

# Exploring the Edge of the Online World: Hashing, Zero Memory, Reservation

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES  
(Dr. sc. ETH Zurich)

presented by

DAVID WEHNER

Master of Mathematics, ETH Zurich  
born on March 7, 1987  
citizen of Switzerland

accepted on the recommendation of

Prof. Dr. Juraj Hromkovič  
Prof. Dr. Martin Dietzfelbinger  
Prof. Dr. Rastislav Kráľovič



# Abstract

This thesis studies three topics in online computing. In the first topic, we regard a problem as being online that is usually only considered to be offline. In the second topic, we tighten the online restriction of a problem close to being an online problem. In the third topic, we consider a model that connects online and offline problems. Thus, figuratively speaking, we explore the edge of the online world.

First, we regard hashing as an online problem. In hashing, we have a large universe  $U$ , from which we receive as input an unknown subset  $S$  and the goal is to map  $S$  to our hash table  $T$ , whose  $m$  elements are called *cells* such that  $S$  is well dispersed in  $T$ . We study *c-ideality*: a hash function family  $\mathcal{H}$  is  $c$ -ideal if, for every set  $S \subseteq U$  of size  $n$ , there is a hash function  $h: U \rightarrow T$  in  $\mathcal{H}$  such that there is no cell where more than  $c \cdot n/m$  elements of  $S$  are mapped to. This notion creates a link to competitive analysis, the classical performance analysis of online algorithms. By studying how many functions are necessary to constitute a  $c$ -ideal hash function family, we are able to measure the *advice complexity* of hashing. Advice complexity is a tool that was invented over a decade ago to measure the information complexity of online problems.

Second, we study a very restrictive graph exploration problem. In our model, an agent without persistent memory is placed on a vertex of a graph and only sees the adjacent vertices. The goal is to visit every vertex of the graph, return to the start vertex, and terminate. The agent does not know through which edge it entered a vertex. The agent may color the current vertex and can see the colors of the neighboring vertices in an arbitrary order. The agent may not recolor a vertex. We investigate the number of colors necessary and sufficient to explore all graphs. We prove that  $n - 1$  colors are necessary and sufficient for exploration in general, 3 colors are necessary and sufficient if only trees are to be explored, and  $\min\{2k - 3, n - 1\}$  colors are necessary and  $\min\{2k - 1, n - 1\}$  colors are sufficient on graphs of size  $n$  and circumference  $k$ , where the circumference is the length of a longest cycle. This only holds if an algorithm has to explore all graphs and not merely certain graph classes. We give an example for a graph class where each graph can be explored with 4 colors, although the graphs have maximal circumference. Moreover, we prove that recoloring vertices is very powerful by designing an algorithm with recoloring that uses only 7 colors and explores all graphs.

Third, we analyze the simple online knapsack problem with reservation. Here, input items of sizes  $s_1, s_2, \dots$  arrive and, for each item, the algorithm either has to pack, discard, or reserve the item. The goal is to maximize the total size of packed items, but the packing size may never exceed 1. When the input ends, the algorithm may still use reserved items to achieve a better packing of higher total size. The algorithm may reserve arbitrarily many items, but pays  $\alpha$  times the maximal total size of reserved items at the end, where  $\alpha \in [0, 1]$  is a parameter of the problem, the *reservation factor*. This model bridges the online and the offline world: For  $\alpha = 0$ , the problem is essentially the offline knapsack problem; for  $\alpha = 1$ , it is the normal online knapsack problem. For large and medium-sized reservation factors, we provide tight upper and lower bounds. For small reservation factors, we prove close upper and lower bounds that tend to 1 for  $\alpha$  tending to zero.



# Zusammenfassung

Die vorliegende Arbeit untersucht drei Themen im Bereich von Online-Berechnungen. Beim ersten Thema betrachten wir ein Problem, das üblicherweise als Offline-Problem gesehen wird, als ein Online-Problem. Anschliessend verschärfen wir beim zweiten Thema die Online-Bedingungen eines Problems, das nahe mit Online-Problemen verwandt ist. Schliesslich untersuchen wir beim dritten Thema ein Modell, das einen nahtlosen Übergang von Online- zu Offline-Problemen gewährt. In diesem Sinne erkunden wir gewissermassen den Rand der Online-Welt.

Als Erstes betrachten wir Hashing als ein Online-Problem: Wir haben ein grosses Universum  $U$  und erhalten davon eine unbekannte Teilmenge  $S$  als Eingabe. Ziel ist, dieses  $S$  auf unsere Hash-Tabelle  $T$ , deren Elemente *Zellen* genannt werden, so abzubilden, dass die Elemente von  $S$  gleichmässig in  $T$  verteilt sind. Wir betrachten dabei *c-Idealität*: eine Familie  $\mathcal{H}$  von Hash-Funktionen nennen wir *c-ideal*, wenn es für jede Menge  $S \subset U$  der Grösse  $n$  eine Hashfunktion  $h: U \rightarrow T$  aus  $\mathcal{H}$  gibt, so dass keiner Zelle mehr als  $c \cdot n/m$  Elemente aus  $S$  zugewiesen werden. Dieser Begriff schafft eine Verknüpfung zu der klassischen Messung der Güte von Online-Algorithmen, der *Competitive Analysis*. Wir untersuchen, wie viele Funktionen eine Familie von Hash-Funktionen mindestens enthalten muss, damit sie *c-ideal* sein kann. Das hilft uns anschliessend, die *Advice-Komplexität* von Hashing zu studieren. Advice-Komplexität ist ein vor mehr als zehn Jahren eingeführtes Konzept, mit dem sich der Informationsgehalt von Online-Problemen messen lässt.

Als Zweites studieren wir ein sehr restriktives Grapherkundungsproblem. In unserem Modell wird ein Agent ohne Erinnerungsvermögen auf einen Knoten eines Graphen gesetzt und sieht dabei nur die angrenzenden Knoten. Das Ziel ist, alle Knoten im Graphen zu besuchen, zum Startknoten zurückzukehren und anschliessend zu terminieren. Der Agent weiss nicht, von welcher Kante er zu einem Knoten gekommen ist. Er darf allerdings den jeweiligen Knoten färben und sieht die Farben der angrenzenden Knoten, allerdings in beliebiger Reihenfolge. Das Umfärben von Knoten ist nicht erlaubt. Wir untersuchen die nötige und hinreichende Anzahl an Farben, mit der sämtliche Graphen erkundet werden können. Wir beweisen, dass im Allgemeinen genau  $n - 1$  Farben genügen; bei Bäumen sind es genau 3 Farben und  $\min\{2k - 3, n - 1\}$  Farben sind nötig und  $\min\{2k - 1, n - 1\}$  genügen bei Graphen mit  $n$  Knoten und Umfang  $k$ , wobei der Umfang die Länge eines längsten Kreises ist. Dies gilt allerdings alles nur, wenn der Algorithmus sämtliche Graphen erkunden muss und nicht nur gewisse Graphklassen. Wir geben ein Beispiel einer Graphklasse, bei der jeder Graph mit nur 4 Farben erkundet werden kann, obwohl alle diese Graphen maximalen Umfang haben. Zudem beweisen wir, dass das Umfärben von Knoten sehr mächtig ist, indem wir einen Umfärbe-Algorithmus angeben, der mit bloss 7 Farben auskommt und damit sämtliche Graphen erkunden kann.

Als Drittes analysieren wir das einfache Online-Rucksackproblem mit Reservie-

rung. Bei diesem Problem erhält der Algorithmus nacheinander die Grössen der Gegenstände  $s_1, s_2, \dots$  und muss jeden Gegenstand einpacken, verwerfen oder reservieren. Das Ziel ist, die Summe der Grössen der eingepackten Gegenstände zu maximieren, wobei der Packinhalt 1 nicht überschreiten darf. Wenn die Eingabe endet, darf der Algorithmus die reservierten Gegenstände noch benutzen, um einen grösseren Packinhalt zu erzielen. Es dürfen beliebig viele Gegenstände reserviert werden, aber am Ende zahlt der Algorithmus die maximale Summe der Grössen der reservierten Gegenstände multipliziert mit einem *Reservierungsfaktor*  $\alpha \in [0, 1]$ , der ein Parameter des Problems ist. Dieses Modell schlägt eine Brücke zwischen der Online- und der Offlinewelt: Für  $\alpha = 0$  ist das Problem im Grunde genommen das Offline-Rucksackproblem und für  $\alpha = 1$  ist es das normale Online-Rucksackproblem. Für grosse und mittlere Reservationsfaktoren geben wir dichte obere und untere Schranken an. Für kleine Reservationsfaktoren beweisen wir nah beieinanderliegende obere und untere Schranken, die gegen 1 streben, wenn  $\alpha$  gegen Null geht.

*The great difference between voyages rests  
not with the ships, but with the people you  
meet on them.*

Amelia E. Barr

## Acknowledgment

Many people helped and encouraged me in the course of writing this thesis. First, I would like to thank God, who strengthened and guided me throughout this work. He guides most often through humans, and here, I would like to thank first my supervisor Juraj Hromkovič and my advisor Hans-Joachim Böckenhauer for their support. Juraj Hromkovič ignited my passion for computer science and he gave me the opportunity to work in his research group while being employed in the industry. He never pushed me and he was very considerate towards family matters. Hans-Joachim Böckenhauer kept encouraging me when I was dispirited and he was always there for questions and research discussions. He provided numerous suggestions on how to write up the thesis, underpinned with computer scientists anti-jokes, and he thoroughly proofread everything. Thank you both very much.

I also thank all members of the research group at the ETH, who made working there fun. Thank you, Elisabet Burjons, for all those interesting discussions over lunch; Urs Hauser, for the chats about board games; Angélica Herrera, for the hiking conversations and your patience with my emails; Dennis Komm, for making me feel I could always come to you with problems; Jan Lichtensteiger, for the discussions about photography; Giovanni Serafini, for always asking about my family. Thank you, Fabian Frei, for your sedulous succor without being condescending, for the numerous discussions about God and mathematics, and for your friendship.

I want to express my gratitude towards the researchers I met along this journey, for all the inspiring discussions, helpful ideas, joint papers, fascinating hikes, and productive Mojitos. Thank you, Edith Hemaspaandra, Lane Hemaspaandra, Ralf Klasing, Rastislav Královič, Richard Královič, Aurea Mömke, Tobias Mömke, Xavier Muñoz, Martin Raszyk, Peter Rossmanith, and Walter Unger.

I am very thankful to both Rastislav Královič and Martin Dietzfelbinger for agreeing to review this thesis.

Most importantly, I thank my mom and my dad, to whom I owe most, for their support and sparking enthusiasm and for always being there for me, together with my sisters Rebekka and Tabita. I am also grateful to my mom-in-law and my dad-in-law, for their advice and ceaseless care, and to my brother-in-law, for all the fun bouldering sessions. I want to thank also my three children, whom I love very much, for the laughter and those rare moments of silence. Finally, I thank my dear Steffi, for her ceaseless support and for being the love of my life.





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
	Basic Model and Notation . . . . .	4
	Contribution . . . . .	7
<b>2</b>	<b>Bounds for <math>c</math>-Ideal Hashing</b>	<b>9</b>
2.1	Introduction . . . . .	9
	General Setting and Notation . . . . .	10
	Applications of Hashing . . . . .	10
	Theory of Hashing . . . . .	11
	Determinism versus Randomization . . . . .	12
	Our Model and the Connection to Advice Complexity . . . . .	13
	Organization . . . . .	14
2.2	Related Work and Contribution . . . . .	14
2.3	General Bounds on $H_c$ . . . . .	15
2.4	Estimations for $\mathbb{P}(\alpha_{\max} \leq c\alpha)$ . . . . .	20
	Upper Bound . . . . .	20
	Lower Bound . . . . .	23
2.5	Improvements For Edge Cases . . . . .	26
2.6	Advice Complexity of Hashing . . . . .	28
2.7	Conclusion . . . . .	29
<b>3</b>	<b>Zero-Memory Graph Exploration with Unknown Incoming Ports</b>	<b>31</b>
3.1	Introduction . . . . .	31
	Related work . . . . .	32
	Basic Definitions . . . . .	34
3.2	Exploration of Trees . . . . .	35
3.3	Exploration of General Graphs . . . . .	38
	Upper Bound . . . . .	38
	Lower Bound . . . . .	41
	Non-Uniform Algorithms . . . . .	45
3.4	Between Trees and General Graphs . . . . .	49
3.5	Special Graph Classes . . . . .	55
3.6	Exploration with Recoloring . . . . .	61
3.7	Conclusion . . . . .	66
<b>4</b>	<b>Online Knapsack with a Storeroom</b>	<b>69</b>
4.1	Introduction . . . . .	69
4.2	Model and Notation . . . . .	72
4.3	Related Work . . . . .	72

4.4	Upper Bounds for Large Reservation Factors . . . . .	75
	Range $\varphi \leq \alpha \leq 1$ . . . . .	77
	Range $\sqrt{2} - 1 \leq \alpha \leq \varphi$ . . . . .	77
	Range $0 < \alpha \leq \sqrt{2} - 1$ . . . . .	78
4.5	Lower Bounds for Large Reservation Factors . . . . .	82
	Range $\varphi \leq \alpha < 1$ . . . . .	82
	Range $\sqrt{2} - 1 \leq \alpha \leq \varphi$ . . . . .	82
	Range $0 \leq \alpha \leq \sqrt{2} - 1$ . . . . .	84
4.6	Upper Bounds for Small Reservation Factors . . . . .	87
4.7	Lower Bound for Small Reservation Factors . . . . .	91
4.8	Conclusion . . . . .	97
<b>5</b>	<b>Conclusion</b>	<b>99</b>
	<b>Appendix</b>	<b>101</b>
A	Proof of <a href="#">Lemma 2.9</a> . . . . .	101
B	Calculation for Intersection Points in <a href="#">Section 4.5</a> . . . . .	102

# 1

## Introduction

The difficulty of NP-hard problems has become one of the prevalent topics in theoretical computer science. It seems unlikely that NP-hard problems can be solved in polynomial time and yet the difficulty of answering whether P indeed is unequal to NP has confounded researchers for decades. Much research has been dedicated to try new techniques or apply new approaches to find solutions for at least some instances of NP-hard problems or to find good approximate solutions. Parameterization has proved to be paramount for many problems in practice. Here, the hardness of an instance is classified based on some parameter of the problem and for problem instances where the parameter value is small—which are often of practical importance—, solutions can often be found efficiently. Approximation, where we do not strive for an optimal solution but only for a solution that is relatively good, has seen similar success for other problems. Another approach is to consider a slightly different problem; often times, small changes to the definition of NP-hard problems decrease the computational complexity significantly.

And yet, many problems in everyday applications are far more difficult than NP-hard problems. There are many situations in real life where the input to a problem is not known completely while at the same time, irrevocable decisions have to be taken. Such problems are called *online problems* and strategies to solve these problems are called *online algorithms*. More precisely, online problems are optimization problems where the input arrives piece by piece, and with each piece of the input, a piece of the final solution has to be given. Consider, for example, an airplane company that wants to optimize the number of passengers per flight. If many passengers want to visit, for instance, Paris, the company can offer more flights or demand a higher price for the ticket. If only few passengers want to go to Paris, the company can cancel some flights and use the airplanes for other flights. If the company knew in advance how many passengers wanted to fly to which destination and which price they are willing to pay for the flight, they could achieve a much higher profit, or, in company terms, deliver a much better service to the passengers. Moreover, if the company knew the needs of the passengers in advance, this would be an easy task,

in some sense. It might still be computationally very expensive—and maybe even infeasible—to calculate an optimal solution, but at least there is a way to calculate an optimal solution and the airplane company can concentrate on the computational difficulties.

An example closer to computer science is scheduling. Here, we have a company with  $m$  machines and  $n$  tasks arriving one by one; each task has to be scheduled on some machine. Each task takes a certain time to be processed and can only be processed on certain machines. The goal is to distribute the tasks on the machines in such a manner that the makespan, the time until the last task has been processed, is minimized. Let us suppose now that there are 2 machines and 2 tasks arrive that both take one unit of time and can both be scheduled on either of the machines. If an algorithm schedules both tasks on a single machine, then there might be no other task and the algorithm has a makespan of 2, but could have achieved a makespan of 1 if it had distributed the tasks evenly. Conversely, if the algorithm distributes the task and each machine processes one task, then a new task might arrive that uses 2 time units. Now, the algorithm has a makespan of 3 but could have achieved a makespan of 2 if it had scheduled the first two tasks on a single machine.

This illustrates that, for online problems, the challenge lies not in the time and space complexity as is the case for (offline) NP-hard problems. Rather, the focus is on finding good solutions compared to a hypothetical optimal solution that could have been computed—albeit maybe with arbitrarily large computational power—if the whole input had been known in advance. Online problems merit being studied on their own; however, designing online algorithms can also be seen as an orthogonal approach to tackling difficult problems: instead of trying to make the problem easier, the problem becomes harder, but the focus on the quality of the solution can be fruitful in the design of classical approximation algorithms.

Online algorithms have been studied for more than 50 years. Therefore, the term *online*, at the beginning often written as *on-line*, does not stem from any connection to the internet and the normal meaning of online nowadays; however, it is difficult to track the exact origin, although it likely comes from being connected to a system via a cable. Typical areas where online problems arise naturally and where much of the early research focused on are scheduling, bin packing, networking, and resource allocation. Traditionally, the performance of online algorithms was measured using stochastic modeling, where one assumes a probability distribution on the input and then compares the solution of the algorithm with an average-case solution. This measurement has the disadvantage that the input distribution is often unknown, and wrong assumptions about the distribution lead to a wrong estimation about the performance of the algorithm in practice. Moreover, even if the distribution is known precisely, it may have to be simplified to enable or facilitate the calculations. The stochastic model rapidly lost popularity after the seminal paper by Daniel Sleator and Robert Tarjan in 1985 [55], where they introduced the concept of comparing the performance of an online algorithm to an optimal *offline* solution and promoted the use of worst-case analysis—which is called *competitive analysis* in the context of online problems—over the then usual average-case analysis. Since then, a great deal of research has followed their approach and competitive analysis became the standard method to analyze online algorithms.

However, competitive analysis has its drawbacks as well. Demanding good

---

solutions in the worst case sets the bar very high, and assessing the performance of an online algorithm by its performance in the worst case seems too pessimistic. Various approaches have been proposed to remedy this issue. However, finding good solutions is like walking a tightrope. If a new feature is introduced that does not help the algorithm enough, as for instance *lookahead*, then this feature often does not help at all. If a new feature is introduced that empowers the algorithm enough, then solving the online problem might have become too easy, rendering the results to be of little importance in practice.

In 2008, Stefan Dobrev, Rastislav Kráľovič and Dana Pardubská invented the notion of *advice complexity* [20, 21] and thus solved this dilemma. The driving question in the development of the model was “How much information is an online algorithm lacking compared to an offline algorithm?” The idea was to introduce a trade-off between how much the algorithm is empowered and how good the quality of its solution becomes. In this model, one measures how many additional bits of information an online algorithm needs to be optimal or, for instance, only 2 times worse than an optimal solution. The model assumes that there is an omniscient oracle that shares with the online algorithm additional information about the given instance in the form of *advice bits*. The more advice is used, the better an online algorithm is able to compete with offline algorithms. If no advice is used, we are in the pessimistic situation of normal competitive analysis. If the advice encodes the whole instance, we are in the case of the offline problem—with the difference that we still measure the quality compared to an optimal offline algorithm and do not analyze the space or time complexity.

In this thesis, as the title suggests, we explore the edge of the online world. By this, we mean that we study various topics that are between offline and online problems or that are “extremely online.” To be more precise, we consider three topics that shed light on the realm of online problems from different angles. As a first topic, after explaining the online model and advice complexity more formally in the next section, we consider in [Chapter 2](#) *hashing* as an online problem and analyze its advice complexity. Hashing is a key problem in connecting theory with applications. Roughly speaking, in hashing, one is required to map a large set to a small set such that the large set is distributed well, that is, the images of the elements of the small set under the chosen function are of equal size. Usually, hashing is not regarded as an online problem; hence, we are first going to define our model and choose an appropriate cost function to be able to use competitive analysis. We will see that this cost function is closely related to  $c$ -ideality and we carefully adapt combinatorial approaches known in the hashing community to find upper and lower bounds on  $c$ -ideal function families and to be able to argue about the advice complexity of hashing. By casting hashing into the online world and analyzing its advice complexity, we in some sense extend the border of the online world.

In [Chapter 3](#), we study a particular *zero-memory graph exploration* problem. Graph exploration is a well-known problem that is close to being an online problem—but the input may depend on the output of the algorithm—and we can roughly describe it as follows: An agent starts on a vertex of a graph and has to visit every vertex of the graph; however, on each vertex, the agent only sees the neighbors or the adjacent edges of the current vertex. Graph exploration is almost always regarded as online problem. We restrict the model as much as possible and require that the agent

has no persistent memory, that is, on each vertex, no information about anything that happened before is available. Moreover, the agent does not even know from which neighbor or which edge it entered the current vertex. Thus, we give an edge to the “online-ness” of graph exploration. In some sense, we study a similar trade-off as in advice complexity by measuring how much information—which is only going to be locally available—the agent needs such that exploration is possible.

As a last topic, in [Chapter 4](#), we study the *online knapsack problem with reservation*. The knapsack problem is a model offline problem, but its online version has been quite well studied, too. In the online knapsack problem, we are given a knapsack of a certain capacity. Then, items arrive and for each item, we have to decide whether we discard it or pack it into the knapsack. The goal is to maximize the total size of packed items. We endow the online algorithm with an additional capability, namely, the algorithm may reserve items for a certain cost. Specifically, we have a parameter  $\alpha \in [0, 1]$  and at the end, the algorithm incurs a fee that is  $\alpha$  times the maximal size of reserved items during the computation. When  $\alpha$  is zero, an algorithm can reserve everything without additional cost and pack optimally at the end; when  $\alpha$  is one, no smart strategy will ever reserve an element since any potential gain is negated by the reservation cost. This way, the parameter  $\alpha$  stands for “how online” our problem is and by comparing the competitiveness of the problem over the whole range of  $\alpha$ , we smooth the edge between the online and the offline knapsack problem.

We conclude the thesis in [Chapter 5](#) by reviewing the main results and providing an outlook on further research.

## Basic Model and Notation

We give a more formal introduction to online problems and advice complexity. We only state the main definitions, for a more in-depth introduction, we refer to the textbooks by Borodin and El-Yaniv [7] and Komm [39]. We use the notation as described by Komm [39] and state the definitions for *optimization problem*, *online problem*, *online algorithm*, and *competitive ratio* here for convenience:

**Definition 1.1 (Optimization Problem [39]).** *An optimization problem  $\Pi$  consists of a set of instances  $\mathcal{I}$ , a set of solutions  $\mathcal{O}$ , and three functions  $\text{sol}: \mathcal{I} \rightarrow \mathcal{P}(\mathcal{O})$ ,  $\text{quality}: \mathcal{I} \times \mathcal{O} \rightarrow \mathbb{R}$ , and  $\text{goal} \in \{\max, \min\}$ . For every instance  $I \in \mathcal{I}$ ,  $\text{sol}(I) \subseteq \mathcal{O}$  denotes the set of feasible solutions for  $I$ . For every instance  $I \in \mathcal{I}$  and every feasible solution  $O \in \text{sol}(I)$ ,  $\text{quality}(I, O)$  denotes the measure of  $I$  and  $O$ . An optimal solution for an instance  $I \in \mathcal{I}$  of  $\Pi$  is a solution  $\text{OPT}(I) \in \text{sol}(I)$  such that*

$$\text{quality}(I, O) = \text{goal}\{\text{quality}(I, O) \mid O \in \text{sol}(I)\}.$$

*If  $\text{goal} = \min$ , we call  $\Pi$  a minimization problem and write “cost” instead of “quality.” Conversely, if  $\text{goal} = \max$ , we say that  $\Pi$  is a maximization problem and write “gain” instead of “quality.”*

As mentioned above, *online problems* are maximization or minimization problems where the input items arrive piecemeal and with each input item, an irrevocable decision on whether to accept this item in the solution set has to be taken. Formally, online problems can be defined as follows.

**Definition 1.2 (Online Problem [39]).** An online problem  $\Pi$  consists of a set of instances  $\mathcal{I}$ , a set of solutions  $\mathcal{O}$ , and three functions  $\text{sol}$ ,  $\text{quality}$ , and  $\text{goal}$  with the same meaning as for general optimization problems according to [Definition 1.1](#). Every instance  $I \in \mathcal{I}$  is a sequence of requests  $I = (x_1, x_2, \dots, x_n)$  and every output  $O \in \mathcal{O}$  is a sequence of answers  $O = (y_1, y_2, \dots, y_n)$ , where  $n \in \mathbb{N}^+$  (thus, all instances and solutions are finite). An optimal solution for an instance  $I \in \mathcal{I}$  of  $\Pi$  is a solution  $\text{OPT}(I) \in \text{sol}(I)$  such that

$$\text{quality}(I, O) = \text{goal}\{\text{quality}(I, O) \mid O \in \text{sol}(I)\}.$$

If  $\text{goal} = \min$ , we call  $\Pi$  an online minimization problem and write “cost” instead of “quality.” Conversely, if  $\text{goal} = \max$ , we say that  $\Pi$  is an online maximization problem and write “gain” instead of “quality.”

An online algorithm, on the other hand, is an algorithm that solves an online problem and that uses for each element of the sequential input only the “past” information to compute its solution. Moreover, the number of input items is usually not known.

**Definition 1.3 (Online Algorithm [39]).** Let  $\Pi$  be an online problem and let  $I = (x_1, x_2, \dots, x_n)$  be an instance of  $\Pi$ . An online algorithm  $\text{ALG}$  for  $\Pi$  computes the output  $\text{ALG}(I) = (y_1, y_2, \dots, y_n)$ , where  $y_i$  only depends on  $x_1, x_2, \dots, x_i$ ;  $\text{ALG}(I)$  is a feasible solution for  $I$ , that is,  $\text{ALG}(I) \in \text{sol}(I)$ .

The performance of an online algorithm is measured by comparing the quality of the solution (that is, the gain or the cost) with the quality of an optimal offline solution, that is, an exact solution for the problem that can be created with the entire input known and unlimited computational resources. Moreover, this comparison is made in a worst-case scenario, that is, the ratio between the quality of the algorithm’s solution and an optimal solution is taken for the worst possible instance from an algorithm’s point of view. This ratio is called the *competitive ratio*. While Sleator and Tarjan [55] popularized the concept, the term *competitive* was introduced by Karlin et al. [35]. We use the following definition.

**Definition 1.4 (Competitive Ratio [39]).** Let  $\Pi$  be an online problem, and let  $\text{ALG}$  be a consistent [that is,  $\text{ALG}$  computes a feasible solution for every input] online algorithm for  $\Pi$ . For  $c \geq 1$ ,  $\text{ALG}$  is  $c$ -competitive for  $\Pi$  if there is a non-negative constant  $\alpha$  such that, for every instance  $I \in \mathcal{I}$ ,

$$\text{gain}(\text{OPT}(I)) \leq c \cdot \text{gain}(\text{ALG}(I)) + \alpha$$

if  $\Pi$  is an online maximization problem, or

$$\text{cost}(\text{OPT}(I)) \leq c \cdot \text{cost}(\text{ALG}(I)) + \alpha$$

if  $\Pi$  is an online minimization problem. If these inequalities hold with  $\alpha = 0$ , we call  $\text{ALG}$  strictly  $c$ -competitive;  $\text{ALG}$  is called optimal if it is strictly 1-competitive. The competitive ratio of  $\text{ALG}$  is defined as

$$c_{\text{ALG}} := \inf\{c \geq 1 \mid \text{ALG is } c\text{-competitive for } \Pi\}.$$

If for an online algorithm ALG there is no  $c$  such ALG is  $c$ -competitive, we say that the competitive ratio of ALG is *unbounded*. Strict  $c$ -competitiveness is a stronger requirement than non-strict  $c$ -competitiveness. In contrast to strict competitiveness, the constant  $\alpha$  in the definition of non-strict competitiveness allows to ignore small instances that are not in line with the general behavior. However, for online problems where there is a natural bound on the maximal gain or cost—such as in our variant of the knapsack problem in Chapter 4—it does not make sense to analyze non-strict competitiveness since the constant  $\alpha$  would completely conceal the difference in performance between the algorithm’s solution and an optimal offline solution.

To calculate the competitive ratio, it is common to think and speak of an *adversary* that wants to create an instance that is hard for the algorithm, or, to be more precise, where the algorithm performs worst in comparison with an optimal solution.

Finally, we are going to use the concept of *advice complexity*. Advice complexity has been introduced as a tool to measure an online problem’s hardness or information complexity. The original model was refined by Hromkovič et al. [32] and Böckenhauer et al. [6]; a similar model was proposed by Emek et al. [23]. For a survey on advice complexity, see Boyar et al. [8]. Again, for a well-written textbook introduction, we recommend Komm [39] and we are going to cite his definitions in the following.

In the advice model, we assume that there is an all-powerful oracle that works together with the online algorithm. The oracle knows the input  $I$  and writes its advice in binary on an advice tape of infinite length. During the computation of the algorithm, the algorithm may read some bits of the advice tape sequentially. For each input, the amount of advice bits may vary; we write  $n$  for the length of the input  $I$  and we write  $b(n)$  for the maximal number of advice bits read on inputs of length  $n$ . In the advice model, we are interested in the trade-off between the amount  $b(n)$  of advice bits read and the competitive ratio  $c$  achieved.

The game between algorithm ALG, oracle, and adversary works as follows. First, the adversary knows ALG and the oracle and constructs its instance. The adversary knows for each possible instance which advice the algorithm is going to receive and how many advice bits the algorithm is going to read and takes this into account to construct an instance  $I$  such that the competitive ratio of ALG is maximized. The oracle then inspects  $I$  and writes its advice for  $I$  on the advice tape. The algorithm then reads  $I$  in an online manner and during its computation, it reads at most a prefix of length  $b(n)$  from the advice tape.

Since the advice tape is of infinite length, the algorithm cannot gain information by observing the length of the advice (see Komm [39]). We have the following definitions for *online algorithm with advice* and *competitive ratio with advice*; we are going to omit the specification “with advice” in this thesis since it will always be clear from the context whether we consider advice complexity.

**Definition 1.5 (Online Algorithm with Advice [39]).** *Let  $\Pi$  be an online problem and let  $I = (x_1, x_2, \dots, x_n)$  be an instance of  $\Pi$ . An online algorithm ALG with advice for  $\Pi$  computes the output  $\text{ALG}^\phi(I) = (y_1, y_2, \dots, y_n)$ , where  $y_i$  depends on  $x_1, x_2, \dots, x_i$  and  $\phi$ ;  $\phi$  denotes a binary advice string.*

**Definition 1.6 (Competitive Ratio with Advice [39]).** *Let  $\Pi$  be an online problem, and let ALG be a consistent [that is, ALG computes a feasible solution for every*



input] online algorithm with advice for  $\Pi$ , and let  $\text{OPT}$  be an optimal offline algorithm for  $\Pi$ . For  $c \geq 1$ ,  $\text{ALG}$  is  $c$ -competitive with advice complexity  $b(n)$  for  $\Pi$  if there is a non-negative constant  $\alpha$  such that, for every instance  $I \in \mathcal{I}$  of size  $n$ , there is an advice string  $\phi$  such that

$$\text{gain}(\text{OPT}(I)) \leq c \cdot \text{gain}(\text{ALG}^\phi(I)) + \alpha$$

if  $\Pi$  is an online maximization problem, or

$$\text{cost}(\text{OPT}(I)) \leq c \cdot \text{cost}(\text{ALG}^\phi(I)) + \alpha$$

if  $\Pi$  is an online minimization problem, and  $\text{ALG}$  uses at most the first  $b(n)$  bits of  $\phi$ . If the above inequality holds for  $\alpha = 0$ ,  $\text{ALG}$  is called strictly  $c$ -competitive with advice complexity  $b(n)$ ;  $\text{ALG}$  is called optimal with advice complexity  $b(n)$  if it is strictly 1-competitive with advice complexity  $b(n)$ . The competitive ratio of an online algorithm  $\text{ALG}$  with advice complexity  $b(n)$  is defined as

$$c_{\text{ALG}} := \inf\{c \geq 1 \mid \text{ALG is } c\text{-competitive for } \Pi \text{ with advice complexity } b(n)\}.$$

To derive lower bounds for online algorithm with advice, the following observation (see Komm [39]) is crucial.

**Observation 1.1.** *Every algorithm that reads at most  $b(n)$  advice bits for inputs of length  $n$  can be seen as a set of  $2^{b(n)}$  deterministic online algorithms from which for each input, the best strategy is chosen.*

Thus, it becomes clear that advice is strictly stronger than randomization: if there is a randomized online algorithm that is  $c$ -competitive in expectation by reading at most  $r(n)$  random bits for inputs of length  $n$ , then there is a  $c$ -competitive online algorithm with advice complexity  $r(n)$ . As an aside, fascinatingly, for many online problems, the converse is true as well if the advice complexity is sub-linear: If there is a  $c$ -competitive online algorithm with advice complexity  $b(n)$ , where  $b(n) \in o(n)$ , then there is a randomized online algorithm that is  $c$ -competitive in expectation (by potentially reading very many random bits). See Mikkelsen [47] for more information.

## Contribution

All chapters of this thesis are joint work with other authors. [Chapter 2](#) was developed with Fabian Frei; [Chapter 3](#) is a joint work together with Hans-Joachim Böckenhauer, Fabian Frei, and Walter Unger; [Chapter 4](#) emerged from a group work with Hans-Joachim Böckenhauer, Elisabet Burjons, Fabian Frei, Rastislav Královič, Richard Královič, and Peter Rossmanith. All ideas and proofs were developed or discussed with the co-authors. Errors and typos are all mine.

The current proof of [Theorem 2.1](#) was sketched by an anonymous reviewer. [Lemma 2.9](#) and [Theorem 2.6](#) were proven by Fabian Frei, who also suggested the main idea for the algorithm and its analysis for large reservation factors in [Chapter 4](#) and came up with the main idea for the lower bound for large reservation factors.



# 2

## Bounds for $c$ -Ideal Hashing

### 2.1 Introduction

Say you wake up one morning in a hotel with the desire to stroll around and view some sights of the city. Considering your peculiar proclivity towards paranoia, you want to know for every car you see whether you have seen this car before. Whenever you see a car, you can quickly jot down the car plate and while doing so, you can check whether you have noted this number before. You have to be quick though, since cars might follow each other closely and you will not have the time to go through your entire list to check whether a new car plate is in the list. There are many possible car plates, maybe around  $36^8$  for 8-character plates. You estimate to see around 100 cars this day. You are not able to compare the new plate against more than 5 car plates jotted down before. This is a situation where hashing excels. However, this chapter is not about normal hashing.

Luckily, you have heard of hashing and you know that when you first apply a hash function to the car plates—for example by choosing the last digit of the car plate—and then sort the strings by their hash value, you will be able to determine very quickly whether you have seen a given number before. But you are paranoid. Very paranoid. You know that you will only be able to quickly know you have seen a number before *on average*. You strive for perfection, however. Your hash function has to be good in the worst case as well. You want to ensure you apply a hash function such that no hash value appears more than, say, 5 times. You know this is not possible without further aid, but fret not, you have a very powerful friend who, provided you present him a small family of hash functions, is miraculously able to point you to the hash function among the family that achieves your goal—if there is a good hash function among your family, that is. And if the family you present is small. But, you wonder, does such a small family of hash functions exist at all? This chapter deals with that question.

Hashing is one of the most popular tools in computing, both from a practical and a theoretical perspective. Hashing was invented as an efficient method to store

and retrieve information. Its success began at latest in 1968 with the seminal paper “Scatter Storage Techniques” by Robert Morris [49], which was reprinted in 1983 with the following laudation [50]:

From time to time there is a paper which summarizes an emerging research area, focuses the central issues, and brings them to prominence. Morris’s paper is in this class. [...] It brought wide attention to hashing as a powerful method of storing random items in a data structure with low average access time.

This quote illustrates that the practical aspects were the focus of the early research, and rightly so. Nowadays, hashing has applications in many areas in computer science. Before we describe how hashing can be applied, we state the general setting.

### General Setting and Notation

We have a large set  $U$ , the *universe*, of all possible elements, the *keys*. In our example, this would be the set of all possible car plates, so all strings of length 8 over the alphabet  $\{A, \dots, Z, 0, \dots, 9\}$ . Then, there is a subset  $S \subseteq U$  of keys from the universe. This set stands for the unknown elements that appear in our application. In our example,  $S$  corresponds to the set of all car plates that we see on this day. Then, there is a small set  $T$ , the *hash table*, whose  $m$  elements are called *cells* or *slots*. In our example,  $T$  corresponds to our notebook and we organize the entries in our notebook according to the last character on the car plate, so each cell corresponds to a single letter of the alphabet. Typically, the universe is huge in comparison to the hash table, e.g.  $|U| = 2^{|T|}$ . Every function from  $U$  to  $T$  is called a *hash function*; it *hashes* (i.e., maps) keys to cells. We have to choose a function  $h$  from  $U$  to  $T$  such that the previously unknown set  $S$  is distributed as evenly as possible among the cells of our hash table. For an introduction to hashing, one of the standard textbooks is the book by Mehlhorn [44]. We recommend the newer book by Mehlhorn and Sanders [45], and to German readers in particular its German translation by Dietzfelbinger, Mehlhorn and Sanders [17].

While  $U$  and  $T$  can be arbitrary finite sets, we choose to represent their elements by integers—which can always be achieved by a so-called pre-hash function—and let  $U := \{1, \dots, u\}$  and  $T := \{1, \dots, m\}$ . For convenience, we abbreviate  $\{1, \dots, k\}$  by  $[k]$  for any natural number  $k$ . We assume the size  $n$  of the subset to be at least as large as the size of the hash table, that is,  $|S| = n \geq m$  and, to exclude the corner case of hashing almost the entire universe, also  $|U| \geq n^2$ .

### Applications of Hashing

Among the numerous applications of hashing, two broad areas stand out. First, as we have seen, hashing can be used to store and retrieve information. With hashing, inserting new information records, deleting records, and searching records can all be done in expected constant time, i.e., the number of steps needed to insert, find, or delete an information record does not depend on the size of  $U$ ,  $S$ , or  $T$ . For an example of such an application, consider dictionaries in Python, which are implemented using a hash table. To create a dictionary, you specify pairs of keys (e.g., scientists) and values (e.g., their Erdős number). Now, retrieving values by

calling the key, inserting new key-value-pairs, and deleting key-value-pairs can all be done in time independent of the number of scientists. Another example is the Rabin-Karp algorithm [37], which searches a pattern in a text and can be used for instance to detect plagiarism in doctoral theses. The naïve approach is to slide the pattern  $p$ , which has length  $m$ , letter by letter over the text  $t$ , which has length  $n$ , and then compare the pattern with each substring  $s$  of length  $m$  of the text. Here, the universe  $U$  consists of all possible texts (in a certain language) and  $S$  are all texts of length  $n$ . This needs  $n - m + 1$  comparisons between pattern and substrings and for each comparison,  $m$  letters have to be compared. In total, there are  $n(n - m + 1)$  comparisons, which is a large computational effort. A much faster approach in expectation is to use the Rabin-Karp algorithm. Here, the hash values of the pattern and all substrings of length  $m$  are computed first. For a hash function with the property that the hash of a string, say  $s_2 = 23456$ , can easily be derived from the hash of the previous string, say  $s_1 = 12345$ , this is computationally not too expensive. Then, the hash value of the pattern is compared to the hash value of each substring. So far, this is not an improvement. Now, however, the pattern and a substring are only compared if their hash values match. This leads to an average time complexity of  $O(n + m)$ , which is much better than the previous  $O(nm)$ .

The second main application area is cryptography. In cryptographic hashing, the hash function has to fulfill additional requirements, for example that it is computationally very hard to reconstruct a key  $x$  from its hash  $h(x)$  or to find a colliding key, i.e., a  $y \in U$  with  $h(y) = h(x)$ . Typical use cases here are, for example, digital signatures and famous examples of cryptographic hash functions are the checksum algorithms such as MD5 or SHA, which are often used to check whether two files are equal. Moreover, and perhaps particularly interesting for computer science, hashing is a useful tool in the design of randomized algorithms and their derandomization. Further examples and more details can be found for example in the useful article by Luby and Wigderson [41].

## Theory of Hashing

Soon after the early focus on the practical aspect, a rich theory on hashing evolved. In this theory, randomization plays a pivotal role. From a theory point of view, we aim for a hash function  $h$  that reduces collisions among the yet-to-be-revealed keys of  $S$  to a minimum. One possibility for selecting  $h$  is to choose the image for each key in  $U$  uniformly at random among the  $m$  cells of  $T$ . We can interpret this as picking a random  $h$  out of the family  $\mathcal{H}_{\text{all}}$  of all hash functions.<sup>1</sup> Then the risk that two keys  $x, y \in U$  collide is only  $1/m$ , that is,

$$\forall x, y \in U, x \neq y: \mathbb{P}_{h \in \mathcal{H}_{\text{all}}} (h(x) = h(y)) = \frac{1}{m}.$$

On the downside, this random process can result in computationally complex hash functions whose evaluation has space and time requirements in  $\Theta(u \ln m)$ . Efficiently computable functions are necessary in order to make applications feasible, however. Consequently, the assumption of such a simple uniform hashing remains in large part a theoretical one with little bearing on practical applications; it is invoked primarily to simplify theoretical manipulations.

<sup>1</sup>Note that the number of hash functions,  $|\mathcal{H}_{\text{all}}| = m^u$ , is huge.

The astonishing discovery of small hashing families that can take on the role of  $\mathcal{H}_{\text{all}}$  addresses this problem. In their seminal paper in 1979, Larry Carter and Mark Wegman [11] introduced the concept of *universal hashing families* and showed that there exist small universal hashing families. A family  $\mathcal{H}$  of hash functions is called *universal* if it can take the place of  $\mathcal{H}_{\text{all}}$  without increasing the collision risk:

$$\mathcal{H} \text{ is universal} \iff \forall x, y \in U, x \neq y: \mathbb{P}_{h \in \mathcal{H}}(h(x) = h(y)) \leq 1/m.$$

In the following years, research has been successfully dedicated to revealing universal hashing families of comparably small size that exhibit the desired properties.

## Determinism versus Randomization

Deterministic algorithms cannot keep up with this incredible performance of randomized algorithms, i.e., as soon as we are forced to choose a single hash function without knowing  $S$ , there is always a set  $S$  such that all keys are mapped to the same cell. Consequently, deterministic algorithms have not been of much interest. Using the framework of advice complexity, we measure how much additional information deterministic algorithms need in order to hold their ground or to even have the edge over randomized algorithms.

In order to analyze the advice complexity of hashing, we have to view hashing as an online problem. This is not too difficult; being forced to take irrevocable decisions without knowing the whole input is the essence of online algorithms, as we describe in the main introduction of this thesis. In the standard hashing setting, we are required to predetermine our complete strategy, i.e. our hash function  $h$ , with no knowledge about  $S$  at all. In this sense, standard hashing is an ultimate online problem. We could relax this condition and require that  $S = \{s_1, \dots, s_n\}$  is given piecemeal and the algorithm has to decide to which cell the input  $s_i$  is mapped only upon its arrival. However, this way, an online algorithm could just distribute the input perfectly, which would lead, as in the case of  $\mathcal{H}_{\text{all}}$ , to computationally complex functions.

Therefore, instead of relaxing this condition, we look for the reason why there is such a gap between deterministic and randomized strategies. The reason is simple: Deterministic strategies are measured according to their performance in the worst case, whereas randomized strategies are measured according to their performance on average. However, if we measure the quality of randomized strategies from a worst-case perspective, the situation changes. In particular, while universal hashing schemes prove incredibly useful in everyday applications, they are far from optimal from a worst-case perspective, as we illustrate now.

We regard a hash function  $h$  as an algorithm operating on the input set of keys  $S$ . We assess the performance of  $h$  on  $S$  by the resulting *maximum cell load*:  $\text{cost}(h, S) := \alpha_{\text{max}} := \max\{\alpha_1, \dots, \alpha_m\}$ , where  $\alpha_i$  is the *cell load* of cell  $i$ , that is,  $\alpha_i := |h^{-1}(i) \cap S|$ , the number of keys hashed to cell  $i$ . This is arguably the most natural cost measurement for a worst-case analysis. Another possibility, the total number of collisions, is closely related.<sup>2</sup>

The *average load* is no useful measurement option as it is always  $\alpha := n/m$ . The worst-case cost  $n$  occurs if all  $n = |S|$  keys are assigned to a single cell. The optimal

<sup>2</sup>This number can be expressed as  $\sum_{k=1}^m \binom{\alpha_k}{2} \in \Theta(\alpha_1^2 + \dots + \alpha_m^2) \subseteq \mathcal{O}(m \cdot \alpha_{\text{max}}^2)$ .

cost, on the other hand, is  $\lceil n/m \rceil = \lceil \alpha \rceil$  and it is achieved by distributing the keys of  $S$  into the  $m$  cells as evenly as possible.

Consider now a randomized algorithm ALG that picks a hash function  $h \in \mathcal{H}_{\text{all}}$  uniformly at random. A long-standing result [26], proven nicely by Raab and Steger [52], shows the expected cost  $\mathbb{E}_{h \in \mathcal{H}_{\text{all}}}[\text{cost}(h, S)]$  to be in  $\Omega\left(\frac{\ln m}{\ln \ln m}\right)$ , as opposed to the optimum  $\lceil \alpha \rceil$ . The same holds true for smaller universal hashing schemes such as polynomial hashing, where we randomly choose a hash function  $h$  out of the family of all polynomials of degree  $\mathcal{O}\left(\frac{\ln n}{\ln \ln n}\right)$ . However, no matter which universal hash function family is chosen, it is impossible to rule out the absolute worst-case: For every chosen hash function  $h$  there is, due to  $u \geq n^2$ , a set  $S$  of  $n$  keys that are all mapped to the same cell.

## Our Model and the Connection to Advice Complexity

We are, therefore, interested in an alternative hashing model that allows for meaningful algorithms better adapted to the worst case: What if we did not choose a hash function at random but could start with a family of hash functions and then always use the best function, for any set  $S$ ? What is the trade-off between the size of such a family and the upper bounds on the maximum load they can guarantee? In particular, how large is a family that always guarantees an almost optimal load?

In our model, the task is to provide a set  $\mathcal{H}$  of hash functions. This set should be small in size while also minimizing the term

$$\text{cost}(\mathcal{H}, \mathcal{S}) := \max_{S \in \mathcal{S}} \min_{h \in \mathcal{H}} \text{cost}(h, S),$$

which is the best cost bound that we can ensure across a given set  $\mathcal{S}$  of inputs by using any hash function in  $\mathcal{H}$ . Hence, we look for a family  $\mathcal{H}$  of hash functions that minimizes both  $|\mathcal{H}|$  and  $\text{cost}(\mathcal{H}, \mathcal{S})$ . These two goals conflict, resulting in a trade-off, which we parameterize by  $\text{cost}(\mathcal{H}, \mathcal{S})$ , using the notion of *c-ideality*.

**Definition 2.1 (c-Ideality).** *Let  $c \geq 1$ . A function  $h : U \rightarrow T$  is called  $c$ -ideal for a subset  $S \subseteq U$  of keys if  $\text{cost}(h, S) = \alpha_{\max} \leq c\alpha$ . In other words, a  $c$ -ideal hash function  $h$  assigns at most  $c\alpha$  elements of  $S$  to each cell of the hash table  $T$ .*

*Similarly, a family of hash functions  $\mathcal{H}$  is called  $c$ -ideal for a family  $\mathcal{S}$  of subsets of  $U$  if, for every  $S \in \mathcal{S}$ , there is a function  $h \in \mathcal{H}$  such that  $h$  is  $c$ -ideal for  $S$ . This is equivalent to  $\text{cost}(\mathcal{H}, \mathcal{S}) \leq c\alpha$ . If  $\mathcal{H}$  is  $c$ -ideal for  $\mathcal{S}_n := \{S \subseteq U; |S| = n\}$  (that is, all sets of  $n$  keys), we simply call  $\mathcal{H}$   $c$ -ideal.*

We see that  $c$ -ideal families of hash functions constitute algorithms that guarantee an upper bound on  $\text{cost}(\mathcal{H}) := \text{cost}(\mathcal{H}, \mathcal{S}_n)$ , which fixes one of the two trade-off parameters. Now, we try to determine the other one and find, for every  $c \geq 1$ , the minimum size of a  $c$ -ideal family, which we denote by

$$H_c := \min\{|\mathcal{H}|; \text{cost}(\mathcal{H}) \leq c\alpha\}.$$

Note that  $c \geq m$  renders the condition of  $c$ -ideality void since  $n = m\alpha$  is already the worst-case cost; we always have  $\text{cost}(\mathcal{H}) \leq n \leq c\alpha$  for  $c \geq m$ . Consequently, every function is  $m$ -ideal, every non-empty family of functions is  $m$ -ideal, and  $H_m = 1$ .

With the notion of  $c$ -ideality, we can now talk about  $c$ -competitiveness of hashing algorithms. Competitiveness is directly linked to  $c$ -ideality since, for

each  $S$ ,  $\text{cost}(\text{ALG}(S)) := \alpha_{\max}$  and the cost of an optimal solution is always  $\text{cost}(\text{OPT}(S)) = \lceil \alpha \rceil$ . Therefore, a hash function that is  $c$ -ideal is  $c$ -competitive as well. Clearly, no single hash function is  $c$ -ideal; however, a hash function family  $\mathcal{H}$  can be  $c$ -ideal and an algorithm that is allowed to choose, for each  $S$ , the best hash function among a  $c$ -ideal family  $\mathcal{H}$  is thus  $c$ -competitive. As we have remarked in [Observation 1.1](#), such a choice corresponds exactly to an online algorithm with advice that reads  $\lceil \log(|\mathcal{H}|) \rceil$  advice bits, which is just enough to indicate the best function from a family of size  $|\mathcal{H}|$ . Since, by definition, there is a  $c$ -ideal hash function family of size  $H_c$ , there is an online algorithm with advice complexity  $\lceil \log(H_c) \rceil$  as well. Moreover, there can be no  $c$ -competitive online algorithm  $\text{ALG}$  that reads less than  $\lceil \log H_c \rceil$  advice bits: If there were such an algorithm, there would be a  $c$ -ideal hash function family of smaller size than  $H_c$ , which contradicts the definition of  $H_c$ .

We will discuss the relation between  $c$ -ideal hashing and advice complexity in more depth in [Section 2.6](#).

## Organization

This chapter is organized as follows. In [Section 2.2](#), we give an overview of related work and our contribution. We present our general method of deriving bounds on the size of  $c$ -ideal families of hash functions in [Section 2.3](#). [Section 2.4](#) is dedicated to precisely calculating these general bounds. In [Section 2.5](#), we give improved bounds for two edge cases. In [Section 2.6](#), we analyze the advice complexity of hashing. We recapitulate and compare our results in [Section 2.7](#).

## 2.2 Related Work and Contribution

There is a vast body of literature on hashing. Indeed, hashing still is a very active research area today and we cannot even touch upon the many aspects that have been considered. Instead, in this section, we focus on literature very closely connected to  $c$ -ideal hashing. For a coarse overview of hashing in general, we refer to the survey by Chi and Zhu [\[12\]](#) and the seminar report by Dietzfelbinger et al. [\[18\]](#).

The advice complexity of hashing has not yet been analyzed; however,  $c$ -ideality is a generalization of perfect  $k$ -hashing, sometimes also called  $k$ -perfect hashing. For  $n \leq m$  (i.e.,  $\alpha \leq 1$ ) and  $c = 1$ , our definition of  $c$ -ideality allows no collisions and thus reduces to perfect hashing, a notion formally introduced by Mehlhorn in 1984 [\[44\]](#). For this case, Fredman and Komlós [\[24\]](#) proved in 1984 the bounds

$$H_1 \in \Omega\left(\frac{m^{n-1} \log(u)(m-n+1)!}{m! \log(m-n+2)}\right) \text{ and } H_1 \in \mathcal{O}\left(\frac{-n \log(u)}{\log\left(1 - \frac{m!}{(m-n)!m^n}\right)}\right).$$

In 2000, Blackburn [\[2\]](#) improved their lower bound for  $u$  large compared to  $n$ . Recently, in 2022, Guruswami and Riazonov [\[27\]](#) improved their bound as well. None of these proofs generalize to  $n > m$ , that is,  $\alpha > 1$ .

Another notion that is similar to  $c$ -ideality emerged in 1995. Naor et al. [\[51\]](#) introduced  $(u, n, m)$ -splitters, which coincide with the notion of 1-ideal families for  $\alpha \geq 1$ . They proved a result that translates to

$$H_1 \in \mathcal{O}\left(\sqrt{2\pi\alpha}^m e^{\frac{m}{12\alpha}} \sqrt{n \ln u}\right). \quad (2.1)$$



Since the requirements for  $c$ -ideality are strongest for  $c = 1$ , this upper bound holds true for the general case of  $c \geq 1$  as well.

We extend these three results to the general case of  $n \geq m$  and  $c \geq 1$ . Moreover, we tighten the third result further for large  $c$ . Specifically, we prove the following new bounds:

$$H_c \geq (1 - \varepsilon) \exp\left(\frac{m}{e^\alpha} (1 - \varepsilon) \left(\frac{\alpha}{c\alpha + 1}\right)^{c\alpha + 1}\right) \quad (\text{Theorem 2.3}) \quad (2.2)$$

$$H_c \geq \frac{\ln u - \ln(c\alpha)}{\ln m} \quad (\text{Theorem 2.5}) \quad (2.3)$$

$$H_c \leq \left[ \left( \sqrt{2\pi c\alpha}^{1/c} c^\alpha \frac{e^{\frac{1}{12c^2\alpha}}}{(\alpha + 1)^{1 - \frac{1}{c}}} \right)^m \sqrt{\frac{n}{2\pi}} \ln u \right] \quad (\text{Theorem 2.4}) \quad (2.4)$$

$$H_c \in \mathcal{O}\left(\frac{n \ln u}{\ln t}\right) \text{ for any } t \geq 1 \text{ and } c \in \omega\left(t \frac{\ln n}{\ln \ln n}\right) \quad (\text{Theorem 2.6}) \quad (2.5)$$

Note that (2.4) coincides with (2.1) for  $c = 1$  and is only an improvement for  $\alpha$  and  $c$  slightly larger than 1. Since  $H_c \leq H_{c'}$  for  $c \geq c'$ , the bound still improves slightly upon (2.1) in general, depending on the constants hidden in the  $\mathcal{O}$ -notation in (2.1). Interestingly, the size  $u$  of the universe does not appear in (2.2); a phenomenon discussed in [24]. Fredman and Komlós used information-theoretic results based on the Hansel Lemma [31] to obtain a bound that takes the universe size into account. Körner [40] expressed their approach in the language of graph entropy. It is unclear, however, how these methods could be generalized to the case  $\alpha > 1$  in any meaningful way.

The straightforward approach of proving a lower bound on  $H_1$  is to use good Stirling estimates for the factorials in  $H_1 \geq \binom{u}{n} / \left(\frac{u}{\alpha}\right)^m$ ; see Mehlhorn [44]. This yields a lower bound of roughly  $\sqrt{2\pi\alpha}^{m-1} / \sqrt{m}$ , which is better than (2.2) for  $c = 1$ , see Lemma 2.3 for details. Unfortunately, it turns out that we cannot obtain satisfying results for  $c > 1$  with this approach. However, the results from Dubhashi and Ranjan [22] and the method of Poissonization enable us to circumnavigate this obstacle and derive (2.2).

We use our bounds to derive bounds for the advice complexity of hashing, which we will discuss in Section 2.6.

## 2.3 General Bounds on $H_c$

We present our bounds on  $H_c$ , which is the minimum size of  $c$ -ideal families of hash functions. First, we establish a general lower and upper bound.

We use a *volume bound* to lower-bound  $H_c$ . We need the following definition. Let  $M_c$  be the maximum number of sets  $S \in \mathcal{S}_n$  that a single hash function can map  $c$ -ideally, i.e.

$$M_c := \max_{h \in \mathcal{H}_{\text{all}}} |\{S \in \mathcal{S}_n; \alpha_{\max} \leq c\alpha\}| = \max_{h \in \mathcal{H}_{\text{all}}} |\{S \in \mathcal{S}_n; \forall i \in [m] : \alpha_i \leq c\alpha\}|.$$

**Lemma 2.1 (Volume Bound).** *The number of hash functions in a family of hash functions that is  $c$ -ideal is at least the number of sets in  $\mathcal{S}_n$  divided by the number of sets for which a single hash function can be  $c$ -ideal, i.e.*

$$H_c \geq |\mathcal{S}_n| / M_c.$$

*Proof.*  $|\mathcal{S}_n| = \binom{u}{n}$  is the number of subsets of size  $n$  in the universe  $U$  and  $M_c$  is the maximum number of such subsets for which a single function  $h$  can be  $c$ -ideal. A function family  $\mathcal{H}$  is thus  $c$ -ideal for at most  $|\mathcal{H}| \cdot M_c$  of these subsets. If  $\mathcal{H}$  is supposed to be  $c$ -ideal for all subsets—that is, to contain a  $c$ -ideal hash function for every single one of them—we need  $|\mathcal{H}| \cdot M_c \geq |\mathcal{S}_n|$ .  $\square$

To be able to estimate  $M_c$ , we consider hash functions that distribute the  $u$  keys of our universe as evenly as possible:

**Definition 2.2 (Balanced Hash Function).** A balanced hash function  $h$  partitions the universe into  $m$  parts by allotting to each cell  $\lceil u/m \rceil$  or  $\lfloor u/m \rfloor$  elements. We denote the set of balanced hash functions by  $\mathcal{H}_{\text{eq}}$ . We write  $h_{\text{eq}}$  to indicate that  $h$  is balanced.<sup>3</sup>

Theorem 2.1 states that exactly all balanced hash functions attain the value  $M_c$ . Therefore, we can limit ourselves to such functions.

**Theorem 2.1 (Balance Maximizes  $M_c$ ).** A function is  $c$ -ideal for the maximal number of subsets if and only if it is balanced. In other words, the number of subsets that are hashed  $c$ -ideally by a hash function equals  $M_c$  if and only if  $h$  is balanced.

*Proof.* For every hash function, the number of keys that are allotted—that is, potentially hashed—to cell  $k$  is  $\beta_k := |h^{-1}(k)|$ . This gives us a decomposition  $(\beta_1, \dots, \beta_m)$  of our universe  $U$ . Note that  $\beta_1 + \dots + \beta_m = u$  and remember that  $h$  is by definition balanced if and only if  $\beta_1, \dots, \beta_m \in \{\lfloor \beta \rfloor, \lceil \beta \rceil\}$ , where we use  $\beta := u/m$ . Note that this is in turn equivalent to the condition that  $\beta_1, \dots, \beta_m$  differ pairwise by at most 1.

Let  $M_c(\beta_1, \dots, \beta_m)$  be the number of subsets  $S \subseteq U$  for which a hash function  $h$  with a decomposition  $(\beta_1, \dots, \beta_m)$  is  $c$ -ideal. Assume that  $h$  is not balanced. Then, there are  $i, j$  such that  $\beta_i \leq \beta_j + 2$ . Assume without loss of generality that  $\beta_1 \leq \beta_2 + 2$ . Choose any element  $a$  from  $\beta_2$ . Let  $h'$  be the function with  $h'(s) = h(s)$  for  $s \neq a$  and  $h'(a) = 1$ , that is,  $h'$  behaves like  $h$  except for mapping  $a$  to cell 1 instead of cell 2. This function has the decomposition  $(\beta_1 + 1, \beta_2 - 1, \beta_3, \dots, \beta_m)$  and is thus more balanced than  $h$ . We show that  $h'$  is  $c$ -ideal for more subsets  $S \subseteq U$  than  $h$ . Thus, we show that by moving towards a more balanced distribution, more sets can be mapped  $c$ -ideally.

Consider any set  $S_1 \subset U$  of  $n - k$  elements from  $U$  with  $k \leq 2c\alpha$  that does not contain any element from  $h^{-1}(1)$  or  $h^{-1}(2)$ . Let  $r$  and  $r'$  be the number of sets  $S_2 \subset h^{-1}(1) \cup h^{-1}(2)$  of size  $k$  such that  $S = S_1 \cup S_2$  is mapped  $c$ -ideally by  $h$  and  $h'$ , respectively. We analyze  $r' - r$ , that is, by how much the number of  $c$ -ideally mapped sets increases when  $a$  is mapped to the first cell instead of the second one. On the one hand, any set  $S_2$  containing  $a$  and exactly  $c\alpha$  elements (different from  $a$ ) from  $h^{-1}(2)$  is mapped  $c$ -ideally by  $h'$  but not by  $h$ . There are  $\binom{\beta_2 - 1}{c\alpha} \binom{\beta_1}{k - c\alpha - 1}$  such sets. On the other hand, for any set  $S_2$  containing  $a$  and exactly  $c\alpha$  elements from  $h^{-1}(1)$  is not mapped  $c$ -ideally by  $h'$  although it was by  $h$ . There are  $\binom{\beta_1}{c\alpha} \binom{\beta_2 - 1}{k - c\alpha - 1}$  such sets. Any set not containing  $a$  is of course hashed identically by  $h$  and  $h'$ . Hence,  $r' - r$  is exactly

$$\binom{\beta_2 - 1}{c\alpha} \binom{\beta_1}{k - c\alpha - 1} - \binom{\beta_1}{c\alpha} \binom{\beta_2 - 1}{k - c\alpha - 1}.$$

<sup>3</sup>Note that, for any  $h_{\text{eq}}$ , there are exactly  $(u \bmod m)$  cells of size  $\lceil u/m \rceil$  since  $\sum_{k=1}^m |h_{\text{eq}}^{-1}(k)| = u$ .

Writing  $s = \beta_2 - \beta_1 - 1$  and  $x = c\alpha$  and  $y = k - c\alpha - 1$ , we find that this expression is larger than or equal to 0 if and only if any of the following equivalent conditions holds.

$$\begin{aligned} \binom{\beta_2 - 1}{c\alpha} \binom{\beta_1}{k - c\alpha - 1} &\geq \binom{\beta_1}{c\alpha} \binom{\beta_2 - 1}{k - c\alpha - 1} \\ \frac{(\beta_1 + s)!}{x!(\beta_1 + s - x)!} \frac{\beta_1!}{y!(\beta_1 - y)!} &\geq \frac{\beta_1!}{x!(\beta_1 - x)!} \frac{(\beta_1 + s)!}{y!(\beta_1 + s - y)!} \\ \frac{(\beta_1 + s - y)!}{(\beta_1 - y)!} &\geq \frac{(\beta_1 + s - x)!}{(\beta_1 - x)!} \\ (\beta_1 + s - y) \dots (\beta_1 + s - y - s + 1) &\geq (\beta_1 + s - x) \dots (\beta_1 + s - x - s + 1) \\ x &\geq y \\ c\alpha &\geq k - c\alpha - 1 \geq 2c\alpha - c\alpha - 1 \geq c\alpha - 1. \end{aligned}$$

Therefore, whenever a hash function is not balanced, there is another hash function that hashes more sets  $c$ -ideally.  $\square$

Now fix a balanced hash function  $h \in \mathcal{H}_{\text{eq}}$ . We switch to a randomization perspective: Draw an  $S \in \mathcal{S}_n$  uniformly at random. The cell loads  $\alpha_k$  are considered random variables that assume integer values based on the outcome  $S \in \mathcal{S}_n$ . The probability that our fixed  $h$  hashes the random  $S$   $c$ -ideally is exactly

$$\mathbb{P}(\alpha_{\max} \leq c\alpha) = \frac{|\{S \in \mathcal{S}_n; h \text{ hashes } S \text{ } c\text{-ideally}\}|}{|\mathcal{S}_n|} = \frac{M_c}{|\mathcal{S}_n|}.$$

We suspend our analysis of the lower bound for the moment and switch to the upper bound to facilitate the comparison of the bounds.

**Lemma 2.2 (Probability Bound).** *We can bound the minimal size of a  $c$ -ideal family by*

$$H_c \leq \left\lceil \frac{|\mathcal{S}_n|}{M_c} n \ln u \right\rceil.$$

*Proof.* Let  $h$  be a hash function and  $S \in \mathcal{S}_n$  be a set of  $n$  keys. Denote by  $G_{h,S}$  the characteristic function indicating whether  $h$  is  $c$ -ideal for  $S$ :

$$G_{h,S} := \begin{cases} 1, & \text{if } \alpha_{\max} \leq c\alpha, \\ 0 & \text{otherwise.} \end{cases}$$

Let  $B_{h,S} := 1 - G_{h,S}$  denote the characteristic function indicating whether  $h$  is *not*  $c$ -ideal for  $S$ . For a subset  $S \in \mathcal{S}_n$  chosen uniformly at random, we can thus write the probability that  $h$  is  $c$ -ideal for  $S$  as follows:

$$\mathbb{P}_{S \in \mathcal{S}_n} (\alpha_{\max} \leq c\alpha) = 1 - \mathbb{P}_{S \in \mathcal{S}_n} (\alpha_{\max} > c\alpha) = 1 - \mathbb{E}_{S \in \mathcal{S}_n} [B_{h,S}]. \quad (2.6)$$

Let  $h \in \mathcal{H}_{\text{eq}}$  be any balanced hash function. Since  $S$  is chosen uniformly at random, the probability that  $h$  is  $c$ -ideal for  $S$  is the number of sets for which  $h$  is  $c$ -ideal divided by the number of sets in total:

$$\mathbb{P}_{S \in \mathcal{S}_n} (\alpha_{\max} \leq c\alpha) = \frac{M_c}{|\mathcal{S}_n|} =: p. \quad (2.7)$$

We now turn towards families of hash functions. For a family of hash functions  $\mathcal{H}$ , the product  $B_{\mathcal{H},S} := \prod_{h \in \mathcal{H}} B_{h,S}$  is equal to 1 if and only if no  $h \in \mathcal{H}$  is  $c$ -ideal for  $S$ . Let  $S \in \mathcal{S}_n$  be again any set of  $n$  keys. Let  $\mathcal{H} \subseteq \mathcal{H}_{\text{eq}}$  be a family of hash functions that consists of an arbitrary number of equidistributing hash functions chosen uniformly at random with replacement. The probability that this  $\mathcal{H}$  is not  $c$ -ideal for  $S$  is

$$\begin{aligned}
\mathbb{P}(\mathcal{H} \text{ is not } c\text{-ideal for } S) &= \mathbb{E}[B_{\mathcal{H},S}] \\
\text{\textit{h chosen independently}} &= \mathbb{E}\left[\prod_{h \in \mathcal{H}} (B_{h,S})\right] \\
\text{\textit{linearity of expectation}} &= \prod_{h \in \mathcal{H}} \mathbb{E}[B_{h,S}] \\
\text{\textit{Equation (2.6) and (2.7)}} &= \prod_{h \in \mathcal{H}} (1-p) \\
\text{\textit{independent of hash function}} &= (1-p)^{|\mathcal{H}|}.
\end{aligned}$$

The union bound yields  $|\mathcal{S}_n|(1-p)^{|\mathcal{H}|}$  as an upper bound for the probability that  $\mathcal{H}$  is not  $c$ -ideal for  $\mathcal{S}_n$ :

$$\begin{aligned}
\mathbb{P}(\mathcal{H} \text{ is not } c\text{-ideal for } \mathcal{S}_n) &= \mathbb{P}\left(\bigcup_{S \in \mathcal{S}_n} \mathcal{H} \text{ is not } c\text{-ideal for } S\right) \\
\text{\textit{union bound}} &\leq \sum_{S \in \mathcal{S}_n} \mathbb{P}(\mathcal{H} \text{ is not } c\text{-ideal for } S) \\
&= |\mathcal{S}_n|(1-p)^{|\mathcal{H}|}.
\end{aligned}$$

If this probability is less than 1, then the converse probability,  $\mathbb{P}(\alpha_{\max} \leq c\alpha)$ , is larger than 0 and we can infer that at least one  $c$ -ideal family of size  $|\mathcal{H}|$  exists. Consider an  $\mathcal{H}$  of size  $|\mathcal{H}| > -\ln |\mathcal{S}_n| / \ln(1-p)$ :

$$\begin{aligned}
|\mathcal{H}| > \frac{-\ln |\mathcal{S}_n|}{\ln(1-p)} &\iff \\
\ln\left((1-p)^{|\mathcal{H}|}\right) > -\ln |\mathcal{S}_n| &\implies \text{\textit{exponentiate}} \\
(1-p)^{|\mathcal{H}|} > \frac{1}{|\mathcal{S}_n|} &\iff \\
|\mathcal{S}_n|(1-p)^{|\mathcal{H}|} < 1 &\implies \\
\mathbb{P}(\alpha_{\max} > c\alpha) < 1.
\end{aligned}$$

Thus, if  $\mathcal{H}$  is strictly larger than the chosen value, it is  $c$ -ideal for  $\mathcal{S}_n$ . The minimal number of hash functions needed to obtain a  $c$ -ideal family of hash functions is thus at most 1 plus the chosen value:

$$H_c \leq 1 + \left\lceil \frac{-\ln |\mathcal{S}_n|}{\ln(1-p)} \right\rceil.$$

We simplify this bound with the first-order Taylor estimate for the natural logarithm, which has the expansion  $\ln(1+x) = x - x^2/2 + x^3/3 - x^4/4 + \dots$  around 1, and use  $|\mathcal{S}_n| = \binom{u}{n} \leq u^n$  to conclude the proof:

$$H_c \leq 1 + \left\lceil \frac{\ln |\mathcal{S}_n|}{-\ln(1-p)} \right\rceil \leq \left\lceil \frac{\ln |\mathcal{S}_n|}{p} \right\rceil \leq \left\lceil \frac{|\mathcal{S}_n|}{M_c} \ln |\mathcal{S}_n| \right\rceil \leq \left\lceil \frac{|\mathcal{S}_n|}{M_c} n \ln u \right\rceil. \quad \square$$

We combine [Lemma 2.1](#) and [Lemma 2.2](#) and summarize our findings:

**Corollary 2.1 (General Bounds on Family Size  $H_c$ ).** *The size of a  $c$ -ideal family of hash functions is bounded by*

$$\frac{1}{\mathbb{P}(\alpha_{\max} \leq c\alpha)} \leq H_c \leq \frac{n \ln u}{\mathbb{P}(\alpha_{\max} \leq c\alpha)}.$$

Now, to lower-bound  $H_c$ , we first consider for  $c = 1$  a straightforward application of [Lemma 2.1](#) suggested by Mehlhorn [44] and then ponder whether we could extend this approach for  $c$  larger than 1.

**Lemma 2.3.** *The number of hash functions in a 1-ideal family of hash functions is bounded from below by approximately*

$$\approx \frac{\sqrt{2\pi\alpha}^{m-1}}{\sqrt{m}}.$$

*Proof.*  $M_1$ , the number of sets that a single hash function can map 1-ideal is attained by balanced hash functions. A balanced hash function divides the universe into  $m$  equal parts of size  $u/m$ . Exactly the sets that consist of  $\alpha = n/m$  keys from each of the  $m$  parts of the universe are mapped 1-ideally. There are exactly

$$M_1 = \binom{u/m}{\alpha}^m. \quad (2.8)$$

such sets. We write again  $\beta := u/m$  and use the volume bound to estimate  $H_1$  as follows:

$$\begin{aligned} \text{Lemma 2.1} \quad H_1 &\geq \frac{\mathcal{S}_n}{M_1} \\ \text{Equation (2.8)} \quad &= \frac{\binom{u}{n}}{\binom{\beta}{\alpha}^m} \\ \text{Definition of binomial} \quad &= \frac{u!(\beta - \alpha)!^m \alpha!^m}{(u - n)! n! \beta!^m} \end{aligned}$$

These factorials can be estimated with Stirling's formula. We are going to use the following estimates by Robbins [53]:

$$\sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\frac{1}{12n+1}} \leq n! \leq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}.$$

We insert these estimations for the factorials and then reorder the terms repeatedly to simplify the expression:

$$\begin{aligned} &\geq \frac{\sqrt{2\pi u} \sqrt{2\pi(\beta - \alpha)}^m \sqrt{2\pi\alpha}^m}{\sqrt{2\pi(u - n)} \sqrt{2\pi n} \sqrt{2\pi\beta}^m} \cdot \frac{\left(\frac{u}{e}\right)^u \left(\frac{\beta - \alpha}{e}\right)^{m(\beta - \alpha)} \left(\frac{\alpha}{e}\right)^{m\alpha}}{\left(\frac{u - n}{e}\right)^{(u - n)} \left(\frac{n}{e}\right)^n \left(\frac{\beta}{e}\right)^{m\beta}} \\ &\quad \cdot \frac{e^{\frac{1}{12u+1}} e^{\frac{1}{12(\beta - \alpha)+1}} e^{\frac{1}{12\alpha+1}}}{e^{\frac{1}{12(u - n)}} e^{\frac{1}{12n}} e^{\frac{1}{12\beta}}} \end{aligned}$$

$$\begin{aligned}
&= \sqrt{2\pi}^{-2m+1-(2+m)} \sqrt{\frac{u(\beta-\alpha)^m \alpha^m}{(u-n)n\beta^m}} \cdot \frac{\left(\frac{u}{e}\right)^u \left(\frac{\beta-\alpha}{e}\right)^{u-n} \left(\frac{\alpha}{e}\right)^n}{\left(\frac{u-n}{e}\right)^{u-n} \left(\frac{n}{e}\right)^n \left(\frac{\beta}{e}\right)^u} \\
&\quad \cdot e^{\frac{1}{12u+1} + \frac{1}{12(\beta-\alpha)+1} + \frac{1}{12\alpha+1} - \frac{1}{12(u-n)} - \frac{1}{12n} - \frac{1}{12\beta}}.
\end{aligned}$$

We pause for a moment and abbreviate  $A := e^{\frac{1}{12u+1} + \frac{1}{12(\beta-\alpha)+1} + \frac{1}{12\alpha+1} - \frac{1}{12(u-n)} - \frac{1}{12n} - \frac{1}{12\beta}}$  before we continue our calculation:

$$\begin{aligned}
\beta = u/m &= \sqrt{2\pi}^{-m-1} \sqrt{\frac{u(u-n)^m \alpha^m}{(u-n)nu^m}} \cdot \frac{u^u (\beta-\alpha)^{u-n} \alpha^n}{(u-n)^{u-n} n^n \beta^u} \cdot A \\
&= \sqrt{2\pi}^{-m-1} \sqrt{\frac{\left(\frac{u-n}{u}\right)^{m-1} \alpha^m}{n}} \cdot \frac{u^u (u-n)^{u-n} n^n}{(u-n)^{u-n} n^n u^u} \cdot A \\
\frac{u^u (u-n)^{u-n} n^n}{(u-n)^{u-n} n^n u^u} = 1 &= \sqrt{2\pi}^{-m-1} \sqrt{\left(\frac{u-n}{u}\right)^{m-1}} \sqrt{\frac{\alpha}{n}} \cdot A \\
\alpha = n/m &= \sqrt{2\pi}^{-m-1} \sqrt{\left(1 - \frac{n}{u}\right)^{m-1}} \sqrt{\frac{1}{m}} \cdot A.
\end{aligned}$$

The factor  $A$  is smaller than  $e$  and tends to 1 with  $\alpha \rightarrow \infty$ . Hence, this is approximately

$$\begin{aligned}
&\approx \sqrt{2\pi}^{-m-1} \sqrt{\left(1 - \frac{n}{u}\right)^{m-1}} \sqrt{\frac{1}{m}} \\
&\approx \frac{1}{\sqrt{m}} \sqrt{2\pi}^{-m-1}. \quad \square
\end{aligned}$$

The natural extension of this approach yields  $H_c \geq K(n, m, c\alpha) \binom{u}{n} / \binom{u/m}{c\alpha}^m$ , where  $K(n, m, c\alpha)$  denotes the number of compositions of  $n$  into  $m$  non-negative integers between 0 and  $c\alpha$ . This factor, which accounts for the number of possibilities to split the  $n$  keys into  $m$  different cells, equals 1 for  $c = 1$ . For general  $c$ , however, to the best of our knowledge, even the strongest approximations for  $K(n, m, c\alpha)$  do not yield a meaningful lower bound. Therefore, we are forced to use a different strategy for general  $c$  and we estimate  $M_c$  via the probability  $\mathbb{P}(\alpha_{\max} \leq c\alpha)$  in the following section.

## 2.4 Estimations for $\mathbb{P}(\alpha_{\max} \leq c\alpha)$

It remains to find good bounds on the probability  $\mathbb{P}(\alpha_{\max} \leq c\alpha)$ , which will immediately yield the desired bounds on  $H_c$ . We start by establishing an upper bound on  $\mathbb{P}(\alpha_{\max} \leq c\alpha)$ .

### Upper Bound

Recall that we fixed a balanced hash function  $h \in \mathcal{H}_{eq}$  and draw an  $S \in \mathcal{S}_n$  uniformly at random. For every cell  $k \in T$ , we model its load  $\alpha_k$  as a random variable. The joint probability distribution of the  $\alpha_i$  follows the hypergeometric distribution, i.e.

$$\mathbb{P}((\alpha_1, \dots, \alpha_m) = (\ell_1, \dots, \ell_m)) = \frac{\binom{|h^{-1}(1)|}{\ell_1} \dots \binom{|h^{-1}(m)|}{\ell_m}}{\binom{u}{n}}.$$

There are two obstacles to overcome. First, calculating with probabilities without replacement is difficult. Second, the sum of the  $\alpha_i$  is required to be  $n$ ; in particular, the variables are not independent. The first obstacle can be overcome by considering drawing elements from  $U$  with replacement instead, i.e. drawing multisets instead of sets. As the following lemma shows, we do not lose much by this assumption.

**Lemma 2.4 (Switching to Drawing With Replacement).** *For any fixed  $\varepsilon > 0$ , there are  $u$  and  $n$  large enough such that*

$$\frac{M_c}{|\mathcal{S}_n|} \leq (1 + \varepsilon)\mathbb{P}(T_i \leq c\alpha, i \in [m]),$$

where we use  $T_1, \dots, T_m$  to model the cell loads as binomially distributed random variables with parameters  $n$  and  $1/m$ . In other words, for  $u$  and  $n$  large enough, we can consider drawing the elements of  $S \in \mathcal{S}_n$  with replacement and only lose a negligible factor.

*Proof.* We use  $\mathcal{K}_n$  to denote the set of multisets with elements from  $U$  of cardinality  $n$ . There are  $\binom{u+n-1}{n}$  such multisets.<sup>4</sup> We write  $K_c$  for the number of those multisets of  $\mathcal{K}_n$  that are hashed  $c$ -ideally by  $h$ . Note that  $\mathbb{P}(T_i \leq c\alpha, i \in [m])$  equals  $K_c/|\mathcal{K}_n|$ . Clearly,  $M_c$  is less than or equal to  $K_c$  since every true set is a multiset. It follows that  $M_c/|\mathcal{S}_n| \leq K_c/|\mathcal{S}_n|$ . To conclude, we multiply the right-hand side with  $|\mathcal{S}_n|/|\mathcal{K}_n|$  and observe that

$$\frac{|\mathcal{S}_n|}{|\mathcal{K}_n|} = \frac{\binom{u}{n}}{\binom{u+n-1}{n}} = \prod_{k=0}^{n-1} \frac{(u-k)}{(u+n-1-k)} \xrightarrow[n/u \rightarrow 0]{u \rightarrow \infty} 1. \quad \square$$

If we consider drawing elements from  $U$  with replacement, we have a multinomial distribution, i.e.

$$\mathbb{P}((T_1, \dots, T_m) = (\ell_1, \dots, \ell_m)) = \binom{n}{\ell_1, \dots, \ell_m} / m^n.$$

This is easier to handle than the hypergeometric distribution. The  $T_i$ ,  $i \in [m]$ , are still not independent, however. The strong methods by Dubhashi and Ranjan [22] provide us with a simple way to overcome this second obstacle:

**Lemma 2.5 (Proposition 28 and Theorem 31 from [22]).** *The joint distribution of the random variables  $\alpha_i$  for  $i \in [m]$  is upper-bounded by their product:*

$$\mathbb{P}(\alpha_i \leq c\alpha, i \in [m]) \leq \prod_{i=1}^m \mathbb{P}(\alpha_i \leq c\alpha).$$

*The same holds if we consider the  $T_i$ ,  $i \in [m]$ , instead of the  $\alpha_i$ .*

We are ready to give an upper bound on the probability that a hash function family is  $c$ -ideal.

<sup>4</sup>Maybe the best way to see this is to observe the following bijection between multisets of size  $n$  with elements from a set of size  $u$  and normal sets of size  $n$  consisting of elements of a set of size  $u+n-1$ : Write the elements  $x_1, \dots, x_n$  of the multiset in non-decreasing order and then map this multiset to the set  $\{x_1, x_2+1, \dots, x_n+n-1\}$ .

**Theorem 2.2 (Upper Bound on  $\mathbb{P}(\alpha_{\max} \leq c\alpha)$ ).** For arbitrary  $\varepsilon > 0$ ,

$$\mathbb{P}(\alpha_{\max} \leq c\alpha) \leq (1 + \varepsilon) \exp\left(-me^{-\alpha}(1 - \varepsilon)\left(\frac{\alpha}{c\alpha + 1}\right)^{c\alpha + 1}\right),$$

where  $n$  tends to infinity.

*Proof.* We can bound the probability that a balanced hash function is  $c$ -ideal as follows.

$$\begin{aligned} &= \mathbb{P}(\alpha_{\max} \leq c\alpha) \\ \text{Definition of maximum} &= \mathbb{P}(\alpha_i \leq c\alpha, i \in [m]) \\ \text{Switching to binomial variables, Lemma 2.4} &\leq (1 + \varepsilon)\mathbb{P}(T_i \leq c\alpha, i \in [m]) \\ \text{Lemma 2.5} &\leq (1 + \varepsilon) \prod_{i=1}^m \mathbb{P}(T_i \leq c\alpha) \\ T_i, i \in [m], \text{ are identically distributed} &= (1 + \varepsilon)\mathbb{P}(T_1 \leq c\alpha)^m \\ \text{Converse probability} &= (1 + \varepsilon)(1 - \mathbb{P}(T_1 > c\alpha))^m \\ e^{-x} \geq 1 - x \text{ for } x \in (0, 1) &\leq (1 + \varepsilon)e^{-m\mathbb{P}(T_1 > c\alpha)}. \end{aligned}$$

To continue our estimation, we need to bound  $\mathbb{P}(T_1 > c\alpha)$  from below. We start with the definition of the binomial distribution, take only the first summand and use the standard bound  $\binom{n}{k} \geq (n/k)^k$  on the binomial coefficient to obtain the following.

$$\begin{aligned} \mathbb{P}(T_1 > c\alpha) &= \sum_{k=c\alpha+1}^n \binom{n}{k} \left(\frac{1}{m}\right)^k \left(1 - \frac{1}{m}\right)^{n-k} \\ \text{Rearrange} &= \left(1 - \frac{1}{m}\right)^n \sum_{k=c\alpha+1}^n \binom{n}{k} \left(\frac{1}{m-1}\right)^k \\ \frac{n}{m} = \alpha \text{ and } \binom{n}{k} \geq \left(\frac{n}{k}\right)^k &\geq \left(1 - \frac{\alpha}{n}\right)^n \sum_{k=c\alpha+1}^n \left(\frac{n}{k}\right)^k \left(\frac{1}{m-1}\right)^k \\ \text{Rearrange} &= \left(1 - \frac{\alpha}{n}\right)^n \sum_{k=c\alpha+1}^n \left(\frac{1}{k}\right)^k \left(\frac{n}{m-1}\right)^k \\ \frac{n}{m-1} = \frac{n}{m} + \frac{n}{m(m-1)} \geq \frac{n}{m} = \alpha &\geq \left(1 - \frac{\alpha}{n}\right)^n \sum_{k=c\alpha+1}^n \left(\frac{1}{k}\right)^k \alpha^k \\ \text{Take only } k = c\alpha + 1 &= \left(1 - \frac{\alpha}{n}\right)^n \left(\frac{\alpha}{c\alpha + 1}\right)^{c\alpha + 1} \\ \left(1 - \frac{\alpha}{n}\right)^n \rightarrow e^{-\alpha} \text{ for } n \text{ large enough} &\geq (1 - \varepsilon')e^{-\alpha} \left(\frac{\alpha}{c\alpha + 1}\right)^{c\alpha + 1}. \end{aligned}$$

If we combine the two estimates and choose  $\varepsilon'$  appropriately, we achieve the desired result.  $\square$

**Corollary 2.1** translates the upper bound from [Theorem 2.2](#) into a lower bound on  $H_c$ .



**Theorem 2.3 (Lower Bound on Family Size  $H_c$ ).** *For arbitrary  $\varepsilon > 0$ , the number of hash functions in a  $c$ -ideal family of hash functions is bounded from below by*

$$H_c \geq \frac{1}{\mathbb{P}(\alpha_{\max} \leq c\alpha)} \geq (1 - \varepsilon) \exp\left(\frac{m}{e^\alpha} (1 - \varepsilon) \left(\frac{\alpha}{c\alpha + 1}\right)^{c\alpha + 1}\right).$$

### Lower Bound

Another way to overcome the obstacle that the variables are not independent would have been to apply a customized Poissonization technique. The main monograph presenting this technique is in the book by Barbour et al. [1]; the textbook by Mitzenmacher and Upfal [48] gives a good illustrating example. A series of arguments would have allowed us to bound the precision loss we incur by a constant factor of 2, leading to

$$\mathbb{P}(T_{\max} \leq c\alpha) \leq 2\mathbb{P}(Y \leq c\alpha)^m,$$

where we use  $T_{\max}$  to denote  $\max\{T_1, \dots, T_m\}$  and where  $Y$  is a Poisson random variable with mean  $\alpha$ . By using Lemma 2.5, we were able to abbreviate this approach. However, part of this Poissonization technique can be used for the lower bound on  $\mathbb{P}(\alpha_{\max} \leq c\alpha)$ .

Recall that a random variable  $X$  following the Poisson distribution—commonly written as  $X \sim P_\lambda$  and referred to as a Poisson variable—takes on the value  $k$  with probability  $\mathbb{P}(X = k) = \frac{1}{e^\lambda} \frac{\lambda^k}{k!}$ . The following lemma is the counterpart to Lemma 2.4.

**Lemma 2.6.** *We can bound  $M_c/|\mathcal{S}_n| = \mathbb{P}(\alpha_{\max} \leq c\alpha)$  from below by  $\mathbb{P}(T_{\max} \leq c\alpha)$ .*

*Proof.* Fix an arbitrary  $S_0 \in \mathcal{S}_n$  and choose an  $h \in \mathcal{H}_{\text{all}}$  uniformly at random. Then each of the  $n$  keys in  $S_0$ , hashed one after another, has the same probability of  $1/m$  to be hashed to a given cell  $k$ . Note that this is equivalent to the definition of the  $T_k$ . Moreover, let  $\chi(h, S)$  denote the characteristic function of  $c$ -ideality, which is 1 if  $\max_{k \in \{1, \dots, m\}} \{|h^{-1}(k) \cap S|\} \leq c\alpha$  and 0 otherwise.

$$\begin{aligned} & \mathbb{P}(T_{\max} \leq c\alpha) \\ \text{Definitions of } \chi \text{ and } T_k &= \mathbb{P}_{h \in \mathcal{H}_{\text{all}}} (\chi(h, S_0) = 1) \\ \text{Definition of } \mathbb{E} &= \mathbb{E}_{h \in \mathcal{H}_{\text{all}}} [\chi(h, S_0)] \\ \text{Valid for arbitrary } S_0 \in \mathcal{S}_n &= \mathbb{E}_{S \in \mathcal{S}_n} \left[ \mathbb{E}_{h \in \mathcal{H}_{\text{all}}} [\chi(h, S)] \right] \\ \text{Definition of } \mathbb{E} &= \sum_{S \in \mathcal{S}_n} \mathbb{P}(S) \sum_{h \in \mathcal{H}_{\text{all}}} \mathbb{P}(h) \cdot \chi(h, S) \\ \text{Swap sums} &= \sum_{h \in \mathcal{H}_{\text{all}}} \mathbb{P}(h) \sum_{S \in \mathcal{S}_n} \mathbb{P}(S) \cdot \chi(h, S) \\ \text{Definition of } \mathbb{E} &= \mathbb{E}_{h \in \mathcal{H}_{\text{all}}} \left[ \mathbb{E}_{S \in \mathcal{S}_n} [\chi(h, S)] \right] \end{aligned}$$

For every  $h \in \mathcal{H}_{\text{all}}$ , we have  $\sum_{S \in \mathcal{S}_n} \chi(h, S) \leq M_c$  by the definition of  $M_c$ , whence

$$\mathbb{E}_{S \in \mathcal{S}_n} [\chi(h, S)] = \mathbb{P}_{S \in \mathcal{S}_n} (\chi(h, S) = 1) = \frac{\sum_{S \in \mathcal{S}_n} \chi(h, S)}{\sum_{S \in \mathcal{S}_n} 1} \leq \frac{M_c}{|\mathcal{S}_n|}.$$

In particular, the inequality still holds true for the expected value over all  $h \in \mathcal{H}_{\text{all}}$ . Together with the equality derived above we conclude

$$\mathbb{P}(T_{\max} \leq c\alpha) = \mathbb{E}_{h \in \mathcal{H}_{\text{all}}} \left[ \mathbb{E}_{S \in \mathcal{S}_n} [\chi(h, S)] \right] \leq \frac{M_c}{|\mathcal{S}_n|}. \quad \square$$

For the counterpart to [Theorem 2.2](#), we use Poissonization to turn the binomial variables into Poisson variables.

The next lemma shows that  $(T_1, \dots, T_m)$  has the same probability mass function as  $(Y_1, \dots, Y_m)$  under the condition  $Y = n$ .

**Lemma 2.7 (Sum Conditioned Poisson Variables).** *For any natural numbers  $\ell_1, \dots, \ell_m$  with  $\ell_1 + \dots + \ell_m = n$ , we have that*

$$\mathbb{P}((Y_1, \dots, Y_m) = (\ell_1, \dots, \ell_m) \mid Y = n) = \mathbb{P}((T_1, \dots, T_m) = (\ell_1, \dots, \ell_m)).$$

*Proof.*

$$\begin{aligned} \mathbb{P}((Y_1, \dots, Y_m) = (\ell_1, \dots, \ell_m) \mid Y = n) &= \frac{\mathbb{P}(Y_1 = \ell_1) \cdots \mathbb{P}(Y_m = \ell_m)}{\mathbb{P}(Y = n)} \\ &\stackrel{Y_1, \dots, Y_m \sim P_\alpha \text{ and } Y \sim P_n}{=} \frac{\frac{1}{e^\alpha} \frac{\alpha^{\ell_1}}{\ell_1!} \cdots \frac{1}{e^\alpha} \frac{\alpha^{\ell_m}}{\ell_m!}}{\frac{1}{e^n} \frac{n^n}{n!}} \\ &\stackrel{\text{Cancel using } \alpha m = n}{=} \frac{\alpha^{\ell_1 + \dots + \ell_m}}{\ell_1! \cdots \ell_m!} \frac{n^n}{n!} \\ &\stackrel{\text{Eliminate double fraction}}{=} \frac{\alpha^{\ell_1 + \dots + \ell_m} \cdot n!}{\ell_1! \cdots \ell_m! \cdot n^n} \\ &\stackrel{\ell_1 + \dots + \ell_m = n \text{ and } \frac{n}{\alpha} = m}{=} \frac{n!}{\ell_1! \cdots \ell_m! \cdot m^n} \\ &\stackrel{\text{Multinomial } \binom{n}{\ell_1, \dots, \ell_m} := \frac{n!}{\ell_1! \cdots \ell_m!}}{=} \frac{\binom{n}{\ell_1, \dots, \ell_m}}{m^n} \\ &\stackrel{\text{Definition of multinomial distribution}}{=} \mathbb{P}((T_1, \dots, T_m) = (\ell_1, \dots, \ell_m)) \quad \square \end{aligned}$$

The result of [Lemma 2.7](#) immediately carries over to the conditioned expected value for any real function  $f(\ell_1, \dots, \ell_m)$ , by the definition of expected values.

**Corollary 2.2 (Conditioned Expected Value).** *Let  $f(\ell_1, \dots, \ell_m)$  be any real function. We have  $\mathbb{E}[f(Y_1, \dots, Y_m) \mid Y = n] = \mathbb{E}[f(T_1, \dots, T_m)]$ .*

We are ready to state the counterpart to [Theorem 2.2](#).

**Lemma 2.8 (Lower Bound on Non-Excess Probability).** *Set  $d = c\alpha$ . We have*

$$\mathbb{P}(T_{\max} \leq d) \geq \frac{\sqrt{2\pi n}}{(2\pi d)^{m/(2c)}} \frac{1}{c^n} \frac{1}{e^{\frac{m}{12cd}}} (\alpha + 1)^{m(1 - \frac{1}{c})}.$$

*Proof.* We only sketch the proof. Using [Corollary 2.2](#), we can formulate this probability with Poisson variables:

$$\begin{aligned} \mathbb{P}(T_{\max} \leq d) &= \mathbb{E}[\chi_{T_{\max} \leq d}] \\ \text{Corollary 2.2} \quad &= \mathbb{E}[\chi_{\max\{Y_1, \dots, Y_m\} \leq d} \mid Y = n] \\ \text{Definition of conditional probability} \quad &= \frac{\mathbb{E}[\chi_{\{Y=n\}} \chi_{\max\{Y_1, \dots, Y_m\} \leq d}]}{\mathbb{P}(Y = n)}. \end{aligned}$$

We know that  $Y \sim P_n$ ; hence, the denominator equals  $n^n / (n!e^n)$ . For the numerator, we use the fact that  $Y_i \sim P_\alpha$  for all  $i$  to write:

$$\mathbb{E}[\chi_{\{Y=n\}} \chi_{\max\{Y_1, \dots, Y_m\} \leq d}] = \sum_{\substack{\ell_1, \dots, \ell_m \in \mathbb{N} \\ \ell_1 + \dots + \ell_m = n \\ \ell_1, \dots, \ell_m \leq d}} \prod_{i=1}^m \frac{1}{e^\alpha} \frac{\alpha^{\ell_i}}{\ell_i!}$$

We estimate the sum by finding a lower bound for both the number of summands and the products that constitute the summands. The product attains its minimum when the  $\ell_i$  are distributed as asymmetrically as possible, that is, if almost all  $\ell_i$  are set to either  $d$  or  $0$ , with only one being  $(n \bmod d)$ . This fact is an extension of the simple observation that for any natural number  $n$ , we have

$$n!n! = n(n-1)(n-2) \cdot \dots \cdot 1 \cdot n! < 2n(2n-1)(2n-2) \cdot \dots \cdot (n+1) \cdot n! = (2n)!$$

A rigorous proof is obtained by extending the range of the expression to the real numbers, taking the derivative, and analyzing the extremal values. As this is quite technical and uses ideas from advanced analysis that are not needed otherwise, we merely state the fact as a lemma and defer the proof to [Appendix A](#).

**Lemma 2.9.** *We have*

$$\min_{\Omega} \left\{ \prod_{i=1}^m \frac{1}{\ell_i!} \right\} = 1/(d!)^{m/c},$$

where  $\Omega := \{(\ell_1, \dots, \ell_m) \in \mathbb{R}^m; \ell_1 + \dots + \ell_m = n \wedge 0 \leq \ell_1, \dots, \ell_m \leq d\}$ .

The next step is to find a lower bound on the number of summands, that is, the number of integer compositions  $\ell_1 + \dots + \ell_m = n$ . As we mentioned at the very end of [Section 2.3](#), we are not aware of strong approximation for this number and the following estimation is very crude for many values of  $n$ ,  $m$ , and  $c$ : If we let the first  $m - \frac{m}{c}$  integers,  $\ell_1, \dots, \ell_{\frac{m}{c}}$ , range freely between  $0$  and  $\alpha$ , we can always ensure that the condition  $\ell_1 + \dots + \ell_m = n$  is satisfied by choosing the remaining  $\frac{m}{c}$  summands  $\ell_{\frac{m}{c}+1}, \dots, \ell_m$  appropriately. Therefore, the number of summands is at least  $(\alpha + 1)^{m(1-\frac{1}{c})}$ . Together with the previous calculation, this leads to

$$\mathbb{E}[\chi_{\{Y=n\}} \chi_{\max\{Y_1, \dots, Y_m\} \leq d}] \geq \frac{\alpha^n (\alpha + 1)^{m(1-\frac{1}{c})}}{e^n (d!)^{m/c}}.$$

Putting everything together and using the Stirling bounds [\[53\]](#) for the factorials, we obtain the desired result.  $\square$

Combining the results in this section in the straightforward way yields the following upper bound on  $H_c$ .

**Theorem 2.4 (Upper Bound on Minimal Family Size  $H_c$ ).** *The minimal number of hash functions in a  $c$ -ideal family of hash functions is bounded from above by*

$$H_c \leq \left\lceil \left( \sqrt{2\pi c\alpha}^{1/c} c^\alpha \frac{e^{\frac{1}{12c^2\alpha}}}{(\alpha+1)^{1-\frac{1}{c}}} \right)^m \sqrt{\frac{n}{2\pi}} \ln u \right\rceil.$$

*Proof.* We start with the probability bound from [Lemma 2.2](#) and use [Lemma 2.6](#) and [Lemma 2.8](#).

$$\begin{aligned} H_c &\leq \left\lceil \frac{|\mathcal{S}_n|}{M_c} n \ln u \right\rceil \\ \text{Lemma 2.6} &\leq \left\lceil \frac{1}{\mathbb{P}(T_{\max} \leq c\alpha)} n \ln u \right\rceil \\ \text{Lemma 2.8 and } d = c\alpha &\leq \left\lceil \frac{(2\pi c\alpha)^{m/(2c)}}{\sqrt{2\pi n}} c^n \frac{e^{\frac{m}{12c^2\alpha}}}{(\alpha+1)^{m(1-\frac{1}{c})}} n \ln u \right\rceil \\ n = m\alpha &= \left\lceil \left( (2\pi c\alpha)^{1/(2c)} (c^\alpha) \frac{e^{\frac{1}{12c^2\alpha}}}{(\alpha+1)^{1-\frac{1}{c}}} \right)^m \sqrt{\frac{n}{2\pi}} \ln u \right\rceil \end{aligned}$$

□

[Theorem 2.4](#) states that for constant  $\alpha$ , the number of functions such that for each set of keys of size  $n$ , there is a function that distributes this set among the hash table cells at least as good as  $c$  times an optimal solution is bounded from above by a number that grows exponentially in  $m$ , but only with the square root in  $n$  and only logarithmically with the universe size. However, our bounds do not match, hence the exact behavior of  $H_c$  within the given bounds remains obscure. It becomes easier if we analyze the advice complexity, using the connection between  $c$ -ideality and advice complexity described in the introduction of this chapter. We first improve our bounds on  $H_c$  for some edge cases in the next section before we then use and interpret the results in [Section 2.6](#).

## 2.5 Improvements For Edge Cases

As discussed in the introduction, the size  $u$  of the universe does not appear in the lower bound on  $H_c$  of [Corollary 2.1](#). The following lower bound on  $H_c$ , which is a straightforward generalization of an argument presented in the classical textbook by Mehlhorn [\[44\]](#), features  $u$  in a meaningful way.

**Theorem 2.5.** *We have that*

$$H_c \geq \frac{\ln(u) - \ln(c\alpha)}{\ln(m)}.$$

*Proof.* Denote the hash functions in the given family  $\mathcal{H}$  by  $h_1, \dots, h_{|\mathcal{H}|}$ , in an arbitrary order. Consider  $h_1 : U \rightarrow T$ . Since it splits up  $u$  keys among  $m$  cells, there exists

a cell  $k_1 \in T$  with at least  $\frac{u}{m}$  keys, that is,  $|h_1^{-1}(k)| \geq \frac{u}{m}$ . In other words, at least  $\frac{u}{m}$  of the keys are indistinguishable under the hash function  $h_1$ . Consider now  $h_2 : U \rightarrow T$  and its behavior on the keys from  $h_1^{-1}(k)$ . By the same argument as before,  $h_2$  cannot do better but to split them up evenly among the  $m$  cells, so there exists a cell  $k_2 \in T$  that is allotted at least an  $m$ th of these  $|h_1^{-1}(k)|$  keys. Hence,  $|h_1^{-1}(k_1) \cap h_2^{-1}(k_2)| \geq \frac{u}{m^2}$ . Iterating this argument through all  $|\mathcal{H}|$  functions in the given family, there is a set of at least

$$d := \left| \bigcap_{i=1}^{|\mathcal{H}|} h_i^{-1}(k_i) \right| \geq \frac{u}{m^{|\mathcal{H}|}}$$

keys in the universe that are indistinguishable under all functions in  $\mathcal{H}$ . By definition,  $\mathcal{H}$  can only be  $c$ -ideal if  $d \leq c\alpha$ . By transforming the equation above, this leads to

$$|\mathcal{H}| \geq \frac{\ln(u) - \ln(c\alpha)}{\ln(m)}. \quad \square$$

This demonstrates that, while it is easy to find bounds that include the size of the universe, it seems to be very difficult to incorporate  $u$  into a general bounding technique that does not take it into account naturally, such as the first inequality of [Corollary 2.1](#). However, it is not too difficult to obtain bounds that improve upon [Corollary 2.1](#) for large—possibly less interesting—values of  $c$ .

**Theorem 2.6 (Yao Bound).** *For every  $c \in \omega\left(t \frac{\ln n}{\ln \ln n}\right)$ , where  $t \geq 1$ , we have  $H_c \in \mathcal{O}(\ln |\mathcal{S}_n| / \ln t)$ . In particular, we obtain that  $H_c \in \mathcal{O}(n \ln u)$  for  $t \in \mathcal{O}(1)$ .*

*Proof.* We apply the Yao-inspired principle described by Komm [39], which can be summarized as follows: Suppose that a hash function  $h \in \mathcal{H}_{\text{all}}$  is chosen uniformly at random. The expected maximum cell load is  $\mu := \mathbb{E}[\alpha_{\max}] \in \Theta\left(\frac{\ln n}{\ln \ln n}\right)$ , as proven in [52]. The Markov inequality guarantees that at most a fraction  $1/t$  of inputs result in a cell load of  $t\mu$  or more:

$$\mathbb{P}_{S \in \mathcal{S}_n} [\alpha_{\max} \geq t\mu] \leq \frac{1}{t}.$$

Therefore, we can find a single hash function  $h$  that guarantees a maximal load of at most  $t\mu$  for all but a fraction of  $1/t$  of all input sets  $S \in \mathcal{S}_n$  simultaneously. Now we can repeat the argument as many times as we desire and will find every time a hash function that causes a maximal load of  $t\mu$  or more for at most a fraction  $1/t$  of inputs for which the previous hash functions did so. After  $r$  iterations, there are only  $|\mathcal{S}_n|t^{-r}$  input sets left that cause a maximal load of at least  $t\mu$ . Seeing that

$$|\mathcal{S}_n| \left(\frac{1}{t}\right)^r < 1 \iff |\mathcal{S}_n| < t^r \iff \frac{\ln |\mathcal{S}_n|}{\ln t} < r,$$

we can conclude that there is a subfamily of size at most  $\lceil \ln |\mathcal{S}_n| / \ln t \rceil + 1$  that contains, for every subset  $S \in \mathcal{S}_n$ , at least one hash function that incurs cell loads of at most  $t\mu$ .  $\square$

## 2.6 Advice Complexity of Hashing

With the conceptualization of hashing as an ultimate online problem mentioned in the introduction of this chapter, we can use the bounds on  $H_c$  to provide bounds on the advice complexity of  $c$ -competitive algorithms. [Theorem 2.5](#) immediately yields the following theorem.

**Theorem 2.7.** *Every ALG for hashing with less than  $\ln(\ln(u) - \ln(c\alpha)) - \ln(\ln(m))$  advice bits cannot achieve a lower cost than  $\text{cost}(\text{ALG}) = c\alpha$  and is thus not better than  $c$ -competitive. In other words, there exists an  $S \subseteq U$  such that the output  $h : U \rightarrow T$  of ALG maps at least  $c\alpha$  elements of  $S$  to one cell.*

With a lower bound on the size of  $c$ -ideal families of hash functions, this bound can be improved significantly, as the following theorem shows.

**Theorem 2.8.** *Every ALG for hashing needs at least*

$$\log\left((1 - \varepsilon) \exp\left(\frac{m}{e^\alpha} (1 - \varepsilon) \left(\frac{\alpha}{c\alpha + 1}\right)^{c\alpha + 1}\right)\right)$$

*advice bits in order to be  $c$ -competitive, for an arbitrary fixed  $\varepsilon > 0$ .*

We want to determine the asymptotic behavior of this bound and hence analyze the term  $(\alpha/(c\alpha + 1))^{c\alpha + 1}$ . We are going to use the fact that  $\lim_{n \rightarrow \infty} (1 - 1/n)^n = 1/e$ .

$$\begin{aligned} \left(\frac{\alpha}{c\alpha + 1}\right)^{c\alpha + 1} &= \left(\frac{\frac{1}{c}(c\alpha + 1 - 1)}{c\alpha + 1}\right)^{c\alpha + 1} \\ &= \left(\frac{1}{c} \left(1 - \frac{1}{c\alpha + 1}\right)\right)^{c\alpha + 1} \\ \text{for any } \varepsilon' > 0 \text{ and } c\alpha + 1 \text{ large enough} &\geq \left(\frac{1}{c}\right)^{c\alpha + 1} (1 - \varepsilon') \frac{1}{e} \end{aligned}$$

Therefore, the bound from [Theorem 2.8](#) is in

$$\Omega\left(\frac{m}{e^\alpha} \left(\frac{1}{c}\right)^{c\alpha + 1}\right).$$

Before we interpret this result, we turn to the upper bounds. [Theorem 2.4](#) yields the following result.

**Theorem 2.9.** *There is a  $c$ -competitive algorithm with advice that reads*

$$\begin{aligned} &\log\left[\left(\sqrt{2\pi c\alpha}^{-1/c} c^\alpha \frac{e^{\frac{1}{12c^2\alpha}}}{(\alpha + 1)^{1 - \frac{1}{c}}}\right)^m \sqrt{\frac{n}{2\pi}} \ln u\right] \\ &\in \mathcal{O}\left(\ln \ln u + \ln n + m\left(\frac{1}{2c} \log(2\pi c\alpha) + \alpha \log(c) + \frac{1}{12c^2\alpha} - \left(1 - \frac{1}{c}\right) \log(\alpha + 1)\right)\right) \end{aligned}$$

*advice bits.*

The factor after  $m$  is minimal for  $c = \alpha = 1$ ; this minimal value is larger than 1.002. This upper bound is therefore always at least linear in  $m$ . For  $c = \omega\left(\frac{\ln(n)}{\ln(\ln(n))}\right)$ , we can improve on this and remove the last summand completely, based on [Theorem 2.6](#).

**Theorem 2.10.** *For  $c \in \omega\left(t\frac{\ln n}{\ln \ln n}\right)$ ,  $t > 1$ , there exists a  $c$ -competitive algorithm that reads  $\mathcal{O}\left(\log\left(\frac{\ln |S_n|}{\ln t}\right)\right)$  many advice bits. In particular, for  $t \in \mathcal{O}(1)$ , there exists an  $\frac{\ln n}{\ln \ln n}$ -competitive algorithm that reads  $\mathcal{O}(\ln \ln u + \ln n)$  many advice bits.*

[Theorem 2.8](#) and [Theorem 2.9](#) reveal the advice complexity of hashing to be linear in the hash table size  $m$ . While the universe size  $u$  still appears in the upper bound, it functions merely as a summand and is mitigated by a double logarithm. Unless the key length  $\log_2(u)$  is exponentially larger than the hash table size  $m$ , the universe size cannot significantly affect this general behavior. The more immediate bounds for the edge cases do not reveal this dominance of the hash table over the universe. Moreover, changing the two parameters  $\alpha$  and  $c$  has no discernible effect on the edge case bounds despite the exponential influence on the main bounds.

## 2.7 Conclusion

This chapter analyzed hashing from an unusual angle by regarding hashing as an online problem and then studying its advice complexity. Online problems are usually studied with competitive analysis, which is a worst-case measurement. As outlined in the introduction, it is impossible to prevent a deterministic algorithm from incurring the worst-case cost by hashing all appearing keys into a single cell of the hash table. Therefore, randomized algorithms are key to the theory and application of hashing. In particular the surprising discovery of small universal hashing families gave rise to efficient algorithms with excellent expected cost behavior. However, from a worst-case perspective, the performance of randomized algorithms is lacking.

This motivated the conceptualization of  $c$ -ideal hashing families as a generalization of perfect  $k$ -hashing families to the case where  $\alpha > 1$ . Our goal was to analyze the trade-off between size and ideality of hashing families since this is directly linked to the competitiveness of online algorithms with advice. Our bounds generalize results by Fredman and Komlós [\[24\]](#) as well as Naor et al. [\[51\]](#) to the case  $\alpha > 1$  and  $c \geq 1$ .

As a first step, we proved that balanced hash functions are suited best for hashing in the sense that they maximize the number of subsets that are hashed  $c$ -ideally. Building on this, we applied results by Dubhashi and Ranjan [\[22\]](#) to obtain our main lower bound of [\(2.2\)](#). Our second lower bound, [\(2.3\)](#), is a straightforward generalization of a direct approach for the special case  $c = \alpha = 1$  by Mehlhorn [\[44\]](#). We used two techniques to find complementing upper bounds. The first upper bound, [\(2.4\)](#), uses a Poissonization method combined with direct calculations and is mainly useful for  $c \in o\left(\frac{\ln m}{\ln \ln m}\right)$ . Our second upper bound, [\(2.5\)](#), relies on a Yao-inspired principle [\[39\]](#) and covers the case  $c \in \omega\left(\frac{\ln m}{\ln \ln m}\right)$ .

With these results on the size of  $c$ -ideal hash function families, we discovered that the advice complexity of hashing is linear in the hash table size  $m$  and only logarithmic in  $n$  and double logarithmic in  $u$  (see Schmidt and Siegel [\[54\]](#) for similar results for perfect hashing). Moreover, the influence of both  $\alpha$  and  $c$  is exponential in the lower

bound. In this sense, by relaxing the pursuit of perfection only slightly, the gain in the decrease of the size of a  $c$ -ideal hash function family can be exponential. Furthermore, only  $O(\ln \ln u + \ln n)$  advice bits are necessary for deterministic algorithms to catch up with randomized algorithms.

Further research is necessary to close the gap between our upper and lower bounds. For the edge cases, that is, for  $c \geq \log n / \log \log n$ , the upper and lower bounds (2.5) and (2.3) differ by a factor of approximately  $n \ln m$ . The interesting case for us, however, is  $c \leq \log n / \log \log n$ , which is the observed worst-case cost for universal hashing. Contrasting (2.2) and (2.4), we note that the difference has two main reasons. First, there is a factor of  $n \ln u$  that appears only in the upper bound; this factor stems from the general bounds in [Corollary 2.1](#). Second, the probability  $\mathbb{P}(\alpha_{\max} \leq c\alpha)$  is estimated from below in a more direct fashion than from above, leading to a difference between these bounds that increases with growing  $c$ . The reason for the more direct approach is the lack of a result similar to [Lemma 2.5](#).

Moreover, it remains an open question whether it is possible to adapt the entropy-related methods in the spirit of Fredman-Komlós and Körner in such a way as to improve our general lower bound (2.2) by accounting for the universe size in a meaningful way.



# 3

## Zero-Memory Graph Exploration with Unknown Incoming Ports

### 3.1 Introduction

Say you wake up one morning in an unknown hotel with the desire to stroll around and visit every place in the city. Considering your terrible headache, you don't bother to remember anything about which places you have visited, but still, at the end of the day, you want to return to your hotel. You know this is not possible without further aid so you decide to take some crayons with you and color every place you visit. You are endowed with keen eyes and you're able to see the colors of the places around you. All you now need to know is how many colors you have to take along and how you color the places. This chapter deals exactly with that situation.

The exploration of an unknown environment by a mobile entity is one of the basic tasks in many areas. Its applications range from robot navigation over image recognition to sending messages over a network. Due to the manifold purposes, there is a great deal of different settings in which exploration has been analyzed. In this chapter, we consider the fundamental problem of a single agent, e.g., a robot or a software agent, that has to first explore all vertices of an initially unknown undirected graph  $G$ , then return to the start vertex and terminate. By exploring we mean that the agent is located at a vertex and can, in each step, either go to an adjacent vertex or terminate.

If the vertices of  $G$  have unique labels and without further restrictions to the agent, this becomes a trivial task. However, in many applications, the environment is unknown and the agent is a simple and inexpensive device. Hence, we consider *anonymous* graphs, that is, there are no unique labels on the vertices or edges. Moreover, the edges have no port labels; the labeling is given implicitly by the order in which the agent sees the edges and can be different at each visit of a vertex. The agent itself is *oblivious*, that is, it has no persistent memory. Such agents are sometimes also called *zero-memory algorithms* or *1-state robots* [13, 15].

Clearly, with these restrictions, there does not exist a feasible exploration algorithm. In fact, not even a graph consisting of one single edge could be explored since the algorithm would not know when to terminate. Therefore, in most models with anonymous graphs and oblivious agents, the agent remembers through which port it entered a vertex. We, in contrast, assume that the agent does not know through which edge it entered a vertex. This is sometimes called *unknown inports* or *no inports* [46]. Instead, we allow the agent to color the current vertex, a feature which is also referred to as placing distinguishable pebbles [19] or labeling vertices [13]. In this chapter, we prefer the notion of coloring<sup>1</sup>. This notion emphasizes that a colored vertex may never be recolored unless we explicitly allow it and then use the term “recoloring.” However, having the ability to color vertices alone is still utterly useless for an oblivious agent that has to return to the start vertex. Therefore, we relax our restrictions by allowing the agent to see the labels (i.e., colors) of the neighboring vertices.

We consider storage efficiency and analyze the minimum amount of colors necessary and sufficient to explore any graph with  $n$  vertices. We prove that 3 colors are both necessary and sufficient to explore trees, whereas  $n - 1$  colors are necessary and sufficient to explore every graph with  $n$  vertices. This striking difference is not limited to planarity, graphs of large treewidth or graphs with a large feedback vertex set; in fact, even planar graphs with treewidth 2 and feedback vertex set number 1 need  $\Omega(n)$  colors. We discover that the driving parameter of a graph is its circumference, the length of a longest cycle. We show that  $2k - 1$  colors are sufficient and  $\min\{2k - 3, n - 1\}$  colors are necessary to explore all graphs of circumference at most  $k$ . We further show that it is possible to explore all so-called squares of paths with 4 colors, which shows that, if an algorithm does not have to explore all possible graphs but only graphs of a particular graph class, much fewer colors suffice. Finally, we make an ostensibly inconspicuous change to our model and analyze the case where we allow recoloring vertices. We show that, in this model, 7 colors are enough to explore all graphs.

This chapter is organized as follows. In the rest of this introduction, we consider related work and lay out the basic definitions. In [Section 3.2](#), we present the analysis when the graphs to be explored are trees. In [Section 3.3](#), we analyze the general case before we then turn to graphs of a certain circumference in [Section 3.4](#). Afterwards, we consider squares of paths in [Section 3.5](#) and finally recoloring in [Section 3.6](#) before we conclude in [Section 3.7](#).

## Related work

There is a vast body of literature on exploration and navigation problems. A great deal of aspects have been analyzed; in general, they can be categorized into one of the following four dimensions: environment, agent, goal, complexity measure. We briefly discuss these dimensions and highlight the setting considered in this chapter.

The environment dimension is concerned with whether there is a specific geometric setting or a more *abstract setting* such as a graph or a *graph class*. Sometimes there are special environmental features such as faulty links or, as in our case, *local memory*,

---

<sup>1</sup>Note that this coloring is just a normal labeling and has nothing to do with graph coloring such as in 3-COLORING; it is perfectly fine to color adjacent vertices with the same color.

sometimes also called storage. In the agent dimension, we find the aspects such as whether there is a *single agent* or whether there are multiple agents; whether the agents are *deterministic* or probabilistic; whether the agents have some restrictions such as *limited memory*, range or *view*; and whether the agents possess special abilities such as *marking of vertices* or teleportation. Some goals include mapping the graph, finding a treasure, meeting, exploring all edges, and *exploring all vertices*. For the latter, most researchers consider one of the following three modes of termination: perpetual exploration, where the agent has to visit every vertex infinitely often; exploration with stop, where the agent has to stop at some point after it has explored everything; and *exploration with return*, where the agent has to return to the starting point after the exploration and then terminate. The main complexity measures are time complexity, space/memory complexity, *storage complexity* and competitive ratio<sup>2</sup>.

As highlighted above, in this chapter, we focus on vertex exploration by a single agent with local memory or storage. For an overview on this segment of exploration problems, we refer to the excellent overview of Das [15] and the first two chapters of [25] by Gašieniec and Radzik. We are not aware of any research on the exploration of anonymous graphs by oblivious agents where the labels of the neighboring vertices are visible. However, the models of Cohen et al. [13] and Disser et al. [19] are similar to ours; they analyze graph exploration with anonymous graphs, local port labels, a single oblivious agent, and the ability to label the current vertex.

In the model of Cohen et al. [13], there is a robot  $\mathcal{R}$  with a finite number of states that has to explore all vertices of an unknown undirected graph and then terminate. The robot sees at each vertex the incident edges as *port numbers*. The order of these edges is fixed per vertex, but unknown to  $\mathcal{R}$ . The robot knows through which port it entered a vertex. In a preprocessing state, the vertices are labeled with pairwise different labels. Cohen et al. analyzed how many labels are necessary to explore all graphs. They proved that a robot with constant memory can explore all graphs with just three labels. Moreover, they showed that for any  $d > 4$ , an oblivious robot that uses at most  $\lfloor \log d \rfloor - 2$  pairwise different labels cannot explore all graphs of maximum degree  $d$ .

Their model does not directly compare to ours: Our labels are not assigned in a preprocessing stage, but during the exploration by the algorithm/robot. Moreover, the incoming port number is not known, the order of the port numbering is not fixed, the labels may not be changed, the algorithm has to return to the start vertex, and, most importantly, the algorithm sees the labels of the adjacent vertices. However, even though the models are quite different, we see that known inports is a much stronger feature than seeing the labels of neighboring vertices: Our general lower bound does not depend on the maximum degree, but on the number of vertices in the graph. As we will see later, we provide a lower bound that is linear in the number of vertices even when the maximum degree is restricted to 3.

The model of Disser et al. [19] is even closer to our model. Here, the labels—they call them distinguishable pebbles—are assigned during the exploration by the agent/algorithm as well. Moreover, the goal is exploration with return. They showed

---

<sup>2</sup>Here, we use “time complexity” as the complexity of the algorithm that calculates the decisions of the agent and “competitive ratio” as the number of time steps of the agent compared to an optimal number of time steps.

that, for any agent with sub-logarithmic memory,  $\Theta(\log \log n)$  labels are necessary and sufficient to explore any graph with  $n$  vertices. Moreover, they characterized the trade-off between memory and the number of labels as follows. When the agent has  $\Omega(\log(n))$  bits of memory, all graphs on  $n$  vertices can be explored without any labels. As soon as the agent only has  $\mathcal{O}(\log(n)^{1-\varepsilon})$  bits of memory,  $\Omega(\log \log(n))$  labels are needed to explore all graphs on  $n$  vertices. However, with that many labels, even a constant amount of memory suffices for the exploration.

As before, this model does not directly compare to ours since neither is contained in the other. Their results seem to support the idea that knowing imports is stronger than seeing the labels of neighboring vertices; however, in contrast to our model, the focus of their work was on constant or sub-logarithmic memory.

### Basic Definitions

We use the usual notions from graph theory as found for example in the textbook by Diestel [16]. The graph exploration setting considered in this chapter is defined as follows. We first describe the setting informally. An agent is placed on a vertex, called the start vertex, of an undirected connected graph and moves along edges, one edge per step. In a step, the agent may use an arbitrary natural number to color the vertex on which it is currently located, if this vertex was uncolored up to now, and then move to a neighbor. As basis for its decisions, the agent may only use the color of the current vertex and the colors of the neighboring vertices. The agent can neither use the identity of the vertices nor any numbering of the edges. The agent has no persistent memory; in particular, the agent does not know through which edge it entered the current vertex (if any; the start vertex is not entered from anywhere). On the basis of the coloring of the current vertex and the neighbors, the agent chooses a neighbor it wants to move to, or it decides to stop. For no decision or choice can the agent distinguish between neighbors that have the same color.

The task is to provide a strategy for the agent—which is called an algorithm *ALG*—that determines for each situation the action of the agent in such a way that no matter on which graph and on which start vertex the agent is placed, and no matter which neighbors are used to go to if there is a choice, the agent will visit all vertices, return to the start vertex, and stop there.

As with classical online problems, the concept of an *adversary* is useful to formulate bounds on the number of colors necessary and sufficient to explore all graphs. This adversary makes the decisions that are left open in the informal description above, namely choosing the start vertex and choosing the vertex the agent visits next if there is a choice among neighbors of the same color to which the agent wants to go. For all graphs and for all possible choices of the adversary, the agent must visit all vertices and then stop on the start vertex.

An algorithm in this model is a function that determines what the agent should do when located on a vertex with a certain color structure in the neighborhood. We denote the color structure by a pair  $(c_0, E)$ , where  $c_0 \in \mathbb{N}$  stands for the color of the current vertex and  $E: \mathbb{N} \rightarrow \mathbb{N}$ , where  $E$  is 0 almost everywhere, stands for the colors of the neighbors. For a number  $c \in \mathbb{N}$ ,  $E(c)$  stands for the number of neighbors of color  $c$ . In cases where we consider only a restricted amount of colors, we consider functions  $c_0$  and  $E$  of smaller domain, for example colors  $c_0 \in \{1, \dots, n_c\}$ , and  $E: \{1, \dots, n_c\} \rightarrow \mathbb{N}$ . We denote the set of all pairs  $(c_0, E)$  by  $\mathcal{E}$  and call it the

*environment.*

An *algorithm* in this model is a function

$$\text{move}: \mathcal{E} \rightarrow \mathbb{N} \times \mathbb{N} \times \{\text{STOP}\},$$

with the restriction that whenever  $\text{move}(c_0, E) = (c_1, d)$ , we must have  $E(d) > 0$ , that is, the algorithm is not allowed to send the agent to a neighbor that does not exist. When the number of colors allowed is at most  $n_c$ , the range of move is  $\{1, \dots, n_c\} \times \{1, \dots, n_c\} \times \{\text{STOP}\}$ . Moreover, we must have  $c_1 = c_0$  unless  $c_0 = 0$ , that is, the agent may only color the current vertex if it was not colored before. We analyze the model where this last restriction is canceled in [Section 3.6](#).

To carry out an algorithm on a given graph  $G$ , we use an adversary, and carry out steps. Initially, all vertices are uncolored, that is, have color 0. The adversary selects a start vertex  $v_0$  and places the agent there. Then, each step works as follows: The agent is located in a vertex  $v$ . The coloring in the neighborhood is translated into an element  $(c_0, f)$  of  $\mathcal{E}$  in the obvious way, by counting the number of neighbors for each occurring color. The value  $\text{move}(c_0, f)$  determined by the algorithm is then used as follows. If it is STOP, the run of the algorithm stops. If it is a pair  $(c_1, d)$ , vertex  $v$  is colored with color  $c_1$ , and the adversary chooses a neighbor of  $v$  of color  $d$ , to which the agent moves for the next step.

An algorithm *successfully explores all graphs* if for all graphs  $G$  and for all adversaries, after finitely many steps, all vertices of  $G$  have been visited (they do not need to have a color  $c > 0$ , but they must have been the “current vertex” in some step), the agent is located on the start vertex  $v_0$ , and the decision of the algorithm is STOP. An algorithm is correct on a graph class  $\mathcal{C}$  if it successfully explores all graphs  $G \in \mathcal{C}$ .

Throughout the chapter, we denote by  $n$  the number of vertices of  $G$  and use  $[n]$  to denote  $\{1, \dots, n\}$ . For a vertex  $v \in V$ , we write  $c(v) \in \mathbb{N}$  for its color;  $N(v)$  for the open neighborhood of  $v$ , that is, the set of vertices adjacent to  $v$ ; and  $N[v]$  for the closed neighborhood of  $v$ , where  $N[v] := N(v) \cup \{v\}$ . We use  $\text{mod}_1$  to denote a modulo operator shifted by 1, i.e.,

$$n \text{ mod}_1 m := ((n - 1) \text{ mod } m) + 1.$$

Instead of having the numbers  $0, \dots, m - 1$  as the outcome of the modulo operation, one thus obtains the numbers  $1, \dots, m$ . For convenience, we sometimes speak of an algorithm behaving in a particular way and mean by this formulation that the agent of the algorithm behaves in a particular way.

## 3.2 Exploration of Trees

We begin by analyzing the problem on trees. We show that only three colors are enough to explore all trees. In line with the research that analyzes graph exploration with pebbles, we do not count 0 as a color. The idea of the algorithm is to alternately label the vertices with colors 1, 2, and 3 and then follow a simple depth-first strategy. Since there are no cycles, there is always a unique path back to the start vertex and backtracking is possible. Moreover, the start vertex is recognized by the fact that it is the only vertex with color 1 where all neighbors have color 2. This strategy is formalized in the algorithm TREEEXPLORATION below.

**Algorithm 1** TREEEXPLORATION

**Input:** An undirected tree in an online fashion. In each step, the input is  $c(v) \in \mathbb{N}$ , which is the color of the current vertex  $v$ , and  $c(v_1), \dots, c(v_d)$ , the colors of the neighbors of  $v$ , where  $d$  is the degree of  $v$ .

**Output:** In each step, the algorithm outputs  $c(v)$ , the color the agent assigns to  $v$ , and  $i \in [d]$ , the vertex the agent goes to next.

**Description:** Assign alternately color 1, 2, and 3 in order to achieve a sense of direction.

---

```

1: if  $c(v) = 0$  then
2:    $c(v) := \left( \left( \max_{v' \in N(v)} c(v') \right) + 1 \right) \bmod_1 3$ 
3: if there is an uncolored neighbor then
4:   go to some uncolored neighbor.
5: else if  $c(v) = 1$  and there is no neighbor  $v'$  having  $c(v') = 3$  then
6:   terminate.
7: else
8:   go to a neighbor  $v'$  with  $c(v') = (c(v) - 1) \bmod_1 3$ .  $\triangleright$  this neighbor is going
   to be unique
9:

```

---

Obviously, TREEEXPLORATION uses at most 3 colors. We prove first that the algorithm terminates on the start vertex and only on the start vertex.

**Lemma 3.1.** *The agent terminates only upon reaching the start vertex.*

*Proof.* The termination rule can only be applied on a vertex with color 1 without any neighbor  $v'$  with  $c(v') = 3$ . Clearly, this is true for the start vertex. Apart from the start vertex, all vertices assigned color 1 have a neighbor with color 3 since there is no vertex with only uncolored neighbors except the start vertex.  $\square$

Then, we prove that TREEEXPLORATION does indeed visit every vertex and return to the start vertex.

**Lemma 3.2.** *TREEEXPLORATION explores all vertices and then returns to the start vertex.*

*Proof.* The color assignment in line 2 has the effect that vertices are colored with  $(d(v)+1) \bmod_1 3$ , where  $d(v)$  is the distance from the start vertex. Clearly, since there are no cycles in the graph, TREEEXPLORATION always returns to the start vertex via line 9. Once all neighbors of the start vertex are explored, TREEEXPLORATION terminates by Lemma 3.1.

We now show that all vertices are explored before TREEEXPLORATION terminates: Assume towards contradiction that there is a tree  $T = (V, E)$  where not all vertices are explored. Let  $C \subset V$  be the vertices that are explored and thus colored and let  $U \subset V$  be the vertices that are not explored.

Since  $T$  is connected, there is an edge  $\{v, w\}$  between two vertices  $v \in C$  and  $w \in U$ . The vertex  $v$  is visited at some point. There is an unvisited neighbor, namely  $w$ , hence line 3 is applied and TREEEXPLORATION visits an unexplored neighbor,



say  $u \in C$ . Let us call the set of vertices in  $C$  whose shortest path to the start vertex contains  $v$  the *branch* of  $v$ . Clearly,  $u$  is in the branch of  $v$ .

After  $u$ , TREEEXPLORATION might visit other vertices in the branch of  $v$ . At some point, all vertices in the branch of  $v$  are visited and only lines 6 or 9 can be applied. By Lemma 3.1, line 6 can only be applied on the start vertex and the start vertex is not in the branch of  $v$ . Therefore, only line 9 can be applied. Since there are no cycles, TREEEXPLORATION thus returns to  $v$  at some point. Now, however, all neighbors of  $v$  except  $w$  and maybe other neighbors in  $U$  are colored. Therefore,  $v$  has to visit  $w$  or another neighbor in  $U$  according to line 8. This contradicts our assumption that  $w \in U$ . Hence, TREEEXPLORATION explores all vertices.  $\square$

Together, these two lemmas yield the following theorem.

**Theorem 3.1.** TREEEXPLORATION is correct on trees and uses at most 3 colors.

We prove that TREEEXPLORATION is optimal in the sense that it is impossible to use fewer than 3 colors to solve graph exploration in our model on trees. Before doing so, we make the following crucial observation.

**Observation 3.1 (Functional Nature).** *Due to its functional nature, once an algorithm is in a vertex  $v$  where all vertices in  $N[v]$  have been colored and takes a decision upon which it goes to a neighbor  $w$  or upon which the adversary chooses neighbor  $w$  as next vertex, this choice can be made by the adversary each time the algorithm returns to  $v$ . Moreover, once the algorithm is in a vertex  $v$  and takes a decision to go to a neighbor of color  $d$ , the same decision will be made in a vertex  $w$  if  $c(w) = c(v)$  and the colors in  $N(w)$  can be mapped bijectively to the colors in  $N(v)$ .*

**Theorem 3.2.** *There is no algorithm that solves graph exploration as in our model on every tree and that uses less than 3 colors.*

*Proof.* Assume by contradiction that there exists such an algorithm. Consider a path with seven vertices. Denote them by  $v_1$  to  $v_7$  in sequential order. Let  $v_1$  be the start vertex.

For  $i \in [5]$ , if an agent does not color  $v_i$ , it cannot visit  $v_{i+2}$ : The agent could continue to  $v_{i+1}$ , but then it cannot distinguish between  $v_i$  and  $v_{i+2}$ . Therefore, the adversary can make the agent go back to  $v_i$ . If the agent now does not color  $v_i$ <sup>3</sup>, it is caught in an endless loop by Observation 3.1, which contradicts the assumption that the algorithm works correctly on all trees.

When the agent has reached  $v_7$ , it has to return to  $v_1$ . It must make a step from  $v_4$  to  $v_3$  at some point. If  $c(v_3) = c(v_5)$ , the adversary can make the agent go back to  $v_5$  instead, which again opens the way to an endless loop. Hence,  $c(v_3) \neq c(v_5)$ . The same is true at  $v_3$  and at  $v_2$ . Hence,  $c(v_4) \neq c(v_2)$  and  $c(v_1) \neq c(v_3)$ . Without loss of generality, assume  $c(v_1) = 1$ . Then this implies that either  $(c(v_1), \dots, c(v_5)) = (1, 1, 2, 2, 1)$  or  $(c(v_1), \dots, c(v_5)) = (1, 2, 2, 1, 1)$ .

If  $(c(v_1), \dots, c(v_5)) = (1, 1, 2, 2, 1)$ , the agent has to go from  $v_4$  to  $v_3$ ; however, then the agent goes from  $v_3$  back to  $v_4$  because of its functional nature. Similarly, if

<sup>3</sup>Note that the agent may have colored  $v_{i+1}$  and thus the environment of the second visit of  $v_i$  may be different from the environment of the first visit, leading to a potentially different decision.

$(c(v_1), \dots, c(v_5)) = (1, 2, 2, 1, 1)$ , the agent has to go from  $v_3$  to  $v_2$ , but then it goes from  $v_2$  back to  $v_3$ .  $\square$

As an aside, we would like to note that the example with the path in the proof above does not work if we change our model and allow recoloring vertices. Then, it would be possible to successfully explore all paths with the start vertex being a leaf by just using one color in addition to the “non-color” 0. A possible algorithm might proceed as follows.

If  $c(v) = 0$  and there is an uncolored neighbor, set  $c(v) = 1$  and go to that neighbor; if  $c(v) = 1$  or if there is no uncolored neighbor, set  $c(v) = 0$  and go to the colored neighbor. If  $c(v) = 1$  and there is no colored neighbor, terminate. This way, the algorithm visits all vertices from the start leaf up to the other leaf and assigns to each vertex except the last one color 1. From the last vertex, the algorithm goes back to the start vertex by always deleting the assigned color. This is possible since the color of the current vertex is given to the algorithm and the algorithm thus knows whether the vertex has been visited at some point before.

We return to recoloring in more depth in [Section 3.6](#), where we corroborate our finding that recoloring radically changes our model, resulting in much more powerful algorithms.

### 3.3 Exploration of General Graphs

In this section, we provide bounds for the exploration of general graphs, that is, without restricting the graph class. We begin with an upper bound, then turn to a corresponding lower bound, and then diverge slightly and discuss non-uniform algorithms, where the algorithm knows the size of the input graph.

#### Upper Bound

How can an agent proceed on graphs that are not necessarily trees? Again, a depth-first search strategy suffices; however, this time, the agent uses almost as many colors as vertices.

Let us first describe the idea. The agent starts at the start vertex and colors it with color 1. This vertex is the root. All other vertices will receive larger colors; hence, the root is easy to recognize. We wish to carry out DFS. It is easy to go whenever possible to an unexplored—or, equivalently, uncolored—neighbor of the current vertex. But how shall the agent find the way back? The idea is to color a new vertex with a color 1 larger than the largest color in the neighborhood. This ensures that colors increase along paths taken forward in the tree. A notable exception is when the agent arrives at a leaf in the DFS tree, that is, a vertex where all neighbors are colored. Then, the agent can save a color by assigning the largest color in the neighborhood—which is the color of the vertex the agent came from—instead of the largest color in the neighborhood plus one. For backtracking from a vertex  $v$ , the agent goes to a vertex whose color is one less than  $c(v)$ . Such a vertex always exists, except if  $v$  is the root. The only thing one has to prove is that this neighbor is unique. The formal description of `DEPTHFIRSTSEARCH` is provided by [Algorithm 2](#).

Clearly, `DEPTHFIRSTSEARCH` is well defined. We prove in the following that `DEPTHFIRSTSEARCH` explores all vertices and uses at most  $n - 1$  colors. We start by



proving that whenever DEPTHFIRSTSEARCH goes to a vertex that has been colored before, this vertex is the predecessor of the current vertex. By *predecessor* of a vertex  $v$  that is not the root, we mean the neighbor  $w$  from which the agent moved to  $v$  when  $v$  was first visited.

---

**Algorithm 2** DEPTHFIRSTSEARCH
 

---

```

1: ▷ Save one color by not assigning a new color for leaves or any vertices where all
   neighbors are colored
2: if  $c(v) = 0$  and all neighbors are colored then
3:    $c(v) := \left( \max_{v' \in N(v)} c(v') \right)$ 
4:   go to a neighbor  $v'$  with maximal color
5:   ▷ Normal case: on a new vertex, there is another uncolored neighbor
6: else if  $c(v) = 0$  then
7:    $c(v) := \left( \max_{v' \in N(v)} c(v') \right) + 1$ 
8: if there is a neighbor  $v'$  with  $c(v') = 0$  then
9:   go to  $v'$ . If there are multiple such  $v'$ , go to the one that is presented first
10: else if there is a neighbor  $v'$  with  $c(v') = c(v) - 1$  then
11:   go to  $v'$ 
12: else
13:   terminate.

```

---

**Lemma 3.3.** *On any vertex  $v$ , either DEPTHFIRSTSEARCH goes to an unvisited vertex or it goes back to the predecessor of  $v$ . In other words, whenever line 11 is applied,  $v'$  is the predecessor of  $v$ .*

*Proof.* We prove this by induction on the number of times line 11 is applied. If line 11 is applied for the first time, there can only be one vertex  $v'$  with color  $c(v') = c(v) - 1$  since assigning a color multiple times requires backtracking, i.e., line 11. Moreover, by the same reasoning,  $v$  indeed receives color  $c(v') + 1$ ; there cannot be another neighbor  $z$  with color  $c(z) > c(v')$ .

Assume that line 11 has been applied  $k - 1$  times and always the neighbor visited right before has been visited. We show that for the  $k$ -th time when line 11 is applied, again the neighbor visited right before will be visited. Call the current vertex  $v$  and let  $v'$  be the vertex visited right before  $v$ . We prove two things. First, we show that there is only one neighbor  $v''$  with color  $c(v'') = c(v) - 1$ ; second, we show that  $c(v) = c(v') + 1$ .

Assume there is another neighbor  $z$  of  $v$  with  $c(z) > c(v')$ . We distinguish two cases: Either  $v'$  was visited before  $z$  or vice versa. In the first case, by the induction hypothesis, the only way for the agent to go from  $z$  back to  $v'$  is via backtracking at some point from  $z$  to its predecessor. However, this is not possible since line 11 can only be applied when there is no unvisited neighbor, but  $v$  is such an unvisited neighbor. In the second case, consider the search sequence  $z = v_1, v_2, \dots, v_k = v'$  of the algorithm between  $z$  and  $v'$ . The only way that  $z$  is assigned a larger color than  $v'$  is that line 11 is applied between  $z$  and  $v'$ . By the induction hypothesis,

whenever this happens, the direct predecessor is visited. So either line 11 is applied right from  $z = v_1$  or the algorithm goes to a previously unvisited vertex  $v_2$  and  $z$  occurs somewhere else in the sequence. At some point, the agent has to backtrack from  $z$ , which is not possible since  $z$  has an unvisited neighbor. Therefore, such a neighbor  $z$  does not exist.

Assume there are two vertices  $v'$  and  $v''$  with  $c(v') = c(v'') = c(v) - 1$  and the agent has visited  $v''$  some time before  $v'$ . Let  $v'' = v_1, \dots, v_s = v'$  be the sequence the algorithm took to get from  $v''$  to  $v'$ . According to the induction hypothesis, whenever line 11 was applied in this sequence, a direct predecessor was visited. The vertex  $v''$  might appear several times in this list. Let  $v_t$  be the last vertex in the sequence that is equal to  $v''$ . By construction, colors change only by 1 between two neighbors in this sequence. If  $c(v_{t+1}) = c(v'') + 1$ , there would have to be another vertex  $\bar{v}$  in the list with color  $c(\bar{v}) = c(v'')$ . However, since there are only direct predecessors when applying line 11 and since the color cannot decrease otherwise, this is not possible. Hence,  $c(v_{t+1}) = c(v'') - 1$ . However, this is not possible either since line 11 cannot be applied when there is an unvisited neighbor, which is the case here with  $v$ . Therefore, the assumption that there are two vertices  $v'$  and  $v''$  with  $c(v') = c(v'') = c(v) - 1$  is wrong and there is only one such vertex. This is the direct predecessor of  $v$ , which proves the lemma.  $\square$

**Lemma 3.4.** *DEPTHFIRSTSEARCH explores all vertices.*

*Proof.* We can argue similar as for Lemma 3.2. Assume towards contradiction that there is a graph  $G = (V, E)$  where not all vertices are explored. Let  $C \subset V$  be the vertices that are explored and thus colored and let  $U \subset V$  be the vertices that are not explored.

Since  $G$  is connected, there is an edge  $\{v, w\}$  between a vertex  $v \in C$  and  $w \in U$ . The vertex  $v$  is visited at some point. According to our observation above, DEPTHFIRSTSEARCH then visits an unexplored neighbor. Going back to the predecessor of  $v$  is not possible since  $v$  has an unexplored neighbor, namely  $w$ , and line 11 is only evaluated if the condition of line 8 is wrong, i.e., if there is no unexplored neighbor. DEPTHFIRSTSEARCH then continues to visit other vertices in  $C$ . However, at some point, there are no unvisited vertices in  $C$  left and DEPTHFIRSTSEARCH has to return according to line 11.

By Lemma 3.3, DEPTHFIRSTSEARCH returns to the direct predecessor every time. Therefore, DEPTHFIRSTSEARCH visits  $v$  again. Now, DEPTHFIRSTSEARCH may visit another unexplored neighbor  $s \in C$ . By the same arguments as before, however, DEPTHFIRSTSEARCH returns to  $v$ . At some point, all neighbors of  $v$  in  $C$  are explored and the algorithm returns to  $v$ . Then, the vertex  $w$  has to be explored next according to line 8. This contradicts our assumption that  $w \in U$ . Therefore, on every graph  $G$ , DEPTHFIRSTSEARCH explores all vertices.  $\square$

**Lemma 3.5.** *The agent in DEPTHFIRSTSEARCH returns to the start vertex and terminates.*

*Proof.* The agent cannot get stuck since, for every vertex  $v$  except the start vertex and except leaves, there is always a neighbor  $v'$  with color  $c(v') = c(v) - 1$ . Once every vertex is explored, the start vertex has no such neighbor, hence the algorithm terminates in such a case in the start vertex.  $\square$

Together with the obvious fact that `DEPTHFIRSTSEARCH` uses at most  $n - 1$  pairwise different colors since no vertex is colored twice and since the last vertex that is colored has no uncolored neighbors, these lemmas prove the correctness of `DEPTHFIRSTSEARCH`:

**Theorem 3.3.** *`DEPTHFIRSTSEARCH` is correct and uses at most  $n - 1$  colors.*

### Lower Bound

At first glance and having seen that `TREEEXPLORATION` only uses three colors, we are tempted to think that `DEPTHFIRSTSEARCH` uses unnecessarily many colors. However, we are going to show in the following that `DEPTHFIRSTSEARCH` is optimal in our model. Hence, our perspective changes. We now deal with an unknown algorithm of which we only know that it successfully explores all graphs. In order to show a lower bound, we have to define a graph and the decisions of an adversary such that every algorithm with a limited amount of colors fails to explore our graph given the decisions of our adversary.

We start with an easy observation and two technical lemmas. These lemmas will be crucial in creating lower bounds for our model. They will allow us to define an instance and then argue that the agent has to take a certain path in this instance and cannot choose a different route.

**Observation 3.2.** *If the agent of an algorithm `ALG` that successfully explores all graphs arrives at a colored vertex  $v$  and there is an uncolored neighbor, then the agent has to visit an uncolored neighbor next.*

*Proof.* The uncolored neighbors might all be leaves in the input graph and if the agent decides to go to a colored neighbor, then it will always take this decision when located on  $v$  and never explore the uncolored leaves, which could only be explored via a move from  $v$ .

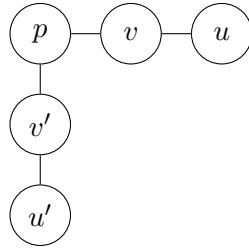
In order to successfully explore input graphs where the uncolored neighbors are all leaves, `ALG` thus has to prefer uncolored neighbors over colored neighbors.  $\square$

### Lemma 3.6 (General Algorithm Cannot Go Back Unless Necessary).

*Let `ALG` be an algorithm that successfully explores all graphs. If the agent of `ALG` is on a vertex  $v$  with only one colored neighbor  $p$ , which is its predecessor, and at least one uncolored neighbor  $u$ , and if  $p$  has an uncolored neighbor  $v' \neq v$ , then the algorithm has to choose to go to an uncolored neighbor of  $v$ .*

*Proof.* First, note that this situation is more delicate than the situation of [Observation 3.2](#) since we do not assume that  $v$  is colored. Assume towards contradiction that there is an algorithm where the lemma is not true. In other words, assume that there is an algorithm `ALG` that explores all graphs and where there is a graph with a vertex  $v$ , a colored predecessor  $p$  as its only colored neighbor and at least one uncolored neighbor  $u$  and assume that  $p$  has at least one uncolored neighbor  $v' \neq v$  and `ALG` does not choose to go to one of the uncolored neighbors while on  $v$ .

We change the part of the graph on which `ALG` runs such that  $v'$  has as many uncolored neighbors as  $v$ . Moreover, we assume that all the uncolored neighbors of  $v$  and  $v'$  are leaves and there are no paths from the rest of the graph to either  $v$  or  $v'$ . This situation is depicted in [Figure 3.1](#).

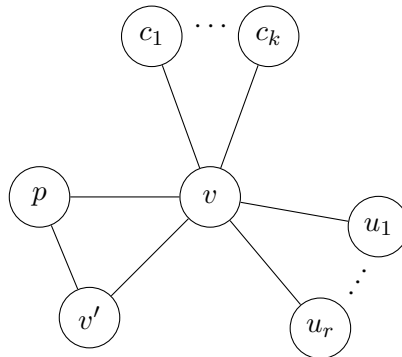


**Figure 3.1.** If the graph that ALG processes contains this structure, ALG cannot explore it successfully.

By assumption, ALG goes back to  $p$  from  $v$ . If ALG has not colored  $v$ , then it will again go to  $v$  from  $p$  and there is an adversary such that ALG never terminates. Hence, ALG colors  $v$ . At some point, ALG has to visit  $v'$ , which is only possible from  $p$ . Whenever the algorithm now wants to visit some uncolored neighbor in  $p$ , the adversary sends the agent to  $v'$ . When the agent arrives in  $v'$ , it is in the same situation as before when it was on  $v$ . Therefore, ALG assigns to  $v'$  the same color as to  $v$  and then it goes back to  $p$ . Now, however, due to [Observation 3.1](#), if ALG decides at some point to go to  $v$  or to  $v'$  from  $p$ , ALG will, when in  $p$ , always go to  $v$  or to  $v'$  and will, therefore, either not be able to explore  $u$  and  $u'$  or will not return to the start vertex and terminate.  $\square$

**Lemma 3.7.** *Let ALG be an algorithm that successfully explores all graphs. Assume the agent has just moved from a (now) colored vertex  $p$  to an uncolored vertex  $v$ , and that  $p$  has at least one other uncolored neighbor  $v'$ . Then the agent must color  $v$  now.*

*Proof.* Assume towards contradiction that there is an algorithm ALG and a graph  $G$  with vertices  $p$ ,  $v$  and  $v'$  such that the described situation arises but  $v$  is not colored. By changing the graph  $G$ , we may assume that  $p$ ,  $v$ , and  $v'$  form a triangle and  $v'$  is not connected to any other vertices. Moreover, assume that all uncolored neighbors of  $v$  apart from  $v'$  are leaves and call them  $u_1$  to  $u_r$ , for some  $r \in \mathbb{N}$ . Denote by  $c_1$  to  $c_k$ , for some  $k \in \mathbb{N}$ , the colored neighbors of  $v$  apart from its predecessor. This situation is depicted in [Figure 3.2](#).



**Figure 3.2.** If the vertices  $p$ ,  $v$ ,  $v'$  are arranged in a triangle, ALG cannot explore the graph successfully.

When ALG leaves  $v$ , it has three options. Either ALG goes back to  $p$ , or ALG goes to an uncolored neighbor, or ALG goes to some other colored neighbor  $c$ . The first option is prohibited by [Lemma 3.6](#). In case of the second option, the adversary can send ALG to  $v'$ . We change the graph such that  $v'$  has the same neighbors as  $v$ . Then, the input for ALG will be the same as in the step before, resulting in an endless loop. In case of the third option, ALG has to return at some point to  $v$  or to  $p$  since  $v'$  has not been visited yet. Consider the first time when ALG returns to  $v$  or to  $p$ . When ALG wants to visit  $v'$  from  $p$ , the adversary sends ALG to  $v$ . When ALG is in  $v$ , the situation looks exactly like before; hence, ALG will again choose the third option and go to the same colored neighbor  $c$ . Therefore, ALG never visits  $v'$  and thus does not successfully explore all graphs, contradicting the assumption.  $\square$

Before we now state and prove that DEPTHFIRSTSEARCH uses the minimal number of colors necessary in general, we prove the statement separately for graphs of size 2 and for graphs of size 3. In fact, we prove a stronger statement, namely without the condition that the algorithm needs to explore all graphs. Hence, we can even assume that the algorithm knows the size of the graph.

**Lemma 3.8.** *For  $k \in \{2, 3\}$ , the following holds: For every algorithm ALG that successfully explores all graphs of size  $k$ , there is a graph  $G$  of size  $k$  such that ALG uses at least  $k - 1$  colors to explore  $G$ .*

*Proof.* Clearly, an algorithm needs at least 1 color to explore a path of two vertices.

For graphs on three vertices and 2 colors, consider the exploration of a triangle with vertices  $x, y, z$ . Without loss of generality, assume the agent starts on  $x$ . If the agent does not color  $x$  and visits the next vertex, it is caught in an endless loop and will always leave the current vertex uncolored and go to the next vertex. Therefore,  $x$  receives some color  $c$  and the agent continues to the next vertex, which we call  $y$ . We show now first that the agent assigns some color to  $y$  and second that the assigned color cannot be  $c$ .

First, if the agent does not color  $y$  and goes back to  $x$ , the agent then has to visit  $z$ . However, since the agent cannot distinguish  $y$  from  $z$ , the adversary can arrange the ordering of the vertices in such a way that the agent visits  $y$  instead of  $z$ , whereupon the agent is stuck in an endless loop. Hence, the agent has to color  $y$ .

Second, if the agent colors  $y$  with the same color as  $x$  and then continues to  $z$ —either directly or via  $x$ —, the agent then cannot distinguish  $x$  from  $y$ . Hence, the adversary can arrange the ordering of the vertices such that the agent will not terminate on the start vertex.  $\square$

**Theorem 3.4.** *For every algorithm ALG that successfully explores all graphs and for every natural number  $n$  there is a graph  $G$  of size  $n$  such that ALG uses at least  $n - 1$  colors to explore  $G$ .*

*Proof.* For  $n = 1$ , there is nothing to prove. For  $n \in \{2, 3\}$ , the statement follows from [Lemma 3.8](#).

For graphs of size 4, we can use [Theorem 3.2](#) and the fact that the algorithm has to explore all graphs—and, in particular, all trees—to prove that an algorithm needs 3 colors to explore all graphs of size 4. We describe how this works: Consider a cycle of 4 vertices, say  $x, y, z, w$ . Assume the agent starts on  $x$ . Clearly, if the agent does

not color  $x$  in the first step, no vertex can ever be colored. Assume that  $x$  receives color 1 and the agent moves to  $y$ . By Lemma 3.6, the agent moves to  $z$  next. If  $y$  were not colored, the next vertex—which is  $z$ —would again receive color 1, and the agent would not be able to distinguish  $x$  from  $z$  when standing on  $w$ . Therefore,  $y$  receives color 2 and the agent moves to  $z$ .

By Lemma 3.6, the agent moves to  $w$  next. If  $z$  were not colored and if the graph were a cycle on 5 vertices with a vertex  $u$  between  $w$  and  $x$ , the next vertex,  $w$ , would again receive color 1 and, as before, the agent would not be able to distinguish  $w$  from  $x$  when standing on  $u$ . Since the algorithm is a general algorithm that has to successfully explore all graphs and that does not know the size of the graph, vertex  $z$  thus has to be colored.

If  $z$  received color 2, and if the graph were a path of seven vertices—as in the proof of Theorem 3.2—in the order  $w, x, y, z, u, v$ , then the vertex  $u$  would receive color 2 as well and on the way back to the start vertex  $x$ , the agent could not distinguish between  $y$  and  $u$  when standing on  $z$ . Therefore,  $z$  has to receive a third color.

We now turn to the proof for graphs of size 5 or larger: Assume towards contradiction that there is an algorithm ALG that, for some  $n_0 \geq 5$ , uses at most  $n_0 - 2$  colors on every graph of size  $n_0$ . We are going to define a family of graphs of size  $2(n_0 - 1) - 1$  and prove that there is a graph  $G_0$  in this family such that ALG uses at least  $n_0 - 1$  colors on its first  $n_0 - 1$  steps in order to successfully explore  $G_0$ . Afterwards, we define a graph  $G_1$  of size  $n_0$  which locally looks, for the first  $n_0 - 1$  steps, exactly as  $G_0$ . Therefore, ALG uses at least  $n_0 - 1$  colors on  $G_0$  as well.

We construct our family of graphs as follows. Consider a graph  $G$  that consists of a path  $v_1, \dots, v_{n_0-1}$  of length  $n_0 - 1$  and where, for each  $r \in [n_0 - 1]$ , a leaf  $l_r$  is connected to  $v_r$ . For fixed  $i, j \in [n_0 - 1]$ , we merge the leaves  $l_i$  and  $l_j$  and call the new vertex  $l_{i/j}$  and the graph  $G_{n_0, i, j}$ . Such a graph is depicted in Figure 3.3.

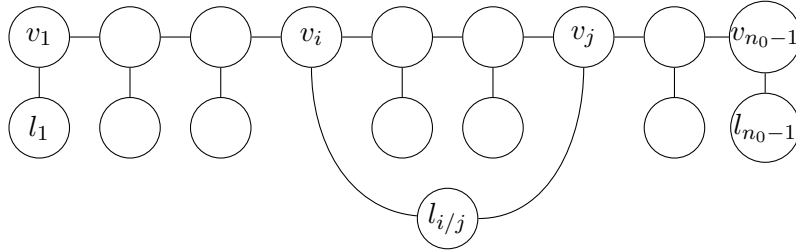


Figure 3.3. The graph  $G_{n_0, i, j}$

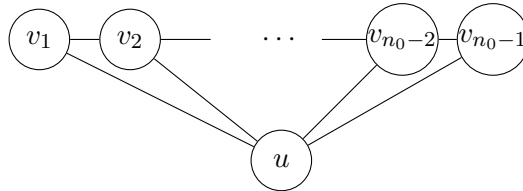
Consider any general algorithm ALG with at most  $n_0 - 2$  colors that explores a graph  $G_{n_0, r, s}$  for yet to be determined  $r$  and  $s$ . The adversary makes the decisions in such a way that ALG walks from  $v_1$  to  $v_{n_0-1}$  without exploring any leaves and without leaving any visited vertex uncolored. This is possible by Lemma 3.6 and Lemma 3.7.

By the pigeonhole principle, there are  $i < j$  such that ALG assigns the same color to  $v_i$  and  $v_j$ . Let  $r = i$  and  $s = j$ ; in other words, consider the exploration of ALG on  $G_{n_0, i, j}$ . Without loss of generality, we assume that  $v_i$  and  $v_j$  are not adjacent since, if that were the case, clearly, all  $v_k$  with  $k > j$  would be assigned the color of  $v_j$  and ALG cannot explore all graphs this way.

On its way back to  $v_1$ , ALG reaches  $v_j$  at some point. From there, ALG has to

visit  $l_{i/j}$  by [Observation 3.2](#). Since ALG cannot distinguish  $v_i$  and  $v_j$  when located in  $l_{i/j}$ , the adversary can send ALG to either vertex if ALG wants to visit a neighbor. If ALG decides not to color  $l_{i/j}$ , then the adversary sends ALG to  $v_i$  and ALG will have to go to  $l_{i/j}$  by [Observation 3.2](#) and then either color  $l_{i/j}$  or be caught in an endless loop. Hence, ALG colors  $l_{i/j}$  and then goes to one of its neighbors. The adversary is going to send ALG to  $v_i$ .

When ALG arrives in  $v_i$ , the closed neighborhood  $N[v_i]$  is colored. If ALG decides to go to  $l_{i/j}$ , it will then again be sent to  $v_i$  and will never terminate. If ALG decides to go to  $v_{i-1}$ , it will never visit any leaf between  $v_i$  and  $v_j$ . If ALG decides to go to  $v_{i+1}$ , then ALG can never terminate since all paths from  $v_{i+1}$  to  $v_1$  lead through  $v_i$  and from there, ALG always chooses  $v_{i+1}$ . Therefore, no matter how ALG behaves, it cannot successfully explore  $G_{n_0,i,j}$ , unless it uses at least  $n_0 - 1$  colors on the first  $n_0 - 1$  steps.



**Figure 3.4.** The graph  $G_1$  for the general lower bound

Consider now the graph  $G_1$  depicted in [Figure 3.4](#).  $G_1$  consists of a path of  $n_0 - 1$  vertices that are connected to a universal vertex  $u$ . Locally,  $G_1$  looks exactly the same as  $G_{n_0,i,j}$ . An algorithm can only successfully explore  $G_{n_0,i,j}$  if it assigns  $n_0 - 1$  colors on the first  $n_0 - 1$  steps during the exploration of  $G_{n_0,i,j}$ . Then, however, such an algorithm assigns  $n_0 - 1$  colors on the first  $n_0 - 1$  steps during the exploration of  $G_1$  as well.  $G_1$  has  $n_0$  vertices; hence, we have discovered a graph on  $n_0$  vertices where ALG uses at least  $n_0 - 1$  colors. This contradicts our assumption and thus finishes the proof.  $\square$

### Non-Uniform Algorithms

Clearly, the proof of [Theorem 3.4](#) relies heavily on the uniformity of the algorithm, that is, the property that the algorithm has to successfully explore every graph. If an algorithm just has to explore all graphs of a fixed size  $n$  or up to a fixed size  $n$ , then the situation is different. We cannot argue anymore that an algorithm needs to use many colors on a certain small graph by claiming that the algorithm has to use many colors to successfully explore a certain large graph and showing that it is not possible to distinguish the large graph from the small graph. Instead, we have to assume that the algorithm knows  $n$  because we want to prove a property for all possible algorithms.

We want to explore this restriction in order to know how useful it is to know the size of the graph in our model. We provide almost matching upper and lower bounds for this case in the next two theorems.

**Theorem 3.5.** *For every  $n \geq 5$  there is an algorithm that successfully explores all graphs of size exactly  $n$  and that uses on each such graph at most  $n - 2$  colors.*



*Proof.* Let  $n \in \mathbb{N}$ ,  $n \geq 5$ . Our algorithm  $\text{DFS}_n$  works as follows. In general,  $\text{DFS}_n$  works exactly as  $\text{DEPTHFIRSTSEARCH}$ . However, we want to assign to the last two vertices color  $n - 2$ ; therefore, we need a few adaptations that we will describe in a moment.

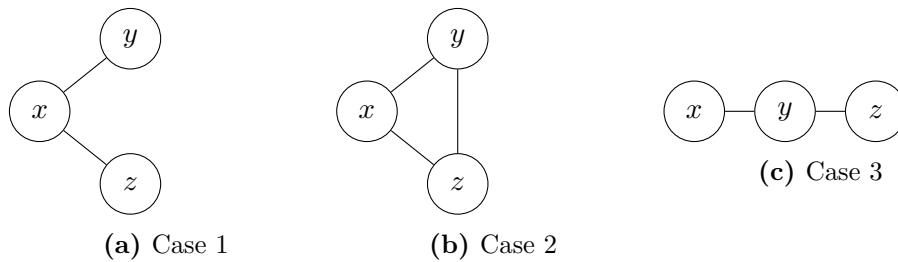
Denote the last three vertices that  $\text{DEPTHFIRSTSEARCH}$  colors by  $x$ ,  $y$ , and  $z$  (in this order). As long as the agent is not on an uncolored vertex where there is a neighbor with color  $n - 2$ ,  $\text{DFS}_n$  works exactly as  $\text{DEPTHFIRSTSEARCH}$ . Therefore, as long as  $\text{DFS}_n$  is not on one of the vertices  $y$  or  $z$ , the behavior of  $\text{DFS}_n$  is exactly like the behavior of  $\text{DEPTHFIRSTSEARCH}$  and there is always a unique predecessor that can be recognized by the agent.

The last three vertices can be arranged in three different ways, as illustrated in Figure 3.5. We adapt  $\text{DEPTHFIRSTSEARCH}$  as follows:

**Rule 1** If the current vertex is uncolored and there is exactly one neighbor with color  $n - 2$ , then  $\text{DFS}_n$  assigns *the smallest color for which there is no neighbor with color number exactly 1 larger*<sup>4</sup>—which we will call *smallest unproblematic color*—and continues to an uncolored neighbor. If there is no such uncolored neighbor, then the agent returns to the neighbor with color  $n - 2$ .

**Rule 2** If the current vertex is uncolored and there are two neighbors with color  $n - 2$  (in this case,  $\text{DFS}_n$  is on  $z$  and just has to go back to the start vertex), then  $\text{DFS}_n$  assigns the smallest unproblematic color and then continues to a vertex of minimal color.

**Rule 3** If the current vertex  $v$  is colored with  $c(v) > 1$  and there is neither an uncolored neighbor nor a neighbor  $v'$  with  $c(v') = c(v) - 1$ , then  $\text{DFS}_n$  goes to the neighbor with the largest color.



**Figure 3.5.** The three cases how  $x$ ,  $y$ , and  $z$  can be connected

For the behavior of  $\text{DFS}_n$  on  $y$  and  $z$ , we have the following cases depending on which color  $x$  receives and how the vertices  $x$ ,  $y$ , and  $z$  are connected to each other. We have three interesting cases and two boring cases and we start our analysis with the two boring cases.

First, consider the case where  $x$ , which is the third-last vertex to be colored, receives color  $n - 4$  or less. Thus,  $\text{DEPTHFIRSTSEARCH}$  never assigns a color larger

<sup>4</sup>For example, unless  $y$  is connected to the vertex 2,  $y$  is assigned color 1; otherwise, unless  $y$  is connected to both the vertex 2 and the vertex 3,  $y$  is assigned color 2 etc. Note that this excludes  $y$  receiving color  $n - 3$  since it is connected to  $x$ . Furthermore, note that  $y$  might be assigned color  $n - 2$ .



than  $n - 2$  and never arrives at a vertex already colored with  $n - 2$ . In this case,  $\text{DFS}_n$  behaves exactly as  $\text{DEPTHFIRSTSEARCH}$  and uses at most  $n - 2$  colors as well.

In the second case,  $x$  receives color  $n - 3$ . Thus,  $\text{DEPTHFIRSTSEARCH}$  may assign to  $y$  color  $n - 2$  and to  $z$  color  $n - 1$ .  $\text{DFS}_n$  assigns to  $y$  color  $n - 2$  as well, but when it arrives on  $z$  and sees exactly one vertex of color  $n - 2$ ,  $\text{DFS}_n$  uses Rule 1 and assigns the smallest unproblematic color and then goes to the vertex with color  $n - 2$ , which is  $y$ . On  $y$ ,  $\text{DFS}_n$  sees that  $y$  was colored in an earlier step and that there are no more uncolored neighbors and thus applies the normal backtracking rule to go back to the vertex with color  $n - 3$ , which is  $x$ . From there, the behavior of  $\text{DFS}_n$  is the same as the behavior of  $\text{DEPTHFIRSTSEARCH}$ .

Note that the color of  $z$  cannot interfere with the behavior of  $\text{DEPTHFIRSTSEARCH}$  since  $\text{DEPTHFIRSTSEARCH}$  always goes back to the direct predecessor, which is a vertex with color number exactly 1 less.

We can now move towards the interesting cases. In all of the remaining three cases,  $x$  receives color  $n - 2$ . The cases are illustrated in [Figure 3.5](#).

We describe the behavior of  $\text{DFS}_n$  on each of these cases and thus show that  $\text{DFS}_n$  can distinguish these cases and can consequently successfully explore graphs of size  $n$  with  $n - 2$  colors.

**Case 1:**  $x$  receives color  $n - 2$ ;  $x$  is adjacent to both  $y$  and  $z$ ; and  $y$  and  $z$  are not adjacent, see [Figure 3.5](#). In this case,  $y$  is going to be assigned by  $\text{DFS}_n$  the smallest unproblematic color and then the agent returns to  $x$  (Rule 1). From there, the agent visits  $z$ , which is the remaining uncolored vertex. On  $z$ ,  $\text{DFS}_n$  is in the same situation as it was on  $y$  (exactly one neighbor with color  $n - 2$ ) and is going to assign the smallest unproblematic color to  $z$  and then return to  $x$  by Rule 1. Now,  $\text{DFS}_n$  behaves again exactly the same as  $\text{DEPTHFIRSTSEARCH}$ .

**Case 2:**  $x$  receives color  $n - 2$  and  $x, y, z$  form a triangle. Again,  $y$  receives the smallest unproblematic color by Rule 1. Then, by Rule 1 as well, the agent does not return to  $x$  but continues to the next unvisited vertex, which is  $z$ . On  $z$ , we have two cases.

**Subcase 2.1:** If  $y$  received a color different from  $n - 2$ , then the agent is in the same situation as before (exactly one neighbor with color  $n - 2$ ) and assigns the smallest unproblematic color and returns to  $x$ . From there,  $\text{DFS}_n$  behaves again exactly the same as  $\text{DEPTHFIRSTSEARCH}$  and terminates as expected.

**Subcase 2.2:** If  $y$  received color  $n - 2$ , the situation is more delicate. The agent now sees two vertices colored  $n - 2$  and if it would assign color  $n - 2$  and go to  $y$ , it might be caught in an endless loop and go back and forth between  $y$  and  $z$ . Hence, by Rule 2, the agent assigns the smallest unproblematic color and then goes to the smallest color it sees. If the smallest unproblematic color is  $k \geq 2$ , then  $z$  is connected with vertices  $2, \dots, k$  and the agent goes directly to the start vertex if  $z$  is connected to the start vertex or the agent goes to vertex 2 and from there to the start vertex. If the smallest unproblematic color is 1, this means that  $z$  is not a neighbor of 2, in which case the agent goes to any vertex with minimal

color. If this vertex is  $y$ , then the agent applies Rule 3 to go to  $x$  and from there with normal backtracking to the start vertex. If this vertex is not  $y$ , then the agent uses normal backtracking to the start vertex right away. Since  $z$  did not receive a color that interferes with backtracking,  $\text{DFS}_n$  behaves exactly as  $\text{DEPTHFIRSTSEARCH}$ .

**Case 3:**  $x$  receives color  $n - 2$  and  $x, y, z$  form a simple path. Again,  $y$  receives the smallest unproblematic color and the agent then continues to  $z$ . We have to distinguish two cases.

**Subcase 3.1:** If  $y$  was assigned color  $n - 2$ , then  $z$  receives the smallest unproblematic color as in Cases 1 and 2 by Rule 1 and the agent then goes back to  $y$ . From there, the agent behaves again exactly as  $\text{DEPTHFIRSTSEARCH}$  and goes to vertex  $n - 3$  by the normal backtracking rule. Note that  $y$  is in this case connected to vertex  $n - 3$  since  $y$  received  $n - 2$  as smallest unproblematic color.

**Subcase 3.2:** If  $y$  is assigned something less than  $n - 2$ , then  $z$  is assigned the largest number the agent sees as is the case with  $\text{DEPTHFIRSTSEARCH}$  and then goes to the vertex with this largest number (see line 4 of Algorithm 2). If this vertex is not  $y$ , we are done since then there is always a clear predecessor until the start vertex is reached. If this vertex is  $y$ , however, Rule 3 will be applied and the agent goes from  $y$  to  $x$  and then backtracks from there to the start vertex.

We see that no matter how the last three vertices are arranged,  $\text{DFS}_n$  successfully explores the graph. This finishes the proof.  $\square$

**Theorem 3.6.** *For every algorithm ALG and for every natural number  $n$  such that ALG explores all graphs of size  $n$  there is a graph  $G$  of size  $n$  such that ALG uses at least  $n - 3$  colors on  $G$ .*

*Proof.* The theorem is clearly true for  $n \leq 5$  by Lemma 3.8: Since we showed that at least two colors are needed to explore all graphs of size 3, two colors are needed to explore all graphs of size 4 and 5 as well. To see this, just assume that there is an additional leaf (for graphs of size 4) or an additional path with two vertices (for graphs of size 5) attached to the start vertex of the triangle in the proof of Lemma 3.8.

Assume now towards contradiction that there is a natural number  $n \geq 6$  and an algorithm ALG that explores every graph of size  $n$  with at most  $n - 4$  colors. Consider the following  $\binom{n-3}{2}$  graphs of size  $n$ : All cliques of size  $n - 3$  where two vertices  $v_i$  and  $v_j$  both have a leaf and the remaining vertices are all connected to some special vertex  $s$ . Note that every vertex in the clique has degree  $n - 3$  and the vertices  $v_i$  and  $v_j$  thus cannot be distinguished from the other vertices in the clique. The special vertex  $s$  has degree  $n - 5$ .

Note that we can use Lemma 3.6 and Lemma 3.7 as long as there are enough unvisited vertices available such that their proofs work. In case of Lemma 3.6, we need three unvisited vertices; in case of Lemma 3.7, we need two unvisited vertices including the current vertex. Therefore, by Lemma 3.6 and Lemma 3.7, an adversary can ensure that ALG first visits all vertices in the clique and then goes to  $s$ .

Now, ALG will have assigned to two vertices the same color. Consider the exploration of ALG on the graph where these two vertices are the two vertices with the attached leaf,  $v_i$  and  $v_j$ . Note that here, we need the fact that all vertices in the clique have the same degree. Otherwise, ALG might recognize  $v_i$  and  $v_j$  and ensure that these two always receive a different color. After visiting all vertices in the clique, ALG has to visit the first leaf at some point. Say the agent first visits the leaf of  $v_i$ . If ALG does not color the leaf, then it is caught in an endless loop. Hence, ALG colors the leaf and then goes back to  $v_i$ . From there, ALG has to visit  $v_j$  at some point.

Before visiting  $v_j$ , ALG may visit  $s$ , but whenever the agent tries to reach  $v_j$ , the adversary can send the agent to  $v_i$ . The only way to reach  $v_j$  is by going directly from  $v_i$  to  $v_j$ . Once the agent does this, it can then visit the leaf of  $v_j$ , color it with the same color as the leaf of  $v_i$  due to its functional nature, and then return to  $v_j$ . Now, the agent is in the same situation as before when it went from  $v_i$  to  $v_j$ . Therefore, the agent will now go to  $v_i$  and from there again to  $v_j$  and is hence caught in an endless loop. This contradicts our assumption and thus finishes the proof.  $\square$

### 3.4 Between Trees and General Graphs

We want to discover the parameter that determines the difficulty of graph exploration of a given graph. Since the exploration of trees is quite easy, it is tempting to think that a good parameter to measure the difficulty for graph exploration could be treewidth, feedback vertex set, number of cycles, etc. However, considering the difficult instance in [Figure 3.3](#), we see that treewidth and feedback vertex set are not suitable parameters. In fact, even restricting the number of cycles to 1 and the maximum degree to 3 does not remove the linear amount of colors. Bipartite graphs as generalizations of trees might still serve as plausible candidates. After all, the proof of [Lemma 3.7](#) does not work for bipartite graphs and the graphs  $G_{n_0,i,j}$  used in the construction for [Theorem 3.4](#) are not bipartite if  $j - i$  is odd. However, the following theorem shows how these defects can be remedied; in fact, only exploring bipartite graphs is almost as difficult as exploring all graphs.

**Theorem 3.7.** *There is no algorithm ALG and no natural number  $n$  such that ALG successfully explores all bipartite graphs and uses strictly fewer than  $n - 2$  colors on every graph on  $n$  vertices.*

*Proof.* Assume towards contradiction that such an algorithm ALG and such a number  $n_0$  exists. As in the proof of [Theorem 3.4](#), we let ALG walk on graphs of the family  $G_{N,i,j}$  with  $j - i$  even and for  $N = 3n_0$ . By [Lemma 3.6](#), we can ensure that ALG walks on the path without exploring any leaves. Since ALG uses at most  $n_0 - 2$  colors on every graph of size  $n_0$ , ALG uses a color twice on its first  $n_0 - 1$  steps, say  $c(v_l) = c(v_m)$  for  $l < m$ . Then, however, ALG is going to repeat its entire pattern after  $v_l$ , so for all  $k \in \mathbb{N}^+$ :

$$c(v_{l+k}) = c(v_{m+k}).$$

Since our graph is large enough, we can ensure that this pattern is repeated at least three times. This means that we can find two vertices of the same color that are in the same partition. We want to apply the reasoning of [Theorem 3.4](#) and

hence connect the vertices as follows to ensure that the graph stays bipartite: If the distance  $(m-l)$  of  $v_l$  and  $v_m$  is even, we define  $v_i := v_{l+1}$  and  $v_j := v_{m+1}$  and connect these vertices via a leaf  $l_{i/j}$ . If the distance  $(m-l)$  is odd, we define  $v_i := v_{l+1}$  and  $v_j := v_{m+1+(m-l)}$  and connect them via a leaf  $l_{i/j}$ .

Now, in order to apply the same reasoning as in the proof of [Theorem 3.4](#), we have to guarantee that the entire neighborhood of  $v_i$  and  $v_j$  is colored once ALG visits and colors  $l_{i/j}$ . To ensure this, we prove that no vertex in the repeating pattern  $v_l, \dots, v_{m-1}$  is left uncolored: Assume that any vertex in the repeating pattern would be left uncolored, say  $c(v_{m-1}) = 0$ . Clearly, ALG cannot leave two consecutive vertices uncolored; otherwise, ALG would leave all future vertices on the path uncolored. Therefore,  $v_{m-2}$  is colored and we denote its color by  $x$ . On vertex  $v_{m-1}$ , ALG received as input  $(0, x, 0, 0)$  since there are two uncolored neighbors and one neighbor with color  $x$ . If  $m-l$  is even, connect vertex  $v_{m+(m-l)}$  with  $v_{m-2}$  instead of  $v_{m+1+(m-l)}$ . If  $m-l$  is odd, connect  $v_{m+2(m-l)}$  with  $v_{m-2}$  instead of  $v_{m+1+2(m-l)}$ . Assume without loss of generality that  $m-l$  is even and observe what happens when the algorithm reaches  $v_{m-1+(m-l)}$ : This vertex is left uncolored as  $v_{m-1}$  and the algorithm then visits  $v_{m+(m-l)}$ . Upon arrival in  $v_{m+(m-l)}$ , the algorithm receives  $(0, x, 0, 0)$  as input and therefore decides not to color  $v_{m+(m-l)}$  and instead goes to an uncolored neighbor. The adversary arranges the port labels such that  $v_{m-1+(m-l)}$  is chosen. Now, ALG is in an endless loop. Therefore, all vertices of the repeating pattern receive a color.

Hence, the entire neighborhood of  $v_i$  and  $v_j$  is colored once ALG visits  $l_{i/j}$  and we can now apply the same reasoning as in the proof of [Theorem 3.4](#) to conclude that ALG may never repeat its pattern. However, this time, the smallest bipartite graph that looks locally like some  $G_{n_0, i, j}$  for the first  $n_0 - 1$  steps is not exactly  $G_1$  from [Figure 3.4](#), but  $G_{\text{bipartite}}$ , where  $u$  is only connected to all odd vertices  $v_h$  and an additional vertex  $u_{\text{even}}$  is connected to all even path vertices, leading to a lower bound of  $n - 2$ .  $\square$

We now argue why the *circumference* of a graph, i.e., the size of a longest simple cycle, is the right parameter. First, we observe that the construction from the proof of [Theorem 3.4](#) immediately results in the following corollary:

**Corollary 3.1.** *Any algorithm needs at least  $k - 2$  colors to color all graphs of circumference at most  $k$ .*

Second, we present with [Algorithm 3](#) an algorithm that uses at most  $2k - 1$  colors and successfully explores all graphs of circumference at most  $k \geq 3$ . This algorithm sorts numbers  $a, b$  with respect to whether the distance between  $a, \dots, b$  or the distance between  $b, b + 1, \dots, 2k - 1, 1, 2, \dots, a$  is larger. We use the following maximum function for three natural numbers  $a, b, k$  with  $a \leq b \leq 2k - 1$ :

$$\max^k\{a, b\} := \begin{cases} a, & \text{if } |b - a + 1| < |2k - b + a| \\ b, & \text{if } |b - a + 1| > |2k - b + a|. \end{cases}$$

Note that, if the two values  $(b - a + 1)$  and  $(2k - b + a)$  were equal, this would imply  $2(b - a) = 2k - 1$ , which is impossible for three natural numbers  $a, b$  and  $k$ . In other words,  $\max^k$  is well-defined. With this maximum function, that is, calculating modulo  $2k - 1$  shifted by 1, a number is smaller than the next  $k - 1$  numbers and larger

than the preceding  $k - 1$  numbers. For example, for  $k \leq 11$ ,  $2k - 10$  is smaller than  $2k - 9, \dots, 2k - 1, 1, \dots, k - 10$  and larger than  $2k - 11, \dots, 2k - 10 - (k - 1) = k - 9$ .

We want to extend  $\max^k$  to multiple values  $a_1 \leq \dots \leq a_n \in [2k - 1]$ , i.e., we would like to define  $\max^k\{a_1, \dots, a_n\}$ . However, it is not immediately clear how to do that since  $\max^k$  is not necessarily transitive. For example,  $\max^k\{1, 2\} = 2$  and  $\max^k\{2, k + 1\} = k + 1$ , but still  $\max^k\{1, k + 1\} = 1$ . Thus,  $\max\{1, 2, k + 1\}$  could be either 1, 2 or  $k + 1$ . This problem vanishes if there is an input element (for the maximum function)  $a_i$ ,  $i \in [n]$  such that all other input elements  $a_j$  are among the  $k - 1$  elements preceding  $a_i$ . It will become clear later that the construction of algorithm  $\text{SMALLDFS}_k$  enforces this situation. In this case, we use the natural extension of  $\max^k$ ; otherwise, we define  $\max^k$  to be just the normal maximum function. It will become clear later that  $\max^k$  is not going to be the “normal maximum function.”

---

**Algorithm 3**  $\text{SMALLDFS}_k$

---

**Input:** A graph whose circumference is at most  $k$ . In each step, the input is  $c(v) \in \mathbb{N}$ , the color of the current vertex  $v$ , and  $c(v_1), \dots, c(v_d)$ , the colors of  $v$ 's neighbors, where  $d$  is the degree of  $v$ .

**Output:** In each step, the algorithm outputs  $c(v)$ , the color the agent assigns to  $v$ , and  $i \in [d]$ , the vertex the agent has to move to next.

```

1: if  $c(v) = 0$  then
2:    $c(v) := \left( \left( \max_{v' \in N(v)}^k c(v') \right) + 1 \right) \bmod_1 (2k - 1)$ 
3: if there is an uncolored neighbor  $w$  then
4:   go to  $w$ 
5: else if there is a neighbor  $w$  with  $c(w) = (c(v) - 1) \bmod_1 (2k - 1)$  then
6:   go to  $w$ 
7: else
8:   terminate

```

---

The strategy of  $\text{SMALLDFS}_k$  is a depth-first search strategy that assigns color  $c \bmod_1 (2k - 1)$  to vertices that are visited in distance  $c - 1$  (in the depth-first search tree) from the start vertex. This is done by looking at all neighbor colors and then assigning the largest color modulo  $(2k - 1)$  to the current vertex. However, in order to be able to assign the color 1 after the color  $2k - 1$ , the algorithm takes the maximum function defined above to determine the largest color number. A problem might arise if the maximum function were not well-defined. However, since there are no cycles of length  $k + 1$ , the colors of the current vertex and all its neighbors are always going to be from  $k$  sequential numbers in the number range  $1, \dots, 2k - 1, 1, \dots, k$ . Thus, it is possible, as we are going to prove in a moment, to determine the direct predecessor and backtrack accurately.

**Lemma 3.9.**  $\text{SMALLDFS}_k$  *terminates*.

*Proof.* We prove that there are no endless loops, i.e., there is no sequence of vertices  $v_1, v_2, \dots, v_l, v_{l+1} = v_1$  such that  $\text{SMALLDFS}_k$  always goes from  $v_i$  to  $v_{i+1} \bmod_1 (2k-1)$ . Assume towards contradiction that such a sequence exists. If  $\text{SMALLDFS}_k$  arrives at

a vertex that has been colored before, it goes to an uncolored neighbor regardless of the color of any other vertices. Therefore, at some point, there will be no more uncolored neighbors of the vertices in the loop. In other words, the only line that can be applied is line 6. However, to obtain an endless loop, the colors in the loop would then have to contain the sequence  $(1, 2, \dots, 2k - 1)$ . This is a contradiction to the assumption that the graph has a circumference of at most  $k$ .  $\square$

**Lemma 3.10.** *SMALLDFS<sub>k</sub> terminates on the start vertex.*

*Proof.* Since every vertex  $v$  except the start vertex has a neighbor  $w$  with  $c(w) = c(v) - 1 \pmod{2k - 1}$  because of line 2, SMALLDFS<sub>k</sub> cannot terminate on a vertex that is not the start vertex. Since SMALLDFS<sub>k</sub> terminates due to Lemma 3.9, it has to terminate on the start vertex.  $\square$

We proved that SMALLDFS<sub>k</sub> terminates on the start vertex and only on the start vertex. We have yet to show that all vertices are indeed explored.

**Lemma 3.11.** *On any vertex  $v$ , either SMALLDFS<sub>k</sub> goes to an unvisited vertex or it goes back to the direct predecessor of  $v$ .*

*Proof.* Similar to the proof of Lemma 3.3, we prove something slightly different, namely that (1) SMALLDFS<sub>k</sub> always assigns color  $c(v') + 1 \pmod{2k - 1}$  to a vertex  $v$  with predecessor  $v'$  and that (2) there is never a vertex  $v$  with two neighbors  $w_1, w_2$  with  $c(w_1) = c(w_2) = c(v) - 1 \pmod{2k - 1}$ . This implies that SMALLDFS<sub>k</sub> always goes back to the direct predecessor.

Assume towards contradiction that one of these two claims is wrong. Consider the first step in which one of these claims is wrong. We have three cases:

**Case 1** In this step, SMALLDFS<sub>k</sub> does not assign color  $c(v') + 1 \pmod{2k - 1}$  to a vertex  $v$  with predecessor  $v'$ . This can only happen if there is a neighbor  $x$  of  $v$  with greater color than  $v'$ , that is,  $\max^k\{c(x), c(v')\} = c(x)$ .

Both  $x$  and  $v'$  are colored before  $v$ . Assume as subcase 1.1 that  $x$  is colored before  $v'$ . Consider the search sequence  $(x, v_1, \dots, v_n, v')$  of SMALLDFS<sub>k</sub> between  $x$  and  $v'$ . The first step when SMALLDFS<sub>k</sub> “skips” a color is between  $v'$  and  $v$ ; hence, the colors between vertices next to each other in this search sequence differ by exactly 1. Since  $c(x)$  is smaller than the next  $k - 1$  numbers and since we have  $\max^k\{c(x), c(v')\} = c(x)$ , there have to be at least  $k - 1$  vertices between  $x$  and  $v'$ . Thus, we have together with  $v$  a cycle of length at least  $k + 2$ , which is not allowed. Hence,  $x$  is not colored first.

Consider subcase 1.2, namely that  $v'$  is colored before  $x$ . Then SMALLDFS<sub>k</sub> went from  $v'$  to  $x$ , then back to  $v'$  and then to  $v$ . However, this is not possible since backtracking (applying line 6) is only allowed when there are no uncolored neighbors and thus, SMALLDFS<sub>k</sub> cannot backtrack from  $x$ , which has  $v$  as an uncolored neighbor.

**Case 2** In this step, SMALLDFS<sub>k</sub> assigns color  $c(v)$  to a vertex  $v$  with two neighbors  $w_1, w_2$  with  $c(w_1) = c(w_2) = c(v) - 1 \pmod{2k - 1}$ . Assume without loss of generality that first  $w_1$  is visited (and colored), then  $w_2$ , then  $v$ . We can argue exactly as before. The search sequence of SMALLDFS<sub>k</sub> between  $w_1$  and  $w_2$  is

$(w_1, v_1, \dots, v_r, w_2)$ . SMALLDFS $_k$  could not backtrack from  $w_1$  and thus went to some uncolored vertex  $v_1$ . Again, backtracking would only be possible up to  $w_1$ ; therefore, we can assume without loss of generality that all vertices  $v_1, \dots, v_r$  in the search sequence are visited for the first time. In order for  $w_2$  to obtain the same color as  $w_1$ , there have to be at least  $2k - 2$  vertices between  $w_1$  and  $w_2$ . Since  $w_1$  and  $w_2$  are both adjacent to  $v$ , this results in a cycle of size  $2k + 1$ .

**Case 3** In this step, SMALLDFS $_k$  assigns color  $c(w_1)$  to a vertex  $w_1$  adjacent to a vertex  $v$  with  $c(w_1) = c(v) - 1 \pmod{2k - 1}$  and  $v$  is adjacent to a vertex  $w_2 \neq w_1$  with  $c(w_2) = c(w_1)$ . Consider the search sequence  $(v, \dots, w_1)$  between  $v$  and  $w_1$ . Since  $w_1$  receives a greater number than  $v$ , there have to be at least  $k - 1$  vertices between  $v$  and  $w_1$ . Together with  $v$  and  $w_1$ , they form a cycle of size  $k + 1$ , which is impossible.

We see that the assumption that one of these two claims is wrong leads to a contradiction; therefore, both claims are true, which proves the lemma.  $\square$

**Lemma 3.12.** SMALLDFS $_k$  explores all vertices.

*Proof.* Since Lemma 3.11 is the analogue to Lemma 3.3, this claim can be proved analogously to Lemma 3.4.  $\square$

These lemmas together yield the following theorem.

**Theorem 3.8.** SMALLDFS $_k$  uses at most  $2k - 1$  colors and is correct on graphs with circumference at most  $k$ .

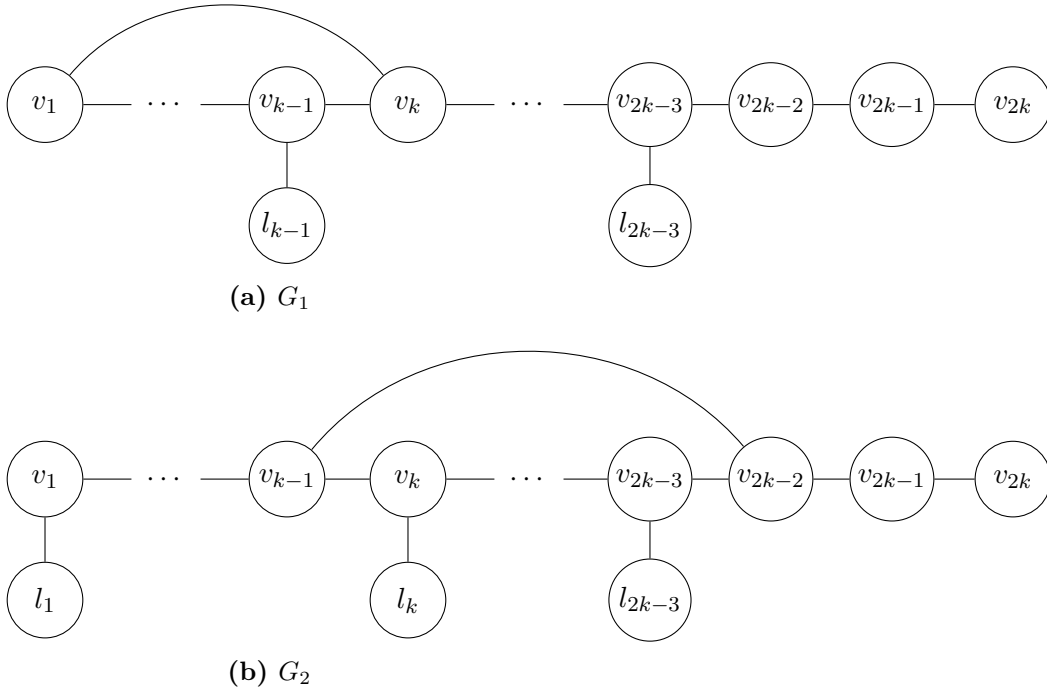
Contrasting Corollary 3.1 and Theorem 3.8, we see that SMALLDFS $_k$  uses at most  $k + 1$  too many colors. We narrow this gap with the following theorem.

**Theorem 3.9.** An algorithm needs at least  $2k - 3$  colors to successfully color all graphs of circumference at most  $k$ .

*Proof.* Let ALG be an algorithm that uses at most  $2k - 4$  colors and colors every graph of circumference  $k$ . Without loss of generality, assume that ALG uses exactly  $2k - 4$  colors. The colors ALG assigns have to repeat at some point, that is, if we let ALG explore a path  $v_1, \dots, v_r$  of length  $r > 2k - 4$ , after some initial color assignments  $c(v_1), c(v_2), \dots, c(v_s)$ , the colors ALG assigns always follow the same pattern  $c(v_s), c(v_{s+1}), \dots, c(v_t), c(v_s), c(v_{s+1})$  etc. Without loss of generality, we assume this pattern starts with  $v_1$ . This behavior can be enforced even if we add some leaves or connect some vertices of the path: Due to Lemma 3.6, a general algorithm may not go back to a visited vertex if there is an unvisited neighbor. Therefore, we may even add some leaves to the path or connect some vertices of the path and there is still an adversary that ensures that ALG goes from  $v_1$  directly to  $v_r$  without visiting any leaf (apart from  $v_1$  and  $v_r$ , of course) and without taking any shortcuts.

Consider now the following two graphs. Take a path  $v_1, \dots, v_{2k}$  of length  $2k - 1$  and add a leaf  $l_{2k-3}$  to  $v_{2k-3}$ . To obtain  $G_1$ , add a leaf  $l_{k-1}$  to  $v_{k-1}$  and connect  $v_1$  and  $v_k$ . To obtain  $G_2$  instead, add leaves  $l_1$  to  $v_1$  and  $l_k$  to  $v_k$  and connect  $v_{k-1}$





**Figure 3.6.** The graphs  $G_1$  and  $G_2$

to  $v_{2k-2}$ .  $G_1$  and  $G_2$  are depicted in [Figure 3.6](#). Note that both  $G_1$  and  $G_2$  have circumference exactly  $k$ .

Consider the exploration of ALG on  $G_1$ . As explained, there is an adversary that lets ALG walk from  $v_1$  to  $v_{2k-2}$  without taking the shortcut  $\{v_1, v_k\}$  and without exploring the leaves. Moreover, by assuming that the repeating pattern of ALG starts with  $v_1$ , we obtain in particular that  $c(v_1) = c(v_{2k-3})$ .

Now, on vertex  $v_k$ , ALG receives as input  $c(v_k) = 0$  and the colors  $c(v_1)$ ,  $c(v_{k-1})$ , and 0 and assigns color  $c(v_k) > 0$ . Then, ALG goes on to explore the rest of the graph. At some point, ALG returns to  $v_k$  and receives as input  $c(v_k)$  and the colors  $c(v_1)$ ,  $c(v_{k-1})$ , and  $c(v_{k+1})$ . Since  $l_{k-1}$  has not yet been explored and since upon going to  $v_1$ , ALG has to terminate, it is crucial that ALG chooses  $v_{k-1}$  as its next vertex.

However, consider now the exploration of ALG on  $G_2$ . Since  $G_2$  looks locally exactly like  $G_1$ , there is an adversary that lets ALG behave on  $G_2$  exactly as on  $G_1$  for at least the first  $2k-3$  steps. On vertex  $v_{2k-2}$ , ALG receives as input  $c(v_{2k-2}) = 0$  and the colors  $c(v_{2k-3}) = c(v_1)$ ,  $c(v_{k-1})$ , and 0. This is the same input as before and because of its functional nature, ALG has to assign color  $c(v_{2k-2}) = c(v_k) > 0$  and then has to go to the unvisited neighbor  $v_{2k-1}$ . On  $v_{2k-1}$ , ALG is in the same situation as during the exploration of  $G_1$  on  $v_{k+1}$  and assigns color  $c(v_{2k-1}) = c(v_{k+1})$ . After then visiting  $v_{2k}$ , ALG returns to  $v_{2k-2}$ . Now, its input is again  $c(v_k) = c(v_{2k-2})$  and the colors  $c(v_1) = c(v_{2k-3})$ ,  $c(v_{k-1})$ , and  $c(v_{2k-1}) = c(v_{k+1})$ .

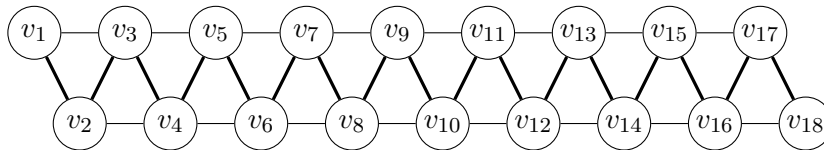
However, this time, since  $l_{2k-3}$  has not yet been explored, it is crucial that ALG chooses  $v_{2k-3}$  and not  $v_{k-1}$ . Since ALG is a function, it cannot output different values on the same input; hence, ALG fails to explore either  $G_1$  or  $G_2$ .  $\square$



### 3.5 Special Graph Classes

In the above, we found that trees cannot be explored with less than 3 colors, general graphs cannot be explored with less than  $n - 1$  colors and in general, the number of colors needed to explore a graph grows with its circumference. However, if we limit ourselves to special graph classes, we can find algorithms that use much fewer colors than the circumference of the graph.

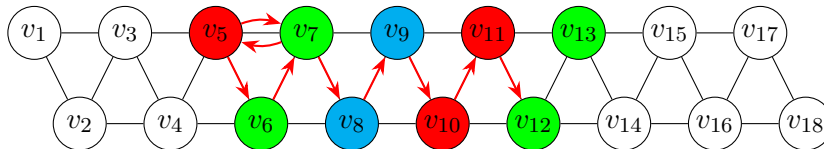
We illustrate this by taking the example of the graph class  $P^2$ , the *square paths*. The graph class  $P^2$  contains all graphs that consist of a path plus edges between all vertices in distance 2 from each other. An example is given in Figure 3.7. Note that



**Figure 3.7.** An example of a square path  $P^2$ . The original path is emphasized.

graphs in  $P^2$  have a circumference of size  $n$ . We show that 4 colors are enough to explore any graph in  $P^2$ .

The strategy of our algorithm, SQUAREPATHEXPLORATION, is similar to TREEEXPLORATION, namely to achieve a sense of direction by coloring the vertices alternately 1, 2, and 3. For example, if we say that color 1 is red, color 2 is green, and color 3 is blue, and given the graph depicted in Figure 3.8 with  $v_3$  as start vertex, the goal would be to color the vertices as depicted in the figure such that the colors alternate nicely.

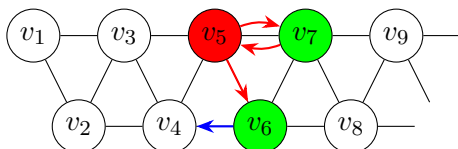


**Figure 3.8.** The desired structure. A possible walk of the agent is depicted with red arrows. The order at the beginning is  $v_5, v_7, v_5, v_6, v_7$ .

We have to be careful that the agent does not produce a triangle with colors 1, 2, and 3, thus being caught in an endless loop. The key idea to achieve this is by exploiting the structure of the square paths: Whenever an agent goes to a new vertex  $v$  and then sees only one colored neighbor  $w$ —thus, in general, three uncolored neighbors—, it can go back to  $w$ , which then has another uncolored neighbor that is a neighbor of  $v$ . Note that since the algorithm does not have to successfully explore all graphs, but only all square paths, returning back to a visited vertex is easily possible.

However, we have to consider some special cases. If we observe the walk of the agent in Figure 3.8, we notice two things. First, the beginning is quite peculiar. The agent starts in  $v_5$  and colors it red. Then it visits an uncolored neighbor,  $v_7$ , colors it green and since there is only one colored neighbor, the agent goes back to this neighbor. Back on  $v_5$ , the agent recognizes that it has been in this vertex before

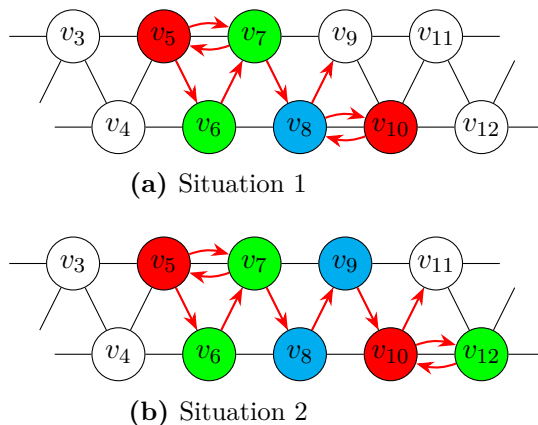
since  $v_5$  is colored. Therefore, it does not go back to the only visited—and thus colored—neighbor, but instead goes to an unvisited neighbor, say  $v_6$ . On  $v_6$ , there are two colored neighbors and usually, the agent would go forward to an unvisited vertex in such a case. However, then, it might happen that the vertices on one side of the start vertex are not colored at all. Such a situation is depicted in Figure 3.9.



**Figure 3.9.** Special case at the beginning: If the agent went from  $v_6$  to  $v_4$ , as depicted with the blue arrow, then  $v_8, v_9, \dots$  might never be explored.

Therefore, the agent has to go back to the vertex with color green—i.e., color 2—if there are two colored neighbors with colors 1 and 2. Intuitively speaking, this ensures that the agent first explores all vertices on the left (right) side of the start vertex if there are two vertices with color 2 on the left (right) side first.

The second thing we notice in Figure 3.8 is that the adversary might send the agent to different vertices. We argue heuristically why it is reasonable to use a fourth color in this case. The adversary might send the agent to a vertex with only one colored neighbor, consider for example the situations depicted in Figure 3.10.

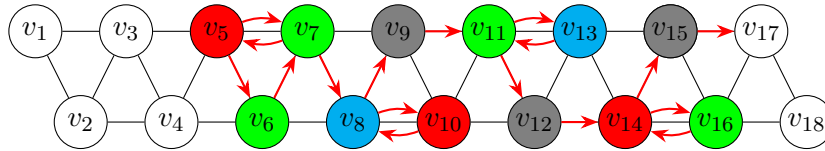


**Figure 3.10.** Cases where the agent sees all three colors at once. The agent cannot distinguish the two cases.

In Situation 1, the adversary sends the agent from  $v_8$  to  $v_{10}$  instead of  $v_9$ . The agent colors  $v_{10}$  red as before and then returns to  $v_8$ . From  $v_8$ , the agent goes to  $v_9$ . Now, the agent sees all three colors at once. This is tricky since the agent cannot distinguish this situation from Situation 2 in Figure 3.10, but the color which is farthest from the start vertex is different and the sense of direction might become lost if the agent assigns the wrong color.

In Situation 2, the adversary sends the agent from  $v_{10}$  to  $v_{12}$  instead of  $v_{11}$  and the agent then colors  $v_{12}$  green, goes back to  $v_{10}$  and from there to  $v_{11}$ . As in Situation 1, the input for the algorithm is  $c(v) = 0$  for the color of the current vertex and  $\{1, 2, 3, 0\}$  for the colors of the neighboring vertices. Hence, on  $v_9$  and given as input

all three colors red, green, and blue, the agent is wise not to assign any of the three normal colors. Instead, the agent is going to assign a new color—say gray—and then goes to the only unvisited neighbor, which is  $v_{11}$ . This way, the algorithm ensures that there is always a clear direction away from and towards the start vertex. Thus, the actual coloring pattern might not look like the one in Figure 3.8 since it depends on the decisions of the adversary. For example, the coloring pattern might even look like the one in Figure 3.11.



**Figure 3.11.** The structure with many vertices with color 4.

We have now covered the situation of the start vertex, where all neighbors are uncolored; the normal cases—namely exactly 1 colored neighbor, exactly 2 colored neighbors, and exactly 3 colored neighbors; and the special case at the beginning, depicted in Figure 3.9. There is one case left, namely when the agent is on a vertex of degree 2. This is easily handled, the only difference is that on such a vertex, the agent has to start going back towards the start vertex. The formal description of algorithm SQUAREPATHEXPLORATION is given in Algorithm 4.

We now turn towards the proof that SQUAREPATHEXPLORATION successfully explores all square paths. We start by proving that SQUAREPATHEXPLORATION is well defined.

**Lemma 3.13.** *SQUAREPATHEXPLORATION is well defined.*

*Proof.* This is clearly true for most of the instructions, we only have to be careful with the color assigning in line 18 and the conditions in lines 19 to 23 and lines 27 to 30. We check the color assigning in line 18 and the conditions in lines 19 to 23; the argumentation for lines 27 to 30 is similar. There, we use the evident fact that there can never be three neighbors of the same color.

The color assigning does not cover the case that the two colored neighbors are both colored with color 4. However, clearly, color 4 is only assigned if there are three colored neighbors of colors 1, 2, and 3; therefore, there are never two neighbors both with color 4. Given this color assignment, the conditions in lines 19 to 23 are clearly fulfilled since color 1 is only assigned if one of the neighbors has color 3 and similar for the other colors.  $\square$

Next, we show that SQUAREPATHEXPLORATION never colors a triangle with the colors 1, 2, and 3.

**Lemma 3.14.** *SQUAREPATHEXPLORATION never colors the three vertices of a triangle with the colors 1, 2 and 3, i.e., there are never three pairwise adjacent vertices  $x, y, z$  such that  $\{c(x), c(y), c(z)\} = \{1, 2, 3\}$ .*

*Proof.* Consider the step in which SQUAREPATHEXPLORATION colors the third vertex  $z$  of a triangle, i.e., the other two vertices  $x$  and  $y$  have been colored in

**Algorithm 4** SQUAREPATHEXPLORATION

**Input:** An undirected  $P^2$  graph in an online fashion. In each step, the input is  $c(v) \in \mathbb{N}$ , the color of the current vertex  $v$ , and  $c(v_1), \dots, c(v_k)$ , the colors of  $v$ 's neighbors, where  $k$  is the degree of  $v$ .

**Output:** In each step, the algorithm outputs  $c(v)$ , the color the agent assigns to  $v$ , and  $i \in [k]$ , the vertex the agent has to take next.

```

1: if  $c(v) = 0$  then
2:   if 0 neighbors are colored then
3:      $\triangleright$  Situation of start vertex
4:      $c(v) := 1$   $\triangleright$  Assign color 1 and then ...
5:     go to the uncolored neighbor that is presented first  $\triangleright$  ... go exploring.
6:   else if 1 neighbor  $w$  is colored then
7:      $\triangleright$  Normal case 1: Only 1 neighbor is colored
8:      $c(v) := (c(w) + 1) \bmod_1 3$   $\triangleright$  Assign the next color ...
9:     go to  $w$   $\triangleright$  ... and go back to this neighbor.
10:  else if the neighbors have colors  $\{1, 2, 0, 0\}$  then
11:     $\triangleright$  Special case to ensure a correct beginning
12:     $c(v) := 2$   $\triangleright$  Assign color 2...
13:    go to the neighbor with color 2  $\triangleright$  ... and go to  $w$ .
14:  else if 2 neighbors  $w, w'$  are colored then
15:     $\triangleright$  Normal case 2: 2 neighbors are colored
16:     $\triangleright$  Assign color and go forward, be careful not to create
17:     $\triangleright$  a triangle with colors 1,2,3
18:     $c(v) := \begin{cases} 1, & \text{if } \{c(w), c(w')\} \in \{\{1, 3\}, \{3, 3\}, \{3, 4\}\} \\ 2, & \text{if } \{c(w), c(w')\} \in \{\{1, 1\}, \{1, 2\}, \{1, 4\}\} \\ 3, & \text{if } \{c(w), c(w')\} \in \{\{2, 2\}, \{2, 3\}, \{2, 4\}\} \end{cases}$ 
19:    if there is an uncolored neighbor  $w$  then
20:      go to  $w$ 
21:    else
22:       $\triangleright$  Special case:  $v$  has degree 2 and both neighbors are colored
23:      go to a neighbor  $w$  with  $c(w) = (c(v) - 1) \bmod_1 3$ 
24:  else if 3 neighbors  $w, w', w''$  are colored then
25:     $\triangleright$  Normal case 3: 3 neighbors are colored
26:     $c(v) := \begin{cases} 4, & \text{if } \{c(w), c(w'), c(w'')\} = \{1, 2, 3\} \\ 1, & \text{if } \{c(w), c(w'), c(w'')\} \in \{\{1, 1, 3\}, \{1, 3, 3\}\} \\ 2, & \text{if } \{c(w), c(w'), c(w'')\} \in \{\{1, 1, 2\}, \{1, 2, 2\}, \\ & \quad \{1, 2, 4\}, \{1, 1, 4\}, \{1, 4, 4\}\} \\ 3, & \text{if } \{c(w), c(w'), c(w'')\} \in \{\{2, 2, 3\}, \{2, 2, 4\}, \{2, 3, 4\}, \{2, 4, 4\}\} \end{cases}$ 
27:    if there is an uncolored neighbor then
28:      go to some uncolored neighbor
29:    else
30:      go to  $z$  with  $c(z) = (c(v) - 1) \bmod_1 3$ 

```

---

```

31: else ▷ We have  $c(v) > 0$ 
32:   ▷ On a colored vertex, go forward if there are uncolored neighbors and
33:   ▷ go back towards the start vertex if there are no uncolored neighbors
34:   if there is an uncolored neighbor then
35:     go to some uncolored neighbor
36:   else if there is a neighbor  $z$  with  $c(z) = (c(v) - 1) \bmod_1 3$  then
37:     go to  $z$ 
38:   else
39:     terminate

```

---

earlier steps. Since at least two neighbors are colored, SQUAREPATHEXPLORATION applies one of the lines 12, 18, or 26 to color  $z$ . Clearly, it is never the case that SQUAREPATHEXPLORATION assigns color 1 and the colors 2 and 3 are present in the neighborhood. Analogously, SQUAREPATHEXPLORATION never assigns color 2 if colors 1 and 3 are present or color 3 if colors 1 and 2 are present.  $\square$

We are now prepared to prove that the algorithm always terminates.

**Lemma 3.15.** SQUAREPATHEXPLORATION *terminates*.

*Proof.* The statement is equivalent to saying that there are no endless loops, i.e., there is no sequence of vertices  $v_1, v_2, \dots, v_k, v_{k+1} = v_1$  such that SQUAREPATHEXPLORATION always goes from  $v_i$  to  $v_{(i+1) \bmod_1 k}$ . Assume towards contradiction that such a sequence exists.

If SQUAREPATHEXPLORATION arrives at a vertex that has been colored before, it goes to an uncolored neighbor regardless of the color of any other vertices. Therefore, there will be at some point no more uncolored neighbors of the vertices in the loop. In other words, the only line that can be applied is line 37. Therefore, by assuming without loss of generality that  $c(v_1) = 1$ , the colors in the sequence must be  $(c(v_1), c(v_2), c(v_3), c(v_4), \dots, c(v_k)) = (1, 2, 3, 1, \dots, 3)$ .

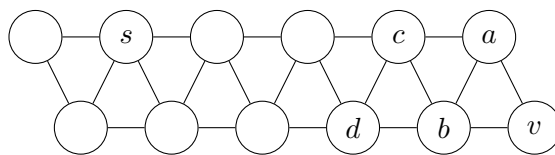
Consider the graph from Figure 3.7 and consider the rightmost vertex  $v$  that is in the endless loop. Say this vertex is  $v_i$ . This vertex has two neighbors  $v_{i-1}$  and  $v_{i+1}$  that are in the endless loop as well. Since we have  $(c(v_{i-1}), c(v_i), c(v_{i+1})) \in \{(1, 2, 3), (2, 3, 1), (3, 1, 2)\}$  as shown before, we have  $v_{i-1} \neq v_{i+1}$ . Moreover, since  $v_i$  is the rightmost vertex in the loop,  $v_{i-1}, v_i, v_{i+1}$  form a triangle. However, by Lemma 3.14, this cannot happen; hence, the assumption was wrong.  $\square$

**Lemma 3.16.** SQUAREPATHEXPLORATION *only terminates on the start vertex*.

*Proof.* SQUAREPATHEXPLORATION only terminates on a vertex  $v$  without uncolored neighbors and where no neighbor  $z$  has color  $c(z) = (c(v) - 1) \bmod_1 3$ .

On the one hand, since the start vertex receives color 1 and its neighbors color 2 or 4, these conditions are true for the start vertex as soon as all its neighbors are colored.

On the other hand, all other vertices are colored according to line 12, 18, or 26. We argue that there is always a neighbor with  $c(z) = (c(v) - 1) \bmod_1 3$ . Indeed, this is clear for all cases with the exception of the case when there are two or three colored neighbors, all with color 4. However, clearly, this can never happen as a 4



**Figure 3.12.** Add a vertex  $v$  in distance  $n$

is only assigned when there are three colored neighbors of colors 1, 2, 3. Thus, the claim follows.  $\square$

We proved that SQUAREPATHEXPLORATION terminates and that it terminates on the start vertex. We have yet to show that all vertices are indeed explored. We first show that SQUAREPATHEXPLORATION does not create a *hole*, i.e., there is never an uncolored vertex or a connected subgraph of uncolored vertices that is surrounded by colored vertices unless the uncolored vertex has degree 2.

Then, we show that, after some initial steps, all vertices on one *side* of the start vertex  $s$  are colored.<sup>5</sup> Together with the fact that SQUAREPATHEXPLORATION returns to the start vertex, the correctness of SQUAREPATHEXPLORATION follows.

**Lemma 3.17.** *Apart from vertices of degree 2 and 3, SQUAREPATHEXPLORATION never creates a hole.*

*Proof.* Consider the step in which SQUAREPATHEXPLORATION would create a hole. In this step, there is just one colored neighbor, but SQUAREPATHEXPLORATION would still go to an uncolored neighbor.  $\square$

**Lemma 3.18.** *As soon as the first two vertices on one side of the start vertex are colored, SQUAREPATHEXPLORATION explores all vertices on this side.*

*Proof.* Before we start, we observe that the two first vertices on one side of the start vertex that become colored will be vertices that are sequential in the path. In other words, if we draw the graph as for example in Figure 3.11, these first two vertices on one side of the start vertex will not be both “in the top row” or both “in the bottom row.”

Having observed this, we now prove the lemma by induction on the number of vertices on one side. If there are only two vertices on one side, there is nothing to prove. If there are three vertices on one side, we make use of the special case in lines 10 to 13 to ensure that the third vertex is colored as well. Now suppose the claim is true for  $n - 1$  vertices on one side. Add a vertex  $v = v_n$  and the according edges to this side. Let  $a$  and  $b$  be the two neighbors of  $v$  and let  $c$  be the common neighbor different from  $v$  of  $a$  and  $b$  and  $d$  the fourth neighbor of  $b$ . This situation is depicted in Figure 3.12.

Note that  $d$  might be the start vertex but by our assumption,  $c$  may not. By the induction assumption, SQUAREPATHEXPLORATION reaches at least  $a$  or  $b$ .

<sup>5</sup>To define the word “side”, we remove  $s$  and cut the edge “opposite of  $s$ ”, that is, the edge between two neighbors of  $s$  with the property that when it is removed together with  $s$ , the graph is not connected any more. If there are two such edges, for example if  $v_2 = s$  in Figure 3.8, we cut the edge that minimizes the difference in the amount of vertices of the two components. If no such edge exists, then  $s$  has degree 2 and there is just one side.

When  $a$  is reached first, then  $a$  has only one colored neighbor  $c$ , and the agent returns to  $c$ . We may invoke the induction assumption again to ascertain that SQUAREPATHEXPLORATION reaches  $b$ . At this point, since there are no holes, the only uncolored neighbor is  $v$ . Therefore, SQUAREPATHEXPLORATION goes to  $v$ , colors it and then returns to either  $a$  or  $b$ .

When  $b$  is reached first, we have two cases. First, SQUAREPATHEXPLORATION could go directly to  $a$  or  $v$ . This can only happen if  $c$  and  $d$  are colored since a vertex needs two colored neighbors in order to go to an uncolored neighbor. If the agent visits  $a$ , it then goes to  $v$ . If the agent visits  $v$ , it first returns to  $b$ . Now,  $a$  is the only uncolored neighbor of  $b$  and  $b$  has more than one colored neighbor. Hence, SQUAREPATHEXPLORATION visits  $a$  next and we are done.

Second, SQUAREPATHEXPLORATION could go back to  $c$  or  $d$ . By the induction assumption, going back to  $c$  is not possible as this would assume that  $d$  is not colored and we have a hole already. Here, we use the fact that the one of the two colored vertices on one side of the start vertex is in the top row and one is in the bottom row by the initial observation. Hence, in this case, the agent goes back to  $d$  and  $c$  is not colored. On  $d$ , the agent visits  $c$  as the only uncolored neighbor. The agent colors  $c$  and then goes to  $a$  as the only uncolored neighbor and again on  $a$ , the agent colors  $a$  and then goes to  $v$  as the only uncolored neighbor.

Thus, SQUAREPATHEXPLORATION explores the larger graph without creating a hole.  $\square$

The previous lemmas can be combined to show that SQUAREPATHEXPLORATION indeed explores all vertices.

**Lemma 3.19.** *SQUAREPATHEXPLORATION explores all vertices.*

*Proof.* By Lemmas 3.15 and 3.16, SQUAREPATHEXPLORATION terminates on the start vertex  $s$ . However, SQUAREPATHEXPLORATION cannot terminate with uncolored neighbors. As soon as two neighbors on one side of  $s$  are colored, all vertices of this side are colored by Lemma 3.18. Of course, again by Lemmas 3.15 and 3.16, SQUAREPATHEXPLORATION then returns to  $s$ . If only one neighbor  $v$  on one side of the start vertex is colored, then either there is another uncolored vertex  $v_2$  on the same side of  $s$  that is also a neighbor of  $s$  or there is no other vertex on this side of  $s$ .  $\square$

This finally leads to the desired theorem:

**Theorem 3.10.** *SQUAREPATHEXPLORATION is correct on graphs of the form  $P^2$  and uses at most 4 colors.*

## 3.6 Exploration with Recoloring

In this section, we allow recoloring of already colored vertices. We prove that in this case, seven colors are enough to explore any graph. This demonstrates the superior strength of strategies with recoloring. We present a formal description of our algorithm RECOLORER in Algorithm 5.

Let us describe the basic strategy. We abuse the term “color” and say a vertex either receives label  $x$  or a label 1, 2 or 3 together with one of the two colors green



and red. The special label  $x$  is assigned to *delete* vertices, that is, to mark vertices to which the algorithm must never return. The algorithm performs breadth-first search and labels every vertex with 1, 2, or 3 depending on *depth*, i.e., its distance from the start vertex; vertices in depth  $k$  receive label  $k + 1 \bmod_1 3$ . Unless a vertex is deleted, the labels are never changed, only the colors.

We imagine the start vertex as top vertex and go from top to bottom when moving further away from the start vertex. Since vertices in depth  $k$  have only neighbors in depth  $k - 1$ ,  $k$ , and  $k + 1$ , this labeling provides a sense of direction for the algorithm.

When RECOLORER is on a vertex  $v$  that is in depth  $k$ , we call neighbors in depth  $k - 1$  *parents* (of  $v$ ), neighbors in depth  $k + 1$  *children* (of  $v$ ), and neighbors in the same depth as  $v$  *siblings*. Note that the agent is going to be able to determine whether labeled neighbors are parents, siblings, or children. To ensure that all vertices in a certain depth are colored, the algorithm colors the vertices from green to red and vice versa, depending on the current phase.

There are green phases and red phases in alternating order.

Initially, the start vertex receives label 1 and color red. Then, a green phase starts. In a green phase, the algorithm follows the red labels from 1 to 2 to 3 to 1 etc. until an unvisited vertex is reached. This vertex is then marked with the next label and colored green. The algorithm then proceeds to a parent  $p$ . If  $p$  has unvisited neighbors—these are all children, as we will see—, then one of these neighbors is visited. Otherwise, if all children of  $p$  are colored, the algorithm goes down to a child of the same color as  $p$  until it eventually finds an unvisited vertex, which is then colored green before going up to a parent. As soon as all children of a parent are of another color, the algorithm recolors the parent and then goes up to a grandparent. This process takes place until the start vertex is reached and all children of the start vertex are colored green. Now, the start vertex is marked green as well and a red phase starts, which works analogously.

This process would suffice to achieve perpetual exploration. To ensure that the algorithm terminates, the algorithm deletes a vertex whenever all its children are deleted or if there are no children at all. An example of an exploration is depicted in [Figure 3.13](#).

To prove that RECOLORER successfully explores all graphs, we first define the term *phase*. Initially, RECOLORER is in phase 0. For each  $k \in \mathbb{N}^+$ , phase  $k$  ends and phase  $k + 1$  begins immediately after the start vertex receives a color and the agent decides to terminate or to go to a neighbor. Before we turn towards the main lemma, we start with two general observations.

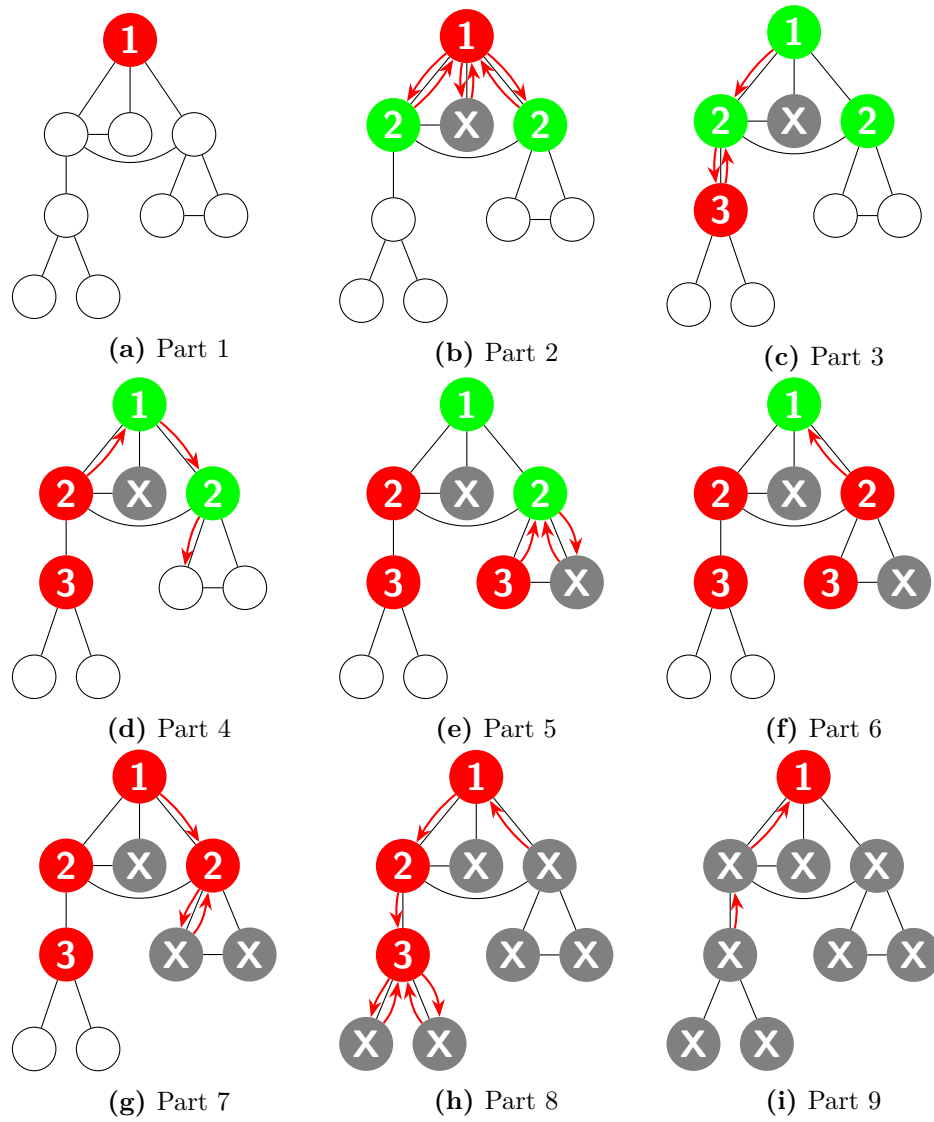
**Observation 3.3.** *Upon visiting, a vertex becomes labeled.*

**Observation 3.4.** *The label of a vertex  $v$  is only changed if  $v$  gets deleted.*

**Lemma 3.20.** *At the end of phase  $k$ , all vertices in depth  $j$  with  $j \leq k$  are either deleted or labeled with  $(j + 1) \bmod_1 3$ ; all non-deleted and labeled vertices are colored with the same color; and no vertices in depth  $k + 1$  or lower are visited. Moreover, the agent is not caught in an endless loop in phase  $k$ .*

*Proof.* We prove this by induction. Clearly, the start vertex is labeled and colored in phase 0 according to lines 6 to 9 and the next phase starts.





**Figure 3.13.** An example of an exploration by RECOLORER.

---

**Algorithm 5** RECOLORER

---

**Input:** An undirected graph in an online fashion. In each step, the input is  $l(v), c(v) \in \mathbb{N}$ , which are label and color of the current vertex  $v$ , and  $l(v_1), c(v_1), \dots, l(v_k), c(v_k)$ , the labels and colors of the neighbors of  $v$ , where  $k$  is the degree of  $v$ .

**Output:** In each step, the algorithm outputs  $l(v)$ , the label the agent assigns to  $v$ ;  $c(v)$ , the color the agent assigns to  $v$ ; and  $i \in [k]$ , the vertex the agent has to take next. It is allowed to change the color of a vertex that has been colored before.

**Description:** For each phase  $k$ , visit all vertices in depth  $k$  and then return to the start vertex. On return, color all vertices green if  $k$  is odd, and red otherwise.

```

1: if all children of  $v$  are deleted then
2:    $\triangleright$  the entire subtree of the current vertex has been explored
3:    $\triangleright$  or there is nothing to explore
4:    $l(v) := x$   $\triangleright v$  is deleted as well
5:   go to any parent and if there is no parent, terminate.
6: else if  $l(v) = 0$  then  $\triangleright v$  has not been visited before
7:    $l(v) := \left( \min_{v' \in N(v)} l(v') \right) + 1 \bmod_1 3$ 
8:    $c(v) := \begin{cases} \text{green,} & \text{if there is parent with color red} \\ \text{red,} & \text{otherwise} \end{cases}$ 
9:   go to any parent and if there is no parent, go to any neighbor  $\triangleright$  there is no
   parent in the case of the start vertex
10: else  $\triangleright v$  has been visited before
11:   if there is an unlabeled child  $v'$  then
12:      $\triangleright$  agent is on level  $i$  and level  $i + 1$  has not been completely explored
13:     go to  $v'$ 
14:   else if there is a child  $v'$  of the same color then
15:      $\triangleright$  agent is on level  $i$  and level  $j$  with  $j > i + 1$  has
16:      $\triangleright$  not been completely explored
17:     go to  $v'$ 
18:   else if there is a non-deleted child and all non-deleted children are of com-
  plementary color  $k$  (= green/red) then
19:      $\triangleright$  the subtree of the current vertex was explored up to some level  $i$  and
20:      $\triangleright$  now the agent has to move towards the start vertex
21:      $c(v) := k$   $\triangleright$  the vertex agrees with the color of its children
22:     go to any parent and if there is no parent, go to any non-deleted child.

```

---

We assume that the first part of the statement is true for  $k - 1$ : At the end of phase  $k - 1$  and thus at the beginning of phase  $k$ ,<sup>6</sup> the agent has labeled every vertex up to depth  $k - 1$ ; all vertices in depth  $j \leq k - 1$  are either deleted or labeled with label  $(j + 1) \bmod_1 3$ ; all non-deleted vertices up to depth  $k - 1$  have the same color; and no vertex in depth  $k$  is visited. Note that we do not need to assume that phase  $k - 1$  ends since this is implied by the “at the end of phase  $k - 1$ ”. We are going to show later in the proof that the agent is not caught in an endless loop.

We start with some observations.

**Observation 3.5.** *Observation 3.4 implies together with the induction assumption that, during phase  $k$ , on every vertex  $v$ , the agent can decide whether  $v$ 's non-deleted neighbors are parents or siblings or children.<sup>7</sup> The only exception are vertices in depth  $k$  from the start vertex. Here, the agent can decide which non-deleted vertices are parents, but it is not possible to distinguish siblings from children.*

**Observation 3.6.** *A vertex  $v$  only becomes deleted if all its children are deleted (lines 1 to 5). Deleted children are never visited again. Hence, a deleted vertex is never visited again and we can safely ignore deleted vertices.*

With these observations, we are able to prove a first claim towards proving the first point of the lemma.

**Claim 3.1.** *If a vertex in depth  $k$  is visited, it is labeled with  $(k + 1) \bmod_1 3$  or deleted.*

*Proof (of Claim 3.1).* Once the agent arrives for the first time at some vertex in depth  $k$ , all siblings and all children have not been visited and are thus unlabeled by Observations 3.3, 3.4, and 3.5. All parents are labeled with  $k \bmod_1 3$  by the induction hypothesis. Therefore, according to line 7, the agent assigns label  $(k + 1) \bmod_1 3$ . Note that this label will not be changed later, according to Observation 3.4.

Later in phase  $k$ , whenever the agent arrives at a vertex in depth  $k$ , there may be parents with label  $k \bmod_1 3$  and siblings with label  $(k + 1) \bmod_1 3$  and again, according to line 7, the agent assigns label  $(k + 1) \bmod_1 3$ .  $\square$

Next, we argue that a vertex cannot be colored several times in a phase, which yields as a byproduct that no vertices in depth  $k + 1$  or lower are visited.

**Claim 3.2.** *A vertex  $v$  cannot change its color twice in a phase and vertices in depth  $k + 1$  cannot be reached.*

*Proof (of Claim 3.2).* Consider any phase  $k$ . Assume towards contradiction that  $v$  is the vertex whose color is changed twice first in phase  $k$ . Assume that  $v$  is in depth  $k - 1$  or higher. The only reason to recolor  $v$  twice would be if all of  $v$ 's children were recolored twice. Thus,  $v$  could not be the first vertex whose color is changed twice.

Therefore,  $v$  must be in depth  $k$ . The first time  $v$  was colored took place because the agent visited  $v$  and  $v$  had not been visited before. To change the color a second time, the agent would have to access  $v$  again, either through one of  $v$ 's children or

<sup>6</sup>Unless phase  $k - 1$  is the final phase, of course.

<sup>7</sup>Neighbors with label  $(l(v) - 1) \bmod_1 3$  are parents, neighbors with label  $l(v)$  are siblings and neighbors with label  $(l(v) + 1) \bmod_1 3$  are children.

through one of  $v$ 's parents or through one of  $v$ 's siblings. The agent clearly never goes to the sibling of a vertex directly; therefore,  $v$  can only be accessed through a parent or through a child. However,  $v$ 's children can only be accessed through vertices in depth  $k$  and whenever a vertex in depth  $k$  is visited, the agent colors it with a different color than its parents and then goes up to any parent  $p$  in depth  $k - 1$ . In particular, the agent does not go down to any vertex in depth  $k + 1$ . Hence, the only possibility to visit  $v$  a second time is through a parent  $p$ .

From any such  $p$ , the only way to go down to  $v$  again would be if the agent visited  $p$  once  $p$  is recolored, that is, once  $p$  had received the same color as  $v$ . This is due to the fact that only children of the same color are visited. However, as soon as  $p$  is recolored, the agent goes to a higher vertex. Therefore, the only way to recolor  $v$  would be to change the color of all vertices on a path between  $v$  and the start vertex—including the start vertex—and then going down again, which is a contradiction to the assumption that  $v$  is colored twice in a phase.  $\square$

At the end of phase  $k$ , that is, when the start vertex becomes recolored, all non-deleted and labeled vertices are thus colored with the same color, which is the second point of the lemma. The only thing left to prove is that indeed all vertices in depth  $k$  are visited. We prove this by showing first that the agent cannot be caught in an endless loop in phase  $k$ .

**Claim 3.3.** *The agent is not caught in an endless loop in phase  $k$ .*

*Proof (of Claim 3.3).* Suppose there were an endless loop. Then there exists a sequence of vertices  $v_1, \dots, v_r$  such that, for  $i \in [r - 1]$ , the agent goes infinitely often from  $v_i$  to  $v_{i+1}$  and from  $v_r$  back to  $v_1$ . Since a vertex is not colored twice in phase  $k$ , all vertices in this sequence have a fixed color. Moreover, there cannot be any unlabeled neighbors since the agent would go to these according to line 11 and the sequence would change. Hence, the only lines that can be applied are lines 14 to 17. However, this implies that, for  $i \in [r - 1]$ ,  $v_i$  is a parent of  $v_{i+1}$  and  $v_r$  is a parent of  $v_1$ . This contradicts [Observation 3.5](#).  $\square$

Now, since there is no endless loop in phase  $k$ , phase  $k$  terminates, which means that the start vertex is recolored. This can only be the case if all children and children of children etc. are recolored or colored for the first time. This in turn means in particular that all vertices in depth  $k$  are colored. Thus, the proof of [Lemma 3.20](#) is finished.  $\square$

[Lemma 3.20](#) immediately implies the correctness of RECOLORER. We thus obtain the following theorem.

**Theorem 3.11.** *There is an algorithm that never uses more than 7 colors and explores every graph with recoloring.*

### 3.7 Conclusion

We investigated graph exploration by a very limited agent and showed tight bounds for the exploration of trees and of general graphs. Essential for our upper bounds was the idea that the algorithm needs a way to uniquely determine the direct predecessor

of a vertex. On a high level, our algorithms try to enumerate the number of situations that can occur locally. Surprisingly, we discovered that this number is not determined by the degree of a vertex, but by the circumference of the graph. We showed almost tight lower bounds for graphs of a certain circumference as well.

While exploring the driving parameter for the amount of colors necessary and sufficient, we proved almost tight bounds for the exploration of general bipartite graphs and for non-uniform algorithms, where we had to assume that the size of the graph is known.

For our lower bounds, [Lemma 3.6](#) and [Lemma 3.7](#) form the basis of our argumentation. These lemmas make it possible to construct specific graphs and know exactly—up to renaming of the colors—how a general algorithm behaves on this graph. Even for the cases where we could not apply the lemmas directly, as for example with the bipartite graphs or the non-uniform algorithms, we used a similar argumentation tailored to the specific case. As soon as the graph class is limited in such a way that these lemmas do not apply at all, it is quite difficult to prove lower bounds, which we demonstrated in [Section 3.5](#) with an algorithm successfully exploring all square paths by merely using four colors.

Finally, we studied the case where recoloring vertices is allowed. We showed that in this case, seven colors are already sufficient to explore all graphs, which stands in stark contrast to the amount of colors needed in general.

There are at least three ways to continue this research. First, it would be interesting to generalize [Lemma 3.6](#) and [Lemma 3.7](#) to certain subgraphs or minors. This would facilitate the analysis of graph exploration in our model for large graph classes. Second, a natural extension of our model would be to allow a constant number of memory bits. This would in particular enable a direct comparison to the existing research on oblivious graph exploration with known inputs. Third, it might be interesting to analyze how additional information can help the agent. On the one hand, one could study the classical advice complexity model for similar graph exploration models; on the other hand, one could restrict the advice to vertices. This would allow comparing global advice and local advice, which is much more restricted, but can be accessed exactly when needed.



# 4

## Online Knapsack with a Storeroom

### 4.1 Introduction

Say you wake up one morning in a nice hotel in the Pyrenees or in the Dolomites with the desire to hike to one of the beautiful mountains nearby and enjoy the breathtaking view from the top. First, you allow yourself a good breakfast, of course, but afterwards, you have to pack your backpack rather quickly. You don't want to carry more than 10 kilograms and you have to decide which of your many belongings you want to pack. Some items are more useful than others, but heavier. Which ones to include? You want to make the most of your packing—but how? Well, luckily, you easily recognize that this is an instance of the general knapsack problem and you know that there is a fully polynomial-time approximation scheme, so you pack your knapsack almost optimally, forget about the problem, and enjoy your day.

Back at home, you want to buy better gear for your next trip. You cannot afford to spend too much, so you go to one of these amazing online marketplaces to buy your gear. Cool items appear and disappear all the time and you are never sure which offer is the best one since there might be even better offers the next day. However, you know a tried and tested technique: you buy only items where you are sure you can sell them again, for just a little less. This way, you can in some sense reserve items at leisure, you merely pay a certain price for the reservation. So whenever a new item appears, you can either dismiss it right away since it is too expensive or you can buy it temporarily and thus reserve it for future use. But which item should you reject and which reserve to optimize your gear? This problem sounds similar to the knapsack problem, you think, but somehow you don't yet know how to solve it. This chapter shall help you with that problem.

There is hardly a more simple—yet practically relevant—maximization problem than the knapsack problem. It is defined as follows: Given a set of items  $I = \{1, \dots, n\}$ , where  $n \in \mathbb{N}$  and each item  $i$  has a weight or a *size*  $s_i \in \mathbb{Q}$  and a *profit*  $p_i \in \mathbb{Q}$ , the goal is to choose a subset  $J \subseteq I$  of the items such that the total profit  $\sum_{j \in J} p_j$  is maximized while the total size  $\sum_{j \in J} s_j$  is smaller or equal than some

*capacity constraint*  $c$ . The decision variant of the knapsack problem was shown to be NP-hard in the seminal paper by Karp [36] in 1972.

As a very general problem, it has even been called “the simplest prototype of a maximization problem” [38]. In 1990, Silvano Martello and Paolo Toth [43] wrote an entire book about the knapsack problem. While some still wondered how anybody can write 250 pages about a single problem [38], 14 years later, Hans Kellerer, Ulrich Pferschy and David Pisinger published a book with twice as many pages dedicated to this problem. Over 30 years after his monograph, Martello and coauthors complemented this work about the knapsack problem with a recent survey [9, 10], demonstrating how active research in this area is. It remains unclear from where the problem originated or who first used the term “knapsack problem”; nevertheless, the knapsack problem certainly has been discussed for more than a century and plenty variants and generalizations exist that have earned their own names, such as the subset sum problem or bin packing.

Because of its popularity, the knapsack problem has become a model problem for method development and a wealth of approaches on how to alleviate or circumvent the seemingly insurmountable difficulty of the P vs. NP problem have been taken on the basis of the knapsack problem. In this chapter, we extend and study the online variant of the simple knapsack problem with unit capacity and positive sizes. In the simple knapsack problem, the sizes of the items correspond to their profit. Restricting the problem to only positive sizes is not a real restriction since it is possible to formulate a knapsack problem with negative sizes and profits as a knapsack problem with positive sizes and profits, see Kellerer et al. [38].

With unit capacity, the capacity constraint  $c$  is taken to be 1, which is not an essential restriction either since the item sizes and profits and the capacity  $c$  can be divided by  $c$  and thus normalized without changing the problem.<sup>1</sup> The decision variant of the simple knapsack problem with unit capacity is still NP-hard. More importantly, however, requiring unit capacity is important for the online version of the knapsack problem. Without unit capacity, the items are usually assumed to be integers and thus to be in the range  $\{1, \dots, c\}$ . Thus, an algorithm knows the smallest possible size items can have. Furthermore, an algorithm can deduce an upper bound on the length of the instance, information which is usually not accessible to online algorithms.

We refer to the main introduction of this thesis for the definitions and notions around online problems. The online knapsack problem is not competitive, that is, the competitive ratio is not bounded by any constant: Given an input item of small size  $\varepsilon$ , an algorithm either has to accept it as part of the final solution set—that is, *pack* it—or *reject* it. In the first case, a second item of size 1 might appear and the instance might terminate. The ratio between the optimal offline solution and the algorithm is  $1/\varepsilon$ . In the second case, the instance might terminate without another item. The ratio between an optimal offline solution and the algorithm is unbounded. Therefore, various restrictions and extensions have been studied that render the problem competitive and hence interesting from an online perspective.

The focus of this chapter is on a particular extension proposed by Böcken-

---

<sup>1</sup>This is true in theory; however, note that this normalization might lead to much bigger numbers and the performance of many algorithms to solve the problem is negatively affected by the size of the numbers involved. See Page 11 in [38] for a more detailed discussion.



hauer et al. [3] in 2021. They suggested a general extension to the online framework, namely paying a certain fee for delaying decisions. In particular, they studied simple knapsack with unit capacity and allowed an algorithm to not only either pack or reject a new item  $i$  but to reserve it for a fee which is a fixed and known fraction  $\alpha$ —the *reservation factor*—of the size (and thus the profit)  $s_i$  of the item. Items which are reserved can be packed at any point.

This is an interesting model because the concept of reservation can be applied to many different online problems. The parameter  $\alpha$  determines the amount of “online-ness” of the problem; when  $\alpha$  is set to 0, an algorithm can reserve everything with no additional cost and then solve the problem optimally once the whole instance is revealed, thus rendering the problem an offline problem. When  $\alpha$  is set to 1, no reasonable algorithm will ever reserve anything, rendering the problem a complete online problem without additional aid. Thus, the reservation factor can be seen as bridging or smoothing the gap between online and offline problems.

However, although the model analyzed by Böckenhauer et al. is certainly well motivated, depending on the application, it is not the most natural model to pay a fixed fraction of the item size. Consider, for example, a freelance software engineer that wants to rent disk space in an online (in the non-technical meaning of the word) storage. It is quite natural that the software engineer would be able to pay just for the space she indeed uses; hence, the amount paid corresponds to the maximum amount of space used. In particular, it would not make sense that she pays for every transaction of data, it is much more natural that she is able to use space for some data, then remove or archive some data and use the freed space for other data so that the amount of space does not grow too fast.

Therefore, we study reservation cost where the algorithm pays a fixed fraction  $\alpha$  of the maximum total size of reserved items during its computation. Moreover, we generalize the model by Böckenhauer et al. and allow that the algorithm discards items from the reservation with no additional cost. Indeed, if an algorithm is not allowed to do that, then our model coincides with the one by Böckenhauer et al. Given our motivation example above, one might wonder why we do not analyze a “average time cost model,” in which the algorithm has to pay in every step a fixed fraction  $\alpha$  of the total size of the currently reserved items and the reservation cost is the average of these time step costs. Thus, we would take into account the time an algorithm has an item in the reserve. The reason we prefer our model is simple: Our model is easier to describe and analyze and at the same time equivalent to this average time cost model from a worst-case perspective. Every algorithm for the average time cost model is an algorithm in our model with the same competitive ratio and vice versa: Given an input in the time cost model, we can insert arbitrarily long sequences of tiny items of infinitesimal size at the point where the algorithm maximizes the total size of reserved items. On this new instance, the algorithm has an average cost that is arbitrarily close to the maximum total size of reserved items.

We show that our model displays the same bounds on the competitive ratio as the model of Böckenhauer et al. for  $\alpha \geq 0.2695$ ; hence, for a large reservation factor, having the additional option to discard reserved items for free does not help at all. For a reservation factor smaller than 0.2695, the competitive ratio of our model is potentially lower than the one of Böckenhauer et al.—which is at least 2—and we show an upper bound on the competitive ratio that is smaller than 2 for  $\alpha < 0.25$ .

The algorithm we use to show this upper bound is an optimal online algorithm for the model of Böckenhauer et al. for the whole range of the reservation factor. However, for  $\alpha$  tending to 0, the competitiveness of this algorithm does not go to 1, as it clearly should, but to  $\Phi := \frac{\sqrt{5}+1}{2} \approx 1.6181$ , the golden ratio, instead. Therefore, we provide an algorithm which performs well for very small  $\alpha$ ; the competitive ratio tends to 1, and is only a small constant factor larger than our lower bound.

This chapter is structured as follows. In this introduction, we continue first with a more formal introduction of our model and the used notation. Then we discuss related work and compare our approach to weaken the online property to similar approaches. We split the rest of the chapter into sections for large reservation factors and sections for small reservation factors since the used techniques are quite different. We begin with our main algorithm for large reservation factors and its analysis and are thus prepared for a corresponding lower bound, which we explain next. We then turn towards small reservation factors and present first upper and then lower bounds there as well, before we conclude this chapter with an outlook on possible future research.

## 4.2 Model and Notation

The input for our problem is a sequence of items  $I = (1, \dots, n)$ , such that for all  $i \in [n]$ , the *size*  $s_i$  of item  $i$  is in the interval  $[0, 1]_{\mathbb{R}}$ .<sup>2</sup> Whenever an item arrives, the algorithm has to either pack it, discard it, or reserve it. Whenever an algorithm decides to reserve an item, it may, just before reserving the new item, discard items—as many as desired—that are in the reserve at that time. The gain of an algorithm ALG on an input  $I$  is

$$\text{ALG}(I) := \text{GROSSGAIN}_{\text{ALG}} - \alpha R_{\max},$$

where  $\alpha$  is the reservation factor,  $\text{GROSSGAIN}_{\text{ALG}} \leq 1$  is the sum of the packed items and also called the *gross gain of ALG*, and  $R_{\max}$  is the largest sum of reserved items during any point in the computation. Note that an algorithm is allowed to optimally pack the reserve after the input ends.

For a given instance  $I$ , we use  $\text{OPT}(I)$  and sometimes just  $\text{OPT}$  to denote the size of an optimal offline solution. When it is clear that we are referring to a specific instance, we sometimes say *competitive ratio* to denote  $\rho_{\text{ALG}} := \text{OPT}(I)/\text{ALG}(I)$ .

Throughout this chapter, we use  $\Phi$  and  $\varphi := 1/\Phi = \frac{\sqrt{5}-1}{2} \approx 0.6180$  to denote the golden ratio and its reciprocal.

## 4.3 Related Work

The study of the online knapsack problem began with Marchetti-Spaccamela and Vercellis [42] in 1995, who performed an average case analysis. In 2002, Iwama and Taketomi [33] analyzed the removable online knapsack problem. Here, the algorithm is allowed to remove any packed item at any point for free. Items thus removed can never be retrieved. They showed a competitive ratio of  $\Phi$ , i.e., they designed a

<sup>2</sup>We take a more mathematical point of view; of course, from an implementation perspective, we would have to restrict the input, e.g., by only allowing rational numbers.

$\Phi$ -competitive algorithm and proved that it is best possible. We can translate their upper bounds for our model as follows. They constructed an algorithm such that

$$\frac{\text{OPT}}{\text{GROSSGAIN}_{\text{ALG}}} = \Phi.$$

Since their algorithm uses the knapsack itself as reserve, we know that the total size of the reserved items is at most 1. Hence, we have the following upper bound on the competitive ratio in our model:

$$\frac{\text{OPT}}{\text{ALG}} = \frac{\text{OPT}}{\text{GROSSGAIN}_{\text{ALG}} - \alpha \cdot 1} = \frac{1}{\frac{\text{GROSSGAIN}_{\text{ALG}}}{\text{OPT}} - \frac{\alpha}{\text{OPT}}} \stackrel{\text{OPT} \leq 1}{\leq} \frac{1}{\Phi^{-1} - \alpha} = \frac{1}{\varphi - \alpha}.$$

For  $\alpha = 0$ , this algorithm is thus  $\Phi$ -competitive and as long as  $\alpha \leq \varphi - 0.5 \approx 0.118$ , the algorithm is better than 2-competitive. They also analyzed the case that the algorithm may fill  $k$  knapsacks and then decide at the end which of the  $k$  knapsacks to choose. Here, they proved a tight bound on the competitive ratio of approximately 1.3815. Interestingly for us, they showed that this bound is achieved already if the algorithm just has 2 knapsacks to choose from. Therefore, if we translate their upper bound into our model, we obtain an algorithm that is  $1/(1.3815^{-1} - 2\alpha)$ -competitive. We show this bound in [Figure 4.4](#) on [page 92](#), when we establish our bound for small reservation factors. We cannot translate their lower bounds for our model: On the one hand, their algorithm is not allowed to group items in a knapsack of size  $k$ —which would correspond to our reserve—and then find the best filling for a knapsack of size 1, but it has to place the items into  $k$  distinct knapsacks. On the other hand,  $k$  is fixed and may not depend on the instance, whereas an algorithm in our model might only keep a large reserve if it knows that the optimal solution of the items so far is large.

In 2012, Han et al. [\[28\]](#) analyzed the online knapsack with removal cost problem, where packed items can be removed from the knapsack for a certain cost. This cost is either a fixed constant  $c$  per removed item  $i$  or a fixed fraction  $f$  of the item size  $s_i$ . In the latter case, they showed tight competitive ratios of 2 in case that  $f \leq 0.5$  and of  $(1 + f + \sqrt{f^2 + 2f + 5})/2$  in case that  $f \geq 0.5$ . Note that there is a remarkable gap when  $f$  is close to 0: For  $f = 0$ , everything can be removed for free and Iwama and Taketomi showed a tight competitive ratio of  $\Phi$ ; however, when the reservation cost is an arbitrarily small positive number, the competitive ratio jumps to 2. As Böckenhauer et al. [\[3\]](#) observed, the results of Han et al. cannot be translated to the model with reservation cost since an algorithm in the removal cost model does not have to pay for items that remain in the final solution. Moreover, in the removal cost model, the “reservation size” cannot exceed 1.

In 2015, Han et al. [\[29\]](#) extended their work further by analyzing online knapsack with randomization and showed that there is a 2-competitive online algorithm for the general online knapsack problem and a  $(1 + 1/e)$ -competitive algorithm for the general online knapsack problem with removability. For the simple online knapsack problem with removability, they proved an upper bound of  $10/7$  and a lower bound of 1.25.

In 2016, Cygan et al. [\[14\]](#) extended the results of Iwama and Taketomi to multiple knapsacks. Here, there are  $k$  identical knapsacks. Each new item has to be either rejected or packed into one of the knapsacks. Items cannot be moved between

knapsacks. Apart from analyzing both simple and general knapsack and both with removability and without, two variants were considered. In the first variant, the goal is to maximize the sum of the gains of all knapsacks. In the second variant, the goal is to maximize the highest gain. They refer to the results of Iwama and Taketomi for the model which is closest to our model, namely simple online knapsack with removability and the goal of maximizing the highest gain.

Another model, proposed by Han et al. [30] in 2019, is the online knapsack with a resource buffer problem. This model is a generalization of removal cost. There is a resource buffer of fixed size  $R \geq 1$ . Every item has to be either rejected or packed into the buffer. Once the input ends, the algorithm is allowed to create an optimal packing with the elements in the buffer. They analyzed both the simple and general online knapsack problem and both the case where these decisions are irrevocable or where items can be removed from the buffer without additional cost. If  $R = 1$  and items can be removed, this model coincides with removable knapsack. For the simple online knapsack problem with removability, they proved an upper bound of  $1 + \mathcal{O}(\log(R)/R)$  for general  $R$  and an upper bound of  $2/R$  for  $R \in [2\sqrt{3} - 2, 3/2]$ . The bound for general  $R$  is difficult to compare with our model since the value of the constant inside the  $\mathcal{O}$ -notation is crucial. The second bound yields an upper bound on the competitiveness of  $1/(R/2 - \alpha R)$ , which is at least

$$\frac{1}{\frac{3}{4} - \frac{3}{2}\alpha},$$

since the bound decreases with growing  $R$ . This bound is depicted in Figure 4.4 (page 92) as well. Note that this expression goes to  $4/3$  for  $\alpha$  tending to 0.

As mentioned above, in 2021, Böckenhauer et al. [3] suggested the model that is closest to our work. They discovered that their model with reservation cost displays a very interesting behavior: When the reservation factor  $\alpha$  is large, namely  $\alpha \geq \varphi$ , the competitive ratio is  $1/(1 + \alpha)$  and hence plummets from infinity to  $2 + \alpha \approx 2.618$ . When  $\alpha$  ranges between  $\sqrt{2} - 1$  and  $\varphi$ , the competitiveness descends gently to  $1 + \sqrt{2} \approx 2.41$ . Afterwards, with  $\alpha$  between  $\sqrt{2} - 1$  and 0.25, the competitive ratio picks up speed again and goes with  $(1 + \sqrt{5 - 4\alpha})/(2(1 - \alpha))$  to 2, where it comes to a grinding halt and stays at 2 for  $\alpha \in (0, 0.25]$ .

There are similar models that are less close to our model. A notable candidate is online knapsack with resource augmentation, which was introduced by Iwama and Zhang [34] in 2010. In this model, the algorithm is allowed to fill a knapsack of capacity  $R$ , but is compared against an optimal solution for a knapsack of unit capacity. Their results cannot be translated to our model. For a short, but recent overview, we refer to the survey by Cacchiani et al. [10].

All models mentioned have in common that the algorithm is endowed with additional abilities. Another approach is to measure how “far away” a model is from being optimally solvable, an idea that is formalized with the notion of advice complexity, as described in the main introduction of this thesis. In 2014, Böckenhauer et al. [5] analyzed the advice complexity of the online simple knapsack problem. They found that a single bit of advice can lower the competitive ratio from unbounded to 2; however, more advice does not help, unless the size of the advice is larger than  $o(\log(n))$ . In 2020, the advice complexity for online knapsack with removable items was analyzed by Böckenhauer et al. [4]. Without advice, this

problem is  $\Phi$ -competitive, as we mentioned above. The competitive ratio then falls to  $(1 + \varepsilon)$  for a constant amount of advice and stays there until there is one advice bit per item, in which case optimality can be achieved.

## 4.4 Upper Bounds for Large Reservation Factors

In this section, we establish an upper bound for our model by describing an algorithm and deriving upper bounds on its competitive ratio.

First, we observe that any valid algorithm in the model of Böckenhauer et al. [3], where reserved items cannot be discarded, is a valid algorithm in our model as well. Therefore, all upper bounds from the article by Böckenhauer et al. [3] hold in our model as well. These upper bounds were derived with three different algorithms. We present a unified algorithm, RESERVE, and derive the same upper bounds with a simpler analysis.

For  $\alpha \leq 1/4$ , our bounds improve upon the bounds one obtains by translating the algorithm by Böckenhauer et al. to our model. Our bounds are tight for  $\alpha \geq 0.2695$ ; the corresponding lower bounds are presented in Section 4.5.

We use the following notation. Depending on  $\alpha$ , a different value for the desired competitive ratio,  $\rho$ , has to be chosen in order to achieve the best behavior. We denote by  $s$  the function that maps a set of items to the sum of their sizes and use  $\mu$  to abbreviate  $1/((1 - \alpha)\rho)$ . Note that if an algorithm has a reserve of size  $\mu$ , it can achieve a competitive ratio of  $\rho$  by packing the whole reserve. We call the largest item in the reserve item  $\ell_{\max}$  and denote its size by  $s_{\max}$ . The detailed description of RESERVE is provided by Algorithm 6, explanations and details will be given in the proof.

---

### Algorithm 6 RESERVE

---

**Parameters:** Any  $\alpha \in (0, 1)$ ; desired competitive ratio  $\rho$ .

**Online Input:** A sequence  $I = (s_1, \dots, s_n)$  with the sizes of the  $n$  items.

**Online Output:** A reserve  $S \subseteq \{1, \dots, n\}$  and return of a final packing.

```

1:  $S \leftarrow \emptyset$ ;  $\ell_{\max} \leftarrow 0$    ▷ Initialize empty reservation storage and maximal reserved
   item of size  $s_{\max} = 0$ 
2: for  $i = 1, \dots, n$  do           ▷ Loop through all items
3:   if  $\text{OPT}(\{i\} \cup S) \geq 1/\rho + \alpha s(S)$  then   ▷ If current items allow it, then ...
4:     return  $\arg \max\{s(T) \mid T \subseteq \{i\} \cup S\}$    ▷ pack a solution of gain at least
    $1/\rho$ .
5:   else                               ▷ We have  $s(S) < \mu$  and  $s_i \geq 1 - \mu$ .
6:     if  $s_i + s(S) \leq \mu$  then ▷ If new item keeps reservation total below 1, ...
7:        $S \leftarrow S \cup \{i\}$            ▷ ... then reserve it.
8:     else ▷ We have  $s_i + s(S) > \mu$  and thus, due to line 4, even  $s_i + s(S) > 1$ 
9:       if  $s_i < s_{\max}$  then ▷ If new item is smaller than largest reserved item,
10:         $\ell_{\max} \leftarrow i$ 
11:         $S \leftarrow S \setminus \{\ell_{\max}\} \cup \{i\}$    ▷ ... reserve  $i$  and discard  $\ell_{\max}$ .
12:      else
13:        reject  $i$                                ▷ Reject new item.
14: return  $S$ 

```

---

We are going to prove the following theorem.

**Theorem 4.1.** *RESERVE achieves a competitive ratio of*

$$\rho \leq \begin{cases} \frac{1}{1-\alpha}, & \text{for } \alpha \in [\varphi, 1] \\ 2 + \alpha, & \text{for } \alpha \in [\sqrt{2} - 1, \varphi] \\ \frac{\sqrt{5-4\alpha}+1}{2(1-\alpha)}, & \text{for } \alpha \in (0, \sqrt{2} - 1]. \end{cases}$$

As the theorem suggests, we analyze RESERVE separately for three different ranges of  $\alpha$ , namely  $[\varphi, 1]$ ,  $[\sqrt{2} - 1, \varphi]$ , and  $(0, \sqrt{2} - 1]$ . Before we dive into the detailed analysis, we make a few general observations and prove a general lemma.

**Observation 4.1.** *If an algorithm can pack a solution of size  $1/\rho + \alpha s(S)$ , where  $S$  is the set of reserved elements and  $s(S)$  is the sum of the sizes of the reserved elements, then the algorithm achieves a competitive ratio of at least  $\rho$ .*

*Proof.* The gain of an optimal solution is at most 1. The gain of the algorithm is the size of the solution minus the reservation cost; hence this is at least

$$1/\rho + \alpha s(S) - \alpha s(S) = \frac{1}{\rho}. \quad \square$$

Therefore, it suffices to restrict our analysis to the case where the algorithm terminates via the only remaining return, namely via line 14.

**Observation 4.2.** *If  $\mu \leq s(S) \leq 1$ , then an algorithm can achieve a competitive ratio of  $\rho$  by packing the current reservation.*

*Proof.* Since  $s(S) \leq 1$ , all reserved elements can be packed. The gain of the algorithm is at least

$$s(S) - \alpha s(S) = (1 - \alpha)s(S) \geq (1 - \alpha)\mu = (1 - \alpha)\frac{1}{(1 - \alpha)\rho} = \frac{1}{\rho}. \quad \square$$

**Lemma 4.1.** *There is never an item  $k$  in the reserve such that  $s_k \geq 1/\rho$ .*

*Proof.* Assume towards contradiction that item  $k$  with  $s_k \geq 1/\rho$  becomes reserved. The condition in line 6 must be fulfilled, so

$$s(S) + s_k \leq \mu \leq 1.$$

In particular, both the whole reservation and  $k$  fit into the knapsack. The condition in line 3 must not have been fulfilled, which means

$$\begin{aligned} \text{OPT}(\{k\} \cup S) &< \frac{1}{\rho} + \alpha s(S) \implies \text{The reservation and } k \text{ fit into the knapsack} \\ s(S) + s_k &< \frac{1}{\rho} + \alpha s(S) \iff \text{Minus } s(S) \\ s_k &< \frac{1}{\rho} - (1 - \alpha)s(S) \implies s(S) \geq 0 \\ s_k &< \frac{1}{\rho}. \end{aligned}$$

This contradicts our assumption and thus finishes the proof.  $\square$

**Observation 4.3.** RESERVE *never reserves more than a total size of 1.*

*Proof.* If a new item  $i$  arrives such that the size of  $i$  together with all reserved items exceeds 1, then  $i$  is only reserved if there is an item larger than  $i$  in the reserve that becomes discarded in the same step.  $\square$

**Observation 4.4.** *The algorithm terminates if it can pack a solution of size  $\mu$  or larger.*

*Proof.* Since the reservation size for the algorithm is at most  $\mu$  due to [Observation 4.2](#) and [Observation 4.3](#), the gain of the algorithm is in this case at least

$$\mu - \alpha s(S) \geq \mu - \alpha \mu = \frac{1}{\rho}. \quad \square$$

### Range $\varphi \leq \alpha \leq 1$

If  $\alpha \in [\varphi, 1]$ , we set  $\rho := 1/(1 - \alpha)$  and claim that RESERVE is at most  $\rho$ -competitive. We show that every item is reserved and no item is discarded until the algorithm terminates.

Consider the first arrival of a new item  $i$  such that  $s_i + s(S) > \mu$ , which might lead to the new item not being reserved in line 7. We claim that the algorithm will return via line 4 in this case. Consider the optimal solution that the algorithm can construct from the items  $\{i\} \cup S$ . If  $s_i + s(S) < 1$ , then obviously  $\text{OPT}(\{i\} \cup S) \geq s_i + s(S) > \mu$  and we are done according to [Observation 4.4](#). Otherwise, we have  $s_i + s(S) > 1$ , thus not all reserved items are part of the algorithm's solution.

Consider a smallest reserved item not in this solution, call it  $\delta$ . Denote the total size of the rest of the reserve by  $r$ . The algorithm's solution leaves a gap smaller than  $s_\delta$ ; otherwise, it would have added  $\delta$  to the solution. If  $r \geq 1 - s_\delta \frac{1}{1 - \alpha}$ , the algorithm would have terminated via line 4 at the latest when the last item from  $S$  appeared since it would have achieved a gain of at least

$$s_\delta + r(1 - \alpha) \geq s_\delta + (1 - \alpha) - s_\delta = 1 - \alpha = \frac{1}{\rho},$$

where we considered the worst case of the smallest item  $\delta$  being the last one and the reserve thus being maximal.

Thus, we have  $r < 1 - s_\delta \frac{1}{1 - \alpha}$  and the gain of the algorithm is at least

$$(1 - s_\delta) - \alpha(r + s_\delta) \geq 1 - s_\delta(1 + \alpha) - \alpha + s_\delta \frac{\alpha}{1 - \alpha} = 1 - \alpha + s_\delta \left( \frac{\alpha}{1 - \alpha} - 1 - \alpha \right).$$

We have  $\frac{\alpha}{1 - \alpha} - \alpha > 1$  for all  $\alpha > \varphi$ , thus the last factor is positive for all  $\alpha \geq \varphi$  and the gain for this solution is at least  $1 - \alpha = 1/\rho$ .

### Range $\sqrt{2} - 1 \leq \alpha \leq \varphi$

If  $\alpha \in [\sqrt{2} - 1, \varphi]$ , we set  $\rho := 2 + \alpha$  and claim that RESERVE is at most  $\rho$ -competitive. As before, we show that every item is reserved and no item is discarded until the algorithm terminates. We argue as before to conclude that the algorithm reserves



everything until  $s_i + s(S) > 1$ , that is, until reserving the new item would let the total reserve exceed 1. We claim that the algorithm can then pack the current optimal solution: This solution has a gain of at least  $1 - s_{\max}$ , where we use  $s_{\max}$  as the size of the largest reserved item, as we mentioned above. This can be seen by considering the solution that is achieved when  $i$  plus all reserved items except  $\ell_{\max}$  are packed, which leads due to  $s_i + s(S) > 1$  to a gain larger than  $1 - s_{\max}$ . If this solution is still larger than 1, then the second largest reserved item, say  $k$ , can be removed as well, leading to a solution larger than  $1 - s_k$ , which is larger than  $1 - s_{\max}$ , and so on.

There might be other reserved items with a total sum of  $x$  that can be fitted into the knapsack as well. The total gain is  $1 - s_{\max} + x$ . The total reserve is at most  $1 - s_{\max} + x$  as well. Using  $s_{\max} < 1/\rho$  from [Lemma 4.1](#), the algorithm's gain is at least

$$(1 - s_{\max} + x)(1 - \alpha) \geq \left(1 - \frac{1}{\rho} + x\right)(1 - \alpha) \geq \left(1 - \frac{1}{\rho}\right)(1 - \alpha) \geq \frac{1}{\rho},$$

where the last inequality is true for  $\rho \geq (2 - \alpha)/(1 + \alpha)$ , which is true for our choice of  $\rho$ .

### Range $0 < \alpha \leq \sqrt{2} - 1$

If  $\alpha \in (0, \sqrt{2} - 1]$ , we set  $\rho := (\sqrt{5 - 4\alpha} + 1)/(2(1 - \alpha)) \in [\Phi, \sqrt{2} + 1]$  and claim that RESERVE is at most  $\rho$ -competitive. Again, the algorithm reserves everything until  $s_i + s(S) > 1$ . We prove with two small lemmas that the size of  $\ell_{\max}$ , the largest item in the reserve, is larger than  $1 - \mu$  and that all other reserved elements are smaller than  $1 - \mu$ .

**Lemma 4.2.** *The size of  $\ell_{\max}$  is larger than  $1 - \mu$ .*

*Proof.* Otherwise, all reserved elements are at most  $1 - \mu$ . Since we have  $s(S) + s_i > 1$ , the algorithm could discard the reserved items one by one until the rest (including  $i$ ) fits. This would lead to a solution of size at least  $\mu$  and the algorithm would have packed according to [Observation 4.4](#) with a competitive ratio of at least  $\rho$ .  $\square$

**Lemma 4.3.** *The reserve contains at most one item of size at least  $1 - \mu$ .*

*Proof.* Consider an arbitrary situation in which the reserve already contains one item  $\ell_{\max}$  of size at least  $1 - \mu$ , and the new item  $i$  has size at least  $1 - \mu$  as well. The only way for the reserve to contain more than one item of size at least  $1 - \mu$  is to reserve  $i$  in such a situation. Reserving item  $i$  would require  $s_i + s(S) \leq \mu \leq 1$  from [line 6](#). However, in this case, packing  $i$  and the whole reserve is a valid solution, implying

$$\begin{aligned} \text{OPT}(\{i\} \cup S) &\geq s_i + s(S)(1 - \alpha) \\ s_i \text{ and } s(S) \text{ both } \geq 1 - \mu &\geq (1 - \mu) + (1 - \mu)(1 - \alpha) \\ \text{Definition of } \mu &= \left(1 - \frac{1}{(1 - \alpha)\rho}\right)(2 - \alpha) \end{aligned}$$

We analyze for which value of  $\rho$  the right hand side is at least  $1/\rho$ . If this is the case, then we can claim that the algorithm would have terminated with sufficient gain. Hence, we calculate:

$$\left(1 - \frac{1}{(1 - \alpha)\rho}\right)(2 - \alpha) \geq \frac{1}{\rho} \iff$$



$$\begin{array}{ll}
\text{Multiply with } \rho & \left(\rho - \frac{1}{1-\alpha}\right)(2-\alpha) \geq 1 \quad \iff \\
\text{Divide by } 2-\alpha, \text{ then add } 1/(1-\alpha) & \rho \geq \frac{1}{2-\alpha} + \frac{1}{1-\alpha} \quad \iff \\
\text{Use common denominator} & \rho \geq \frac{3-2\alpha}{(2-\alpha)(1-\alpha)}.
\end{array}$$

We want to know whether this is true for our choice of  $\rho$ . Therefore, we insert our definition of  $\rho$  and check for which values of  $\alpha$  this holds:

$$\begin{array}{ll}
\frac{\sqrt{5-4\alpha}+1}{2(1-\alpha)} \geq \frac{3-2\alpha}{(1-\alpha)(2-\alpha)} & \iff \cdot 2(1-\alpha)(2-\alpha) \\
(2-\alpha)\sqrt{5-4\alpha} + (2-\alpha) \geq 6-4\alpha & \iff -(2-\alpha) \\
(2-\alpha)\sqrt{5-4\alpha} \geq 4-3\alpha & \iff \text{Take squares} \\
(4-2\alpha+\alpha^2)(5-4\alpha) \geq 16-24\alpha+9\alpha^2 & \iff \text{Expand} \\
20-10\alpha+5\alpha^2-16\alpha+8\alpha^2-4\alpha^2 \geq 16-24\alpha+9\alpha^2 & \iff \\
4-2\alpha \geq 0 & \iff \\
2 \geq \alpha. & 
\end{array}$$

This is always true; therefore, the algorithm would indeed have terminated with sufficient gain and the reserve contains thus at most one item  $\ell_{\max}$  of size at least  $1-\mu$  throughout the processing of the instance.  $\square$

Denote by  $r$  the size of the reserve except for  $\ell_{\max}$ , that is,  $r := s(S) - s_{\max}$ . If  $s_i + r > 1$ , then the algorithm could have terminated earlier: Since all elements in the reserve except  $\ell_{\max}$  are smaller than  $1-\mu$  because of [Lemma 4.3](#), the algorithm could have rejected items from the reserve one by one until the remaining items form a solution with a gap smaller than  $1-\mu$ .

Therefore, we have  $s_i + r \leq 1$  and hence a solution of size  $s_i + r$  is possible. This solution would lead to a gain of at least

$$s_i + r - \alpha s_{\max} - \alpha r \geq s_i - \alpha s_{\max}.$$

If this value exceeded  $1/\rho$ , the new item  $i$  would have been packed in [line 3](#) and we are done. Therefore, either we are done or we have

$$s_i - \alpha s_{\max} < 1/\rho \tag{4.1}$$

and the algorithm rejects or discards the larger item of the two items  $i$  and  $\ell_{\max}$ . We show with the following lemma that either  $s_i \geq 1/2$  or  $s_{\max} \geq 1/2$  in this case and that an optimal solution can consequently contain at most one item that has been rejected.

**Lemma 4.4.** *We have  $s_i + s_{\max} > 1$ , where  $s_{\max}$  is the size of the largest reserved item.*

*Proof.* If  $s_i + s_{\max} \leq 1$ , the algorithm could discard all reserved items except for  $\ell_{\max}$  one by one, leading to a valid solution with a gap of at most  $1-\mu$  including  $i$  and  $\ell_{\max}$  since all reserved items except  $\ell_{\max}$  are smaller than  $1-\mu$  by [Lemma 4.3](#).

The reservation has a size of at most  $\mu$  as we observed in [Observation 4.2](#). Thus, this solution would have a gain of at least  $\mu - \alpha\mu = 1/\rho$ ; hence, the algorithm would have terminated in [line 4](#). We thus indeed have  $s_i + s_{\max} > 1$ .  $\square$

As we mentioned before, RESERVE is thus going to reject  $i$  only if  $s_i \geq 1/2$  and RESERVE discards  $\ell_{\max}$  only if  $s_{\max} \geq 1/2$ .<sup>3</sup> Therefore, at most one item that was rejected or discarded by RESERVE is part of a fixed optimal solution.

Consider the case that the optimal solution contains an item  $\ell_{\max}$  with  $s_{\max} > s_i$  and that this item has been discarded and was replaced by item  $i$ . We know that  $s_{\max} < 1/\rho$  and  $s_i > 1 - 1/\rho$ ; therefore, the competitive ratio in this case is at most

$$\frac{\text{OPT}(I)}{\text{ALG}(I)} \leq \frac{s_{\max} + r}{s_i - \alpha s_{\max} + r(1 - \alpha)} \leq \frac{s_{\max}}{s_i - \alpha s_{\max}} \leq \frac{\frac{1}{\rho}}{1 - \frac{1}{\rho} - \frac{\alpha}{\rho}} = \frac{1}{\rho - 1 - \alpha}.$$

This is smaller than  $\rho$  if

$$0 \leq \rho^2 - (1 + \alpha)\rho - 1.$$

We check whether this is true for our choice of  $\rho$ :

$$\begin{aligned} 0 &\leq \left( \frac{\sqrt{5 - 4\alpha} + 1}{2(1 - \alpha)} \right)^2 - (1 + \alpha) \frac{\sqrt{5 - 4\alpha} + 1}{2(1 - \alpha)} - 1 \\ &= \frac{6 - 4\alpha + 2\sqrt{5 - 4\alpha}}{4(1 - \alpha)^2} - (1 + \alpha) \frac{\sqrt{5 - 4\alpha} + 1}{2(1 - \alpha)} - 1 && \iff \\ 0 &\leq 6 - 4\alpha + 2\sqrt{5 - 4\alpha} - 2(1 - \alpha)(1 + \alpha)(\sqrt{5 - 4\alpha} + 1) - 4(1 - \alpha)^2 \\ &= 6 - 4\alpha - 2(1 - \alpha^2) - 4(1 - \alpha)^2 + \sqrt{5 - 4\alpha}(2 - 2(1 - \alpha^2)) \\ &= -4\alpha + 2\alpha^2 + 8\alpha - 4\alpha^2 + 2\alpha^2\sqrt{5 - 4\alpha} \\ &= 4\alpha - 2\alpha^2 + 2\alpha^2\sqrt{5 - 4\alpha}. \end{aligned}$$

Since  $\alpha > \alpha^2$  for any non-negative  $\alpha < 1$ , this is clearly true.

Consider the case that the optimal solution contains an item  $i$  with  $s_i > 1/2$  and that this item has been rejected. By rejecting any item that is not in the optimal solution, the algorithm incurs no loss. The competitive ratio in this case is

$$\frac{\text{OPT}(I)}{\text{ALG}(I)} \leq \frac{s_i + r}{(s_{\max} + r)(1 - \alpha)} \leq \frac{s_i}{s_{\max}(1 - \alpha)}.$$

We show in the following that this is smaller than  $\rho$  by proving that

$$\min \left\{ s_i - \alpha s_{\max}, \frac{s_{\max}(1 - \alpha)}{s_i} \right\} \geq \frac{1}{\rho}.$$

Together with [inequality \(4.1\)](#), this proves that  $s_i/(s_{\max}(1 - \alpha))$  is smaller than  $\rho$ . Let  $y$  be such that  $1 + y = s_i + s_{\max}$ . We want to show that  $y = 0$  realizes our minimum. Consider the two expressions of the minimum and observe how they change with changing  $y$ :

$$1 + y - s_{\max} - \alpha s_{\max} \quad \text{and} \quad \frac{(1 + y - s_i)(1 - \alpha)}{s_i}.$$

<sup>3</sup>Note that combining [Lemma 4.4](#) and [Lemma 4.1](#) yields  $s_i > 1 - 1/\rho$  and that this is at least  $1/2$  for  $\rho \geq 2$ , which is the case for  $\alpha \geq 1/4$ . Thus, discarding  $\ell_{\max}$  only occurs for  $\alpha < 1/4$ .

Clearly, with increasing  $y$ , both expressions increase. Therefore, we may assume  $y = 0$ .<sup>4</sup> Consider for our next step the same two expressions, where we write  $s_{\max} = 1 - s_i$ :

$$s_i(1 + \alpha) - \alpha \quad \text{and} \quad \frac{(1 - s_i)(1 - \alpha)}{s_i}.$$

If we increase  $s_i$ , the left expression increases and the right expression decreases. If we decrease  $s_i$ , the opposite is true. Thus, the minimum of these expressions is maximized exactly where these two expressions are equal. Hence, we calculate

$$\begin{aligned} s_i(1 + \alpha) - \alpha &= \frac{(1 - s_i)(1 - \alpha)}{s_i} && \iff \\ s_i^2(1 + \alpha) - \alpha s_i &= 1 - \alpha - s_i + \alpha s_i && \iff \\ s_i^2(1 + \alpha) + s_i(1 - 2\alpha) - (1 - \alpha) &= 0. \end{aligned}$$

Solving this quadratic equation for  $s_i$  leads to two solutions for  $s_i$ . Only one of these options is positive, namely

$$s_i = \frac{\sqrt{5 - 4\alpha} - 1 + 2\alpha}{2(1 + \alpha)}.$$

If we insert this value for  $s_i$  into  $s_i(1 + \alpha) - \alpha$ , we obtain a simple expression for  $s_i - \alpha s_{\max}$ :

$$\begin{aligned} s_i(1 + \alpha) - \alpha &\geq \frac{\sqrt{5 - 4\alpha} - 1 + 2\alpha}{2(1 + \alpha)}(1 + \alpha) - \alpha \\ \text{Use same denominator} &= \frac{\sqrt{5 - 4\alpha} - 1 + 2\alpha - 2\alpha}{2} \\ &= \frac{\sqrt{5 - 4\alpha} - 1}{2}. \end{aligned}$$

This value is at least  $1/\rho$  if  $\rho$  satisfies the following:

$$\begin{aligned} \frac{\sqrt{5 - 4\alpha} - 1}{2} &\geq \frac{1}{\rho} \iff \\ \rho &\geq \frac{2}{\sqrt{5 - 4\alpha} - 1} \\ \text{Multiply with } 1 = \frac{\sqrt{5 - 4\alpha} + 1}{\sqrt{5 - 4\alpha} + 1} &= \frac{2}{\sqrt{5 - 4\alpha} - 1} \cdot \frac{\sqrt{5 - 4\alpha} + 1}{\sqrt{5 - 4\alpha} + 1} \\ (a - b)(a + b) = a^2 - b^2 &= \frac{2(\sqrt{5 - 4\alpha} + 1)}{5 - 4\alpha - 1} \\ 5 - 4\alpha - 1 = 4(1 - \alpha) &= \frac{\sqrt{5 - 4\alpha} + 1}{2(1 - \alpha)}. \end{aligned}$$

This is exactly our definition of  $\rho$ . This finishes the proof, which concludes at the same time the proof of [Theorem 4.1](#).

<sup>4</sup>Of course, we cannot have exactly  $y = 0$  and instead think of  $s_i + s_{\max} = 1 + \varepsilon$  and let  $\varepsilon$  go to zero.

## 4.5 Lower Bounds for Large Reservation Factors

In this section, we complement the upper bound from the previous section by lower bounds. The following lower bounds are all based on the strategy of presenting a small subset of instances. Depending on what an algorithm does, the adversary presents the following item or stops. Note that while the basic idea is the same as for the model presented by Böckenhauer et al. [3], the bounds from there do not carry over since an algorithm in our model has the additional option to discard any reserved item.

We summarize the bounds we prove in this section with the following theorem:

**Theorem 4.2.** *Every algorithm ALG has a competitive ratio of at least*

$$\rho_{\text{ALG}} \geq \begin{cases} \frac{1}{1-\alpha}, & \text{for } \alpha \in [\varphi, 1] \\ 2 + \alpha, & \text{for } \alpha \in [\sqrt{2} - 1, \varphi] \\ \frac{\sqrt{5-4\alpha}+1}{2(1-\alpha)}, & \text{for } \alpha \in [x_2 \approx 0.2695, \sqrt{2} - 1] \\ \frac{1}{u-\alpha}, & \text{for } \alpha \in [x_1 \approx 0.1818, x_2] \\ \frac{u}{(1-\alpha)u-\alpha}, & \text{for } \alpha \in (0, x_1], \end{cases}$$

where

$$u = \frac{(3 - \sqrt{5 - 4\alpha})(2 - \alpha)}{2(1 + \alpha)^2} + \frac{\alpha}{1 + \alpha}.$$

### Range $\varphi \leq \alpha < 1$

We describe an instance family of just two instances and show that ALG cannot achieve a better competitive ratio than  $1/(1 - \alpha)$ . First, an item  $s$  of size  $1 - \alpha$  is presented. We have the following three cases:

**Case 1: ALG accepts  $s$ .** A final item of size 1 is presented. This leads to a competitive ratio of  $1/(1 - \alpha)$  (on this instance).

**Case 2: ALG rejects  $s$ .** The instance ends. This leads to an unbounded competitive ratio.

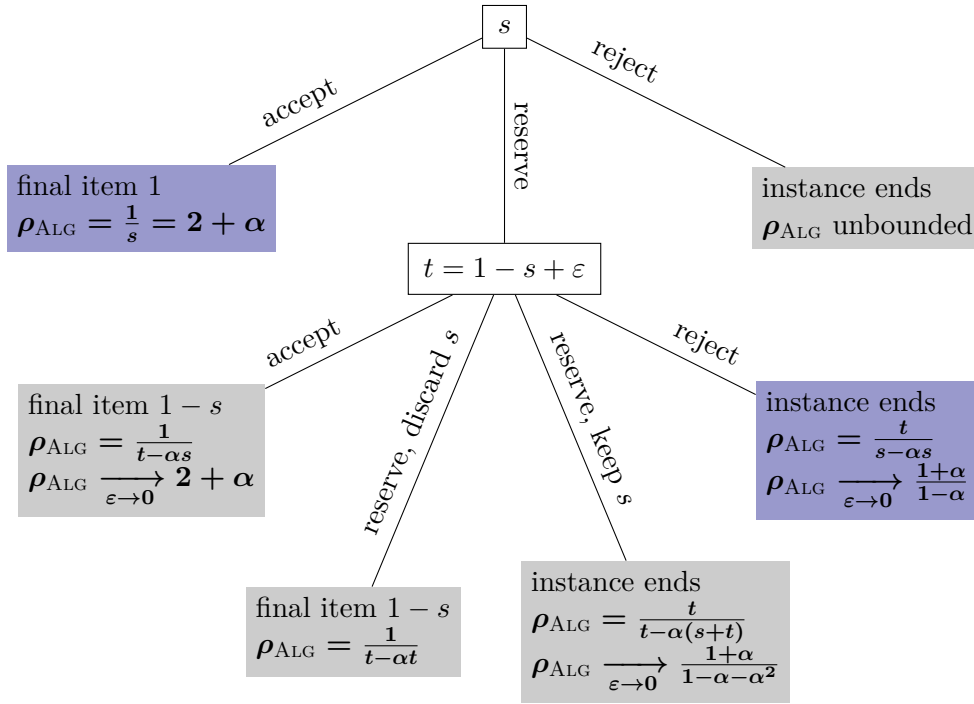
**Case 3: ALG reserves  $s$ .** The instance ends. This leads to a competitive ratio of

$$\frac{1 - \alpha}{1 - \alpha - \alpha(1 - \alpha)} = \frac{1}{1 - \alpha}.$$

The algorithm thus achieves a competitive ratio of at most  $\frac{1}{1-\alpha}$ . Note that if  $\alpha$  is smaller than  $\varphi$ , this bound holds as well; however, in that case we are able to show even larger lower bounds, as we will see in the following.

### Range $\sqrt{2} - 1 \leq \alpha \leq \varphi$

An item  $s$  of size  $1/(2 + \alpha)$  is presented. If  $s$  is reserved, an item  $t$  of size  $1 - s + \varepsilon$  is presented. Note that  $s + t > 1$  and  $t$  tends to  $1 - 1/(2 + \alpha) = (1 + \alpha)/(2 + \alpha)$  as  $\varepsilon$  goes to zero. The adversarial instance is depicted in Figure 4.1. Since there are only six cases, we provide a full description as well:



**Figure 4.1.** The adversarial instance class with two items  $s < t$ . Instances that yield worse or equal competitive ratios than other instances are marked with gray boxes.

**Case 1: ALG accepts  $s$ .** A final item of size 1 is presented. This leads to a competitive ratio of  $1/s = 2 + \alpha$ .

**Case 2: ALG rejects  $s$ .** The instance ends. This leads to an unbounded competitive ratio.

**Case 3: ALG reserves  $s$ .** The item  $t$  is presented. This leads to the following subcases.

**Case 3a: ALG accepts  $t$ .** A final item of size  $1 - s$  is presented. This results in a competitive ratio of  $\frac{1}{t - \alpha s}$ , which goes to  $2 + \alpha$  as  $\epsilon$  goes to zero.

**Case 3b: ALG rejects  $t$ .** The instance ends, leading to a competitive ratio of  $\frac{t}{s - \alpha s}$ , which goes to  $\frac{1 + \alpha}{1 - \alpha}$  as  $\epsilon$  goes to zero.

**Case 3c: ALG reserves  $t$ .** This leads to the following two subcases.

**Case 3c.1: ALG discards  $s$ .** A final item of size  $1 - s$  is presented. This results in a competitive ratio of  $\frac{1}{t - \alpha t}$ , which is smaller than  $\frac{1}{t - \alpha s}$ , which goes to  $2 + \alpha$  as  $\epsilon$  goes to zero.

**Case 3c.2: ALG keeps  $s$ .** The instance ends, leading to a competitive ratio of  $\frac{t}{t - \alpha(s+t)}$ , which goes to  $(1 + \alpha)/(1 - \alpha - \alpha^2)$  as  $\epsilon$  goes to zero. Note that this is always worse from the perspective of ALG than  $(1 + \alpha)/(1 - \alpha)$ .

For  $\epsilon$  tending to zero, the best achievable competitive ratio is  $\min\left\{2 + \alpha, \frac{1 + \alpha}{1 - \alpha}\right\}$ , which equals  $2 + \alpha$  for  $\sqrt{2} - 1 \leq \alpha$ .

### Range $0 \leq \alpha \leq \sqrt{2} - 1$

For  $\alpha \leq \sqrt{2} - 1$ , we need three items for our lower bound. We are going to choose  $0 < s < t < u < 1$  with  $s + t > 1$  and determine the worst values for the algorithm later in the analysis. Note that up to the presentation of the third item  $u$ , the instances are identical to the instances for larger values of  $\alpha$ . The adversarial instance family is depicted in Figure 4.2. Since there are many cases, we omit a description with text.

There are 11 potentially different options for an algorithm on our instance. Three options are strictly worse than other options and can be ignored. Two of the remaining eight options, namely accepting  $u$  and reserving  $u$  while discarding  $s$  and  $t$ , lead to the same competitive ratio. Thus, any algorithm is left with the seven options marked with light blue in Figure 4.2. The total competitive ratio is the minimum of

$$\left\{ \frac{1}{s}, \frac{1}{t - \alpha s}, \frac{t}{s(1 - \alpha)}, \frac{1}{u - \alpha(s + t)}, \frac{u}{t - \alpha(s + t)}, \frac{u}{u - \alpha(s + t + u)}, \frac{1}{s + 1 - t - \alpha(s + u)} \right\}.$$

We can choose  $s$ ,  $t$ , and  $u$  arbitrarily, as long as they fulfill the criteria  $s < t < u$  and  $s + t > 1$  to ensure that the competitive ratio is indeed the minimum of the expressions above. To determine the values of  $s$ ,  $t$ , and  $u$ , we require first  $s + t = 1 + \varepsilon \xrightarrow{\varepsilon \rightarrow 0} 1$  and then equate twice three options or equate three options in order to obtain three equations with which we can determine our three unknown values. In order to find the best values, we would have to do this for all possible combinations. Even worse, we would have to do this for each  $\alpha$ . However, we do not have to determine the best values for  $s$ ,  $t$ , and  $u$ . All values that fulfill the criteria lead to some competitive ratio. Therefore, we make a sophisticated guess about which expressions to equate using a popular mathematical tool and then just calculate the competitive ratio using these values without claiming that the chosen values are optimal.

We start by equating  $1/(t - \alpha s)$  and  $t/(s - \alpha s)$ , which together with  $s + t = 1 + \varepsilon \xrightarrow{\varepsilon \rightarrow 0} 1$  yields

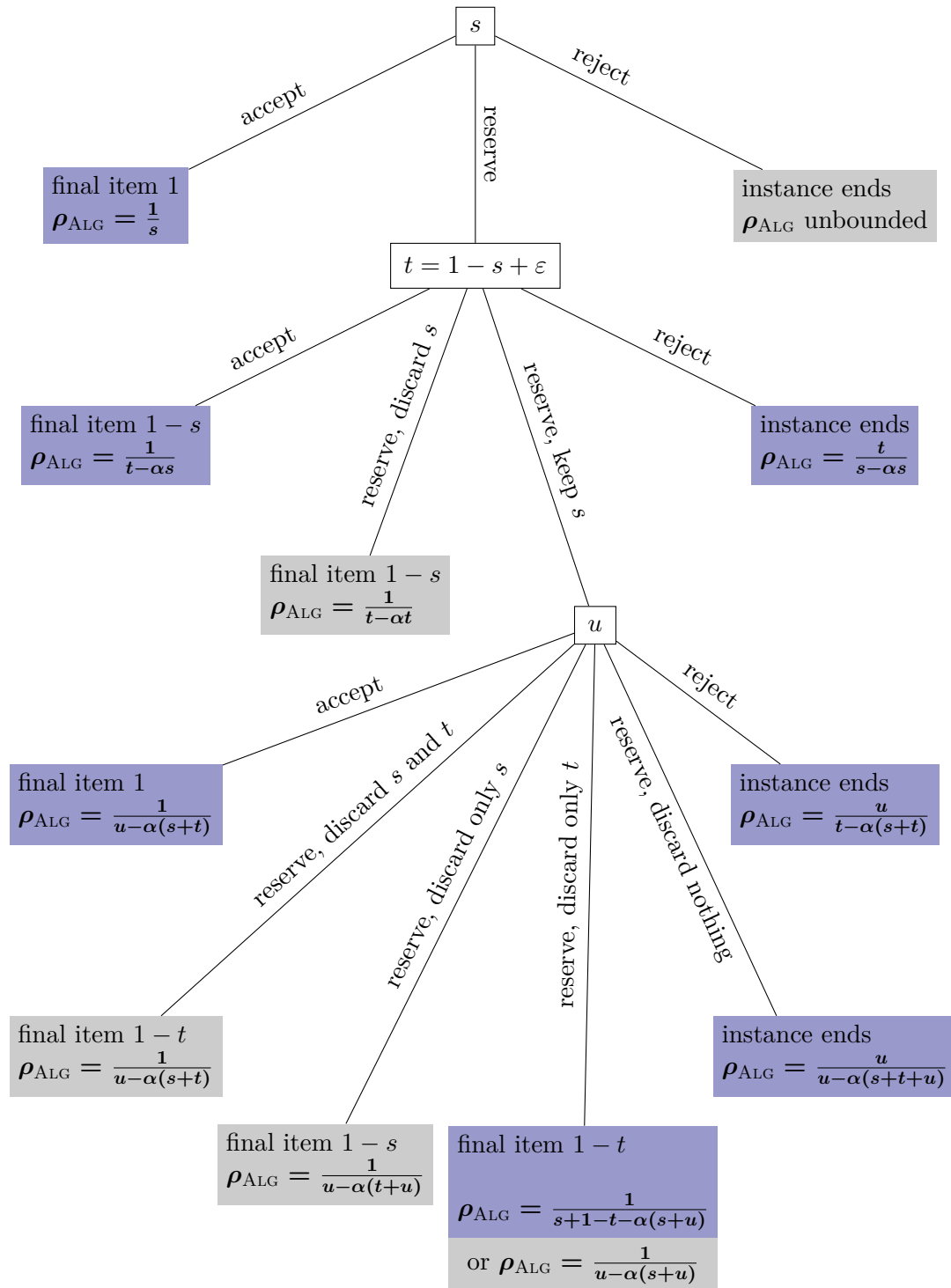
$$s = \frac{3 - \sqrt{5 - 4\alpha}}{2(1 + \alpha)} = \frac{2}{3 + \sqrt{5 - 4\alpha}} \text{ and } t = \frac{\sqrt{5 - 4\alpha} - 1 + 2\alpha}{2(1 + \alpha)},$$

as the following calculation shows:

$$\begin{aligned} t = 1 - s \quad & \frac{1}{t - \alpha s} = \frac{t}{s - \alpha s} && \iff \\ & \frac{1}{1 - s(1 + \alpha)} = \frac{1 - s}{s(1 - \alpha)} && \iff \\ & s(1 - \alpha) = 1 - s(1 + \alpha) - s(1 - s(1 + \alpha)) && \iff \\ & 0 = s^2(1 + \alpha) - 3s + 1 \\ & s < 1 \quad \implies s = \frac{3 - \sqrt{5 - 4\alpha}}{2(1 + \alpha)}. \end{aligned}$$

We now have

$$\frac{1}{t - \alpha s} = \frac{t}{s - \alpha s} = \frac{1 + \sqrt{5 - 4\alpha}}{2(1 - \alpha)}.$$



**Figure 4.2.** The adversarial instance class with three items  $s < t < u$ . Instances that yield worse or equal competitive ratios than other instances are marked with gray boxes.

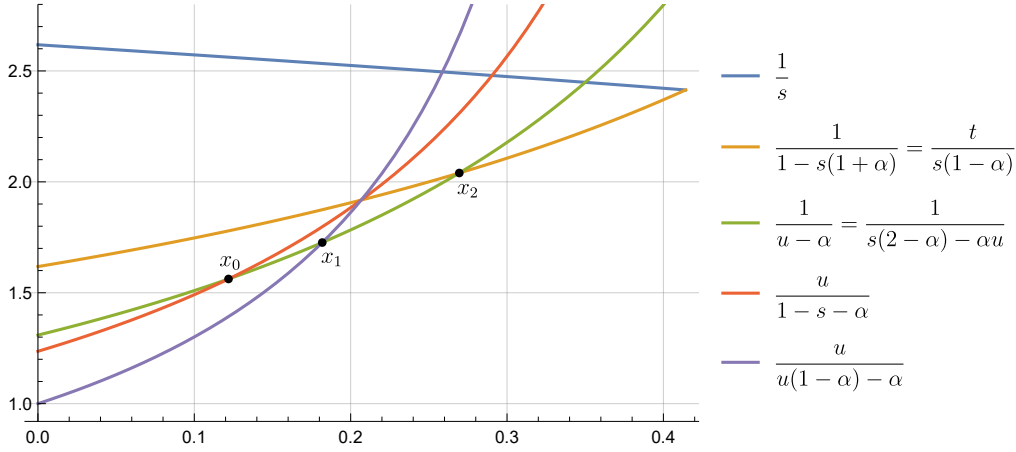
With these values for  $s$  and  $t$  and with the restriction that  $\alpha \leq \sqrt{2} - 1$ , we have  $1/(t - \alpha s) \leq 1/s$ ; in other words, accepting the second element is always better than accepting the first. Therefore, writing  $t = 1 - s$  and  $s + t = 1$ , the best achievable competitive ratio now evaluates to

$$\min \left\{ \frac{1}{1 - s(1 + \alpha)}, \frac{1}{u - \alpha}, \frac{u}{1 - s - \alpha}, \frac{u}{u(1 - \alpha) - \alpha}, \frac{1}{s(2 - \alpha) - \alpha u} \right\},$$

where we know that the first expression in the minimum,  $1/(1 - s(1 + \alpha))$  is equal to  $(1 + \sqrt{5 - 4\alpha})/(2(1 - \alpha))$ . We continue by choosing  $u$  such that  $1/(u - \alpha)$  equals  $1/(s(2 - \alpha) - \alpha u)$ ; this yields

$$u = \frac{s(2 - \alpha) + \alpha}{1 + \alpha}.$$

We can plot the equations for  $\alpha \leq \sqrt{2} - 1$  in [Figure 4.3](#) to visualize which expressions realize the minimum competitive ratio. With these values for  $s$ ,  $t$ , and  $u$ , we have



**Figure 4.3.** The ratios achieved by choosing the various options.

$t/(s(1 - \alpha))$  as the smallest achievable competitive ratio until  $1/(u - \alpha)$  takes over this role for smaller values of  $\alpha$  and this expression in turn stays the smallest expression until  $u/(u(1 - \alpha) - \alpha)$  becomes smaller for even smaller values of  $\alpha$ . We show in [Appendix B](#) that there are  $x_0 < x_1 < x_2 \in (0, 1)$  such that

$$\frac{1}{u - \alpha} \leq \frac{u}{1 - s - \alpha} \quad (\text{green line under red line}) \quad \text{for } \alpha \in [x_0, 1] \quad (4.2)$$

$$\frac{u}{u(1 - \alpha) - \alpha} \leq \frac{1}{u - \alpha} \quad (\text{purple line under green line}) \quad \text{for } \alpha \in (0, x_1] \quad (4.3)$$

$$\frac{t}{s(1 - \alpha)} \leq \frac{1}{u - \alpha} \quad (\text{yellow line under green line}) \quad \text{for } \alpha \in [x_2, 1], \quad (4.4)$$

and such that these inequalities are reversed for  $\alpha \in (0, 1]$  but not in the mentioned intervals. Thus, we establish the situation depicted in [Figure 4.3](#), namely that the purple line is the minimal line for  $\alpha \in (0, x_1]$ , the green line is the minimal line for  $\alpha \in [x_1, x_2]$ , and the yellow line is the minimal line for  $\alpha \in [x_2, 1]$ .



We obtain that  $x_0 \approx 0.1219$ ,  $x_1 \approx 0.1818$ , and  $x_2 \approx 0.2695$  and conclude that the competitive ratio is at least

$$\begin{aligned} & \frac{u}{(1-\alpha)u-\alpha} \text{ for } \alpha \in (0, x_1], \\ & \frac{1}{u-\alpha} \text{ for } \alpha \in [x_1, x_2], \text{ and} \\ & \frac{1}{1-s(1+\alpha)} \text{ for } \alpha \in [x_2, \sqrt{2}-1], \end{aligned}$$

where  $u = (s(2-\alpha) + \alpha)/(1+\alpha)$  and  $s = (3 - \sqrt{5-4\alpha})/(2(1+\alpha))$ .

## 4.6 Upper Bounds for Small Reservation Factors

The upper bound presented in [Section 4.4](#) is not tight for  $\alpha \leq 1/4$  and does not, in particular, tend to 1 for  $\alpha$  tending to zero. In this section, we present an algorithm that improves this upper bound for  $\alpha < 0.11$  and tends to a competitive ratio of 1 for  $\alpha$  tending to zero, which is exactly the behavior one would expect.

Let us first describe informally how our algorithm, RESERVE SMALLALPHA, behaves. Let  $d$  be some small fraction yet to be determined. For ease of presentation, we assume that  $1/d \in \mathbb{N}$ . All elements are categorized into  $1/d$  intervals of size  $d$ , namely  $I_i = ((i-1)d, id]$  for  $i \in \{1, \dots, 1/d\}$ .

In each step, RESERVE SMALLALPHA temporarily reserves the new element and then keeps the smallest reserved subset for each interval: This means that for each interval, all elements of the subset of reserved elements whose total size lies in the interval and whose total size attains the minimal total size of all subsets whose total size lies in the interval are kept. In other words, the algorithm calculates all total sizes of all subsets of reserved elements and if such a size is the minimal size in an interval, all the elements in the subset are kept. All reserved elements that are not kept this way are discarded. If there is no reserved subset at all in the interval above or if the interval above does not exist, then RESERVE SMALLALPHA keeps the elements of both the smallest and the largest subset of an interval. As soon as there is a reserved subset of size at least  $1-2d$ , the algorithm packs and terminates. Formally, the algorithm is described in [Algorithm 7](#).

We now turn towards the analysis of RESERVE SMALLALPHA. Let  $I$  be an arbitrary instance. We will write OPT and GROSSGAIN<sub>ALG</sub> instead of OPT( $I$ ) and GROSSGAIN<sub>ALG</sub>( $I$ ), respectively. Moreover, we are going to use the following notation. Let  $A, B$  be subsets of the input. In our calculations, we use  $A$  to abbreviate  $\sum_{a \in A} a$ . We say  $A$  is smaller than  $B$  and we write  $A \leq B$  if  $\sum_{a \in A} a \leq \sum_{b \in B} b$ . We say that  $A$  is in an interval  $J$  if  $\sum_{a \in A} a \in J$ . We are going to make repeated use of the following observation.

**Observation 4.5.** *Elements of a set  $A$  are only discarded if there is a smaller set  $S \leq A$  in the same interval as  $A$  and a larger set  $L \geq A$  in the same interval or one interval above.*

This observation leads to the following main lemma.

**Algorithm 7** RESERVESMALLALPHA**Parameter:** Any  $\alpha \in (0, 1)$ .**Online Input:** A sequence  $I = (x_1, \dots, x_n)$  with the sizes of the  $n$  items.**Online Output:** A reserve  $R \subseteq I$  and return of a final packing.

---

```

1:  $R := \emptyset$ ;
2:  $R_{\text{temp}} := \emptyset$ ;
3: for  $k = 1, \dots, n$  do
4:    $R_{\text{temp}} = R \cup \{x_k\}$ 
5:   if  $\text{OPT}(R \cup \{x_k\}) \geq 1 - 2d$  then pack  $\text{OPT}(R \cup \{x_k\})$  and stop;
6:   for each interval  $I_i$  do
7:     if in the interval above,  $I_{i+1}$ , there is no subset  $B \subseteq R_{\text{temp}}$  within  $I_{i+1}$ 
then choose a smallest subset  $S \subseteq R_{\text{temp}}$  and a largest subset  $L \subseteq R_{\text{temp}}$  within
 $I_i$  (if such subsets exist) and mark all elements that belong to  $S \cup L$ ;
8:     else choose a smallest subset  $S \subseteq R_{\text{temp}}$  within  $I_i$  and mark all elements
that belong to  $S$ ;
9:   Discard all unmarked elements from  $R_{\text{temp}}$  and then remove all markers;
10:   $R := R_{\text{temp}}$ ;
11: Pack the reserved elements optimally

```

---

**Lemma 4.5.** *For each subset  $A$  of input items with  $A \leq 1 - 2d$ , either  $A$  is reserved or there are reserved subsets  $S \leq A \leq L$  such that  $L - S \leq 2d$ .*

*Proof.* We prove this by induction on the number  $n$  of items. The induction basis for  $n = 1$  is trivially true. Assume that the claim is true after step  $n - 1$  and a new item  $x_k$  arrives.

Consider the situation before the for-loop in line 6. The claim is true for all subsets that do not contain  $x_k$  because of the induction hypothesis. All subsets that contain  $x_k$  consist of  $x_k$  and a (possibly empty) subset that does not contain  $x_k$ . For each such  $A \cup \{x_k\}$ , there are  $S \leq A \leq L$  with  $L - S \leq 2d$  by the induction hypothesis and we have that  $S \cup \{x_k\} \leq A \cup \{x_k\} \leq L \cup \{x_k\}$  and  $L \cup \{x_k\} - S \cup \{x_k\} = L - S \leq 2d$ . Therefore, the claim is true before the for-loop in line 6.

We have to prove that the for-loop does not destroy this condition. Consider any subset  $A$  with the subsets  $S, L$  from the claim. In the worst case, elements from both  $S$  and  $L$  are discarded. However, then there are sets  $S_S \leq S \leq L_S$  and  $S_L \leq L \leq L_L$  with  $L_S - S_S \leq 2d$  and  $L_L - S_L \leq 2d$  that stay in the reserve by [Observation 4.5](#).

We have three cases: If  $L_S$  is larger than  $A$ , then  $S_S \leq S \leq A \leq L_S$  and we are done. If the set  $S_L$  is smaller than  $A$ , then  $S_L \leq A \leq L \leq L_L$  and we are done. Otherwise,  $L_S \leq A \leq S_L$  and  $S_L - L_S \leq L - S \leq 2d$ .  $\square$

With that groundwork in hand, we are prepared to analyze the maximal reservation cost and the size of the solution of RESERVESMALLALPHA. Recall that we defined  $\text{GROSSGAIN}_{\text{ALG}}$  as the size of the solution of the algorithm, i.e., the gain plus the reservation costs. We start with estimating the value of  $\text{GROSSGAIN}_{\text{ALG}}$  as a corollary.

**Corollary 4.1.** *If  $\text{OPT} \leq 1 - 2d$ , then  $\text{GROSSGAIN}_{\text{ALG}} = \text{OPT}$ , i.e., the size of the solution of Algorithm 7 is optimal.*

*Proof.* Lemma 4.5 is true for each subset of input items and for each step. Hence, it is true for OPT as well.  $\square$

**Corollary 4.2.** *If  $\text{OPT} > 1 - 2d$ , then  $\text{GROSSGAIN}_{\text{ALG}} \geq 1 - 2d$ .*

*Proof.* Consider the order of the elements of OPT and let  $x$  be the first element such that  $\text{OPT} > 1 - 2d$ . Let  $A$  be the set that is going to be OPT before  $x$  is presented. According to Lemma 4.5, there are sets  $S, L \subseteq R$  such that  $S \leq A \leq L$ . When  $x$  is presented, either  $1 \geq L \geq A$  or  $L > 1$ . In the first case, we are done. In the latter case, we know that  $S \leq A \leq 1$  and  $S \geq L - 2d > 1 - 2d$ .  $\square$

We now turn towards the reservation cost in the following lemma.

**Lemma 4.6.** *The reservation size of Algorithm 7 is at most*

$$R \leq \min \left\{ \frac{1}{2d} - d - \frac{1}{2}, \frac{3}{2} \text{OPT} + \frac{\text{OPT}^2}{2d} \right\}.$$

*Proof.* The largest reserved subset cannot exceed  $1 - 2d$ . If there are two reserved subsets in an interval, the interval above does not contain any reserved subset. Therefore, to bound the reservation size, it suffices to count only one reserved subset per interval and an additional subset for the topmost interval containing a reserved subset. This leads to

$$\begin{aligned} R &\leq (1 - 2d) + \sum_{i=1}^{(1-2d)/d} id \\ \sum_{i=1}^n i &= n(n+1)/2 &&= (1 - 2d) + d \frac{\frac{(1-2d)}{d} \left( \frac{(1-2d)}{d} + 1 \right)}{2} \\ &&&= (1 - 2d) + \frac{(1 - 2d)^2}{2d} + \frac{(1 - 2d)}{2} \\ &&&= \frac{3}{2}(1 - 2d) + \frac{(1 - 2d)^2}{2d} \\ &&&= \frac{3}{2} - 3d + \frac{1 - 4d + 4d^2}{2d} \\ &&&= \frac{3}{2} - 3d + \frac{1}{2d} - \frac{4d}{2d} + \frac{4d^2}{2d} \\ &&&= \frac{1}{2d} - d - \frac{1}{2}. \end{aligned}$$

For the second part, we observe that the largest reserved subset cannot exceed OPT either. Using the same argumentation as above, we have

$$\begin{aligned} R &\leq \text{OPT} + \sum_{i=1}^{\text{OPT}/d} id \\ &= \text{OPT} + d \frac{\frac{\text{OPT}}{d} \left( \frac{\text{OPT}}{d} + 1 \right)}{2} \\ &= \text{OPT} + \frac{\text{OPT}^2}{2d} + \frac{\text{OPT}}{2} \\ &= \frac{3}{2} \text{OPT} + \frac{\text{OPT}^2}{2d}. \end{aligned} \quad \square$$

**Theorem 4.3.** *The competitive ratio of Algorithm 7 is at most*

$$\left(1 - \sqrt{2\alpha(2-\alpha)} + \frac{\alpha}{2}\right)^{-1},$$

which is roughly  $1 + 2\sqrt{\alpha}$  for small  $\alpha$ .

*Proof.* We have two cases. After the algorithm chooses  $d$ , the adversary determines the value of  $\text{OPT}$  and we have either  $\text{OPT} > 1 - 2d$  or  $\text{OPT} \leq 1 - 2d$ . In the first case, we can assume  $\text{GROSSGAIN}_{\text{ALG}} \geq 1 - 2d$  by Corollary 4.2 and then use Lemma 4.6 to bound the competitive ratio  $r$  as follows:

$$r \leq \frac{\text{OPT}}{1 - 2d - \alpha\left(\frac{1}{2d} - d - \frac{1}{2}\right)}.$$

If we divide numerator and denominator by  $\text{OPT}$  and then only look at the denominator, which is the inverse  $\rho$  of the competitive ratio, we have

$$\rho \geq \frac{1}{\text{OPT}} \left(1 - 2d - \alpha\left(\frac{1}{2d} - d - \frac{1}{2}\right)\right). \quad (4.5)$$

The value in the outer parentheses is non-negative for any  $d$  between  $\alpha/(2-\alpha)$  and  $1/2$  since

$$\begin{aligned} 0 \leq 1 - 2d - \alpha\left(\frac{1}{2d} - d - \frac{1}{2}\right) &\iff \\ \frac{1}{2d} - d - \frac{1}{2} > 0 \text{ for } d < \frac{1}{2} &\quad \alpha \leq \frac{1 - 2d}{\frac{1}{2d} - d - \frac{1}{2}} \\ &= \frac{2d(1 - 2d)}{1 - 2d^2 - d} \\ &= \frac{2d(1 - 2d)}{(1 - 2d)(d + 1)} \\ &= \frac{2d}{d + 1}. \end{aligned}$$

Therefore, inequality (4.5) is minimized for  $\text{OPT}$  as large as possible, which is  $\text{OPT} = 1$ . Thus, we have

$$\rho \geq 1 - 2d - \alpha\left(\frac{1}{2d} - d - \frac{1}{2}\right). \quad (4.6)$$

In the second case, we have  $\text{GROSSGAIN}_{\text{ALG}} = \text{OPT}$  by Corollary 4.1 and then again use Lemma 4.6 to write:

$$\begin{aligned} \rho &\geq \frac{\text{OPT}}{\text{OPT}} - \frac{\alpha}{\text{OPT}} \left(\frac{3}{2}\text{OPT} + \frac{\text{OPT}^2}{2d}\right) \\ &= 1 - \frac{3\alpha}{2} - \frac{\alpha\text{OPT}}{2d} \\ \text{OPT} \leq 1 &\quad \geq 1 - \frac{3\alpha}{2} - \frac{\alpha}{2d}. \end{aligned} \quad (4.7)$$

The algorithm wants to choose  $d$  such that the minimum of the two expressions (4.6) and (4.7) is maximized. We check for which values of  $d$  the bound from the second case is larger than the bound from the first case:

$$\begin{aligned} 1 - \frac{3\alpha}{2} - \frac{\alpha}{2d} &\geq 1 - 2d - \frac{\alpha}{2d} + \alpha d + \frac{\alpha}{2} && \iff \\ 2d - \alpha d &\geq 2\alpha && \iff \\ d &\geq \frac{2\alpha}{2 - \alpha}. \end{aligned}$$

Thus, if  $d \geq 2\alpha/(2 - \alpha)$ , the adversary is always going to choose OPT such that first case occurs, that is,  $\text{OPT} > 1 - 2d$ . We see that the best choice for  $d$  in case 1 satisfies this condition; therefore, it suffices to consider case 1. Case 1 is maximized for  $d = \sqrt{\alpha/(2(2 - \alpha))}$ , as we see by taking the derivative:

$$\begin{aligned} \frac{d}{dd} \left( 1 - 2d - \frac{\alpha}{2d} + \alpha d + \frac{\alpha}{2} \right) &\stackrel{!}{=} 0 && \iff \\ -2 + \frac{\alpha}{2d^2} + \alpha &= 0 && \implies (d \geq 0) \\ d &= \sqrt{\frac{\alpha}{2(2 - \alpha)}}. \end{aligned}$$

Inserting this value for  $d$  yields

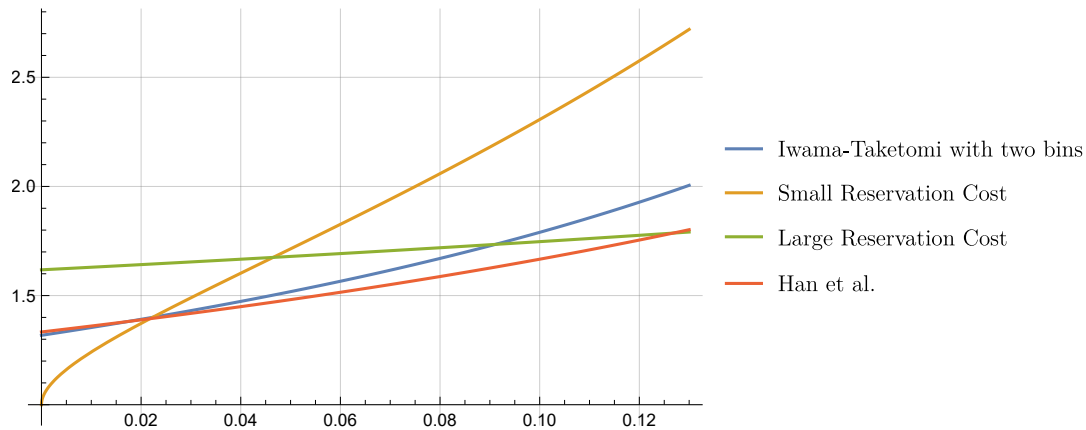
$$\begin{aligned} \rho &\geq 1 - 2\sqrt{\frac{\alpha}{2(2 - \alpha)}} - \frac{\alpha}{2}\sqrt{\frac{2(2 - \alpha)}{\alpha}} + \alpha\sqrt{\frac{\alpha}{2(2 - \alpha)}} + \frac{\alpha}{2} \\ \text{Group terms} &= 1 - (2 - \alpha)\sqrt{\frac{\alpha}{2(2 - \alpha)}} - \sqrt{\frac{\alpha}{2}}\sqrt{2 - \alpha} + \frac{\alpha}{2} \\ x \cdot \sqrt{1/x} = \sqrt{x} &= 1 - \sqrt{\frac{\alpha}{2}}\sqrt{2 - \alpha} - \sqrt{\frac{\alpha}{2}}\sqrt{2 - \alpha} + \frac{\alpha}{2} \\ \text{Group terms} &= 1 - \sqrt{2\alpha(2 - \alpha)} + \frac{\alpha}{2}. \end{aligned}$$

□

The bound from [Theorem 4.3](#) for Algorithm 7 is depicted in [Figure 4.4](#) together with the bound for Algorithm 6, the bound derived by Iwama and Taketomi [33] for knapsack with removal and two bins, and the bound by Han et al. [30] for knapsack with a resource buffer. We see that the algorithm derived in this section is superior for  $\alpha < 0.05$  to the algorithm for a large reservation factor, but it is only if  $\alpha$  is smaller than 0.02 that our algorithm yields better results than the bound from Iwama and Taketomi and the even better bound by Han et al.

## 4.7 Lower Bound for Small Reservation Factors

We saw in [Section 4.5](#) how a lower bound can be achieved by presenting a small set of items, considering all possible choices of an algorithm, and then choosing the



**Figure 4.4.** The upper bound from this section compared with the upper bound from Section 4.4 and the bounds from Iwama and Taketomi [33] and Han et al. [30] translated into our model.

value of the items such that the competitive ratio is maximized. For large reservation factors, our lower bounds are tight. For small reservation factors, however, they are not. The straightforward way to improve our analysis for small reservation factors consists of creating hard instances with more items. However, the number of choices of an algorithm grows exponentially with the number of items presented. Even if we could argue that some choices are always worse than other choices—and thus factually eliminate an exponential growth of cases to be considered—the calculations necessary to find good values for the items would become more and more cumbersome. Therefore, we present in this section a general lower bound that works for arbitrarily small reservation factors.

Let  $q \in (\frac{1}{2}, 1)$  be defined later. As prefix, the adversary presents the items  $x \in [\frac{1}{2}, q]$  in ascending order. Of course, this does not really make sense, but let us just assume for the ease of the argument that we can talk for each number  $r \in [\frac{1}{2}, q]$  about an item  $x$  of size  $r$ . This would have to be realized by presenting  $\frac{1}{2}, \frac{1}{2} + \varepsilon, \frac{1}{2} + 2\varepsilon, \dots$  and then letting  $\varepsilon$  go to zero. The prefix stops as soon as the adversary has presented an item  $x$ , the algorithm has given its answer (pack, discard, reserve and potentially discard several previously reserved items), and one of the following conditions applies:

1. ALG packs any item, in which case the adversary then presents an item of size 1. We will omit this case in the rest of the discussion.
2. ALG has discarded the item  $\frac{1}{2}$ , in which case the adversary then presents another item of size  $\frac{1}{2}$ .
3. The largest gap of unreserved items is at least  $1 - x$ , that is, there is an interval  $[a, b]$  of size at least  $b - a \geq 1 - x$  such that no item within this interval is reserved.

In the last case, the adversary looks at the items the algorithm reserved and considers the largest sequential gap among the reserved items, i.e., the largest distance  $y - z$  where both  $y$  and  $z$  (with  $y > z$ ) are in the reserve but no item between  $z$  and  $y$  is in the reserve. Then, the adversary presents an item of size  $1 - y$ , where  $y$  is the size

of the largest item in the gap. Note that since the items are presented in ascending order, Condition 3 demands with each new presented item that the algorithm has a smaller gap among its reserved items than before in order that a new item is presented. For example, with the first item, which is  $x = 1/2$ , gaps of size  $1/2$  are allowed. When later item  $x = 3/4$  is presented, only gaps of size  $1/4$  are allowed. Therefore, intuitively speaking, with each new item, it is more and more likely that this condition does not hold.

Clearly, the optimal solution has a gain of 1. For the analysis of the algorithm, we first look at the total reservation size of any algorithm that has no gap of size  $1 - s$  for some  $s \in (0, 1)$ .

**Lemma 4.7.** *If an algorithm ALG has the item  $\frac{1}{2}$  in its reserve and no gap (of unreserved items) larger than  $1 - s$  anywhere between  $\frac{1}{2}$  and  $x$ , then the total reservation size of ALG is at least*

$$R(x, s) := \frac{1}{4} - \frac{x}{2} + \frac{x^2 - \frac{1}{4}}{2(1-s)}.$$

*Proof.* An algorithm reserves at least the item  $\frac{1}{2}$  and the items  $x - (1 - s), x - 2(1 - s), \dots, x - n_x(1 - s)$ , where  $n_x$  is the largest natural number such that  $x - n_x(1 - s)$  is still larger than  $\frac{1}{2}$ . In symbols, we have  $x - n_x(1 - s) > \frac{1}{2}$ . This means that

$$n_x \geq \left\lfloor \frac{x - \frac{1}{2}}{1 - s} \right\rfloor \geq \frac{x - \frac{1}{2}}{1 - s} - 1.$$

Summing up the reserved items and omitting the ceilings on  $n_x$ , we find that the reservation size of ALG is at least

$$\frac{1}{2} + \sum_{i=1}^{n_x} (x - i(1 - s)) = \frac{1}{2} + n_x \cdot x - (1 - s) \frac{n_x(n_x + 1)}{2}.$$

Since the factor that is multiplied by  $n_x$  is non-negative, we can use our lower bound on  $n_x$  to write

$$\begin{aligned} \frac{1}{2} + \sum_{i=1}^{n_x} (x - i(1 - s)) &\geq \frac{1}{2} + \left( \frac{x - \frac{1}{2}}{1 - s} - 1 \right) x - \frac{1 - s}{2} \left( \frac{x - \frac{1}{2}}{1 - s} - 1 \right) \left( \frac{x - \frac{1}{2}}{1 - s} \right) \\ &= \frac{1}{2} + \frac{x^2 - \frac{x}{2}}{1 - s} - x - \frac{(1 - s)}{2} \frac{\left( x - \frac{1}{2} \right)^2}{(1 - s)^2} + \frac{x - \frac{1}{2}}{2} \\ &= \frac{1}{4} - \frac{x}{2} + \frac{1}{1 - s} \left( x^2 - \frac{x}{2} - \frac{x^2 - x + \frac{1}{4}}{2} \right) \\ &= \frac{1}{4} - \frac{x}{2} + \frac{x^2 - \frac{1}{4}}{2(1 - s)}. \quad \square \end{aligned}$$

We first analyze the case that only part of the prefix is presented, that is, Conditions 2 or 3 are applied at some point, and then analyze the case where the whole prefix is presented. We have the following lemma.

**Lemma 4.8.** *Let  $x < q$ . If only the prefix  $[\frac{1}{2}, x]$  is presented, ALG has a gain of at most*

$$x \cdot \text{OPT} - \alpha R(x, x).$$

*Proof.* The presentation of the prefix did not stop when the item right before  $x$ , say  $x - \varepsilon$ , was presented. This means that, when  $x - \varepsilon$  was presented, there was no gap of size  $1 - x + \varepsilon$  and the item  $\frac{1}{2}$  had not been discarded. Hence, the reservation size was at least  $R(x - \varepsilon, x - \varepsilon)$ .

Moreover, the prefix stopped when  $x$  was presented. We distinguish two cases. First, if ALG discarded  $\frac{1}{2}$ , an item of size  $\frac{1}{2}$  is presented as last item and the best ALG can do is to take the largest presented item, which is of size  $x$ . Therefore, the gain of the optimal solution is 1 and the gross gain of ALG is  $x = x \cdot \text{OPT}$ .

Second, if ALG has a gap of at least  $1 - x$ , the prefix stops and the complement of the largest item within the gap is presented as last item. The best the algorithm can do is to pack this complement, say  $1 - y$ , and the largest reserved item smaller than  $y$ , say  $z \leq y - (1 - x)$ . This leads to a gross gain of at most  $1 - y + z \leq 1 - y + y - (1 - x) = x = x \cdot \text{OPT}$ .  $\square$

Second, we analyze the case where the whole prefix is presented. We have the following lemma.

**Lemma 4.9.** *At the end of the prefix, ALG has a gain of at most*

$$\max_{s \in (q, 1)} (s \cdot \text{OPT} - \alpha R(q, s)).$$

*Proof.* We first note that the variable  $s$  stands for the desired gross gain. We prove that in order to achieve a gross gain of  $s \cdot \text{OPT}$ , ALG cannot have a gap of unreserved items of size  $y > 1 - s$ . Together with Lemma 4.7, this proves the claim.

Consider the largest item  $t$  of the largest gap and assume towards contradiction this largest gap is of size  $y > 1 - s$ . At the end of the prefix, the adversary presents  $1 - t$ . The best the algorithm can do is to take  $1 - t$  and the largest reserved item smaller than  $t$ , which is by assumption an item of size  $t - y$ . So the gross gain of ALG is at most  $1 - t + t - y < 1 - t + t - (1 - s) = s = s \cdot \text{OPT}$ , that is, strictly smaller than  $s \cdot \text{OPT}$ .  $\square$

Combining the case that the prefix is interrupted with the case that the whole prefix is presented leads to the following bound on the competitive ratio, which is depicted in Figure 4.5 together with some upper bounds from Figure 4.3.

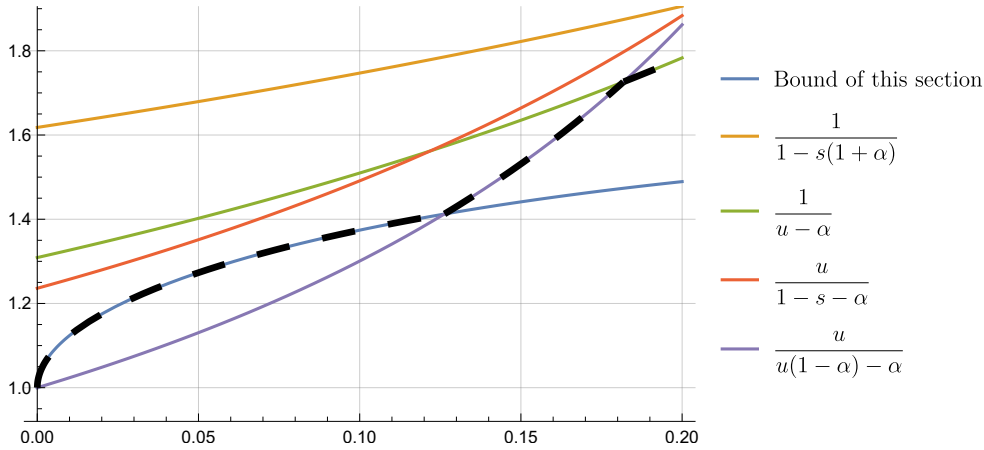
**Theorem 4.4.** *The competitive ratio of any algorithm is bounded from below by*

$$\left( 1 - \sqrt{\alpha \left( \frac{3}{2} - \sqrt{\frac{6\alpha}{1+\alpha} + \frac{3\alpha}{4(1+\alpha)}} \right)} + \frac{\alpha}{4} \left( 1 - \sqrt{\frac{3\alpha}{2(1+\alpha)}} \right) \right)^{-1},$$

which is roughly  $1 + \sqrt{\frac{3\alpha}{2}}$  for small  $\alpha$ .

*Proof.* Either the prefix stops at  $x < q$  and the gain of the algorithm is  $x \cdot \text{OPT} - \alpha R(x, x)$  according to Lemma 4.8; or the whole prefix is presented and the gain of the algorithm is  $\max_{s \in (q, 1)} (s \cdot \text{OPT} - \alpha R(q, s))$  according to Lemma 4.9. The adversary is going to choose  $q$  such that it is always better for the algorithm to let the whole prefix be presented. However, let us first analyze the case where not the whole prefix is presented.





**Figure 4.5.** The blue line indicates the bound presented in this section. The competitive ratio is the maximum between the blue line and the minimum of the other lines, marked with the dashed black line.

We start by simplifying the expression  $x - \alpha R(x, x)$ .

$$\begin{aligned}
 x - \alpha R(x, x) &= \\
 \text{Lemma 4.7} \quad &= x - \alpha \left( \frac{1}{4} - \frac{x}{2} + \frac{x^2 - \frac{1}{4}}{2(1-x)} \right) \\
 \text{Add } 0 = -3/4 + 3/4 \quad &= x - \alpha \left( \frac{1}{4} - \frac{x}{2} + \frac{x^2 - 1 + 3/4}{2(1-x)} \right) \\
 x^2 - 1 = (x-1)(x+1) \quad &= x - \alpha \left( \frac{1}{4} - \frac{x}{2} + \frac{(x-1)(x+1) + 3/4}{2(1-x)} \right) \\
 \text{Split fraction} \quad &= x - \alpha \left( \frac{1}{4} - \frac{x}{2} - \frac{x+1}{2} + \frac{3}{8(1-x)} \right) \\
 &= x - \alpha \left( -\frac{1}{4} - x + \frac{3}{8(1-x)} \right).
 \end{aligned}$$

We take the first derivative of this expression to determine possible extrema:

$$\begin{aligned}
 \frac{d}{dx} \left( x - \alpha \left( -\frac{1}{4} - x + \frac{3}{8(1-x)} \right) \right) &= 1 - \frac{3\alpha}{8(1-x)^2} + \alpha \stackrel{!}{=} 0 && \iff \\
 8(1+\alpha)(1-x)^2 &= 3\alpha && \iff \\
 x^2 - 2x + 1 - \frac{3\alpha}{8(1+\alpha)} &= 0
 \end{aligned}$$

The two possible values for  $x$  are  $x_{1,2} = 1 \pm \sqrt{\frac{3\alpha}{8(1+\alpha)}}$ . Since  $x$  is at most 1, we obtain

$$x = 1 - \sqrt{\frac{3\alpha}{8(1+\alpha)}}.$$

We take the second derivative to confirm that this is indeed a maximum:

$$\frac{d^2}{dx^2} \left( x - \alpha \left( -\frac{1}{4} - x + \frac{3}{8(1-x)} \right) \right) = \frac{-3\alpha}{4(1-x)^3} \stackrel{!}{<} 0.$$

This means that, if  $q$  is smaller than  $q_{\max} := 1 - \sqrt{\frac{3\alpha}{8(1+\alpha)}}$ , the maximum of  $x - \alpha R(x, x)$  is achieved at  $q$ , so the maximum is  $q - \alpha R(q, q)$ , which is trivially always smaller or equal than  $\max_{s \in [q, 1]} s \cdot \text{OPT} - \alpha R(q, s)$ . Hence, by choosing  $q \leq q_{\max}$ , the adversary ensures that ALG waits for the presentation of the whole prefix.

Therefore, we now turn towards the case where the whole prefix is presented and the algorithm wants to maximize  $s \cdot \text{OPT} - \alpha R(q, s)$ . The extremum of  $s - \alpha R(q, s)$  is given by:

$$\begin{aligned} \frac{d}{ds} \left( s - \alpha \left( \frac{1}{4} - \frac{q}{2} + \frac{q^2 - \frac{1}{4}}{2(1-s)} \right) \right) &= 1 - \frac{\alpha(q^2 - 1/4)}{2(1-s)^2} \stackrel{!}{=} 0 && \iff \\ &2(1-s)^2 = \alpha(q^2 - 1/4) && \iff \\ \text{Divide by 2, then take root} \quad 1-s &= \sqrt{\frac{\alpha(q^2 - 1/4)}{2}} && \iff \\ &1 - \sqrt{\frac{\alpha(q^2 - 1/4)}{2}} = s. \end{aligned}$$

We check the second derivative to confirm that this is indeed the maximum:

$$\begin{aligned} \frac{d^2}{ds^2} \left( s - \alpha \left( \frac{1}{4} - \frac{q}{2} + \frac{q^2 - \frac{1}{4}}{2(1-s)} \right) \right) &= \frac{-\alpha(q^2 - 1/4)}{(1-s)^3} \stackrel{!}{<} 0 \\ \alpha \geq 0, s \leq 1 &\implies q^2 - 1/4 > 0. \end{aligned}$$

This is true for any  $q$  larger than  $\frac{1}{2}$ , which is the case in our situation. Inserting the found maximum back into the gain  $s - \alpha R(q, s)$  yields

$$\begin{aligned} s - \alpha R(q, s) &\leq 1 - \sqrt{\frac{\alpha(q^2 - 1/4)}{2}} \\ &\quad - \alpha \left( \frac{1}{4} - \frac{q}{2} + \frac{q^2 - \frac{1}{4}}{2 \left( 1 - \left( 1 - \sqrt{\frac{\alpha(q^2 - 1/4)}{2}} \right) \right)} \right) \\ &= 1 - \sqrt{\frac{\alpha(q^2 - 1/4)}{2}} - \frac{\alpha}{4} + \frac{\alpha q}{2} - \frac{\alpha(q^2 - 1/4)}{2\sqrt{\frac{\alpha(q^2 - 1/4)}{2}}} \\ \text{Since } x/\sqrt{x} &= \sqrt{x} && = 1 - \sqrt{\frac{\alpha(q^2 - 1/4)}{2}} - \frac{\alpha}{4} + \frac{\alpha q}{2} - \sqrt{\frac{\alpha(q^2 - 1/4)}{2}} \\ &= 1 - \sqrt{2\alpha(q^2 - 1/4)} + \frac{\alpha}{2} \left( q - \frac{1}{2} \right). \end{aligned}$$

This expression decreases with increasing  $q$ , since the decreasing part decreases approximately like  $\sqrt{2\alpha}$  while the increasing part increases only like  $\alpha/2$ . Therefore, the best strategy for the adversary is to choose  $q$  as large as possible, namely  $q = q_{\max}$ . Thus, we obtain the following maximal gain:

$$s - \alpha R(q, s) \leq 1 - \sqrt{2\alpha \left( \left( 1 - \sqrt{\frac{3\alpha}{8(1+\alpha)}} \right)^2 - 1/4 \right)} + \dots$$

$$\begin{aligned}
& \dots + \frac{\alpha}{2} \left( \left( 1 - \sqrt{\frac{3\alpha}{8(1+\alpha)}} \right) - \frac{1}{2} \right) \\
&= 1 - \sqrt{\frac{3\alpha}{2} - 4\alpha \sqrt{\frac{3\alpha}{8(1+\alpha)}} + \frac{6\alpha^2}{8(1+\alpha)}} + \frac{\alpha}{4} - \frac{\alpha}{2} \sqrt{\frac{3\alpha}{8(1+\alpha)}} \\
&= 1 - \sqrt{\alpha \left( \frac{3}{2} - \sqrt{\frac{6\alpha}{1+\alpha}} + \frac{3\alpha}{4(1+\alpha)} \right)} + \frac{\alpha}{4} \left( 1 - \sqrt{\frac{3\alpha}{2(1+\alpha)}} \right).
\end{aligned}$$

The claimed competitive ratio follows immediately. The claimed estimation comes from the fact that the second summand is majorized by the first one and the Taylor expansion of  $1/(1-z)$  around 0 yields  $1+z+O(z^2)$ .  $\square$

## 4.8 Conclusion

This chapter analyzed the simple online knapsack problem with a “storeroom” reservation model. While it is natural for some applications to pay reservation costs depending on the size of the reserved item, there are applications where it is more realistic to pay only for the maximum total size of items reserved. This latter cost definition was the one we used here to analyze online knapsack with reservation.

For cases where the reservation factor  $\alpha$  is larger than approximately 0.26, we provided tight upper and lower bounds on the competitiveness. Interestingly, these bounds coincide with the bounds by Böckenhauer et al. [3] for the case where the reservation cost depends on the size of every single reserved item—although our model is less restrictive since reserved items can be exchanged for smaller items at no additional cost. This shows that having the ability to exchange reserved items for other items is only helpful when the reservation cost is small. Considering our algorithm for large reservation factors, the reason for this seems to be that reserving more than the knapsack size is only attractive for quite small reservation costs.

For cases where the reservation factor is small, it seems that good algorithms have to over-reserve, that is, reserve more than the knapsack size. Moreover, the instance family for the lower bound has to use more than 4 items, which results in a very extensive case distinction. Therefore, we devised a more general algorithm that works well for small reservation factors and used a different technique in establishing a lower bound. The lower bound we thus discovered grows approximately like  $1 + \sqrt{3\alpha/2}$ ; the upper bound like  $1 + 2\sqrt{\alpha}$ . Therefore, the upper bound grows  $\sqrt{8/3} \approx 1.63$  times faster than the lower bound. We conjecture that an even more general algorithm might prove more successful; for example, an algorithm that bases its decision on all possible optimal solutions for the current input plus an arbitrary future item. However, the analysis for such a general algorithm seems to be rather difficult.

Moreover, it would be interesting to see the behavior of other online problems with our reservation model. Böckenhauer et al. [3] suggested online call admission problems in networks for their model, which seems natural to analyze with our model as well, given the relationship to the time-cost model mentioned in the introduction.

Furthermore, restricting the study of online problems with reservation to polynomial-time algorithms seems to be worthwhile as well. Clearly, without any such restriction, the competitive ratio always goes from 1 for  $\alpha = 0$  to  $f(n)$  for  $\alpha = 1$ , where  $f(n)$

is the competitive ratio of the online problem without advice. By just considering polynomial-time algorithms, one might discover for which value of  $\alpha$  further progress on NP-hard problems appears difficult.

*And the end of one journey is  
the beginning of another.*

Margaret Atwood

# 5

## Conclusion

We started our journey to the edge of the online world by studying hashing as an online problem. Since online problems are gauged by their performance on the worst possible instance, we strived for algorithms that are suitable for every possible input set  $S$ . We expanded the notion of perfect  $k$ -hashing to the case where the input set is larger than the hash table size  $m$  and coined the term  $c$ -ideality to create a clear connection to  $c$ -competitiveness. As no single hash function can distribute every set well enough to be  $c$ -ideal, we looked for  $c$ -ideality among hash function families. This led us to the study of  $H_c$ , the minimal number of functions necessary to constitute a  $c$ -ideal hash function family. In other words, we analyzed how many functions have to be in a hash function family such that, for every  $S \in \mathcal{S}_n$ , there is a function that has cost at most  $c$ . This is an intriguing mathematical question on its own and is particularly interesting for  $c$  at most  $\log(m)/\log \log(m)$ , since this is the competitiveness of randomized hashing algorithms. Given a  $c$ -ideal hash function family  $\mathcal{H}$ , an online algorithm with advice can read, for every input  $S$ ,  $\lceil \log |\mathcal{H}| \rceil$  advice bits to select the best hash function among  $\mathcal{H}$  and thus perform at most  $c$  times worse than an optimal offline solution. With our bounds, we found that the advice complexity of hashing depends linearly on  $m$  and only logarithmically on  $n$ ; the universe size can be neglected for many applications. In particular, to be on par with randomized algorithms, it suffices to read  $O(\ln \ln u + \ln n)$  advice bits. Further progress is necessary to tighten the gaps between our upper and lower bounds and we hope that future research can benefit from and improve upon our analysis.

The next stage of our journey took us to graph exploration. We tightened the online restriction as much as possible. First, we assumed that the agent is oblivious and thus basically just an automaton. Second, we did not allow that the agent knows through which edge it entered a vertex. Third, to ensure that the order of the presented neighbors cannot help either, we let the adversary determine an arbitrary order for each visit of the agent. With such severe restrictions, successful exploration is not possible anymore. Therefore, we slightly relaxed our constraints by endowing the agent with the ability to permanently color a vertex and to see the colors of

the neighboring vertices. To measure how difficult the problem is, we counted the number of colors necessary and sufficient to explore all graphs. We were able to completely characterize the difficulty of this problem. We found that  $n - 1$  colors are necessary and sufficient on graphs of size  $n$  to be able to explore all graphs. Even if the size of the graph at hand is known and even if only bipartite graphs have to be explored, the number of colors needed and sufficient is almost as high. Since only 3 colors are needed to explore all trees, we wondered what exactly it is that makes a graph difficult. Having considered treewidth, feedback vertex set, number of cycles, and other graph properties, we discovered that the key property is, surprisingly, the circumference of a graph. We proved that  $\min\{2k - 3, n - 1\}$  colors are necessary and  $\min\{2k - 1, n - 1\}$  colors are sufficient on graphs of size  $n$  and circumference  $k$  to explore all graphs. This situation changes radically if only certain graph classes have to be explored or if colored vertices may be colored again. We gave an example of a graph class where four colors are already sufficient for the exploration, although the graphs have maximal circumference. Moreover, we proved that, if recoloring vertices is allowed, seven colors are sufficient to explore all graphs, which stands in stark contrast to the linear amount of colors needed without recoloring.

Having observed the advice complexity of hashing and enjoyed—and immediately after forgotten—the view from our graph exploration model, we continued our journey and explored the edge between online and offline problems consisting of the online knapsack problem with reservation. We studied how the additional ability to reserve items for a certain cost instead of just packing or reserving them changed the competitiveness of the online knapsack problem. For large and medium-sized reservation factors, we provided tight upper and lower bounds. Our algorithm is tight in the model by Böckenhauer et al. [3] as well and in fact unifies and simplifies their algorithms. For cases where the reservation factor  $\alpha$  is smaller than 0.26, however, our algorithm loses its edge quickly. Therefore, we devised an algorithm suited for small  $\alpha$  and proved that its competitiveness tends to one with  $\alpha$  tending to zero, which is exactly the behavior one would expect, seeing that the problem becomes essentially offline. We also provided a lower bound that, although growing at a slower rate than the upper bound, is very close to the upper bound for  $\alpha$  close to zero. Note that neither the model with removal cost by Han et al. [28] nor the model with reservation per item discussed above feature this smooth transition between free reservation/removal cost and reservation/removal cost equal to the size of the item. Therefore, we think that there is great potential for future research to explore our reservation model and reservation cost in general for other online problems. Moreover, in the case of our model, we would like to see, on the one hand, tight bounds for small reservation cost and, on the other hand, a general upper bound that provides the best possible competitive ratio over the whole range of  $\alpha$ .

# Appendix

## A Proof of Lemma 2.9

Let  $f(\ell_1, \dots, \ell_m) := \prod_{i=1}^m \frac{1}{\ell_i!}$ . Note that the relaxation used in the definition of  $f$  from  $\ell_1, \dots, \ell_m \in \mathbb{N}$  to  $0 \leq \ell_1, \dots, \ell_m \in \mathbb{R}$  can only increase the minimization range, decreasing the entire term further.

First, we use that  $f$  is extremal exactly if its logarithm  $g := \log(1/f) = \sum_{i=1}^m \log \ell_i!$  is extremal. We now imagine that this is a hyperplane in  $m$  dimensions, and we want to take the derivative and set it to zero in order to determine the minimum. The derivative in all dimensions corresponds to the gradient, which is  $\nabla g = (\partial_{\ell_1} \log \ell_1!, \dots, \partial_{\log \ell_m} \log \ell_m!)$ .

To calculate the derivative of a factorial, we use the Gamma function, which is defined as  $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$  and which satisfies  $\Gamma(z) = (z-1)!$ . The components of the gradient are thus the logarithmic derivative of  $\Gamma(z)$ . The derivative of the Gamma function is

$$\Gamma'(z) = \int_0^\infty \partial_z t^{z-1} e^{-t} dt = \int_0^\infty \partial_z e^{(z-1)\ln(t)} e^{-t} dt = \int_0^\infty \ln(t) t^{z-1} e^{-t} dt.$$

These values are strictly monotonically increasing because the second derivative, which is

$$\partial_z^2 \log z! = \partial_z \frac{\Gamma'(z)}{\Gamma(z)} = \frac{\Gamma(z)\Gamma''(z) - \Gamma'(z)\Gamma'(z)}{\Gamma(z)^2},$$

is strictly positive: To see that the numerator is positive, we use the Cauchy-Schwarz inequality, which states

$$\langle v, v \rangle \langle w, w \rangle - \langle v, w \rangle^2 \geq 0.$$

We can then take the scalar product on functionals

$$\langle v, w \rangle(z) := \int_0^\infty v(t)w(t) \frac{t^{z-1}}{e^t} dt$$

with  $v = 1$  and  $w = \ln$  and obtain exactly  $\Gamma(z)\Gamma''(z) - \Gamma'(z)\Gamma'(z) = \langle v, v \rangle \langle w, w \rangle - \langle v, w \rangle \langle w, v \rangle \geq 0$ .

The normal vector  $(1, \dots, 1)$  of the hyperplane defined by the side condition  $\ell_1 + \dots + \ell_m - n = 0$  is collinear with the gradient for  $\ell_1 = \dots = \ell_m = \alpha$  and nowhere else by the proven monotonicity. This shows that  $f$  has only a single inner extremum with value  $1/\alpha^m$  attained at  $\ell_1 = \dots = \ell_m = \alpha$ , since the projection of the gradient onto the hyperplane—which gives the direction of steepest ascent under the given restrictions—vanishes if and only if it is perpendicular to the hyperplane, that is, collinear with the normal vector.

We must also check the boundary for extrema. Recall from the statement of the lemma that our area is  $\Omega := \{(\ell_1, \dots, \ell_m) \in \mathbb{R}^m; \ell_1 + \dots + \ell_m = n \wedge 0 \leq \ell_1, \dots, \ell_m \leq d\}$ . All candidates  $(\ell_1, \dots, \ell_m) \in \partial\Omega$ —that is, on the boundary of our area—have at least one  $\ell_i$  equal to either zero or  $d$ . We can now repeat our entire argument with the new area being the boundary of  $\Omega$  and obtain that again, the extremal value is exactly if all  $\ell_i$  are equal except for the new boundary.

Iterating the entire argument  $m$  times shows that the global boundary extrema are attained whenever  $\ell_1, \dots, \ell_m \in \{0, d\}$ . Under this assumption, the restriction  $\ell_1 + \dots + \ell_m = n$  implies that  $\ell_i = d$  for exactly  $n/d = m/c$  of the  $m$  indices  $i \in \{1, \dots, m\}$  and  $\ell_i = 0$  for the rest, yielding the global minimal value  $(d!)^{m/c}$ .

## B Calculation for Intersection Points in Section 4.5

Note first that our definition of  $s$  is equivalent to  $s = \frac{2}{3 + \sqrt{5 - 4\alpha}}$ , which can be seen by multiplying our definition of  $s$  with  $1 = (3 + \sqrt{5 - 4\alpha}) / (3 + \sqrt{5 - 4\alpha})$ . We begin with inequality (4.4). We calculate:

$$\begin{aligned}
& \frac{t}{s(1-\alpha)} \leq \frac{1}{u-\alpha} && \iff \\
1/(t-\alpha s) = t/(s(1-\alpha)) & \quad \frac{1}{1-s-\alpha s} \leq \frac{1}{u-\alpha} && \iff \\
& \text{Invert both sides} && u-\alpha \leq 1-s(1+\alpha) && \iff \\
& \text{Definition of } u && \frac{s(2-\alpha)+\alpha}{1+\alpha} - \alpha \leq 1-s(1+\alpha) && \iff \\
& \text{Multiply with } (1+\alpha) && s(2-\alpha)+\alpha \leq (1+\alpha)^2 - s(1+\alpha)^2 && \iff \\
& \text{Add } s(1+\alpha)^2 && s(2-\alpha+(1+\alpha)^2) \leq -\alpha+(1+\alpha)^2 && \iff \\
& \text{Definition of } s && 2(2-\alpha+(1+\alpha)^2) \leq (-\alpha+(1+\alpha)^2)(3+\sqrt{5-4\alpha}) && \iff \\
& \text{Simplify left} && 4+2(-\alpha+(1+\alpha)^2) \leq (-\alpha+(1+\alpha)^2)(3+\sqrt{5-4\alpha}) && \iff \\
& \text{Subtract } 3(-\alpha+(1+\alpha)^2): && && \iff \\
& && 4 - ((1+\alpha)^2 - \alpha) \leq ((1+\alpha)^2 - \alpha)\sqrt{5-4\alpha} && \iff \\
& \text{Square} && (4 - ((1+\alpha)^2 - \alpha))^2 \leq ((1+\alpha)^2 - \alpha)^2(5-4\alpha) && \iff \\
& \text{Expand} && (3-\alpha-\alpha^2)^2 \leq (1+\alpha+\alpha^2)^2(5-4\alpha) && \iff \\
& \text{Expand and divide by 4} && 0 \leq -\alpha^5 + \alpha^4/2 - \alpha^3 + 3\alpha^2 + 3\alpha - 1 && 
\end{aligned}$$

We can determine numerically the unique root in the interval  $[0, 1]$  and obtain  $x_2 \approx 0.2695$ . We turn towards inequality (4.3):

$$\begin{aligned}
& \frac{u}{u(1-\alpha)-\alpha} \leq \frac{1}{u-\alpha} && \iff \\
& u^2 - u\alpha \leq u - u\alpha - \alpha && \iff \\
& u^2 - u + \alpha \leq 0
\end{aligned}$$



This is true for  $u \leq 1/2 + \sqrt{1/4 - \alpha}$  and we determine numerically that this holds for  $\alpha \leq x_0 \approx 0.1818$ , which is the unique positive root between 0 and 1 of the polynomial  $u^2 - u + \alpha$ .

Finally, we turn towards the inequality (4.2), and rearrange the expression as follows:

$$\begin{aligned}
 & \frac{u}{1-s-\alpha} \leq \frac{1}{u-\alpha} && \iff \\
 \text{Definition of } u & \frac{s(2-\alpha)+\alpha}{(1+\alpha)(1-s-\alpha)} \leq \frac{1}{\frac{s(2-\alpha)+\alpha}{1+\alpha} - \alpha \frac{1+\alpha}{1+\alpha}} && \iff \\
 & \frac{s(2-\alpha)+\alpha}{(1+\alpha)(1-s-\alpha)} \leq \frac{1+\alpha}{s(2-\alpha)-\alpha^2} && \iff \\
 & (s(2-\alpha)+\alpha)(s(2-\alpha)-\alpha^2) \leq (1+\alpha)^2(1-s-\alpha) && \iff \\
 & s^2(2-\alpha)^2 + s(2-\alpha)\alpha(1-\alpha) - \alpha^3 \leq (1+\alpha)^2(1-\alpha) - s(1+\alpha)^2
 \end{aligned}$$

By moving all terms on one side, we obtain the following polynomial in  $s$  and  $\alpha$ :

$$s^2(2-\alpha)^2 + s\alpha(1-\alpha)(2-\alpha)(1+\alpha)^2 - (1+\alpha)^2(1-\alpha) - \alpha^2 \leq 0$$

We now insert our definition of  $s$  and determine the first positive root of this polynomial numerically, which is  $x_0 \approx 0.1219$ .



# Bibliography

- [1] Andrew D. Barbour, Lars Holst, and Svante Janson. *Poisson approximation*. Oxford: Clarendon Press, 1992.
- [2] Simon R. Blackburn. Perfect hash families: Probabilistic methods and explicit constructions. *J. Comb. Theory, Ser. A*, 92(1):54–60, 2000. URL: <https://doi.org/10.1006/jcta.1999.3050>, doi:10.1006/jcta.1999.3050.
- [3] Hans-Joachim Böckenhauer, Elisabet Burjons, Juraj Hromkovič, Henri Lotze, and Peter Rossmanith. Online simple knapsack with reservation costs. In Markus Bläser and Benjamin Monmege, editors, *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*, volume 187 of *LIPICs*, pages 16:1–16:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL: <https://doi.org/10.4230/LIPICs.STACS.2021.16>, doi:10.4230/LIPICs.STACS.2021.16.
- [4] Hans-Joachim Böckenhauer, Jan Dreier, Fabian Frei, and Peter Rossmanith. Advice for online knapsack with removable items. *CoRR*, abs/2005.01867, 2020. URL: <https://arxiv.org/abs/2005.01867>, arXiv:2005.01867.
- [5] Hans-Joachim Böckenhauer, Dennis Komm, Richard Královic, and Peter Rossmanith. The online knapsack problem: Advice and randomization. *Theor. Comput. Sci.*, 527:61–72, 2014. URL: <https://doi.org/10.1016/j.tcs.2014.01.027>, doi:10.1016/j.tcs.2014.01.027.
- [6] Hans-Joachim Böckenhauer, Dennis Komm, Rastislav Královic, Richard Královic, and Tobias Mömke. On the advice complexity of online problems. In Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra, editors, *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, volume 5878 of *Lecture Notes in Computer Science*, pages 331–340. Springer, 2009. URL: [https://doi.org/10.1007/978-3-642-10631-6\\_35](https://doi.org/10.1007/978-3-642-10631-6_35), doi:10.1007/978-3-642-10631-6\_35.
- [7] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.
- [8] Joan Boyar, Lene M. Favrholdt, Christian Kudahl, Kim S. Larsen, and Jesper W. Mikkelsen. Online algorithms with advice: A survey. *ACM Comput. Surv.*, 50(2):19:1–19:34, 2017. URL: <https://doi.org/10.1145/3056461>, doi:10.1145/3056461.

- [9] Valentina Cacchiani, Manuel Iori, Alberto Locatelli, and Silvano Martello. Knapsack problems - an overview of recent advances. part I: single knapsack problems. *Comput. Oper. Res.*, 143:105692, 2022. URL: <https://doi.org/10.1016/j.cor.2021.105692>, doi:10.1016/j.cor.2021.105692.
- [10] Valentina Cacchiani, Manuel Iori, Alberto Locatelli, and Silvano Martello. Knapsack problems - an overview of recent advances. part II: multiple, multidimensional, and quadratic knapsack problems. *Comput. Oper. Res.*, 143:105693, 2022. URL: <https://doi.org/10.1016/j.cor.2021.105693>, doi:10.1016/j.cor.2021.105693.
- [11] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979. URL: [https://doi.org/10.1016/0022-0000\(79\)90044-8](https://doi.org/10.1016/0022-0000(79)90044-8), doi:10.1016/0022-0000(79)90044-8.
- [12] Lianhua Chi and Xingquan Zhu. Hashing techniques: A survey and taxonomy. *ACM Comput. Surv.*, 50(1), apr 2017. URL: <https://doi.org/10.1145/3047307>, doi:10.1145/3047307.
- [13] Reuven Cohen, Pierre Fraigniaud, David Ilcinkas, Amos Korman, and David Peleg. Label-guided graph exploration by a finite automaton. *ACM Trans. Algorithms*, 4(4):42:1–42:18, 2008. URL: <https://doi.org/10.1145/1383369.1383373>, doi:10.1145/1383369.1383373.
- [14] Marek Cygan, Lukasz Jez, and Jiri Sgall. Online knapsack revisited. *Theory Comput. Syst.*, 58(1):153–190, 2016. URL: <https://doi.org/10.1007/s00224-014-9566-4>, doi:10.1007/s00224-014-9566-4.
- [15] Shantanu Das. Graph explorations with mobile agents. In Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors, *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*, pages 403–422. Springer, 2019. URL: [https://doi.org/10.1007/978-3-030-11072-7\\_16](https://doi.org/10.1007/978-3-030-11072-7_16), doi:10.1007/978-3-030-11072-7\_16.
- [16] Reinhard Diestel. *Graph Theory: 5th edition*. Springer Graduate Texts in Mathematics. Springer-Verlag, © Reinhard Diestel, 2017. URL: <https://books.google.ch/books?id=zIxRDwAAQBAJ>.
- [17] Martin Dietzfelbinger, Kurt Mehlhorn, and Peter Sanders. *Algorithmen und Datenstrukturen - die Grundwerkzeuge*. eXamen.press. Springer, 2014. URL: <https://doi.org/10.1007/978-3-642-05472-3>, doi:10.1007/978-3-642-05472-3.
- [18] Martin Dietzfelbinger, Michael Mitzenmacher, Rasmus Pagh, David P. Woodruff, and Martin Aumüller. Theory and applications of hashing (dagstuhl seminar 17181). *Dagstuhl Reports*, 7(5):1–21, 2017. URL: <https://doi.org/10.4230/DagRep.7.5.1>, doi:10.4230/DagRep.7.5.1.
- [19] Yann Disser, Jan Hackfeld, and Max Klimm. Tight bounds for undirected graph exploration with pebbles and multiple agents. *J. ACM*, 66(6):40:1–40:41, 2019. URL: <https://doi.org/10.1145/3356883>, doi:10.1145/3356883.

- [20] Stefan Dobrev, Rastislav Kráľovic, and Dana Pardubská. Measuring the problem-relevant information in input. *RAIRO Theor. Informatics Appl.*, 43(3):585–613, 2009. URL: <https://doi.org/10.1051/ita/2009012>, doi:10.1051/ita/2009012.
- [21] Stefan Dobrev, Rastislav Kráľovič, and Dana Pardubská. How much information about the future is needed? In Viliam Geffert, Juhani Karhumäki, Alberto Bertoni, Bart Preneel, Pavol Návrat, and Mária Bieliková, editors, *SOFSEM 2008: Theory and Practice of Computer Science, 34th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 19-25, 2008, Proceedings*, volume 4910 of *Lecture Notes in Computer Science*, pages 247–258. Springer, 2008. URL: [https://doi.org/10.1007/978-3-540-77566-9\\_21](https://doi.org/10.1007/978-3-540-77566-9_21), doi:10.1007/978-3-540-77566-9\_21.
- [22] Devdatt P. Dubhashi and Desh Ranjan. Balls and bins: A study in negative dependence. *Random Struct. Algorithms*, 13(2):99–124, 1998.
- [23] Yuval Emek, Pierre Fraigniaud, Amos Korman, and Adi Rosén. Online computation with advice. *Theor. Comput. Sci.*, 412(24):2642–2656, 2011. URL: <https://doi.org/10.1016/j.tcs.2010.08.007>, doi:10.1016/j.tcs.2010.08.007.
- [24] Michael L. Fredman and János Komlós. On the size of separating systems and families of perfect hash functions. *SIAM J. Alg. Disc. Meth.*, 5:61–68, 03 1984.
- [25] Leszek Gasieniec and Tomasz Radzik. Memory efficient anonymous graph exploration. In Hajo Broersma, Thomas Erlebach, Tom Friedetzky, and Daniël Paulusma, editors, *Graph-Theoretic Concepts in Computer Science, 34th International Workshop, WG 2008, Durham, UK, June 30 - July 2, 2008. Revised Papers*, volume 5344 of *Lecture Notes in Computer Science*, pages 14–29, 2008. URL: [https://doi.org/10.1007/978-3-540-92248-3\\_2](https://doi.org/10.1007/978-3-540-92248-3_2), doi:10.1007/978-3-540-92248-3\_2.
- [26] Gaston H. Gonnet. Expected length of the longest probe sequence in hash code searching. *J. ACM*, 28(2):289–304, 1981. URL: <https://doi.org/10.1145/322248.322254>, doi:10.1145/322248.322254.
- [27] Venkatesan Guruswami and Andrii Riazanov. Beating Fredman-Komlós for perfect  $k$ -hashing. *J. Comb. Theory, Ser. A*, 188:105580, 2022. URL: <https://doi.org/10.1016/j.jcta.2021.105580>, doi:10.1016/j.jcta.2021.105580.
- [28] Xin Han, Yasushi Kawase, and Kazuhisa Makino. Online knapsack problem with removal cost. In Joachim Gudmundsson, Julián Mestre, and Taso Viglas, editors, *Computing and Combinatorics - 18th Annual International Conference, COCOON 2012, Sydney, Australia, August 20-22, 2012. Proceedings*, volume 7434 of *Lecture Notes in Computer Science*, pages 61–73. Springer, 2012. URL: [https://doi.org/10.1007/978-3-642-32241-9\\_6](https://doi.org/10.1007/978-3-642-32241-9_6), doi:10.1007/978-3-642-32241-9\_6.
- [29] Xin Han, Yasushi Kawase, and Kazuhisa Makino. Randomized algorithms for online knapsack problems. *Theor. Comput. Sci.*, 562:395–405, 2015. URL: <https://doi.org/10.1016/j.tcs.2014.10.017>, doi:10.1016/j.tcs.2014.10.017.

- [30] Xin Han, Yasushi Kawase, Kazuhisa Makino, and Haruki Yokomaku. Online knapsack problems with a resource buffer. In Pinyan Lu and Guochuan Zhang, editors, *30th International Symposium on Algorithms and Computation, ISAAC 2019, December 8-11, 2019, Shanghai University of Finance and Economics, Shanghai, China*, volume 149 of *LIPIcs*, pages 28:1–28:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. URL: <https://doi.org/10.4230/LIPIcs.ISAAC.2019.28>, doi:10.4230/LIPIcs.ISAAC.2019.28.
- [31] Georges Hansel. Nombre minimal de contacts de fermeture nécessaires pour réaliser une fonction booléenne symétrique de  $n$  variables. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, 258, 1964.
- [32] Juraj Hromkovič, Rastislav Kráľovič, and Richard Kráľovič. Information complexity of online problems. In *Mathematical Foundations of Computer Science 2010, 35th International Symposium, MFCS 2010, Brno, Czech Republic, August 23–27, 2010. Proceedings*, pages 24–36, 2010. URL: [https://doi.org/10.1007/978-3-642-15155-2\\_3](https://doi.org/10.1007/978-3-642-15155-2_3), doi:10.1007/978-3-642-15155-2\_3.
- [33] Kazuo Iwama and Shiro Taketomi. Removable online knapsack problems. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan J. Eidenbenz, and Ricardo Conejo, editors, *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings*, volume 2380 of *Lecture Notes in Computer Science*, pages 293–305. Springer, 2002. URL: [https://doi.org/10.1007/3-540-45465-9\\_26](https://doi.org/10.1007/3-540-45465-9_26), doi:10.1007/3-540-45465-9\_26.
- [34] Kazuo Iwama and Guochuan Zhang. Online knapsack with resource augmentation. *Inf. Process. Lett.*, 110(22):1016–1020, 2010. URL: <https://doi.org/10.1016/j.ipl.2010.08.013>, doi:10.1016/j.ipl.2010.08.013.
- [35] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel Dominic Sleator. Competitive snoopy caching. In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 244–254. IEEE Computer Society, 1986. URL: <https://doi.org/10.1109/SFCS.1986.14>, doi:10.1109/SFCS.1986.14.
- [36] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. URL: [https://doi.org/10.1007/978-1-4684-2001-2\\_9](https://doi.org/10.1007/978-1-4684-2001-2_9), doi:10.1007/978-1-4684-2001-2\_9.
- [37] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987. URL: <https://doi.org/10.1147/rd.312.0249>, doi:10.1147/rd.312.0249.
- [38] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer, 2004. URL: <https://doi.org/10.1007/978-3-540-24777-7>, doi:10.1007/978-3-540-24777-7.

- [39] Dennis Komm. *An Introduction to Online Computation – Determinism, Randomization, Advice*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016. URL: <https://doi.org/10.1007/978-3-319-42749-2>, doi:10.1007/978-3-319-42749-2.
- [40] János Körner. Fredman–Kömlös bounds and information theory. *SIAM Journal on Algebraic Discrete Methods*, 7(4):560–570, 1986. URL: <https://doi.org/10.1137/0607062>, doi:10.1137/0607062.
- [41] Michael Luby and Avi Wigderson. Pairwise independence and derandomization. *Foundations and Trends in Theoretical Computer Science*, 1(4), 2005. URL: <https://doi.org/10.1561/0400000009>, doi:10.1561/0400000009.
- [42] Alberto Marchetti-Spaccamela and Carlo Vercellis. Stochastic on-line knapsack problems. *Math. Program.*, 68:73–104, 1995. URL: <https://doi.org/10.1007/BF01585758>, doi:10.1007/BF01585758.
- [43] Silvano Martello and Paolo Toth. *Knapsack problems : algorithms and computer implementations*. Wiley-Interscience series in discrete mathematics and optimization. J. Wiley & Sons, Chichester, 1990 - 1990.
- [44] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1984. URL: <https://doi.org/10.1007/978-3-642-69672-5>, doi:10.1007/978-3-642-69672-5.
- [45] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008. URL: <https://doi.org/10.1007/978-3-540-77978-0>, doi:10.1007/978-3-540-77978-0.
- [46] Artur Menc, Dominik Pajak, and Przemyslaw Uznanski. Time and space optimality of rotor-router graph exploration. *Inf. Process. Lett.*, 127:17–20, 2017. URL: <https://doi.org/10.1016/j.ipl.2017.06.010>, doi:10.1016/j.ipl.2017.06.010.
- [47] Jesper W. Mikkelsen. Randomization can be as helpful as a glimpse of the future in online computation. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPICs*, pages 39:1–39:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. URL: <https://doi.org/10.4230/LIPICs.ICALP.2016.39>, doi:10.4230/LIPICs.ICALP.2016.39.
- [48] Michael Mitzenmacher and Eli Upfal. *Probability and Computing*. Cambridge University Press, 2017.
- [49] Robert H. Morris. Scatter storage techniques. *Commun. ACM*, 11(1):38–44, jan 1968. URL: <https://doi.org/10.1145/362851.362882>, doi:10.1145/362851.362882.

- [50] Robert H. Morris. Scatter storage techniques (reprint). *Commun. ACM*, 26(1):39–42, 1983. URL: <https://doi.org/10.1145/357980.357996>, doi:10.1145/357980.357996.
- [51] Moni Naor, Leonard J. Schulman, and Aravind Srinivasan. Splitters and near-optimal derandomization. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23–25 October 1995*, pages 182–191, 1995. URL: <https://doi.org/10.1109/SFCS.1995.492475>, doi:10.1109/SFCS.1995.492475.
- [52] Martin Raab and Angelika Steger. “Balls into bins” – A simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science, Second International Workshop, RANDOM’98, Barcelona, Spain, October 8–10, 1998, Proceedings*, pages 159–170, 1998. doi:10.1007/3-540-49543-6\_13.
- [53] Herbert E. Robbins. A remark on Stirling’s formula. *Am. Math. Mon.*, 62:26–29, 1955. doi:10.2307/2308012.
- [54] Jeanette P. Schmidt and Alan Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM J. Comput.*, 19(5):775–786, 1990. URL: <https://doi.org/10.1137/0219054>, doi:10.1137/0219054.
- [55] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985. URL: <https://doi.org/10.1145/2786.2793>, doi:10.1145/2786.2793.