A. Muscholl, editors, *In Proceedings of the 44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 132:1–132:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[35] J. Hromkovič, R. Královič, and R. Královič. Information complexity of online problems. In P. Hlinený and A. Kucera, editors, *In Proceedings of the 35th International Symposium on Mathematical Foundations of Computer Science 2010, MFCS 2010, Brno, Czech Republic, August 23-27, 2010.*, volume 6281 of *Lecture Notes in Computer Science*, pages 24–36. Springer, 2010.

[36] B. Kalyanasundaram and K. Pruhs. Constructing competitive tours from local information. In A. Lingas, R. G. Karlsson, and S. Carlsson, editors, *In Proceedings of the 20nd International Colloquium on Automata, Languages and Programming, ICALP 1993, Lund, Sweden, July 5-9, 1993*, volume 700 of *Lecture Notes in Computer Science*, pages 102–113. Springer, 1993.

[37] Y. Kobayashi and H. Tamaki. A fast and simple subexponential fixed parameter algorithm for one-sided crossing minimization. *Algorithmica*, 72(3):778–790, 2015.

[38] D. Komm. *An Introduction to Online Computation - Determinism, Randomization, Advice*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.

[39] D. Komm, R. Královič, R. Královič, and J. Smula. Treasure hunt with advice. In C. Scheideler, editor, *In Proceedings of the 22nd International Colloquium on Structural Information and Communication Complexity, SIROCCO 2015, Montserrat, Spain, July 14-16, 2015*, volume 9439 of *Lecture Notes in Computer Science*, pages 328–341. Springer, 2015.

[40] R. Královič, 2017. Personal communication.

[41] R. Kumar, M. Purohit, A. Schild, Z. Svitkina, and E. Vee. Semi-online bipartite matching. In A. Blum, editor, *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA*, volume 124 of *LIPIcs*, pages 50:1–50:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[42] X. Y. Li and M. F. M. Stallmann. New bounds on the barycenter heuristic for bipartite graph drawing. *Inf. Process. Lett.*, 82(6):293–298, 2002.

[43] N. Megow, K. Mehlhorn, and P. Schweitzer. Online graph exploration: New results on old and new algorithms. *Theor. Comput. Sci.*, 463:62–72, 2012.

[44] X. Muñoz, W. Unger, and I. Vrto. One sided crossing minimization is np-hard for sparse graphs. In *Graph Drawing, 9th International Symposium, GD 2001 Vienna, Austria, September 23-26, 2001, Revised Papers*, pages 115–123, 2001.

[45] H. Nagamochi. An improved bound on the one-sided minimum crossing number in two-layered drawings. *Discret. Comput. Geom.*, 33(4):569–591, 2005.

[46] H. Nagamochi. On the one-sided crossing minimization in a bipartite graph with large degrees. *Theor. Comput. Sci.*, 332(1-3):417–446, 2005.

[47] E. Nikolova and D. R. Karger. Route planning under uncertainty: The canadian traveller problem. In D. Fox and C. P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 969–974. AAAI Press, 2008.

[48] E. Nikolova, J. A. Kelner, M. Brand, and M. Mitzenmacher. Stochastic shortest paths via quasi-convex maximization. In Y. Azar and T. Erlebach, editors, *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*, volume 4168 of *Lecture Notes in Computer Science*, pages 552–563. Springer, 2006.

[49] S. C. North and G. Woodhull. Online hierarchical graph drawing. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing, 9th International Symposium, GD 2001 Vienna, Austria, September 23-26, 2001, Revised Papers*, volume 2265 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 2001.

[50] C. H. Papadimitriou and M. Yannakakis. Shortest paths without a map. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, editors, *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings*, volume 372 of *Lecture Notes in Computer Science*, pages 610–620. Springer, 1989.

[51] G. H. Polychronopoulos and J. N. Tsitsiklis. Stochastic shortest path problems with recourse. *Networks*, 27(2):133–143, 1996.

[52] M. Schaefer. The graph crossing number and its variants: A survey. *The electronic journal of combinatorics*, 2012.

[53] R. Shannon and A. J. Quigley. Considerations in dynamic graph drawing: A survey. `http://rossshannon.com/publications/softcopies/rs2007-dynamic-graphs-survey.pdf`, 2007. Accessed: 2022-09-07.

[54] C. E. Sigal, A. A. B. Pritsker, and J. J. Solberg. The stochastic shortest route problem. *Oper. Res.*, 28(5):1122–1129, 1980.

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

[56] B. Tauer, D. Fischer, J. Fuchs, L. V. Koch, and S. Zieger. Waiting for trains: Complexity results. In M. Changat and S. Das, editors, *Algorithms and Discrete Applied Mathematics - 6th International Conference, CALDAM 2020, Hyderabad, India, February 13-15, 2020, Proceedings*, volume 12016 of *Lecture Notes in Computer Science*, pages 282–303. Springer, 2020.

[57] S. Thrun. Robotic mapping: A survey. In *Exploring artificial intelligence in the new millennium*. 2003.

[58] S. Thrun, S. Thayer, W. Whittaker, C. R. Baker, W. Burgard, D. Ferguson, D. Hähnel, M. D. Montemerlo, A. Morris, Z. Omohundro, and C. F. Reverte. Autonomous exploration and mapping of abandoned mines. *IEEE Robotics Autom. Mag.*, 11(4):79–91, 2004.

[59] S. Westphal. A note on the k-canadian traveller problem. *Inf. Process. Lett.*, 106(3):87–89, 2008.

[60] Y. Xu, M. Hu, B. Su, B. Zhu, and Z. Zhu. The canadian traveller problem and its competitive analysis. *J. Comb. Optim.*, 18(2):195–205, 2009.