Diss. ETH No. 22591

Information Content of Online Problems
Advice versus Determinism and Randomization

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

JASMIN SMULA

Dipl.-Inf., TU Dortmund
born on September 14, 1984
citizen of Germany

accepted on the recommendation of

Prof. Dr. Juraj Hromkovič, examiner
Prof. Dr. Peter Widmayer, co-examiner
Prof. Dr. Rastislav Královič, co-examiner
Dr. Dennis Komm, co-examiner

2015

Abstract

In online computation, an algorithm has to solve some optimization problem while receiving the input instance gradually, without any knowledge about the future input. Such an online algorithm has to compute parts of the output for parts of the input, based on what it knows about the input so far and without being able to revoke its decisions later. Almost inevitably, the algorithm makes a bad choice at some point that leads to a solution that is suboptimal with respect to the whole input instance. Compared to an offline algorithm that is given the entire input instance at once, the online algorithm thus has a substantial handicap. Developing online algorithms that nonetheless compute solutions of some adequate quality is a large and rich field of research within computer science.

The quality of online algorithms is traditionally measured in terms of the competitive ratio, which compares the solutions computed by the online algorithm to those of an optimal offline algorithm. Depending on the online problem at hand, it can differ considerably how much an online algorithm's competitiveness suffers from the lack of knowledge about the input. For some problems such as the ski rental problem, there are online algorithms that can guarantee to compute 2-competitive solutions on any input; on the other hand, there are online problems for which no competitive algorithms exist at all (meaning there are no algorithms achieving any constant competitive ratio).

In some sense, this way of measuring the hardness of online problems is rather rough as a complete lack of knowledge is unrealistic for many real-world applications. With its additional knowledge about the input, the optimal offline algorithm has a huge advantage compared to any online algorithm. Therefore, another way of measuring the complexity of a given online problem has been introduced: the *advice complexity*. Advice complexity theory deals with the question of how much information an online algorithm lacks to be able to compute solutions with some satisfactory competitive ratio. More precisely, assuming some all-knowing oracle of unlimited computing power that knows the entire input, we are interested in the number of bits that are necessary and/or sufficient for any online algorithm to compute solutions of some specified competitive ratio; we call these bits *advice bits* and the corresponding algorithm an *online algorithm with advice*. The minimum number of advice bits necessary and sufficient to be optimal is called the *information content* of the online problem at hand.

The advice complexity and the information content have already been analyzed for many different online problems and are a topic of ongoing research. In this thesis, we further investigate the following online problems with respect to their advice complexity; the k-server problem, the disjoint path allocation problem, online graph searching and graph exploration, and the string guessing problem.

For the k-server problem, we prove a lower bound on the advice complexity of any online algorithm with advice with a competitive ratio of up to $3/2$ that already holds for paths of length 2. Our result improves upon the yet best known trade-off, which only applies for competitive ratios of less than $5/4$. For finite paths of arbitrary length, we give another lower bound yielding better results for near-optimal competitive ratios.

For the disjoint path allocation problem, we show a general lower bound on the advice complexity for a wide range of competitive ratios, from constant up to logarithmic in the path length. From this general lower bound, several lower bounds for concrete ranges of the competitive ratio can be derived. One result we present in this thesis is a lower bound of a linear number of advice bits necessary to achieve any constant strict competitive ratio. This bound implies a surprising threshold behavior of the advice complexity of the disjoint path allocation problem on paths. Although a double logarithmic number of advice bits is sufficient to obtain a competitive ratio that is logarithmic in the path length, any sublinear number of additional advice bits is not enough to further decrease this competitive ratio by another constant factor.

For the graph searching problem, we give asymptotically matching lower and upper bounds of $\Theta(n/c)$ advice bits to achieve c-competitiveness, for any (not necessarily constant) c. In the context of graph exploration, we present a lower-bound result that makes use of a new reduction technique developed to prove trade-offs between the number of advice bits necessary and the competitive ratio of a certain class of online problems, which includes the graph exploration and the graph searching problem.

We also investigate the advice complexity of the string guessing problem in a new probabilistic model featuring a more powerful adversary. In this scenario, we consider two different ways of modeling the oracle. As the setting contains a probabilistic element, the quality of the solution computed by an algorithm is a random variable, and an algorithm can only try to optimize the expected value of this random variable in its favor. For both kinds of oracles considered, we give asymptotically matching lower and upper bounds for the number of advice bits necessary and sufficient to obtain solutions for which this expected value is optimal.

Zusammenfassung

Algorithmen für Online-Optimierungsprobleme erhalten die jeweilige Eingabe-Instanz stückweise, ohne dabei zukünftige Teile der Eingabe zu kennen. Dies bedeutet, dass ein solcher Online-Algorithmus Teile der Ausgabe berechnen muss, die nur auf dem bereits bekannten Teil der Eingabe basieren. Ferner darf er seine Entscheidungen hiernach nicht mehr revidieren. Es ist fast unvermeidbar, dass er dabei früher oder später eine schlechte Entscheidung trifft, die zu einer suboptimalen Lösung bezüglich der gesamten Instanz führt. Wir sehen, dass Online-Algorithmen einen offensichtlichen Nachteil gegenüber Offline-Algorithmen haben, die die gesamte Eingabe von Beginn an kennen. Die Entwicklung von Online-Algorithmen, die dennoch eine gute Lösungsqualität erzielen, ist ein grosses und interessantes Forschungsgebiet innerhalb der Informatik.

Die Qualität von Online-Algorithmen wird klassischerweise durch den kompetitiven Faktor beschrieben, der ihre Lösungen mit denen von optimalen Offline-Algorithmen vergleicht. Je nach betrachtetem Problem ist der Qualitätsverlust aufgrund der teils unbekannten Eingabe sehr unterschiedlich ausgeprägt. Es existieren Online-Probleme, beispielsweise das Ski-Rental-Problem, für die Online-Algorithmen bekannt sind, die eine 2-kompetitive Lösung auf jeder Eingabe garantieren können. Andererseits gibt es Online-Probleme, für die keine kompetitiven Algorithmen existieren (was bedeutet, dass es keine Algorithmen mit konstantem kompetitiven Faktor für sie gibt).

Der kompetitive Faktor ist allerdings in einem gewissen Sinne recht grob, da es in vielen praktischen Situationen unrealistisch ist anzunehmen, dass gar nichts über die Eingabe bekannt ist. Durch sein zusätzliches Wissen über die Eingabe hat ein optimaler Offline-Algorithmus einen enormen Vorteil gegenüber Online-Algorithmen. Aus diesem Grund wurde ein weiteres Mass für die Komplexität von Online-Problemen eingeführt: *Advice-Komplexität*. Das Modell der Advice-Komplexität untersucht die Frage, was für Informationen ein Online-Algorithmus benötigt, um Lösungen mit einem zufriedenstellenden kompetitiven Faktor zu erreichen. Hierzu gehen wir von einem allwissenden Orakel aus, das über unbeschränkte Ressourcen verfügt und welches die gesamte Eingabe kennt. Dieses Orakel kann dem Online-Algorithmus binäre Informationen über die Eingabe bereitstellen; diese werden als *Advice-Bits* bezeichnet und ein entsprechender Algorithmus als *Online-Algorithmus mit Advice*. Wir interessieren uns für die Anzahl dieser Advice-Bits, die nötig bzw. ausreichend sind, damit Lösungen mit einem gegebenen kompetitiven Faktor berechnet werden können. Die minimale Anzahl, die es ermöglicht eine optimale Lösung zu berechnen, wird als *Informationsgehalt* des gegebenen Online-Problems bezeichnet.

Die Advice-Komplexität und der damit verbundene Informationsgehalt wurden bereits für viele verschiedene Online-Probleme untersucht und sind Gegenstand laufender Forschung. In dieser Arbeit untersuchen wir die folgenden Probleme hinsichtlich ihrer Advice-Komplexität: Das k-Server-Problem, das Disjoint-Path-Allocation-Problem, Online-Graph-Searching und Online-Graph-Exploration und das String-Guessing-Problem.

Für das k-Server-Problem beweisen wir eine untere Schranke für die Advice-Komplexität für einen beliebigen Online-Algorithmus mit Advice, der einen kompetitiven Faktor von bis zu $3/2$ erzielt. Diese Schranke gilt bereits für Pfade der Länge 2. Unser Ergebnis verbessert den bisher besten bekannten Trade-Off, der nur eine Aussage über kompetitive Faktoren von bis zu $5/4$ macht. Für endliche Pfade beliebiger Länge zeigen wir eine weitere untere Schranke, die verbesserte Resultate für fast-optimale kompetitive Faktoren liefert.

Des Weiteren zeigen wir eine allgemeine untere Schranke für das Disjoint-Path-Allocation-Problem, die für einen grossen Bereich von kompetitiven Faktoren gilt, von konstanten Werten bis hin zu Werten, die logarithmisch in der Pfadlänge sind. Dieses allgemeine Ergebnis impliziert diverse untere Schranken für konkrete Bereiche des erreichbaren kompetitiven Faktors. Wir erhalten so unter anderem eine untere Schranke, die aussagt, dass eine lineare Anzahl an Advice-Bits nötig ist, um einen beliebigen konstanten strikten kompetitiven Faktor garantieren zu können. Dies zeigt wiederum ein überraschendes Schwellwertverhalten der Advice-Komplexität des Disjoint-Path-Allocation-Problems auf Pfaden. Obwohl bereits doppelt logarithmisch viele Advice-Bits ausreichen, um einen kompetitiven Faktor zu erhalten, der logarithmisch in der Pfadlänge ist, reicht keine sublineare Anzahl an Advice-Bits, um diesen Faktor um eine weitere Konstante zu verbessern.

Für das Graph-Searching-Problem zeigen wir asymptotisch scharfe untere und obere Schranken von $\Theta(n/c)$ Advice-Bits, um c-Kompetitivität zu erreichen, für jedes beliebige (nicht notwendigerweise konstante) c. Für Graph-Exploration beweisen wir eine untere Schranke mithilfe einer neuen Reduktionstechnik, die es erlaubt, für eine gewisse Klasse von Online-Problemen Trade-Offs zwischen der Anzahl der Advice-Bits und dem kompetitiven Faktor zu beweisen, zu der sowohl Graph-Exploration als auch Graph-Searching gehört.

Wir untersuchen darüber hinaus das String-Guessing-Problem in einem neuen probabilistischen Modell, das einen stärkeren Gegenspieler besitzt. In diesem Szenario betrachten wir zwei verschiedene Arten, das Orakel zu modellieren. Da das gewählte Setting eine probabilistische Komponente enthält, ist die Ausgabequalität der vom Online-Algorithmus berechneten Lösung eine Zufallsvariable, und der Algorithmus kann lediglich versuchen, deren Erwartungswert zu seinen Gunsten zu optimieren. Für beide untersuchten Orakel-Modelle beweisen wir asymptotisch scharfe untere und obere Schranken bezüglich der Anzahl an Advice-Bits, die nötig und ausreichend sind, um diesen Erwartungswert zu optimieren.

Acknowledgements

First of all, I would like to express my deepest gratitude to Juraj Hromkovič, who gave me the opportunity to write my PhD thesis in his group. He was the best supervisor I could ever have imagined, giving me free rein to do research on my own while at the same time letting me know that he was always available whenever his help was required. Furthermore, I would like to thank Peter Widmayer, Rastislav Královič, and Dennis Komm for agreeing to review this dissertation. I am also thankful to Hans-Joachim "Hajo" Böckenhauer, who always gave extremely valuable advice, in particular for this thesis, which he proof-read very carefully and for which he provided very constructive feedback.

It was a real pleasure to be part of the group *Information Technology and Education*. The working atmosphere in this group was always both productive and friendly. During my time here, I enjoyed not only working with the members of this group, but also spending time with them during non-working hours. Several results that are associated with this thesis were developed in joint work with members of the group, former members of the group, or people that are closely related to the group. Some of these results have already been published, and I would like to thank my co-authors Kfir Barhum, Hajo, Michal Forišek, Heidi Gebauer, Juraj, Dennis, Rastislav, Richard Královič, Sacha Krug, Andreas Sprock, and Björn Steffen for the good collaboration. Furthermore, the very first attempts concerning a few topics were discussed on one of the mountain workshops regularly arranged by Juraj; I would like to thank Hajo, Jérôme Dohrau, and Antonio Fernández Anta for some helpful input. Moreover, I want to thank Dennis again for putting up with me as his office mate for the last few years. Despite his busy schedule, he always found the time to counsel, discuss, and help wherever possible, but most of all, I appreciated his earthy sense of humor and that he was always up for going to have a beer with me after a long day of work.

Additionally, I would like to thank my former working colleagues of the *Distributed Computing Group*. I would never have thought that a bunch of working colleagues could become such a tight-knit clique, and it was a great experience to be a part of it. I am very grateful especially for sharing an office with Barbara Keller during my time there. Barbara was one of the reasons that I was very sad when I left the group, leaving this awesome time behind. I have also had many brilliant moments with the rest of the group, whom I would also like to thank; in particular Philipp Brandes, Christian Decker, Klaus-Tycho Förster, Tobias Langner, Jochen Seidel, Jara Uitto, and Samuel Welten.

During my studies at the University of Dortmund, the first lectures about theoretical computer science that I attended were held by Ingo Wegener and Thomas Hofmeister. Both of them gave their lectures with such an enthusiasm that it spread to the audience, and especially to me. To them I owe my interest in theoretical computer science in the first place, for which I am extremely grateful.

Furthermore, I want to thank my friend Eva Gotthardt, who has accompanied me for my entire stay in Zurich. I have been through a lot during my PhD studies here, and Eva was always there for me, no matter whether I needed moral support or if I was looking for company to celebrate. Finally, I would like to express my deepest gratitude to my family, in particular my parents Marianne and Hans-Jürgen and my sister Mareike Smula, for their unconditional and tireless support. I know I can always count on you.

Contents

1 Introduction

In everyday life, we are often confronted with *online problems*. Informally, this means that we have to make decisions—often banal ones, but sometimes also ones with significant impact—without knowing the future. Imagine, for example, you are in the car, on the way into your long-awaited vacations; taking the fastest route, you know it would be possible to reach your vacation destination within 8 hours—if it were not for traffic. An hour ago, you decided to leave your planned route due to a congested road ahead, just to find yourself in a bumper-to-bumper traffic jam right after that. In the end, you arrive at your destination, totally exhausted, after a 12-hour drive. It seems that every time you made the decision to alter your planned route or to stick to your current one, you made a bad choice. Being confronted with choices, not knowing what consequences each possible decision will eventually have, is a frustrating daily routine. There are various other examples for situations in real life in which we are forced to make decisions without knowing the future; ranging from rather insignificant ones, such as choosing appropriate clothing for a hike without knowing how the weather is going to develop, to choices with a great impact on our financial situation, such as deciding in which stock to invest.

Online computation is the field of computer science that deals with the formalization of such online problems and the development and analysis of algorithms to solve them. An algorithm that is supposed to solve some online problem receives its input piecewise and has to choose how to proceed with each piece of the input immediately, without any information about the future input and without the possibility to revise its decisions. Such an algorithm is called an *online algorithm*. The quality of an online algorithm A is traditionally measured in terms of its *competitive ratio*, which relates the quality of the solutions computed by A to the quality of the solutions computed by an optimal *offline algorithm* that knows the whole input in advance. Since we desire algorithms that also perform reasonably well when given a worst-case input, we usually assume that the input given to the algorithm is chosen by a malicious *adversary* whose goal is to maximize the algorithm's competitive ratio, whereas the algorithm's goal is to minimize it.

For many problems in real life, we sometimes wish we could get information from some source of unlimited knowledge. We would like to know, for example, which route will be least congested, or what stock is going to increase soon. And, as a matter of fact, for all examples mentioned above, great efforts have been made to be able to make predictions about the future; there are the weather forecast, navigation systems with built-in algorithms for bypassing traffic jams, and attempts to predict fluctuation in the stock market. In online computation, this concept also exists, in the form of an omniscient *oracle* with unlimited computing power and full knowledge about the input instance at hand. This oracle can provide the online algorithm with *advice bits* to reveal crucial information about the input instance and thus improve the quality of the solution computed by the algorithm. The field of *advice complexity theory* deals with the question of how many advice bits are necessary and sufficient to compute solutions of a certain quality.

An especially interesting and challenging task is to prove lower bounds on the number of advice bits. One tool has proven to be extremely helpful in this regard, namely the *string guessing problem*. This problem is a very generic online problem, maybe even the most generic online problem, and it can be discovered in many other online problems. The input being an unknown string of length n, an online algorithm for the string guessing problem is asked to guess this input string, letter by letter. The task of the algorithm is to guess as many letters correctly as possible. Although this problem is extremely elementary, even very elaborate online problems can be interpreted in one way or the other as guessing letters of an unknown input string. This circumstance can be exploited to specify a *reduction* from the string guessing problem to a given online problem. The concept of reductions is often applied in computer science to prove that some given problem P is not easier to solve than another problem Q, transfering already known hardness results for Q to hardness results for P. This thesis has a strong focus on the string guessing problem, and on constructing reductions to obtain lower bounds on the advice complexity of other online problems.

1.1 This Dissertation

The remainder of this chapter serves to introduce the mathematical foundations that we will need throughout this thesis in a formal way. Apart from fixing some mathematical concepts and notation in Section 1.2, we give formal descriptions of the concepts of online computation (Section 1.3), online compuation with advice (Section 1.4), and the string guessing problem (Section 1.5).

Each of the following four chapters deals, in one way or the other, with the string guessing problem and how it can be used to infer results concerning the advice complexity of other online problems. Chapter 2 covers the k-*server problem*, Chapter 3 the *disjoint path allocation problem*, and Chapter 4 addresses two related problems, the *graph exploration* and the *graph searching problem*. In Chapter 5, we introduce a more powerful adversary that is able to choose random bits, and analyze the string guessing problem thoroughly in this new model. We show that, also in this model, the string guessing problem can be used to transfer results for this problem to other online problems.

Several problems analyzed in this dissertation have been proposed by or developed in collaboration with my colleagues. In particular, Juraj Hromkovič pointed me to all problems that are investigated in this thesis. First ideas for the lower bounds presented in Chapter 2 were developed during a workshop in Montserrat; the technical details were developed autonomously afterwards. Most of the results in Chapter 3 have been developed together with Heidi Gebauer, Dennis Komm, Rastislav Královič, and Richard Královič, and those in Chapter 4 in collaboration with Dennis Komm, Rastislav Královič, and Richard Královič. The model of the probabilistic adversary from Chapter 5 has been proposed by Juraj Hromkovič; all results and technical details therein were found and elaborated in independent work by the current author.

1.2 Mathematical Foundations

In this section, we present a short overview of the most important mathematical concepts and notation being used throughout this thesis. However, we broach every subject only briefly, mainly to fix our notation. For a more general introduction to online algorithms, see the textbook of Borodin and El-Yaniv [BEY98]; the concept of advice complexity is discussed in detail by Komm [Kom12].

1.2.1 Sets

A *set* is a collection of objects. These objects are called *elements* of the set and are usually required to be pairwise distinct. We write, for example, $\{0, 1\}$ for a set containing the two elements 0 and 1. The *cardinality* or *size* of a set S is the number of elements contained in S and denoted by $|S|$. The *empty set* is denoted by $\emptyset$. For each set S, we denote by $\mathcal{P}(S)$ the *power set* of S, defined as

$$\mathcal{P}(S) = \{R \mid R \subseteq S\}.$$

Whenever the order in which the elements of a set are listed matters, we talk about *ordered sets*. For ordered sets, we drop the requirement of all elements being pairwise distinct and allow multiple occurrences of the same element. To distinguish ordered sets from unordered ones, we use parentheses instead of braces to denote the former; for example, we write $(0, 1)$ instead of $\{0, 1\}$. An ordered set is also called a *sequence* or a *tuple*. For sequences, often the term *length* is used instead of size or cardinality. Tuples of size n are also called n-*tuples*; furthermore, tuples of size 2 are called *pairs*.

Throughout this thesis, we use the standard notation for the ordered sets of integers and real, rational, and natural numbers. For the set of *real numbers*, we use the symbol $\mathbb{R}$. We denote the set of *rational numbers* by $\mathbb{Q}$ and the set of *integers* by $\mathbb{Z}$. The set of *natural numbers* is denoted by $\mathbb{N}$. We often need to constrain our considerations to numbers that do not exceed or fall below a certain threshold. In such cases, we sometimes add a superscript to the set symbol to indicate this threshold. For example, in this notation, the set of negative real numbers can be denoted by $\mathbb{R}^{<0}$, and we have $\mathbb{N} = \mathbb{Z}^{\geq 0}$. Concerning the latter statement, though, the literature is not completely consistent. Although in this thesis we usually assume that 0 is included in the set of natural numbers, in some literature it is not (hence, $\mathbb{N} = \mathbb{Z}^{\geq 1}$). Therefore, whenever we are talking about $\mathbb{N}$ and want to make completely clear whether 0 is to be included in our considerations or not, we also make use of this superscript notation and write either $\mathbb{N}^{\geq 0}$ or $\mathbb{N}^{\geq 1}$.

1.2.2 Alphabets and Strings

An *alphabet* is a nonempty finite set of *letters*, and is usually denoted by Σ throughout this thesis. Often we consider the *binary alphabet* $\Sigma_2 = \{0, 1\}$. The letters 0 and 1 in Σ_2 are called *bits*. A *string* over an alphabet Σ is a sequence $r = (r_1, \ldots, r_n)$ of letters from Σ, for some natural number $n \in \mathbb{N}^{\geq 0}$, and if the letter r_i is contained in Σ_2, for each i with $1 \leq i \leq n$, we call r a *binary string* or *bit string*. If $n = 0$, we say that r is the *empty string*, which we denote by ε. Instead of writing $r = (r_1, \ldots, r_n)$, we also use $r = r_1 \ldots r_n$ as a shorthand notation. A string $r' = r_1 \ldots r_m$ with $m \leq n$ is called a *prefix* of r. Comparably, a string $r' = r_m \ldots r_n$ with $m \geq 1$ is called a *suffix* of r.

1.2.3 Functions and Constants

For any subset S' of an ordered set S, we define the *minimum of* S', denoted by $\min(S')$, to be an element x of S' such that $x \leq y$ for all elements $y \in S'$; analogously, we define the *maximum of* S', denoted by $\max(S')$, to be an element x of S' such that $x \geq y$ for all elements $y \in S'$.

For any real number $x \in \mathbb{R}$, we use $\lfloor x \rfloor$ to denote the largest integer y with $y \leq x$ and call it the *floor* of x. Accordingly, $\lceil x \rceil$ denotes the smallest integer y such that $y \geq x$ and is called the *ceiling* of x.

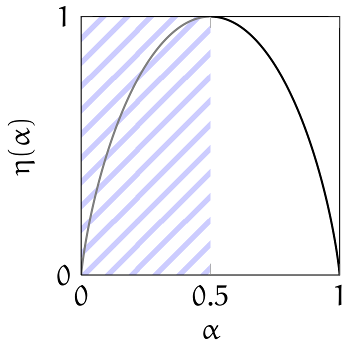


Figure 1.1. The binary entropy function $\eta(\alpha) = -\alpha \log(\alpha) - (1-\alpha)\log(1-\alpha)$. For our purposes, α is only considered in the range $1/2 \leq \alpha < 1$. Therefore, the other part of the graph is hatched.

For any two real numbers $x \in \mathbb{R}$ and $y \in \mathbb{R}^{>0}$ with $y \neq 1$, we denote the *logarithm to base* y *of* x by $\log_y(x)$. In this thesis, logarithms are usually to base 2 if not stated otherwise. Hence, usually we take a pass on mentioning the base explicitly and just write $\log(x)$ instead of $\log_2(x)$. If it does not introduce any ambiguity, we also often omit the parentheses and write $\log x$ instead of $\log(x)$.

Furthermore, we will often encounter the so-called *entropy*, which is, originally, a measure of the information content of a given string and plays a great role in the field of coding theory (see, for example, Roth [Rot06]). For any real number $p \in \mathbb{R}$ with $0 \leq p \leq 1$ and every natural number $q \in \mathbb{N}^{\geq 2}$, the q-*ary entropy function of* p is defined as

$$\eta_q(p) = p\log_q(q-1) - p\log_q(p) - (1-p)\log_q(1-p),$$

where $0\log_q(0)$ is assumed to be 0. For the binary entropy function, which is the version we are usually considering, this yields

$$\eta_2(p) = p\log_2(p) - (1-p)\log_2(1-p).$$

A plot of the binary entropy function is shown in Figure 1.1. As for logarithms, we allow us to drop the subscript in the binary case and often write $\eta(p)$ instead of $\eta_2(p)$.

We follow the convention to give complexity measures in terms of orders of magnitude. To this end, we use the *Landau symbols* to group functions into classes according to their asymptotical growth. For any two functions $f\colon \mathbb{N}^{\geq 0} \to \mathbb{R}^{\geq 0}$ and $g\colon \mathbb{N}^{\geq 0} \to \mathbb{R}^{\geq 0}$, we say that f does not grow asymptotically faster than g, denoted by $f(n) \in O(g(n))$, if

$$\exists\, n_0, c > 0, \text{ such that } \forall\, n \geq n_0 : f(n) \leq c \cdot g(n),$$

and we say that g grows asymptotically faster than f, denoted by $f(n) \in o(g(n))$, if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Moreover, we use the notations

$$\begin{aligned}
f(n) \in \Omega(g(n)) &\iff g(n) \in O(f(n)),\\
f(n) \in \omega(g(n)) &\iff g(n) \in o(f(n)), \text{ and}\\
f(n) \in \Theta(g(n)) &\iff g(n) \in O(f(n)) \cap \Omega(f(n)).
\end{aligned}$$

In some contexts, we will come across *Euler's number*, a mathematical constant that we denote by e and which can be approximated by e ≈ 2.718.

1.2.4 Combinatorics

The *factorial* of n, i. e., the product of all positive natural numbers from 1 to n, is denoted by $n!$, for any natural number $n \in \mathbb{N}^{\geq 0}$, where 0! is assumed to be 1. For natural numbers $n, k \in \mathbb{N}^{\geq 0}$, the *binomial coefficient* $\binom{n}{k}$ indicates the number of possibilities to choose k elements out of a set containing n elements; it can be calculated as

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}.$$

1.2.5 Probability Theory

At some point in this thesis, we will add a certain random element to the game between the online algorithm, the adversary, and the oracle. More precisely, we will allow the adversary to "toss a coin" (if necessary several times) and choose the instance given to the algorithm as its input depending on the outcome of these coin tosses. Such random elements with an uncertain outcome are called *experiments*, and each possible outcome is called an *elementary event*. The set S of all elementary events of an experiment is the *sample space*, and an *event* is a subset of the sample space and thus an element of the power set $\mathcal{P}(S)$ of S. In this thesis, we only encounter discrete probabilistic models, in which the sample space is a finite set. To assign a probability to each event, we use a function $\Pr\colon \mathcal{P}(S) \to [0, 1]$. This function is called a *probability distribution over* S and has to fulfill the following constraints.

(a) $\Pr(\{s\}) \geq 0$ for every elementary event $\{s\} \subseteq S$,

(b) $\Pr(S) = 1$, and

(c) $\Pr(A \cup B) = \Pr(A) + \Pr(B)$ for all events $A, B \subseteq S$ with $A \cap B = \emptyset$.

The pair $(S, \Pr)$ forms a so-called *probability space*. If the function Pr is such that each elementary event $\{s\} \subseteq S$ occurs with the same probability, i. e., if

$$\Pr(\{s\}) = \frac{1}{|S|} \quad \text{for all } \{s\} \subseteq S,$$

then Pr is called the *uniform distribution*.

A *random variable* in the probability space $(S, \Pr)$ is a function $X \colon S \to \mathbb{R}$ assigning a real number to every elementary event from the sample space. The probability that the random variable X attains a certain exact value y is given by the *probability mass function* $f_X \colon \mathbb{R} \to [0, 1]$, defined by

$$f_X(y) = \Pr(X = y) = \Pr\big(\{\{s\} \subseteq S \mid X(s) = y\}\big).$$

The probability mass function characterizes the *probability distribution of* X, which is formalized by a function $d_X \colon \mathbb{R} \to [0, 1]$, defined as

$$d_X(y) = \Pr(X \leq y) = \sum_{\substack{z \leq y \\ z \in D_X}} \Pr(X = z),$$

where $D_X := \{y \in \mathbb{R} \mid \exists\, s \in S \text{ such that } X(s) = y\}$ is the co-domain of X. For a discrete probability space $(S, \Pr)$ and a random variable X in $(S, \Pr)$, the *expected value of* X is defined as

$$\mathbb{E}[X] = \sum_{y \in D_X} y \cdot \Pr(X = y).$$

A detailed introduction to randomized computation and probability theory is given by, e. g., Hromkovič [Hro05].

1.2.6 Graphs

In every one of the subsequent chapters, we will deal with certain classes of graphs. A *graph* is a pair $G = (V, E)$, where $V = \{v_0, \ldots, v_{n-1}\}$ is a set of *vertices*, some of which are connected by *edges*. The set of edges is given by $E \subseteq \{(v_i, v_j) \mid 0 \leq i, j \leq n - 1\}$. Throughout this thesis, we constrain ourselves to graphs that do not contain any loops, i. e., edges of the form (v_i, v_i).

Graphs can either be weighted or unweighted. In an *edge-weighted* or just *weighted graph*, each edge is assigned a *cost* or *weight* according to a *weight function* $\omega \colon E \to \mathbb{R}$. In an *unweighted graph*, such a weight function does not exist, and we usually assume every edge to have a weight of 1.

Both weighted and unweighted graphs can either be directed or undirected. In a *directed graph*, each edge (v_i, v_j) has an orientation, with v_i being the *startpoint* and v_j being the *endpoint* of (v_i, v_j). To v_i, the edge (v_i, v_j) is an *outgoing edge* and to v_j, it is an *incoming edge*. The *outdegree* of v_i is the number of outgoing edges of v_i, and the *indegree* of v_i is the number of incoming edges of v_i. A sequence of pairwise distinct vertices $U := (u_0, u_1, \dots, u_\ell)$ with $u_i \in V$, for $0 \le i \le \ell$, is called a *path from* u_0 *to* u_ℓ if, for every pair (u_i, u_{i+1}) with $0 \le i \le n-1$, there is an edge $(u_i, u_{i+1}) \in E$. The *length of the path* is the sum of the edge weights of all these edges (u_i, u_{i+1}). In the unweighted case, this coincides with the number of edges on the path U, and hence, the length of this path is ℓ. We say that U is a *shortest path from* u_0 *to* u_ℓ if, among all paths from u_0 to u_ℓ, the path U has minimal length. If there exists a path from u_0 to u_ℓ, we also say that u_ℓ is *reachable* from u_0.

In an *undirected graph*, all edges are undirected, meaning that the edge (v_i, v_j) is identical to the edge (v_j, v_i). Thus, edges are not pairs but unordered sets of size 2, and any undirected edge (v_i, v_j) is usually written as $\{v_i, v_j\}$. We say that both v_i and v_j are *endpoints* of the edge $\{v_i, v_j\}$. For each edge $\{v_i, v_j\} \in E$, the vertex v_j is said to be a *neighbor of* v_i or *adjacent to* v_i. For each vertex v_i, all edges containing v_i are called *incident to* v_i. The *degree* of v_i is defined as the number of its neighbors. In an undirected graph $G = (V, E)$, a *path between* u_0 *and* u_ℓ is a sequence of pairwise distinct vertices $(u_0, u_1, \dots, u_\ell)$ such that, for each i with $0 \le i \le \ell - 1$, the two vertices u_i and u_{i+1} are adjacent to one another. As in the directed case, the *length of a path* is the sum of the weights and thus the number of edges in the unweighted case. A sequence $(u_0, u_1, \dots, u_\ell, u_0)$ is called a *simple cycle* in G if $(u_0, u_1, \dots, u_\ell)$ is a path with $\ell \ge 2$ and if the edge $\{u_0, u_\ell\}$ exists in E. If there is a path from v_i to v_j, for any pair of vertices $(v_i, v_j) \in V \times V$, the graph is called *connected*.

For any directed or undirected, weighted or unweighted graph $G = (V, E)$, the graph $G' = (V', E')$ is called a *subgraph of* G if $V' \subseteq V$ and $E' \subseteq E$. The subgraph G' is called *induced by* V' if E' contains all edges that are also contained in E, constrained to the vertex set V'. Hence, the *subgraph of* G *induced by* $V' \subseteq V$ is the graph $G' = (V', E')$ with $E' = \{(v_i, v_j) \mid v_i, v_j \in V' \wedge (v_i, v_j) \in E\}$.

Throughout this thesis, we will sometimes consider particular classes of graphs, namely paths, cycles, and trees. In the following, we will give brief descriptions for these three classes. Since we consider each of these classes in its respective undirected unweighted version, we constrain our descriptions to these restricted versions, without mentioning this explicitly from now on.

A *path graph*, for short also named *path* (not to confuse with a path within a graph as described above), is a graph $G = (V, E)$ with $V = \{v_0, \dots, v_\ell\}$ and $E = \{\{v_i, v_{i+1}\} \mid 0 \le i \le \ell - 1\}$. The *length of the path* is the number of edges contained in E, which is ℓ. A *cycle graph* or *cycle* is a path graph as described above with an additional edge between v_ℓ and v_0. Hence, $G = (V, E)$ is a cycle graph if $V = \{v_0, \dots, v_\ell\}$ and $E = \{\{v_i, v_{i+1}\} \mid 0 \le i \le \ell - 1\} \cup \{\{v_\ell, v_0\}\}$. The length of the cycle is the number of edges in E, which is $\ell + 1$. A graph $G = (V, E)$ is called a *tree* if G is connected and does not contain any simple cycles. In every tree, vertices of degree at most 1 are called *leaves*; all other vertices, i. e., those with a degree of at least 2, are called *inner vertices*. Sometimes, we choose one designated vertex of a tree $G = (V, E)$ to be the *root* of G. In this case, G is said to be *rooted*. For each vertex $v_i \in V$, there is exactly one shortest path U from the root to v_i, and if this path has length ℓ, we say that v_i is on *level* ℓ of the tree. For each vertex $v_i \in V$ on level ℓ, every neighbor v_j of v_i on level $\ell + 1$ is called a *child of* v_i. For each such child v_j, the vertex v_i is the only adjacent vertex on level ℓ and is called the *parent of* v_j. All other children of v_i that are not v_j itself are called *siblings of* v_j. If the maximum level of any vertex in the tree is d, then d is called the *depth of the tree*.

Sometimes, we consider q-ary trees, for some $q \in \mathbb{N}^{\geq 2}$. In the literature, a q-*ary tree of depth* d is often defined as a rooted tree in which all inner vertices have at most q children and the maximum level among all vertices is d. For our purposes, we choose a more restrictive definition and define a q-*ary tree of depth* d to be a rooted tree in which all inner vertices have exactly q children and all leaves are on the same level d. In our case, each q-ary tree has exactly q^d leaves. For $q = 2$, we call such a q-ary tree a *binary tree*. Hence, in a binary tree of depth d, each inner vertex has 2 children, and the number of leaves is 2^d.

1.3 Online Computation

The classical scenario of online computation can be viewed as a game between an online algorithm and an adversary. The game played is determined by the given *online optimization problem*. The goal of the *adversary* is to construct a problem instance that is as hard as possible for the online algorithm. The aim of the *online algorithm*, also called *online strategy*, is to compute a good solution on the input instance generated by the adversary. The *input instance* (also *input sequence* or just *input*) is given to the algorithm as a sequence $I = (x_1, \ldots, x_n)$ of *requests*, exactly one request in each *round*. Hence, the number of rounds corresponds to the length of the input sequence, which we usually denote by n. The online algorithm has to respond immediately to each request given in round i, i. e., before the next request arrives, with an irrevocable *output* y_i. The output sequence of an online algorithm A on an input I is then $(y_1, \ldots, y_n)$, and we denote it by $A(I)$.

To be able to compare the quality of online algorithms and their computed solutions, we assign a value to each solution according to its quality. Depending on the nature of the optimization problem, the algorithm is to achieve values either as small or as large as possible. In the former case, the optimization problem is called an *online minimization problem*; in the latter case, an *online maximization problem*. For minimization problems, the function that serves to assign a value to each solution is usually called a *cost function*; for maximization problems, we call this function a *gain function* accordingly. The value of a particular solution is then called the *cost* or the *gain* of this solution, respectively. For any instance I, a solution

$A(I)$ computed by an algorithm A is *optimal* if it has minimum cost or maximum gain, respectively, among all solutions computed on I by all possible algorithms. Then we say that A is *optimal on* I. An algorithm that is optimal on every possible input instance is called *optimal* and usually denoted by Opt throughout this thesis. Obviously, without knowing the whole input instance in advance, it is not possible for an online algorithm to be optimal in general; with this lack of knowledge, the online algorithm A might make a decision in some round i that turns out to be suboptimal later, when a larger part of the input sequence is known. Thus, receiving the input sequentially is a huge drawback compared to receiving it completely before the start of the computation, as so-called *offline algorithms* do. Hence, we are interested in the quality of the given online algorithm, which is usually measured by means of the *competitive ratio*, a measure of "how close to optimal" the algorithm is. This means that we compare the online algorithm to an optimal offline algorithm with unbounded memory and computing power. For any instance I and any online algorithm A, the solution $A(I)$ computed by A on I is c-*competitive* if there is a constant α independent of I such that

$$\mathrm{cost}(A(I)) \leq c \cdot \mathrm{cost}(\mathrm{Opt}(I)) + \alpha \tag{1.1}$$

for a minimization problem, and

$$\mathrm{gain}(\mathrm{Opt}(I)) \leq c \cdot \mathrm{gain}(A(I)) + \alpha \tag{1.2}$$

for a maximization problem. An online algorithm A is c-competitive if (1.1) or (1.2), respectively, holds for any possible input instance; hence, if A computes a c-competitive solution on any possible input instance I. Thus, A has a *competitive ratio* of at most c if there is a constant α such that, for any instance I, the solution computed by A on I is c-competitive. We say that an online algorithm as well as a solution is *strictly* c-*competitive* if the corresponding inequality holds for $\alpha \leq 0$. An optimal algorithm is strictly 1-competitive. (Diverging from some examples in the literature, we use two different formulas for minimization and maximization problems, making sure that the competitive ratio is always at least 1.)

In this way, we can analyze the competitive ratio of a given online algorithm or investigate what is the best achievable competitive ratio of any online algorithm for a given online optimization problem. Since it has been introduced in 1985 by Sleator and Tarjan [ST85], the competitive ratio has developed to the most relevant measure of the quality of online algorithms. Being a worst-case measurement, the competitive ratio in the game between an online algorithm and an adversary has proven to be very helpful in analyzing the hardness of online problems [BEY98].

In this game, the online algorithm can be strengthened by allowing it to use random bits. In this case, we are talking about a *randomized online algorithm*. If the adversary already knows these random bits before it has to construct its hard input instance, the randomization is obviously utterly useless. Therefore, usually it is assumed that the adversary knows the randomized online algorithm, but not the random bits generated during the computation, and it must hence construct its hard input before the random bits are generated. (Note that also other models exist; for an overview, we refer to Chapter 4.1.1 in the textbook by Borodin and El-Yaniv [BEY98].) In our case, the competitive ratio is a random variable that depends on the sequence of random bits. Therefore, when measuring the quality of randomized online algorithms, we commonly analyze the *expected competitive ratio*, i. e., the expected value of this random variable over all possible random sequences. A randomized online algorithm can be interpreted as a set of deterministic online strategies. For the input constructed by the adversary, one of these determinisic strategies is chosen at random, and thus, the adversary has to play against a collection of several deterministic online algorithms at once. For many online problems, randomization can be very helpful; in some cases, the competitive ratio can even be decreased exponentially or more, compared to the competitive ratio of the provably best deterministic online algorithm. One example for an exponential improvement is the paging problem (with cache size k), for which a $\Theta(\log k)$-competitive randomized algorithm exists, whereas any deterministic algorithm cannot be better than k-competitive [FKL$^+$91].

1.4 Online Computation With Advice

Dobrev et al. [DKP08] introduced the idea of a third player in online computation. This third player is called an *oracle* or *advisor* and is omniscient, i. e., it has knowledge of the whole input sequence chosen by the adversary, and it is computationally unbounded. This oracle is allowed to provide *advice bits* to the given online algorithm in order to reveal some information about the yet unknown parts of the input and thereby help the online algorithm to achieve a better competitive ratio. For a given online problem, the research questions posed are quantative ones, i. e., they are related to the number of advice bits necessary and sufficient to improve the competitive ratio of a corresponding online algorithm essentially. Properties that are typically investigated are the number of advice bits necessary and sufficient to obtain optimal solutions or solutions with a specified upper or lower bound on the competitive ratio.

Since the model of Dobrev et al. was too rough in measuring the amount of advice bits, Hromkovič et al. [HKK10] proposed a general model which has already been successfully explored in many papers [Bar14, BBF$^+$14, BBH$^+$13, BBH$^+$14, BBHK12, BHK$^+$14, BKK$^+$09, BKKK11, BKKR12, BKLL14a, BKLL14b, DHZ12, DKK12, DKM12, Doh15, EFKR11, FKS12, GKK$^+$15, GKLO13, KK11, KKM12, RR11, SSU13, Weh15] and which we also consider in this thesis. In this model, given an online algorithm $\mathcal{A}$ with advice for some online optimization problem P with the set $\mathcal{I}_{\text{all}}$ of all possible inputs, the adversary first constructs a hard problem instance $I \in \mathcal{I}_{\text{all}}$, knowing both the oracle and $\mathcal{A}$. The oracle knows this instance I, which is given to $\mathcal{A}$ as its input sequence, and also the online algorithm itself. It provides an infinite sequence of advice bits depending on I and $\mathcal{A}$, which it writes onto a so-called *advice tape* before $\mathcal{A}$ starts its computation. This binary sequence is called the *advice string*. The online algorithm has access to the advice tape and may read some prefix of the advice string during its computation. The length of this prefix of the advice string read by $\mathcal{A}$ is the number of advice bits used throughout $\mathcal{A}$'s computation on I. For each problem P, we fix a problem parameter in which the number of advice bits used by $\mathcal{A}$ is measured; most often, this is the length n of the input sequence, but it may also be, for example, the graph size for online graph problems. We define $\mathcal{I}_{\text{all}}^{(m)}$ to be the set of all possible inputs of P for which the problem parameter is at most m; moreover, we define the *advice complexity* of $\mathcal{A}$ as the maximum number of advice bits read by $\mathcal{A}$ over all inputs $I \in \mathcal{I}_{\text{all}}^{(m)}$, denoted by $b(m)$. The online algorithm $\mathcal{A}$ augmented with an advice string τ is denoted by $\mathcal{A}^{\tau}$, or, omitting the superscript, just by $\mathcal{A}$, when the meaning is clear from the context. It may seem peculiar at first sight that the advice string is of infinite size; this property, however, ensures that $\mathcal{A}$ may not gain any additional knowledge from the length of the advice string.

The *competitive ratio* of $\mathcal{A}$ is defined according to the competitive ratio of online algorithms without advice. Thus, $\mathcal{A}$ is c*-competitive with advice complexity* $b(m)$ if there is a constant a such that, for every m and every input sequence $I \in \mathcal{I}_{\text{all}}^{(m)}$, there is an advice string τ such that

$$\text{cost}(\mathcal{A}^{\tau}(I)) \leq c \cdot \text{cost}(\mathtt{Opt}(I)) + a \tag{1.3}$$

for minimization problems, and

$$\text{gain}(\mathtt{Opt}(I)) \leq c \cdot \text{gain}(\mathcal{A}^{\tau}(I)) + a \tag{1.4}$$

for maximization problems, and if at most the first $b(m)$ bits of τ are accessed during the computation of $\mathcal{A}^{\tau}$ on I. As before, we say that an online algorithm with advice is *strictly* c*-competitive* or has a *strict competitive ratio of* c if the corresponding inequality holds for some $a \leq 0$.

Since the oracle knows the online algorithm $\mathcal{A}$, it also knows how $\mathcal{A}^{\tau}$, the online algorithm augmented with the advice string τ, operates on the input I, for each particular advice string τ. Thus, it can already generate an advice string yielding a particular desired competitive ratio before the computation starts, and it does not gain anything by changing the advice string throughout the computation. Therefore, we assume that the advice string is left unaltered once the computation starts. This implies that, for any online algorithm $\mathcal{A}$ with advice and any advice string τ, the algorithm $\mathcal{A}^{\tau}$ operates completely deterministic. In other words, for a fixed advice string, the online algorithm with advice is a deterministic algorithm, and we can interpret an online algorithm with advice (just like a randomized online algorithm) as a set of deterministic strategies. Hence, for any online algorithm $\mathcal{A}$ with an advice complexity of $b(m)$ (which thus reads at most $b(m)$ advice bits on any input from $\mathcal{I}_{\text{all}}^{(m)}$), we can derive the following fact, which has been discussed in detail by, for example, Komm [Kom12].

Fact 1.1. *Let* P *be an online optimization problem and* m *be a problem parameter of* P*. Furthermore, let* $\mathcal{A}$ *be an online algorithm for* P *reading* $b := b(m)$ *advice bits on each input from* $\mathcal{I}_{\textit{all}}^{(m)}$*. Then, we can interpret* $\mathcal{A}$ *as a set* $\mathcal{A} = \{A_1, \ldots, A_{2^b}\}$ *of* 2^b *different deterministic online algorithms without advice.* □

We benefit from this in the following way. Sometimes, when attempting to prove lower bounds on the advice complexity, we take the following approach. For each possible value of m, we determine a set $\mathcal{I} := \mathcal{I}(m)$ of hard input instances for the problem considered and show that any deterministic algorithm A can only achieve a competitive ratio larger than c on many instances from $\mathcal{I}$. From this, we can derive a lower bound on the number of advice bits necessary for each online algorithm with advice to achieve c-competitiveness, as the following observation shows.

Observation 1.2. *Let* $\mathcal{I}$ *be a subset of all possible input instances for some online optimization problem* P*. If any deterministic online algorithm* A *for* P *can achieve a competitive ratio of* c *on at most* $f \cdot |\mathcal{I}|$ *instances from* $\mathcal{I}$*, then every online algorithm with advice needs to read at least* $\log(1/f)$ *advice bits to be* c*-competitive.*

Proof. We prove the claim by contradiction. Hence, let us assume that there is a c-competitive online algorithm $\mathcal{A}$ with advice reading $b := b(m) < \log(1/f)$ advice bits. According to Fact 1.1, this online algorithm can be interpreted as a set $\mathcal{A} = \{A_1, \ldots, A_{2^b}\}$ of $2^b < 1/f$ deterministic online algorithms. Let $\mathcal{I}_i$ be the set of instances from $\mathcal{I}$ on which A_i achieves a competitive ratio of at most c, and let us say that A_i *covers* the instances from $\mathcal{I}_i$. Since $\mathcal{A}$ is c-competitive, it computes a c-competitive solution on any instance, thus in particular on any instance from $\mathcal{I}$. Hence, for every instance $I \in \mathcal{I}$, at least one of the 2^b deterministic algorithms A_i covers I. On average, each algorithm A_i thus covers $|\mathcal{I}|/2^b$ instances, and by the pigeonhole principle, there must be at least one deterministic algorithm in $\{A_1, \ldots, A_{2^b}\}$ covering $|\mathcal{I}|/2^b > f \cdot |\mathcal{I}|$ instances. This is a contradiction to our assumption that each deterministic online algorithm can achieve a competitive ratio of c on at most $f \cdot |\mathcal{I}|$ instances. □

We will make use of this observation several times throughout this thesis.

1.5 The String Guessing Problem

In this section, we give a formal definition of the already mentioned string guessing problem, which will play an important role throughout this thesis. The version we use here was introduced by Böckenhauer et al. [BHK+13]. A very similar problem with a different cost function has already been defined by Emek et al. [EFKR11], where it was called the *generalized matching pennies problem (GMP)*.

Definition 1.3 (String Guessing Problem (Known History)). *The* string guessing problem with known history over an alphabet Σ of size $z \geq 2$ *(z-*GUESS *for short) is the following online minimization problem. The input sequence $I_r = (n, r_1, \ldots, r_n)$ consists of a natural number n and the letters $r_1, \ldots, r_n \in \Sigma$ that are revealed one by one in the corresponding rounds. An online algorithm $\mathcal{A}$ for z-*GUESS *computes the output sequence $\mathcal{A}(I_r) = (y_1, \ldots, y_n)$, where $y_i = \mu(n, r_1, \ldots, r_{i-1}) \in \Sigma$, for some computable function μ. The algorithm is not required to respond with any output in the last round. The cost* $\mathrm{cost}(\mathcal{A}(I_r))$ *of a solution $\mathcal{A}(I_r)$ is the number of incorrectly guessed letters, i. e., the Hamming distance* $\mathrm{Ham}(r, y)$ *between $r = r_1 \ldots r_n$ and $y = y_1 \ldots y_n$.*

Böckenhauer et al. [BHK+14] also defined an alternative version of the problem, called the *string guessing problem with unknown history*. In this variant, the algorithm does not get any feedback about the validity of its guesses in former rounds, whereas in the version considered in this thesis, the algorithm is always informed whether its previous guess was correct. Surprisingly, an online algorithm with advice might actually profit from this knowledge in later rounds [Kru15]. In this thesis, however, we will only consider the version with known history as defined in Definition 1.3. Thus, whenever mentioning the string guessing problem, we are referring to the string guessing problem with known history, without explicitly stating this.

In the subsequent chapters, we will often make use of the following two results that were proven by Böckenhauer et al. [BHK+14].

Fact 1.4. *Consider an input string of length n for z-*GUESS*, for some $n \in \mathbb{N}$. Every online algorithm that guesses more than αn letters correctly, for $1/z \leq \alpha < 1$, needs to read at least*

$$\left(1 + (1-\alpha)\log_z\left(\frac{1-\alpha}{z-1}\right) + \alpha \log_z \alpha\right) n \log_2 z = \big(1 - \eta_z(1-\alpha)\big) n \log_2 z$$

advice bits. □

Fact 1.5. *Consider an input string of length n for 2-*GUESS*, for some $n \in \mathbb{N}$. Every online algorithm that guesses more than αn bits correctly, for $1/2 \leq \alpha < 1$, needs to read at least*

$$(1 + (1-\alpha)\log_2(1-\alpha) + \alpha \log_2 \alpha) n = (1 - \eta_2(1-\alpha)) n$$

advice bits. □

The second fact is a direct consequence of the first one, obtained by setting the alphabet size to $z := 2$. Recall that η_z is the z-ary entropy function and that η_2 is the binary entropy function, which we already defined in Section 1.2.3 and which is depicted in Figure 1.1.

2 k-SERVER on a Path

The first problem we deal with in this dissertation is the so-called k-*server problem*, a very famous online problem that has been extensively studied since 1988, when it was proposed by Manasse et al. [MMS88].

The k-server problem, also denoted by k-SERVER, is the following online minimization problem. Given is a metric space and a sequence of *requests*, each of them represented by a point in the metric space, and k servers. Often the metric space is considered to be a weighted graph. In this case, the servers can be located at any vertex of the graph and can be moved independently by the k-SERVER algorithm along the edges of the graph. Each request has to be satisfied by moving at least one server to the requested vertex. In the beginning, each server is located at some specified position; these positions are given to the k-SERVER algorithm in form of a k-tuple called the *starting configuration*. The algorithm's response to each request is a configuration of the servers such that the current request is always covered by a server. The cost for satisfying a request is the number of edges traversed by all the servers in total. The goal is to serve all requests with as little costs as possible for the whole request sequence.

Many results concerning the classical k-server problem are summarized in Chapters 10 and 11 of the textbook by Borodin and El-Yaniv [BEY98] (status as of 1998). More recently, Koutsoupias recorded a detailed survey of the k-server problem, covering the most important results that were published until 2009 [Kou09].

When the k-server problem was introduced by Manasse et al. [MMS88], the authors also proposed the famous k-server conjecture, which states that the competitive ratio of the k-server problem is exactly k, independently of the underlying metric space. That no online algorithm can have a better competitive ratio than k is a straight forward result, as long as the graph contains at least $k+1$ vertices (also shown in the previously cited paper [MMS88]). The question whether or not the other part of the conjecture holds, namely that each graph allows for a deterministic algorithm with competitive ratio k, has been unsolved ever since. So far, the conjecture has only been proven for some special cases, including, for example, the case that the underlying metric space is the real line [CKPV91], and the case that $k = 2$ [MMS88]. Chrobak and Larmore showed that there is a k-competitive algorithm on trees [CL91]. Fiat et al. presented the first upper bound on the competitive ratio for general metric spaces that depends only on k and not on the underlying graph [FRR94]. The bound they give is a function that is exponentially growing in k. Since then, there have been several publications, gradually improving upon the yet best known result. One of these improvements was presented by Koutsoupias and Papadimitriou [KP95], who showed that the so-called work function algorithm has a competitive ratio of $2k - 1$ on general metric spaces. Up until now, this remains the best known deterministic algorithm for general weighted graphs. Hence, despite a tremendous amount of research that has been conducted in this area, all efforts to prove the k-server conjecture for the general case have been unsuccessful so far. Nevertheless, the conjecture is commonly believed to be true.

In the hope of achieving better competitive ratios, it is possible to augment an online algorithm with random bits. Some results for this randomized setting are summarized in Chapter 11 of the already cited textbook [BEY98]. Among other things, they adapt a randomized k-SERVER algorithm presented by Blum et al. [BRS97] that works on a path with $n = k + 1$ vertices. The upper bound on the expected competitive ratio of the original algorithm grows exponentially in $\sqrt{\log n \log\log n}$; the competitive ratio of the adapted algorithm, working on a circle with circumference n and $k = n - 1$ servers, is at most $O\left(\sqrt{n}\log n\right)$. For the randomized setting, there exists the so-called *randomized k-server conjecture*, stating that for every weighted graph, there is a randomized online algorithm with competitive ratio $O(\log k)$. Just like the conjecture in the deterministic scenario, the randomized k-server conjecture has been an open question for many years. It was only in 2011 that Bansal et al. [BBMN11] proposed the first randomized algorithm for general graphs with a competitive ratio of $O(\log^2 k \log^3 n \log\log n)$, which is polylogarithmic in k if the number of vertices in the underlying graph is polynomial in k. Very recently, an article was published on arXiv [Che14] in which the author claims to improve this result to a competitive ratio of $O(\log k \log n)$ for general graphs and to give randomized algorithms with competitive ratios of at most $O(\log k)$ for some metric spaces, including lines and circles.

The most interesting line of research to us is, of course, the research concerning the advice complexity of the k-server problem. As with random bits, one can also augment the algorithm with advice bits to obtain better competitive ratios. The first result on the k-server problem with advice was presented by Emek et al. [EFKR11].

However, they considered a different model, in which the algorithm reads the same number of advice bits in every round. Concerning the advice model in this thesis, Böckenhauer et al. [BKKK11] showed that a competitive ratio of $2\lceil\lceil\log k\rceil/(d-1)\rceil$ can be achieved with dn advice bits, for $d \leq k$. Renault and Rosén improved this result [RR11] to a competitive ratio of $\lceil\lceil\log k\rceil/(d-2)\rceil$. Gupta et al. [GKLO13] investigated the k-server problem with advice within the framework of sparse metric spaces. They presented online algorithms with constant competitive ratios for many kinds of sparse graphs; amongst others, a 3-competitive algorithm for planar graphs reading $O(n \log\log m)$ advice bits, with m being the number of vertices in the graph. Furthermore, they proved a linear lower bound on the advice bits necessary for competitive ratios in the range of $1 < c < 5/4$, by a reduction from the string guessing problem.

In this thesis, we focus on the k-server problem on unweighted undirected paths (also denoted by k-PATHSERVER) for $k = 2$. We will use this rather simple setting in this chapter to demonstrate an application of the string guessing problem 2-GUESS presented in Chapter 1. The result we are aiming at is a statement of the form that there is no online algorithm with advice for 2-PATHSERVER reading only few advice bits that achieves a good competitive ratio. By giving a reduction from 2-GUESS to 2-PATHSERVER, a result from Böckenhauer et al. (Fact 1.5) directly translates to such a non-existence proof. A remarkable fact is that, although very generic, the string guessing problem often proves to be very useful in obtaining such lower bounds. We demonstrate this exemplarily for 2-PATHSERVER.

More precisely, the purpose of this chapter is the following. We want to acquaint the reader with the string guessing problem, reductions from 2-GUESS to other online problems, and the method of using these reductions to obtain lower bounds on the number of advice bits any online algorithm needs to achieve certain competitive ratios. We do so by giving two different reductions from 2-GUESS to 2-PATHSERVER. These reductions yield a certain trade-off, respectively, between the number of advice bits read by an online algorithm and the competitive ratio it achieves. As a side effect, we also obtain results for the case that the online algorithm does not read any advice bits at all, which is equivalent to the deterministic scenario. As mentioned before, in the paper by Chrobak et al. [CKPV91], the k-server conjecture was already proven for the real line, and hence also for unweighted paths, in the deterministic setting. Hence, it is clear that the competitive ratio in our scenario is exactly 2 in the deterministic case. The results we acquire in this chapter for the deterministic case are not better than these already known ones; our interest rather lies in the obtained trade-offs.

The remainder of this chapter is organized in three sections. In Section 2.1, we present a few simple already known results that we will need in the following sections, and we briefly describe the general procedure of reducing the string guessing problem to another online problem. Then, in Section 2.2, we present a reduction from 2-GUESS to 2-PATHSERVER, which yields a trade-off for competitive ratios between 1 and 3/2. More concretely, we will prove a lower bound of roughly $(1-\eta(\alpha))n/5$ advice bits necessary to obtain a strict competitive ratio of less than $2-\alpha$ on instances of length n, for any α with $1/2 \leq \alpha < 1$, where η is the binary entropy function as defined in Section 1.2. This is an improvement over the result mentioned above by Gupta et al. [GKLO13] as the trade-off obtained applies to a much wider range of competitive ratios. Finally, Section 2.3 will deal with the second such reduction, which gives a better trade-off for competitive ratios very close to 1, namely a lower bound of roughly $(1-\eta(\alpha))n/2$ advice bits for a strict competitive ratio of roughly $1+1/2^{\alpha n}$ on instances of length n.

2.1 Preliminaries

Before we give a formal definition of the general k-server problem, let us formally introduce the notion of a configuration of an algorithm. Let $G = (V, E)$ be the underlying graph with vertex set V and edge set E, and let each edge from E be assigned a weight by a given weight function ω. For each pair of vertices $(u, v) \in V^2$, let $\sigma(u, v)$ denote the length of the shortest path from u to v in G. For each algorithm $\mathcal{A}$ for k-SERVER and each round t, let $\kappa_t^{\mathcal{A}} \in V^k$ be an ordered set, such that each $\kappa_t^{\mathcal{A}}(i)$ indicates the position of server s_i in the graph at the beginning of round t during the execution of $\mathcal{A}$ on an instance I containing n requests. Whenever the algorithm we refer to is clear from the context, we also write κ_t instead of $\kappa_t^{\mathcal{A}}$ and $\kappa_t(i)$ instead of $\kappa_t^{\mathcal{A}}(i)$.

Definition 2.1. *The* k-server problem on weighted graphs *is the following online minimization problem on a graph $G = (V, E)$ with a metric weight function $\omega\colon E \to \mathbb{Q}$ of the edges. Given is an input sequence $I = (q_0, \ldots, q_n)$ that consists of $n+1$ queries, presented one by one in consecutive rounds. The query q_0 arriving in round 0 is a pair consisting of the underlying graph and the starting configuration $\kappa_0 \in W^k$, i. e., the starting positions of the k servers $s_0, \ldots, s_{k-1}$. The subsequent n queries $q_1, \ldots, q_n$ are the positions at which the requests arrive.*

An algorithm $\mathcal{A}$ for k-SERVER *has to satisfy each request q_i, for $1 \leq i \leq n$, by sending at least one of the servers to the vertex q_i immediately, i. e., before the next request arrives. Hence, $\mathcal{A}$ has to respond to each request q_i with a configuration κ_{i+1} of the servers, such that at least one server is located at position q_i in κ_{i+1}. Hence, for each i with $1 \leq i \leq n$, it must hold that $q_i \in \kappa_{i+1}^{\mathcal{A}}$. Formally, the output of $\mathcal{A}$ is a sequence $\mathcal{A}(I) = (\kappa_1, \ldots, \kappa_{n+1})$.*

For each request q_i and each server s_j, the distance traveled by s_j in round i during the execution of $\mathcal{A}$ on an instance I is $\sigma(\kappa_{i+1}^{\mathcal{A}}(j), \kappa_i^{\mathcal{A}}(j))$. The cost incurred in round i is then $\sum_{j=0}^{k-1} \sigma(\kappa_{i+1}^{\mathcal{A}}(j), \kappa_i^{\mathcal{A}}(j))$, and the total cost of $\mathcal{A}$ on I is $\mathrm{cost}(\mathcal{A}(I)) = \sum_{i=1}^{n} \sum_{j=0}^{k-1} \sigma(\kappa_{i+1}^{\mathcal{A}}(j), \kappa_i^{\mathcal{A}}(j))$. The goal is to minimize $\mathrm{cost}(\mathcal{A}(I))$.

We start with a few basic observations. First, let us remark that a k-SERVER algorithm could position several servers at the same vertex at any time during its computation. Sometimes it makes the argumentation easier to assume that this does not happen. We observe that, for every algorithm placing more than one server at the same vertex in some round t, there is an algorithm that always places at most one server at the same vertex and that does not have a larger cost. Thus, in the following, we will assume without loss of generality that, in all rounds, all servers are positioned at pairwise different vertices.

Moreover, it has been shown that we can restrict our considerations to so-called lazy algorithms [MMS90]. We call an algorithm for k-PATHSERVER *lazy* if it moves at most one server per request q_i, and only if the request is not already covered by a server. Formally speaking, an algorithm is lazy if, for each i with $1 \leq i \leq n$, the following holds. If there is a server s_h such that $\kappa_i^{\mathcal{A}}(h) = q_i$, then $\kappa_{i+1}^{\mathcal{A}}(j) = \kappa_i^{\mathcal{A}}(j)$ for all servers s_j, and otherwise, there is at most one server s_h such that $\kappa_{i+1}^{\mathcal{A}}(h) \neq \kappa_i^{\mathcal{A}}(h)$. Constraining ourselves to lazy algorithms is reasonable in so far as every non-lazy algorithm can be simulated by a lazy one without increasing the cost of the computed solution. Due to this restriction, we can assume that each response y_i of $\mathcal{A}$, for $1 \leq i \leq n$, consists only of the index of the server that is sent by $\mathcal{A}$ to satisfy the corresponding request q_i. (Or, if it is already covered, the server that is currently located at q_i.) Furthermore, because in round 0 no position on the path is requested, we can assume that the output y_0 of $\mathcal{A}$ is the empty string ε. (On the other hand, remarkably, one of the most famous k-SERVER algorithms does not operate in a lazy manner, namely the so-called *Double Coverage algorithm* [CKPV91]).

Due to these considerations, we can make a few simplifications, leading to the following definition of the version of the k-server problem that we consider in this thesis, the special case in which the underlying graph is an unweighted path of finite length.

Definition 2.2. *The* k-server problem on finite paths (k-PATHSERVER)*, is the following online minimization problem on a path $P = (0, \ldots, \ell)$. Given is an input sequence $I = (q_0, \ldots, q_n)$ that consists of $n + 1$ queries which are given one by one in consecutive rounds. The query q_0 arriving in round 0 is a pair consisting of the length ℓ of the path and the starting configuration $\kappa_0 \in W^k$. The subsequent n queries $q_1, \ldots, q_n$ are the positions at which the requests arrive. The online algorithm $\mathcal{A}$ has to respond to each such request q_i with the index of the server that it chooses to cover this request. Formally, the output of the algorithm $\mathcal{A}$ for* k-PATHSERVER *on an instance I is $\mathcal{A}(I) = (\varepsilon, y_1, \ldots, y_n)$ with $y_i \in \{0, \ldots, k-1\}$ for $1 \leq i \leq n$.*

For each request q_i, the number of edges traversed by s_{y_i} in round i during the execution of $\mathcal{A}$ on I is $|\kappa_{i+1}^{\mathcal{A}}(y_i) - \kappa_i^{\mathcal{A}}(y_i)|$. The total cost of $\mathcal{A}$ on I is $\mathrm{cost}(\mathcal{A}(I)) = \sum_{i=1}^{n} |\kappa_{i+1}^{\mathcal{A}}(y_i) - \kappa_i^{\mathcal{A}}(y_i)|$.

Now, let us briefly consider why a simple greedy strategy cannot be competitive for 2-PATHSERVER (and therefore not for k-SERVER, either). The most straight forward greedy approach is to serve each request with the server that is closest to it. Let the given path be $(0,1,2,3)$ of length 3, with server s_0 starting at vertex 0 and s_1 at vertex 3. Consider the request sequence $I = (1,0,1,0,1,0,\ldots)$ of length n. The greedy algorithm A serves every request from I with server s_0, since it is always the closest one, resulting in the output sequence $A(I) = (0,0,0,\ldots)$, which induces a cost of n. An optimal algorithm $\mathtt{Opt}$, on the other hand, sends s_1 to the first request and then does not have to move any server during the remaining computation. The resulting output sequence is $\mathtt{Opt}(I) = (1,0,1,0,1,0,\ldots)$ with a cost of 2, which yields a competitive ratio of $n/2$.

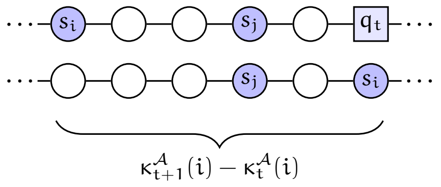
Another observation that will come in handy later is that it is never useful for an algorithm to swap the order of two servers. Hence, we show that, if $\kappa_0(i) < \kappa_0(j)$ holds in the initial configuration for any two servers s_i, s_j, then $\kappa_t(i) < \kappa_t(j)$ holds for every round $t \geq 0$.

Lemma 2.3. *For any online algorithm $\mathcal{A}$ for* 2-PATHSERVER*, there exists an online algorithm $\mathcal{A}'$ for* 2-PATHSERVER *with* $\text{cost}(\mathcal{A}') \leq \text{cost}(\mathcal{A})$ *that never swaps the order of any two servers during its execution.*

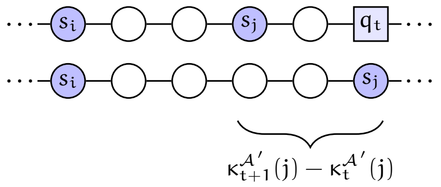
Proof. If algorithm $\mathcal{A}$ never swaps the order of two servers, the claim is obviously true. Hence, let us assume that there is at least one round in which $\mathcal{A}$ swaps the order of two servers, and let the first such round be t. Furthermore, let s_i and s_j be these two servers, and without loss of generality, let us assume that during the execution of $\mathcal{A}$ at the beginning of round t, the server s_i is to the left of server s_j, i. e., $\kappa_t^{\mathcal{A}}(i) < \kappa_t^{\mathcal{A}}(j)$. Also without loss of generality, we assume that the server sent to satisfy request q_t in round t is s_i. As we only consider lazy algorithms, s_i is the only server moved in round t. Since according to our assumption the order of the servers is swapped in round t, the request q_t must be located to the right of s_j. Analogous considerations can be made for the case that $\mathcal{A}$ sends s_j to q_t.

Let us compare the cost of algorithm $\mathcal{A}$ with an algorithm $\mathcal{A}'$ that operates exactly as $\mathcal{A}$ up to round $t-1$, but then sends server s_j to q_t instead of s_i. In round t, the algorithm $\mathcal{A}$ incurs a cost of $\kappa_{t+1}^{\mathcal{A}}(i) - \kappa_t^{\mathcal{A}}(i)$, whereas $\mathcal{A}'$ would incur a cost of $\kappa_{t+1}^{\mathcal{A}'}(j) - \kappa_t^{\mathcal{A}'}(j) = \kappa_{t+1}^{\mathcal{A}}(i) - \kappa_t^{\mathcal{A}}(j) < \kappa_{t+1}^{\mathcal{A}}(i) - \kappa_t^{\mathcal{A}}(i)$, since $\kappa_t^{\mathcal{A}}(i) < \kappa_t^{\mathcal{A}}(j)$.

In the remaining execution, $\mathcal{A}'$ can now satisfy all requests that $\mathcal{A}$ satisfies using s_i by sending server s_j instead and vice versa. Clearly, sending s_j instead of s_i cannot increase the cost compared to $\mathcal{A}$. The first round in which $\mathcal{A}'$ sends s_i instead of s_j (in case such a round exists) might actually have a larger cost compared to $\mathcal{A}$, but clearly, the extra cost cannot exceed $\kappa_t^{\mathcal{A}}(j) - \kappa_t^{\mathcal{A}}(i)$. Also, other servers are not affected, so the total cost of $\mathcal{A}'$ is at most $\text{cost}(\mathcal{A})$. The situation is depicted in Figure 2.1. □



(a) Algorithm $\mathcal{A}$ sends s_i to satisfy the request q_t appearing to the right of s_j in round t.


s_i
s_j
q_t
s_i
s_j
$\kappa^{\mathcal{A}'}_{t+1}(j) - \kappa^{\mathcal{A}'}_t(j)$

(b) Algorithm $\mathcal{A}'$ sends server s_j instead of s_i to q_t.

Figure 2.1. An example for the situation before (top) and after (bottom) round t for the algorithms $\mathcal{A}$ (left) and $\mathcal{A}'$ (right) from Lemma 2.3. At the beginning of round t, the server s_i is located to the left of s_j.

In the next two sections, we give two lower bounds for the number of advice bits an algorithm for 2-PATHSERVER needs to compute an optimal solution and to achieve certain competitive ratios, respectively. Both lower bounds are obtained by reducing 2-GUESS, the problem we defined in Chapter 1, to 2-PATHSERVER. Let us briefly describe how we usually proceed in this thesis when giving such a reduction from 2-GUESS to some other online problem P. The goal is to show the following implication. If there exists an algorithm for P with a competitive ratio of at most $c = c(\alpha)$, for $1/2 \leq \alpha < 1$, reading less than b advice bits, then there is an algorithm for 2-GUESS guessing more than αn bits correctly, while reading less than b advice bits. We prove this implication in the following way. Let $\mathcal{A}$ be an online algorithm for P with competitive ratio c, reading b advice bits. From $\mathcal{A}$, we construct an algorithm $\mathcal{B}$ for 2-GUESS guessing more than αn bits correctly, reading less than b advice bits, by doing the following. The algorithm $\mathcal{B}$ transforms its input $I_{\mathcal{B}}$ into an input $I_{\mathcal{A}}$ for $\mathcal{A}$, simulates $\mathcal{A}$ on $I_{\mathcal{A}}$, and transforms the output $\mathcal{A}(I_{\mathcal{A}})$ into an output $\mathcal{B}(I_{\mathcal{B}})$ for 2-GUESS. Every time that $\mathcal{A}$ needs a certain number of advice bits, $\mathcal{B}$ reads this number of advice bits from the oracle's advice tape and writes them onto an own dummy advice tape that $\mathcal{A}$ can access during its execution.

In the following two sections, we will demonstrate this procedure in greater detail.

2.2 2-PATHSERVER on a Path of Length 2

In this section, we only consider 2-PATHSERVER for paths of length 2, i. e., containing 3 vertices. Even for this very restricted case of 2-PATHSERVER, we prove a lower bound on the number of advice bits necessary to achieve a competitive ratio of c, for c in the range $1 \leq c \leq 1.5$. We obtain a trade-off between the number of advice bits and the competitive ratio by giving a reduction from 2-GUESS to 2-PATHSERVER. Hence, let $\mathcal{A}$ be an algorithm for 2-PATHSERVER. From $\mathcal{A}$, we construct an algorithm $\mathcal{B}$ for 2-GUESS. To do so, we have to show how to generate an input $I_{\mathcal{A}}$ for $\mathcal{A}$ from an input $I_{\mathcal{B}}$ for $\mathcal{B}$, and how to transform the output of $\mathcal{A}$ on $I_{\mathcal{A}}$ into an output for $I_{\mathcal{B}}$.

Let $I_{\mathcal{B}} := (n, r_1, \ldots, r_n)$ be the input instance for the string guessing algorithm $\mathcal{B}$ corresponding to the bit string $r := r_1 \ldots r_n$. From this, we generate an input instance $I_{\mathcal{A}}$ for $\mathcal{A}$ according to Definition 2.2 as follows. The first query of $I_{\mathcal{A}}$ is the pair $q_0 := (2, (0, 2))$, consisting of the length of the path and the starting configuration $\kappa_0 = (0, 2)$. After this query is presented in round 0, in each subsequent round i, the algorithm is given the request q_i, for $1 \leq i \leq 5n$. For each r_i with $1 \leq i \leq n$, we build a gadget consisting of five requests each. Let us define this gadget to be

$$\mathcal{Q}_i := \begin{cases} (1, 2, 1, 2, 0) & \text{if } r_i = 0, \\ (1, 0, 1, 0, 2) & \text{otherwise.} \end{cases}$$

Let us call the sequence $(1, 0, 1, 0, 2)$ a *type-1-gadget* and the sequence $(1, 2, 1, 2, 0)$ a *type-0-gadget*.

The complete request sequence $I_{\mathcal{A}}$ is obtained by concatenating the first query and then all gadgets $\mathcal{Q}_i$ in ascending order of their indices. Denoting the concatenation operator by $\circ$, we obtain

$$I_{\mathcal{A}} := \big((\ell, \kappa_0)\big) \circ \bigcirc_{i=1}^{n} \mathcal{Q}_i = \big((2, (0, 2))\big) \circ \mathcal{Q}_1 \circ \ldots \circ \mathcal{Q}_n .$$

Hence, $I_{\mathcal{A}}$ has length $5n + 1$. Giving this sequence to $\mathcal{A}$ as its input, $\mathcal{A}$ generates the output $\mathcal{A}(I_{\mathcal{A}}) := (\varepsilon, y_1, \ldots, y_{5n})$. From this output sequence, the output sequence for $\mathcal{B}$ is constructed as follows. If, for $0 \leq i \leq n - 1$, the first request of gadget $\mathcal{Q}_{i+1}$ is satisfied by server s_0, algorithm $\mathcal{B}$ guesses r_{i+1} to be 0; if this request is satisfied by s_1, the algorithm guesses r_{i+1} to be 1. Formally speaking, $g_{i+1} := y_{5i+1}$, for $0 \leq i \leq n - 1$.

We now investigate how to serve the request sequence $I_{\mathcal{A}}$ optimally. To this end, let us from now on say that, for each first request of a type-i-gadget, the server s_i is the *appropriate* server and s_{1-i} is the *inappropriate* server, for $i \in \{0, 1\}$.

Lemma 2.4. *Consider an input sequence $I_{\mathcal{A}}$ for* 2-PATHSERVER *constructed from an input instance $I_{\mathcal{B}}$ for* 2-GUESS *corresponding to a string r, for some $n \in \mathbb{N}^{\geq 1}$ and some $r \in \{0, 1\}^n$. Then, if $\mathcal{A}$ serves the first request of exactly αn gadgets with its appropriate server, the algorithm $\mathcal{A}$ has a cost of exactly* $\text{cost}(\mathcal{A}(I_{\mathcal{A}})) = 2n \cdot (2 - \alpha)$.

Proof. First we observe that, due to Lemma 2.3, each request at position 0 always has to be satisfied by s_0, and each request at position 2 by s_1. From this, it follows that, directly before the first request of each gadget $\mathcal{Q}_i$, the server s_0 is at position 0 and server s_1 is at position 2. This is true for the first request of $\mathcal{Q}_1$ due to the starting configuration, and it holds for the other gadgets $\mathcal{Q}_i$ since the last two requests of each gadget $\mathcal{Q}_{i-1}$ are always 0 and 2.

Hence, we can investigate the performance of algorithm $\mathcal{A}$ on each gadget separately. To this end, let us determine what would be the optimal way to satisfy the requests of gadget $\mathcal{Q}_1 = (q_1, \ldots, q_5)$. Without loss of generality, let $\mathcal{Q}_1$ be a type-1-gadget, i. e., $\mathcal{Q}_1 = (1, 0, 1, 0, 2)$. Analogous considerations can be made if $\mathcal{Q}_1$ is a gadget of type 0. The appropriate server for the first request of $\mathcal{Q}_1$ is s_1. The first request q_1 of $\mathcal{Q}_1$ is the middle vertex 1. No server is located there, so $\mathcal{A}$ must send one of the servers to this position in round 1. Let us first analyze the situation if $\mathcal{A}$ sends the appropriate server s_1 to satisfy q_1. In this case, the requests that arrive in rounds 2, 3, and 4 are all already covered by servers, and hence, $\mathcal{A}$ does not move any servers in these rounds. To satisfy the last request q_5 of this gadget, $\mathcal{A}$ has to send s_1 back to position 2. The cost incurred by $\mathcal{A}$ on $\mathcal{Q}_1$ if it sends s_1 to request q_1 is hence 2. See Figure 2.2a.

On the other hand, let us observe what happens if, instead of the appropriate server, s_0 is sent to q_1. In round 2, the server s_0 has to be sent back to position 0 to satisfy q_2, such that in the next round, the request q_3 at position 1 is uncovered. Now, $\mathcal{A}$ can either send s_0 to satisfy q_3, but then s_0 has to be moved back to position 0 in round 4 (depicted in Figure 2.2b), or $\mathcal{A}$ can send s_1 to satisfy q_3, but then s_1 has to be moved to position 2 in round 5 (see Figure 2.2c). In either case, the cost induced by $\mathcal{A}$ on gadget $\mathcal{Q}_1$ is exactly 4.

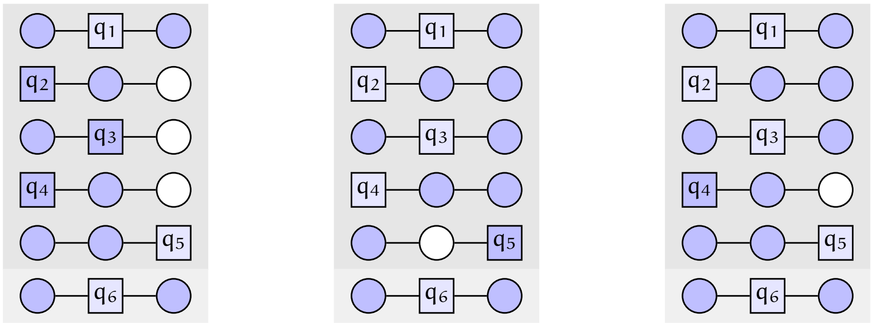
As a result, satisfying the first request of a type-1-gadget by s_1 incurs a cost of 2, satisfying it with s_0 incurs a cost of 4. For gadgets of type 0, the situation is symmetric. To obtain the cost for the whole request sequence, we can add up the costs of all the gadgets. Hence, if algorithm $\mathcal{A}$ serves the first request of exactly αn gadgets with the appropriate server and thus exactly $(1-\alpha)n$ requests with the inappropriate server, the algorithm has a total cost of exactly

$$2\alpha n + 4(1-\alpha)n = 2n\alpha + 2n(2-2\alpha) = 2n(2-\alpha). \qquad \square$$

Corollary 2.5. *Consider an input sequence $I_{\mathcal{A}}$ for 2-PATHSERVER constructed from an input instance $I_{\mathcal{B}}$ for 2-GUESS corresponding to a string r, for some $n \in \mathbb{N}^{\geq 1}$ and some $r \in \{0,1\}^n$. Then, the optimal solution on $I_{\mathcal{A}}$ has a cost of $2n$.*

Proof. According to Lemma 2.4, the cost of an algorithm on $I_{\mathcal{A}}$ is minimized for $\alpha = 1$, i. e., if the first requests of all n gadgets are satisfied by their appropriate servers. This yields a cost of $2n(2-\alpha) = 2n$. $\square$

To complete the reduction, we have to show a connection between the performance of the algorithm $\mathcal{A}$ for 2-PATHSERVER and the 2-GUESS algorithm $\mathcal{B}$



(a) Request q_1 is served by server s_1.

(b) Requests q_1 and q_3 are both served by server s_0.

(c) Request q_1 served by s_0 and q_3 by s_1.

Figure 2.2. The only three possibilities how to serve the requests of the gadget $(1, 0, 1, 0, 2)$. Depicted are the situations at the beginning of each round, when the request has arrived already, but the servers have not been moved yet. Positions with requests are shown as rectangles, those with servers are colored dark blue. The server s_0 is always left of s_1. The highlighting indicates which gadget the single requests belong to; request q_6 is already the first request of the next gadget.

constructed from $\mathcal{A}$. Therefore, we describe in greater detail how the string guessing algorithm $\mathcal{B}$ can be obtained from $\mathcal{A}$. The algorithm $\mathcal{B}$ gets the instance $I_{\mathcal{B}} = (n, r_1, \ldots, r_n)$ as its input, one request per round in $n+1$ consecutive rounds. To each request sent in round i with $0 \leq i \leq n-1$, it has to respond with its guess g_{i+1} for bit r_{i+1}. Now, in each round of its execution, $\mathcal{B}$ generates a piece of the input $I_{\mathcal{A}}$ according to the construction given above for 2-PATHSERVER, then simulates the algorithm $\mathcal{A}$ on this part of $I_{\mathcal{A}}$, and bases its own output on the output of $\mathcal{A}$. Since $\mathcal{B}$ and $\mathcal{A}$ have a different number of rounds and several rounds of $\mathcal{A}$ are simulated during one round of $\mathcal{B}$, let us call the rounds of $\mathcal{A}$ time steps for now. In round 0, the algorithm $\mathcal{B}$ simulates the first two time steps of $\mathcal{A}$ on $I_{\mathcal{A}} = (q_0, q_1, \ldots, q_{5n})$. Recall that the query q_0 sent to $\mathcal{A}$ in time step 0 is a special one, to which $\mathcal{A}$ only responds with $y_0 = \varepsilon$. Therefore, during the first round of $\mathcal{B}$, we simulate this time step and the first "real" one, in which the first request of gadget $\mathcal{Q}_1$ is sent to $\mathcal{A}$. In each following round i with $1 \leq i \leq n$, the algorithm $\mathcal{B}$ simulates the next five time steps of $\mathcal{A}$ on $I_{\mathcal{A}}$, with the exception of round n, in which only the last four time steps of $\mathcal{A}$ remain to be simulated. Hence, in round i with $1 \leq i \leq n-1$, the algorithm $\mathcal{B}$ sends requests $q_{5i-3}, \ldots, q_{5i+1}$ to $\mathcal{A}$ in five consecutive time steps, with q_{5i+1} being the first request of gadget $\mathcal{Q}_{i+1}$, and in round n, it sends requests $q_{5n-3}, \ldots, q_{5n}$.

To deal with the advice bits that $\mathcal{A}$ might try to read from the tape, $\mathcal{B}$ constructs its own "dummy advice tape". Every time $\mathcal{A}$ wants to access the oracle's advice tape in some time step j to read a certain number of advice bits, $\mathcal{B}$ reads the same number of advice bits from the oracle's advice tape and writes them onto its own one, unmodified. Then, $\mathcal{A}$ reads all its advice bits from the dummy advice tape, if any. After that, it computes its output y_j for time step j, potentially depending on the advice bits it read so far.

Finally, $\mathcal{B}$ adopts $\mathcal{A}$'s output y_{5i+1} as its guess g_{i+1} in each round i with $0 \leq i \leq n-1$, and ignores the output of $\mathcal{A}$ in all other time steps.

Lemma 2.6. *Consider a type-i-gadget $\mathcal{Q}_{j+1}$ and its first request q_{5j+1}, for any i, j with $i \in \{0,1\}$ and $0 \leq j \leq n-1$. If algorithm $\mathcal{A}$ sends the appropriate server s_i to satisfy this request, the algorithm $\mathcal{B}$ guesses the bit r_{j+1} correctly.*

Proof. Since $\mathcal{Q}_{j+1}$ is a type-i-gadget, due to our construction of the request sequence $I_{\mathcal{A}}$, it must hold that $r_{j+1} = i$.

On the other hand, the output of $\mathcal{A}$ in each time step consists of the index of the server sent to satisfy the current request. If algorithm $\mathcal{A}$ sends s_i to request q_{5j+1}, the output of $\mathcal{A}$ in time step $5j+1$ is hence $y_{5j+1} = i$. As we have defined above, $\mathcal{B}$'s guess for the bit r_{j+1} is defined as $g_{j+1} := y_{5j+1}$, for $0 \leq j \leq n-1$, and therefore, we have $g_{j+1} = i$. Thus, $\mathcal{B}$'s guess is obviously correct. □

Now we can derive a lower bound for the number of advice bits necessary to achieve certain competitive ratios for 2-PATHSERVER by using the result by Böckenhauer et al. [BHK+14] that we already presented in Chapter 1 (Fact 1.5). Recall that η is the binary entropy function, also defined in Chapter 1.

Theorem 2.7. *For any α with $1/2 \leq \alpha < 1$, any online algorithm $\mathcal{A}$ for 2-PATHSERVER with a strict competitive ratio of $c < 2-\alpha$ has to read $b \geq \big(1-\eta(\alpha)\big)n$ advice bits on instances of length $5n+1$, or, rewritten, $b \geq \big(1-\eta(\alpha)\big)/5 \cdot (n'-1)$ advice bits on instances of length n'.*

Proof. Towards contradiction, let us assume that there is an algorithm $\mathcal{A}$ that has a strict competitive ratio of less than $2-\alpha$, reading less than $\big(1-\eta(\alpha)\big)\, n$ advice bits. We show that, with the description given above, we can construct an algorithm for 2-GUESS from $\mathcal{A}$ reading the same number of advice bits as $\mathcal{A}$ and guessing more than αn bits correctly.

Since $\mathcal{A}$ has a strict competitive ratio of $c < 2-\alpha$, according to the definition given in Section 1.3, for every instance $I_{\mathcal{A}}$ of 2-PATHSERVER corresponding to an input string r for 2-GUESS,

$$\mathrm{cost}(\mathcal{A}(I_{\mathcal{A}})) < (2-\alpha)\,\mathrm{cost}(\mathrm{Opt}(I_{\mathcal{A}})).$$

Due to Corollary 2.5, the cost of an optimal solution on $I_{\mathcal{A}}$ is $\mathrm{cost}(\mathrm{Opt}(I_{\mathcal{A}})) = 2n$, and thus, we have $\mathrm{cost}(\mathcal{A}(I_{\mathcal{A}})) < 2n(2-\alpha)$. Then according to Lemma 2.4, $\mathcal{A}$ must serve the first request of more than αn gadgets with the appropriate server. Now, due to Lemma 2.6, the algorithm $\mathcal{B}$ for 2-GUESS that we constructed from $\mathcal{A}$ guesses more than αn bits correctly on any input string of length n, while reading less than $\big(1 - \eta(\alpha)\big)\, n$ advice bits. This is a contradiction to Fact 1.5; hence, the claim follows. □

As any deterministic online algorithm is an online algorithm with advice reading 0 advice bits, we can also derive a result for deterministic algorithms from Theorem 2.7. Let us keep in mind, though, that the k-server conjecture is already settled for the case of two servers on a path, as already mentioned earlier. Hence, the following result is not new, but only serves as a demonstration of the fact that lower bounds on the competitive ratio of online algorithms with advice can be transferred to lower bounds on the competitive ratio of deterministic online algorithms.

Corollary 2.8. *No deterministic online algorithm for* 2-PATHSERVER *can achieve a better strict competitive ratio than* 1.5*.*

Proof. Plugging $\alpha = 1/2+\delta$ into Theorem 2.7, for some arbitrary small $\delta > 0$, yields that every online algorithm with a strict competitive ratio of $c < 2-\alpha = 1.5-\delta$ on any input instance of length $5n+1$ has to read $b > 0$ advice bits, since $\eta(1/2+\delta) < 1$ for every $\delta > 0$, and hence $b \geq \big(1-\eta(\alpha)\big)\, n = \big(1-\eta(1/2+\delta)\big)\, n > 0$. Note that this also holds if δ is a function of n converging to 0. Thus, any online algorithm for 2-PATHSERVER with a strict competitive ratio of $c < 1.5$ needs to read at least one advice bit, and the claim follows. □

2.3 2-PATHSERVER on Finite Paths

For very small competitive ratios (in concrete terms, competitive ratios near 1), the result of the previous section can even be improved. The reduction in the last section was based on gadgets of size 5, each of which corresponded to one bit in the bit string of the input for 2-GUESS. As a result, we cannot expect to prove a better lower bound for the number of necessary advice bits than roughly $n/5$ for instances of length n. But, of course, we restricted ourselves to paths of length 2 in the last section. In this section, we want to make use of the fact that, actually, we may have a path of length ℓ at hand.

Hence, let the path we consider be $P = (0, \ldots, \ell)$. Using the same reduction technique as above, we want to generate an input for 2-PATHSERVER from a 2-GUESS instance. Let the input instance for 2-GUESS be $I_{\mathcal{B}} = (n, r_1, \ldots, r_n)$ with the corresponding bit string $r = r_1 \ldots r_n \in \{0,1\}^n$. From the bit string r, we construct an input instance $I_r = (q_0, q_1, \ldots, q_{2n})$ with $2n+1$ requests for 2-PATHSERVER on a path of length $\ell = 2^n$. The first request of I_r is a pair consisting of the length ℓ

of the path and the starting configuration, which we determine to be $\kappa_0 := (0, \ell)$. Thus, $q_0 := \big(\ell, (0, \ell)\big)$.

The remaining requests are chosen as follows. For each bit r_i, we build a gadget $\mathcal{Q}_i$ of size 2. The request sequence I_r is obtained by concatenating q_0 and then all gadgets in ascending order of their indices, hence,

$$I_r := q_0 \circ \mathcal{Q}_1 \circ \ldots \circ \mathcal{Q}_n.$$

This implies $\mathcal{Q}_i := (q_{2i-1}, q_{2i})$ for $1 \le i \le n$. As in the last section, we will have two different types of gadgets, and we will define $\mathcal{Q}_i$ to be a type-0-gadget if $r_i = 0$ and a type-1-gadget otherwise. In this section, we use the same notion as before, with server s_j being the appropriate and s_{1-j} being the inappropriate server for each type-j-gadget. Furthermore, similarly to the procedure in the last section, we build the gadgets such that it is optimal to serve the first request of a type-j-gadget with the appropriate server s_j and suboptimal to serve it with the inappropriate one, s_{1-j}.

We now give a more detailed description of how to choose the q_i. For convenience, let us define $\mathcal{Q}_0 := (0, \ell)$, such that $\min\{\mathcal{Q}_0\} = 0$ and $\max\{\mathcal{Q}_0\} = \ell$. Any first request of a gadget $\mathcal{Q}_i$, i. e., any odd request q_{2i-1}, arrives at position

$$q_{2i-1} := \frac{\min\{\mathcal{Q}_{i-1}\} + \max\{\mathcal{Q}_{i-1}\}}{2}.$$

Any even request q_{2i} appears at position

$$q_{2i} := \begin{cases} \min\{\mathcal{Q}_{i-1}\} & \text{if } r_i = 1, \\ \max\{\mathcal{Q}_{i-1}\} & \text{if } r_i = 0. \end{cases} \tag{2.1}$$

If $q_i = \min\{\mathcal{Q}_{i-1}\}$, let us call $\mathcal{Q}_i$ a type-1-gadget, and if $q_i = \max\{\mathcal{Q}_{i-1}\}$, let us call it a type-0-gadget. This request sequence has the following useful property.

Observation 2.9. *At the beginning of round* $2i + 1$*, there have not been any requests between* q_{2i-1} *and* q_{2i} *so far.*

Proof. We prove the claim by induction on the number of rounds. For $i = 1$, the claim obviously holds, since, at the beginning of round 3, requests q_1 and q_2 were the only requests presented so far. Let us now assume that the claim holds for $i - 1$. Then, at the beginning of round $2i - 1$, there have not been any requests between q_{2i-3} and q_{2i-2} yet. According to the construction of the request sequence, the next request appears in the middle between these two positions, hence, q_{2i-1} is the first request between q_{2i-3} and q_{2i-2} so far. The next one, q_i, appears at either the same position as q_{2i-3} or q_{2i-2}, depending on the string r. In any case, at the beginning of round $2i + 1$, there have not been any requests between q_{2i-1} and q_{2i} so far, which proves the claim. □

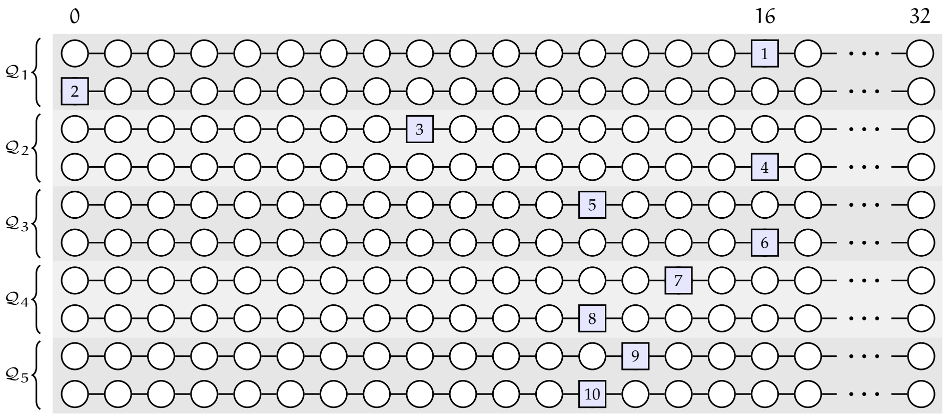


Figure 2.3. Example of the construction of the request sequence $(q_1, \ldots, q_{10})$ corresponding to the bit string $r = 10011$ of length $n = 5$. Every gadget $\mathcal{Q}_i = (q_{2i-1}, q_{2i})$ corresponds to one bit r_i. The picture shows the positions of the requests q_i on the path $P = (0, \ldots, \ell)$, with $\ell = 2^n = 32$. For better readability, each request q_i is only labeled with its index i. Every pair of requests (q_{2i-1}, q_{2i}) is determined according to the bit r_i, as indicated by the highlighting.

An example is shown in Figure 2.3.

Having constructed the instance $I_r = (q_0, \ldots, q_{2n})$ from the 2-GUESS-input $I_{\mathcal{B}}$, we can now run a 2-PATHSERVER-algorithm $\mathcal{A}$ on I_r. This algorithm generates the output $\mathcal{A}(I_r) = (y_0, \ldots, y_{2n})$, consisting of the indices of the servers sent to satisfy q_i for $1 \leq i \leq 2n$. From this output, we construct an output for $\mathcal{B}$ by adopting y_{2i+1} as $\mathcal{B}$'s guess in round $i+1$; hence, $g_{i+1} \coloneqq y_{2i+1}$ for $0 \leq i \leq n-1$.

To give another lower bound for the number of advice bits for 2-PATHSERVER, let us consider the following algorithm $\mathcal{A}_r$ for any bit string r. The algorithm $\mathcal{A}_r$ satisfies every odd request q_{2i-1} of I_r, i. e., the first request of each gadget $\mathcal{Q}_i$, by sending the appropriate server, for $1 \leq i \leq n$. Then, given the instance I_r as its input, $\mathcal{A}_r$ does not have to move any servers at all in even rounds, as we will see shortly. We show that $\mathcal{A}_r$ is optimal on the instance I_r corresponding to r. To do so, we first have to prove a few simple results. To follow the argumentation, it might help to view the example in Figure 2.4, which depicts the execution of $\mathcal{A}_r$ on the instance from Figure 2.3.

Observation 2.10. *For the instance* I_r,

$$r_i = 0 \iff \mathcal{Q}_i \textit{ is a type-0-gadget} \iff q_{2i-1} < q_{2i}.$$

Proof. This follows directly from the definitions of q_{2i-1} and q_{2i}. □

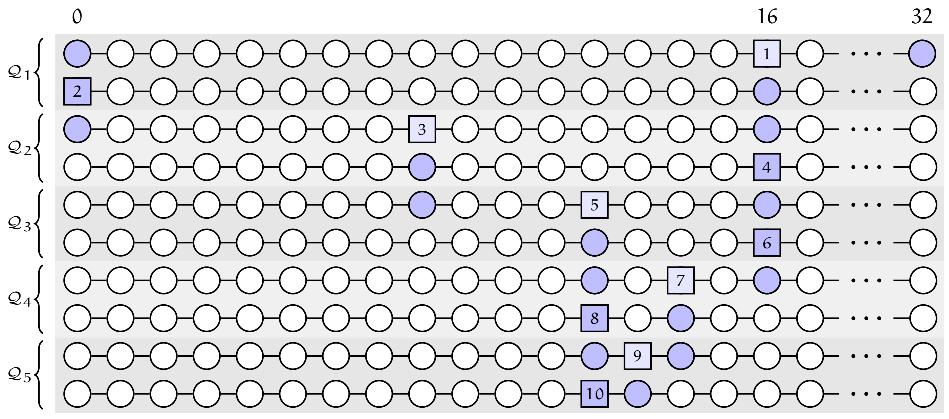


Figure 2.4. This picture shows how the algorithm $\mathcal{A}_r$ operates on the request sequence from Figure 2.3. In each even round, there is already one server positioned at the requested vertex q_{2i}, so no server has to be moved. The cost of $\mathcal{A}_r$ on this instance is $\ell - 1 = 31$.

Lemma 2.11. *During the execution of $\mathcal{A}_r$ on I_r, at the beginning of each even round $2i$, the appropriate server for q_{2i-1} is located at position $\kappa_{2i}(r_i) = q_{2i-1}$, and the inappropriate one is at position $\kappa_{2i}(1 - r_i) = q_{2i}$.*

Proof. We prove the claim by induction. If $r_1 = 1$, the algorithm $\mathcal{A}_r$ sends server s_1 to satisfy request q_1 according to the definition of $\mathcal{A}_r$. Hence, the configuration at the beginning of round 2 is $\kappa_2 = (0, \ell/2)$. Furthermore, according to the definitions of q_1 and q_2, we have $q_1 = \ell/2$ and $q_2 = 0$, and thus, $\kappa_2(r_i) = \kappa_2(1) = \ell/2 = q_1$ and $\kappa_2(1 - r_i) = \kappa_2(0) = 0 = q_2$. The case $r_1 = 0$ is analogous, so the base case is covered.

Due to the induction hypothesis, at the beginning of round $2i - 2$, servers are at positions $\kappa_{2i-2}(r_{i-1}) = q_{2i-3}$ and $\kappa_{2i-2}(1 - r_{i-1}) = q_{2i-2}$. Then no server has to be moved for request q_{2i-2}, and for q_{2i-1}, the algorithm $\mathcal{A}_r$ moves the appropriate server s_{r_i} according to its definition. Hence, at the beginning of round $2i$, the servers are at positions $\kappa_{2i}(r_i) = q_{2i-1}$ and $\kappa_{2i}(1 - r_i) = q_{2i-2}$. If $q_{2i-2} < q_{2i-3}$ and therefore $q_{2i-2} = \min\{\mathcal{Q}_{i-1}\}$, the server that $\mathcal{A}_r$ moves to q_{2i-1} must be s_1. Thus, $q_i = 1$, and due to (2.1), $q_{2i} = \min\{\mathcal{Q}_{i-1}\} = q_{2i-2}$. Otherwise, if $q_{2i-1} = \max\{\mathcal{Q}_{i-1}\}$, the server moved to q_{2i-1} must be s_0. Hence, $q_i = 0$, and due to (2.1), $q_{2i} = \max\{\mathcal{Q}_{i-1}\} = q_{2i-2}$. In both cases, we have $q_{2i-2} = q_{2i}$ and thus $\kappa_{2i}(1 - r_i) = q_{2i}$. □

Corollary 2.12. *The algorithm $\mathcal{A}_r$ does not move any servers in even rounds.*

Proof. This follows directly from Lemma 2.11, because at the beginning of each even round $2i$, one of the servers is already located at the position at which request q_{2i} arrives. □

Lemma 2.13. *During the execution of $\mathcal{A}_r$ on I_r, at the beginning of each odd round $2i-1$, the servers are located at positions $\kappa_{2i-1}(0) = \min\{\mathcal{Q}_{i-1}\}$ and $\kappa_{2i-1}(1) = \max\{\mathcal{Q}_{i-1}\}$.*

Proof. From Lemma 2.11 and Corollary 2.12 we know that, at the beginning of any odd round $2i-1$, the servers are at positions $\kappa_{2i-1}(r_{i-1}) = q_{2i-3}$ and $\kappa_{2i-1}(1 - r_{i-1}) = q_{2i-2}$. We have

$$\kappa_{2i-1}(r_{i-1}) = \begin{cases} \min\{\mathcal{Q}_{i-1}\} & \text{if } q_{2i-3} < q_{2i-2} \\ \max\{\mathcal{Q}_{i-1}\} & \text{if } q_{2i-3} > q_{2i-2} \end{cases} \tag{2.2}$$

$$= \begin{cases} \min\{\mathcal{Q}_{i-1}\} & \text{if } r_{i-1} = 0 \\ \max\{\mathcal{Q}_{i-1}\} & \text{if } r_{i-1} = 1, \end{cases} \tag{2.3}$$

where (2.2) holds due to the definition of $\mathcal{Q}_{i-1} := (q_{2i-3}, q_{2i-2})$ and (2.3) holds due to Observation 2.10. Analogously, we obtain

$$\kappa_{2i-1}(1 - r_{i-1}) = \begin{cases} \min\{\mathcal{Q}_{i-1}\} & \text{if } q_{2i-2} < q_{2i-3} \\ \max\{\mathcal{Q}_{i-1}\} & \text{if } q_{2i-2} > q_{2i-3} \end{cases}$$

$$= \begin{cases} \min\{\mathcal{Q}_{i-1}\} & \text{if } r_{i-1} = 1 \\ \max\{\mathcal{Q}_{i-1}\} & \text{if } r_{i-1} = 0. \end{cases}$$

Thus, for the positions of the servers at the beginning of round $2i-1$, we obtain $\kappa_{2i-1}(0) = \min\{\mathcal{Q}_{i-1}\}$ and $\kappa_{2i-1}(1) = \max\{\mathcal{Q}_{i-1}\}$. □

Lemma 2.14. *For $1 \leq i \leq n$, the distance between $\min\{\mathcal{Q}_{i-1}\}$ and $\max\{\mathcal{Q}_{i-1}\}$ is exactly $\ell/2^{i-1} = 2^{n-i+1}$.*

Proof. Due to Lemma 2.13, at the beginning of round $2i-1$, the servers are located at positions $\min\{\mathcal{Q}_{i-1}\}$ and $\max\{\mathcal{Q}_{i-1}\}$. The request in this round arrives at position $q_{2i-1} = (\min\{\mathcal{Q}_{i-1}\} + \max\{\mathcal{Q}_{i-1}\})/2$, i. e., in the middle between the two servers. Hence, in this round, one of the two servers is moved there, and the other one stays at its current position. Therefore, in every odd round, the distance between the two servers is halved, and due to Corollary 2.12, the distance does not change in even rounds in the solution computed by $\mathcal{A}_r$.

Prior to the start of computation, the distance between s_0 and s_1 is $\max\{\mathcal{Q}_0\} - \min\{\mathcal{Q}_0\} = \ell - 0 = \ell/2^0$. Thus, at the beginning of every round $2i-1$ with $0 \leq i \leq n$, the distance between $\min\{\mathcal{Q}_{i-1}\}$ and $\max\{\mathcal{Q}_{i-1}\}$ (and, therefore, also between the two servers as they are positioned in the solution of $\mathcal{A}_r$) is $\ell/2^{i-1} = 2^{n-i+1}$. □

Lemma 2.15. *The cost of $\mathcal{A}_r$ on instance I_r is* $\mathrm{cost}(\mathcal{A}_r(I_r)) = \ell - 1 = 2^n - 1$.

Proof. Due to Corollary 2.12, adding up the costs of all odd rounds yields the total cost of $\mathcal{A}_r$ on I_r. In round $2i-1$, the appropriate server s_{r_i} is moved by $\mathcal{A}_r$. At the beginning of this round, s_{r_i} is located at position

$$\kappa_{2i-1}(r_i) = \begin{cases} \min\{\mathcal{Q}_{i-1}\} & \text{if } r_i = 0, \\ \max\{\mathcal{Q}_{i-1}\} & \text{if } r_i = 1 \end{cases}$$

due to Lemma 2.13. After this round, i. e., at the beginning of round $2i$, server s_{r_i} is located at position

$$\kappa_{2i}(r_i) = q_{2i-1} = \frac{\min\{\mathcal{Q}_{i-1}\} + \max\{\mathcal{Q}_{i-1}\}}{2}$$

due to Lemma 2.11 and the definition of q_{2i-1}. No matter whether $r_i = 0$ or $r_i = 1$, in both cases s_{r_i} has to travel distance $1/2 \cdot 2^{n-i+1} = 2^{n-i}$ in round $2i-1$ due to Lemma 2.14, for $1 \leq i \leq n$. The total cost of algorithm $\mathcal{A}_r$ is

$$\mathrm{cost}(\mathcal{A}_r(I_r)) = \sum_{i=1}^{n} \frac{\ell}{2^i} = \ell\left(1 - \frac{1}{2^n}\right) = 2^n - 1.$$ □

We now know what cost is incurred by $\mathcal{A}_r$ when it gets the request sequence I_r as its input. Next we want to show that this algorithm is optimal on I_r. Therefore, let us consider an arbitrary but fixed algorithm $\mathcal{C}$ that sends the inappropriate server s_{1-r_i} instead of s_{r_i} to satisfy request q_{2i-1} in some round $2i-1$. We prove that each such algorithm $\mathcal{C}$ has a larger cost than $\mathcal{A}_r$ and therefore show that $\mathcal{A}_r$ is indeed optimal and unique.

Lemma 2.16. *Any algorithm $\mathcal{C}$ that sends the inappropriate server in round $2i-1$ to satisfy the first request q_{2i-1} of gadget $\mathcal{Q}_i$ causes a cost of at least 2^{n-i+1} on gadget $\mathcal{Q}_i$ in total.*

Proof. For some arbitrary but fixed i with $1 \leq i \leq n$, let $\mathcal{Q}_i$ be a type-j-gadget, and let $\mathcal{C}$ send server s_{1-j} to the odd request q_{2i-1}. Let us consider the situation at the beginning of round $2i-1$. According to Observation 2.9, there have been no requests between the positions q_{2i-3} and q_{2i-2} so far. Therefore, due to Lemma 2.3, no server can be placed anywhere strictly in between $\min\{\mathcal{Q}_{i-1}\}$ and $\max\{\mathcal{Q}_{i-1}\}$. Hence, s_0 and s_1 are currently at positions $\kappa_{2i-1}(0) \leq \min\{\mathcal{Q}_{i-1}\}$ and $\kappa_{2i-1}(1) \geq \max\{\mathcal{Q}_{i-1}\}$. The distance between those two positions is at least 2^{n-i+1} due to Lemma 2.14.

Due to our assumption, $\mathcal{C}$ sends the inappropriate server, namely s_{1-j}, to the requested vertex $q_{2i-1} = (\min\{\mathcal{Q}_{i-1}\} + \max\{\mathcal{Q}_{i-1}\})/2$. Therefore, s_{1-j} has to traverse a distance of at least 2^{n-i} in this round. After that, the configuration of $\mathcal{C}$ is

$$\kappa_{2i} = \begin{cases} \left(\min\{\mathcal{Q}_{i-1}\}, \frac{\min\{\mathcal{Q}_{i-1}\}+\max\{\mathcal{Q}_{i-1}\}}{2}\right) & \text{if } \mathcal{Q}_i \text{ is a type-0-gadget,} \\ \left(\frac{\min\{\mathcal{Q}_{i-1}\}+\max\{\mathcal{Q}_{i-1}\}}{2}, \max\{\mathcal{Q}_{i-1}\}\right) & \text{if } \mathcal{Q}_i \text{ is a type-1-gadget.} \end{cases}$$

However, the request in round $2i$ arrives at position

$$q_{2i} = \begin{cases} \max\{\mathcal{Q}_{i-1}\} & \text{if } \mathcal{Q}_i \text{ is a type-0-gadget,} \\ \min\{\mathcal{Q}_{i-1}\} & \text{if } \mathcal{Q}_i \text{ is a type-1-gadget.} \end{cases}$$

Due to Lemma 2.3, this request is served by the closest server, which is in both cases s_{1-j}. Hence, s_{1-j} traverses 2^{n-i} edges in round $2i$, which corresponds to the cost incurred by $\mathcal{C}$ in this round.

All in all, $\mathcal{C}$ incurs a cost of at least $2 \cdot 2^{n-i} = 2^{n-i+1}$ in rounds $2i-1$ and $2i$ and therefore on gadget $\mathcal{Q}_i$ in total. □

Corollary 2.17. *Sending the inappropriate server instead of the appropriate one to satisfy the first request q_{2i-1} of some gadget $\mathcal{Q}_i$, for some i with $1 \le i \le n$, increases the cost for gadget $\mathcal{Q}_i$ by $2^{n-i+1} - 2^{n-i} = 2^{n-i}$ compared to $\mathcal{A}_r$.*

Proof. This follows directly from Lemma 2.16. □

Lemma 2.18. *Any algorithm that sends the appropriate server to satisfy the first request of at most αn gadgets $\mathcal{Q}_i$, has a cost of at least $2^n + 2^{(1-\alpha)n} - 2$.*

Proof. Consider an algorithm $\mathcal{A}$ that sends the appropriate server to at most αn first requests q_{2i-1} of gadgets $\mathcal{Q}_i$ (and therefore the inappropriate server to at least $(1-\alpha)n$ such requests). The additional cost compared to $\mathcal{A}_r$ incurred by sending the inappropriate server to q_{2i-1} decreases with increasing i due to Corollary 2.17. Hence, we can assume in $\mathcal{A}$'s favor that it satisfies the first requests of the last $(1-\alpha)n$ gadgets with the wrong server. Still, the cost of $\mathcal{A}$ on I_r is

$$\begin{aligned} \text{cost}(\mathcal{A}(I_r)) &\ge \text{cost}(\mathcal{A}_r(I_r)) + \sum_{i=n-(1-\alpha)n+1}^{n} 2^{n-i} && (2.4) \\ &= 2^n - 1 + \sum_{i=0}^{(1-\alpha)n-1} 2^i && (2.5) \\ &= 2^n - 1 + 2^{(1-\alpha)n} - 1 \\ &= 2^n + 2^{(1-\alpha)n} - 2, \end{aligned}$$

where (2.4) holds due to Corollary 2.17 and (2.5) due to Lemma 2.15. □

Finally, this leads to the conclusion that serving each type-i-request with its appropriate server s_i is indeed optimal.

Corollary 2.19. *Algorithm* $\mathcal{A}_r$ *is optimal on any instance* I_r *corresponding to a bit string* r.

Proof. This follows directly from Corollary 2.17. □

Similarly as before, we now want to complete the reduction by showing that an algorithm for 2-PATHSERVER can be used to construct an algorithm for the bit string guessing problem. The procedure is very similar to the one presented in Section 2.2.

The 2-GUESS-algorithm $\mathcal{B}$ gets the instance $I_{\mathcal{B}} = (n, r_1, \dots, r_n)$ as its input, and for each i with $0 \le i \le n-1$, it has to respond to each request sent in round i with its guess g_{i+1} for bit r_{i+1}. Now, during its execution, $\mathcal{B}$ generates the input $I_{\mathcal{A}}$ for 2-PATHSERVER in an online manner, simulates $\mathcal{A}$ on $I_r = (q_0, q_1, \dots, q_{2n})$, and bases its own output on the output of $\mathcal{A}$. Again, q_0 is the special query only consisting of the path length and the starting configuration, to which $\mathcal{A}$ must respond with $y_0 = \varepsilon$. In contrast to the procedure from Section 2.2, the algorithm $\mathcal{B}$ now simulates two time steps of $\mathcal{A}$ in each round, except for round n, in which only the single remaining time step of $\mathcal{A}$ is simulated. Hence, in round i with $0 \le i \le n-1$, the algorithm $\mathcal{B}$ sends requests q_{2i} and q_{2i+1} to $\mathcal{A}$ and adopts $\mathcal{A}$'s output y_{2i+1} to the first request of gadget $\mathcal{Q}_{i+1}$ as its guess g_{i+1}, and ignores the output of $\mathcal{A}$ in all even rounds. As before, it also constructs a dummy advice tape as a copy of the oracle's advice tape, from which $\mathcal{A}$ reads all of its advice bits, if any at all.

Lemma 2.20. *If* $\mathcal{A}$ *sends the appropriate server to satisfy the first request* q_{2i+1} *of gadget* $\mathcal{Q}_{i+1}$*, then* $\mathcal{B}$ *guesses the bit* r_{i+1} *correctly, for* $0 \le i \le n-1$.

Proof. Let $\mathcal{Q}_{i+1}$ be a type-j-gadget. Then, due to the construction of the request sequence I_r, it must hold that $r_{i+1} = j$.

On the other hand, the output of $\mathcal{A}$ in each round consists of the index of the server sent to satisfy the current request. If $\mathcal{A}$ sends s_j to request q_{2i+1}, the output of $\mathcal{A}$ in round $2i+1$ is therefore $y_{2i+1} = j$. As we have defined above, $\mathcal{B}$'s guess for the bit r_{i+1} is $g_{i+1} \coloneqq y_{2i+1}$, for $0 \le i \le n-1$, and consequently, we have $g_{i+1} = j$. Thus, $g_{i+1} = r_{i+1}$, and thus $\mathcal{B}$'s guess is obviously correct. □

For the following theorem, recall once more that η is used to denote the binary entropy function. The theorem gives us a lower bound on the number of advice bits necessary for online algorithms that achieve near-optimal competitive ratios; more precisely, competitive ratios of $c < 1 + (2^{(1-\alpha)n} - 1)/(2^n - 1)$, which tends to $1 + 1/2^{\alpha n}$ for growing n.

Theorem 2.21. *For any* α *with* $1/2 \le \alpha < 1$, *any online algorithm* $\mathcal{A}$ *for* 2-PATHSERVER *with a strict competitive ratio of* $c < 1 + (2^{(1-\alpha)n} - 1)/(2^n - 1)$ *has to read* $b \ge \big(1 - \eta(\alpha)\big)n$ *advice bits on instances of length* $2n + 1$, *or, rewritten,* $b \ge \big(1 - \eta(\alpha)\big)/2 \cdot (n' - 1)$ *advice bits on instances of length* n'.

Proof. For the sake of contradiction, assume that there exists an online algorithm $\mathcal{A}$ with advice for 2-PATHSERVER that reads less than $\big(1 - \eta(\alpha)\big)n$ advice bits and that has a strict competitive ratio of $c < 1 + (2^{(1-\alpha)n} - 1)/(2^n - 1)$. We show that, by using the construction given above, we can use $\mathcal{A}$ to obtain an algorithm $\mathcal{B}$ for 2-GUESS that reads the same number of advice bits as $\mathcal{A}$ and that guesses more than αn bits correctly. As such an algorithm does not exist due to Böckenhauer et al. [BHK+14], our assumption leads to a contradiction.

According to our initial assumption, $\mathcal{A}$ has a strict competitive ratio of $c < 1 + \big(2^{(1-\alpha)n} - 1\big)/(2^n - 1)$. Thus, for every instance I_r for 2-PATHSERVER corresponding to an input string r for 2-GUESS, the definition of the strict competitive ratio yields

$$\begin{aligned} \mathrm{cost}(\mathcal{A}(I_r)) &< \mathrm{cost}(\mathcal{A}_r(I_r))\left(1 + \frac{2^{(1-\alpha)n} - 1}{2^n - 1}\right) \\ &= (2^n - 1)\left(1 + \frac{2^{(1-\alpha)n} - 1}{2^n - 1}\right) \qquad (2.6) \\ &= 2^n + 2^{(1-\alpha)n} - 2, \end{aligned}$$

where (2.6) holds due to Lemma 2.15.

According to Lemma 2.18, $\mathcal{A}$ must hence send the appropriate server to satisfy the first request of more than αn gadgets. Then, due to Lemma 2.20, the algorithm $\mathcal{B}$ constructed from $\mathcal{A}$ guesses more than αn bits correctly, while reading as many advice bits as $\mathcal{A}$, namely $b < \big(1 - \eta(\alpha)\big)n$. This is a contradiction to Fact 1.5 on page 14, and thus, our initial assumption must have been false. □

3 Disjoint Path Allocation

This chapter addresses the *disjoint path allocation problem* on a path, from now on also called PATHDPA, which is the following online maximization problem. Given is a path P and a sequence of requests, each of them being a subpath of P. Two requests are said to be *intersecting* if they have a common edge. An algorithm for the disjoint path allocation problem must decide for each request if it *admits* or *rejects* it. The goal of such an algorithm is to admit as many requests as possible, such that no two admitted requests intersect. The requests are presented to the algorithm sequentially, and the decision whether to admit a given request or not must be made instantaneously, before the next request arrives. Once a request is admitted, it cannot be preempted later.

The disjoint path allocation problem is, in some sense, a simplified version of the *call admission problem*. In this much more general scenario, the requests do not necessarily appear on a path, but in a general weighted graph, where the edge weights indicate the capacity of the edges, and the requests (in this scenario called *calls*) can have different bandwidths, durations, and profits. For further information on the call admission problem, we refer to Chapter 13 from the textbook of Borodin and El-Yaniv [BEY98], and to Section 13.5 in the same book for the disjoint path allocation problem. The latter is a special case of the call admission problem in which all edges have capacity 1 and all calls have bandwidth 1, a profit of 1, and unlimited duration. PATHDPA, in turn, is a special case of the disjoint path allocation problem in which the underlying graph is a path.

When analyzing the advice complexity of the disjoint path allocation problem, it is possible to measure in terms of two different parameters, namely the length of the path or the number of requests. Most classical results, analyzing online algorithms for the disjoint path allocation problem without advice, measure in the path length. The first paper concerning algorithms with advice for the disjoint path allocation problem deals with the special case PATHDPA and measures the quality of an algorithm in the number of requests (Böckenhauer et al. [BKK+09]), whereas in a later paper by Barhum et al., it is measured in the path length ℓ [BBF+14]. Among other things, the former paper establishes a lower bound of roughly $n/(2c)$ advice bits to obtain a strictly c-competitive solution. The latter paper contains several upper and lower bounds for different ranges of the competitive ratio or the number of advice bits used, including the following. Without advice, there is an algorithm with a competitive ratio of $2\sqrt{\ell}$, and there is an almost matching lower bound on the competitive ratio of $2\sqrt{\ell}-4$. Moreover, Barhum et al. show that linear advice is sufficient to achieve a constant competitive ratio by giving several upper bounds. They also show that, to solve the disjoint path allocation problem optimally, the number of necessary and sufficient advice bits is $\ell - 1$. Another result presented by Barhum et al. is that there is an algorithm with a competitive ratio of $(2^b+1)(\ell^{1/(2^b+1)}+2)-4$ when the number b of advice bits read by the algorithm is in $O(\log\log\ell)$. Plugging in $b = \log\log(\ell/2)$, for example, this proves the existence of an algorithm with a competitive ratio of $4\log\ell - 4$, reading only $\log\log(\ell/2)$ advice bits. Again, they provide an almost matching lower bound of $(2^b+1)(\ell^{1/(2^b+1)}-2^{b-1})-3\cdot 2^b$ to the general upper bound. However, this lower bound is only valid for very small values of b, as this expression is already negative for b in the order of $\Theta(\log\log\ell)$ and therefore does not yield any meaningful result in this case. The result by Böckenhauer et al. mentioned above can be translated into a bound measured in ℓ, yielding a lower bound of roughly $(\log\ell)/c$ advice bits to achieve strict c-competitiveness. So far, no lower bound on the competitive ratio is known if the number of advice bits read is in the order of $\omega(\log\ell)$, except for the result for optimality mentioned above.

In this chapter, we provide a lower bound on the number of advice bits necessary to achieve competitive ratios from constant up to approximately $(\log\ell)/2$. We follow the tradition of the classical results on the disjoint path allocation problem and measure the competitive ratio of the presented algorithms in the length of the given path.

In Section 3.1, we consider how to obtain the desired lower bound with the help of a reduction from the bit string guessing problem. We actually prove with a rather simple reduction that a linear number of advice bits is necessary to achieve a certain constant competitive ratio. However, we will see that, in this case, the straight forward application of the reduction method does not provide any results for competitive ratios in the order of $\Theta(\log\ell)$, but only for a rather small range of constant competitive ratios. We will also see that our idea cannot be adapted easily such that a reduction from 2-GUESS would yield the desired result, pointing out the limitations of the reduction method. In Section 3.2, we then adapt our idea from Section 3.1, obtaining a lower bound for PATHDPA for a wide range of competitive ratios without applying the reduction method. The resulting bound is very general, giving a lower bound for competitive ratios from constant up to approximately $(\log \ell)/2$, and from this, we derive more concrete results. For example, we extrapolate the fact that to achieve a competitive ratio of $c \leq 1/2 \cdot (\log \ell)/(\log\log \ell)^{1/4}$, any online algorithm needs to read at least $\Omega(\ell/(4^c c^4))$ advice bits. Additionally, we can derive that to obtain a competitive ratio of $(\delta/2)\log \ell$, any algorithm needs at least $\omega(\ell^{1-\varepsilon})$ advice bits, for any two constants δ and ε with $0 < \delta < \varepsilon < 1$.

To give an overview on the significance of our results, we have a closer look at the results by Barhum et al. [BBF+14] we mentioned above. As we have seen, by augmenting an online algorithm for PATHDPA with very few advice bits, namely at most $\log\log(\ell/2)$, the competitive ratio can quickly be decreased from $2\sqrt{\ell}$ to $4\log \ell - 4$. However, to decrease the competitive ratio by only another constant factor to $(\delta/2)\log \ell$, at least $\omega(\ell^{1-\varepsilon})$ advice bits are necessary. Hence, a huge range of advice bits, namely those from $\log\log(\ell/2) + 1$ to any number in $O(\ell^{1-\varepsilon})$, basically does not help at all, which is a very sharp threshold that is worth remarking.

Before we start, we give the formal definition of the special case PATHDPA of the disjoint path allocation problem that we consider in this chapter.

Definition 3.1. *The* disjoint path allocation problem on a path (PATHDPA) *is the following maximization problem on a path $P = (0, \dots, \ell)$. In round 0, the number ℓ is revealed. In every successive round i, for $1 \leq i \leq n$, a request is given, represented by a subpath of P. Any online algorithm for* PATHDPA *has to admit or reject each such request immediately, before the next request arrives, without having the opportunity to revoke its decision later. The goal is to admit as many pairwise edge-disjoint requests as possible. The* gain *of an algorithm is the number of admitted requests.*

3.1 Reduction from 2-GUESS

Let us consider the following simple idea for a reduction from the bit string guessing problem to the disjoint path allocation problem. As we already did in Sections 2.2 and 2.3, we present a method to transform any input $I_{\mathcal{B},r}$ for 2-GUESS into an instance $I_{\mathcal{A},r}$ for PATHDPA on which the algorithm $\mathcal{A}$ is simulated and describe how to generate the output of $\mathcal{B}$ from the output of $\mathcal{A}$ on $I_{\mathcal{A},r}$. Let $I_{\mathcal{B},r} = (n, r_1, \dots, r_n)$ for a given bit string $r = r_1 \dots r_n$, and let $\zeta(r)$ be the number of zeros in r. We generate the following input $I_{\mathcal{A},r} = (q_0, \dots, q_{n+2\,\zeta(r)})$ for PATHDPA according to Definition 3.1 (similar to a construction by Komm [Kom12]). The first query, $q_0 := 2n = \ell$, is the length of the path, i. e., the number of edges of P. Then the requests q_i, for $1 \leq i \leq n + 2\,\zeta(r)$, are presented in two subsequent phases. In

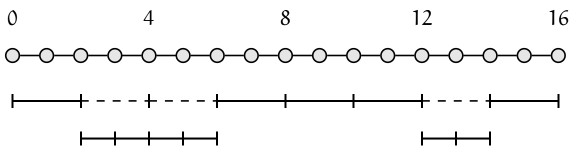


Figure 3.1. Instance for PathDPA constructed from the string $r = 10011101$ of length 8 containing 3 zeros.

the first one, exactly n edge-disjoint requests of length 2 are presented, with q_i starting at vertex $2i-2$ and ending at $2i$, for $1 \leq i \leq n$. In the second phase, the remaining $2\,\zeta(r)$ requests are presented as follows. For every i with $1 \leq i \leq n$, if and only if $r_i = 0$, there appear two requests of length 1 each intersecting with q_i, one starting at position $2i-2$, the other one at position $2i-1$. A small example is shown in Figure 3.1.

The 2-Guess algorithm $\mathcal{B}$ generates this input $I_{\mathcal{A},r}$ for $\mathcal{A}$ on the fly and can therefore simulate the PathDPA algorithm $\mathcal{A}$ on $I_{\mathcal{A},r}$. To do so, it sends the requests to $\mathcal{A}$ as follows. In round 0, when $\mathcal{B}$ is asked to guess the bit r_1, it simulates two time steps of $\mathcal{A}$, sending the requests q_0 (the special one only consisting of the length of the path) and q_1 to it. In every subsequent round $i-1$ with $2 \leq i \leq n$, when $\mathcal{B}$ is asked to guess the bit r_i, it simulates another time step of $\mathcal{A}$, feeding it the request q_i. To every request q_i with $1 \leq i \leq n$, the algorithm $\mathcal{A}$ responds with its output $y_i = 1$ if it accepts q_i and $y_i = 0$ if it rejects it. The algorithm $\mathcal{B}$ adopts $\mathcal{A}$'s output y_i as its guess for the bit r_i, i.e., $g_i := y_i$. In round n, the last one, $\mathcal{B}$ simulates all remaining time steps of $\mathcal{A}$, sending all requests of phase 2. Any feedback given to $\mathcal{B}$ by the adversary concerning the correctness of the guessed bit is passed on by $\mathcal{B}$ to $\mathcal{A}$. Furthermore, should $\mathcal{A}$ at any point want to read some advice bits, $\mathcal{B}$ reads the according number of bits instead and writes them onto a dummy advice tape that $\mathcal{A}$ can access, as we already explained in Section 2.2.

Let $\mathcal{I} = \{I_{\mathcal{A},r} \mid r \in \{0,1\}^n\}$ be the set containing every instance $I_{\mathcal{A},r}$ constructed in the way described above from a bit string r, and let Opt be an algorithm that is optimal on every such instance $I_{\mathcal{A},r} \in \mathcal{I}$. Obviously, Opt admits a request q_i from phase 1 if and only if no requests intersecting with q_i appear in phase 2, i.e., if and only if $r_i = 1$. Thus, the optimal solution for $I_{\mathcal{A},r}$ has a gain of $\mathrm{gain}(\mathrm{Opt}(I_{\mathcal{A},r})) = n + \zeta(r)$, and therefore, for all $I_{\mathcal{A},r}$,

$$n \leq \mathrm{gain}(\mathrm{Opt}(I_{\mathcal{A},r})) \leq 2n. \tag{3.1}$$

From now on, we will say that the algorithm $\mathcal{A}$ *makes the correct decision* for q_i if either $\mathcal{A}$ admits q_i and $r_i = 1$ or if it rejects q_i and $r_i = 0$, for $1 \leq i \leq n$. Otherwise, we say that $\mathcal{A}$ *makes the wrong decision* for q_i. Without loss of generality, we can assume that any algorithm for PathDPA, given such an PathDPA instance $I_{\mathcal{A},r}$

constructed from a string r as its input, admits all requests from phase 2 that do not intersect with any admitted requests from phase 1.

For any such instance $I_{\mathcal{A},r} \in \mathcal{I}$ and any i with $1 \leq i \leq n$, if $\mathcal{A}$ makes the wrong decision for q_i, then $\mathcal{A}$'s gain decreases by 1 compared to the gain of Opt on $I_{\mathcal{A},r}$. If $\mathcal{A}$ makes the correct decision for at most αn requests q_i from phase 1 and thus the wrong decision for at least $(1-\alpha)n$ such requests, then $\mathcal{A}$'s gain is

$$\mathrm{gain}(\mathcal{A}(I_{\mathcal{A},r})) \leq \mathrm{gain}(\mathrm{Opt}(I_{\mathcal{A},r})) - (1-\alpha)n, \tag{3.2}$$

where n is, as we recall again, the length of the string r and not the length of the PATHDPA instance.

Now let us show that the existence of an algorithm $\mathcal{A}$ for PATHDPA with a good competitive ratio implies the existence of an algorithm for 2-GUESS that guesses many bits correctly. Hence, let us assume that there is an algorithm $\mathcal{A}$ with a strict competitive ratio of $c < 1 + (1-\alpha)/(1+\alpha)$. Then,

$$c < 1 + \frac{(1-\alpha)n}{(1+\alpha)n} = 1 + \frac{(1-\alpha)n}{2n-(1-\alpha)n} \leq 1 + \frac{(1-\alpha)n}{\mathrm{gain}(\mathrm{Opt}(I_{\mathcal{A},r})) - (1-\alpha)n},$$

where the last inequality holds due to (3.1). This implies

$$c < \frac{\mathrm{gain}(\mathrm{Opt}(I_{\mathcal{A},r})) - (1-\alpha)n + (1-\alpha)n}{\mathrm{gain}(\mathrm{Opt}(I_{\mathcal{A},r})) - (1-\alpha)n} = \frac{\mathrm{gain}(\mathrm{Opt}(I_{\mathcal{A},r}))}{\mathrm{gain}(\mathrm{Opt}(I_{\mathcal{A},r})) - (1-\alpha)n},$$

and thus

$$\frac{\mathrm{gain}(\mathrm{Opt}(I_{\mathcal{A},r}))}{\mathrm{gain}(\mathcal{A}(I_{\mathcal{A},r}))} < \frac{\mathrm{gain}(\mathrm{Opt}(I_{\mathcal{A},r}))}{\mathrm{gain}(\mathrm{Opt}(I_{\mathcal{A},r})) - (1-\alpha)n}.$$

As a consequence, we conclude that the gain of $\mathcal{A}$ must be $\mathrm{gain}(\mathcal{A}(I_{\mathcal{A},r})) > \mathrm{gain}(\mathrm{Opt}(I_{\mathcal{A},r})) - (1-\alpha)n$. Then, according to (3.2), $\mathcal{A}$ must make the correct decision for more than αn requests q_i from phase 1. This implies that there are more than αn requests q_i such that

$$\begin{aligned}
& (\mathcal{A} \text{ admits } q_i \text{ and } r_i = 1) \quad \text{or} \quad (\mathcal{A} \text{ rejects } q_i \text{ and } r_i = 0) \\
\iff \quad & (y_i = 1 \text{ and } r_i = 1) \quad \text{or} \quad (y_i = 0 \text{ and } r_i = 0) \\
\iff \quad & y_i = r_i \\
\iff \quad & g_i = r_i .
\end{aligned}$$

Therefore, the 2-GUESS algorithm $\mathcal{B}$ constructed from $\mathcal{A}$ guesses more than αn bits correctly. Since $\mathcal{B}$ reads the same number of advice bits as $\mathcal{A}$, the following holds. If there is an algorithm for PATHDPA with a strict competitive ratio of $c < 1 + (1-\alpha)/(1+\alpha)$, for $1/2 \leq \alpha < 1$, reading less than $(1-\eta(\alpha))n$ advice bits on all instances from $\mathcal{I}$, then there is an algorithm for 2-GUESS reading less than $(1-\eta(\alpha))n$ advice bits, guessing more than αn bits correctly on any input string of length n.

Theorem 3.2. *There is no online algorithm for* PATHDPA *with a strict competitive ratio of* $c < 1 + (1-\alpha)/(1+\alpha)$, *for* $1/2 \leq \alpha < 1$, *reading less than* $(1-\eta(\alpha))\ell/2$ *advice bits on instances with path length* ℓ.

Proof. This follows directly from our considerations, the choice $\ell := 2n$, and the fact that any algorithm for 2-GUESS guessing more than αn bits correctly has to read at least $(1-\eta(\alpha))n$ advice bits due to Fact 1.5. □

Corollary 3.3. *Any online algorithm for* PATHDPA *with a strict competitive ratio of* $c < 1 + \delta$, *for* $0 < \delta < 1/3$, *needs to read at least* $\Theta(\ell)$ *advice bits on instances with path length* ℓ.

Proof. Plugging $\alpha := 1/2 + \varepsilon$ into Theorem 3.2 yields that any algorithm with a strict competitive ratio of

$$c < 1 + \frac{1/2 - \varepsilon}{3/2 + \varepsilon}$$

needs to read at least $(1-\eta(1/2+\varepsilon))\ell/2$ advice bits. Solving $1+(1/2-\varepsilon)/(3/2+\varepsilon) = 1 + \delta$ for ε, we obtain

$$\frac{1/2 - \varepsilon}{3/2 + \varepsilon} = \delta \iff \tfrac{1}{2} - \varepsilon = \delta\big(\tfrac{3}{2} + \varepsilon\big) \iff \tfrac{1}{2} - \tfrac{3}{2} \cdot \delta = \delta\varepsilon + \varepsilon \iff \varepsilon = \frac{1 - 3\delta}{2(\delta + 1)}.$$

Thus, $0 < \varepsilon < 1/2$, and $(1-\eta(1/2+\varepsilon))\ell/2 \in \Theta(\ell)$. □

By this simple reduction from 2-GUESS, we have shown that to achieve any strict competitive ratio c with $1 < c < 4/3$ for PATHDPA, a linear (in the length of the path) number of advice bits is necessary. This complements the upper bounds given by Barhum et al. [BBF+14].

However, one drawback of this reduction is the fact that the range of competitive ratios for which we obtain a lower bound on the necessary advice bits is rather small. More desirable would be a statement for a much wider range for c, maybe even for ratios $c \in \omega(1)$. A straight forward idea how to achieve this could be the following. In the reduction given above, the requests are presented in two phases, and the algorithm for PATHDPA virtually has to guess for each request given in phase 1 whether two requests will appear within that request in phase 2 or not. For every request in phase 1 for which the algorithm makes a wrong guess, its gain decreases by 1. Now, we could extend this idea by presenting requests with decreasing lengths in more than two, say h phases, such that each request in phase 1 has length 2^{h-1}, and within every request in phase i, there might either appear two requests of halved length in phase $i+1$ or none at all. This way, if an algorithm admits a request in phase 1, and within this request, 2^{h-1} requests appear in phase h, the gain of the algorithm will decrease by $2^{h-1} - 1$. Although this idea looks promising, it presents us with several problems. If the algorithm does not admit a request from phase 1 and no further requests appear within this request in later phases, the gain only decreases by 1. Hence, making a wrong decision in phase 1 leads to different penalties, depending on the decision the algorithm made. Another problem is that, if the algorithm makes a wrong decision in phase 2, for example, the gain cannot be decreased by $2^{h-1} - 1$ any more by presenting corresponding requests in phase h, but only by at most $2^{h-2} - 1$. This is where we are pushed to the envelope of string guessing reductions. To reflect this behavior of the gain function of these PATHDPA instances, the cost function of the string guessing problem would have to be asymmetric and somehow vary over time. Hence, unfortunately, we cannot project such behavior to the string guessing problem. Nevertheless, the idea described above can actually be used to obtain a better lower bound for the PATHDPA problem. In what follows, we will show how this is done, without using a string guessing reduction.

3.2 A Lower Bound Without Using a Reduction

In this section, we want to extend and adapt the idea from Section 3.1 to give a lower bound on the number of advice bits necessary to achieve small competitive ratios. For our calculations, we will need Bernoulli's inequality, which states the following (see, for example, Carothers [Car00]).

Fact 3.4. *For every real number $x \in \mathbb{R}^{\geq -1}$ and every natural number $n \in \mathbb{N}^{\geq 0}$,*

$$(1+x)^n \geq 1 + nx. \qquad \square$$

We will give a lower bound on the number of advice bits necessary to achieve a competitive ratio of $c := c(\ell)$ that is a slowly growing function in ℓ. Therefore, we construct a set $\mathcal{I}$ of instances such that any deterministic algorithm can achieve the competitive ratio c only on a small fraction of the instances from $\mathcal{I}$. Then, following Observation 1.2 (page 13), any algorithm with advice needs to read many advice bits to achieve a competitive ratio of at most c on all instances from $\mathcal{I}$.

The following argumentation will involve a random variable with hypergeometric distribution. Therefore, we will now establish a result that follows from a well-known bound for the tail of the hypergeometric distribution. First, let us recall that a random variable with hypergeometric distribution with parameters M, N, and n counts the number of black balls drawn from an urn containing N balls, out of which exactly M are black, when drawing n balls uniformly at random without replacement (see, for example, Rice [Ric07]). The following bound was established by Chvátal [Chv79].

Fact 3.5. *Consider a discrete random variable X with hypergeometric distribution with parameters M, N, and n, i. e.,*

$$\Pr(X = i) = \frac{\binom{M}{i}\binom{N-M}{n-i}}{\binom{N}{n}},$$

for every i with $0 \leq i \leq M$. Then,

$$\mathbb{E}(X) = n \cdot \frac{M}{N},$$

and

$$P(X \leq \mathbb{E}(X) - tn) \leq e^{-2t^2 n},$$

for any $t \geq 0$, where e is Euler's number. □

We have to adapt this result slightly for our purposes.

Corollary 3.6. *Let X be a discrete random variable with hypergeometric distribution with parameters M, N, and n, and let $t \geq 0$. Then, for every $M' \leq M$, we have*

$$\Pr\left(X \leq n \cdot \frac{M'}{N} - tn\right) \leq e^{-2t^2 n}.$$

Proof. Since $M' \leq M$,

$$\Pr\left(X \leq n \cdot \frac{M'}{N} - tn\right) \leq \Pr\left(X \leq n \cdot \frac{M}{N} - tn\right).$$

Applying Fact 3.5, we obtain that the expected value of X is $\mathbb{E}(X) = n \cdot M/N$, and thus

$$\Pr\left(X \leq n \cdot \frac{M'}{N} - tn\right) \leq \Pr(X \leq \mathbb{E}(X) - tn) \leq e^{-2t^2 n}.$$ □

Now let us describe how to construct the set $\mathcal{I}$ of instances for PATHDPA. For the sake of simplicity, let ℓ be a power of 2. Furthermore, let $h := h(\ell)$ be a parameter that depends on the length ℓ of the path with

$$h \in \mathbb{N}^{\geq 1} \quad \text{and} \quad h \leq \log(\ell) - 1. \tag{3.3}$$

Then the requests are presented to the algorithm in $h + 1$ phases. In each phase i with $1 \leq i \leq h + 1$, the algorithm is given exactly $\ell/2^h$ edge-disjoint requests of length 2^{h-i+1}. Hence, in the first phase, $\ell/2^h$ edge-disjoint subpaths of length 2^h are presented, whose concatenation forms the complete path P. Half of the requests from phase i with $1 \leq i \leq h$ are so-called *closed* requests, for which no intersecting

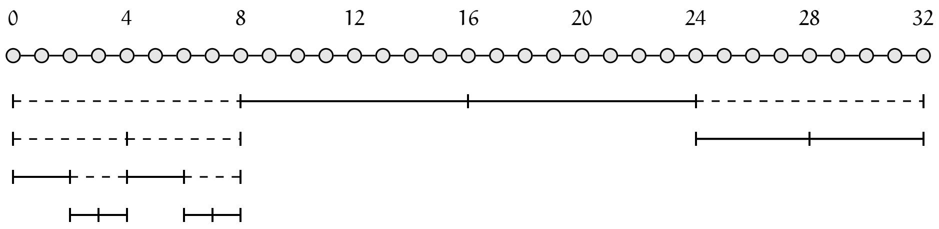


Figure 3.2. An example of an instance from the constructed instance set $\mathcal{I}$ for a path of length $\ell = 32$ and $h = 3$. There are four phases with $\ell/2^h = 4$ requests each. Open requests are depicted by dashed lines. All other requests are closed (and are therefore contained in the optimal solution).

requests will be presented anymore, and which are hence contained in the optimal solution. The other half of these requests are *open*, i. e., they are split into two edge-disjoint requests of length 2^{h-i} each, which are then presented in phase $i+1$. Finally, in phase $h+1$, the algorithm is given $\ell/2^h$ subpaths of length 1 each, which are all contained in the optimal solution. For an example, see Figure 3.2.

Observation 3.7. *There are*

$$\binom{\ell/2^h}{\ell/2^{h+1}}^h$$

instances in $\mathcal{I}$.

Proof. For every i with $1 \le i \le h$ and every possibility to divide the $\ell/2^h$ subpaths from phase i into $\ell/2^{h+1}$ open and $\ell/2^{h+1}$ closed ones, we construct one instance I and add it to $\mathcal{I}$. Obviously, there are

$$\binom{\ell/2^h}{\ell/2^{h+1}}$$

such possibilities for every phase i, which directly implies the statement. □

Observation 3.8. *The optimal solution on any instance I from $\mathcal{I}$ has a gain of*

$$\mathrm{gain}(\mathrm{Opt}(I)) = \frac{(h+2)\ell}{2^{h+1}}.$$

Proof. For every phase i with $1 \le i \le h$, there are $\ell/2^{h+1}$ requests that are contained in the optimal solution; in phase $h+1$, there are an additional $\ell/2^h$ such requests, which yields

$$h \cdot \frac{\ell}{2^{h+1}} + \frac{\ell}{2^h} = \frac{h\ell}{2^{h+1}} + \frac{2\ell}{2^{h+1}} = \frac{(h+2)\ell}{2^{h+1}}.$$ □

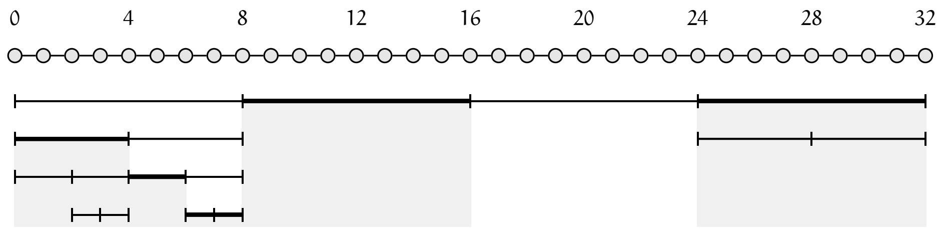


Figure 3.3. When given the instance from Figure 3.2 as an input, a deterministic algorithm might admit the requests as depicted in this picture. Admitted requests are marked by thick lines. By admitting a request, all requests that intersect with this request become blocked, which is depicted by areas shaded in gray.

Now consider a deterministic algorithm A. When admitting a request q_j in phase i, all subpaths intersecting with q_j become *blocked*. In every phase i, the algorithm can only admit requests that were not blocked in any phase preceding phase i. Figure 3.3 shows how an examplary algorithm A operates on the instance from Figure 3.2.

Let us introduce another parameter $f := f(c)$, such that $f > 0$. Both parameters f and h must be chosen according to the competitive ratio that an algorithm is supposed to achieve. In the remainder of this chapter, we prove the following as a main result. For any online algorithm for PATHDPA to achieve a competitive ratio of

$$c = \frac{h+2}{2 \cdot (1 + \frac{h}{f})},$$

it must read at least

$$b = \frac{\ell}{2^h \cdot f^2} \cdot \log e - \log h$$

advice bits. After we have proven this main result, we choose concrete values for f and h to obtain more tangible lower bounds.

We start our argumentation by making the following observation. The set $\mathcal{I}$ of instances can naturally be represented by a $\binom{\ell/2^h}{\ell/2^{h+1}}$-ary tree of depth h, as depicted in Figure 3.4. The root is on level 0 and corresponds to all instances from $\mathcal{I}$. There are $\binom{\ell/2^h}{\ell/2^{h+1}}$ instances on level 1, each of them representing all instances with the same particular set of open requests from phase 1. In general, any vertex on level i represents the set of all instances with the same particular sets of open requests from phases $1, \ldots, i$. For each vertex v, let us call the set of instances represented by v be called $\mathcal{I}_v$. Then for each vertex v on level i, the requests presented in the first $i+1$ phases are exactly the same in all instances from $\mathcal{I}_v$. Every leaf is located on level h and thus corresponds to a unique instance from $\mathcal{I}$. Furthermore, for each vertex v on level i with $0 \leq i \leq h$ and each

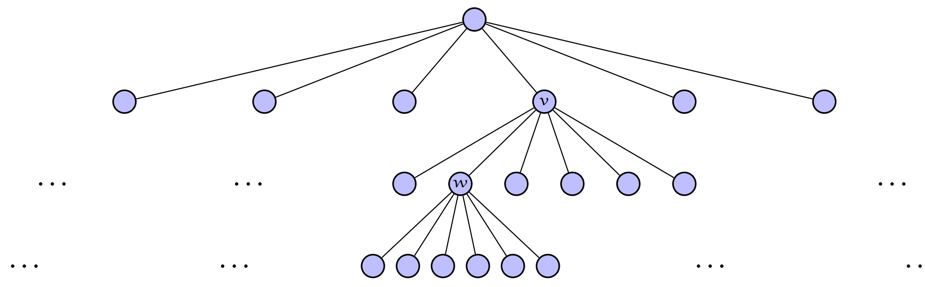


Figure 3.4. An example of an instance tree. Figures 3.2 and 3.3 are both placed in a scenario with a path of length 32 and $h + 1 = 4$ phases (hence, with $\ell/2^h = 4$ requests presented per phase). In this scenario, there are $\binom{4}{2} = 6$ possibilities to choose $\ell/2^{h+1} = 2$ out of the 4 requests to be open in every phase. Hence, every inner vertex of the corresponding instance tree has exactly 6 children (most of which are only indicated by dots for the sake of clear presentation). The root represents all instances, the vertex v represents all instances in which the same set of requests from phase 1 are open, and each leaf on level h represents all instances in which the same set of requests from phases $1, \ldots, h$ are open, hence, each leaf represents a single instance.

deterministic algorithm A, the following holds. Given any instance from $\mathcal{I}_v$ as its input, A is always in the same state at the beginning of phase $i + 1$, i. e., it has seen and admitted the same requests so far; see Figure 3.5.

From now on, consider A to be an arbitrary but fixed deterministic algorithm for PATHDPA. For a given vertex v on level i, let $\gamma^{(i)}$ be the gain of A on any instance from $\mathcal{I}_v$ during phase i, i. e., the number of accepted requests during this phase. Moreover, let $\hat{\gamma}^{(i)}$ be the gain of A during all phases up to and including phase i, hence, $\hat{\gamma}^{(i)} := \sum_{j=1}^{i} \gamma^{(j)}$. Then let us introduce the following notion of bad phases and vertices. We call a phase i *bad* for A if, at the beginning of this phase, at least

$$d_{i-1} := \hat{\gamma}^{(i-1)} - (i-1) \cdot \frac{\ell}{2^h \cdot f} \tag{3.4}$$

requests from phase i are already blocked. Furthermore, let us call a vertex v on level $i - 1$ *bad* for A if, when A is given any instance from $\mathcal{I}_v$ as its input, phase i is bad for A. Phases and vertices that are not bad for A are called *good* for A. When A is clear from the context, we may also call a phase or a vertex just *good* or *bad*, without mentioning A explicitly. Moreover, let us define the set of requests from phase i that are blocked at the beginning of phase $i + 1$ (including those that were admitted in phase i, which are blocking themselves) to be R_i.

Lemma 3.9. *If at least $d_i/2$ requests from R_i are open, then phase $i + 1$ is bad for A.*

Proof. Within each open request from phase i, two requests appear in phase $i+1$. If $d_i/2$ requests from R_i are open, then at least d_i requests are presented in phase $i+1$

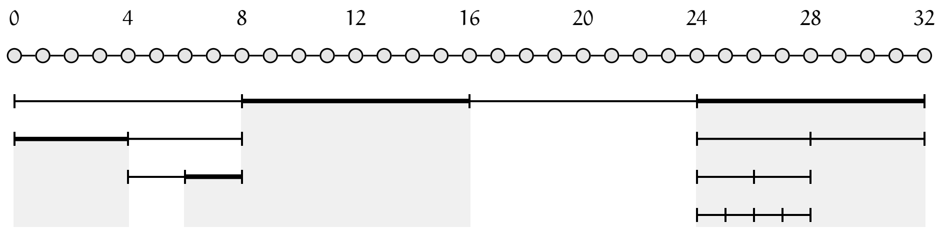


Figure 3.5. In the instance tree from Figure 3.4, on every level there must be one vertex that contains the instance from Figure 3.3. Without loss of generality, on level 1, let this vertex be v. Then, v also contains the instance depicted in this figure. In both instances, the same requests from phase 1 are open, and thus the requests presented in the first two phases are the same for *all* instances represented by v. Hence, at the beginning of and also throughout phase 2, each fixed deterministic algorithm A must be in the same state given any instance corresponding to v as its input, having seen and admitted the exact same requests so far.

that are already blocked at the beginning of phase $i+1$. This matches the definition of a bad phase for A. □

Lemma 3.10. *For any deterministic online algorithm A, the fraction of vertices on level i that are bad for A is at least*

$$\left(1 - e^{-\ell/(2^h \cdot f^2)}\right)^i.$$

Proof. We prove the claim by induction on i. On level 0, there is only one vertex, namely the root representing all instances from $\mathcal{I}$. Obviously, A did not admit any requests before phase 1, and thus $\hat{\gamma}^{(0)} = 0$. According to its definition, the root is bad if phase 1 is bad for A. This, in turn, is the case if at least 0 requests are blocked at the beginning of phase 1 when A is processing any instance; see (3.4). This is obviously true; therefore, the base case is covered.

Let us now assume that the claim holds for some level $i-1$. We will show that the claim then also holds for level i. From now on, let v be some bad vertex on level $i-1$. First, we prove that the fraction of bad vertices among the children of v is at least $1-e^{-\ell/(2^h \cdot f^2)}$. Since v is bad, phase i must be bad for A, given any instance from $\mathcal{I}_v$ as its input. Therefore, for each such instance, at least d_{i-1} requests from phase i are already blocked at the beginning of phase i. As A admits $\gamma^{(i)}$ further requests in this phase, at least $d_{i-1} + \gamma^{(i)}$ requests from phase i are blocked at the beginning of phase $i+1$, including the admitted requests from phase i. This set of requests corresponds to the set R_i defined earlier. From Lemma 3.9, we know that, if at least $d_i/2$ requests from R_i are open, then phase $i+1$ is bad for A, given an arbitrary instance from $\mathcal{I}_v$ as its input. Since every instance from $\mathcal{I}_w$ is also contained in $\mathcal{I}_v$, the same is true if A is given any instance from $\mathcal{I}_w$ as its input. Figure 3.6 gives an example.

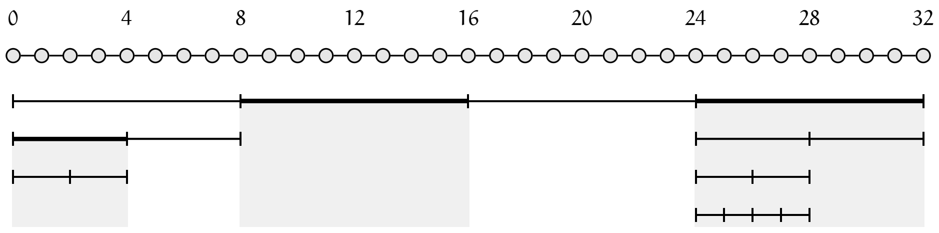


Figure 3.6. This instance, like those from Figures 3.3 and 3.5, corresponds to the vertex v from Figure 3.4. The deterministic algorithm A admits $\hat{\gamma}^{(1)} = 2$ requests in phase 1. Hence, $d_1 = \hat{\gamma}^{(1)} - 1 \cdot \ell/(2^h f) = 2 - 32/(2^3 f) = 2 - 4/f < 2$. Since 2 requests from phase 2 are already blocked at the beginning of this phase, v is a bad vertex. Without loss of generality, let w from Figure 3.4 be the vertex on level 2 containing the instance from this picture. The vertex w is bad if, out of all requests from phase 3, at least $d_2 = \hat{\gamma}^{(2)} - 2 \cdot 32/(2^3 f) = 3 - 8/f < 3$ are blocked at the beginning of phase 3. Hence, in this instance, out of the 3 requests from phase 2 that are blocked after phase 2, at least $d_2/2 < 1.5$ must be open. This is clearly the case, since 2 such requests are open; thus, w is bad.

Hence, a sufficient condition for w to be bad is that at least $d_i/2$ requests from R_i are open when giving A an instance from $\mathcal{I}_w$ as its input. Thus, we have the following scenario. There are $N := \ell/2^h$ requests in phase i, as in every phase. Out of these, $M \geq M' := d_{i-1} + \gamma^{(i)}$ are blocked at the beginning of phase $i+1$. The set of these requests is R_i. Each child w of v corresponds to the set of instances in which the same set of $n := \ell/2^{h+1}$ requests from phase i are open requests. We are interested in the fraction p of children of v that correspond to instances in which at least $d_i/2$ requests from R_i are open. This is equivalent to considering an urn containing N balls (= requests), out of which $M \geq M'$ are black (= in R_i), drawing n balls (= opening n requests) without replacement, and in this setting determining the probability that the number of black balls drawn (= open requests from R_i) is at least $d_i/2$.

Let X be a random variable that counts the number of open requests from R_i in this scenario. Then X has a hypergeometric distribution with parameters $M \geq d_{i-1} + \gamma^{(i)}$, $N = \ell/2^h$, and $n = \ell/2^{h+1}$, and we are interested in $\Pr(X \geq d_i/2)$. With

$$\begin{aligned}\frac{d_i}{2} &= \frac{1}{2}\left(\hat{\gamma}^{(i)} - i \cdot \frac{\ell}{2^h \cdot f}\right)\\ &= \frac{1}{2}\left(\hat{\gamma}^{(i-1)} + \gamma^{(i)} - (i-1) \cdot \frac{\ell}{2^h \cdot f} - \frac{\ell}{2^h \cdot f}\right)\\ &= \frac{d_{i-1}}{2} + \frac{\gamma^{(i)}}{2} - \frac{\ell}{2^{h+1} \cdot f},\end{aligned}$$

we obtain

$$\Pr\left(X \geq \frac{d_i}{2}\right) \geq \Pr\left(X > \frac{d_i}{2}\right)$$
$$= 1 - \Pr\left(X \leq \frac{d_i}{2}\right)$$
$$= 1 - \Pr\left(X \leq \frac{d_{i-1}}{2} + \frac{\gamma^{(i)}}{2} - \frac{\ell}{2^{h+1} \cdot f}\right). \tag{3.5}$$

Corollary 3.6 gives us a means to bound

$$\Pr\left(X \leq n \cdot \frac{M'}{N} - t \cdot n\right) = \Pr\left(X \leq \frac{\ell}{2^{h+1}} \cdot \frac{d_{i-1} + \gamma^{(i)}}{\frac{\ell}{2^h}} - t \cdot \frac{\ell}{2^{h+1}}\right)$$
$$= \Pr\left(X \leq \frac{d_{i-1} + \gamma^{(i)}}{2} - t \cdot \frac{\ell}{2^{h+1}}\right)$$

from above for any $t \geq 0$. Hence, choosing $t := 1/f$ yields

$$\Pr\left(X \leq \frac{d_{i-1} + \gamma^{(i)}}{2} - t \cdot \frac{\ell}{2^{h+1}}\right) = \Pr\left(X \leq \frac{d_{i-1}}{2} + \frac{\gamma^{(i)}}{2} - \frac{\ell}{2^{h+1} \cdot f}\right).$$

Then, according to Corollary 3.6,

$$\Pr\left(X \leq \frac{d_{i-1} + \gamma^{(i)}}{2} - \frac{\ell}{2^{h+1} f}\right) \leq e^{-2t^2 n} = e^{-\ell/(2^h f^2)}. \tag{3.6}$$

Finally, combining (3.5) and (3.6), we obtain

$$\Pr\left(X \geq \frac{d_i}{2}\right) \geq 1 - \Pr\left(X \leq \frac{d_{i-1}}{2} + \frac{\gamma^{(i)}}{2} - \frac{\ell}{2^{h+1} \cdot f}\right)$$
$$\geq 1 - e^{-\ell/(2^h f^2)}.$$

Hence, we have now shown that, for each bad vertex v on level $i - 1$, the fraction of bad vertices among its children is at least $1 - e^{-\ell/(2^h f^2)}$.

Now we are almost done. The only thing that remains to do is to exhibit a connection to the number of bad vertices on level i. All vertices on level $i - 1$ have the same number of children, and due to the induction hypothesis, for every bad vertex on level $i - 1$, a fraction of at least $1 - e^{-\ell/(2^h f^2)}$ of its children is bad. Hence, the fraction of bad vertices on level i is at least

$$\left(1 - e^{-\ell/(2^h f^2)}\right)^{i-1} \cdot \left(1 - e^{-\ell/(2^h f^2)}\right) = \left(1 - e^{-\ell/(2^h f^2)}\right)^i. \qquad \square$$

A direct consequence of this result that many vertices are bad is that many instances are bad for A.

Corollary 3.11. *For any deterministic online algorithm A, the fraction of instances in $\mathcal{I}$ that are bad for A is at least*

$$\left(1 - e^{-\ell/(2^h f^2)}\right)^h.$$

Proof. Every single instance corresponds to a leaf in the instance tree, and the leaves of the tree are located at level h. Plugging in the result of Lemma 3.10 proves the statement. □

We have now shown that there are many bad instances for a given deterministic algorithm A for PATHDPA. What we will show next is that the choice of the term "bad" was indeed justified for these instances, i. e., that A can indeed only admit few requests on any bad instance.

Lemma 3.12. *Let A be an arbitrary but fixed deterministic algorithm for* PATHDPA, *and let $I \in \mathcal{I}$ be a bad instance for A. Then, the gain of A on I is at most*

$$\operatorname{gain}(A(I)) \le \frac{\ell}{2^h}\left(1 + \frac{h}{f}\right).$$

Proof. According to the definition of bad vertices, an instance (corresponding to a vertex on level h of the instance tree) is bad if there are at least

$$d_h = \hat{\gamma}^{(h)} - h \cdot \frac{\ell}{2^h f}$$

requests from phase $h+1$ that are already blocked at the beginning of phase $h+1$. In this last phase, A is presented $\ell/2^h$ requests, and thus the number of requests A can admit in this phase is

$$\begin{aligned}
\gamma^{(h+1)} &\le \frac{\ell}{2^h} - \left(\hat{\gamma}^{(h)} - h \cdot \frac{\ell}{2^h f}\right) \\
&= \frac{\ell}{2^h} + \frac{h}{f} \cdot \frac{\ell}{2^h} - \hat{\gamma}^{(h)} \\
&= \frac{\ell}{2^h} \cdot \left(1 + \frac{h}{f}\right) - \hat{\gamma}^{(h)}.
\end{aligned}$$

For the number of admitted requests at the end of the computation, we obtain

$$\hat{\gamma}^{(h+1)} = \hat{\gamma}^{(h)} + \gamma^{(h+1)} \le \hat{\gamma}^{(h)} + \frac{\ell}{2^h} \cdot \left(1 + \frac{h}{f}\right) - \hat{\gamma}^{(h)} = \frac{\ell}{2^h} \cdot \left(1 + \frac{h}{f}\right).$$

This corresponds to the total gain of the algorithm on any bad instance I, i. e., $\operatorname{gain}(A(I))$. □

All in all, we have shown that, for any fixed deterministic algorithm A, there are many instances on which A has only a small gain. We now combine these results to obtain our main result.

Theorem 3.13. *Any online algorithm for* PATHDPA *with a strict competitive ratio of*

$$\frac{h+2}{2\cdot(1+\frac{h}{f})}$$

needs to read

$$b \geq \frac{\ell}{2^h f^2}\cdot \log e - \log h$$

advice bits.

Proof. Consider an arbitrary but fixed deterministic algorithm A for PATHDPA. According to Lemma 3.12 and Observation 3.8, the strict competitive ratio of A on an arbitrary bad instance I is

$$c = \frac{\mathrm{gain}(\mathrm{Opt}(I))}{\mathrm{gain}(A(I))} \geq \frac{\frac{(h+2)\ell}{2^{h+1}}}{\frac{\ell}{2^h}\cdot(1+\frac{h}{f})} = \frac{2^h}{2^{h+1}}\cdot\frac{h+2}{(1+\frac{h}{f})} = \frac{h+2}{2(1+\frac{h}{f})}.$$

Now consider an arbitrary online algorithm $\mathcal{A}$ with advice for PATHDPA reading $b := b(\ell)$ advice bits. We can interpret $\mathcal{A}$ in the usual way as a set of 2^b deterministic algorithms, $\mathcal{A} = \{A_1, \ldots, A_{2^b}\}$, as stated in Fact 1.1. From Corollary 3.11, we know that, for every such deterministic algorithm A_i, the fraction of good instances from $\mathcal{I}$, and hence the fraction of instances on which A_i has a competitive ratio of at most $(h+2)/(2(1+h/f))$, is at most

$$1-\left(1-\frac{1}{e^{\ell/(2^h f^2)}}\right)^h \leq \frac{h}{e^{\ell/(2^h f^2)}},$$

where we used Bernoulli's inequality (Fact 3.4), plugging in the values $n := h$ and $x := -1/e^{\ell/(2^h f^2)}$. Note that this is legitimate as long as $2^h > 0$ and $f > 0$, since then $\ell/(2^h f^2) \geq 0$ and hence $x \geq -1$.

Now we can apply the method we described in Observation 1.2. The number of deterministic algorithms necessary to guarantee a competitive ratio of at most $(h+2)/(2(1+h/f))$ for every instance from $\mathcal{I}$ is at least

$$\frac{e^{\ell/(2^h f^2)}}{h}.$$

To be able to distinguish this many different deterministic strategies, $\mathcal{A}$ has to read at least

$$\log\left(\frac{e^{\ell/(2^h f^2)}}{h}\right) = \frac{\ell}{2^h f^2}\cdot\log e - \log h$$

advice bits. □

Now, by choosing concrete values for h and f, we infer two lower bounds for different ranges of c.

Corollary 3.14. *For any $c = c(\ell)$ with $1 < c \leq 1/2 \cdot (\log \ell)/(\log\log \ell)^{1/4}$, any online algorithm for* PathDPA *that achieves a strict competitive ratio of c needs to read at least $\Omega\big(\ell/(4^c\, c^4)\big)$ advice bits.*

Proof. Let $h := \lceil 2c \rceil$ and $f := 2c^2$. As $c \leq 1/2 \cdot (\log \ell)/(\log\log \ell)^{1/4} \leq (\log \ell)/2 - 1$ for sufficiently large ℓ, we have $h \leq 2c + 1 \leq \log \ell - 1$, as demanded in (3.3).

From Theorem 3.13, we know that any algorithm using $\ell/(2^h \cdot f^2) \cdot \log e - \log h$ advice bits has a competitive ratio of at least

$$\frac{h+2}{2\big(1+\frac{h}{f}\big)} = \frac{h+2}{2 \cdot \frac{h+f}{f}} = \frac{f}{2} \cdot \frac{h+2}{h+f}. \tag{3.7}$$

Since $c \geq 1$, we have $f = 2c^2 \geq 2$, and thus $(h+2)/(h+f) \leq 1$. For any fixed f, this term (and therefore also the competitive ratio) grows with growing h. As $h \geq 2c$, (3.7) implies that the competitive ratio is at least

$$\frac{h+2}{2\big(1+\frac{h}{f}\big)} \geq \frac{2c+2}{2\big(1+\frac{2c}{2c^2}\big)} = \frac{c+1}{1+\frac{1}{c}} = \frac{c+1}{\frac{c+1}{c}} = c.$$

As $h \leq 2c+1$ and $h \leq \log \ell - 1 < \log \ell$, we obtain for the number b of advice bits necessary

$$b \geq \frac{\ell}{2^h\, f^2} \cdot \log e - \log h \geq \frac{\ell}{2^{2c+1}\, 4c^4} \cdot \log e - \log h > \frac{\ell}{4^c\, c^4} \cdot \frac{\log e}{8} - \log\log \ell.$$

It remains to show that this term is in $\Omega\big(\ell/(4^c\, c^4)\big)$. For sufficiently large ℓ, we have $c \leq 1/2 \cdot (\log \ell)/(\log\log \ell)^{1/4} \leq 1/2 \cdot (\log \ell - 4 \log\log \ell)$, and therefore

$$4^c \leq 2^{\log \ell - 4 \log\log \ell} = \frac{2^{\log \ell}}{(2^{\log\log \ell})^4} = \frac{\ell}{(\log \ell)^4},$$

which implies

$$c^4 \cdot 4^c \leq \frac{1}{2^4} \cdot \frac{(\log \ell)^4}{\log\log \ell} \cdot \frac{\ell}{(\log \ell)^4} = \frac{\ell}{2^4\, \log\log \ell}.$$

Hence, we have

$$\frac{\ell}{c^4 \cdot 4^c} \geq 2^4\, \log\log \ell$$

and thus,

$$\log\log \ell \leq \frac{\ell}{c^4 \cdot 4^c} \cdot \frac{1}{2^4}.$$

Finally, we obtain

$$\begin{aligned} b &> \frac{\ell}{4^c\, c^4} \cdot \frac{\log e}{8} - \log\log \ell \\ &\geq \frac{\ell}{4^c\, c^4} \cdot \left(\frac{\log e}{8} - \frac{1}{2^4}\right) \\ &> 0.11 \cdot \frac{\ell}{4^c\, c^4} \\ &\in \Omega\left(\frac{\ell}{4^c\, c^4}\right), \end{aligned}$$

and thus the advice complexity to achieve a competitive ratio of c is indeed in $\Omega\big(\ell/(4^c\, c^4)\big)$. □

From Theorem 3.13, we can also derive a more concrete result concerning the number of advice bits necessary to achieve competitive ratios in the order of $\log \ell$, as we will see now.

Corollary 3.15. *Let δ be an arbitrary constant with $0 < \delta < 1$. Any online algorithm for* PATHDPA *that achieves a strict competitive ratio of $\delta/2 \cdot \log \ell$ needs to read at least $\omega(\ell^{1-\varepsilon})$ advice bits, for any constant ε with $\delta < \varepsilon < 1$.*

Proof. Let $h := \lfloor \delta \log \ell \rfloor \leq \delta \log \ell$ and $f := (\log \ell)^2$. Then, due to Theorem 3.13, the number of advice bits necessary to achieve competitive ratio c is

$$\begin{aligned} b &\geq \frac{\ell}{2^h\, f^2} \cdot \log e - \log h \\ &\geq \frac{\ell}{2^{\delta \log \ell}\, (\log \ell)^4} \cdot \log e - \log(\delta \log \ell) \\ &= \frac{\ell}{\ell^{\delta}\, (\log \ell)^4} \cdot \log e - \log \delta - \log\log \ell \\ &= \frac{\ell^{1-\delta}}{(\log \ell)^4} \cdot \log e - \log \delta - \log\log \ell. \end{aligned} \tag{3.8}$$

For any constant $\varepsilon > \delta$, we have $\varepsilon - \delta > 0$ and thus

$$\frac{\ell^{1-\delta}}{\ell^{1-\varepsilon}} = \ell^{\varepsilon-\delta} \in \omega\big((\log \ell)^4\big),$$

which is equivalent to

$$\frac{\ell^{1-\delta}}{(\log \ell)^4} \in \omega(\ell^{1-\varepsilon}).$$

Therefore, the number of advice bits necessary to obtain the competitive ratio c is in $\omega(\ell^{1-\varepsilon})$, for any constant $\varepsilon > \delta$. The value of c obtained by plugging in the values for h and f as chosen above is

$$\begin{aligned}
c &= \frac{h+2}{2(1+\frac{h}{f})} \\
&\geq \frac{\delta \log \ell + 1}{2\left(1 + \frac{\delta \log \ell}{\log^2 \ell}\right)} \\
&= \frac{\delta \log \ell + 1}{2} \cdot \frac{(\log \ell)^2}{(\log \ell)^2 + \delta \log \ell} \\
&= \frac{\delta}{2}\left(\log \ell + \frac{1}{\delta}\right) \cdot \frac{(\log \ell)^2}{\log \ell \cdot (\log \ell + \delta)} \\
&= \frac{\delta}{2} \cdot \log \ell \cdot \frac{\log \ell + 1/\delta}{\log \ell + \delta} \\
&> \frac{\delta}{2} \cdot \log \ell,
\end{aligned}$$

where the last inequality holds since $\delta < 1$ and thus $\delta < 1/\delta$. □

4 Graph Searching and Exploration

In this chapter, we consider two different problems which, in some sense, differ from classical online problems. More precisely, we study the so-called *graph searching problem*, also denoted by SEARCH, and the *graph exploration problem*, for short denoted by EXPLORE. In both scenarios, an algorithm $\mathcal{A}$ is given a graph and a *starting vertex* in the graph, at which a so-called *agent* is located at the beginning of the computation. The algorithm can explore the graph by moving the agent along the edges of the graph. Each vertex has a unique *identifier (ID)*, and as soon as the agent visits a vertex v, the algorithm $\mathcal{A}$ learns all neighboring vertices of v, including their IDs and the costs of the edge between v and this neighbor, respectively. While moving through the graph, the agent gains information about the graph topology. The agent has unlimited memory and can hence record all information it gained about the graph. Thus, $\mathcal{A}$ always knows the ID of the vertex the agent is currently located at and is able to recognize vertices the agent has already visited or already seen from one of its neighbors, based on their IDs.

The goal of the algorithm in the graph searching problem is to guide the agent to a distinguished target vertex, using a shortest possible path. As opposed to this, in the graph exploration problem, the agent has to explore the graph completely; each vertex has to be visited at least once, and in the end, the agent must be located at the starting vertex again. In both scenarios, the performance of the algorithm is measured in the number of vertices.

Problems concerned with navigation in unknown territories have been studied for a long time, in a lot of different variants. Although all dealing with navigating an agent through unknown terrain, many problems considered differ substantially, for example, in the exact goal that the algorithm is supposed to fulfill, the character of the underlying environment, and the properties of the agent. Considered scenarios range from reaching a specified position in a plane with obstacles, over drawing a complete map of a graph, to chasing moving targets in a graph with multiple agents. For an overview of many different versions of graph searching and exploration problems, we point the reader to the surveys of Ghosh and Klein [GK10] and Berman [Ber96].

The problem of exploring an unweighted graph by visiting all its edges was introduced by Deng and Papadimitriou [DP90]. Kalyanasundaram and Pruhs presented a slight adaptation of this scenario, in which all vertices of a weighted graph have to be visited [KP94]. They also introduced the so-called *fixed graph scenario*, which defines the amount and type of information an agent gets upon visiting a new vertex, and which we also consider in this thesis. Papadimitriou and Yannakakis introduced the problem of finding a shortest s-t-path from a source s to a target t in the fixed graph scenario [PY91], which corresponds to the graph searching problem studied in this thesis.

Kalyanasundaram and Pruhs also proposed a generalization of depth first search that is applicable to general graphs and which is 16-competitive for EXPLORE on planar graphs [KP94]. For a long time, it was commonly assumed that this algorithm had a constant competitive ratio even on general graphs, until Megow et al. showed that, as a matter of fact, it does not [MMS11]. Among other things, they presented an algorithm that achieves a competitive ratio of $2k$ on general graphs with k distinct weights, implying a constant competitive ratio when the number of different edge weights is bounded. Still, it remains unknown whether there is an algorithm for EXPLORE achieving a constant competitive ratio for general graphs. Miyazaki et al. presented an algorithm for EXPLORE that achieves a competitive ratio on simple cycles of $(1+\sqrt{3})/2$ and showed that there is no deterministic algorithm with a competitive ratio of $(1+\sqrt{3})/2-\varepsilon$, for any positive constant ε [MMO09]. Additionally, for undirected unweighted graphs, they gave a 2-competitive algorithm, and proved a lower bound on the competitive ratio of at least $2-\varepsilon$ for any constant ε. This result was improved by Dobrev et al. [DKM12], who showed that every deterministic algorithm for EXPLORE has a competitive ratio of at least $5/2-\varepsilon$ on any undirected weighted graph. Foerster and Wattenhofer studied both the graph exploration and the graph searching problem in directed graphs [FW12]. For the exploration problem in directed weighted graphs, they gave a lower bound of $n-1$ on the competitive ratio for any deterministic algorithm, which exactly matches the upper bound of $n-1$ for a greedy algorithm. For the graph searching problem in unweighted directed graphs, they proved a lower bound of $\Omega(n^2)$ on the competitive ratio for any deterministic algorithm and a lower bound of $\Omega(n)$ for any randomized algorithm.

Again, our interest focuses on the research that has been carried out in the field of such graph searching or exploration algorithms with advice. Regarding this, Nisse and Soguet studied a problem they also called “graph searching problem”, but their setting involved several agents chasing a fugitive in a graph [NS07]. A problem similar to the exploration problem studied in this thesis was considered by Fraigniaud et al. [FIP08], who investigated the problem of visiting all vertices of a tree, when the agent may end in an arbitrary vertex. For d being the diameter of the tree, the authors presented an algorithm with a competitive ratio of less than 2 that reads $\log \log d - c$ bits of advice, for some constant c, whereas no algorithm without advice can achieve a better competitive ratio than 2. The same graph exploration problem as in this thesis was considered by Dobrev et al. [DKM12], who proved a lower bound of $\Omega(n \log n)$ necessary advice bits to achieve optimality and presented an algorithm reading $O(n)$ advice bits that achieves a constant competitive ratio. A yet unpublished thesis [Ful14] shows a lower bound of $\Omega(n)$ advice bits necessary for any EXPLORE algorithm to be better than approximately 1.564-competitive. To the best of our knowledge, there has not been any research on SEARCH with advice so far.

In this chapter, we further analyze the graph searching and the graph exploration problems with respect to their advice complexity. Both of these problems can naturally be interpreted as online problems. Whenever an algorithm for either SEARCH or EXPLORE enters a new vertex, the information it receives about its neighbors is interpreted as a request; the neighboring vertex to which the algorithm moves subsequently forms the algorithm’s output. In this sense, both of these problems are online problems, as they receive the request sequence in an online fashion, and they have to compute their output without knowing future requests. Thus, we measure their output quality in terms of the competitive ratio as we are used to. Yet, these problems differ from usual online problems in the sense that the request sequence is dependent on the given algorithm, which is not the case for most common online problems. This actually makes a notable difference, as we will see in this chapter. For such kinds of online problems, we cannot apply the same reduction technique that we used in the two preceding chapters to obtain a trade-off between the number of advice bits and the competitive ratio of an online algorithm. Hence, in this chapter, we will first demonstrate that algorithms for this kind of online problems can be supplied with advice to improve their competitive ratio, just like common online algorithms. Moreover, we will show that by adapting the familiar reduction technique it is also possible to transfer lower bounds for the bit string guessing problem to algorithms for such problems, using the example of EXPLORE to demonstrate this.

In Section 4.1, we briefly elaborate on a few details concerning the model and our notation used throughout this chapter. Section 4.2 is dedicated to the graph searching problem. In Section 4.2.1, we give an algorithm reading n bits of advice that solves the graph searching problem optimally on any directed weighted graph containing n vertices, which complements the lower bound of $\Omega(n^2)$ for the competitive ratio in unweighted directed graphs mentioned above [FW12]. In the following two sections, we give asymptotically matching lower and upper bounds for the competitive ratio of SEARCH; in Section 4.2.2, we give a lower bound of $\Omega(n/c)$ advice bits necessary to achieve a competitive ratio of c, and in Section 4.2.3, a c-competitive algorithm for SEARCH reading $O(n/c)$ advice bits is presented. Section 4.3 deals with EXPLORE. By giving a reduction from the string guessing problem, we obtain a trade-off for the number of advice bits necessary to achieve certain competitive ratios; we prove a lower bound of $\big(1 - \eta(1 - \alpha)\big)\, n$ advice bits to achieve a competitive ratio of $c \leq (4 - \alpha)/3 - \varepsilon$ for any constant $\varepsilon > 0$.

4.1 Preliminaries

Throughout this chapter, the environment in which the agent moves is usually an undirected unweighted graph. An exception is made for an upper bound given in Section 4.2.1, for which we consider a directed weighted graph. Although we will not always mention this explicitly, we assume for both SEARCH and EXPLORE that the input graphs admit a feasible solution. For SEARCH, this means that the designated destination vertex has to be reachable from the starting vertex; for EXPLORE, this means that the graph has to be connected. Moreover, we assume for both problems that the size of the graph (i. e., the number n of vertices) is given as the first request in round 0.

Classically, the advice complexity of an online algorithm is measured as a function of the input length. For SEARCH and EXPLORE (and other online problems for which the length of the instance depends on the algorithm's actions), however, there are some formal problems with this. Even on a very small graph, an algorithm for such an online problem could read an arbitrary number of advice bits by simply moving between two vertices repeatedly. In this case, the number of advice bits used may be, for example, linear in the input length, while being exponential in n. Therefore, for this certain class of online problems, we measure the advice complexity in the number of vertices of the underlying graph.

Let us now briefly describe what kind of information the agent gets when entering a vertex. Throughout this whole chapter, we consider the fixed graph scenario mentioned above [KP94]. In this setting, each vertex has a unique identifier (ID), and as soon as the agent visits a vertex v, it learns all edges incident to v, including their costs (in the case of weighted graphs) and (the ID of) the endpoint of each such edge. In the case of directed graphs (which we only consider for an upper bound given in Section 4.2.1, as already mentioned) it is in our case sufficient if the agent only gets this information about all outgoing edges, and not about the incoming ones.

In the remainder of this chapter, we will often refrain from distinguishing between the algorithm and the agent, but instead refer to the agent $\mathcal{A}$ as well as the algorithm $\mathcal{A}$.

4.2 Graph Searching

We start our considerations with the graph searching problem. This problem has been considered in many variants, but apparently there has not been any research concerning the advice complexity of this problem so far. In the following sections, we present the first results on graph searching with advice.

4.2.1 Optimality

First, we give an optimality result. As we already mentioned, it has been proven that in unweighted directed graphs any deterministic algorithm for SEARCH has a competitive ratio of $\Omega(n^2)$ and any randomized algorithm at least $\Omega(n)$, no matter how many random bits it uses [FW12]. These lower bounds naturally also apply for weighted directed graphs. In the following, we show that it is possible to solve the graph searching problem optimally in any weighted directed graph by using only n advice bits. Compared to any deterministic algorithm, the competitive ratio can thus be decreased by a factor of $\Omega(n^2)$ by using n advice bits, and compared to a randomized one, it can be decreased by a factor of $\Omega(n)$. Hence, despite providing an arbitrary amount of random bits, the competitive ratio cannot be pushed below $\Omega(n)$, whereas only n advice bits are already sufficient to achieve competitive ratio 1. This shows that advice bits are a lot more powerful than random bits for SEARCH.

Before we present an optimal algorithm for SEARCH with the above-mentioned properties, we give a technical result we will need in the following proof.

Lemma 4.1. *Let $G = (V, E)$ be a directed weighted graph with a designated starting vertex $u_0 \in V$, a designated destination vertex $u_\ell \in V$, and a weight function $\omega\colon E \to \mathbb{Q}^{>0}$ assigning positive weights to the edges, and let $U := U(u_0, u_\ell) = (u_0, u_1, \ldots, u_\ell)$ be a shortest path from u_0 to u_ℓ in G. Then for all vertices u_i, the following holds. Among all edges from u_i to vertices $u_{j'}$ with $j' > i$, the edge (u_i, u_{i+1}) is the only one that has minimum weight, i. e., $\omega(u_i, u_{i+1}) < \omega(u_i, u_{j'})$ for all $j' > i + 1$.*

Proof. For some fixed u_i with $1 \leq i \leq \ell - 1$, consider an arbitrary vertex u_j with $j > i + 1$ for that there is an edge $(u_i, u_j) \in U$. For the sake of contradiction, let us assume that the weight of (u_i, u_j) is at most $\omega(u_i, u_{i+1})$. Since U is a shortest path from u_0 to u_ℓ, a shortest path from u_i to u_j must be $(u_i, u_{i+1}, \ldots, u_j)$ with a total weight of

$$\omega(u_i, u_{i+1}) + \omega(u_{i+1}, u_{i+2}) + \ldots + \omega(u_{j-1}, u_j).$$

As $\omega(u_i, u_j) \leq \omega(u_i, u_{i+1})$, the weight of the shortest path from u_i to u_j has a weight of

$$\omega(u_i, u_j) + \omega(u_{i+1}, u_{i+2}) + \ldots + \omega(u_{j-1}, u_j) > \omega(u_i, u_j),$$

which cannot be the case since then the direct edge between u_i and u_j would be a shorter path (with weight $\omega(u_i, u_j)$). Thus, for all j with $j > i+1$, we have $\omega(u_i, u_j) > \omega(u_i, u_{i+1})$. $\square$

Theorem 4.2. *There is an online algorithm $\mathcal{A}$ with advice reading n advice bits that solves the graph searching problem optimally on any graph with n vertices.*

Proof. Let $G = (V, E)$ be a directed weighted graph with a designated starting vertex $u_0 \in V$, a designated destination vertex $u_\ell \in V$, and a weight function $\omega\colon E \to \mathbb{Q}^{>0}$ assigning positive weights to the edges. Let the set of vertices be $V = \{v_1, \ldots, v_n\}$, and let $U := U(u_0, u_\ell) = (u_0, u_1, \ldots, u_\ell)$ be some shortest path from u_0 to u_ℓ in G. We present an algorithm $\mathcal{A}$ that reads n advice bits and computes an optimal solution on any such input.

Before the beginning of the computation, $\mathcal{A}$ reads an advice string $\tau_1 \ldots \tau_n$ of length n from the tape, where the oracle sets the advice bit $\tau_i = 1$ to indicate that v_i belongs to U and $\tau_i = 0$ otherwise, for $1 \leq i \leq n$. Recall that $\mathcal{A}$ knows n as it is given with the first request. At the beginning of the computation, i. e., at the beginning of round 1, the agent is located at vertex u_0. During the computation, it holds as an invariant that, for every i with $0 \leq i \leq \ell - 1$, at the beginning of round $i+1$, the walk that the agent has taken through the graph so far is $(u_0, \ldots, u_i)$. At the beginning of round $i+1$, when located at vertex u_i, the agent can see all outgoing edges of u_i, including their weights and the (IDs of) their corresponding endpoints. From the advice string that $\mathcal{A}$ read before, it knows which of these neighboring vertices belong to the shortest path U. The agent then moves along an outgoing edge to some neighbor v of u_i that has not been visited yet, lies on U, and for which the edge weight $\omega(u_i, v)$ is minimal. Due to Lemma 4.1, there is exactly one such neighbor of u_i, namely u_{i+1}. Therefore, the agent travels from u_i to u_{i+1} in this round, making sure that the invariant remains true. Hence, $\mathcal{A}$ is an algorithm with advice that needs only n advice bits to solve the graph searching problem optimally on any input. $\square$

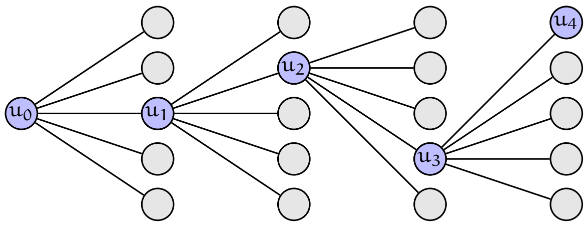


Figure 4.1. Example of the graph G for $n = 21$, $p = 5$, and $q = 4$. The starting vertex u_0 is the only vertex on level 0. There are q additional levels and p vertices per level. There are no additional dummy vertices on level 1 since $m = n - pq - 1 = 0$. There is one designated vertex per level, colored in blue; u_i is the designated vertex on level i. Each such vertex u_i for $0 \leq i \leq q - 1$ is connected to all vertices on level $i + 1$. The vertex u_q is the destination vertex.

4.2.2 Lower Bound

We have just seen that n advice bits are sufficient to achieve optimality in any directed weighted graph. Questions that naturally come to mind are: If we do not want to achieve optimality, but only some competitive ratio c, how many advice bits are sufficient? How many are necessary? In this section, we answer the latter question by giving a lower bound of $(n - 1)/(16(c + 2)) \in \Omega(n/c)$ advice bits to achieve a competitive ratio of c, for any $c \geq 1$, in any undirected unweighted graph. This lower bound immediately translates to directed weighted graphs as well, since any undirected unweighted graph can be modeled as a directed graph with unit weights.

In the following proof, we will make use of a class $\mathcal{I}$ of instances for SEARCH called pq-trees (see Figure 4.1), which are defined as follows. For given p, q, n such that $n > pq$, each instance from $\mathcal{I}$ is an undirected unweighted graph $G = (V, E)$ with $|V| := n$ vertices and $q+1$ designated vertices $u_i \in V$, for $0 \leq i \leq q$. Each such tree has a root on level 0 that serves as the designated vertex u_0, and pq additional vertices on q different levels, each containing p vertices. On each level i, there is exactly one designated vertex u_i, and for every level i with $0 \leq i \leq q - 1$, the designated vertex u_i is connected to all vertices on level $i + 1$. Additionally, there is a designated vertex u_q on level q that serves as the destination vertex. The root u_0 serves as the starting vertex and is connected to $m := n - pq - 1$ additional vertices, so-called *dummy vertices*.

There are p^q possible ways to choose the vertices $u_1, \ldots, u_q$, and thus $\mathcal{I}$ contains $|\mathcal{I}| = p^q$ different instances. The shortest path between the starting and the destination vertex on any instance is the path $(u_0, \ldots, u_q)$ containing all designated vertices. Hence, the optimal solution on any instance $I \in \mathcal{I}$ has a cost of $\text{cost}(\text{Opt}(I)) = q$.

Theorem 4.3. *Let* c *be any function of* n *such that* $1 \leq c < n/18$ *for each* n*. Then any online algorithm for the graph searching problem on graphs with* n *vertices needs to read at least* $\Omega(n/c)$ *advice bits to achieve a competitive ratio of* c *on graphs with* n *vertices.*

Proof. Let us fix some large enough n. For the sake of contradiction, let us assume that there is an algorithm $\mathcal{A}$ for SEARCH reading at most $b := b(n) \in o(n/c)$ advice bits with a competitive ratio of at most c on pq-trees with n vertices. As $\text{cost}(\texttt{Opt}(I)) = q$, there must exist a constant a such that

$$\text{cost}(\mathcal{A}(I)) \leq c \cdot \text{cost}(\texttt{Opt}(I)) + a = qc + a,$$

for all instances I from the class $\mathcal{I}$ of pq-trees with n vertices. We construct a particular pq-tree with n vertices on which the algorithm reads more than $\beta \cdot n/c$ advice bits, for some constant $\beta > 0$ that depends on the algorithm and a but neither on the function c nor on n, leading to a contradiction to the assumption that $\mathcal{A}$ reads $o(n/c)$ advice bits.

Set $q := \lfloor n/(kc) \rfloor$ for a suitable constant k specified later, and choose p such that $n = pq + m$ for some m with $1 \leq m \leq q$. Consider all p^q instances I with parameters p, q, n. The agent traverses the graph, starting at u_0, until it finally reaches the destination vertex u_q. Until it does, it must visit at least one vertex on each level. The order in which A traverses the vertices on each level i is arbitrary and might even depend on the position of the designated vertices on levels $1, \ldots, i-1$, but it is fixed for any fixed instance. On any level i with $1 \leq i \leq q$, we define e_i to be such that A first visits $e_i - 1$ leaves on level i and, after visiting each such leaf, has to return to u_{i-1}, before finding the designated vertex u_i on level i. Let us assume that A never visits a leaf twice and never returns to level $i-1$ once it has found u_i. We can do so without loss of generality because, for every algorithm A' that makes such an unnecessary move, there is an algorithm A that does not and that has a cost of $\text{cost}(A(I)) < \text{cost}(A'(I))$ on any instance I. Then we can identify each instance $I \in \mathcal{I}$ with the characteristic vector $(e_1, e_2, \ldots, e_q)$, with $e_i \in \{1, \ldots, p\}$ for $1 \leq i \leq q$, and the property that the cost of A on instance I is at least $\sum_{i=1}^{q} (2e_i - 1)$ (ignoring the potential dummy vertices).

Let us call an instance I *good* if A achieves a competitive ratio of at most c on it, i. e., if the cost of A on I is at most $\text{cost}(\texttt{Opt}(I)) \cdot c + a = qc + a$. Making some simple transformations, we obtain

$$\begin{aligned} & \text{cost}(A(I)) \leq qc + a \\ \iff & \sum_{i=1}^{q} (2e_i - 1) \leq qc + a \\ \iff & \sum_{i=1}^{q} e_i \leq \frac{q(c + 1 + a/q)}{2}. \end{aligned} \tag{4.1}$$

For convenience, let us denote $d \coloneqq (c + 1 + a/q)/2$. This implies that an instance is good if, for its corresponding characteristic vector $(e_1, \ldots, e_q)$,

$$\sum_{i=1}^{q} e_i \leq qd. \tag{4.2}$$

According to Fact 1.1, we can interpret each graph searching algorithm that uses b bits of advice as a set $\{A_1, A_2, \ldots, A_{2^b}\}$ of deterministic algorithms, as we usually do with online algorithms with advice. Thus, as $\mathcal{A}$ reads at most b advice bits and is c-competitive according to our initial assumption, there is at least one such deterministic algorithm A that computes a solution with a competitive ratio of at most c on at least $p^q/2^b$ instances. From now on, let us consider this particular deterministic algorithm A, and let us define the set of good instances for A to be $\mathcal{I}^+$. Hence,

$$|\mathcal{I}^+| \geq \frac{p^q}{2^b}. \tag{4.3}$$

Now let us bound $|\mathcal{I}^+|$, the number of good instances for A, from above. For any good instance, the corresponding characteristic vector must contain at least $q/2$ entries e_i with value at most $2d$, otherwise $\sum_{i=1}^{q} e_i > q/2 \cdot 2d = qd$, contradicting (4.2). Hence, the number of good instances is upper-bounded by the number of vectors $(e_1, \ldots, e_q)$, where $e_i \in \{1, \ldots, p\}$, with at least $q/2$ entries with value at most $2d$. To bound this term from above, we make the following considerations. The number of vectors of length $q/2$ with values of at most $2d$ is $(2d)^{q/2}$; the number of vectors of length $q/2$ with values between 1 and p is $p^{q/2}$. The number of possibilities to join two vectors of these two different kinds to construct a vector of length q is $\binom{q}{q/2}$. The same vector of length q might be generated by joining different pairs of vectors of length $q/2$. Nevertheless, these considerations yield an upper bound. The number of characteristic vectors with at least $q/2$ entries with value at most $2d$, and thus also the number of good instances, is therefore

$$|\mathcal{I}^+| \leq (2d)^{\frac{q}{2}} \cdot p^{\frac{q}{2}} \cdot \binom{q}{\frac{q}{2}}. \tag{4.4}$$

Putting (4.3) and (4.4) together yields

$$\frac{p^q}{2^b} \leq |\mathcal{I}^+| \leq (2d)^{\frac{q}{2}} \cdot p^{\frac{q}{2}} \cdot \binom{q}{\frac{q}{2}} \leq (2dp)^{\frac{q}{2}} \cdot 2^q \leq (2dp)^{\frac{q}{2}} \cdot 4^{\frac{q}{2}} = (8dp)^{\frac{q}{2}}.$$

We rearrange this inequality to solve it for b and obtain

$$\begin{aligned} & \frac{p^q}{2^b} \leq (8dp)^{\frac{q}{2}} \\ \iff\ & 2^b \geq \frac{p^q}{(8dp)^{\frac{q}{2}}} \\ \iff\ & 2^b \geq \left(\frac{p^2}{8dp}\right)^{\frac{q}{2}} \\ \iff\ & b \geq \frac{q}{2} \cdot \log\left(\frac{p}{8d}\right). \end{aligned} \tag{4.5}$$

Recall that $p = (n-m)/q \geq (n-q)/q$ as $m \leq q$. Resubstituting d, we get

$$\frac{p}{8d} = \frac{p}{8\left(\frac{c+1+a/q}{2}\right)} \geq \frac{n-q}{4q(c+1+a/q)}.$$

If it holds that

$$q < \frac{n-8a}{8c+9}, \tag{4.6}$$

we obtain

$$\begin{aligned} & q < \frac{n-8a}{8c+9} \\ \iff\ & q \cdot (8(c+1)+1) < n-8a \\ \iff\ & q \cdot 8(c+1) + 8a < n-q \\ \iff\ & 8q(c+1+a/q) < n-q, \end{aligned}$$

and thus

$$\frac{p}{8d} > \frac{8q(c+1+a/q)}{4q(c+1+a/q)} = 2.$$

Combining this with (4.5) and $q = \lfloor n/(kc) \rfloor \geq n/(kc) - 1$ as defined above, we obtain

$$b \geq \frac{q}{2} \cdot \log\left(\frac{p}{8d}\right) > \frac{q}{2} \geq \frac{1}{2}\left(\frac{n}{kc} - 1\right) = \frac{1}{2k} \cdot \frac{n}{c} - \frac{1}{2}.$$

Since $n/c > 18$ according to our assumption, choosing $\beta := 1/(2k) - 1/36$ yields

$$b > \frac{1}{2k} \cdot \frac{n}{c} - \frac{1}{2} = \left(\beta + \frac{1}{36}\right) \cdot \frac{n}{c} - \frac{1}{2} > \beta \cdot \frac{n}{c} + \frac{18}{36} - \frac{1}{2} = \beta \cdot \frac{n}{c}.$$

Hence, if we can show that there is a suitable choice for k such that $\beta > 0$ and (4.5) is true, we have already proven the desired result. For $\beta > 0$ to be true, it must hold that $1/(2k) > 1/36$ and thus $k < 18$. Now we finally show that there is a suitable choice of k such that $k < 18$ and (4.6) is true. For (4.6) to hold, it is sufficient to choose k such that

$$k > \frac{n}{n - 8a} \cdot \frac{8c + 9}{c}.$$

The first factor converges to 1 with increasing n. As $c \geq 1$, the second factor is always at most 17. Consequently, there is a large enough n (depending on a) such that $k := 17.5$ is suitable. □

4.2.3 Upper Bound

We have now shown a lower bound of $\Omega(n/c)$ advice bits to achieve a competitive ratio of c, and in what follows, we want to prove an upper bound of $O(n/c)$ advice bits sufficient to achieve a competitive ratio of c on any undirected unweighted graph. For our argumentation, we will need the following already known result from the field of graph separators. Any tree with n vertices can be divided into two parts, each with at least $n/3$ and at most $2n/3$ vertices, by removing a single vertex and then allocating the entire vertices of each subtree to one of the two parts appropriately (see, for example, [LT79]). In our case, we need the following own formulation of this result. We also give a proof for the sake of completeness.

Lemma 4.4. *For any $c \in \mathbb{R}^{>6}$, let $G = (V, E)$ be a connected graph with $|V| = n \geq c/3$ vertices. Then there are two sets of vertices $C, D \subseteq V$ such that $C \cup D = V$, $|C \cap D| = 1$, both $|C| \geq c/9 + 1$ and $|D| \geq c/9 + 1$, and both subgraphs of G induced by C and D are connected, respectively.*

Proof. It is sufficient to prove the lemma for trees, since then it can be applied to a spanning tree of an arbitrary connected graph. Hence, let $G = (V, E)$ be an arbitrary tree. For any vertex w, let G decompose into n_w trees $T_1^{(w)}, \ldots, T_{n_w}^{(w)}$ when removing w from V. For every $w \in V$ and every i with $1 \leq i \leq n_w$, let us define $V_i^{(w)}$ to be the vertex set of $T_i^{(w)}$.

First, we prove that there is a vertex $w \in V$ such that, for each i with $1 \leq i \leq n_w$, we have $|V_i^{(w)}| \leq n/2$. To this end, let us assume towards contradiction that for each vertex $w \in V$ there exists some index j such that $|V_j^{(w)}| > n/2$. Since for any vertex w there cannot be more than one such subtree containing more than $n/2$ vertices, there must be exactly one such tree per vertex w, and without loss of generality, let this be $T_1^{(w)}$. Now consider a vertex w such that $|V_1^{(w)}|$ is minimal among all vertex sets $V_1^{(u)}$ for all $u \in V$, and let v be the (unambiguous) neighbor of w in $T_1^{(w)}$.

From the point of view of v, the subtree $T_i^{(v)}$ that is rooted in w has less than $n/2$ vertices and thus cannot be the subtree $T_1^{(v)}$ with $|V_1^{(v)}| > n/2$. Hence, the

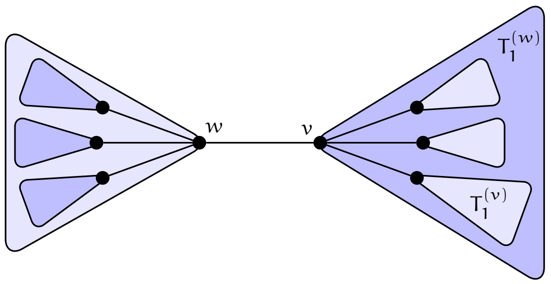


Figure 4.2. Schematic drawing of the graph G in the situation described in Lemma 4.4. When removing v from G, the graph decomposes into the light blue components. When removing w, it decomposes into the darker ones. The tree $T_1^{(w)}$ is the one of these subtrees that remain after removing w containing more than $n/2$ vertices. Hence, the light blue subtree rooted at w must contain less than $n/2$ vertices. Thus, the subtree $T_1^{(v)}$ that contains more than $n/2$ vertices must be one of the subtrees within $T_1^{(w)}$, and therefore it has less vertices than $T_1^{(w)}$.

subtree of v with more than $n/2$ vertices must be one of the subtrees contained in $T_1^{(w)}$, and therefore, $T_1^{(v)}$ has less vertices than $T_1^{(w)}$. This is a contradiction to the minimality of $T_1^{(w)}$. A schematic picture is shown in Figure 4.2.

Consequently, we can now consider a vertex $w \in V$ such that each $|V_i^{(w)}| \leq n/2$ for all $1 \leq i \leq n_w$. If at least one of the trees $T_i^{(w)}$ has $|V_i^{(w)}| \geq c/9$ vertices, say $T_j^{(w)}$, then we can define the vertex sets C and D to be $C := T_j^{(w)} \cup \{w\}$ and $D := (V \setminus C) \cup \{w\}$. It holds that

$$|D| = n - |T_j^{(w)}| + 1 \geq n - \frac{n}{2} + 1 \geq \frac{c}{6} + 1 \geq \frac{c}{9} + 1$$

for every $c \geq 0$. All other requirements of the statement are clearly also fulfilled.

On the other hand, if all the $T_i^{(w)}$ only have $|V_i^{(w)}| < c/9$ vertices, we define C and D as follows. We assign the vertices of the subtrees $T_i^{(w)}$ one after another greedily and subtree-wise to one of the vertex sets C and D, namely to the one that currently contains fewer vertices. When all vertices of all these subtrees are assigned, we additionally add w to both C and D. Hence, in the end, the cardinalities of the two parts differ by at most $c/9$, which means that both sets contain at least

$$\frac{n}{2} - \frac{c/9}{2} + 1 \geq \frac{\frac{c}{3} - \frac{c}{9}}{2} + 1 \geq \frac{3c - c}{2 \cdot 9} + 1 = \frac{c}{9} + 1$$

vertices, and the sets C and D clearly also fulfill all other requirements of the statement. □

Now we use this result to prove an upper bound that asymptotically matches the lower bound of $\Omega(n/c)$ advice bits that are necessary to achieve competitive ratio c. We give a c-competitive algorithm that reads only $9n/c$ advice bits for any $c \geq 6$.

Theorem 4.5. *For any $c \in \mathbb{R}^{>6}$ and any $n \in \mathbb{N}^{\geq 1}$, there is a c-competitive online algorithm $\mathcal{A}$ with advice for the graph searching problem in undirected unweighted graphs with n vertices that uses only $\lfloor 9n/c \rfloor \in O(n/c)$ advice bits.*

Proof. Consider any undirected unweighted graph $G = (V, E)$ with $|V| = n$ vertices, with a starting vertex u_0 and a destination vertex u_ℓ, that the algorithm $\mathcal{A}$ gets as an input. To analyze $\mathcal{A}$'s advice complexity and competitive ratio, we use two types of accounting, a *charge* for each vertex and a *credit* for the algorithm $\mathcal{A}$. Initially, every vertex has charge $9/c$, and the algorithm's credit is 0. The algorithm may *harvest* a vertex v, which adds the charge of v to $\mathcal{A}$'s credit, increasing the latter by $9/c$ credit units. At any time, $\mathcal{A}$ may read an advice bit from the tape, but we subtract one credit unit from $\mathcal{A}$'s account every time it does. If we make sure that no vertex is harvested twice, the overall number of advice bits that $\mathcal{A}$ uses is at most $9n/c$.

Let $U := U(u_0, u_\ell) = (u_0, u_1, \ldots, u_\ell)$ be a shortest path from u_0 to u_ℓ in G. Although u_0 and u_ℓ are known to the algorithm, U, of course, is not. For the purpose of our analysis, we book each move of the agent to some edge $\{u_i, u_{i+1}\}$. If we make sure that we do not book costs of more than c to any such edge, the algorithm will be c-competitive.

The algorithm $\mathcal{A}$ works in rounds. During its computation, $\mathcal{A}$ maintains two disjoint sets of vertices that are updated in each round i. The first such set, H_i, contains all vertices that have already been harvested in previous rounds. The other one, T_i, contains all vertices that have already been traversed but not harvested yet. Let each vertex that is a neighbor of a vertex in T_i but that is not contained in T_i or H_i be called a *boundary vertex*, and let the set of all boundary vertices be called the *boundary set*, denoted by B_i. In round i, all vertices of T_i have already been traversed, so the agent knows the entire boundary set B_i at any time. At the beginning of $\mathcal{A}$'s computation, we have $T_1 := \{u_0\}$, $H_1 := \emptyset$, and accordingly $B_1 := \big\{v \in V \mid \{u_0, v\} \in E\big\}$.

There are two different kinds of rounds. In each round i, the agent either traverses the set $T_i \cup B_i$ of vertices in order to find the destination vertex u_ℓ, or it reads one advice bit from the tape to narrow down the set of vertices in which it has to search for u_ℓ. The latter is done whenever $T_i \cup B_i$ is so large that traversing it completely would incur too large costs, and can be repeated by $\mathcal{A}$ for several rounds, until the size of $T_i \cup B_i$ falls below a certain threshold. We will call each round in which $\mathcal{A}$ executes a traversal of some subgraph of G a *traversal round* and each round in which it reads an advice bit instead an *advice round*.

Let us group the rounds of $\mathcal{A}$ into phases such that each phase starts with a number (possibly zero) of advice rounds, in which the search space is reduced gradually until it is of reasonable size, and then ends with a traversal round, in which the entire search space is scanned for u_ℓ. Let us consider some phase p consisting of $m \geq 1$ rounds $h+1, \ldots, h+m$. (Hence, the last round in phase $p-1$ was round h.) We make sure that the following invariants hold for each such phase p and each round i within this phase, for $h+1 \leq i \leq h+m$.

(a) At the beginning of each round i of phase p, there is some boundary vertex $u_j \in B_i$ belonging to the shortest path U, such that no vertices u_k with $k \geq j$ are contained in H_i. If there are several such vertices, let $u^* := u_{j^*}$ be the last such vertex of U, and let $e_i := \{u_{j^*}, u_{j^*+1}\}$.

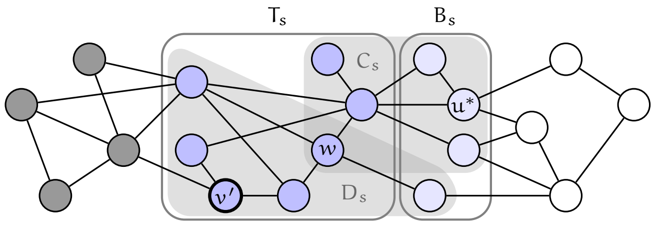
(b) At the beginning of each round of phase p, no costs have been booked to any edge $\{u_k, u_{k+1}\}$ yet, for $j^* \leq k \leq \ell - 1$.

(c) At the beginning of each round of phase p, the agent is located at some vertex $v \in T_{h+1}$.

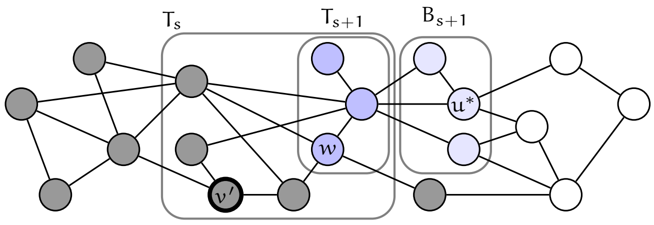
From now on, we will call u_{j^*} the distinguished vertex of round i, and the edge $e_i = \{u_{j^*}, u_{j^*+1}\}$ the distinguished edge. Let us keep in mind that the shortest path U is not known to the algorithm. We introduce these distiguished vertices and edges solely for the purpose of the analysis.

Now let us describe how $\mathcal{A}$ works in greater detail. Figure 4.3 depicts its computation on an examplary graph, which might be helpful for the following description. The computation starts with round 1 of phase 1. For the sake of simplicity, we say that the preceding (dummy) round was round $h = 0$. At the beginning of its computation, the agent is located at u_0. The initial values for the sets are, as we have already mentioned above, $T_1 := \{u_0\}$, $H_1 := \emptyset$, and $B_1 := \big\{v \in V \mid \{u_0, v\} \in E\big\}$. The distinguished vertex u^* is the last vertex from U that is a neighbor of u_0. It is easy to verify that all invariants are fulfilled.

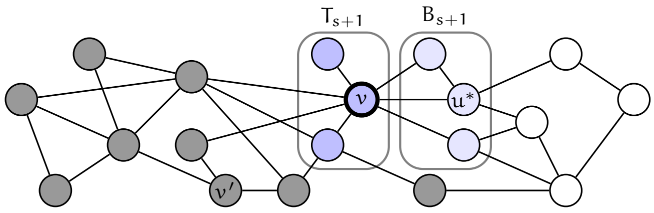
If, in round i, it holds that $|T_i \cup B_i| \geq c/3$, the agent executes an advice round, as the search space is too large. Thus, $\mathcal{A}$ internally splits $T_i \cup B_i$ into two parts C_i and D_i using Lemma 4.4. Then it reads one bit of advice indicating which one of the sets C_i and D_i contains the distinguished vertex u^*. Without loss of generality, let this be C_i. Note that u^* might even be contained in both C_i and D_i, since these sets intersect in one vertex w. If this is the case, the oracle specifies the set C_i to be the one containing u^*. Then the vertices from $D_i \setminus \{w\}$ are harvested by $\mathcal{A}$ to pay for the advice bit that it just read. We have $D_i \setminus \{w\} \subseteq T_i \cup B_i$, so D_i and H_i are disjoint. Hence, the vertices in $D_i \setminus \{w\}$ have not been harvested yet and thus still hold charge $9/c$ each, and Lemma 4.4 guarantees that both C_i and D_i contain at least $c/9 + 1$ vertices. Thus, the agent gains enough credit by harvesting the vertices from $D_i \setminus \{w\}$ to pay for one advice bit. It sets $H_{i+1} := H_i \cup D_i \setminus \{w\}$,



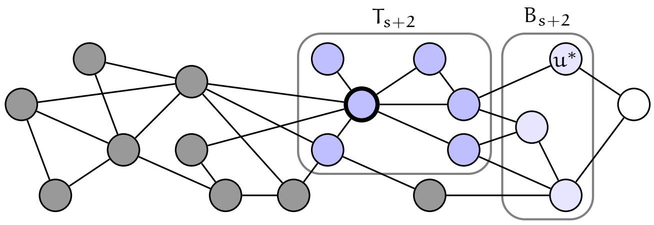
(a) In the preceding round, $s-1$, the agent traversed the set T_s and ended in some vertex $v' \in T_s$. Now, in round s, $T_s \cup B_s$ is too large to be traversed, so $\mathcal{A}$ performs an advice round, splitting $T_s \cup B_s$ into two parts C_s and D_s, which overlap in w.



(b) After round s, $\mathcal{A}$ harvested all vertices from $D_s \setminus \{w\}$ and updated the sets accordingly. The set $T_{s+1} \cup B_{s+1}$ is small enough to be traversed, but $\mathcal{A}$ is located at the vertex $v' \in T_s \setminus T_{s+1}$. Thus, before starting its traversal, $\mathcal{A}$ must move to some vertex in T_{s+1}.



(c) When round $s+1$ begins, the agent has moved to $v \in T_{s+1}$ and can now start its traversal.



(d) After round $s+1$, all vertices from $T_{s+1} \cup B_{s+1} = T_{s+2}$ have been traversed. The agent updates the sets T_{s+2} and B_{s+2} and the distinguished vertex u^* accordingly.

Figure 4.3. An example of a sequence of rounds of the algorithm $\mathcal{A}$. Vertices that have already been harvested are colored gray, those that have been traversed but not harvested are colored blue, those that have been seen as a neighbor but neither been traversed nor harvested are colored light blue, and those that have not even been seen yet are left white.

$T_{i+1} \coloneqq T_i \cap C_i \subseteq T_i$, and $B_{i+1} \coloneqq C_i \setminus T_i$, such that $T_{i+1} \cup B_{i+1} = C_i$. This implies that $u^* \in B_{i+1}$, and thus the new distinguished edge e_{i+1} stays the same edge as before, $e_{i+1} \coloneqq e_i = \{u_{j^*}, u_{j^*+1}\}$, such that invariant (a) is trivially fulfilled. As no costs were booked to any edges in this round, also invariant (b) remains true. Invariant (c) holds since the agent did not move at all in this round.

If, on the other hand, $|T_i \cup B_i| < c/3$, the agent executes a traversal round. Hence, this is the last round of phase p, i. e., round $h + m$. Right now, due to invariant (c), $\mathcal{A}$ is positioned at some vertex $v \in T_{h+1}$ since the last traversal round was round h. Hence, before $\mathcal{A}$ starts to traverse $T_{h+m} \cup B_{h+m}$, it has to make sure to be positioned somewhere in this set. We determine that the agent moves to some vertex $v' \in T_{h+m}$. Since all rounds $h + 1, \ldots, h + m - 1$ have been advice rounds, we have $T_{h+m} \subseteq T_{h+m-1} \subseteq \ldots \subseteq T_{h+2} \subseteq T_{h+1}$. As round h was a traversal round, we know that $|T_{h+1}| = |T_h \cup B_h| < c/3$, and thus traveling from v to v' incurs costs of less than $c/3$. These costs are booked onto the distinguished edge $e_{h+m} = \{u_{j^*}, u_{j^*+1}\}$ (which exists due to invariant (a)).

Now, the agent uses a depth-first search to traverse $T_{h+m} \cup B_{h+m}$. As soon as it comes across the destination vertex u_ℓ, the algorithm terminates. Let us hence assume that $\mathcal{A}$ does not find u_ℓ in this round, i. e., $u_\ell \notin T_{h+m} \cup B_{h+m}$. The traversal incurs costs of less than $2c/3$ that are also booked to e_{h+m}. The agent sets $T_{h+m+1} \coloneqq T_{h+m} \cup B_{h+m}$, since all vertices contained in these sets have been traversed now but have not been harvested yet, and $H_{h+m+1} \coloneqq H_{h+m}$, as no vertices have been harvested in this round. The agent also computes B_{h+m+1} according to T_{h+m+1} and H_{h+m+1}. From invariant (a) we can conclude that one endpoint of the present distinguished edge e_{h+m} must be $u_{j^*} \in B_{h+m}$ and the other one $u_{j^*+1} \notin (H_{h+m+1} \cup T_{h+m+1} \cup B_{h+m+1})$, since otherwise u_{j^*} would not have been the distinguished vertex in round $h + m$. Thus, $u_{j^*+1} \in B_{h+m+1}$ is a vertex that guarantees that invariant (a) is fulfilled. However, u_{j^*+1} is not necessarily the distinguished vertex of the next round; this is the last vertex $u_k \in U$ that is in B_{h+m+1}, for some $k \geq j^*+1$. Then, the new distinguished edge $e_{h+m+1} = \{u_k, u_{k+1}\}$ is the edge where the path U leaves B_{h+m+1} for the last time. Thus, invariants (a) and (b) remain true. At the end of round $h + m$, and therefore also at the beginning of phase $p + 1$, the agent is located at some vertex $v \in T_{h+m} \cup B_{h+m} = T_{h+m+1}$, making sure that also invariant (c) holds.

At some point, there will be a round in which the agent sees the identifier of the destination vertex u_ℓ for the first time, as a neighbor to a vertex v it just visited. Then, $\mathcal{A}$ immediately interrupts its traversal and moves from v to u_ℓ. This might incur additional costs of one in case the traversal was completed in the exact moment that the agent reached v. In this case, there is one cost unit that we cannot book to any edge and that we have to keep in mind.

For every i with $0 \leq i \leq \ell - 1$, every edge $\{u_i, u_{i+1}\}$ is the distinguished edge of at most one phase, and thus costs of at most c are booked to it; not more than $c/3$ for the adjustment move before the traversal of the current search space and at most $2c/3$ for the traversal itself. Hence, the total cost incurred during the computation of $\mathcal{A}$ on G, adding the additional cost for the last move, is at most $\text{cost}(\mathcal{A}(G)) \leq c \cdot \ell + 1$, whereas the cost of an optimal solution is $\text{cost}(\text{Opt}(G)) = \ell$. Choosing $a \geq 1$ as the constant in the definition of the competitive ratio (see Section 1.4) yields

$$\text{cost}(\mathcal{A}(G)) \leq c \cdot \ell + 1 = c \cdot \text{cost}(\text{Opt}(G)) + 1 \leq c \cdot \text{cost}(\text{Opt}(G)) + a,$$

and thus the algorithm $\mathcal{A}$ is c-competitive. Furthermore, each vertex is harvested at most once, so the credit units that $\mathcal{A}$ invests during the entire computation to read advice bits cannot exceed $9/c \cdot n$. This corresponds to the number of advice bits read. As a result, $\mathcal{A}$ has an advice complexity of at most $9n/c$. □

4.3 Graph Exploration

Now we turn our attention to the graph exploration problem, which is, like the graph searching problem SEARCH we considered in the previous section, not a classic online problem since the request sequence depends on the given algorithm. Let us briefly recall the graph exploration problem. Given a graph and a starting vertex at which an agent is located at the beginning of the computation, the goal of an algorithm for EXPLORE is to visit each vertex at least once and then return to the starting vertex. We again constrain our considerations to undirected unweighted graphs. In Section 4.3.1, we describe a class of undirected unweighted graphs that has already been investigated in the literature. Brandstädt et al. [BLS99] call this graph class *sunlet graphs*, whereas Anitha and Lekshmi [AL08] and Wallis [Wal01] refer to them as *sun graphs*. We will stick to the naming of the latter two. Later, we will use this class to prove a lower bound on the number of advice bits necessary to achieve any competitive ratio c in the range $1 \leq c \leq 7/6 - \varepsilon$, for any constant $\varepsilon > 0$. In Section 4.3.2, first the general reduction technique used to obtain this lower bound is introduced. This method also makes use of a reduction from the string guessing problem, as we have already seen in preceding chapters, but the familiar technique has to be adapted slighty to be applicable to this non-standard online problem. Then this reduction technique is applied to the graph exploration problem, yielding a lower bound of $(1 - \eta(1 - \alpha))n$ advice bits necessary to achieve any competitive ratio c with $c \leq (4 - \alpha)/3 - \varepsilon$, for any constant $\varepsilon > 0$ and any α with $1/2 \leq \alpha < 1$. This implies that to obtain a competitive ratio of c in the range $1 \leq c \leq 7/6 - \varepsilon$, for every constant $\varepsilon > 0$, a linear number of advice bits must be read. Very recently, another lower bound has been independently obtained by Fulla [Ful14], who showed that $\Omega(n)$ advice bits are necessary to achieve a competitive ratio better than 1.564. The results presented in this section are discussed rather due to the novel reduction technique that is being used to obtain them.

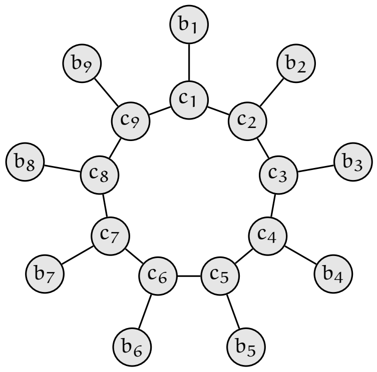


Figure 4.4. The sun graph $G^{(n)}$ for $n = 9$.

4.3.1 Sun Graphs

For every natural number $n \in \mathbb{N}^{\geq 5}$, let us consider the following unweighted undirected graph $G^{(n)} = (V^{(n)}, E^{(n)})$ that consists of n so-called *cycle vertices* c_i arranged in a cycle, and n so-called *beam vertices* b_i, each of which is connected to one cycle vertex and no additional vertices. To avoid some dreadful notation involving the modulo operator, let us define $c_{n+i} := c_i$ and $b_{n+i} := b_i$ for every i with $1 \leq i \leq n$. Hence, formally we have

$$\begin{aligned} V^{(n)} &= \{c_1, \dots, c_n, b_1, \dots, b_n\}, \\ E^{(n)} &= \big\{\{c_i, c_{i+1}\} \mid 1 \leq i \leq n\big\} \cup \big\{\{c_i, b_i\} \mid 1 \leq i \leq n\big\}. \end{aligned}$$

An example of such a sun graph for $n = 9$ is depicted in Figure 4.4. To avoid confusion, let us stress again that the naming is not completely consistent in the literature; sometimes, the term *sun graph* is used to refer to another graph class, for example by Brandstädt et al. [BLS99] as already mentioned above.

Observation 4.6. *The optimal solution on a sun graph $G^{(n)}$ with starting vertex c_1 has a cost of* $\text{cost}\big(\text{Opt}(G^{(n)})\big) = 3n$.

Proof. Let us recall that the agent does not only have to visit all vertices, but has to be positioned at the starting vertex again in the end. Thus, in any valid solution, every edge between two cycle vertices must be traversed at least once (except for one edge that might be omitted, but obviously, this would only increase the total costs). Hence, the costs incurred by traversing these edges is at least n. Every edge between one cycle vertex and one beam vertex must be traversed at least twice, incurring costs of at least $2n$. Hence, the cost of any optimal solution is at least $3n$.

On the other hand, given c_1 as the starting vertex, an optimal way to traverse such a sun graph $G^{(n)}$ is clearly $(c_1, b_1, c_1, c_2, b_2, c_2, \dots, c_n, b_n, c_n, c_1)$. This solution has a cost of $3n$, which is therefore optimal. □

When an agent traverses the graph, though, it does not necessarily take the optimal route. Quite the contrary, it might visit each vertex arbitrarily often and also change directions arbitrarily. However, intuitively, it is obvious that an algorithm achieving a good competitive ratio should not revisit the same vertices too often and, therefore, visit each beam vertex b_i as soon as possible, preferably when the corresponding cycle vertex c_i is visited for the first time.

For the reduction in the following section, we exploit the fact that an agent that is located in some cycle vertex c_i cannot tell the corresponding beam vertex b_i apart from the yet unvisited neighboring cycle vertex, given no further information than the IDs of these vertices. We will make sure that any algorithm, without getting any advice bits at all, might make the wrong decision at every cycle vertex c_i and move to another cycle vertex from there instead of moving to b_i. Note that if this happens, the algorithm might still achieve a competitive ratio of $4/3$, as we will see now.

Observation 4.7. *There is a deterministic online algorithm $\mathcal{A}$ for* EXPLORE *that achieves a strict competitive ratio of* $4/3$ *on any sun graph* $G^{(n)}$.

Proof. The algorithm $\mathcal{A}$ operates in two phases. In the first one, as long as there exists at least one neighbor of the current vertex which has not been visited yet, let $\mathcal{A}$ do the following. When located at a cycle vertex, the next vertex that $\mathcal{A}$ visits is the unique yet unvisited vertex with the lowest ID. When located at a beam vertex (which $\mathcal{A}$ can recognize based on its degree), $\mathcal{A}$ moves back to the corresponding cycle vertex. Obviously, $\mathcal{A}$ visits all cycle vertices either in clockwise or in anti-clockwise order, but never changes the direction during this phase. Without loss of generality, let $\mathcal{A}$ move in clockwise direction in the first phase. Then, when this phase is over, $\mathcal{A}$ is located at c_n, has visited each cycle vertex exactly once, and has additionally visited m beam vertices, for some m with $0, \ldots, n$. Anyway, $\mathcal{A}$ has visited or at least seen each vertex after phase 1, such that it now knows the whole graph.

In the second phase, $\mathcal{A}$ visits every cycle vertex once more, this time in anti-clockwise direction, ending at c_1. On its way, it visits each yet unvisited beam vertex b_i of each cycle vertex c_i that it passes. In the end, $\mathcal{A}$ is located at c_1 again.

Now we analyze $\mathcal{A}$'s cost on $G^{(n)})$ separately for both phases. In the first phase, $\mathcal{A}$ incurs a cost of $n - 1 + 2m$; in the second phase, $2(n - m) + n - 1$. The overall costs are therefore $n-1+2m+2n-2m+n-1 = 4n-2$. According to the definition of the strict competitive ratio given in Section 1.3, $\mathcal{A}$ is strictly $4/3$-competitive on the class of sun graphs if

$$\operatorname{cost}\Big(\mathcal{A}(G^{(n)})\Big) \leq \frac{4}{3} \cdot \operatorname{cost}\Big(\mathrm{Opt}(G^{(n)})\Big)$$

for any sun graph $G^{(n)}$. This is obviously the case, as

$$\mathrm{cost}\Big(\mathcal{A}(G^{(n)})\Big) = 4n - 2 \leq 4n = \frac{4}{3} \cdot 3n = \frac{4}{3} \cdot \mathrm{cost}\Big(\mathrm{Opt}(G^{(n)})\Big),$$

where the last equality holds due to Observation 4.6. □

From now on, let us say that the agent enters a vertex v when it travels to v from one of its neighbors. Hence, starting in c_1 does not count as entering it, but revisiting it later on does, of course. The following lemma gives some information on the prize an agent has to pay for not traveling to a beam vertex b_i right after the corresponding cycle vertex c_i has been entered for the first time.

Lemma 4.8. *For any sun graph $G^{(n)} = \big(V^{(n)}, E^{(n)}\big)$ and every cycle vertex $c_i \in V^{(n)}$, the following holds. When the agent enters c_i for the first time and directly after that travels to another cycle vertex without having visited the corresponding beam vertex b_i, the cost of the solution calculated by the corresponding algorithm increases by at least 1 compared to the optimal solution.*

Proof. Every time the agent enters a vertex c_i or b_i, let us book a cost of 1 to this vertex. Obviously, at the end of the computation, the total cost booked to all vertices is the cost of the algorithm's computed solution. Also, clearly, in an optimal solution, every beam vertex is entered exactly once and every cycle vertex exactly twice, once from a neighboring cycle vertex and once from its corresponding beam vertex. Conversely, if a cycle vertex c_i is entered and then left without having visited the corresponding beam vertex b_i yet, the cycle vertex must still be entered at least two more times in any valid solution. Hence, in the end, c_i will have a cost of at least 3 booked to it, increasing the total cost of the solution by at least 1. □

4.3.2 Reduction from String Guessing

Before we get to the reduction from 2-GUESS to EXPLORE, we briefly describe how to proceed in general when giving a reduction from 2-GUESS to an online problem like EXPLORE, for which the request sequence depends on the output of the algorithm. Let P be such an online problem, and let $\mathcal{A}$ be an algorithm for P. To reduce the bit string guessing problem to P, we construct a 2-GUESS-algorithm $\mathcal{B}$ by simulating the algorithm $\mathcal{A}$. Let the input instance for 2-GUESS be $I_r = (n, r_1, \ldots, r_n)$ for some binary string r. Hence, from I_r, we must now construct an input instance for P to simulate $\mathcal{A}$ on it. This is where the reduction technique has to differ from the one we are used to. Usually, we would now construct an input for P that is independent of the algorithm $\mathcal{A}$, but since P is an online problem for which the input must depend on $\mathcal{A}$, we have to proceed differently here. We construct an input $I_{\mathcal{A},r}$ for P, depending on r and $\mathcal{A}$. Then, the 2-GUESS-algorithm $\mathcal{B}$ simulates $\mathcal{A}$ on $I_{\mathcal{A},r}$ in an online fashion, and $\mathcal{A}$ produces some output $\mathcal{A}(I_{\mathcal{A},r})$, from which $\mathcal{B}$ derives its own output for every round. The goal is to show a connection between the performances of $\mathcal{A}$ and $\mathcal{B}$, such that a statement like the following can be obtained. For any string r and any algorithm $\mathcal{A}$ for P, if $\mathcal{A}$ achieves a competitive ratio of c on $I_{\mathcal{A},r}$ reading only b advice bits, then there is an algorithm for 2-GUESS that guesses at least $f_1(c)$ bits correctly, reading only $f_2(b)$ advice bits, for some functions f_1, f_2. If such a relation between the algorithms can be shown, the lower bound for 2-GUESS (Fact 1.5) directly translates to a lower bound for P.

Let us elaborate on the concrete reduction to EXPLORE now. Although we do not know the exact walk that an agent takes through the graph $G^{(n)}$, we know that any algorithm solving the graph exploration problem must have visited every vertex at least once in the end. Hence, we know that, sooner or later, the following situation occurs. At some point (while it has not visited all cycle vertices yet) the agent travels to a so far unvisited cycle vertex, and it can see a neighboring vertex from there that it has already seen before as a neighbor of a different cycle vertex than the one it is currently located at. Let c_{i^*} be the first cycle vertex that the agent sees as a neighbor of two different cycle vertices. Without loss of generality, let us assume that this situation occurs while the agent is traveling in clockwise direction. Hence, it must already have visited c_{i^*+1} and have seen c_{i^*} from there, but then it never traveled to c_{i^*} before visiting c_{i^*-1} and seeing c_{i^*} for the second time from there. This situation is depicted in Figure 4.5.

From now on, we will always make this assumption that the agent encounters c_{i^*} for the second time while traveling in clockwise direction. If this happens while traveling in anti-clockwise direction, all following considerations can be made analogously.

Now, to reduce 2-GUESS to EXPLORE, we proceed as follows. For one thing, we transform an input instance I_r for 2-GUESS into an input for EXPLORE that depends on r and on the walk that the graph exploration algorithm $\mathcal{A}$ takes through the graph. For another thing, we present a way to transform the output of $\mathcal{A}$ into an output for the string guessing problem. Thus consider, for some $n \in \mathbb{N}^{\geq 5}$ and some string $r \in \{0,1\}^n$, an input instance $I_r = (n, r_1, \ldots, r_n)$ for 2-GUESS. We generate a graph exploration instance $G^{(n)}_{\mathcal{A},r}$ with $2n$ vertices by assigning the IDs that are presented to $\mathcal{A}$ to the vertices of the graph $G^{(n)}$ defined above, depending on the string $r = r_1 \ldots r_n$ and on the way that $\mathcal{A}$ has traversed the graph so far. This assignment of the IDs to the vertices is crucial since a deterministic algorithm can only distinguish vertices on the basis of their IDs.

In the following paragraphs, our aim is to show a connection between the agent's walk through the graph and the performance of the algorithm $\mathcal{B}$ for 2-GUESS constructed from $\mathcal{A}$. More precisely, we want to show that, if $\mathcal{A}$ visits the

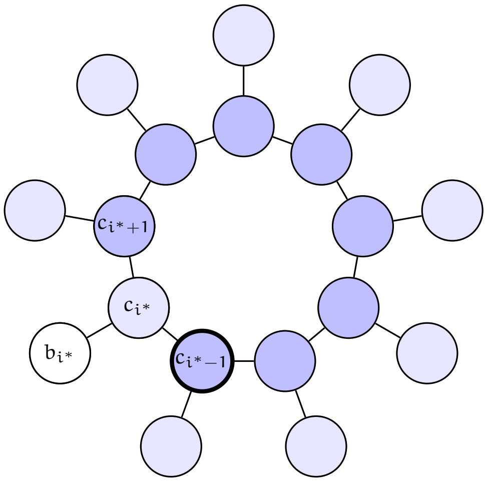


Figure 4.5. The agent just traveled from c_{i^*-2} to c_{i^*-1} (marked by a thick fringe). It sees the vertex c_{i^*}, which it has already seen before when it was located at c_{i^*+1}, but has not entered it yet. The dark vertices have already been visited. The light blue vertices have at least been seen as a neighbor of an already visited vertex. For the light blue beam vertices, we do not know if they have already been visited or not, but for c_{i^*} we know that it definitely has not. The white vertex, b_{i^*}, has not even been seen yet.

corresponding beam vertex b_i directly after visiting c_i for the first time for many cycle vertices c_i, then $\mathcal{B}$ guesses many bits correctly. To this end, we now describe the assignment of IDs to the vertices of $G^{(n)}$ as well as the algorithm $\mathcal{B}$.

As before, we fix the cycle vertex c_1 as the starting vertex of the graph $G^{(n)}_{\mathcal{A},r}$. Hence, in the beginning of the computation, the agent is located at c_1 and must be given the IDs of c_1 and its three neighbors c_n, c_2, and b_1. We assign IDs to these vertices as follows; ID 0 to c_1, ID 1 to c_n, ID 2 to b_1, and ID 3 to c_2, as depicted in Figure 4.6.

Usually, while traversing the graph, at every newly visited cycle vertex, the agent must be presented the IDs of the two yet unknown neighbors, one being the corresponding beam vertex and the other one a neighboring cycle vertex. One exception to this is, as we have just discussed, the starting vertex c_1, at which the agent is presented not only two, but four IDs. The second exception is c_{i^*-1}, where only one new ID must be presented, namely the one of the corresponding beam vertex b_{i^*-1}, because the ID of c_{i^*} has already been presented at c_{i^*+1} (see Figure 4.7a). The last exception is the vertex c_{i^*}, where also only one ID must be presented, namely the one of b_{i^*}, because both neighboring cycle vertices have already been visited (see Figure 4.7b).

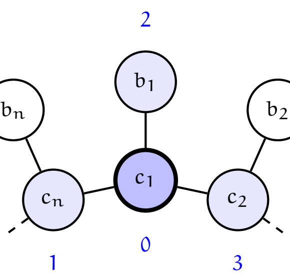
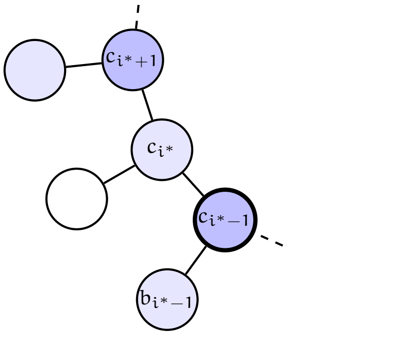


Figure 4.6. The IDs of the first four vertices that the agent sees from the starting vertex c_1 in the graph $G_r^{(n)}$.

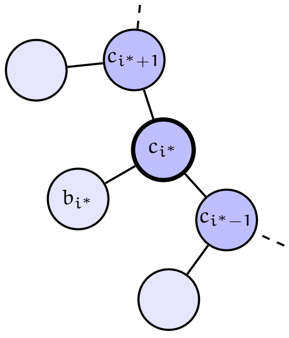
Let us from now on consider any other cycle vertex c_i with $i \in \{1,\ldots,n\} \setminus \{1, i^* - 1, i^*\}$. Assume that c_i is the j-th newly visited cycle vertex, for some j with $2 \le j \le n-2$. (Note that the first newly visited cycle vertex is c_1 and the last two newly visited cycle vertices are c_{i^*-1} and c_{i^*}, in this order.) Then, upon visiting c_i for the first time, the agent is presented the IDs $2j$ and $2j+1$. At every such vertex c_i, we know that the agent sooner or later travels to a neighboring vertex with one of these two IDs. When this happens for the first time, we say that the agent chooses the vertex that it traveled to from c_i as the successor of c_i. Only when the agent actually visits one of these two vertices, we have to determine which ID belongs to the cycle and which to the beam vertex. We make this decision depending on the string r. Let $c_i \in C := \{c_1,\ldots,c_n\} \setminus \{c_1, c_{i^*-1}, c_{i^*}, c_{i^*+1}\}$ be the k-th cycle vertex for which the agent chooses a successor. Note that the order in which the successors are chosen for the cycle vertices might deviate from the order in which they are visited for the first time. This is because the agent might visit c_i, but then decide to travel back into the other direction until it reaches another cycle vertex for which it has not chosen a successor yet and do so now, before it chooses the successor for c_i. As already mentioned, which of the IDs $2j$ and $2j+1$ belongs to the neighboring unvisited cycle vertex of c_i and which one to the neighboring unvisited beam vertex is dependent on the string r. If $r_k = 0$, the beam vertex has ID $2j$ and the cycle vertex has the ID $2j+1$; if $r_k = 1$, it is the other way around.

The string guessing algorithm $\mathcal{B}$ we build from the graph exploration algorithm $\mathcal{A}$ then bases its guess g_k for the bit r_k on the ID of the vertex that the agent chooses as a successor for c_i. If $\mathcal{A}$ chooses the vertex with ID $2j$, the algorithm $\mathcal{B}$ outputs $g_k := 0$; if, otherwise, $\mathcal{A}$ chooses the vertex with ID $2j+1$, the 2-Guess algorithm $\mathcal{A}$ outputs $g_k := 1$. This leads to the following coherence.

Lemma 4.9. *Consider an algorithm $\mathcal{A}$ for the graph exploration problem, and an input instance $I_r = (n, r_1, \ldots, r_n)$ for the string guessing problem. Furthermore, consider an algorithm $\mathcal{B}$ for the string guessing problem constructed from $\mathcal{A}$ as in the reduction we*



(a) The agent just traveled from c_{i^*-2} to c_{i^*-1} (marked by a thick fringe). The vertex c_{i^*+1} has already been visited before, so the ID of their common neighbor c_{i^*} has already been presented to $\mathcal{A}$. Hence, at the vertex c_{i^*-1}, the algorithm is only presented one new ID, the one of b_{i^*-1}.



(b) The agent is currently located at c_{i^*} (marked by a thick fringe). Both neighboring cycle vertices c_{i^*-1} and c_{i^*+1} have already been visited before, so their IDs have already been presented to $\mathcal{A}$. The only new ID that is presented to $\mathcal{A}$ is the one of c_{i^*}'s corresponding beam vertex, b_{i^*}.

Figure 4.7. Two exceptions for the two-new-IDs-per-cycle-vertex-rule.

described above, and a cycle vertex $c_i \in C$ with two yet unvisited neighbors with IDs $2j$ and $2j+1$, respectively, that is the k-th cycle vertex for which the agent $\mathcal{A}$ chooses a successor, for $2 \leq k \leq n-2$. Then, if $\mathcal{A}$ chooses the beam vertex b_i as c_i's successor, $\mathcal{B}$ guesses the bit r_k correctly.

Proof. To travel from c_i to the beam vertex b_i, the agent must choose the vertex with ID $2j$ as c_i's successor if $r_k = 0$ and, on the other hand, choose the vertex with ID $2j+1$ as the successor if $r_k = 1$. Hence, if $\mathcal{A}$ travels to the beam vertex, $\mathcal{B}$ outputs $g_k := 0$ if $r_k = 0$ and $g_k := 1$ if $r_k = 1$. Therefore, in both cases, it guesses the bit r_k correctly. □

Corollary 4.10. *Assume there is an algorithm $\mathcal{A}$ for the graph exploration problem that, given any sun graph with $2n$ vertices as its input, chooses the beam vertex b_i as the successor for c_i for more than αn cycle vertices $c_i \in C$, reading only b advice bits. Then, there is also a string guessing algorithm $\mathcal{B}$ that guesses more than αn bits correctly on any input instance I_r corresponding to some string $r \in \{0,1\}^n$, reading only b advice bits.*

Proof. From I_r and $\mathcal{A}$, we construct the graph $G^{(n)}_{\mathcal{A},r}$ as described in the reduction given above. We then simulate the algorithm $\mathcal{A}$ on the instance $G^{(n)}_{\mathcal{A},r}$ with c_1 as the starting vertex. We also construct the string guessing algorithm $\mathcal{B}$ as in the reduction given above, basing its guesses on the way the agent $\mathcal{A}$ traverses the graph. Additionally, as we have already seen a few times before, $\mathcal{B}$ keeps a dummy advice tape that $\mathcal{A}$ can access during its computation. Every time $\mathcal{A}$ needs some advice bits, $\mathcal{B}$ reads the corresponding number of advice bits from the oracle's advice tape and writes them onto its dummy tape, which are then read by $\mathcal{A}$.

Due to Lemma 4.9, for every cycle vertex $c_i \in C$, the algorithm $\mathcal{B}$ guesses the bit r_k correctly if $\mathcal{A}$ chooses the beam vertex b_i as the successor for c_i. Hence, if $\mathcal{A}$ chooses the corresponding beam vertices as the successors for more than αn cycle vertices, $\mathcal{B}$ guesses more than αn of n bits correctly.

Obviously, the numbers of advice bits that the algorithms $\mathcal{A}$ and $\mathcal{B}$ use are exactly the same. □

Lemma 4.11. *For any α with $1/2 \leq \alpha < 1$, any constant $\varepsilon > 0$, any string $r \in \{0,1\}^n$, and any algorithm $\mathcal{A}$ for* Explore*, the following holds. If $\mathcal{A}$ computes a solution with a competitive ratio of $c \leq (4-\alpha)/3-\varepsilon$ on the graph $G^{(n)}_{\mathcal{A},r}$, it must choose the corresponding beam vertex b_i as the successor for c_i for more than αn cycle vertices $c_i \in C$.*

Proof. Assume there is an algorithm $\mathcal{A}$ that achieves a competitive ratio of c on the sun graph $G^{(n)}_{\mathcal{A},r}$. Then, due to the definition of the competitive ratio, there must exist some constant a such that

$$\mathrm{cost}\Big(\mathcal{A}\Big(G^{(n)}_{\mathcal{A},r}\Big)\Big) \leq c \cdot \mathrm{cost}\Big(\mathrm{Opt}\Big(G^{(n)}_{\mathcal{A},r}\Big)\Big) + a.$$

Plugging in $c \leq (4-\alpha)/3-\varepsilon$ and using the fact that the cost of any optimal algorithm Opt on any sun graph $G^{(n)}$ is $3n$ due to Observation 4.6, we make the following transformation.

$$\begin{aligned}\mathrm{cost}\Big(\mathcal{A}\Big(G^{(n)}_{\mathcal{A},r}\Big)\Big) &\leq \left(\frac{4-\alpha}{3}-\varepsilon\right)\cdot 3n + a\\ &= (4-\alpha)n - 3\varepsilon n + a\\ &= \mathrm{cost}\Big(\mathrm{Opt}\Big(G^{(n)}_{\mathcal{A},r}\Big)\Big) + (1-\alpha)n - (3\varepsilon n - a)\\ &< \mathrm{cost}\Big(\mathrm{Opt}\Big(G^{(n)}_{\mathcal{A},r}\Big)\Big) + (1-\alpha)n - 4,\end{aligned}$$

for sufficiently large n. Hence, the cost of the solution computed by $\mathcal{A}$ may only differ from the optimal solution by less than $(1-\alpha)n-4$ in order to be $((4-\alpha)/3-\varepsilon)$-competitive. Due to Lemma 4.8, the cost of the computed solution increases by at least 1 compared to the optimum for every cycle vertex c_i at which the agent does not choose the beam vertex b_i as the successor. Thus, there may only be less than $(1-\alpha)n-4$ such cycle vertices at which it chooses another cycle vertex as the successor. Therefore, the number of cycle vertices c_i at which the agent chooses the corresponding beam vertex as c_i's successor must be more than $\alpha n + 4$. More than αn of these cycle vertices must be from C (since there are exactly four cycle vertices that are not contained in C), which completes the proof. □

Theorem 4.12. *For any* α *with* $1/2 \leq \alpha < 1$ *and any constant* $\varepsilon > 0$*, there is no online algorithm for* EXPLORE *with a competitive ratio of* $c \leq (4 - \alpha)/3 - \varepsilon$ *reading less than* $b = \big(1 - \eta(1 - \alpha)\big)\, n$ *advice bits.*

Proof. For the sake of contradiction, assume there is such a graph exploration algorithm $\mathcal{A}$. Hence, for some constant $\varepsilon > 0$, this algorithm must compute a solution with a competitive ratio of $c \leq (4 - \alpha)/3 - \varepsilon$ on any input instance, in particular also on each sun graph $G^{(n)}$. Thus, according to Lemma 4.11, $\mathcal{A}$ chooses the corresponding beam vertex b_i as the successor for c_i for more than αn cycle vertices $c \in C$. However, due to Corollary 4.10, if there is such a graph exploration algorithm reading less than $b = \big(1 - \eta(1 - \alpha)\big)\, n$ advice bits and choosing the corresponding beam vertex b_i as the successor for c_i for more than αn cycle vertices $c_i \in C$, then there also is a string guessing algorithm that guesses more than αn bits correctly on any input string of length n, reading less than b advice bits. Then again, as we know from Fact 1.5, such a string guessing algorithm does not exist, and thus our initial assumption must have been false. □

4.3.3 Further Adjustments

When thinking a bit further about the idea we used in the previous section, a question that naturally comes to mind is whether adjusting the lengths of the beams or the paths between each two beams in the examined sun graphs might result in a graph class that yields results for wider ranges of competitive ratios. Hence, let us investigate generalized sun graphs with kn cycle vertices, of which every k-th one has a beam attached to it (hence, the number of beams is n), and the length of each beam is ℓ. For each pair $(\ell, k) \in \mathbb{N}^2$, let us call this class of graphs (ℓ, k)-*sun graphs*. The sun graphs examined in Section 4.3.2 are $(1, 1)$-sun graphs, accordingly. In the following, let us briefly and informally reason about generalized sun graphs and explain why we chose the simple class of sun graphs for the reduction in Section 4.3.2. In the previous section, we knew from the beginning that we could not hope to obtain a lower bound on the advice complexity for competitive ratios larger than 4/3 since we showed that there is a 4/3-competitive deterministic algorithm on sun graphs with n cycle vertices and n beam vertices. However, there might exist some class of (ℓ, k)-sun graphs with adjusted values of ℓ and k as described above, for which there are no deterministic algorithms with that good competitive ratios, such that we might obtain lower bounds on the advice complexity for larger competitive ratios than 4/3 with the same approach. Let us look at this more closely. On any (ℓ, k)-sun graph, the optimal solution obviously has a cost of $(2\ell + k)n$. Furthermore, in the following, we describe a deterministic algorithm that has a cost of $(2\ell + 2 + k)n$ on any (ℓ, k)-sun graph. Located at some cycle vertex c_{ki} with an adjacent beam vertex b_i, the algorithm visits some yet unvisited neighbor. In case it is the wrong one (i. e., another cycle vertex), it immediately notices its error based on the degree of the current vertex and moves back to c_{ki}. From there, it moves to b_i, traverses the beam up to its extremal vertex, and then moves back to c_{ki}. Up to now, this generated costs of $2\ell + 2$. The agent can now move on a direct path to the next cycle vertex with an appended beam, incurring a further cost of k. In total, the cost of this deterministic algorithm is $(2\ell + 2 + k)n$.

Thus, for any class of (ℓ, k)-sun graphs, there is a deterministic algorithm with a competitive ratio of at most $(2\ell + 2 + k)/(2\ell + k) = 1 + 2/(2\ell + k)$. This term is maximized for $(\ell, k) = (1, 1)$. However, for $(1, 1)$-sun graphs, the deterministic algorithm given in the previous section has competitive ratio 4/3, which is even better than $(2+2+1)/(2+1) = 5/3$. As both ℓ and k are natural numbers, the next largest value is attained for $(\ell, k) = (1, 2)$, and this value is $(2 + 2 + 2)/(2 + 2) = 3/2$; for all other pairs (ℓ, k), there is a deterministic algorithm that achieves a competitive ratio of less than 3/2 on any (ℓ, k)-sun graph. Therefore, even for generalized sun graphs, we cannot prove a lower bound on the advice complexity for competitive ratios larger than 3/2. However, Fulla already presented a lower bound on the advice complexity for competitive ratios of approximately 1.56 using a more complicated class of graphs as inputs [Ful14]. By such simple adaptations as we just described, we do not gain enough to beat this lower bound. However, our main goal was not to achieve the best possible lower bound for EXPLORE anyway, but to demonstrate the application of the adapted reduction technique. Thus, for the sake of clarity of presentation, we used $(1, 1)$-sun graphs in the reduction from the previous section, although we might have obtained a slightly better lower bound by choosing, for example, $(1, 2)$-sun graphs.

5 Probabilistic Adversary

In previous chapters, we have always considered a setting in which the adversary chooses a single instance that is given to the online algorithm as its input, and the oracle knows the complete input. In this chapter, we analyze a novel probabilistic model, in which the adversary specifies a set of possible inputs, from which the actual input for the algorithm is chosen uniformly at random. The oracle knows this set of possible inputs, but does not have access to the random bits used by the adversary. In this setting, we consider the string guessing problem.

The reason why we study this new model is that the all-knowing oracle is, intrinsically, only a theoretical concept and as such not implementable. Making the oracle less powerful is an attempt to design a more realistic model. By strengthening the adversary as described above, we obtain an oracle that is not omniscient anymore, but only knows a probability distribution according to which the input for the algorithm is chosen by the adversary. Such an oracle could, for example, represent knowledge that we have a priori about typical input instances for some online problem.

A model involving a probabilistic adversary has already been considered by Wehner [Weh14]. The author also adapts the game between the three parties in such a way that the adversary does not only construct one input sequence for the algorithm, but a set of input sequences, and a probability distribution according to which the actual input instance is chosen. In this setting, the author analyzes a variant of the job shop scheduling problem (for example, see Hromkovič et al. [HMSW07]) and shows that the problem is more difficult with this strengthened adversary than in the classical adversary model.

Recall that we can model the classical scenario of online computation by a game between two parties; the adversary and the online algorithm. In the scenario of online computation with advice, the algorithm is supported by a third party, an all-knowing oracle. Before the computation of the algorithm starts, the adversary constructs a hard input instance, knowing the oracle and the algorithm. Then, the oracle inspects this input instance and writes some binary string onto its infinite advice tape. Only after that, the algorithm starts its computation, and during computation, it may access the advice tape whenever it desires and read as many advice bits as it wants. This way, the algorithm computes an output for the input instance using the advice string. This is the setting we have considered in previous chapters.

In the classical online scenario without advice, providing random bits to the adversary does not increase the power of the adversary in any way since, for any given online algorithm, no set of problem instances can provide an expected competitive ratio that is worse than the competitive ratio on a worst-case-instance. Hence, when analyzing the competitive ratio in online computation without advice, usually only deterministic adversaries are considered.

Naturally, the question arises whether the situation is the same in online computation with advice, or if randomization might help the adversary in this case. Actually, in the scenario with an oracle, the situation is completely different. In this setting, if the adversary generates a set of hard input instances and then chooses one of them at random, the oracle only knows the set of generated inputs and the probability distribution according to which the actual input is chosen, but not the actual input itself. In this case, the competitive ratio of the given online algorithm is the expected competitive ratio with respect to the probability distribution over the input set chosen by the adversary. This means that, for most online problems, we cannot guarantee to reach optimal solutions, even when knowing the complete set of hard instances.

Hence, in this chapter, we consider a generalization of the classical scenario of online computation with advice in which we make the adversary more powerful by allowing it to use a source of random bits. In this thesis, we restrict ourselves to the case that the adversary chooses a set of input instances and then one of these instances is chosen at random with respect to the uniform probability distribution. This game with a randomized adversary can be played in two different ways. In the first version, the oracle offers a sequence of advice bits to the online algorithm only once before the online computation starts. We will call this model the *monolog model*. The second version, which we will call the *dialog model*, allows the oracle to provide advice bits after each request. This is the model that has already been analyzed by Wehner [Weh14] for job shop scheduling. Although it might seem astonishing at first, the choice of the model can actually make a huge difference. This is due to the fact that, after each request, the oracle might have learned some of the random bits used by the adversary to pick an instance at random from the set of hard instances; hence, being able to provide advice after each request increases the power of the oracle tremendously. We will witness this in this chapter.

In the following sections, we consider the scenario of online computation with advice with a probabilistic adversary in the two models described above. In this setting, we analyze the bit string guessing problem. For strings of length n, the set of possible input strings for an online algorithm for 2-GUESS contains 2^n strings. We consider the case that the adversary excludes $\tilde{k}$ of these strings and chooses one of the remaining ones uniformly at random as the actual input string for the algorithm. As we have already mentioned, in this case, no algorithm can compute the optimal solution in general, even with the help of an oracle. However, we determine the maximal number of bits an online algorithm with advice can guess correctly in expectation when the adversary excludes $\tilde{k}$ strings, and give almost matching lower and upper bounds for the number of advice bits necessary and sufficient to achieve this optimal number of correctly guessed bits in both models.

The remainder of this chapter is structured as follows. In Section 5.1, we present formal definitions of the two different advice models and make a few arrangements concerning our notation. Section 5.2 deals with the first of the two models, the monolog model. For this model, we present an algorithm with advice achieving the optimal number of correctly guessed bits in expectation that reads at most $\lceil \tilde{k} \cdot (n - \log \tilde{k} + \log e) \rceil$ advice bits. Moreover, we give an almost matching lower bound of at least $\tilde{k} \cdot (n - \log \tilde{k})$ advice bits that are necessary for any algorithm to guess this many bits correctly. Section 5.3 deals with the dialog model, and we give an upper bound of n advice bits to achieve the optimal expected number of correctly guessed bits. For the case that the set of possible input instances consists of an odd number of strings, we also give a tight lower bound of n advice bits necessary. The differences between the two models from Sections 5.2 and 5.3 are briefly discussed in Section 5.4. In Section 5.5, we show that the lower bounds we proved for the bit string guessing problem in the two probabilistic models can be translated to lower bounds for other online problems by making reductions as we are used to. We give an example by showing a reduction from bit string guessing to set cover.

5.1 Preliminaries

Let us define the two models of the oracle and the model of the probabilistic adversary that we consider in this chapter more formally. Compared to the deterministic adversary, the probabilistic adversary is strengthened in the following way. Given an online problem P and the set of all possible input instances $\mathcal{I}_{\text{all}}$ for P, the adversary may determine an arbitrary probability distribution ψ over the instances in $\mathcal{I}_{\text{all}}$, such that the actual instance I that an algorithm $\mathcal{A}$ gets as an input is drawn from the set $\mathcal{I}_{\text{all}}$ according to ψ.

In this setting, we consider two different models of the oracle. The first one we consider is an adaptation of a model introduced by Hromkovič et al. [HKK10], which we call the *monolog model*. Earlier in this chapter, we briefly summarized the procedure of choosing a hard input instance, writing the advice onto the tape, and then computing an output for the input instance using the advice string in the classical scenario. Now, to incorporate the probabilistic component, we additionally demand that the advice is written onto the tape before the adversary draws the instance I from $\mathcal{I}_{\text{all}}$. This way, we make sure that the oracle is able to encode information about the distribution ψ in the advice string, but not about the actual input instance I. Formally, we can define an online algorithm with advice in the monolog model playing against a probabilistic adversary as follows.

Definition 5.1 (Online Algorithm with Advice (Monolog Model)). *Let P be an online minimization problem and let the set of all possible input instances for P be $\mathcal{I}_{all}$. Furthermore, consider an input $I = (x_1, \ldots, x_n)$ of P that is drawn from $\mathcal{I}_{all}$ according to some probability distribution ψ. An online algorithm $\mathcal{A}$ with advice in the monolog model computes the output sequence $\mathcal{A}^\tau(I) = (y_1, \ldots, y_n)$ such that every y_i is computed from $x_1, \ldots, x_i, y_1, \ldots, y_{i-1}$ and τ, where τ is the content of the advice tape that can be described as a function of P, ψ, and $\mathcal{A}$. The algorithm $\mathcal{A}$ is said to have an advice complexity of $b(n)$ if, for every n and for any input sequence I of length at most n, at most the first $b(n)$ bits of τ are accessed during the computation of $\mathcal{A}^\tau(I)$.*

The second model of the oracle we want to consider is a slight adaptation of the so-called *answerer model* introduced by Dobrev et al. [DKP08]. In contrast to the first model that we called the monolog model, we denote the second one by the *dialog model*. In this model, the algorithm can demand any amount of advice bits from the oracle after each request, one after another. The oracle is only allowed to send any advice bits when the algorithm demands them and is bound to send each demanded advice bit immediately. Hence, upon receiving a request, the algorithm decides if it wants to demand an advice bit from the oracle at all and if so, sends the first demand. Upon receiving an advice bit, it decides if it wants to send a demand for another advice bit. If it does not want to demand any additional advice bits, it produces its output and, after that, receives the next request in the next round.

We let the algorithm demand the exact number of required advice bits instead of giving the oracle the power to send some advice string by its own accord, since otherwise the oracle could encode some additional information into the length of the advice string sent or even into the circumstance that it sends any advice string at all. This is a problem leading to unnecessary complications in the model of Dobrev et al. [DKP08] that we want to avoid here.

Let the number of advice bits the algorithm demands from the oracle in round i be $d_i \in \mathbb{N}^{\geq 0}$ in total, and let the advice string the oracle sends to the algorithm in round i be $\hat{\tau}_i \in \{0,1\}^{d_i}$. Furthermore, let $\tau := \hat{\tau}_1\hat{\tau}_2 \ldots \hat{\tau}_n$ be the total advice string the algorithm reads during its computation on an instance of size n. We define an online algorithm with advice playing against a probabilistic adversary in the dialog model as follows.

Definition 5.2 (Online Algorithm with Advice (Dialog Model)). *Let P be an online minimization problem and let the set of all possible input instances for P be $\mathcal{I}_{all}$. Furthermore, consider an input $I = (x_1, \ldots, x_n)$ of P that is drawn from $\mathcal{I}_{all}$ according to some probability distribution ψ. An online algorithm $\mathcal{A}$ with advice in the dialog model computes the output sequence $\mathcal{A}^{\tau}(I) = (y_1, \ldots, y_n)$ as follows. Assume that $\mathcal{A}$ has so far read the advice string $\tau_1 \ldots \tau_{j-1}$. Upon receiving a request x_i or an advice bit τ_j, the algorithm either demands another advice bit from the oracle, or it produces the output y_i. This decision is made based on $x_1, \ldots, x_i, y_1, \ldots, y_{i-1}, \tau_1, \ldots, \tau_j$. Assume $\mathcal{A}$ produces its output y_i in round i after having read the advice string $\hat{\tau}_i$ of length d_i. Then, the output y_i is computed as a function of $x_1, \ldots, x_i, y_1, \ldots, y_{i-1}, \hat{\tau}_1, \ldots, \hat{\tau}_i$. The algorithm $\mathcal{A}$ is said to have an advice complexity of $b(n)$ if, for every n and for any input sequence I of length at most n, the total advice string $\tau = \hat{\tau}_1 \ldots \hat{\tau}_n$ that $\mathcal{A}$ reads during the computation of $\mathcal{A}^{\tau}(I)$ has length at most $b(n)$.*

In this setting, we consider the bit string guessing problem. Since each string r unambiguously corresponds to an input sequence I_r and vice versa, we will also often speak of the "input string r" instead of the "input sequence I_r".

Hence, in this setting, the set of all possible input instances of length n is the set of all 2^n binary strings of length n, and the adversary chooses a probability distribution over these strings. The goal of an algorithm for the string guessing problem is to minimize its cost, hence the number of incorrectly guessed bits, or, phrasing it as a maximization problem, to maximize the number of correctly guessed bits. Let us additionally define the number of correctly guessed bits of a string guessing algorithm $\mathcal{A}$ on the input I_r as $\text{gain}_{\mathcal{A}}(I_r) = n - \text{cost}(\mathcal{A}(I_r))$ for any input $I_r \in \mathcal{I}_{\text{all}}$. Since I_r is drawn from $\mathcal{I}_{\text{all}}$ according to some probability distribution ψ, $\text{gain}_{\mathcal{A}}$ is a random variable, i. e., a function that maps any input $I_r \in \mathcal{I}_{\text{all}}$ to a number of correctly guessed bits $\text{gain}_{\mathcal{A}}(I_r) \in \{0, \ldots, n\}$. When talking about the quality of an algorithm $\mathcal{A}$, we are interested in the expected value $\mathbb{E}_{\psi}[\text{gain}_{\mathcal{A}}]$ of $\text{gain}_{\mathcal{A}}$ with respect to ψ, i. e., the expected number of correctly guessed bits of $\mathcal{A}$ when the input for $\mathcal{A}$ is chosen from $\mathcal{I}_{\text{all}}$ according to ψ.

In this thesis, we only consider a special class of probability distributions, which we call Ψ. In each distribution from Ψ, the adversary excludes a set $\tilde{\mathcal{I}}$ containing $\tilde{k}$ strings from $\mathcal{I}_{\text{all}}$ (i. e., it assigns a probability of 0 to them) and draws the actual input string r uniformly at random from the set $\mathcal{I} := \mathcal{I}_{\text{all}} \setminus \tilde{\mathcal{I}}$ of remaining strings, with $|\mathcal{I}| = 2^n - \tilde{k} =: k$. The size $\tilde{k}$ of the excluded set $\tilde{\mathcal{I}}$ is known to $\mathcal{A}$ in advance, together with the fact that r is chosen from $\mathcal{I}$ uniformly at random, but the concrete distribution is not. In other words, $\mathcal{A}$ does not know the excluded set $\tilde{\mathcal{I}}$.

Since we constrain ourselves to this special class Ψ of probability distributions, we can identify each set $\tilde{\mathcal{I}}$ with one unambiguous distribution ψ. Therefore, we will write $\mathbb{E}_{\tilde{\mathcal{I}}}[\text{gain}_{\mathcal{A}}]$ instead of $\mathbb{E}_{\psi}[\text{gain}_{\mathcal{A}}]$, whenever it aids comprehension.

Since we will often consider prefixes or suffixes of strings, let us introduce some helpful notation. For each string s of length n and every pair of natural numbers i, j with $1 \leq i \leq j \leq n$, we write $[s]_i^j$ for the substring of s that starts at position i and ends at position j, with i and j included. Hence, the notation r_i for the i-th bit of r that we already used above is actually a shorthand notation for $[r]_i^i$. As a shorthand notation for the prefix of length j of a string s, we write $[s]^j$ instead of $[s]_1^j$. As a shorthand notation for the suffix of s that starts at position i, we use $[s]_i$ instead of $[s]_i^n$. Additionally, for $j < i$ we define $[s]_i^j = \varepsilon$, where ε is the empty string.

5.2 Monolog Model

Having settled these technicalities, let us start with the analysis of the monolog model. This section is structured as follows. In Section 5.2.1, we present an upper bound on the expected number of advice bits for 2-GUESS sufficient to obtain the optimal achievable number of correctly guessed bits in expectation in the setting we described in detail in Section 5.1. After describing the corresponding 2-GUESS algorithm in Section 5.2.1.1, we have to gather a few technical results in Section 5.2.1.2 that we will need later for the analysis of the algorithm in Section 5.2.1.3. Since the probabilistic adversary model does not allow for optimal solutions, as we have already mentioned, we give a reasonable definition for the optimality of online algorithms in the probabilistic setting in Section 5.2.1.4 and show that the algorithm from Section 5.2.1.1 is optimal according to this definition. We proceed by giving an almost matching lower bound in Section 5.2.2 and conclude our considerations concerning the monolog model with a comparison of the upper and the lower bound in Section 5.2.3.

5.2.1 Upper Bound

In this section, we present and analyze an algorithm $\mathcal{A}$ with advice for the string guessing problem with a probabilistic adversary that can only choose probability distributions from the class Ψ. Let $\psi \in \Psi$ be the distribution chosen by the adversary, and let the set of excluded strings corresponding to ψ be $\tilde{\mathcal{I}}$, containing $\tilde{k}$ strings. Intuitively, the performance of $\mathcal{A}$ suffers with growing size of $\mathcal{I}$, hence with decreasing $\tilde{k}$. Recall that $\tilde{k}$ is known to $\mathcal{A}$ in advance, but the set of excluded strings is not.

5.2.1.1 The Algorithm

In the first round, $\mathcal{A}$ is given the first request, consisting of the length n of the input sequence. Knowing n and $\tilde{k}$, the algorithm $\mathcal{A}$ reads $\lceil \log \binom{2^n}{\tilde{k}} \rceil$ advice bits from the tape. These bits serve to communicate the set $\tilde{\mathcal{I}}$ of excluded strings to $\mathcal{A}$. Since there are $\binom{2^n}{\tilde{k}}$ different possibilities to choose $\tilde{k}$ out of 2^n strings, $\lceil \log \binom{2^n}{\tilde{k}} \rceil$ advice bits are sufficient for $\mathcal{A}$ to know the set $\mathcal{I} = \mathcal{I}_{\text{all}} \setminus \tilde{\mathcal{I}}$ that the actual input string is drawn from uniformly at random.

In each round i with $1 \leq i \leq n$, the algorithm $\mathcal{A}$ is asked to guess the i-th bit r_i of the input string r. The algorithm's guess g_i is dependent on the set $\tilde{\mathcal{I}}$ and the feedback that $\mathcal{A}$ got in rounds $2, \ldots, i$ so far, consisting of the bits $r_1, \ldots, r_{i-1}$. (An obvious exception is round 1, when $\mathcal{A}$ did not get any feedback yet.)

Based on the set $\tilde{\mathcal{I}}$ and the feedback of rounds $2, \ldots, i$, one can gather some information about the form of r and conclude that some of the strings in $\mathcal{I}$ certainly do not match the to-be-guessed string r, namely those that do not have the prefix $[r]^{i-1}$. Let us call a string s a *candidate* for r if s might still be the to-be-guessed string r from all that can be told from the set $\tilde{\mathcal{I}}$ and the feedback provided so far. Of course, in the beginning of the execution, the set of these candidates is $\mathcal{I}$. Then, during the execution of the algorithm, the feedback can be used to narrow down the set of candidates gradually, round after round.

To decide which bit to guess in round i, the algorithm $\mathcal{A}$ keeps track of the set C_i of strings that are still candidates for r in round i. Due to the advice, $\mathcal{A}$ knows the set $\tilde{\mathcal{I}}$ of excluded strings before it has to make the first guess, and sets $C_1 := \mathcal{I}_{\text{all}} \setminus \tilde{\mathcal{I}}$. Then, $\mathcal{A}$ has to guess the bit r_1. It determines the bit that appears most often at position 1 among all strings in C_1 and outputs it.

In round $i \geq 2$, when $\mathcal{A}$ is given the bit r_{i-1} (the one that $\mathcal{A}$ should have guessed in round $i-1$), $\mathcal{A}$ computes C_i from C_{i-1} and r_{i-1}. It excludes all strings from C_{i-1} that do not have the bit r_{i-1} at position $i-1$. The strings that remain are exactly those strings from $\mathcal{I}$ with the prefix $[r]^{i-1}$. Hence, in each round i, the set of remaining candidates is $C_i = \{s \in \mathcal{I} \mid [s]^{i-1} = [r]^{i-1}\}$. After that, $\mathcal{A}$ guesses the bit r_i depending on the set C_i. It examines the i-th position of all strings in C_i. If the number of ones is at least $|C_i|/2$, then $\mathcal{A}$ outputs $g_i = 1$, otherwise $g_i = 0$.

Hence, in round i, the algorithm $\mathcal{A}$ always guesses the bit that is most likely to be the bit r_i.

5.2.1.2 Analysis of Total Hamming Weights

Before we can start with the actual analysis of the algorithm, we need a few technical results.

In the following, let $\omega(\tilde{k})$ be the *Hamming weight* of $\tilde{k}$, i.e., the number of ones in the (shortest) binary representation of $\tilde{k}$ for all $\tilde{k} \in \mathbb{N}^{\geq 0}$. Furthermore, let $\hat{\omega}(\tilde{k})$ be the sum of the Hamming weights of all numbers from 0 to $\tilde{k}-1$, i.e., $\hat{\omega}(\tilde{k}) := \sum_{i=0}^{\tilde{k}-1} \omega(i)$ for all $\tilde{k} \in \mathbb{N}^{\geq 1}$. We call $\hat{\omega}(\tilde{k})$ the *total Hamming weight* of $\tilde{k}$. Additionally, let us define $\hat{\omega}(0) = 0$. We make the following observations concerning the values of $\omega(\tilde{k})$ and $\hat{\omega}(\tilde{k})$.

Observation 5.3. *For every $n \in \mathbb{N}^{\geq 0}$ and all $i \in \mathbb{N}^{\geq 1}$ with $i \leq 2^n - 1$, we have*

$$\omega(2^n - i) + \omega(i - 1) = n.$$

Proof. Let us pad all shortest binary representations of the numbers from 0 to $2^n - 1$ from the left with zeros such that each string has length n. This does not affect the number of ones in those strings. The binary representation of the number $2^n - i$ is the exact inverse of the binary representation of the number $i - 1$, i. e., one can be obtained from the other by flipping each of the n bits. Hence, adding the Hamming weight of $2^n - i$ and $i - 1$ yields $\omega(2^n - i) + \omega(i - 1) = n$. □

Observation 5.4. *For every $n \in \mathbb{N}^{\geq 0}$, the total Hamming weight of 2^n is*

$$\hat{\omega}(2^n) = n \cdot 2^{n-1}.$$

Proof. Let us again pad all shortest binary representations of the numbers from 0 to $2^n - 1$ from the left with zeros such that each string has length n. For any n, the value of $\hat{\omega}(2^n)$ equals the total number of ones in all 2^n binary strings of length n. Clearly, these are exactly half of all these $n \cdot 2^n$ bits, thus $n \cdot 2^{n-1}$ bits. □

Observation 5.5. *For every $n \in \mathbb{N}^{\geq 0}$, the total Hamming weight of $2^n - 1$ is*

$$\hat{\omega}(2^n - 1) = n \cdot 2^{n-1} - n.$$

Proof. The total Hamming weight of $2^n =: \tilde{k} + 1$ can be computed as $\hat{\omega}(\tilde{k} + 1) = n \cdot 2^{n-1}$ according to Observation 5.4. To derive the total Hamming weight of $\tilde{k}$, the number of ones in the binary representation of $\tilde{k}$ has to be subtracted from $\hat{\omega}(\tilde{k}+1)$, hence $\hat{\omega}(\tilde{k}) = \hat{\omega}(\tilde{k} + 1) - n = n \cdot 2^{n-1} - n$. □

Observation 5.6. *For every $n \in \mathbb{N}^{\geq 0}$, the total Hamming weight of $2^n + 1$ is*

$$\hat{\omega}(2^n + 1) = n \cdot 2^{n-1} + 1.$$

Proof. The total Hamming weight of $2^n =: \tilde{k} - 1$ can be computed as $\hat{\omega}(\tilde{k} - 1) = n \cdot 2^{n-1}$ according to Observation 5.4. To derive the total Hamming weight of $\tilde{k}$, the number of ones in the binary representation of $\tilde{k} - 1$ has to be added to $\hat{\omega}(\tilde{k} - 1)$, hence $\hat{\omega}(\tilde{k}) = \hat{\omega}(\tilde{k} - 1) + 1 = n \cdot 2^{n-1} + 1$. □

Observation 5.7. *For all $\tilde{k} \in \mathbb{N}^{\geq 2}$, let $n := \lceil \log \tilde{k} \rceil - 1$. In other words, n is the unambiguous natural number such that $2^n + 1 \leq \tilde{k} \leq 2^{n+1}$, and $n \in \mathbb{N}^{\geq 0}$. Hence, $\tilde{k} = 2^n + x$ for some $x \in \mathbb{N}^{\geq 1}$ with $1 \leq x \leq 2^n$. Then we have*

$$\hat{\omega}(\tilde{k}) = \hat{\omega}(2^n + x) = n \cdot 2^{n-1} + x + \hat{\omega}(x).$$

Proof. We can rewrite the total Hamming weight of $\tilde{k}$ as

$$\hat{\omega}(\tilde{k}) = \hat{\omega}(2^n + x) = \sum_{i=0}^{\tilde{k}-1} \omega(i) = \sum_{i=0}^{2^n-1} \omega(i) + \sum_{i=2^n}^{\tilde{k}-1} \omega(i) = \hat{\omega}(2^n) + \sum_{i=2^n}^{\tilde{k}-1} \omega(i).$$

We now have to analyze the sum on the right-hand side. It consists of $\tilde{k} - 2^n = x$ summands. All the shortest binary representations of the numbers 2^n to $\tilde{k} - 1$ have $n+1$ positions, and the leftmost bit is one. The binary representations of these numbers $2^n, \dots, \tilde{k} - 1$ without the leftmost bits equal the binary representations of the numbers $0, \dots, x-1$. Hence, $\sum_{i=2^n}^{\tilde{k}-1} \omega(i) = x + \sum_{i=0}^{x-1} \omega(i) = x + \hat{\omega}(x)$, and thus $\hat{\omega}(\tilde{k}) = \hat{\omega}(2^n) + x + \hat{\omega}(x)$. With Observation 5.4, the claim follows. □

Observation 5.8. *For all* $n, x \in \mathbb{N}^{\geq 1}$ *with* $x \leq 2^n$*, we have*

$$\hat{\omega}(2^n - x) = \hat{\omega}(2^n) - x \cdot n + \hat{\omega}(x).$$

Proof. We use Observation 5.3 to make the following calculation.

$$\begin{aligned}
\hat{\omega}(2^n - x) &= \hat{\omega}(2^n) - \omega(2^n - 1) - \omega(2^n - 2) - \ldots - \omega(2^n - x) \\
&= \hat{\omega}(2^n) - \sum_{i=1}^{x} \omega(2^n - i) \\
&= \hat{\omega}(2^n) - \left(\sum_{i=1}^{x} n - \omega(i-1) \right) \\
&= \hat{\omega}(2^n) - n \cdot x + \sum_{i=0}^{x-1} \omega(i) \\
&= \hat{\omega}(2^n) - n \cdot x + \hat{\omega}(x).
\end{aligned}$$
□

Observation 5.9. *For all* $n, k \in \mathbb{N}^{\geq 1}$ *with* $k \leq 2^n$ *and* $\tilde{k} = 2^n - k$*, we have*

$$\frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}} = n - \frac{\hat{\omega}(k)}{k}.$$

Proof. We make the following calculation, using Observation 5.4 for (5.1) and Observation 5.8 for (5.2).

$$\begin{aligned}
\frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}} &= \frac{\hat{\omega}(2^n) - \hat{\omega}(2^n - k)}{k} && (5.1) \\
&= \frac{\hat{\omega}(2^n) - (\hat{\omega}(2^n) - kn + \hat{\omega}(k))}{k} && (5.2) \\
&= \frac{kn - \hat{\omega}(k)}{k} \\
&= n - \frac{\hat{\omega}(k)}{k}.
\end{aligned}$$
□

Observation 5.10. *For each* $m, \ell \in \mathbb{N}^{\geq 1}$ *with* ℓ *odd, we have*

$$\frac{\hat{\omega}(2^m \cdot \ell)}{2^m} = \frac{m}{2} \cdot \ell + \hat{\omega}(\ell).$$

Proof. We consider the binary representations of the numbers $0, \ldots, \hat{\omega}(2^m \cdot \ell)$. Let us again pad all these binary representations with zeros such that they have length n each, for some $n \geq m + \log \ell$. We put them into ascending order and divide them into ℓ blocks, such that block i consists of the representations of the numbers $i \cdot 2^m, \ldots, (i+1) \cdot 2^m - 1$, for $0 \leq i \leq \ell - 1$. Moreover, we consider the suffixes of length m and the prefixes of length $n - m$ separately. In each block i, each such prefix equals the binary representation of length $n - m$ of the number i. Hence, the prefixes in block i with $0 \leq i \leq \ell - 1$ contain $2^m \cdot \omega(i)$ ones in total. All prefixes together then contain $2^m \cdot \sum_{i=0}^{\ell-1} \omega(i) = 2^m \cdot \hat{\omega}(\ell)$ ones.

Let us now consider the suffixes of length m of the binary representations of the numbers in block i. Each such suffix that corresponds to a number $i \cdot 2^m + j$ with $0 \leq j \leq 2^m - 1$ equals the binary representation of length m of the number j. Hence, the suffixes in block i contain $\omega(0) + \ldots + \omega(2^m - 1) = \hat{\omega}(2^m)$ ones in total, and thus the suffixes altogether contain $\ell \cdot \hat{\omega}(2^m)$ ones. We obtain

$$\hat{\omega}(2^m \cdot \ell) = 2^m \cdot \hat{\omega}(\ell) + \ell \cdot \hat{\omega}(2^m) \tag{5.3}$$

and, therefore,

$$\frac{\hat{\omega}(2^m \cdot \ell)}{2^m} = \frac{2^m \cdot \hat{\omega}(\ell) + \ell \cdot \hat{\omega}(2^m)}{2^m} = \hat{\omega}(\ell) + \ell \cdot \frac{m \cdot 2^{m-1}}{2^m} = \hat{\omega}(\ell) + \ell \cdot \frac{m}{2},$$

using (5.3) and Observation 5.4. □

Observation 5.7 gives us the possibility to derive the following recursive formula for $\hat{\omega}(\tilde{k})$.

Lemma 5.11. *For all* $\tilde{k} \in \mathbb{N}^{\geq 0}$, *we have*

$$\begin{aligned}
\hat{\omega}(0) &= 0,\\
\hat{\omega}(1) &= 0,\\
\hat{\omega}(2\tilde{k}) &= 2 \cdot \hat{\omega}(\tilde{k}) + \tilde{k},\\
\hat{\omega}(2\tilde{k}+1) &= \hat{\omega}(\tilde{k}) + \hat{\omega}(\tilde{k}+1) + \tilde{k}.
\end{aligned}$$

Proof. We prove the claim by induction on $\tilde{k}$. By the definition of $\hat{\omega}$, for $\tilde{k} = 0$, we have $\hat{\omega}(0) = 0$, which coincides with $2 \cdot \hat{\omega}(0) + 0$, and $\hat{\omega}(1) = 0$, which coincides with $\hat{\omega}(0) + \hat{\omega}(1) + 0$. In addition, for $\tilde{k} = 1$, we have $\hat{\omega}(2) = 1$, which coincides with $2 \cdot \hat{\omega}(1) + 1 = 2 \cdot 0 + 1$, and $\hat{\omega}(3) = 2$, which coincides with

$\hat{\omega}(1) + \hat{\omega}(2) + 1 = 0 + 1 + 1$. For any given $\tilde{k} = 2^x + y$ with $1 \leq y \leq 2^x$, assume that the claim holds for y. We will show that it then also holds for $2\tilde{k}$ and $2\tilde{k} + 1$.

First, we analyze the case for even values $2\tilde{k}$. We can write $2\tilde{k}$ as $2\tilde{k} = 2^{x+1} + 2y$. As $1 \leq y \leq 2^x$, we have $2 \leq 2y \leq 2^{x+1}$ and $2^{x+1} + 2 \leq 2\tilde{k} \leq 2^{x+2}$. Hence, we can apply Observation 5.7 to $\hat{\omega}(2\tilde{k})$ and calculate it as

$$\begin{aligned}
\hat{\omega}(2\tilde{k}) &= \hat{\omega}(2^{x+1} + 2y) \\
&= (x+1) \cdot 2^x + 2y + \hat{\omega}(2y) \\
&= 2 \cdot x \cdot 2^{x-1} + 2^x + 2y + 2\,\hat{\omega}(y) + y \qquad (5.4) \\
&= 2 \cdot (x \cdot 2^{x-1} + y + \hat{\omega}(y)) + 2^x + y \\
&= 2 \cdot \hat{\omega}(\tilde{k}) + \tilde{k},
\end{aligned}$$

where we used the induction hypothesis $\hat{\omega}(2y) = 2 \cdot \hat{\omega}(y) + y$ for (5.4).

Now let us consider odd values $2\tilde{k}+1$. We write $2\tilde{k}+1$ as $2\tilde{k}+1 = 2^{x+1} + 2y + 1$. Since $1 \leq y \leq 2^x$, we have $3 \leq 2y+1 \leq 2^{x+1} + 1$ and $2^{x+1} + 3 \leq 2\tilde{k}+1 \leq 2^{x+2} + 1$. We distinguish two cases, depending on the value of $\tilde{k}$.

Case 1. $\tilde{k} \leq 2^{x+1} - 1$.

In this case, we can derive that $y \leq 2^x - 1$ and thus $2y + 1 \leq 2^{x+1} - 1$; in addition, we have $\tilde{k} + 1 \leq 2^{x+2} - 1$. Applying Observation 5.7 for $\hat{\omega}(2\tilde{k} + 1)$ yields

$$\begin{aligned}
\hat{\omega}(2\tilde{k}+1) &= \hat{\omega}(2^{x+1} + 2y + 1) \\
&= (x+1) \cdot 2^x + 2y + 1 + \hat{\omega}(2y+1) \\
&= 2 \cdot x \cdot 2^{x-1} + 2^x + 2y + 1 + \hat{\omega}(y) + \hat{\omega}(y+1) + y \qquad (5.5) \\
&= (x \cdot 2^{x-1} + y + \hat{\omega}(y)) + (x \cdot 2^{x-1} + y + 1 + \hat{\omega}(y+1)) + 2^x + y \\
&= \hat{\omega}(\tilde{k}) + \hat{\omega}(\tilde{k}+1) + \tilde{k},
\end{aligned}$$

where we used the induction hypothesis $\hat{\omega}(2y+1) = \hat{\omega}(y) + \hat{\omega}(y+1) + y$ for (5.5).

Case 2. $\tilde{k} = 2^{x+1}$.

In this case, we derive that $y = 2^x$ and $2\tilde{k} + 1 = 2^{x+2} + 1$. We cannot apply Observation 5.7 for $\hat{\omega}(2\tilde{k} + 1)$ here, since the constraint $2\tilde{k} + 1 \leq 2^{x+2}$ does not hold. Instead, we calculate $\hat{\omega}(2\tilde{k} + 1)$ as follows.

$$\begin{aligned}
\hat{\omega}(2\tilde{k}+1) &= \hat{\omega}(2^{x+2} + 1) \\
&= (x+2) \cdot 2^{x+1} + 1 \qquad (5.6) \\
&= (x+1) \cdot 2^{x+1} + 2^{x+1} + 1 \\
&= (x+1) \cdot 2^x + (x+1) \cdot 2^x + 1 + \tilde{k} \\
&= \hat{\omega}(2^{x+1}) + \hat{\omega}(2^{x+1} + 1) + \tilde{k} \qquad (5.7) \\
&= \hat{\omega}(\tilde{k}) + \hat{\omega}(\tilde{k}+1) + \tilde{k},
\end{aligned}$$

where we used Observation 5.6 for (5.6) and (5.7).

As all possible cases for $\tilde{k}$ are covered, this completes the proof for odd values $2\tilde{k}+1$; thus, we are done. □

Corollary 5.12. *For all $\tilde{k} \in \mathbb{N}^{\geq 0}$, we have*

$$\hat{\omega}(\tilde{k}) = \left\lfloor \tfrac{\tilde{k}}{2} \right\rfloor + \hat{\omega}\left(\left\lfloor \tfrac{\tilde{k}}{2} \right\rfloor\right) + \hat{\omega}\left(\left\lceil \tfrac{\tilde{k}}{2} \right\rceil\right).$$

Proof. This follows directly from Lemma 5.11. □

Lemma 5.13. *For all $\tilde{k} \in \mathbb{N}^{\geq 0}$, we have*

$$\hat{\omega}(\tilde{k}) \leq \frac{\tilde{k}}{2} \cdot \log \tilde{k}.$$

Proof. We prove the claim by induction on $\tilde{k}$. It clearly holds for all $\tilde{k} \leq 3$. Now let us assume that the claim holds for $\tilde{k}$ and $\tilde{k}+1$. We show that it then also holds for $2\tilde{k}$ and $2\tilde{k}+1$. We have

$$\begin{aligned}
\hat{\omega}(2\tilde{k}) &= 2\,\hat{\omega}(\tilde{k}) + \tilde{k} && (5.8)\\
&\leq 2 \cdot \frac{\tilde{k}}{2} \cdot \log \tilde{k} + \tilde{k} && (5.9)\\
&= \tilde{k} \cdot \big(\log \tilde{k} + 1\big)\\
&= \tilde{k} \cdot \log(2\tilde{k})\\
&= \frac{2\tilde{k}}{2} \cdot \log(2\tilde{k}),
\end{aligned}$$

where (5.8) holds due to Lemma 5.11 and (5.9) is the induction hypothesis for $\tilde{k}$. Furthermore,

$$\begin{aligned}
\hat{\omega}(2\tilde{k}+1) &= \hat{\omega}(\tilde{k}) + \hat{\omega}(\tilde{k}+1) + \tilde{k} && (5.10)\\
&\leq \tilde{k}/2 \cdot \log \tilde{k} + (\tilde{k}+1)/2 \cdot \log(\tilde{k}+1) + \tilde{k} && (5.11)\\
&= 0.5 \cdot \big(\tilde{k} \cdot \log \tilde{k} + (\tilde{k}+1) \cdot \log(\tilde{k}+1) + 2\tilde{k}\big)\\
&= 0.5 \cdot \big(\tilde{k} \cdot \log \tilde{k} + \tilde{k} + \tilde{k} \cdot \log(\tilde{k}+1) + \tilde{k} + \log(\tilde{k}+1)\big)\\
&= 0.5 \cdot \big(\tilde{k} \cdot (\log \tilde{k} + 1) + \tilde{k} \cdot (\log(\tilde{k}+1) + 1) + \log(\tilde{k}+1)\big)\\
&= 0.5 \cdot \big(\tilde{k} \cdot \log(2\tilde{k}) + \tilde{k} \cdot \log(2\tilde{k}+2) + \log(\tilde{k}+1)\big)\\
&= 0.5 \cdot \big(\tilde{k} \cdot (\log(2\tilde{k}) + \log(2\tilde{k}+2)) + \log(\tilde{k}+1)\big)\\
&\leq 0.5 \cdot \big(\tilde{k} \cdot (2 \cdot \log(2\tilde{k}+1)) + \log(\tilde{k}+1)\big) && (5.12)\\
&\leq 0.5 \cdot \big(2\tilde{k} \cdot \log(2\tilde{k}+1) + \log(2\tilde{k}+1)\big)\\
&= 0.5 \cdot \big((2\tilde{k}+1) \cdot \log(2\tilde{k}+1)\big),
\end{aligned}$$

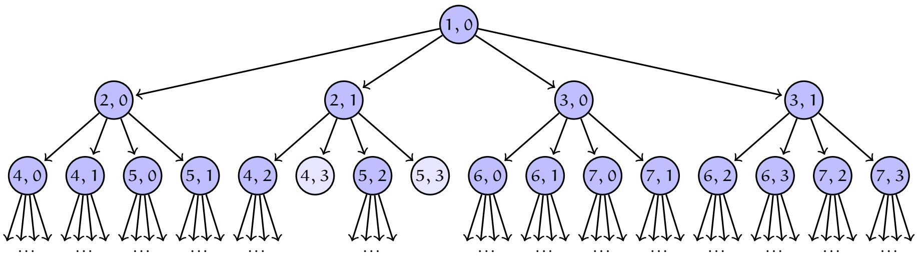


Figure 5.1. Graphical illustration of the induction proof of Lemma 5.14. Each vertex of the tree represents an appropriate pair $(\tilde{k}, j)$. If the claim holds for such a pair $(\tilde{k}, j)$ and this induces by induction that the claim also holds for another appropriate pair $(\tilde{k}', j')$, this is indicated by an edge from $(\tilde{k}, j)$ to $(\tilde{k}', j')$. Hence, each vertex can have at most four children, namely $(2\tilde{k}, 2j)$, $(2\tilde{k}, 2j+1)$, $(2\tilde{k}+1, 2j)$, and $(2\tilde{k}+1, 2j+1)$. Note that not every vertex has exactly four children, because it might be that some of them are not appropriate, which is indicated in the graph by the light blue colored vertices.

where (5.10) again holds due to Lemma 5.11 again, (5.11) is the induction hypothesis for $\tilde{k}+1$ and $\tilde{k}$, and (5.12) holds due to the concavity of the logarithm function. □

Lemma 5.14. *For all $\tilde{k}, j \in \mathbb{N}^{\geq 0}$ with $j \leq \lfloor \tilde{k}/2 \rfloor$, we have*

$$\hat{\omega}(j) + \hat{\omega}(\tilde{k} - j) \leq \hat{\omega}(\tilde{k}) - j.$$

Proof. Let us call a pair $(\tilde{k}, j)$ *appropriate* if it satisfies $0 \leq j \leq \lfloor \tilde{k}/2 \rfloor$. We prove the claim by induction on j and $\tilde{k}$. It can be easily verified that it holds for all appropriate pairs $(\tilde{k}, j)$ with $\tilde{k} \leq 4$ and $j \leq 2$. Let us assume that the claim holds for all appropriate pairs $(\tilde{k}', j')$ with $\tilde{k}' \leq \tilde{k}$ and $j' \leq j$. We show that the claim then also holds for all appropriate pairs $(2\tilde{k}, 2j)$, $(2\tilde{k}, 2j+1)$, $(2\tilde{k}+1, 2j)$, and $(2\tilde{k}+1, 2j+1)$. Hence, in the following, we distinguish four cases. In each of these cases, we use the induction hypothesis and Lemma 5.11 to obtain the desired result. The structure of the induction is shown in Figure 5.1.

Case 1. $(2\tilde{k}, 2j)$.

In this case, we use the fact that, due to the induction hypothesis, $\hat{\omega}(j) + \hat{\omega}(\tilde{k}-j) \leq \hat{\omega}(\tilde{k}) - j$ holds for all appropriate pairs $(\tilde{k}, j)$, i. e., for all j and $\tilde{k}$ with $j \leq \lfloor \tilde{k}/2 \rfloor$. Thus,

$$\begin{aligned}
\hat{\omega}(2j) + \hat{\omega}\big(2\tilde{k} - 2j\big) &= j + \hat{\omega}(j) + \hat{\omega}(j) + \tilde{k} - j + \hat{\omega}(\tilde{k} - j) + \hat{\omega}(\tilde{k} - j) \\
&= \tilde{k} + \hat{\omega}(j) + \hat{\omega}(\tilde{k} - j) + \hat{\omega}(j) + \hat{\omega}(\tilde{k} - j) \\
&\leq \tilde{k} + 2\,\hat{\omega}(\tilde{k}) - 2j \\
&= \hat{\omega}(2\tilde{k}) - 2j.
\end{aligned}$$

Case 2. $(2\tilde{k}, 2j+1)$.

Here, we use the fact that, due to the induction hypothesis, $\hat{\omega}(j) + \hat{\omega}(\tilde{k} - j) \leq \hat{\omega}(\tilde{k}) - j$ and $\hat{\omega}(j+1) + \hat{\omega}(\tilde{k} - j - 1) \leq \hat{\omega}(\tilde{k}) - (j+1)$ hold for all j and $\tilde{k}$ satisfying $j \leq \lfloor \tilde{k}/2 \rfloor$ and $j + 1 \leq \lfloor \tilde{k}/2 \rfloor$, respectively, and hence $j \leq \lfloor \tilde{k}/2 \rfloor - 1$. We obtain

$$\begin{aligned}
\hat{\omega}(2j+1) + \hat{\omega}\big(2\tilde{k} - 2j - 1\big) &= \hat{\omega}(2j+1) + \hat{\omega}(2(\tilde{k} - j - 1) + 1) \\
&= j + \hat{\omega}(j) + \hat{\omega}(j+1) \\
&\qquad + \tilde{k} - j - 1 + \hat{\omega}(\tilde{k} - j - 1) + \hat{\omega}(\tilde{k} - j) \\
&= \tilde{k} - 1 + \hat{\omega}(j) + \hat{\omega}(\tilde{k} - j) + \hat{\omega}(\tilde{k} - j - 1) + \hat{\omega}(j+1) \\
&\leq \tilde{k} - 1 + \hat{\omega}(\tilde{k}) - j + \hat{\omega}(\tilde{k}) - j - 1 \\
&= \hat{\omega}(2\tilde{k}) - 2j - 2 \\
&< \hat{\omega}(2\tilde{k}) - 2j - 1.
\end{aligned}$$

In this case, it remains to show that the claim also holds for $j = \lfloor \tilde{k}/2 \rfloor$ when $\tilde{k}$ is odd, because this is the only case that is not covered yet. Hence, we know that $j = (\tilde{k} - 1)/2$ and therefore $\tilde{k} = 2j + 1$. Doing some simple transformations, we obtain

$$\begin{aligned}
\hat{\omega}(2j+1) + \hat{\omega}(2\tilde{k} - 2j - 1) &= \hat{\omega}(\tilde{k}) + \hat{\omega}(2\tilde{k} - (\tilde{k} - 1) - 1) \\
&= 2\,\hat{\omega}(\tilde{k}) \\
&= \hat{\omega}(2\tilde{k}) - \tilde{k} \\
&= \hat{\omega}(2\tilde{k}) - 2j - 1.
\end{aligned}$$

Case 3. $(2\tilde{k}+1, 2j)$.

Due to the induction hypothesis, we have $\hat{\omega}(j) + \hat{\omega}(\tilde{k} - j) \leq \hat{\omega}(\tilde{k}) - j$, for all appropriate pairs $(\tilde{k}, j)$, and $\hat{\omega}(j) + \hat{\omega}(\tilde{k} - j + 1) \leq \hat{\omega}(\tilde{k} + 1) - j$, for all appropriate pairs $(\tilde{k}+1, j)$, hence for all j and $\tilde{k}$ with $j \leq \lfloor \tilde{k}/2 \rfloor$ in particular. This leads to

$$\begin{aligned}
\hat{\omega}(2j) + \hat{\omega}\big(2\tilde{k} - 2j + 1\big) &= j + \hat{\omega}(j) + \hat{\omega}(j) + \tilde{k} - j + \hat{\omega}(\tilde{k} - j) + \hat{\omega}(\tilde{k} - j + 1) \\
&= \tilde{k} + \hat{\omega}(j) + \hat{\omega}(\tilde{k} - j) + \hat{\omega}(j) + \hat{\omega}(\tilde{k} - j + 1) \\
&\leq \tilde{k} + \hat{\omega}(\tilde{k}) - j + \hat{\omega}(\tilde{k} + 1) - j \\
&= \hat{\omega}(2\tilde{k} + 1) - 2j.
\end{aligned}$$

Case 4. $(2\tilde{k}+1, 2j+1)$.

For this last case, we use the fact that, due to the induction hypothesis, we have $\hat{\omega}(j) + \hat{\omega}(\tilde{k} - j) \leq \hat{\omega}(\tilde{k}) - j$, for all appropriate pairs $(\tilde{k}, j)$, and $\hat{\omega}(j+1) + \hat{\omega}(\tilde{k} - j) \leq$

$\hat{\omega}(\tilde{k}+1)-(j+1)$, for all appropriate pairs $(\tilde{k}+1, j+1)$. Thus, both inequalities hold for all j and $\tilde{k}$ with $j \leq \lfloor(\tilde{k}-1)/2\rfloor$ in particular. Hence,

$$\begin{aligned}\hat{\omega}(2j+1)+\hat{\omega}\big(2\tilde{k}-2j\big) &= j+\hat{\omega}(j)+\hat{\omega}(j+1)+\tilde{k}-j+\hat{\omega}(\tilde{k}-j)+\hat{\omega}(\tilde{k}-j)\\ &= \tilde{k}+\hat{\omega}(j)+\hat{\omega}(\tilde{k}-j)+\hat{\omega}(j+1)+\hat{\omega}(\tilde{k}-j)\\ &\leq \tilde{k}+\hat{\omega}(\tilde{k})-j+\hat{\omega}(\tilde{k}+1)-(j+1)\\ &= \hat{\omega}(2\tilde{k}+1)-2j-1.\end{aligned}$$

It remains to show that each appropriate pair is "reached" by the induction. To do so, we prove that every appropriate pair $(\tilde{k}, j)$ has an appropriate pair as its parent in the tree of Figure 5.1. From the way we constructed the tree, it is obvious that the parent of every pair $(\tilde{k}, j)$ is $\big(\lfloor\tilde{k}/2\rfloor, \lfloor j/2\rfloor\big)$. Since $(\tilde{k}, j)$ is appropriate, we have $j \leq \lfloor\tilde{k}/2\rfloor$, and thus

$$\lfloor j/2\rfloor \leq \left\lfloor\frac{\lfloor\tilde{k}/2\rfloor}{2}\right\rfloor.$$

This is exactly the definition of $\big(\lfloor\tilde{k}/2\rfloor, \lfloor j/2\rfloor\big)$ being an appropriate pair. □

Corollary 5.15. *For all $\tilde{k}, j \in \mathbb{N}^{\geq 0}$ with $j \leq \lfloor\tilde{k}/2\rfloor$, the term $j+\hat{\omega}(j)+\hat{\omega}(\tilde{k}-j)$ attains its maximum for $j = \lfloor\tilde{k}/2\rfloor$ and for $j = 0$. This maximum value is*

$$\max_{0\leq j\leq\lfloor\frac{\tilde{k}}{2}\rfloor}\{j+\hat{\omega}(j)+\hat{\omega}(\tilde{k}-j)\} = \hat{\omega}(\tilde{k}).$$

Proof. Due to Lemma 5.14, the above term never exceeds the value $\hat{\omega}(\tilde{k})$, i. e., for all j and $\tilde{k}$ with $0 \leq j \leq \lfloor\tilde{k}/2\rfloor$, we have $\hat{\omega}(j)+\hat{\omega}(\tilde{k}-j)+j \leq \hat{\omega}(\tilde{k})$. Then again, due to Corollary 5.12, the value $\hat{\omega}(\tilde{k})$ is actually attained by setting $j := \lfloor\tilde{k}/2\rfloor$ since $\hat{\omega}\big(\lfloor\tilde{k}/2\rfloor\big)+\hat{\omega}\big(\lceil\tilde{k}/2\rceil\big)+\lfloor\tilde{k}/2\rfloor = \hat{\omega}(\tilde{k})$. The fact that the value $\hat{\omega}(\tilde{k})$ is also attained for $j = 0$ can easily be seen since $0+\hat{\omega}(0)+\hat{\omega}(\tilde{k}-0) = 0+\hat{\omega}(\tilde{k}) = \hat{\omega}(\tilde{k})$. □

5.2.1.3 Analysis of the Algorithm

Having provided all the technical results we need, we finally want to analyze the expected number of correctly guessed bits of the online algorithm $\mathcal{A}$ with advice from Section 5.2.1.1.

Recall that the input string r for $\mathcal{A}$ is drawn uniformly at random from a set of instances $\mathcal{I} = \mathcal{I}_{\text{all}} \setminus \tilde{\mathcal{I}}$ that contains all strings of length n except for $\tilde{k}$ strings that are contained in $\tilde{\mathcal{I}}$. The number $\text{gain}_{\mathcal{A}}$ of bits that $\mathcal{A}$ guesses correctly on its input is a discrete random variable that can attain any integer value between 0 and n. Also recall that we can identify each set $\tilde{\mathcal{I}}$ of excluded strings with one unambiguous distribution ψ from Ψ and that therefore the expected gain of $\mathcal{A}$ is $\mathbb{E}_{\tilde{\mathcal{I}}}[\text{gain}_{\mathcal{A}}] = \mathbb{E}_{\psi}[\text{gain}_{\mathcal{A}}]$, when the set of excluded strings is $\tilde{\mathcal{I}}$ with the corresponding probability

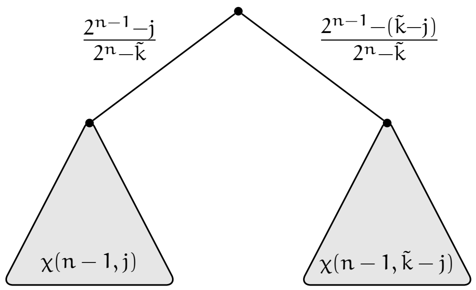

$\frac{2^{n-1}-j}{2^n-\tilde{k}}$
$\frac{2^{n-1}-(\tilde{k}-j)}{2^n-\tilde{k}}$
$\chi(n-1,j)$
$\chi(n-1,\tilde{k}-j)$

Figure 5.2. The expected number $\chi(n, \tilde{k})$ of correctly guessed bits on a worst-case set of instances consisting of strings of length n when $\tilde{k}$ strings are excluded. The root of the tree represents the guess of the first bit. Taking the left branch corresponds to guessing the bit r_1 correctly, which happens with probability $(2^{n-1} - j)/(2^n - \tilde{k})$ if j excluded strings start with the bit that occurs less often at position 1 among the excluded strings. The number of correctly guessed bits in this case is 1 plus the expected number $\chi(n-1, j)$ of correctly guessed bits on an instance consisting of all strings of length $n-1$ except for j excluded strings. Taking the right branch corresponds to making a wrong guess for r_1, which happens with probability $(2^{n-1} - (\tilde{k}-j))/(2^n - \tilde{k})$. The expected number of correctly guessed bits in this case can be calculated recursively as the expected number $\chi(n-1, \tilde{k}-j)$ of correctly guessed bits on an instance of length $n-1$ with $\tilde{k}-j$ excluded strings.

distribution ψ. We define the term $\chi(n, \tilde{k})$ to be the expected number of bits that $\mathcal{A}$ guesses correctly when r is drawn from a worst-case instance set $\mathcal{I} = \mathcal{I}_{\text{all}} \setminus \tilde{\mathcal{I}}$ that contains $k = 2^n - \tilde{k}$ strings of length n. Hence,

$$\chi(n, \tilde{k}) \coloneqq \min_{\substack{\tilde{\mathcal{I}} \subseteq \mathcal{I}_{\text{all}}, \\ |\tilde{\mathcal{I}}| = \tilde{k}}} \mathbb{E}_{\tilde{\mathcal{I}}}[\text{gain}_{\mathcal{A}}].$$

In what follows, we analyze the term $\chi(n, \tilde{k})$.

The value $\chi(n, \tilde{k})$ can be calculated recursively according to the tree shown in Figure 5.2. To see this, recall that $r = r_1 \ldots r_n$ is the binary string that is chosen uniformly at random from $\mathcal{I}$ as the input for our algorithm $\mathcal{A}$. Also recall that the bit that $\mathcal{A}$ outputs in round i is denoted by g_i. Hence, $\mathcal{A}$'s guess for the whole bit string is $g_1 \ldots g_n$.

Let us define $\tilde{k}_{1,0}$ to be the number of excluded strings that have a zero at position 1 and $\tilde{k}_{1,1} = \tilde{k} - \tilde{k}_{1,0}$ to be the number of excluded strings that have a one at position 1. We know that $0 \leq \tilde{k}_{1,0}, \tilde{k}_{1,1} \leq \tilde{k}$. Let $j \coloneqq \min\{\tilde{k}_{1,0}, \tilde{k}_{1,1}\}$. According to its definition in Section 5.2.1.1, $\mathcal{A}$ chooses as its guess g_1 the bit that appears more often at position 1 among the strings in C_1. Hence, we know that j excluded strings have the bit g_1 and $\tilde{k} - j$ excluded strings have the bit $1 - g_1$ at position 1. On the other hand, there are $2^{n-1} - j$ strings in C_1 whose first bit is g_1 and $2^{n-1} - (\tilde{k} - j)$ strings whose first bit is $1 - g_1$. The probability that

$g_1 = r_1$ is hence $(2^{n-1} - j)/(2^n - \tilde{k})$. If $\mathcal{A}$ guesses g_1 correctly, $\mathcal{A}$ computes C_2 by removing the strings from C_1 that start with $1 - r_1 = 1 - g_1$. All binary strings starting with $1 - g_1$ remain as candidates except for j strings from $\tilde{\mathcal{I}}$. These are $2^{n-1} - j$ strings. In this case, the expected number of correctly guessed bits on the remaining instance can be calculated as $\chi(n-1, j)$.

If, on the other hand, $\mathcal{A}$ makes the wrong guess in round 1, which happens with probability $(2^{n-1} - (\tilde{k} - j))/(2^n - \tilde{k})$, the 2^{n-1} strings starting with $1 - r_1 = g_1$ are excluded from C_1. Then C_2 contains all strings of length n that start with $r_1 = 1-g_1$ except for the $\tilde{k}-j$ strings of $\tilde{\mathcal{I}}$. Hence, all binary strings remain as candidates for r that start with $1 - g_1$, except for $\tilde{k} - j$ excluded strings from $\tilde{\mathcal{I}}$. In total, these are $2^{n-1} - (\tilde{k} - j)$ strings. Hence, the expected number of correctly guessed bits on this remaining instance can be calculated recursively as $\chi(n-1, \tilde{k} - j)$.

Now, considering the one additional correctly guessed bit every time we take a left branch in the tree (which complies with guessing the corresponding bit correctly), we obtain the following recursive formula for $\chi(n, \tilde{k})$:

$$\chi(n, \tilde{k}) = \min_{0 \le j \le \lfloor \frac{\tilde{k}}{2} \rfloor} \left\{ \frac{2^{n-1} - j}{2^n - \tilde{k}} (1 + \chi(n-1, j)) + \frac{2^{n-1} - (\tilde{k} - j)}{2^n - \tilde{k}} \chi(n-1, \tilde{k} - j) \right\}, \tag{5.13}$$

for all $n \ge 1$ and all $\tilde{k} \ge 0$, and $\chi(0, \tilde{k}) = 0$ for all $\tilde{k} \ge 0$.

Lemma 5.16. *Let $n \in \mathbb{N}$ and let $\mathcal{I} = \mathcal{I}_{all} \setminus \tilde{\mathcal{I}}$ be a worst-case set of instances that contains all strings of length n except for $0 \le \tilde{k} \le 2^n - 1$ strings that are contained in $\tilde{\mathcal{I}}$. Furthermore, let $\mathcal{A}$ be an online algorithm with advice that chooses as its guess g_i a bit that appears in at least half of the strings in C_i at position i in every round i, where $1 \le i \le n$. Then the expected number of correctly guessed bits of $\mathcal{A}$ on $\mathcal{I}$ in the monolog model is given by*

$$\chi(n, \tilde{k}) = \frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}}. \tag{5.14}$$

Proof. We prove (5.14) by induction n and $\tilde{k}$. We validate that (5.14) holds for $n = 1$ and all $\tilde{k}$ with $\tilde{k} \le 2^n - 1 = 1$. If $\tilde{k} = 0$, according to (5.13), we have

$$\begin{aligned}
\chi(1, 0) &= \min_{0 \le j \le 0} \left\{ \frac{2^0 - j}{2^1 - 0} \cdot (1 + \chi(0, j)) + \frac{2^0 - (0 - j)}{2^1 - 0} \cdot \chi(0, 0 - j) \right\} \\
&= \frac{1 - 0}{2} \cdot (1 + \chi(0, 0)) + \frac{1 - 0}{2} \cdot \chi(0, 0) \\
&= \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 0 \\
&= \frac{1}{2},
\end{aligned}$$

and according to (5.14), this coincides with

$$\frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}} = \frac{1 \cdot 2^0 - \hat{\omega}(0)}{2^1 - 0} = \frac{1-0}{2} = \frac{1}{2}.$$

If $\tilde{k} = 1$, according to (5.13), we have

$$\begin{aligned}\chi(1,1) &= \min_{0 \le j \le 0} \left\{ \frac{2^0 - j}{2^1 - 1} \cdot (1 + \chi(0,j)) + \frac{2^0 - (1-j)}{2^1 - 1} \cdot \chi(0, 1-j) \right\} \\ &= \frac{1-0}{1} \cdot (1 + \chi(0,0)) + \frac{1-1}{1} \cdot \chi(0,1) \\ &= 1 \cdot 1 + 0 \\ &= 1,\end{aligned}$$

and according to (5.14), this coincides with

$$\frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}} = \frac{1 \cdot 2^0 - \hat{\omega}(1)}{2^1 - 1} = \frac{1-0}{1} = 1.$$

Now that we have proven the base case, let us assume as an induction hypothesis that

$$\chi(n-1, j) = \frac{(n-1)2^{n-2} - \hat{\omega}(j)}{2^{n-1} - j}, \qquad \text{for all } j \le 2^{n-1} - 1, \text{ and} \tag{5.15}$$

$$\chi(n-1, \tilde{k} - j) = \frac{(n-1)2^{n-2} - \hat{\omega}(\tilde{k} - j)}{2^{n-1} - (\tilde{k} - j)}, \quad \text{for all } \tilde{k} - j \le 2^{n-1} - 1. \tag{5.16}$$

Next, we will show that (5.14) also holds for all $\tilde{k} \le 2^n - 1$. Before we do so, let us calculate the two terms of the sum in (5.13). Due to the induction hypothesis (5.15), we have

$$\begin{aligned}\frac{2^{n-1} - j}{2^n - \tilde{k}} \cdot (1 + \chi(n-1, j)) &= \frac{2^{n-1} - j}{2^n - \tilde{k}} \cdot \left(1 + \frac{(n-1)2^{n-2} - \hat{\omega}(j)}{2^{n-1} - j} \right) \\ &= \frac{2^{n-1} - j}{2^n - \tilde{k}} \cdot \frac{2^{n-1} - j + (n-1)2^{n-2} - \hat{\omega}(j)}{2^{n-1} - j},\end{aligned}$$

yielding

$$\frac{2^{n-1} - j}{2^n - \tilde{k}} \cdot (1 + \chi(n-1, j)) = \frac{2^{n-1} - j + (n-1)2^{n-2} - \hat{\omega}(j)}{2^n - \tilde{k}}. \tag{5.17}$$

Similarly, using (5.16), we obtain

$$\frac{2^{n-1} - (\tilde{k} - j)}{2^n - \tilde{k}} \cdot \chi(n-1, \tilde{k} - j) = \frac{(n-1)2^{n-2} - \hat{\omega}(\tilde{k} - j)}{2^n - \tilde{k}}. \tag{5.18}$$

Now we can analyze $\chi(n,\tilde{k})$ for $\tilde{k} \le 2^n - 2$. We have

$$\begin{aligned}
\chi(n,\tilde{k}) &= \min_{0\le j\le \lfloor \frac{\tilde{k}}{2}\rfloor} \left\{ \frac{2^{n-1}-j}{2^n-\tilde{k}}(1+\chi(n-1,j)) + \frac{2^{n-1}-(\tilde{k}-j)}{2^n-\tilde{k}}\chi(n-1,\tilde{k}-j) \right\} \\
&= \min_{0\le j\le \lfloor \frac{\tilde{k}}{2}\rfloor} \left\{ \frac{2^{n-1}-j+(n-1)2^{n-2}-\hat{\omega}(j)}{2^n-\tilde{k}} + \frac{(n-1)2^{n-2}-\hat{\omega}(\tilde{k}-j)}{2^n-\tilde{k}} \right\} \qquad (5.19) \\
&= \min_{0\le j\le \lfloor \frac{\tilde{k}}{2}\rfloor} \left\{ \frac{2^{n-1}\cdot\left(1+\frac{n-1}{2}+\frac{n-1}{2}\right)-\hat{\omega}(\tilde{k}-j)-\hat{\omega}(j)-j}{2^n-\tilde{k}} \right\} \\
&= \min_{0\le j\le \lfloor \frac{\tilde{k}}{2}\rfloor} \left\{ \frac{n\cdot 2^{n-1}-\left(\hat{\omega}(j)+\hat{\omega}(\tilde{k}-j)+j\right)}{2^n-\tilde{k}} \right\} \\
&= \frac{n\cdot 2^{n-1}-\max_{0\le j\le\lfloor\tilde{k}/2\rfloor}\{\hat{\omega}(j)+\hat{\omega}(\tilde{k}-j)+j\}}{2^n-\tilde{k}} \\
&= \frac{n\cdot 2^{n-1}-\hat{\omega}(\tilde{k})}{2^n-\tilde{k}}, \qquad (5.20)
\end{aligned}$$

where we used (5.17) and (5.18) for (5.19), and Corollary 5.15 for (5.20).

This only proves (5.14) for $\tilde{k} = (\tilde{k}-j)+j \le 2\cdot(2^{n-1}-1) = 2^n-2$ since we used (5.15) and (5.16) in the proof, which require both j and $\tilde{k}-j$ to be at most $2^{n-1}-1$. For $\tilde{k} = 2^n-1$, we make the following considerations. All strings are excluded except for one, r, and thus $\mathcal{A}$ knows r and can guess each bit correctly; hence, $\chi(n,2^n-1) = n$. Applying Observation 5.5 yields

$$\frac{n\cdot 2^{n-1}-\hat{\omega}(\tilde{k})}{2^n-\tilde{k}} = \frac{n\cdot 2^{n-1}-\hat{\omega}(2^n-1)}{2^n-\tilde{k}} = \frac{n\cdot 2^{n-1}-\left(n\cdot 2^{n-1}-n\right)}{2^n-(2^n-1)} = n.$$

Therefore, the claim (5.14) holds for all $\tilde{k}$ and n with $\tilde{k} \le 2^n-1$. □

Let us reason about the meaning of what we have just proven. We have seen that we do not know exactly how the tree representing the expected number of correctly guessed bits per round looks like. But we know that the number of correctly guessed bits of $\mathcal{A}$ in round i is smallest when all strings from $\tilde{\mathcal{I}}$ that have the prefix $[r]^{i-1}$ have the same bit at position i, or the number of strings from $\tilde{\mathcal{I}}$ with a one and the number of those with a zero at position i differ by at most 1.

To explain this more formally, recall the definition of the set of candidates $C_i = \{s \in \mathcal{I} \mid [s]^{i-1} = [r]^{i-1}\}$, and let us define the set $\tilde{C}_i := \{s \in \tilde{\mathcal{I}} \mid [s]^{i-1} = [r]^{i-1}\}$ with $|\tilde{C}_i| := \tilde{k}_i$. Let the number of strings from $\tilde{C}_i$ that have a zero at position i be $\tilde{k}_{i,0}$ and the number of those that have a one at position i be $\tilde{k}_{i,1}$. Also, let $j_i := \min\{\tilde{k}_{i,0}, \tilde{k}_{i,1}\}$ for $1 \le i \le n$.

The astonishing result of this proof, combined with Corollary 5.15, is that the worst-case expected number of correctly guessed bits of $\mathcal{A}$ occurs if the function $\hat{\omega}(j_i) + \hat{\omega}(\tilde{k}_i - j_i) + j_i$ attains is maximum in each round i. According to Corollary 5.15, this is the case for $j_i = \lfloor \tilde{k}_i/2 \rfloor$ and $j_i = 0$. Hence, the expected number of correctly guessed bits of $\mathcal{A}$ is minimal if, for each i with $1 \le i \le n$,

(a) $\lfloor \tilde{k}_i/2 \rfloor$ excluded strings have a one and $\lceil \tilde{k}_i/2 \rceil$ strings have a zero at position i (or vice versa) or

(b) all excluded strings have a one and none have a zero at position i (or vice versa).

Theorem 5.17. *In the monolog model, given a probabilistic adversary that excludes the set $\tilde{\mathcal{I}}$ containing $\tilde{k}$ strings of length n from the set $\mathcal{I}_{all}$ of possible strings and chooses one of the remaining strings from $\mathcal{I} = \mathcal{I}_{all} \setminus \tilde{\mathcal{I}}$ uniformly at random, there is an online algorithm $\mathcal{A}$ with advice for the bit string guessing problem with a probabilistic adversary that guesses*

$$\frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}}$$

bits correctly and thus at most

$$\frac{\hat{\omega}(k)}{k}$$

bits incorrectly in expectation, using $\left\lceil \log \binom{2^n}{\tilde{k}} \right\rceil$ advice bits.

Proof. The number of advice bits needed is indicated in Section 5.2.1.1 in the description of algorithm $\mathcal{A}$, i. e., $\mathcal{A}$ is told the set $\tilde{\mathcal{I}}$. The number of correctly guessed bits follows from Lemma 5.16. Using Observation 5.9, we can derive the expected number of incorrectly guessed bits as

$$n - \frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}} = n - \left(n - \frac{\hat{\omega}(k)}{k} \right) = \frac{\hat{\omega}(k)}{k}. \qquad \square$$

Corollary 5.18. *For $\tilde{k} = 0$ excluded strings, the expected number of correctly guessed bits of $\mathcal{A}$ is $n/2$, while $\mathcal{A}$ uses no advice bits at all.*

Proof. Plugging $\tilde{k} = 0$ into Theorem 5.17, we obtain

$$\left\lceil \log \binom{2^n}{0} \right\rceil = \log(1) = 0$$

as the number of advice bits and

$$\chi(n, 0) = \frac{n \cdot 2^{n-1} - \hat{\omega}(0)}{2^n - 0} = \frac{n \cdot 2^{n-1}}{2^n} = \frac{n}{2}$$

as the expected number of correctly guessed bits. $\square$

The previous corollary states that if each bit is 0 or 1 with probability exactly 1/2 and $\mathcal{A}$ does not get any advice, it guesses half of the bits correctly in expectation, which is in accordance with our intuition.

Corollary 5.19. *For one excluded string ($\tilde{k} = 1$), the expected number of correctly guessed bits of $\mathcal{A}$ is $n/2 \cdot (1 + 1/(2^n - 1))$, while $\mathcal{A}$ uses n advice bits.*

Proof. Plugging $\tilde{k} = 1$ into Theorem 5.17, we obtain

$$\left\lceil \log \binom{2^n}{1} \right\rceil = \log(2^n) = n$$

as the number of advice bits and

$$\chi(n, 1) = \frac{n \cdot 2^{n-1} - \hat{\omega}(1)}{2^n - 1} = \frac{n \cdot 2^{n-1}}{2^n - 1} = \frac{n \cdot 2^{n-1}}{2^n} \cdot \frac{2^n}{2^n - 1} = \frac{n}{2} \cdot \left(1 + \frac{1}{2^n - 1}\right)$$

as the expected number of correctly guessed bits. □

Corollary 5.20. *For $\tilde{k} = 2^n - 1$ excluded strings, the expected number of correctly guessed bits of $\mathcal{A}$ is n, while $\mathcal{A}$ uses n advice bits.*

Proof. Plugging $\tilde{k} = 2^n - 1$ into Theorem 5.17, we obtain

$$\left\lceil \log \binom{2^n}{2^n - 1} \right\rceil = \log(2^n) = n$$

as the number of advice bits and

$$\chi(n, 2^n - 1) = \frac{n \cdot 2^{n-1} - \hat{\omega}(2^n - 1)}{2^n - 2^n + 1} = \frac{n \cdot 2^{n-1} - n \cdot 2^{n-1} + n}{1} = n$$

as the expected number of correctly guessed bits. □

In this last case of $\tilde{k} = 2^n - 1$ excluded strings in Corollary 5.20, the instance set from which the input is chosen randomly has size 1. This corresponds to the classical advice model, in which the adversary can only choose the input string for the algorithm, without any randomness being involved. This case has already been analyzed by Böckenhauer et al. [BHK+14], who showed that n advice bits are necessary and sufficient to be optimal. As the oracle knows the input string r, it can write r as the advice string onto the tape, and it is easy to see that there is an optimal algorithm reading these n advice bits that is optimal. Conversely, any algorithm reading $b < n$ advice bits cannot be optimal on all input strings, because there are 2^n different inputs and $2^b < 2^n$ advice strings, and thus the oracle has to give the same advice string to the algorithm for two different inputs. Since we can view an algorithm with a fixed advice string as a deterministic algorithm and this algorithm behaves exactly the same on two different input strings, it cannot be optimal on both of them. Hence, any algorithm needs at least n advice bits to be optimal.

5.2.1.4 Optimality

In this section, we prove that the algorithm $\mathcal{A}$ with advice that we introduced is optimal. To this end, we have to define the term "optimality" in the setting of a probabilistic adversary. Note that the question whether or not an algorithm is optimal depends solely on the quality of the computed solution and not on any quantitive aspects of $\mathcal{A}$ like the number of advice bits used, the running time, or the space complexity.

Definition 5.21. *Consider an online minimization problem* P *and an online algorithm* $\mathcal{A}$ *with advice playing against a probabilistic adversary. Let* $\mathcal{I}_{all}$ *be the set of instances from which the input for* $\mathcal{A}$ *is picked according to a certain probability distribution* $\phi\colon \mathcal{I}_{all} \to [0,1]$. *Algorithm* $\mathcal{A}$ *is called* ϕ-optimal for P *if there is no algorithm* $\mathcal{B}$ *solving* P *with a smaller expected cost with respect to* ϕ *than* $\mathcal{A}$*, i. e.,* $\mathbb{E}_\phi[\mathrm{gain}_\mathcal{A}] \le \mathbb{E}_\phi[\mathrm{gain}_\mathcal{B}]$ *for all algorithms* $\mathcal{B}$.

The algorithm $\mathcal{A}$ *is called* Φ-optimal for P *if it is* ϕ*-optimal for all probability distributions* $\phi \in \Phi$*, i. e.,* $\mathbb{E}_\phi[\mathrm{gain}_\mathcal{A}] \le \mathbb{E}_\phi[\mathrm{gain}_\mathcal{B}]$ *for all algorithms* $\mathcal{B}$ *and all* $\phi \in \Phi$.

We defined Φ-optimality for general probability distributions, but from now on, we will constrain ourselves to the class Ψ of probability distributions we defined in the beginning of the chapter.

Recall the definition of the set C_i of candidates that we gave in Section 5.2.1.1; we defined C_i to be the set of strings from $\mathcal{I}$ that might still be the to-be-guessed string r in round i, according to the set $\mathcal{I}$ and the prefix $[r]^{i-1}$ of r of length $i-1$. The following lemma states that an algorithm is Ψ-optimal if and only if, in each round i, it chooses a bit that occurs in at least half of the strings in C_i at position i.

Lemma 5.22. *In the monolog model, an online algorithm for the bit string guessing problem with a probabilistic adversary that excludes* $\tilde{k}$ *strings from an instance consisting of* 2^n *strings of length* n *is* Ψ*-optimal if and only if it chooses its guess* g_i *from the set*

$$\arg\max_{s_i} \left\{ |\{s \in C_i \mid s_i = 0\}|, |\{s \in C_i \mid s_i = 1\}| \right\}$$

in each round i.

Proof. Let $\tilde{\mathcal{I}}$ be the set of excluded strings corresponding to the probability distribution $\psi \in \Psi$ chosen by the adversary. Let us consider a set of input strings $\mathcal{I} = \mathcal{I}_{\mathrm{all}} \setminus \tilde{\mathcal{I}}$ that consists of all strings of length n except for the $\tilde{k}$ excluded strings from $\tilde{\mathcal{I}}$. Furthermore, consider an arbitrary but fixed algorithm $\mathcal{A}$ that chooses its guess g_i from $\arg\max_{s_i} \left\{ |\{s \in C_i \mid s_i = 0\}|, |\{s \in C_i \mid s_i = 1\}| \right\}$ in each round i, and an algorithm $\mathcal{B}$ whose output differs from $\mathcal{A}$'s output in round i for some arbitrary but fixed i. We show that the expected number of correctly guessed bits of $\mathcal{B}$ on $\mathcal{I}$ cannot exceed the expected number of correctly guessed bits of $\mathcal{A}$ on $\mathcal{I}$. It suffices to show that the expected number of correctly guessed bits by $\mathcal{B}$ in this round i cannot be larger than those of $\mathcal{A}$. This is because, independent of the algorithms' guesses in round i, the same strings are removed from the set of remaining candidates, hence the initial situation in round $i+1$ is the same for both algorithms, and all following decisions in subsequent rounds are independent of round i.

Since the outputs of $\mathcal{A}$ and $\mathcal{B}$ differ in round i, we know that $\mathcal{B}$'s guess in round i is $1-g_i$, i. e., the bit that appears less often (or at most with the same abundance as g_i). Let us define $o_i \geq |C_i|/2$ to be the number of occurrences of the bit g_i at position i among the strings in C_i and $\tilde{o}_i \leq |C_i|/2$ the number of occurrences of $1-g_i$. If $o_i = \tilde{o}_i = |C_i|/2$, the expected number of correctly guessed bits in round i is the same for both $\mathcal{A}$ and $\mathcal{B}$. It gets more interesting when $\tilde{o}_i < o_i$ and thus $\tilde{o}_i < |C_i|/2$. Then the expected number of correctly guessed bits of $\mathcal{A}$ in round i can be calculated as

$$\Pr(r_i = g_i) \cdot 1 + \Pr(r_i = 1-g_i) \cdot 0 = \frac{o_i}{|C_i|} > \frac{|C_i|/2}{|C_i|} = \frac{1}{2},$$

and, analogously, the expected number of correctly guessed bits of $\mathcal{B}$ in round i is

$$\Pr(r_i = 1-g_i) \cdot 1 + \Pr(r_i = g_i) \cdot 0 = \frac{\tilde{o}_i}{|C_i|} < \frac{|C_i|/2}{|C_i|} = \frac{1}{2}.$$

We have seen that, only if both bits occur equally often in round i, both possible guesses lead to the same expected number of correctly guessed bits for round i. In such rounds any algorithm makes an optimal guess. On the other hand, we know that, in each round i in which one bit occurs more often than the other one among the strings in C_i, any algorithm has to guess the bit with larger occurrence to be Ψ-optimal. Hence, in such rounds, any algorithm that makes a different guess than $\mathcal{A}$ cannot be optimal. We conclude that $\mathcal{A}$ must be ψ-optimal with respect to all probability distributions $\psi \in \Psi$, and thus $\mathcal{A}$ is Ψ-optimal. Since we defined $\mathcal{A}$ to be an arbitrary algorithm that chooses its guess in round i from $\arg\max_{s_i} \big\{|\{s \in C_i \mid s_i = 0\}|, |\{s \in C_i \mid s_i = 1\}|\big\}$, any algorithm with this property is Ψ-optimal. □

Theorem 5.23. *Algorithm $\mathcal{A}$ from Section 5.2.1.1 is Ψ-optimal for 2-GUESS in the monolog model with a probabilistic adversary that excludes $\tilde{k}$ strings from an instance set consisting of 2^n strings of length n. Hence, the expected number of correctly guessed bits of any Ψ-optimal algorithm playing against such an adversary is*

$$\frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}},$$

and the number of incorrectly guessed bits is at most

$$\frac{\hat{\omega}(k)}{k}.$$

Proof. For each round i, let $k_{i,1}$ be the number of occurrences of the bit 1 at position i among the strings in C_i. In Section 5.2.1.1, we defined $\mathcal{A}$'s guess in round i, when the set of remaining candidates is C_i, to be

$$g_i = \begin{cases} 1 & \text{if } k_{i,1} \geq |C_i|/2, \\ 0 & \text{otherwise.} \end{cases}$$

Therefore, for each guess g_i in round i of $\mathcal{A}$, we have $g_i \in \arg\max_{s_i} \{|\{s \in C_i \mid s_i = 0\}|, |\{s \in C_i \mid s_i = 1\}|\}$. Hence, due to Lemma 5.22, $\mathcal{A}$ is Ψ-optimal. As a consequence, each Ψ-optimal algorithm must guess as many bits correctly in expectation as $\mathcal{A}$, and as we know from Theorem 5.17, this number of bits is exactly

$$\frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}}.$$

□

5.2.2 Lower Bound

Recall the notation $[t]^i$ for the prefix of length i of a string t of length n, for every i with $0 \leq i \leq n$. Furthermore, let us denote the set of all 2^n binary strings of length n by $T_\varepsilon^{(n)} = \mathcal{I}_{\text{all}}$. Whenever the length of the strings considered is clear from the context, we leave out the superscript n and write T_ε instead. The ε in the index is used to indicate that all strings from T_ε share the common prefix ε. In general, for each set $S \subseteq T_\varepsilon$, each i with $0 \leq i \leq n$, and each prefix $[t]^i \in \{0,1\}^i$, let us denote the set of all binary strings from S with the same common prefix $[t]^i$ by $S_{[t]^i}$.

Let us call a set S of strings *balanced* if, in each of the sets $S_{[t]^{i-1}}$ for each string t and every i with $1 \leq i \leq n$, the number of strings with a 0 at position i and the number of strings with a 1 at position i differ by at most 1; otherwise, we call such a set *unbalanced*. For an example of a balanced and an unbalanced set, see Table 5.1.

Let us denote by $\beta(\tilde{k}, n)$ the number of balanced sets that are subsets of $T_\varepsilon^{(n)}$ for $n \geq 1$ and that have size $\tilde{k}$ with $0 \leq \tilde{k} \leq 2^n - 1$. We want to analyze the number of those balanced sets contained in $T_\varepsilon^{(n)}$.

Lemma 5.24. *The number of balanced sets of size $\tilde{k}$ that are subsets of $T_\varepsilon^{(n)}$ is*

$$\beta(\tilde{k}, n) = 2^{\tilde{k} \cdot n - 2 \cdot \hat{\omega}(\tilde{k})}.$$

Proof. We show the claim by induction on $\tilde{k}$ and n. We have to consider three different base cases. At first, let us consider the case $\tilde{k} = 0$ and $n \geq 1$. There is only one balanced set of size 0, namely the empty set, so we have $\beta(0, n) = 1$. This

$[t]^{i-1}$	ε	0	1	00	01	10	11
	001	001	001	001	001	001	001
$S_{[t]^{i-1}}$	110	110	110	110	110	110	110
	111	111	111	111	111	111	111
	001	001	001	001	001	001	001
$S'_{[t]^{i-1}}$	101	101	101	101	101	101	101
	111	111	111	111	111	111	111

Table 5.1. Example of an unbalanced set $S = \{001, 110, 111\}$ and a balanced set $S' = \{001, 101, 111\}$. For every i with $1 \leq i \leq n$, and every prefix $[t]^{i-1}$, the sets $S_{[t]^{i-1}}$ and $S'_{[t]^{i-1}}$ are shown. The empty prefix is denoted by ε. In the sets $S_{[t]^{i-1}}$ and $S'_{[t]^{i-1}}$, the relevant position of every string is colored in blue (this is position i). The cell for the set S_1 is marked with a blue background color. For this set, the relevant position is 2, and both strings 110 and 111 contained in S_1 have a 1 at this position. Thus, the numbers of zeros and ones differ by more than one, and thus the set S is unbalanced. The set S', on the other hand, is balanced.

coincides with $2^{0 \cdot n - 2 \cdot \hat{\omega}(0)} = 2^0 = 1$. Now let us consider the case $\tilde{k} = 1$ and $n \geq 1$. There are 2^n strings in $T_\varepsilon^{(n)}$. No matter which string s we choose, the set $\{s\}$ is always balanced, and hence $\beta(1, n) = 2^n$. This coincides with $2^{1 \cdot n - 2 \cdot \hat{\omega}(1)} = 2^n$. The last base case we have to consider is the case $\tilde{k} = 2^n - 1$ and $n \geq 1$. Every way of choosing $2^n - 1$ out of the 2^n strings in $T_\varepsilon^{(n)}$ obviously yields a balanced set. There are $\beta(2^n - 1, n) = \binom{2^n}{2^n - 1} = 2^n$ ways to choose such a set, which coincides with $2^{(2^n - 1) \cdot n - 2 \cdot \hat{\omega}(2^n - 1)} = 2^{2^n \cdot n - n - 2 \cdot (n \cdot 2^{n-1} - n)} = 2^{2^n \cdot n - n - 2^n \cdot n + 2n} = 2^n$.

Now let us discuss how a balanced set S of size $2\tilde{k}$ containing strings of length $n + 1$ can be constructed. Each such set S must have two properties.

(a) The set $S_0 \subseteq S$ of strings from S starting with a 0 and the set $S_1 \subseteq S$ of strings from S starting with a 1 must contain $\tilde{k}$ strings each, and

(b) when removing the leading bits from all strings in S_0 and S_1, the resulting sets $\hat{S}_0$ and $\hat{S}_1$ must be balanced.

If (a) or (b) do not hold, S cannot be balanced. Hence, to construct S, let us start with an arbitrary balanced set containing $\tilde{k}$ strings of length n and add a leading 0 to all strings in this set. Let the resulting set be denoted by S_0. Now let us repeat this with another balanced set (not necessarily a different one) containing $\tilde{k}$ strings of length n, but add a leading 1 instead of a 0 to all strings in this set, and call the resulting set S_1. Now consider $S := S_0 \cup S_1$. We argue that S must be balanced as follows. In the set of strings from S that share the same common prefix ε, there are exactly as many strings with a 0 at position 1 as there are strings with a 1 at position 1, namely $\tilde{k}$. Thus, the numbers of these strings differ by at most 1, as demanded in the definition of a balanced set. Moreover, the set of strings from S with the same common prefix 0 is exactly the set S_0, and those with the same prefix 1 are the strings from S_1. Both these sets S_0 and S_1 are balanced, and thus the condition also holds for all other sets $S_{[t]^{i-1}}$ of strings, for all common prefixes $[t]^{i-1}$. Therefore, the set S must be balanced, too.

Hence, a balanced set S containing $2\tilde{k}$ strings of length $n+1$ can be constructed by taking two arbitrary balanced sets, each containing $\tilde{k}$ strings of length n, adding a leading 0 to all strings from one of them and a leading 1 to all strings from the other one. The sets constructed in this way are no more and no less than those satisfying the two properties (a) and (b) given above. The number of balanced sets of size $\tilde{k}$ containing strings of length $n+1$ can hence be computed recursively as $\beta(2\tilde{k}, n+1) = (\beta(\tilde{k}, n))^2$.

For odd sizes $2\tilde{k}+1$, we can use a similar construction, but we have to be a bit more careful, because $\tilde{k}$ strings from S start with a 0 and $\tilde{k}+1$ start with a 1, or vice versa. Hence, setting $S := S_0 \cup S_1$ again, we have to consider all possibilities to choose S_0 and S_1 with either $|S_0| = \tilde{k}$ and $|S_1| = \tilde{k}+1$ or $|S_0| = \tilde{k}+1$ and $|S_1| = \tilde{k}$. This yields that the number of balanced sets of size $2\tilde{k}+1$ with strings of length $n+1$ is $\beta(2\tilde{k}+1, n+1) = \beta(\tilde{k}, n) \cdot \beta(\tilde{k}+1, n) \cdot 2$.

Using the induction hypothesis for $\beta(\tilde{k}, n)$ for (5.21) and Lemma 5.11 for (5.22), we obtain

$$\begin{aligned}
\beta(2\tilde{k}, n+1) &= (\beta(\tilde{k}, n))^2 \\
&= 2^{2(\tilde{k}n - 2\cdot\hat{\omega}(\tilde{k}))} &\quad (5.21)\\
&= 2^{2\tilde{k}n - 4\cdot\hat{\omega}(\tilde{k})} \\
&= 2^{2\tilde{k}n + 2\tilde{k} - 2\cdot\hat{\omega}(2\tilde{k})} &\quad (5.22)\\
&= 2^{2\tilde{k}\cdot(n+1) - 2\cdot\hat{\omega}(2\tilde{k})}.
\end{aligned}$$

Using both induction hypotheses for $\beta(\tilde{k}, n)$ and $\beta(\tilde{k}+1, n)$ in (5.23) and Lemma 5.11 for (5.24), we obtain

$$\begin{aligned}
\beta(2\tilde{k}+1, n+1) &= \beta(\tilde{k}, n) \cdot \beta(\tilde{k}+1, n) \cdot 2 \\
&= 2^{\tilde{k}n - 2\cdot\hat{\omega}(\tilde{k})} \cdot 2^{(\tilde{k}+1)n - 2\cdot\hat{\omega}(\tilde{k}+1)} \cdot 2 &\quad (5.23)\\
&= 2^{\tilde{k}n + (\tilde{k}+1)n - 2\cdot\hat{\omega}(\tilde{k}) - 2\cdot\hat{\omega}(\tilde{k}+1) + 1} \\
&= 2^{(2\tilde{k}+1)n - 2\cdot(\hat{\omega}(2\tilde{k}+1) - \tilde{k}) + 1} &\quad (5.24)\\
&= 2^{(2\tilde{k}+1)n - 2\cdot\hat{\omega}(2\tilde{k}+1) + (2\tilde{k}+1)} \\
&= 2^{(2\tilde{k}+1)(n+1) - 2\cdot\hat{\omega}(2\tilde{k}+1)}.
\end{aligned}$$

Let us argue why all relevant $\beta(\tilde{k}, n)$ can be generated with this construction method. We have to construct $\beta(\tilde{k}, n)$ for every n and $\tilde{k}$ with $0 \leq \tilde{k} \leq 2^n - 1$ that is not already covered by one of the base cases. Hence, for even $\tilde{k}$, all those satisfying $2 \leq \tilde{k} \leq 2^n - 2$ have to be generated. For this construction, we need $\beta(\tilde{k}/2, n-1)$, which has to be given as a base case or must be generated itself. However, this is possible since

$$\begin{aligned} 2 \leq \tilde{k} \ &\leq 2^n - 2 \\ \iff 1 \leq \frac{\tilde{k}}{2} &\leq 2^{n-1} - 1. \end{aligned}$$

For odd $\tilde{k}$, we have to construct all $\beta(\tilde{k}, n)$ satisfying $3 \leq \tilde{k} \leq 2^n - 3$. To construct $\beta(\tilde{k}, n)$, we need $\beta((\tilde{k}-1)/2, n-1)$ and $\beta((\tilde{k}+1)/2, n-1)$. These can also either be generated or are already given as a base case since

$$\begin{aligned} 3 \leq \tilde{k} \quad\ &\leq 2^n - 3 \\ \iff 2 \leq \tilde{k} - 1 \ &\leq 2^n - 4 \\ \iff 1 \leq \frac{\tilde{k}-1}{2} &\leq 2^{n-1} - 2 \end{aligned}$$

and

$$\begin{aligned} 3 \leq \tilde{k} \quad\ &\leq 2^n - 3 \\ \iff 4 \leq \tilde{k} + 1 \ &\leq 2^n - 2 \\ \iff 2 \leq \frac{\tilde{k}+1}{2} &\leq 2^{n-1} - 1. \end{aligned}$$

□

Hence, there are $\beta(\tilde{k}, n) = 2^{\tilde{k} \cdot n - 2 \cdot \hat{\omega}(\tilde{k})}$ possibilities to choose a balanced set of size $\tilde{k}$ containing binary strings of length n. In the following, we show that each Ψ-optimal algorithm with advice playing against an adversary that chooses only balanced sets as the set $\tilde{\mathcal{I}}$ needs one advice string for each possible choice of $\tilde{\mathcal{I}}$. Consequently, as $\beta(\tilde{k}, n)$ different advice strings are already necessary to be Ψ-optimal with such a weak adversary, $\beta(\tilde{k}, n)$ advice strings are certainly also necessary to be Ψ-optimal when the adversary can choose an arbitrary set as $\tilde{\mathcal{I}}$.

Consider two different balanced sets U and V, each containing $\tilde{k}$ strings of length n, and two corresponding deterministic algorithms A_U and A_V, which are optimal on the input sets $\mathcal{I}_{\text{all}} \setminus U$ and $\mathcal{I}_{\text{all}} \setminus V$, respectively.

We now show that no deterministic algorithm can be optimal on both instance sets $\mathcal{I}_{\text{all}} \setminus U$ and $\mathcal{I}_{\text{all}} \setminus V$. We do so by proving that A_U (being optimal on $\mathcal{I}_{\text{all}} \setminus U$) cannot be optimal on $\mathcal{I}_{\text{all}} \setminus V$ at the same time. Hence, a deterministic algorithm behaving optimally on the one set of instances will certainly be suboptimal on the other one.

In the following, we prove two claims. One states that

$$\mathbb{E}_U[\mathrm{gain}_{A_U}] = \mathbb{E}_V[\mathrm{gain}_{A_V}] = \frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}}. \tag{5.25}$$

The second one states that

$$\mathbb{E}_V[\mathrm{gain}_{A_U}] < \mathbb{E}_U[\mathrm{gain}_{A_U}]. \tag{5.26}$$

In combination, they yield the result that $\mathbb{E}_V[\mathrm{gain}_{A_U}] < \mathbb{E}_V[\mathrm{gain}_{A_V}]$ and hence that A_U is not optimal on $\mathcal{I}_{\mathrm{all}} \setminus V$. We start by proving (5.25).

Lemma 5.25. *Let U and V be two different balanced sets and let A_U and A_V be two corresponding optimal deterministic online algorithms for the input sets $\mathcal{I}_{all} \setminus U$ and $\mathcal{I}_{all} \setminus V$, respectively. Then we have*

$$\mathbb{E}_U[\mathrm{gain}_{A_U}] = \mathbb{E}_V[\mathrm{gain}_{A_V}] = \frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}}.$$

Proof. Let us consider the algorithm $\mathcal{A}$ with advice we described before. Since $\mathcal{A}$ is optimal on any set of instances, as we have seen in Theorem 5.23, $\mathcal{A}$ must be optimal on $\mathcal{I}_{\mathrm{all}} \setminus U$ and $\mathcal{I}_{\mathrm{all}} \setminus V$ in particular. Furthermore, we know that $\mathcal{A}$ gets the set $\tilde{\mathcal{I}}$ of excluded strings as advice and then behaves just like a deterministic algorithm that is optimal on $\mathcal{I}_{\mathrm{all}} \setminus \tilde{\mathcal{I}}$. Hence, without loss of generality, $\mathcal{A} = A_U$ if $\tilde{\mathcal{I}} = U$, and $\mathcal{A} = A_V$ if $\tilde{\mathcal{I}} = V$.

Due to Lemma 5.16, $\mathcal{A}$ guesses $(n \cdot 2^{n-1} - \hat{\omega}(\tilde{k}))/(2^n - \tilde{k})$ bits correctly in expectation on any worst-case instance set. Since U and V are balanced, $\mathcal{I}_{\mathrm{all}} \setminus U$ and $\mathcal{I}_{\mathrm{all}} \setminus V$ must be balanced, too. According to our considerations following Lemma 5.16, if the instance set for $\mathcal{A}$ is balanced, this leads to the worst-case expected number of correctly guessed bits, hence to $(n \cdot 2^{n-1} - \hat{\omega}(\tilde{k}))/(2^n - \tilde{k})$ correctly guessed bits in expectation. □

To prove (5.26), we have to make some more effort. We want to compare the expected number of correctly guessed bits of A_U on the two different instance sets $\mathcal{I}_{\mathrm{all}} \setminus U$ and $\mathcal{I}_{\mathrm{all}} \setminus V$. To make our argumentation easier, let us introduce a different view on balanced sets.

We can identify the set $T_\varepsilon^{(n)} = T_\varepsilon$ of all 2^n different strings of length n with a complete binary tree of depth n, where each leaf represents one string, ordered from left to right in increasing lexicographic order. Naturally, each vertex then represents a prefix of a string. The vertex on the path from the root to a string s in the tree T_ε that has distance i to the root can be identified with the prefix $[s]^i$ of length i of the string s. Hence, in the following, we can use the terms "string s in the set T_ε" and "leaf s in the tree T_ε" interchangeably, and we can say "prefix $[s]^i$"

just as we say "vertex $[s]^i$". In particular, the root of the tree can from now on be referred to as ε, the empty string. Also, this indicates that the set of vertices of the binary tree $T_\varepsilon^{(n)}$ is $W_\varepsilon^{(n)} = W_\varepsilon = \{[s]^i \mid s \in T_\varepsilon, 0 \leq i \leq n\}$. A balanced set now is a selection of leaves such that, for each inner vertex v, the number of selected leaves in the left subtree of v differs by at most one from the number of selected leaves in the right subtree of v. An example of such a binary tree with a selection of leaves that form a balanced set is given in Figure 5.3.

We can identify the computation of any algorithm $\mathcal{A}$ with a route through the binary tree T_ε from the root to the leaf r, i. e., the vertex that corresponds to the to-be-guessed string. For each string $s \in T_\varepsilon$, each vertex $[s]^{i-1}$ with $1 \leq i \leq n$ on the path from the root to the leaf s has two children. One of these, the one that also lies on the path from ε to s, is $[s]^i$, and the other one is called $([s]^i)^*$. Recall that we call these two children of $[s]^{i-1}$ *siblings*. If, in round i with $1 \leq i \leq n$, the algorithm's guessed bit is $g_i = 0$, this corresponds to $\mathcal{A}$ taking the left branch at the current vertex $[r]^{i-1}$; if $g_i = 1$, this corresponds to $\mathcal{A}$ taking the right branch. If $\mathcal{A}$ guesses correctly in round i, it walks from $[r]^{i-1}$ to its child $[r]^i$. If it makes a wrong guess, it walks to the wrong child $([r]^i)^*$ instead. In round $i+1$, the algorithm learns about its mistake (being sent the request r_i) and reacts to this by "jumping" from $([r]^i)^*$ to the correct child $[r]^i$. Regardless of the path that $\mathcal{A}$ takes through the tree, the algorithm always ends up at vertex $[r]^n = r$ at the end of round n. The number of "jumps" of $\mathcal{A}$ from one vertex to another vertex on the same level is the number of incorrectly guessed bits of $\mathcal{A}$. Figure 5.3 also gives an example of a route through T_ε that represents the run of an algorithm on the input r.

Now let us consider algorithm A_U. For each inner vertex $[r]^i$ of the tree T_ε, let us mark one of its outgoing edges as follows. We know that A_U passes each inner vertex $[r]^i$ at some point during its computation. Since A_U is deterministic, it must choose the next vertex on the path based only on the strings contained in U and the feedback that A_U got up to now. We know that, at the point when A_U reaches $[r]^i$, the feedback that it got until then consists of $[r]^i$. Hence, A_U's decision is unambiguously determined by the strings in U. Thus, there is one unambiguous outgoing edge from $[r]^i$ that leads to the child of $[r]^i$ that A_U chooses as the successor of $[r]^i$. Let this edge be the marked one, for every inner vertex $[r]^i$. If the number of strings from $\mathcal{I}_{\text{all}} \setminus U$ in the two subtrees beneath $[r]^i$ differs, the marked edge is the one pointing to the subtree with less strings from $\mathcal{I}_{\text{all}} \setminus U$ since A_U is optimal on $\mathcal{I}_{\text{all}} \setminus U$. If the number of strings from $\mathcal{I}_{\text{all}} \setminus U$ in the two subtrees is the same but the number of strings from $\mathcal{I}_{\text{all}} \setminus V$ differs, we can assume that the marked edge is the one pointing to the subtree with less strings from $\mathcal{I}_{\text{all}} \setminus V$. If this is also the same for both algorithms, the marked edge can be an arbitrary but fixed one. For an example, see Figure 5.4.

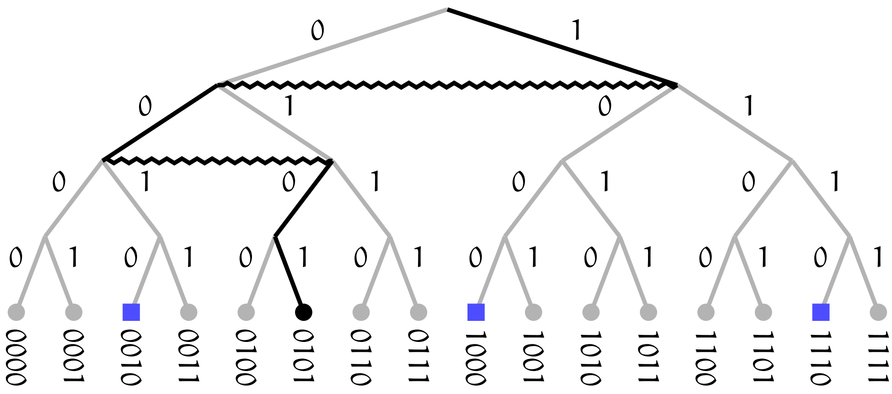


Figure 5.3. Representation of the set of possible input strings as a binary tree. Strings from U are marked as blue rectangles. The set $U = \{0010, 1000, 1110\}$ is obviously balanced. The black leaf is the string $r = 0101$. The black lines illustrate a route through the tree from the root to r, corresponding to a run of the algorithm. The algorithm's guess is 1001. Hence, its first two guesses are wrong and it has to "jump" from the chosen child to the other one, denoted by zigzag lines. The number of zigzag lines of the route is equivalent to the number of incorrectly guessed bits of the algorithm.

For each deterministic algorithm A, let us call the marked outgoing edge of a vertex $[r]^i$ the *favored edge* of A at $[r]^i$ and the vertex that it leads to the *favored child* of A at $[r]^i$. Let $\text{gain}_{A_U}(r)$ denote the number of bits that A_U guesses correctly on the input string r. We make the following observation concerning $\text{gain}_{A_U}(r)$; for an example, see Figure 5.4.

Observation 5.26. *The number* $\text{gain}_{A_U}(r)$ *of correctly guessed bits of* $\mathcal{A}$ *on the input* r *equals the number of favored edges on the path from the root to the leaf* r *in the binary tree.*

Proof. This follows directly from the definition of the favored edges. □

As one part of the proof of (5.26), we need to show that $\sum_{s\in V} \text{gain}_{A_U}(s) - \sum_{s\in U} \text{gain}_{A_U}(s) > 0$. The way we chose the marked edges if the number of strings from $\mathcal{I}_{\text{all}} \setminus U$ is the same in the two subtrees of $[s]^i$ but the number of strings from $\mathcal{I}_{\text{all}} \setminus V$ differs minimizes this difference; hence, we are considering the worst case. If the algorithm A_U should instead prefer the other outgoing edge of $[s]^i$, this difference can only increase. Hence, we will from now on assume that A_U favors the marked edges as described above.

Before we prove (5.26), we need two more rather technical results and the following well-known property of the floor and ceiling functions that has been proven for example by Graham et al. [GKP89].

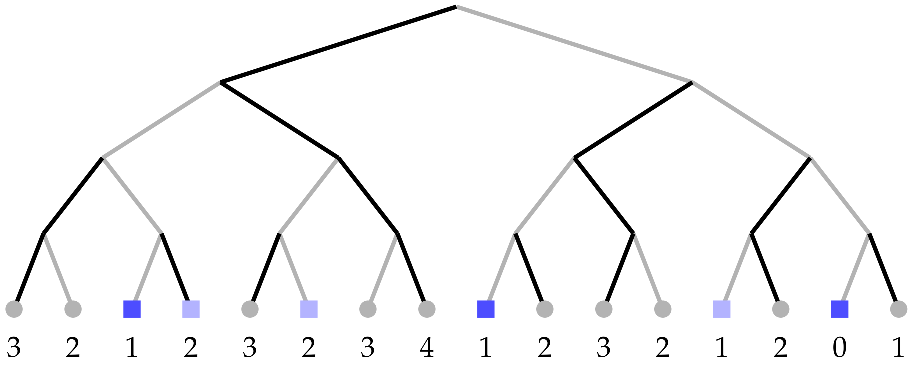


Figure 5.4. Representation of the set of possible input strings as a binary tree. Light blue rectangular leaves denote strings from V, the darker blue rectangles denote strings from U. The favored edges of A_U are marked in black. The number beneath a leaf s indicates the number $\text{gain}_{A_U}(s)$ of correctly guessed bits of A_U if $r = s$. This coincides with the number of favored edges on the path from the root to s.

Fact 5.27. *For $x \in \mathbb{R}$ and $y, z \in \mathbb{Z}$ with $z \geq 1$,*

$$\left\lfloor \frac{\lfloor x \rfloor + y}{z} \right\rfloor = \left\lfloor \frac{x+y}{z} \right\rfloor \quad \text{and} \quad \left\lceil \frac{\lceil x \rceil + y}{z} \right\rceil = \left\lceil \frac{x+y}{z} \right\rceil. \qquad \square$$

For each vertex $[s]^i \in W_\varepsilon^{(n)}$, let the subtree rooted at $[s]^i$ be $T_{[s]^i}^{(n)}$ with the vertex set

$$W_{[s]^i}^{(n)} = \{[t]^j \mid t \in T_{[s]^i}^{(n)}, i \leq j \leq n\} = \{[t]^j \mid t \in T_\varepsilon^{(n)}, [t]^i = [s]^i, i \leq j \leq n\}.$$

Again, if n is clear from the context, we also write $T_{[s]^i}$ and $W_{[s]^i}$ instead, respectively.

Lemma 5.28. *Let $[s]^i \in W_\varepsilon$ be a vertex on level i in T_ε, and let the height of the subtree $T_{[s]^i}$ be $h := n - i$. Let the set U be balanced. Then, either $\lfloor \tilde{k} \cdot 2^{-i} \rfloor$ or $\lceil \tilde{k} \cdot 2^{-i} \rceil$ leaves from $T_{[s]^i}$ belong to U.*

Proof. The claim obviously holds for the root of the binary tree, because it is on level 0 and $\lfloor \tilde{k} \cdot 2^0 \rfloor = \lceil \tilde{k} \cdot 2^0 \rceil = \tilde{k}$. Now let us assume that the claim holds for a vertex $[s]^i$ on level i. Thus, $T_{[s]^i}$ contains either $\lfloor \tilde{k} \cdot 2^{-i} \rfloor$ or $\lceil \tilde{k} \cdot 2^{-i} \rceil$ strings from U. Let the children of $[s]^i$ be f and f^*. They both are on level $i+1$. If $T_{[s]^i}$ contains ℓ strings from U, due to the balance properties of U, one of the subtrees T_f and T_{f^*} must contain $\lfloor \ell/2 \rfloor$ strings from U and the other one $\lceil \ell/2 \rceil$. If $\ell = \lfloor \tilde{k} \cdot 2^{-i} \rfloor$, we have, due to Fact 5.27,

$$\left\lfloor \frac{\ell}{2} \right\rfloor = \left\lfloor \frac{\lfloor \tilde{k} \cdot 2^{-i} \rfloor}{2} \right\rfloor = \left\lfloor \frac{\tilde{k} \cdot 2^{-i}}{2} \right\rfloor = \left\lfloor \frac{\tilde{k}}{2^{i+1}} \right\rfloor,$$

and

$$\left\lceil \frac{\ell}{2} \right\rceil = \left\lceil \frac{\lfloor \tilde{k} \cdot 2^{-i} \rfloor}{2} \right\rceil = \begin{cases} \left\lceil \frac{\tilde{k} \cdot 2^{-i}}{2} \right\rceil = \left\lceil \frac{\tilde{k}}{2^{i+1}} \right\rceil & \text{if } \tilde{k} \cdot 2^{-i} \in \mathbb{N}, \\ \left\lceil \frac{\lceil \tilde{k} \cdot 2^{-i} \rceil - 1}{2} \right\rceil = \left\lceil \frac{\tilde{k} \cdot 2^{-i} - 1}{2} \right\rceil & \text{otherwise.} \end{cases}$$

Thus, in the case that $\ell = \lfloor \tilde{k} \cdot 2^{-i} \rfloor$, if $\tilde{k} \cdot 2^{-i} \in \mathbb{N}$, we are done. If $\tilde{k} \cdot 2^{-i} \notin \mathbb{N}$, it suffices to show that

$$\left\lfloor \tilde{k} \cdot 2^{-(i+1)} \right\rfloor \leq \left\lceil \frac{\tilde{k} \cdot 2^{-i} - 1}{2} \right\rceil \leq \left\lceil \tilde{k} \cdot 2^{-(i+1)} \right\rceil$$

to prove the induction step. The right inequality obviously holds. Furthermore, since $\tilde{k} \cdot 2^{-i} \notin \mathbb{N}$, also $\tilde{k} \cdot 2^{-(i+1)} \notin \mathbb{N}$, and thus

$$\left\lfloor \tilde{k} \cdot 2^{-(i+1)} \right\rfloor = \left\lceil \tilde{k} \cdot 2^{-(i+1)} \right\rceil - 1 = \left\lceil \frac{\tilde{k} \cdot 2^{-i}}{2} \right\rceil - 1 \leq \left\lceil \frac{\tilde{k} \cdot 2^{-i} - 1}{2} \right\rceil.$$

We make an analogous calculation for the case that $\ell = \lceil \tilde{k} \cdot 2^{-i} \rceil$. Again, due to Fact 5.27, we have

$$\left\lceil \frac{\ell}{2} \right\rceil = \left\lceil \frac{\lceil \tilde{k} \cdot 2^{-i} \rceil}{2} \right\rceil = \left\lceil \frac{\tilde{k} \cdot 2^{-i}}{2} \right\rceil = \left\lceil \frac{\tilde{k}}{2^{i+1}} \right\rceil,$$

and

$$\left\lfloor \frac{\ell}{2} \right\rfloor = \left\lfloor \frac{\lceil \tilde{k} \cdot 2^{-i} \rceil}{2} \right\rfloor = \begin{cases} \left\lfloor \frac{\tilde{k} \cdot 2^{-i}}{2} \right\rfloor = \left\lfloor \frac{\tilde{k}}{2^{i+1}} \right\rfloor & \text{if } \tilde{k} \cdot 2^{-i} \in \mathbb{N}, \\ \left\lfloor \frac{\lfloor \tilde{k} \cdot 2^{-i} \rfloor + 1}{2} \right\rfloor = \left\lfloor \frac{\tilde{k} \cdot 2^{-i} + 1}{2} \right\rfloor & \text{otherwise.} \end{cases}$$

If $\tilde{k} \cdot 2^{-i} \notin \mathbb{N}$, it suffices to show that

$$\left\lfloor \tilde{k} \cdot 2^{-(i+1)} \right\rfloor \leq \left\lfloor \frac{\tilde{k} \cdot 2^{-i} + 1}{2} \right\rfloor \leq \left\lceil \tilde{k} \cdot 2^{-(i+1)} \right\rceil$$

to prove the induction step, and the left inequality obviously holds. Moreover, with the same argument as above,

$$\left\lfloor \frac{\tilde{k} \cdot 2^{-i} + 1}{2} \right\rfloor \leq \left\lfloor \frac{\tilde{k} \cdot 2^{-i}}{2} \right\rfloor + 1 = \left\lfloor \tilde{k} \cdot 2^{-(i+1)} \right\rfloor + 1 = \left\lceil \tilde{k} \cdot 2^{-(i+1)} \right\rceil,$$

completing the induction step in the second case. Hence, the claim holds for all vertices on all levels i with $0 \leq i \leq n$. $\square$

Lemma 5.29. *For two different balanced sets* U *and* V *of size* $\tilde{k}$ *each,*

$$\sum_{s \in V} \mathrm{gain}_{A_u}(s) - \sum_{s \in U} \mathrm{gain}_{A_u}(s) = |U \setminus V|.$$

Proof. For a vertex $v \in W_\varepsilon$, let us define e_v to be the number of marked edges on the path from the root to v. Furthermore, for a set $S \subseteq T_\varepsilon$ of strings of length n and a vertex $v \in W_\varepsilon$, let $\vartheta(S, v)$ denote the number of strings from S in T_v. Then we assign a score to each vertex v by

$$\mathrm{score}(v) \coloneqq \sum_{s \in V \cap W_v} \mathrm{gain}_{A_u}(s) - \sum_{s \in U \cap W_v} \mathrm{gain}_{A_u}(s).$$

Now we prove the following for every vertex v.

$$\mathrm{score}(v) = \begin{cases} \vartheta(V \setminus U, v) + \hat{\omega}(\vartheta(V, v)) - \hat{\omega}(\vartheta(U, v)) \\ \qquad + (\vartheta(V, v) - \vartheta(U, v))\, e_v & \text{if } \vartheta(V, v) \leq \vartheta(U, v) \\ \vartheta(V \setminus U, v) + \hat{\omega}(\vartheta(V, v)) - \hat{\omega}(\vartheta(U, v)) \\ \qquad + (\vartheta(V, v) - \vartheta(U, v))\, e_v - 1 & \text{if } \vartheta(V, v) > \vartheta(U, v) \end{cases} \tag{5.27}$$

Note that, if $\vartheta(V, v) = \vartheta(U, v)$, we have

$$\begin{aligned} \mathrm{score}(v) &= \vartheta(V \setminus U, v) + \hat{\omega}(\vartheta(V, v)) - \hat{\omega}(\vartheta(U, v)) + e_v\big(\vartheta(V, v) - \vartheta(U, v)\big) \\ &= \vartheta(V \setminus U, v). \end{aligned} \tag{5.28}$$

Let us prove (5.27) by induction on the level of v in the tree, i. e., the number of edges on the shortest path from the root to v. We start with a vertex v on level n (the lowermost level in the tree). We distinguish several cases. The leaf v might neither be an element of U nor V, or it might be an element of U but not of V, or an element of V but not of U, or v might be an element of both. The subtree T_v rooted at v consists of exactly one vertex, namely v, and contains exactly 0 or 1 strings from U and V, respectively, as specified by the four cases above.

Case 1. $\vartheta(V, v) = 0$, $\vartheta(U, v) = 0$.

According to its definition, the score of the leaf v is

$$\mathrm{score}(v) = \sum_{s \in V \cap W_v} \mathrm{gain}_{A_u}(s) - \sum_{s \in U \cap W_v} \mathrm{gain}_{A_u}(s) = 0 - 0 = 0.$$

With e_v being the number of marked edges from the root to v as defined before, this coincides with

$$\begin{aligned}\operatorname{score}(v) &= 0\\ &= 0 + \hat{\omega}(0) - \hat{\omega}(0) + 0 \cdot e_v\\ &= \vartheta(V \setminus U, v) + \hat{\omega}(\vartheta(V, v)) - \hat{\omega}(\vartheta(U, v)) + (\vartheta(V, v) - \vartheta(U, v))e_v.\end{aligned}$$

Case 2. $\vartheta(V, v) = 1, \vartheta(U, v) = 0.$

Again, according to its definition, the score of v is

$$\operatorname{score}(v) = \sum_{s \in V \cap W_v} \operatorname{gain}_{A_u}(s) - \sum_{s \in U \cap W_v} \operatorname{gain}_{A_u}(s) = \operatorname{gain}_{A_u}(v) - 0 = e_v,$$

where the last equality holds due to Observation 5.26. This coincides with

$$\begin{aligned}\operatorname{score}(v) &= e_v\\ &= 1 + \hat{\omega}(1) - \hat{\omega}(0) + 1 \cdot e_v - 1\\ &= \vartheta(V \setminus U, v) + \hat{\omega}(\vartheta(V, v)) - \hat{\omega}(\vartheta(U, v)) + (\vartheta(V, v) - \vartheta(U, v))\, e_v - 1.\end{aligned}$$

Case 3. $\vartheta(V, v) = 0, \vartheta(U, v) = 1.$

Using Observation 5.26 again, the score of v can be calculated as

$$\operatorname{score}(v) = \sum_{s \in V \cap W_v} \operatorname{gain}_{A_u}(s) - \sum_{s \in U \cap W_v} \operatorname{gain}_{A_u}(s) = 0 - \operatorname{gain}_{A_u}(v) = -e_v.$$

This coincides with

$$\begin{aligned}\operatorname{score}(v) &= -e_v\\ &= 0 + \hat{\omega}(0) - \hat{\omega}(1) + (-1)\, e_v\\ &= \vartheta(V \setminus U, v) + \hat{\omega}(\vartheta(V, v)) - \hat{\omega}(\vartheta(U, v)) + (\vartheta(V, v) - \vartheta(U, v))\, e_v.\end{aligned}$$

Case 4. $\vartheta(V, v) = 1, \vartheta(U, v) = 1.$

In this case, $v \in U$ and $v \in V$. Then, $\vartheta(V \setminus U, v) = 0$. The score of v is therefore

$$\operatorname{score}(v) = \sum_{s \in V \cap W_v} \operatorname{gain}_{A_u}(s) - \sum_{s \in U \cap W_v} \operatorname{gain}_{A_u}(s) = e_v - e_v = 0.$$

This coincides with

$$\begin{aligned}\operatorname{score}(v) &= 0\\ &= 0 + \hat{\omega}(1) - \hat{\omega}(1) + 0 \cdot e_v\\ &= \vartheta(V \setminus U, v) + \hat{\omega}(\vartheta(V, v)) - \hat{\omega}(\vartheta(U, v)) + (\vartheta(V, v) - \vartheta(U, v))\, e_v.\end{aligned}$$

Hence, the statement is true for all vertices on level n, and thus the base case is covered. Let us now turn to the induction step. To this end, assume that the induction statement (5.27) holds for two vertices f and f^* on level i, which are children of a vertex v on level $i-1$, where f is the favored child of v. We will show that the statement then also holds for v. Obviously, the value $\text{score}(v)$ can be calculated as

$$\text{score}(v) = \text{score}(f) + \text{score}(f^*).$$

We make a case distinction depending on the number of strings from U and V contained in T_f and T_{f^*}, respectively. The first case deals with the scenario $\vartheta(V,f) \le \vartheta(V,f^*)$, and the second one addresses the case $\vartheta(V,f) > \vartheta(V,f^*)$. Before we consider the first case, let us do some simple transformations under the assumption that $\vartheta(V,f) \le \vartheta(V,f^*)$.

Due to the balance properties of the subtree T_v and since f is the favored child at the vertex v, we know that $\vartheta(U,f) \le \vartheta(U,f^*) \le \vartheta(U,f)+1$. Furthermore, we can infer that $\vartheta(U,f) = \lfloor \vartheta(U,v)/2 \rfloor$ and $\vartheta(V,f) = \lceil \vartheta(U,v)/2 \rceil$. Applying Corollary 5.12 yields

$$\begin{aligned}\hat{\omega}(\vartheta(U,v)) &= \hat{\omega}\left(\left\lceil \tfrac{\vartheta(U,v)}{2} \right\rceil\right) + \hat{\omega}\left(\left\lfloor \tfrac{\vartheta(U,v)}{2} \right\rfloor\right) + \left\lfloor \tfrac{\vartheta(U,v)}{2} \right\rfloor \\ &= \hat{\omega}(\vartheta(U,f^*)) + \hat{\omega}(\vartheta(U,f)) + \vartheta(U,f), \end{aligned} \tag{5.29}$$

and since we are assuming $\vartheta(V,f) \le \vartheta(V,f^*)$, we get an analogous result for V, namely

$$\hat{\omega}(\vartheta(V,v)) = \hat{\omega}(\vartheta(V,f)) + \hat{\omega}(\vartheta(V,f^*)) + \vartheta(V,f). \tag{5.30}$$

We define $h_1 := \hat{\omega}(\vartheta(V,f)) - \hat{\omega}(\vartheta(U,f)) + \hat{\omega}(\vartheta(V,f^*)) - \hat{\omega}(\vartheta(U,f^*))$ and use (5.29) and (5.30) to transform it to

$$\begin{aligned} h_1 &= \hat{\omega}(\vartheta(V,f)) - \hat{\omega}(\vartheta(U,f)) + \hat{\omega}(\vartheta(V,f^*)) - \hat{\omega}(\vartheta(U,f^*)) \\ &= (\hat{\omega}(\vartheta(V,f)) + \hat{\omega}(\vartheta(V,f^*))) - (\hat{\omega}(\vartheta(U,f)) + \hat{\omega}(\vartheta(U,f^*))) \\ &= (\hat{\omega}(\vartheta(V,v)) - \vartheta(V,f)) - (\hat{\omega}(\vartheta(U,v)) - \vartheta(U,f)) \\ &= \hat{\omega}(\vartheta(V,v)) - \hat{\omega}(\vartheta(U,v)) - \vartheta(V,f) + \vartheta(U,f). \end{aligned} \tag{5.31}$$

As the number of marked edges from the root to f^* is exactly e_v and the number of marked edges from the root to f is e_v+1, for $h_2 := (\vartheta(V,f) - \vartheta(U,f))\, e_f + (\vartheta(V,f^*) - \vartheta(U,f^*))\, e_{f^*}$, it holds that

$$\begin{aligned} h_2 &= (\vartheta(V,f) - \vartheta(U,f))\, e_f + (\vartheta(V,f^*) - \vartheta(U,f^*))\, e_{f^*} \\ &= (\vartheta(V,f) - \vartheta(U,f))(e_v+1) + (\vartheta(V,f^*) - \vartheta(U,f^*))\, e_v \\ &= (\vartheta(V,f) - \vartheta(U,f) + \vartheta(V,f^*) - \vartheta(U,f^*))\, e_v + (\vartheta(V,f) - \vartheta(U,f)) \\ &= (\vartheta(V,v) - \vartheta(U,v))\, e_v + \vartheta(V,f) - \vartheta(U,f). \end{aligned} \tag{5.32}$$

Using (5.31) and (5.32) yields

$$\begin{aligned} h_1 + h_2 &= \hat{\omega}(\vartheta(V, v)) - \hat{\omega}(\vartheta(U, v)) - \vartheta(V, f) + \vartheta(U, f) \\ &\quad + (\vartheta(V, v) - \vartheta(U, v))\, e_v + \vartheta(V, f) - \vartheta(U, f) \\ &= \hat{\omega}(\vartheta(V, v)) - \hat{\omega}(\vartheta(U, v)) + (\vartheta(V, v) - \vartheta(U, v))\, e_v. \end{aligned} \tag{5.33}$$

Now let us make the already mentioned case distinction to calculate $\mathrm{score}(v)$.

Case 1. $\vartheta(V, f) \leq \vartheta(V, f^*)$.

We divide this case into three subcases. First, we consider the case that in both subtrees there are at least as many strings from U as from V; then we deal with the cases that exactly one of the subtrees, either T_f or T_{f^*}, contains more strings from V than from U. Note that we do not have to consider the case that both subtrees T_f and T_{f^*} contain more strings from V than from U since then the number of strings from V and the number of strings from U in the subtree T_v would differ by at least 2, which is not possible due to Lemma 5.28.

Case 1.1. $\vartheta(V, f) \leq \vartheta(U, f) \wedge \vartheta(V, f^*) \leq \vartheta(U, f^*)$.

Using the induction hypothesis (5.27), the score of v can be calculated from the scores of f and f^* as

$$\begin{aligned} \mathrm{score}(v) &= \mathrm{score}(f^*) + \mathrm{score}(f) \\ &= \vartheta(V \setminus U, f^*) + \hat{\omega}(\vartheta(V, f^*)) - \hat{\omega}(\vartheta(U, f^*)) + e_{f^*}\,(\vartheta(V, f^*) - \vartheta(U, f^*)) \\ &\quad + \vartheta(V \setminus U, f) + \hat{\omega}(\vartheta(V, f)) - \hat{\omega}(\vartheta(U, f)) + e_f\,(\vartheta(V, f) - \vartheta(U, f)). \end{aligned}$$

Furthermore, using the definitions of h_1 and h_2 and (5.33) yields

$$\begin{aligned} \mathrm{score}(v) &= \vartheta(V \setminus U, v) + h_1 + h_2 \\ &= \vartheta(V \setminus U, v) + \hat{\omega}(\vartheta(V, v)) - \hat{\omega}(\vartheta(U, v)) + e_v\,(\vartheta(V, v) - \vartheta(U, v)). \end{aligned}$$

This already proves the induction claim for this case.

Case 1.2. $\vartheta(V, f) > \vartheta(U, f) \wedge \vartheta(V, f^*) \leq \vartheta(U, f^*)$.

If exactly one of the subtrees has more strings from V than from U, the calculation is almost the same as in the first case; we only have to subtract 1, as can be seen from the induction hypothesis. More concretely, we obtain

$$\begin{aligned} \mathrm{score}(v) &= \mathrm{score}(f^*) + \mathrm{score}(f) \\ &= \vartheta(V \setminus U, f^*) + \hat{\omega}(\vartheta(V, f^*)) - \hat{\omega}(\vartheta(U, f^*)) + e_{f^*}\,(\vartheta(V, f^*) - \vartheta(U, f^*)) - 1 \\ &\quad + \vartheta(V \setminus U, f) + \hat{\omega}(\vartheta(V, f)) - \hat{\omega}(\vartheta(U, f)) + e_f\,(\vartheta(V, f) - \vartheta(U, f)) \end{aligned}$$

using the induction hypothesis (5.27). Again, with the definitions of h_1 and h_2 and (5.33) we get

$$\begin{aligned}\mathrm{score}(v) &= \vartheta(V \setminus U, v) + h_1 + h_2 - 1 \\ &= \vartheta(V \setminus U, v) + \hat{\omega}(\vartheta(V, v)) - \hat{\omega}(\vartheta(U, v)) + e_v\,(\vartheta(V, v) - \vartheta(U, v)) - 1.\end{aligned}$$

This proves that the claim also holds in this case.

Case 1.3. $\vartheta(V, f^*) > \vartheta(U, f^*) \wedge \vartheta(V, f) \le \vartheta(U, f)$.

This case is completely analogous to case 1.2.

Case 2. $\vartheta(V, f) > \vartheta(V, f^*)$.

In this case, we cannot use equations (5.30) to (5.33) since the assumption $\vartheta(V, f) \le \vartheta(V, f^*)$ that we needed for these equations does not hold.
The tree T_f rooted at the favored child f at v may only contain more strings from V than T_{f^*} if T_{f^*} contains more strings from U than T_f. Otherwise, f^* would be the favored child of v. Hence, $\vartheta(U, f^*) > \vartheta(U, f)$.
As f and f^* are vertices on level i, the subtrees T_f and T_{f^*} have height $n - i$. Hence, from Lemma 5.28, we know that both T_f and T_{f^*} must contain either $\lfloor \tilde{k} \cdot 2^{-i} \rfloor$ or $\lceil \tilde{k} \cdot 2^{-i} \rceil$ strings from U and, equally, either $\lfloor \tilde{k} \cdot 2^{-i} \rfloor$ or $\lceil \tilde{k} \cdot 2^{-i} \rceil$ strings from V. Since $\vartheta(V, f) > \vartheta(V, f^*)$ and $\vartheta(U, f^*) > \vartheta(U, f)$, we can conclude $\vartheta(V, f) = \vartheta(U, f^*) = \lceil \tilde{k} \cdot 2^{-i} \rceil$ and $\vartheta(V, f^*) = \vartheta(U, f) = \lfloor \tilde{k} \cdot 2^{-i} \rfloor$ with $\lceil \tilde{k} \cdot 2^{-i} \rceil = \lfloor \tilde{k} \cdot 2^{-i} \rfloor + 1$. Moreover, as $\vartheta(V, v) = \vartheta(V, f) + \vartheta(V, f^*) = \vartheta(U, f) + \vartheta(U, f^*) = \vartheta(U, v)$, due to (5.28), the score of v should be $\mathrm{score}(v) = \vartheta(V \setminus U, v)$. Hence, we calculate the score of v, taking into account that $\vartheta(V, f) > \vartheta(U, f)$ and $\vartheta(V, f^*) \le \vartheta(U, f^*)$, and we obtain

$$\begin{aligned}\mathrm{score}(v) &= \mathrm{score}(f) + \mathrm{score}(f^*) \\ &= \vartheta(V \setminus U, f) + \hat{\omega}(\vartheta(V, f)) - \hat{\omega}(\vartheta(U, f)) + e_f\,(\vartheta(V, f) - \vartheta(U, f)) - 1 \\ &\quad + \vartheta(V \setminus U, f^*) + \hat{\omega}(\vartheta(V, f^*)) - \hat{\omega}(\vartheta(U, f^*)) + e_{f^*}\,(\vartheta(V, f^*) - \vartheta(U, f^*))\end{aligned}$$

using the induction hypothesis (5.27). Further calculations yield

$$\begin{aligned}\mathrm{score}(v) &= \vartheta(V \setminus U, v) + e_f - 1 - e_{f^*} \\ &= \vartheta(V \setminus U, v) + e_v - e_v \\ &= \vartheta(V \setminus U, v),\end{aligned}$$

proving the claim also in this case.

As a result, we have proven the claim for all vertices, and we can apply it to the root ε of the binary tree. The complete binary tree T_ε contains as many strings from U as from V, thus we obtain

$$\mathrm{score}(\varepsilon) = \vartheta(V \setminus U, \varepsilon) = |V \setminus U| = |U \setminus V|. \tag{5.34}$$

Furthermore, according to its definition, the score of ε is

$$\begin{aligned}\text{score}(\varepsilon) &= \sum_{s \in V \cap W_\varepsilon} \text{gain}_{A_U}(s) - \sum_{s \in U \cap W_\varepsilon} \text{gain}_{A_U}(s) \\ &= \sum_{s \in V} \text{gain}_{A_U}(s) - \sum_{s \in U} \text{gain}_{A_U}(s),\end{aligned}$$

and combining this with (5.34) yields the proposition of the lemma, namely

$$|U \setminus V| = \sum_{s \in V} \text{gain}_{A_U}(s) - \sum_{s \in U} \text{gain}_{A_U}(s). \qquad \square$$

Corollary 5.30. *For two different balanced sets* U *and* V *of size* $\tilde{k} \geq 1$ *each, we have*

$$\sum_{s \in V} \text{gain}_{A_U}(s) - \sum_{s \in U} \text{gain}_{A_U}(s) > 0.$$

Proof. Since U and V are two different balanced sets that contain at least one string each, $U \setminus V$ must also contain at least one string, and thus Lemma 5.29 directly implies

$$\sum_{s \in V} \text{gain}_{A_U}(s) - \sum_{s \in U} \text{gain}_{A_U}(s) = |U \setminus V| > 0. \qquad \square$$

Lemma 5.31. *Let* U *and* V *be two different balanced sets of size* $\tilde{k} \geq 1$ *each, and let* A_U *and* A_V *be two corresponding optimal deterministic online algorithms for the input sets* $\mathcal{I}_{all} \setminus U$ *and* $\mathcal{I}_{all} \setminus V$*, respectively. Then,*

$$\mathbb{E}_V[\text{gain}_{A_U}] < \mathbb{E}_U[\text{gain}_{A_U}].$$

Proof. Let us first analyze $\mathbb{E}_V[\text{gain}_{A_U}] = \sum_{s \in \mathcal{I}_{\text{all}} \setminus V} \Pr(r = s \mid \tilde{\mathcal{I}} = V) \cdot \text{gain}_{A_U}(s)$. Since we know that all strings from $\mathcal{I}_{\text{all}} \setminus V$ are equally likely to be the to-be-guessed string r when $\tilde{\mathcal{I}} = V$, and since $|\mathcal{I}_{\text{all}} \setminus V| = 2^n - \tilde{k}$, we have

$$\begin{aligned}\mathbb{E}_V[\text{gain}_{A_U}] &= \sum_{s \in \mathcal{I}_{\text{all}} \setminus V} \Pr\big(r = s \mid \tilde{\mathcal{I}} = V\big) \cdot \text{gain}_{A_U}(s) \\ &= \sum_{s \in \mathcal{I}_{\text{all}} \setminus V} \frac{1}{2^n - \tilde{k}} \cdot \text{gain}_{A_U}(s) \\ &= \frac{1}{2^n - \tilde{k}} \cdot \left(\sum_{s \in \mathcal{I}_{\text{all}} \setminus (U \cup V)} \text{gain}_{A_U}(s) + \sum_{s \in U \setminus V} \text{gain}_{A_U}(s) \right).\end{aligned}$$

As U and V have the same size, also $|\mathcal{I}_{\text{all}} \setminus U| = 2^n - \tilde{k}$; hence, we can do an analogous calculation for $\mathbb{E}_U[\text{gain}_{A_U}]$, which yields

$$\begin{aligned}\mathbb{E}_U[\text{gain}_{A_U}] &= \sum_{s \in \mathcal{I}_{\text{all}} \setminus U} \Pr\big(r = s \mid \tilde{\mathcal{I}} = U\big) \cdot \text{gain}_{A_U}(s) \\ &= \frac{1}{2^n - \tilde{k}} \cdot \left(\sum_{s \in \mathcal{I}_{\text{all}} \setminus (U \cup V)} \text{gain}_{A_U}(s) + \sum_{s \in V \setminus U} \text{gain}_{A_U}(s) \right).\end{aligned}$$

We conclude that

$$\begin{aligned}\mathbb{E}_U[\text{gain}_{A_U}] - \mathbb{E}_V[\text{gain}_{A_U}] &= \frac{1}{2^n - \tilde{k}} \cdot \left(\sum_{s \in V \setminus U} \text{gain}_{A_U}(s) - \sum_{s \in U \setminus V} \text{gain}_{A_U}(s) \right) \\ &= \frac{1}{2^n - \tilde{k}} \cdot \left(\sum_{s \in V} \text{gain}_{A_U}(s) - \sum_{s \in U} \text{gain}_{A_U}(s) \right) \\ &> 0,\end{aligned}$$

where the last inequality follows from Corollary 5.30. □

We have now seen that, for an arbitrary balanced set U of size $\tilde{k}$ containing binary strings of length n, the corresponding deterministic algorithm A_U that is optimal on $\mathcal{I}_{\text{all}} \setminus U$ cannot be optimal on $\mathcal{I}_{\text{all}} \setminus V$, for any balanced set $V \neq U$. Let us consider an algorithm $\mathcal{A}$ with advice that gets the same advice string for an arbitrary pair of two different input sets $\mathcal{I}_{\text{all}} \setminus U$ and $\mathcal{I}_{\text{all}} \setminus V$. We want to show that $\mathcal{A}$ cannot be optimal on both $\mathcal{I}_{\text{all}} \setminus U$ and $\mathcal{I}_{\text{all}} \setminus V$. Given an appropriate advice string, $\mathcal{A}$ can be optimal on one of these input sets. Without loss of generality, let us assume that $\mathcal{A}$ is optimal on $\mathcal{I}_{\text{all}} \setminus U$, hence $\mathcal{A} = A_U$ for some optimal deterministic algorithm A_U for $\mathcal{I}_{\text{all}} \setminus U$. As $U \neq V$, we conclude that $\mathcal{A} \neq A_V$ for all optimal deterministic algorithms A_V for $\mathcal{I}_{\text{all}} \setminus V$. As we can see from Lemmata 5.25 and 5.31, $\mathbb{E}_V[\text{gain}_{A_V}] > \mathbb{E}_V[\text{gain}_{A_U}] = \mathbb{E}_V[\text{gain}_{\mathcal{A}}]$, i. e., the expected number of correctly guessed bits of A_V on $\mathcal{I}_{\text{all}} \setminus V$ is larger than the number of correctly guessed bits of $\mathcal{A}$ on this set of inputs. Therefore, A_V is better than $\mathcal{A}$ on $\mathcal{I}_{\text{all}} \setminus V$, and hence, $\mathcal{A}$ cannot be optimal on this set of inputs.

Theorem 5.32. *In the monolog model, any Ψ-optimal online algorithm $\mathcal{A}$ needs at least $\tilde{k} \cdot n - 2\,\hat{\omega}(\tilde{k})$ advice bits on any set of instances of size $k = 2^n - \tilde{k}$ containing strings of length n. Hence, $\mathcal{A}$ needs to read at least $\tilde{k} \cdot n - 2\,\hat{\omega}(\tilde{k})$ advice bits to guess*

$$\frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}}$$

bits correctly and thus at most

$$\frac{\hat{\omega}(k)}{k}$$

bits incorrectly in expectation.

Proof. For $\tilde{k} = 0$, the statement is obviously true. Thus, let $\tilde{k} \geq 1$. Assume that an optimal algorithm $\mathcal{A}$ uses $b < \tilde{k} \cdot n - 2\,\hat{\omega}(\tilde{k})$ advice bits. We can view $\mathcal{A}$ as a set $\{A_1, \ldots, A_{2^b}\}$ of $2^b < 2^{\tilde{k} \cdot n - 2\,\hat{\omega}(\tilde{k})}$ different deterministic algorithms according to Fact 1.1. In Lemma 5.24, we have seen that, among all sets of size $\tilde{k}$ with strings of length n that the adversary might choose as the set of excluded strings, there are also $2^{\tilde{k} \cdot n - 2 \cdot \hat{\omega}(\tilde{k})}$ balanced ones. Hence, one of the deterministic algorithms A_i has to be optimal on two different input sets $\mathcal{I}_{\text{all}} \setminus U$ and $\mathcal{I}_{\text{all}} \setminus V$, for two balanced sets U and V.

However, as we know from Lemma 5.31 and the subsequent considerations, a deterministic algorithm that is optimal on the set of inputs $\mathcal{I}_{\text{all}} \setminus U$ cannot be optimal on $\mathcal{I}_{\text{all}} \setminus V$ as well. Hence, $\mathcal{A}$ cannot be optimal on every set of inputs $\mathcal{I}_{\text{all}} \setminus \tilde{\mathcal{I}}$ for all possible choices of $\tilde{\mathcal{I}}$ (of size $\tilde{k}$) if it uses less than $\tilde{k} \cdot n - 2 \cdot \hat{\omega}(\tilde{k})$ advice bits; therefore, $\mathcal{A}$ cannot be Ψ-optimal. □

Corollary 5.33. *For one excluded string ($\tilde{k} = 1$), any Ψ-optimal algorithm needs at least n advice bits.*

Proof. According to Theorem 5.32, the number of advice bits necessary in this case is

$$\tilde{k} \cdot n - 2\,\hat{\omega}(\tilde{k}) = 1 \cdot n - 2 \cdot 0 = n. \qquad \square$$

Corollary 5.34. *For $\tilde{k} = 2^n - 1$ excluded strings, any Ψ-optimal algorithm needs at least n advice bits.*

Proof. The number of advice bits necessary in this case is

$$\begin{aligned}
\tilde{k} \cdot n - 2\,\hat{\omega}(\tilde{k}) &= (2^n - 1) \cdot n - 2 \cdot \hat{\omega}(2^n - 1) \\
&= (2^n - 1) \cdot n - 2 \cdot (n \cdot 2^{n-1} - n) \\
&= (2^n - 1) \cdot n - n \cdot 2^n + 2n \\
&= 2^n \cdot n - n \cdot 2^n + n \\
&= n. \qquad \square
\end{aligned}$$

As we have already discussed at the end of Section 5.2.1.3 in the context of Corollary 5.20, this last case with $\tilde{k} = 2^n - 1$ coincides with the classical advice model. Note that both the upper and the lower bound of n advice bits necessary and sufficient for optimality that have been proven by Böckenhauer et al. [BHK+14] coincide with the upper and lower bounds given in this thesis for the special case of $2^n - 1$ excluded strings and thus an instance set of size 1.

5.2.3 Comparing Upper and Lower Bound

We will need the following well-known estimation for the binomial coefficient (proven, for example, by Knuth [Knu97]). Recall that we use e to denote Euler's number.

Fact 5.35. *For all $n, k \in \mathbb{N}^{\geq 0}$ with $n \geq k$, we have*

$$\binom{n}{k} \leq \left(\frac{e\,n}{k}\right)^k. \qquad \square$$

As a direct consequence of Theorem 5.32 and Lemma 5.13, we obtain the following.

Corollary 5.36. *Any Ψ-optimal online algorithm with advice for 2-*Guess *with a probabilistic adversary needs at least $\tilde{k}(n - \log \tilde{k})$ advice bits on a set of instances of size $\tilde{k}$ containing strings of length n.*

Proof. As we have just proven in Theorem 5.32, the number of advice bits a Ψ-optimal algorithm needs is at least

$$\begin{aligned}\tilde{k}\,n - 2\,\hat{\omega}(\tilde{k}) &\geq \tilde{k}\,n - \tilde{k}\,\log \tilde{k} \qquad (5.35)\\ &= \tilde{k}\,(n - \log \tilde{k}),\end{aligned}$$

where we used Lemma 5.13 for (5.35). □

This statement is contrasted by the following result, which emerges from Theorem 5.17.

Corollary 5.37. *There is a Ψ-optimal online algorithm with advice for 2-*Guess *with a probabilistic adversary that reads $\tilde{k}\,(n - \log \tilde{k} + \log e)$ advice bits.*

Proof. In Theorem 5.17, we have seen that there exists an online algorithm for 2-Guess that reads $\left\lceil \log \binom{2^n}{\tilde{k}} \right\rceil$ advice bits. Applying Fact 5.35 yields

$$\left\lceil \log \binom{2^n}{\tilde{k}} \right\rceil \leq \left\lceil \log\left(\left(\frac{e \cdot 2^n}{\tilde{k}}\right)^{\tilde{k}}\right) \right\rceil \leq \left\lceil \tilde{k}\,(\log e + n - \log \tilde{k}) \right\rceil. \qquad \square$$

Note that these two bounds are almost matching. For some values of $\tilde{k}$, the upper and lower bound match exactly. For example, for $\tilde{k} = 0$, the upper bound for the number of necessary advice bits is 0, as we see from Corollary 5.18, which is obviously tight. For $\tilde{k} = 1$, both bounds are n, due to Corollaries 5.19 and 5.33, and for $\tilde{k} = 2^n - 1$, Corollaries 5.20 and 5.34 yield that the upper and the lower bound is both n, respectively.

5.3 Dialog Model

In this section, we consider a different model with an adaptive oracle, as we already announced in the introduction of this chapter. Let us quickly recall the details of the dialog model. In our previous discussions concerning the monolog model, the algorithm could only receive advice from the oracle before the start of the computation. Conversely, in the dialog model, we give the algorithm the opportunity to ask for advice after every request. More precisely, after receiving the i-th request x_i, where $1 \leq i \leq n$, and before answering this request, the algorithm can demand an advice bit from the oracle, and upon receiving an advice bit, it can either demand another bit or eventually produce the output y_i. Upon receiving such a demand, the oracle is obliged to send exactly one advice bit immediately and is not allowed to send any advice bits other than as an answer to a demand. We denote the number of advice bits demanded by the algorithm in round i by $d_i \in \mathbb{N}^{\geq 0}$ and the advice string it received bit by bit in round i by $\hat{\tau}_i \in \{0,1\}^{d_i}$. We will see that, in this model, the upper and lower bounds for the bit string guessing problem differ substantially from those in the monolog model we considered in Section 5.2. This is due to the fact that the dialog model allows for a more efficient use of advice bits.

In Section 5.3.1, we first give a 2-Guess algorithm reading n advice bits that is optimal according to the definition given in Section 5.2.1.4. The subsequent Section 5.3.2 deals with a complementing lower bound. In Section 5.3.2.1, we present a tight lower bound for the case that the number k of strings in the set of hard input instances constructed by the adversary is odd. Then, in Section 5.3.2.2, we generalize this idea for arbitrary values of k. We conclude the section with a comparison of the upper and lower bounds in Section 5.3.3.

5.3.1 Upper Bound

In this section, we present an algorithm $\mathcal{B}$ with advice for the bit string guessing problem with a probabilistic adversary in the dialog model. Assume that the adversary chooses a probability distribution ψ from the class Ψ defined in Section 5.1 and that the set of excluded strings corresponding to ψ is $\tilde{\mathcal{I}}$, containing $\tilde{k}$ strings of length n. Again, let $\mathcal{I} = \mathcal{I}_{\text{all}} \setminus \tilde{\mathcal{I}}$ contain $k = 2^n - \tilde{k}$ strings. As before, the oracle knows the set $\mathcal{I}$, from which the input string r is drawn uniformly at random. The algorithm only knows the number $\tilde{k}$, and therefore the number k of included strings, but not the actual set $\mathcal{I}$. None of them, neither the algorithm nor the oracle, know the string r in the beginning. In every round i with $1 \leq i \leq n$, though, when the algorithm has already guessed the first $i-1$ bits $r_1, \ldots, r_{i-1}$ of r, the request sent to the algorithm is r_{i-1}. Recall that we use the notation $[r]^i$ for the prefix of length i of a string r. Hence, in every round i, the prefix $[r]^{i-1}$ of r is known to both the algorithm and the oracle. Also recall the definition of C_i as the set of strings that are still candidates for r in round i, according to the knowledge the oracle has about the set $\mathcal{I}$ and the string r so far. Moreover, recall that, for each vertex $[s]^i \in W_\varepsilon$, the subtree rooted at $[s]^i$ is denoted by $T_{[s]^i}$. In the beginning, C_1 is equal to $\mathcal{I}$, and in every subsequent round i, after request r_{i-1} has been sent, the set C_i can be derived by removing all strings from C_{i-1} that do not have the bit r_{i-1} at position $i-1$. Formally, we have $C_i = \{s \in \mathcal{I} \mid [s]^{i-1} = [r]^{i-1}\} = \mathcal{I} \cap T_{[r]^{i-1}}$.

The algorithm $\mathcal{B}$ now works as follows, depending on the number k. If $k = 2^n$, the algorithm's guess in each round i, where $1 \leq i \leq n$, is $g_i := 1$. Otherwise, we have $1 \leq k \leq 2^n - 1$. Then, in round i, after $\mathcal{B}$ has received the request to guess the i-th bit r_i of r but before making its decision, $\mathcal{B}$ asks the oracle for one bit of advice. The oracle has to fulfill this demand by sending exactly one advice bit. If the number of ones at position i in C_i is at least $|C_i|/2$, the oracle sends the bit 1, and otherwise 0. The algorithm $\mathcal{B}$ does not demand any more advice bits from the oracle, but adopts this bit as its guess for r_i. Hence, the advice string $\hat{\tau}_i$ that $\mathcal{B}$ reads in this round only consists of this bit, and we have $g_i := \hat{\tau}_i$.

Lemma 5.38. *In each round i with $1 \leq i \leq n$, the algorithm $\mathcal{B}$ chooses a bit that appears in at least half of the strings in C_i at position i as its guess g_i.*

Proof. If $k = 2^n$, the set $\mathcal{I}$ equals T_ε. The algorithm $\mathcal{B}$ outputs the guess $g_i = 1$ in each round i; hence, we have to show that at least half of the strings in $C_i = \{s \in T_\varepsilon \mid [s]^{i-1} = [r]^{i-1}\}$ have the bit 1 at position i, for $1 \leq i \leq n$. This is clearly the case, because half the strings from T_ε with the same prefix $[r]^{i-1}$ have a one at position i, for $1 \leq i \leq n$.

The claim is also true in the case $1 \leq k \leq 2^n - 1$. In each round i, the algorithm $\mathcal{B}$ chooses $g_i = \hat{\tau}_i$ as its guess for r_i, and $\hat{\tau}_i$ is always a bit that occurs in at least half of the strings in C_i. □

To further analyze $\mathcal{B}$, recall Lemma 5.16, in which we showed that any algorithm that chooses a bit that appears in at least half of the strings in C_i at position i in every round i guesses at least

$$\frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}}$$

bits correctly in expectation. We observe that the proof of Lemma 5.16 does not depend on the model used, so that we can make the same statement about the number of correctly guessed bits of any such algorithm in the dialog model, using the same proof as for Lemma 5.16.

Lemma 5.39. *Let $n \in \mathbb{N}$ and let $\mathcal{I} = \mathcal{I}_{all} \setminus \tilde{\mathcal{I}}$ be a worst-case set of instances that contains all strings of length n except for $0 \leq \tilde{k} \leq 2^n - 1$ strings that are contained in $\tilde{\mathcal{I}}$.*

Furthermore, let $\mathcal{A}$ be an online algorithm with advice that chooses as its guess g_i a bit that appears in at least half of the strings in C_i at position i in every round i, where $1 \leq i \leq n$. Then the expected number of correctly guessed bits of $\mathcal{B}$ on $\mathcal{I}$ in the dialog model is given by

$$\chi(n, \tilde{k}) = \frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}}.$$

Proof. See the proof of Lemma 5.16. □

Thus, we can already make a statement about the number of advice bits that are necessary to achieve the same expected number of correctly guessed bits as the Ψ-optimal algorithm $\mathcal{A}$ from Section 5.2.1.1 in the monolog model.

Theorem 5.40. *In the dialog model, given a probabilistic adversary that excludes the set $\tilde{\mathcal{I}}$ containing $\tilde{k}$ strings of length n from the set $\mathcal{I}_{all}$ of possible strings and chooses one of the remaining strings from $\mathcal{I} = \mathcal{I}_{all} \setminus \tilde{\mathcal{I}}$ uniformly at random, there is an online algorithm $\mathcal{B}$ with advice for the bit string guessing problem with a probabilistic adversary that guesses*

$$\frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}}$$

bits correctly and thus at most

$$\frac{\hat{\omega}(k)}{k}$$

bits incorrectly in expectation, using 0 advice bits if $\tilde{k} = 0$ and n advice bits if $1 \leq \tilde{k} \leq 2^n - 1$.

Proof. The number of advice bits needed is indicated in the description of algorithm $\mathcal{B}$ in this section. The number of correctly guessed bits follows from Lemmata 5.38 and 5.39. The expected number of incorrectly guessed bits then follows from Observation 5.9 again. □

Thus, we know how to achieve the same number of correctly guessed bits in the dialog model as in the monolog model; but is the algorithm $\mathcal{B}$ also Ψ-optimal in the dialog model, just like $\mathcal{A}$ is in the monolog model? As it turns out, we can again reuse a result from before, where we have proven which properties an algorithm must have to be Ψ-optimal in the monolog model. In Lemma 5.22, we have proven that any algorithm for the bit string guessing problem with a probabilistic adversary in the monolog model is Ψ-optimal if and only if in each round i, it chooses a bit that occurs in at least half of the strings in C_i at position i as its guess g_i. The proof of Lemma 5.22 is totally independent of the model used. Hence, the result carries over to the dialog model, yielding the statement that an algorithm in the dialog model is Ψ-optimal if and only if in each round i, it chooses a bit that occurs in at least half of the strings in C_i at position i.

Lemma 5.41. *In the dialog model, an online algorithm for the bit string guessing problem with a probabilistic adversary that excludes $\tilde{k}$ strings from an instance consisting of 2^n strings of length n is Ψ-optimal if and only if it chooses its guess g_i from the set $\arg\max_{s_i}\{|\{s \in C_i \mid s_i = 0\}|, |\{s \in C_i \mid s_i = 1\}|\}$ in each round i.*

Proof. See the proof of Lemma 5.22. □

This leads to the same number of correctly guessed bits that any Ψ-optimal algorithm can achieve in the dialog model as in the monolog model.

Theorem 5.42. *Algorithm $\mathcal{B}$ is Ψ-optimal for 2-GUESS in the dialog model with a probabilistic adversary that excludes $\tilde{k}$ strings from an instance consisting of 2^n strings of length n. Hence, the expected number of correctly guessed bits of any Ψ-optimal online algorithm playing against such an adversary is*

$$\frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}},$$

and the expected number of incorrectly guessed bits is at most

$$\frac{\hat{\omega}(k)}{k}.$$

Proof. According to Lemma 5.38, in each round i with $1 \leq i \leq n$, the algorithm $\mathcal{B}$ chooses a bit that occurs in at least half the strings from C_i at position i as its guess g_i. Hence, according to Lemma 5.41, $\mathcal{B}$ is Ψ-optimal.

Therefore, each Ψ-optimal algorithm must guess as many bits correctly in expectation as $\mathcal{B}$, and as we know from Theorem 5.40, this number of bits is exactly

$$\frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}}.$$

□

The Ψ-optimal algorithm $\mathcal{A}$ for the monolog model that we described in Section 5.2.1.1 uses $\log\binom{2^n}{\tilde{k}}$ advice bits. Moreover, we showed that at least $\tilde{k}\,(n - \log\tilde{k})$ advice bits are necessary to achieve Ψ-optimality in the monolog model. Hence, for $\tilde{k} = 0$, the algorithm $\mathcal{A}$ does not read any advice bits, just like the algorithm $\mathcal{B}$ in the dialog model. For all other values of $\tilde{k}$, however, we have proven that in the dialog model, Ψ-optimality can be achieved with only n advice bits, whereas in the monolog model, the number of advice bits necessary can be exponential in n. For a more detailed discussion, see Section 5.4.

5.3.2 Lower Bound

Now that we have proven that n bits of advice are sufficient for any algorithm to be Ψ-optimal in the dialog model, let us turn our attention to the lower bound for the number of advice bits necessary to be Ψ-optimal.

5.3.2.1 Bound for Odd k

We will give a tight lower bound of n advice bits in the case that $\tilde{k}$ is odd (and thus $k = 2^n - \tilde{k}$ is odd) soon. However, before we do so, we have to make a few technical considerations.

Observation 5.43. *For any odd natural number $k \in \mathbb{N}^{\geq 1}$, either $\lfloor k/2 \rfloor$ or $\lceil k/2 \rceil$ is an odd number again.*

Proof. Since k is odd, $k/2$ is not a natural number. Hence, the two values $\lfloor k/2 \rfloor$ and $\lceil k/2 \rceil$ cannot be equal, and as the difference between them is at most 1, one of them must be odd and the other one even. □

Observation 5.44. *For every odd natural number $k =: k_0 \in \mathbb{N}^{\geq 1}$ and every natural number $n \in \mathbb{N}^{\geq 0}$, there is a unique sequence $K := (k_0, k_1, \dots, k_n)$ of $n + 1$ odd natural numbers $k_i \in \mathbb{N}^{\geq 1}$, such that for each i with $1 \leq i \leq n$, we have*

$$k_i = \left\lfloor \frac{k_{i-1}}{2} \right\rfloor \quad \textit{or} \quad k_i = \left\lceil \frac{k_{i-1}}{2} \right\rceil.$$

Proof. For any i with $1 \leq i \leq n$, exactly one of the values $\lfloor k_{i-1}/2 \rfloor$ and $\lceil k_{i-1}/2 \rceil$ is odd due to Observation 5.43; hence, the sequence is unique. The sequence starts with a value $k_0 \geq 1$. If, for some j with $0 \leq j \leq n-1$, we have $k_j \geq 1$, the next value will be $k_{j+1} \geq \lceil 1/2 \rceil = 1$. Therefore, all values are at least 1. □

We identify the set T_ε of all 2^n strings of length n with a complete binary tree of depth n, as we did before. Each leaf represents one string, and a computation of an algorithm on the string r is represented by a route through the tree from the root ε to the leaf r. Let us define the level of a vertex v as the length of the shortest path from the root to v. Thus, the root is on level 0, and the leaves are on level n. Again, we identify an inner vertex v on level i that lies on the shortest path from the root to a string s with the prefix $[s]^i$ of s, for $0 \leq i \leq n$. This is exactly the part of the string s that an algorithm already knows when it reaches the vertex $v = [s]^i$ on its route through the tree. Recall that we defined the subtree rooted at a vertex v to be called T_v. The binary tree representing T_ε has the vertex set $W_\varepsilon = \{[s]^i \mid s \in T_\varepsilon, 0 \leq i \leq n\}$. Each inner vertex $[s]^{i-1}$ on the path from ε to the string s has two children, namely $[s]^i$, which also lies on the path to s, and $([s]^i)^*$, which does not lie on the path to s. Recall that $[s]^i$ and $([s]^i)^*$ are called siblings.

To give a lower bound on the number of advice bits necessary to be Ψ-optimal, we consider only a restricted set of instance sets the adversary may choose $\mathcal{I}$ from. Actually, we will again consider only balanced sets, as we already did for the lower bound in the monolog model. This time, however, we will consider balanced sets as sets of *included* strings, whereas before we considered them as sets of *excluded* strings. Anyway, as before, the lower bound for the number of advice bits necessary if the adversary might choose *any* possible set as $\mathcal{I}$ cannot be smaller than the one we obtain by restricting $\mathcal{I}$ to certain balanced sets. First, recall the definition of balanced sets from Section 5.2.2, in particular the notion of a balanced set as a selection of leaves in the binary tree T_ε. A balanced set is a selection of leaves such that, for each inner vertex $[s]^{i-1}$, the number of selected leaves in $T_{[s]^i}$ differs by at most one from the number of selected leaves in $T_{([s]^i)^*}$. In addition, we will from now on say that a vertex $[s]^{i-1}$ is balanced if the number of selected leaves in $T_{[s]^i}$ differs by at most one from the number of selected leaves in $T_{([s]^i)^*}$.

Now let us describe how to choose the class $\mathcal{C}$ of balanced sets of size k that the adversary may choose $\mathcal{I}$ from. For each string $r \in T_\varepsilon$, we add a corresponding balanced set U_r to $\mathcal{C}$. To define the set U_r, we first generate a labeling $\kappa_r \colon W_\varepsilon \to \{0, \ldots, k\}$ of the vertices of T_ε. The labeling κ_r assigns the number of strings from the balanced set U_r in the subtree T_v to each vertex v. Thus, for each such labeling κ_r that corresponds to a balanced set U_r and each inner vertex $[s]^{i-1}$ with label $\kappa_r([s]^{i-1})$, where $1 \leq i \leq n$, the labels $\kappa_r([s]^i)$ and $\kappa_r(([s]^i)^*)$ of its two children must add up to $\kappa_r([s]^{i-1})$. To generate a labeling κ_r corresponding to a balanced set of size k, the label of the root must be $\kappa_r(\varepsilon) = k$, and each leaf must be labeled either with 0 or with 1.

Let $K = (k_0, k_1, \ldots, k_n)$ be the unique sequence of $n+1$ odd numbers that corresponds to $k = k_0$ satisfying the condition from Observation 5.44. Furthermore, for any i with $1 \leq i \leq n$, let $k_i^* := k_{i-1} - k_i$. For each string $r \in T_\varepsilon$, we generate the labeling κ_r corresponding to r as follows.

(a) For each i with $0 \leq i \leq n$, the vertex $[r]^i$ on the path from the root $[r]^0 = \varepsilon$ to $[r]^n = r$ is labeled with $\kappa_r([r]^i) := k_i$.

(b) For each i with $1 \leq i \leq n$, the sibling $([r]^i)^*$ of the vertex $[r]^i$ is labeled with $\kappa_r(([r]^i)^*) := k_i^* = k_{i-1} - k_i$.

(c) The labels of the remaining vertices are generated top-down, level by level. For each vertex v with label $\kappa_r(v)$ and two yet unlabeled children, the left child is assigned the label $\lceil \kappa_r(v)/2 \rceil$ and the right one $\lfloor \kappa_r(v)/2 \rfloor$.

An example can be found in Figure 5.5.

Recall that, for every set S of strings and each vertex v, we defined $\vartheta(S, v)$ to be the number of leaves from S in T_v. Now we want to determine the balanced set U_r that corresponds to r. To this end, for each corresponding labeling κ_r, we interpret the label $\kappa_r(v)$ of each vertex v as the number of strings from U_r in the subtree T_v, i. e., $\vartheta(U_r, v) := \kappa_r(v)$. When we generated the labelings, we made sure that they actually correspond to a balanced set each, as we will show now.

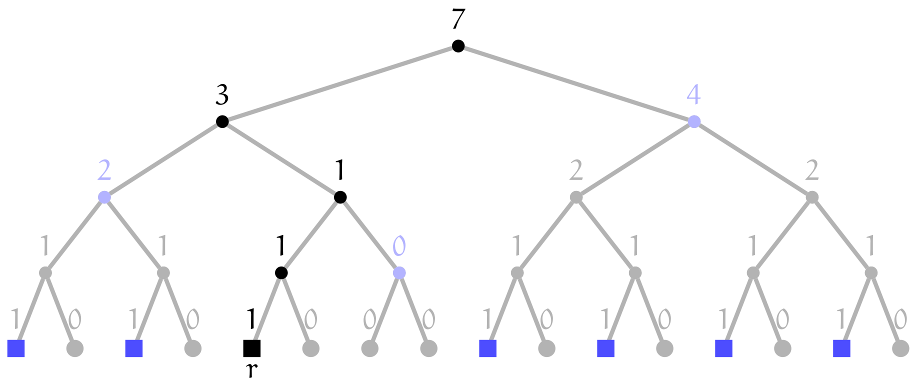


Figure 5.5. An example of the case $n = 4$ and $k = 7$. The unique sequence of $n + 1 = 5$ odd numbers corresponding to k is $(7, 3, 1, 1, 1)$. The string r is marked in black, the balanced set U_r corresponding to r consists of all rectangular leaves (black or blue). The vertices on the path from the root to r are labeled according to (a) with the values $7, 3, 1, 1, 1$, from top to bottom; these vertices are drawn in black. Vertices that are labeled according to (b) are colored blue; the remaining vertices, which are labeled according to (c), are colored gray.

Observation 5.45. *For any $n, k \in \mathbb{N}^{\geq 1}$, where $1 \leq k \leq 2^n - 1$ and k odd, and each string $r \in T_\varepsilon$, the generated labeling κ_r corresponds to a balanced set U_r of size k.*

Proof. As we have already mentioned, the label of a vertex v indicates the number of strings from U_r in the subtree T_v. Properties (a) and (b) make sure that the labels $\kappa_r([r]^i) = k_i$ and $\kappa_r(([r]^i)^*) = k_{i-1} - k_i$ of $[r]^i$ and $([r]^i)^*$, respectively, add up to $\kappa_r([r]^{i-1}) = k_{i-1}$, i. e., the label of their parent $[r]^{i-1}$. Together with Observation 5.44, we obtain taht $\lfloor k_{i-1}/2 \rfloor$ of the k_{i-1} strings from U_r in the subtree $T_{[r]^{i-1}}$ are located in $T_{[r]^i}$ and $\lceil k_{i-1}/2 \rceil$ in $T_{([r]^i)^*}$ or vice versa, making sure that every vertex on the path from ε to r is balanced. All remaining vertices are balanced due to (c).

Due to Observation 5.44 and (a), we know that the root has label $\kappa_r(\varepsilon) = k_0 = k$, and since $k \leq 2^n - 1$, all leaves are either labeled with a 0 or with a 1. Hence, the generated labeling corresponds to a balanced set U_r of size k. $\square$

Lemma 5.46. *For any $n, k \in \mathbb{N}^{\geq 1}$, where $1 \leq k \leq 2^n - 1$ and k odd, and any string $r \in T_\varepsilon$, the balanced set U_r of size k corresponding to r contains the string r.*

Proof. Consider the sequence $K = (k_0, \ldots, k_n)$ with $k_i \geq 1$ for $0 \leq i \leq n$. As $k \leq 2^n - 1$, the last value k_n must be exactly 1, and according to our construction of the labeling κ_r, this value k_n is assigned to the leaf r. Due to the construction of the balanced set U_r corresponding to κ_r, the set U_r must contain r. $\square$

Lemma 5.47. *For any $n, k \in \mathbb{N}^{\geq 1}$, where $1 \leq k \leq 2^n - 1$ and k odd, consider the unique sequence $(k_0, \ldots, k_n)$ of length $n+1$ corresponding to k from Observation 5.44. Then, $k_i^* \neq k_i$, for all i with $1 \leq i \leq n$.*

Proof. We have

$$k_i^* = k_{i-1} - k_i = \begin{cases} k_{i-1} - \left\lfloor \frac{k_{i-1}}{2} \right\rfloor = \left\lceil \frac{k_{i-1}}{2} \right\rceil & \text{if } k_i = \left\lfloor \frac{k_{i-1}}{2} \right\rfloor, \\ k_{i-1} - \left\lceil \frac{k_{i-1}}{2} \right\rceil = \left\lfloor \frac{k_{i-1}}{2} \right\rfloor & \text{if } k_i = \left\lceil \frac{k_{i-1}}{2} \right\rceil. \end{cases}$$

Due to Observation 5.44, each k_{i-1} is an odd number, and due to Observation 5.43, either k_i or k_i^* is odd and the other one even. Thus, k_i and k_i^* differ by exactly 1. □

Now let us consider all strings s' in the subtree $T_{[s]^i}$ rooted at $[s]^i$, for some vertex $[s]^i$. All these strings have the same common prefix $[s]^i$. Let us define the class $\mathcal{C}_{[s]^i} \subseteq \mathcal{C}$ to contain every balanced set $U_{s'}$ corresponding to such a string s'.

Observation 5.48. *For any $n, k \in \mathbb{N}^{\geq 1}$, where $1 \leq k \leq 2^n - 1$ and k odd, consider only balanced sets of size k. Then, $\mathcal{C}_{[s]^i} \cup \mathcal{C}_{([s]^i)^*} = \mathcal{C}_{[s]^{i-1}}$ for each inner vertex $[s]^{i-1}$.*

Proof. This is a direct consequence of the definition of the classes $\mathcal{C}_{([s]^i)^*}$, $\mathcal{C}_{[s]^i}$, and $\mathcal{C}_{[s]^{i-1}}$. □

Lemma 5.49. *For any $n, k \in \mathbb{N}^{\geq 1}$, where $1 \leq k \leq 2^n - 1$ and k odd, consider two strings $r, s \in T_\varepsilon$ with the same prefix $[r]^i = [s]^i$, for some i with $0 \leq i \leq n$. Furthermore, consider the balanced set U_r of size k corresponding to r. Then, the number of strings from U_r in $T_{[s]^i}$ is k_i, and the number of strings from U_r in $T_{([s]^i)^*}$ is k_i^*.*

Proof. As $[r]^i = [s]^i$, we know that $U_r \in \mathcal{C}_{[s]^i}$, and that $([r]^i)^* = ([s]^i)^*$. Due to properties (a) and (b), we have $\kappa_r([r]^i) = \kappa_s([s]^i) = k_i$ and $\kappa_r(([r]^i)^*) = \kappa_s(([s]^i)^*) = k_i^*$. Hence, according to our construction of the labelings, the vertex $[r]^i = [s]^i$ has the same label k_i in the labelings corresponding to r and s, respectively, and the sibling $([r]^i)^*$ of $[r]^i$ has the same label k_i^* in the labelings corresponding to r and s, respectively.

The claim follows, since the labels of the vertices $[r]^i$ and $([r]^i)^*$ indicate the number of strings from the corresponding balanced set in the subtrees $T_{[r]^i}$ and $T_{([r]^i)^*}$, respectively. □

An example is given in Figure 5.6.

Corollary 5.50. *For any $n, k \in \mathbb{N}^{\geq 1}$, where $1 \leq k \leq 2^n - 1$ and k odd, consider only balanced sets of size k. Then, for each inner vertex $[s]^{i-1}$, the two classes of balanced sets $\mathcal{C}_{[s]^i}$ and $\mathcal{C}_{([s]^i)^*}$ are disjoint.*

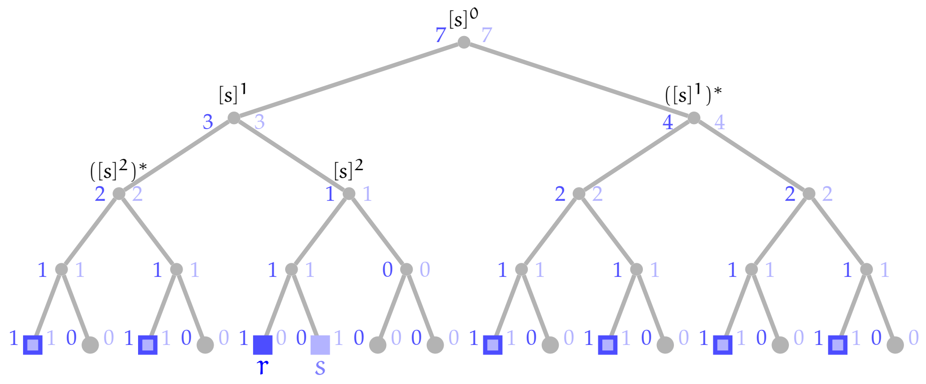


Figure 5.6. The example for $n = 4$ and $k = 7$ from Figure 5.5 for $i = 2$. The strings $r = 0100$ and $s = 0101$ have the same prefix $[s]^2 = 01$ (even if this is not their longest common prefix); hence, $U_r \in \mathcal{C}_{[s]^2}$. The labels of κ_r and κ_s are written next to each vertex; those of κ_r to the left, those of κ_s to the right. The vertices $[s]^0$, $[s]^1$, and $[s]^2$ have the same labels in κ_r and κ_s due to (a). The siblings $([s]^1)^*$ and $([s]^2)^*$ have the same labelings in κ_r and κ_s due to (b). Hence, the number of strings from U_r in $T_{[s]^2}$ is the same as the number of strings from U_s in $T_{[s]^2}$, namely k_i. Furthermore, the number of strings from U_r in $T_{([s]^2)^*}$ is the same as the number of strings from U_s in $T_{([s]^2)^*}$, namely k_i^*.

Proof. According to Lemma 5.49, for every balanced set $U_r \in \mathcal{C}_{[s]^i}$ that corresponds to a string $r \in T_\varepsilon$, there are k_i strings from U_r in $T_{[r]^i}$, and for every balanced set $U_{r'} \in \mathcal{C}_{([s]^i)^*}$ corresponding to a string $r' \in T_\varepsilon$, there are k_i strings from $U_{r'}$ in $T_{([r]^i)^*}$ and k_i^* strings from $U_{r'}$ in $T_{([r']^i)^*} = T_{[r]^i}$. Due to Lemma 5.47, $k_i \neq k_i^*$, and thus U_r and $U_{r'}$ must be different. $\square$

Now, for each vertex $[s]^i$, let us define the class $\mathcal{D}_{[s]^i}$ as the set of optimal deterministic algorithms for the balanced sets from $\mathcal{C}_{[s]^i}$. Note that there may be several different optimal deterministic algorithms for the same balanced set.

Observation 5.51. *For any $n, k \in \mathbb{N}^{\geq 1}$, where $1 \leq k \leq 2^n - 1$ and k odd, consider the optimal deterministic algorithms for balanced sets of size k. Then, for each inner vertex $[s]^{i-1}$, we have $\mathcal{D}_{[s]^i} \cup \mathcal{D}_{([s]^i)^*} = \mathcal{D}_{[s]^{i-1}}$.*

Proof. This is a direct consequence of the definition of the classes $\mathcal{D}_{([s]^i)^*}$, $\mathcal{D}_{[s]^i}$, and $\mathcal{D}_{[s]^{i-1}}$. $\square$

For each labeling κ_s and each inner vertex $[s]^{i-1}$, one of the two children $[s]^i$ and $([s]^i)^*$ of $[s]^{i-1}$ is labeled with k_i and the other one with k_i^*. Let us define the one with the label $\max\{k_i, k_i^*\}$ to be the child $f_{s,i}$ and the one with the label $\min\{k_i, k_i^*\}$ the child $f_{s,i}^*$.

Recall that we can represent any deterministic algorithm by a marking of the edges, as we described in Section 5.2.2 on page 113. This marking maps every inner vertex in the tree to one of its two outgoing edges. For each vertex v, we call the marked outgoing edge the favored edge of v.

Lemma 5.52. *For any $n, k \in \mathbb{N}^{\geq 1}$, where $1 \leq k \leq 2^n - 1$ and k odd, consider some vertex $[s]^i$ in the tree T_ε. Then, for all algorithms from the class $\mathcal{D}_{[s]^i}$ of deterministic algorithms that are optimal for some balanced set of size k from $\mathcal{C}_{[s]^i}$, the favored edge at vertex $[s]^{i-1}$ is $([s]^{i-1}, f_{s,i})$.*

Proof. Consider some arbitrary but fixed balanced set $U_r \in \mathcal{C}_{[s]^i}$ of size k and an arbitrary but fixed optimal deterministic algorithm A_r for U_r, for some string $r \in T_\varepsilon$. Hence, $A_r \in \mathcal{D}_{[s]^i}$.

Due to Lemma 5.49, the number of strings from U_r in $T_{[s]^i}$ is k_i, and the number of strings from U_r in $T_{([s]^i)^*}$ is k_i^*. These two values k_i and k_i^* are different due to Lemma 5.47. The favored edge at $[s]^{i-1}$ of any algorithm that is optimal on U_r must point to the child with the larger label, i. e., to the child $f_{s,i}$ with the label $\max\{k_i, k_i^*\}$.

Since we chose A_r to be an arbitrary algorithm from $\mathcal{D}_{[s]^i}$, each algorithm from $\mathcal{D}_{[s]^i}$ has the favored edge $([s]^{i-1}, f_{s,i})$ at vertex $[s]^{i-1}$. □

An example can be found in Figure 5.7.

Corollary 5.53. *For any $n, k \in \mathbb{N}^{\geq 1}$, where $1 \leq k \leq 2^n - 1$ and k odd, consider the optimal deterministic algorithms for balanced sets of size k. Then, for each inner vertex $[s]^{i-1}$, the two classes of algorithms $\mathcal{D}_{[s]^i}$ and $\mathcal{D}_{([s]^i)^*}$ are disjoint.*

Proof. Let $s' \in T_\varepsilon$ be the string with $[s]^{i-1} = [s']^{i-1}$ and $[s]^i \neq [s']^i$. Hence, $[s']^i = ([s]^i)^*$. Then, the claim follows directly from Lemma 5.52, because any algorithm from $\mathcal{D}_{[s]^i}$ has the favored edge $([s]^{i-1}, f_{s,i})$, and any algorithm from $\mathcal{D}_{([s]^i)^*}$ has the favored edge $([s]^{i-1}, f_{s',i}) = ([s]^{i-1}, f^*_{s,i}) \neq ([s]^{i-1}, f_{s,i})$. □

Lemma 5.54. *For any $n, k \in \mathbb{N}^{\geq 1}$, where $1 \leq k \leq 2^n - 1$ and k odd, consider two different strings $r, r' \in T_\varepsilon$ with the same common prefix $[r]^{i-1} = [r']^{i-1}$ that differ at position i, for some i with $1 \leq i \leq n$. Furthermore, consider the balanced sets $U_r \in \mathcal{C}_{[r]^i}$ and $U_{r'} \in \mathcal{C}_{([r]^i)^*}$ of size k corresponding to r and r', respectively. Assume that the adversary chose one of these two balanced sets as the set $\mathcal{I}$. Then, any online algorithm $\mathcal{B}$ with advice has to read a different advice string on U_r than on $U_{r'}$ to be optimal on both.*

Proof. We prove the claim by contradiction. Assume that $\mathcal{B}$ reads the same advice string, regardless of whether the adversary has chosen U_r or $U_{r'}$ as $\mathcal{I}$. Both balanced sets U_r and $U_{r'}$ are contained in $\mathcal{C}_{[r]^{i-1}}$ due to Observation 5.48, so without loss of generality we can assume that the deterministic algorithm used

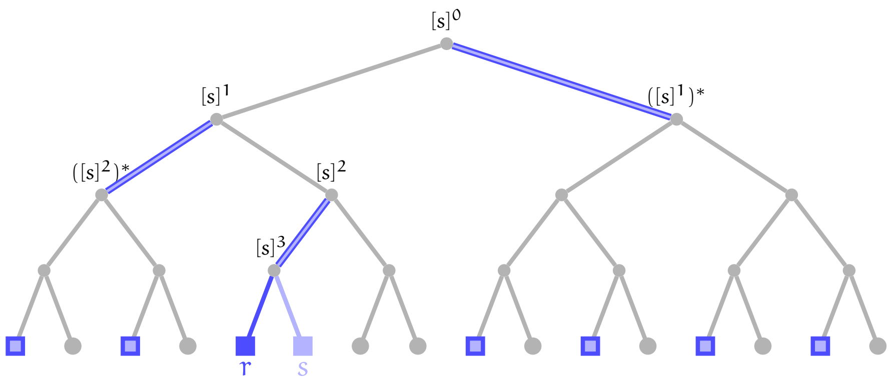


Figure 5.7. The example for $n = 4$ and $k = 7$ from Figure 5.5 for $i = 2$. The strings $r = 0100$ and $s = 0101$ have the same prefix $[s]^i = 01$. The strings from the balanced sets U_r and U_s are marked by blue rectangles; those of U_s in a light blue, and those from U_r in a darker one. The favored edges of A_s at vertices $[s]^0, \dots, [s]^3$ are marked in a light blue and those of A_r in a darker blue. The vertices $[s]^1$, $[s]^2$, and their siblings have the same labels in κ_r and κ_s, respectively. Hence, the favored edges of A_r and A_s at the vertices $[s]^0$ and $[s]^1$ are the same.

is from $\mathcal{D}_{[r]^{i-1}}$, the class of optimal deterministic algorithms for balanced sets from $\mathcal{C}_{[r]^{i-1}}$. We consider two cases depending on which set the adversary chose.

For the first case, we assume that the adversary chose $U_r \in \mathcal{C}_{[r]^i}$ as $\mathcal{I}$. Then, due to Lemma 5.46, it is possible that the string that was drawn uniformly at random from U_r was r. Due to Observation 5.51, the algorithm used from the class $\mathcal{D}_{[r]^{i-1}}$ must be from $\mathcal{D}_{[r]^i}$ or $\mathcal{D}_{([r]^i)^*}$. As $U_r \in \mathcal{C}_{[r]^i}$ and since $\mathcal{D}_{[r]^i}$ and $\mathcal{D}_{([r]^i)^*}$ are disjoint as a consequence of Corollary 5.53, no algorithm from $\mathcal{D}_{([r]^i)^*}$ can be optimal on U_r. Hence, in the case that the adversary chose U_r as $\mathcal{I}$, a deterministic algorithm from $\mathcal{D}_{[r]^i}$ has to be used to be optimal.

For the second case, let us assume that the adversary chose $U_{r'} \in \mathcal{C}_{([r]^i)^*}$ as $\mathcal{I}$, and that the string that was drawn uniformly at random from $U_{r'}$ was r', which is again possible due to Lemma 5.46. In this case, with a similar argument as before, a deterministic algorithm from $\mathcal{D}_{([r]^i)^*}$ has to be used to be optimal on $U_{r'}$.

As we have seen, two different deterministic algorithms have to be chosen by $\mathcal{B}$ in these two cases. However, in both cases, $\mathcal{B}$ has read the same prefix $[r]^{i-1} = [r']^{i-1}$ so far in round i, and due to our assumption, it has also read the same advice string. Hence, it has to make the same deterministic choice in both cases, and is therefore not optimal on either U_r or $U_{r'}$. □

Lemma 5.55. *Assume that for some $n, k \in \mathbb{N}$, where $1 \leq k \leq 2^n$ and k odd, and some string $r \in T_\varepsilon$, the adversary chooses the balanced set U_r of size k from $\mathcal{C}$ as the set $\mathcal{I}$ containing strings of length n, and that the string r is drawn from $\mathcal{I}$ as the actual input* *string. Then, there is no online algorithm $\mathcal{B}$ that reads less than n advice bits and that is optimal for all strings $r \in T_\varepsilon$.*

Proof. Consider two arbitrary but fixed balanced sets $U_r, U_{r'} \in \mathcal{C}$ corresponding to the strings $r, r' \in T_\varepsilon$. Let i be the position at which r and r' differ for the first time, where $1 \leq i \leq n$; hence, $[r]^{i-1} = [r']^{i-1}$ and $[r]^i \neq [r']^i$. Thus, $U_r \in \mathcal{C}_{[r]^i}$ and $U_{r'} \in \mathcal{C}_{([r]^i)^*}$. Due to Lemma 5.54, to be optimal on both balanced sets, $\mathcal{B}$ has to read a different advice string in the case that the adversary chose U_r as $\mathcal{I}$ and r is drawn as the actual input string than in the case that the adversary chose $U_{r'}$ as $\mathcal{I}$ and then r' is drawn as the actual input string. There are 2^n strings r in T_ε, and therefore 2^n corresponding balanced sets U_r. Due to Corollary 5.50, all classes of balanced sets $\mathcal{C}_{[s]^i}$ and $\mathcal{C}_{([s]^i)^*}$ are pairwise disjoint, for all inner vertices $[s]^{i-1}$, which directly implies that all balanced sets U_r are pairwise different, for all $r \in T_\varepsilon$. Hence, $\mathcal{C}$ contains 2^n balanced sets.

To be optimal on each balanced set from $\mathcal{C}$, any algorithm $\mathcal{B}$ needs to read a different advice string for each balanced set from $\mathcal{C}$, as we have just seen. □

We are now ready to prove a matching lower bound to the upper bound of n advice bits that are sufficient for an algorithm to be Ψ-optimal in the dialog model. Recall that $\hat{\omega}(\tilde{k})$ is the total Hamming weight of the binary representations of all natural numbers from 0 to $\tilde{k} - 1$.

Theorem 5.56. *In the dialog model, any Ψ-optimal online algorithm needs at least n advice bits on a set of instances of size $k = 2^n - \tilde{k}$ containing strings of length n, where k is odd and $1 \leq k \leq 2^n - 1$. Hence, any algorithm needs to read at least n advice bits to guess*

$$\frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}}$$

bits correctly and thus at most

$$\frac{\hat{\omega}(k)}{k}$$

bits incorrectly in expectation.

Proof. This follows directly from Lemma 5.55 and Theorem 5.42. □

5.3.2.2 Generalization for General k

Now we want to use the results from Section 5.3.2.1 to give a lower bound for all (reasonable) values of k, i. e., for all values in the range $1 \leq k \leq 2^n$. To this end, we use the fact that we can write each natural number k as $k = 2^m \cdot \ell$ for natural numbers m, ℓ with $m \in \mathbb{N}^{\geq 0}$ and $\ell \in \mathbb{N}^{\geq 1}$ and ℓ being odd.

Again, for each string $r \in T_\varepsilon$, we add a balanced set U_r to a class $\mathcal{C}$ of balanced sets of size k that the adversary may choose $\mathcal{I}$ from. To define the set U_r, we first generate a corresponding labeling $\lambda_r \colon W_\varepsilon \to \{0, \dots, k\}$ of the vertices, as we already did in Section 5.3.2.1.

As $k = 2^m \cdot \ell \leq 2^n$, we have $m + \log \ell \leq n$, and thus $n - m \geq 0$. Then, due to Observation 5.44, there is a unique sequence $L = (\ell_0, \ell_1, \dots, \ell_{n-m})$ of $n - m + 1$ odd numbers corresponding to $\ell =: \ell_0$ with $\ell_i \geq 1$, for $0 \leq i \leq n - m$.

Observation 5.57. *For every natural number $k =: k_0 \in \mathbb{N}^{\geq 1}$ and every natural number $n \in \mathbb{N}^{\geq 0}$, there is a unique sequence $K = (k_0, k_1, \dots, k_n)$ of $n + 1$ natural numbers $k_i \in \mathbb{N}^{\geq 1}$, such that for each i with $1 \leq i \leq n$, we have*

$$k_i = \left\lfloor \frac{k_{i-1}}{2} \right\rfloor \quad \textit{or} \quad k_i = \left\lceil \frac{k_{i-1}}{2} \right\rceil,$$

and, furthermore, for each i with $1 \leq i \leq m$, we have

$$k_i = \frac{k_{i-1}}{2}.$$

Proof. We define the sequence $K = (k_0, k_1, \dots, k_n)$ of length $n + 1$ as follows. Let

$$k_i := \begin{cases} k/2^i & \text{for } 0 \leq i \leq m, \\ \ell_{i-m} & \text{for } m + 1 \leq i \leq n. \end{cases}$$

For $1 \leq i \leq m$, we have $k_i = k/2^i = k_{i-1}/2$, so for these values, the claim follows directly. Furthermore, $k_m = k/2^m = \ell = \ell_0$ is the first number of the unique sequence $(\ell_0, \ell_1, \dots, \ell_{n-m})$. Thus, for all values k_i with $m + 1 \leq i \leq n$, we have $k_i = \lfloor k_{i-1}/2 \rfloor$ or $k_i = \lceil k_{i-1}/2 \rceil$ due to Observation 5.44. Since $\ell_i \geq 1$ for all $\ell_i \in L$, also $k_i \geq 1$ for all $k_i \in K$. □

Additionally, for any i with $1 \leq i \leq n$, let $k_i^* := k_{i-1} - k_i$. We generate a corresponding labeling λ_r for each string $r \in T_\varepsilon$ in the same manner as in Section 5.3.2.1.

(a) For each i with $0 \leq i \leq n$, the vertex $[r]^i$ on the path from the root $[r]^0 = \varepsilon$ to $[r]^n = r$ is labeled with $\lambda_r([r]^i) := k_i$.

(b) For each i with $1 \leq i \leq n$, the sibling $([r]^i)^*$ of the vertex $[r]^i$ is labeled with $\lambda_r(([r]^i)^*) := k_i^* = k_{i-1} - k_i$.

(c) The labels of the remaining vertices are generated top-down, level by level. For each vertex v with label $\lambda_r(v)$ and two yet unlabeled children, the left child is assigned the label $\lceil \lambda_r(v)/2 \rceil$ and the right one $\lfloor \lambda_r(v)/2 \rfloor$.

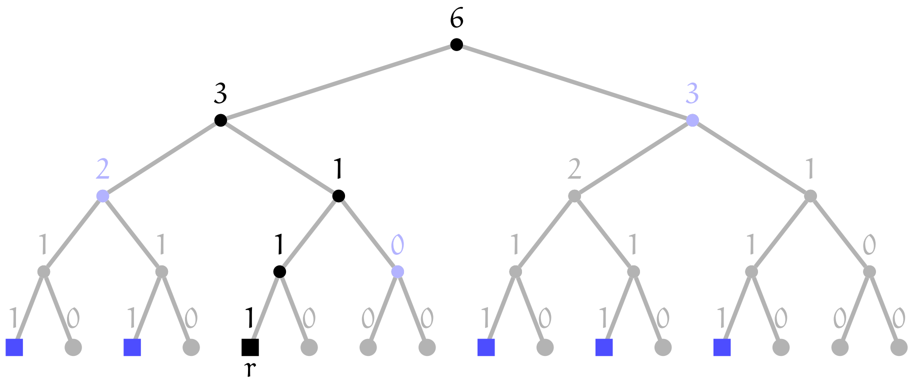


Figure 5.8. An example for the case $n = 4$ and $k = 6 = 2^1 \cdot 3$, implying that $m = 1$ and $\ell = 3$. Hence, the unique sequence of $n - m + 1 = 4$ odd numbers corresponding to ℓ is $L = (3, 1, 1, 1)$. The unique sequence of $n + 1 = 5$ numbers corresponding to k is $K = (6, 3, 1, 1, 1)$. The string r is marked as a black square; the balanced set U_r corresponding to r consists of all rectangular leaves (black or blue). The vertices on the path from the root to r are labeled according to (a) with the values $6, 3, 1, 1, 1$ from top to bottom and are colored black. All vertices that are labeled according to (b) are colored blue; all remaining vertices are labeled according to (c).

An example can be found in Figure 5.8.

Just as before, we can show that each generated labeling corresponds to a balanced set.

Observation 5.58. *For any $n, k \in \mathbb{N}^{\geq 1}$, where $1 \leq k \leq 2^n$, and each string $r \in T_\varepsilon$, the generated labeling λ_r corresponds to a balanced set U_r of size k.*

Proof. Again, the label of a vertex v indicates the number of strings from the set U_r in the subtree T_v. Properties (a) and (b) make sure that the labels $\lambda_r([r]^i) = k_i$ and $\lambda_r(([r]^i)^*) = k_{i-1} - k_i$ of $[r]^i$ and $([r]^i)^*$, respectively, add up to $\lambda_r([r]^{i-1}) = k_{i-1}$, i. e., the label of their parent $[r]^{i-1}$. Together with Observation 5.57, we obtain that $\lfloor k_{i-1}/2 \rfloor$ of the k_{i-1} strings from U_r in the subtree $T_{[r]^{i-1}}$ are located in $T_{[r]^i}$ and $\lceil k_{i-1}/2 \rceil$ in $T_{([r]^i)^*}$ or vice versa, making sure that every vertex on the path from ε to r is balanced. All remaining vertices are balanced due to (c).

Since the first value of the sequence, which is $k_0 = k$, is assigned to the root due to (a), we know that the root has label $\lambda_r(\varepsilon) = k$, and as $k \leq 2^n$, all leaves are either labeled with a 0 or with a 1. Hence, the generated labeling corresponds to a balanced set U_r of size k. □

Lemma 5.59. *For any $n, k \in \mathbb{N}^{\geq 1}$, where $1 \leq k \leq 2^n$, and any string $r \in T_\varepsilon$, the balanced set U_r of size k corresponding to r contains the string r.*

Proof. Consider the unique sequence K of length $n+1$ corresponding to k with $k_i \geq 1$, for $0 \leq i \leq n$. As $k \leq 2^n$, the last value k_n must be exactly 1, and according to our construction of the labeling λ_r, this value k_n is assigned to the leaf r. Due to the construction of the balanced set U_r corresponding to λ_r, the set U_r must contain r. □

As before, let $\mathcal{C}_{[s]^i}$ be the class of balanced sets $U_{s'}$ corresponding to a string s' from the subtree $T_{[s]^i}$, and let $\mathcal{C} := \mathcal{C}_\varepsilon$.

Now let us assume towards contradiction that, for some arbitrary but fixed $k = 2^m \cdot \ell$ with ℓ being odd, there is an algorithm $\mathcal{A}$ that is optimal if the adversary chooses an instance set of size k containing strings of length n, and that reads less than $n - m$ advice bits. We will show that then there also exists an algorithm $\mathcal{B}$ that is optimal if the adversary chooses an instance set of size ℓ containing strings of length $n - m$, while reading less than $n - m$ advice bits, which is clearly a contradiction to Theorem 5.56. We conclude that there is no k with $1 \leq k \leq 2^n$ for that such an algorithm $\mathcal{A}$ as described above exists.

In what follows, it will often be important to tell instance sets, advice strings etc. of $\mathcal{A}$ and $\mathcal{B}$ apart from one another. To this end, whenever we have to do so, we will add a superscript (n) or $(n - m)$ to the corresponding variables, the former for the algorithm $\mathcal{A}$, the latter for $\mathcal{B}$.

Let us assume that the adversary chooses the set $U_r^{(n-m)}$ corresponding to the string $r = r_1 \ldots r_{n-m}$ as the set $\mathcal{I}^{(n-m)}$ of possible input strings for $\mathcal{B}$. Moreover, let $\hat{r}$ be the string that we get from r by padding r with m zeros in the front, i. e., $\hat{r} := \underbrace{0 \ldots 0}_{m} r_1 \ldots r_{n-m}$.

As $\mathcal{A}$ is optimal, it is optimal in particular if the adversary chooses the set $U_{\hat{r}}^{(n)} \in \mathcal{C}^{(n)}$ corresponding to the string $\hat{r}$ as the set $\mathcal{I}^{(n)}$ of possible input instances and the string $\hat{r}$ is then drawn from $\mathcal{I}^{(n)}$ as the actual input string for $\mathcal{A}$. Note that it is indeed possible to draw $\hat{r}$ from $\mathcal{I}^{(n)}$ since $\hat{r} \in U_{\hat{r}}^{(n)}$ due to Lemma 5.59.

Hence, let us assume that $I_\mathcal{B} = (n-m, r_1, \ldots, r_{n-m})$ is the input sequence for $\mathcal{B}$. During its computation on $I_\mathcal{B}$, algorithm $\mathcal{B}$ simulates the computation of $\mathcal{A}$ on the input $I_\mathcal{A} = (n, \hat{r}_1, \ldots, \hat{r}_n) = (n, 0, \ldots, 0, r_1, \ldots, r_{n-m})$. Note that $\mathcal{B}$ generates this input sequence for $\mathcal{A}$ during its computation from the requests from $I_\mathcal{B}$.

Since $\mathcal{B}$ and $\mathcal{A}$ have a different number of rounds and several rounds of $\mathcal{A}$ are simulated during one round of $\mathcal{B}$, let us call the rounds of $\mathcal{A}$ time steps for now. In the first round, when $\mathcal{B}$ gets a request to guess the first bit of r, it simulates the first $m + 1$ time steps of $\mathcal{A}$. Hence, in its first round, it sends $m + 1$ requests $n, 0, \ldots, 0$ in $m + 1$ subsequent time steps to $\mathcal{A}$, one after another. During its first $m + 1$ time steps, $\mathcal{A}$ demands $d_1^{(n)}, \ldots, d_{m+1}^{(n)}$ advice bits in total. When $\mathcal{B}$ receives these demands, it passes them on to the oracle, and the bits that $\mathcal{B}$ receives from the oracle in return are passed on to $\mathcal{A}$ again. Hence, in its first round, $\mathcal{B}$ demands

$d_1^{(n-m)} = d_1^{(n)} + \ldots + d_{m+1}^{(n)}$ advice bits from the oracle in total. Whatever output $\mathcal{A}$ produces in the first $m-1$ time steps is completely ignored by $\mathcal{B}$, but the output $g_{m+1}^{(n)}$ that $\mathcal{A}$ generates in time step $m+1$ is adopted by $\mathcal{B}$ as its output for round 1; hence, $g_1^{(n-m)} \coloneqq g_{m+1}^{(n)}$.

In each following round, $\mathcal{B}$ simulates exactly one further time step of $\mathcal{A}$. In every round i with $2 \leq i \leq n-m$, the adversary sends a request $x_i^{(n-m)} = r_{i-1}$ to $\mathcal{B}$. Then $\mathcal{B}$ passes this request on to $\mathcal{A}$ as its request $x_{m+i}^{(n)}$, whereupon $\mathcal{A}$ demands $d_{m+i}^{(n)}$ advice bits in time step $m+i$ in total, bit by bit. Again, whenever $\mathcal{B}$ receives such a demand, it passes it on to the oracle, and all advice bits the oracle sends to $\mathcal{B}$ as a response are passed on by $\mathcal{B}$ to $\mathcal{A}$, such that $\hat{\tau}_{m+i}^{(n)} \coloneqq \hat{\tau}_i^{(n-m)}$.

The output that $\mathcal{A}$ generates in time step $m+i$ is adapted by $\mathcal{B}$ as its own for round i; hence, $g_i^{(n-m)} \coloneqq g_{m+i}^{(n)}$.

Observation 5.60. *Suppose the adversary chooses a set $U_{\hat{r}}^{(n)}$ for some string $\hat{r} \in T^{(n)}$ as $\mathcal{I}^{(n)}$ and then $\hat{r}$ is drawn as input string for $\mathcal{A}$. Then, $\mathcal{A}$ is always optimal in the first m time steps, no matter which bits it guesses.*

Proof. Due to Observation 5.57, each value k_i with $1 \leq i \leq m$ in the sequence $K = (k_0, k_1, \ldots, k_n)$ is $k_{i-1}/2$. Hence, in each labeling corresponding to a string $\hat{r}$, each vertex on level i is labeled with $k_{i-1}/2$. This directly implies that, for $1 \leq i \leq m$, both the left and the right subtree of each vertex v on level $i-1$ contain $k_{i-1}/2$ vertices from the corresponding set $U_{\hat{r}}^{(n)}$.

Therefore, in the first m time steps, the guesses of $\mathcal{A}$ do not matter at all; each decision is optimal. □

From the reduction given above, we gain the following statement about the number of advice bits necessary to be optimal in the general case of input instances of arbitrary size k.

Theorem 5.61. *In the dialog model, any Ψ-optimal online algorithm needs at least $n-m$ advice bits on a set of instances of size $k = 2^n - \tilde{k} = 2^m \cdot \ell$ containing strings of length n, where ℓ is odd and $1 \leq k \leq 2^n - 1$. Hence, any algorithm needs to read at least $n-m$ advice bits to guess*

$$\frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}}$$

bits correctly in expectation.

Proof. For the sake of contradiction, assume there is an algorithm $\mathcal{A}$ that reads less than $n-m$ advice bits and that is optimal if the adversary chooses a set of $k = 2^m \cdot \ell$ strings of length n, for some $k, \ell, m, n \in \mathbb{N}^{\geq 1}$ with $k \leq 2^n$ and ℓ being odd. Hence, for each string $\hat{r} \in T^{(n)}$ with $\hat{r}_i = 0$ for every i with $1 \leq i \leq m$, the algorithm $\mathcal{A}$ must be optimal in particular if the adversary chooses the set $U_{\hat{r}}^{(n)}$ as $\mathcal{I}^{(n)}$ and $\hat{r}$ is drawn from $U_{\hat{r}}^{(n)}$ as the actual input string.

Now consider a setting with strings of length $n - m$ and an adversary that chooses an instance set of size ℓ as the set of possible input strings. We construct an algorithm $\mathcal{B}$ for this setting, using $\mathcal{A}$ as described in the reduction given above. No matter which bits $\mathcal{A}$ guesses in the first m time steps, these guesses are optimal according to Observation 5.60. Since every such guess is correct with probability $1/2$, $\mathcal{A}$ guesses $m/2$ bits correctly in expectation in the first m time steps.

As $\mathcal{A}$ is optimal, due to Theorem 5.42, its expected number of correctly guessed bits in all n time steps in total is $n - \hat{\omega}(k)/k$. Thus, the number of bits guessed correctly by $\mathcal{A}$ in time steps $m+1, \ldots, n$ in total is

$$\begin{aligned}
n - \frac{\hat{\omega}(k)}{k} - \frac{m}{2} &= n - \frac{\hat{\omega}(2^m \cdot \ell)}{2^m \cdot \ell} - \frac{m}{2} \\
&= n - \frac{\frac{\hat{\omega}(2^m \cdot \ell)}{2^m}}{\ell} - \frac{m}{2} \\
&= n - \frac{\frac{m}{2} \cdot \ell + \hat{\omega}(\ell)}{\ell} - \frac{m}{2} \qquad (5.36) \\
&= n - \frac{m}{2} - \frac{\hat{\omega}(\ell)}{\ell} - \frac{m}{2} \\
&= (n - m) - \frac{\hat{\omega}(\ell)}{\ell},
\end{aligned}$$

where (5.36) follows from Observation 5.10.

This is obviously also the number of correctly guessed bits of the algorithm $\mathcal{B}$ in rounds $1, \ldots, n$ in total. Therefore, for every $r \in T_{\varepsilon}^{(n-m)}$, the constructed algorithm $\mathcal{B}$ guesses $(n - m) - \hat{\omega}(\ell)/\ell$ bits correctly if the adversary chooses the instance set $U_r^{(n-m)}$ of size ℓ as $\mathcal{I}^{(n-m)}$ and r is drawn from $\mathcal{I}^{(n-m)}$ as the actual input string for $\mathcal{B}$. Moreover, $\mathcal{B}$ reads at most as many advice bits as $\mathcal{A}$, which is less than $n - m$ according to our initial assumption. However, due to Lemma 5.55, such an algorithm $\mathcal{B}$ does not exist. Hence, our initial assumption must have been false and $\mathcal{A}$ cannot exist. □

5.3.3 Comparing Upper and Lower Bound

We have now proven an upper and a lower bound for the bit string guessing problem in the dialog model and a probabilistic setting in which the adversary chooses a set of k strings, from which one is chosen as the actual input string uniformly at random. We have seen that there is a Ψ-optimal algorithm for 2-GUESS reading n advice bits if $1 \leq k \leq 2^n - 1$ and no advice bits at all if $k = 2^n$. As a lower bound, for $0 \leq k \leq 2^n$, we have proven that at least $n - m$ advice bits are necessary to be Ψ-optimal if the number of included strings is $k = 2^m \cdot \ell$. Note that for $k = 2^n = 2^m \cdot \ell$, we have $\ell = 1$ and $n = m$, implying $n - m = 0$, so we also obtain a reasonable lower bound of 0 for $k = 2^n$. Hence, for $k = 2^n$ and odd k, the upper and lower bound are tight. For even $k = 2^m \cdot \ell \geq 1$, the upper and lower bound differ by an additive term of m.

5.4 Comparing Monolog and Dialog Model

Let us briefly emphasize the differences and similarities between the two models we analyzed in the previous sections.

In the dialog model, the number of advice bits necessary and sufficient to be Ψ-optimal are both independent of the cardinality of the set $\mathcal{I}$ of possible input instances, at least for odd values of $k = |\mathcal{I}|$. In constrast to this, in the monolog model, the number of necessary and sufficient advice bits are both in $\Theta\big(\tilde{k}\,(n - \log \tilde{k})\big)$; hence, both depend on $\tilde{k} = 2^n - k$ and thus on k.

For $k = 2^n$ (and thus $\tilde{k} = 0$), in both models, both the upper and lower bound yield 0; in this case, the bounds are therefore tight. For $k = 2^n - 1$ (and thus $\tilde{k} = 1$), the upper and lower bound in the monolog model are $\tilde{k} \cdot (n - \log \tilde{k}) = n$ and $\log \binom{2^n}{\tilde{k}} = \log 2^n = n$, respectively. In the dialog model, they are both n when k is odd. Therefore, in both models, n advice bits are necessary and sufficent to be Ψ-optimal for $k = 1$.

In the dialog model, there is a Ψ-optimal algorithm that uses only n advice bits for every k with $1 \leq k \leq 2^n - 1$. In the monolog model, on the other hand, we proved a lower bound of $\tilde{k} \cdot (n - \log \tilde{k})$ advice bits necessary to be Ψ-optimal. For any $\tilde{k}$ with $1 \leq \tilde{k} \leq 2^n - 1$, we have $\tilde{k} \cdot (n - \log \tilde{k}) \geq n$. Hence, for all these values of $\tilde{k}$, the number of advice bits necessary in the monolog model is at least n, which equals the number of advice bits sufficient to be Ψ-optimal in the dialog model. This impressively demonstrates the power of basing the advice on the random decisions the adversary has made so far.

Furthermore, for some values of k, there is a huge gap between the bounds in the two models; for example, if $\tilde{k} = 2^{n-1}$, then $\tilde{k} \cdot (n - \log \tilde{k}) = 2^{n-1} \cdot (n - (n-1)) = 2^{n-1}$ advice bits are necessary to be Ψ-optimal in the monolog model. This shows an exponential gap to the upper bound of n advice bits that are sufficient to be Ψ-optimal in the dialog model. This proves again that the dialog model is, in fact, much more powerful in terms of how many advice bits the algorithm has to be supplied with to achieve Ψ-optimality.

5.5 Reductions in the Probabilistic Setting

In this section, we want to show how the string guessing problem can be used to prove lower bounds for other online problems in the model with a probabilistic setting. We illustrate this using the example of the set cover problem.

Definition 5.62 (Online Set Cover Problem). *The* online set cover problem, *denoted by* SETCOVER, *is the following online minimization problem. Let* Q *be a* ground set, *and let* $\mathcal{F}$ *be a family of sets with* $F \subseteq Q$, *for all* $F \in \mathcal{F}$. *Without loss of generality, let no set* F *be the subset of another set* $F' \neq F$, *for* $F, F' \in \mathcal{F}$, *and for every* $q \in Q$, *let there be some set in* $\mathcal{F}$ *that contains* q. *The set* Q *and the family* $\mathcal{F}$ *are given as the first request in round* 0. *The algorithm does not have to respond to this request in any way. After that,* n *requests arrive consecutively in* n *subsequent rounds, such that in each round* i *with* $1 \leq i \leq n$, *a request* $q_i \in Q$ *is presented. The total request sequence is thus* $I = ((\mathcal{F}, Q), q_1, \ldots, q_n)$.

An online algorithm $\mathcal{A}$ *solves* SETCOVER *if, immediately after each request* q_i *with* $1 \leq i \leq n$, *it specifies a set* $F_i \in \mathcal{F}$ *such that* $\{q_1, \ldots, q_i\} \subseteq \bigcup_{j=1}^{i} F_j$. *We can assume that its response to the first request is* ε. *Hence, the output of* $\mathcal{A}$ *on the input sequence* I *is a sequence* $\mathcal{A}(I) = (\varepsilon, F_1, \ldots, F_n)$ *with* $F_i \in \mathcal{F}$ *such that after each round* j, *each of the requests* $q_1, \ldots, q_j$ *is contained in at least one of the sets* $F_1, \ldots, F_j$. *The cost* $\text{cost}(\mathcal{A}(I))$ *of a solution* $\mathcal{A}(I)$ *is the number of different sets contained in it, i.e.,* $\text{cost}(\mathcal{A}(I)) = |\{F \mid F \in \mathcal{A}(I)\}|$.

We say that a request q_i is *covered* by some set $F_j \in \mathcal{A}(I)$ if $q_i \in F_j$. Furthermore, we define the family of sets chosen by $\mathcal{A}$ up to round i to be $\mathcal{C}_i := \{F_j \mid 1 \leq j \leq i\}$. Then the family of sets that are contained in $\mathcal{A}(I)$ is $\mathcal{C} := \mathcal{C}_n = \{F_j \mid 1 \leq j \leq n\} = \{F \mid F \in \mathcal{A}(I)\}$. The sets contained in $\mathcal{C}$ are called *covering sets*. The aim of any algorithm for SETCOVER is to cover all the elements from I while using as few different covering sets as possible. Note that, whenever a request q_i arrives that is not covered yet by any of the sets $F_1, \ldots, F_{i-1}$, any algorithm for SETCOVER has to pick a new covering set F_i, and whenever the current request q_i is already covered, the algorithm can pick an arbitrary set $F_j \in \mathcal{C}_{i-1}$ without increasing the size of $\mathcal{C}$.

Whenever we have to distinguish the outputs of different algorithms, let us use the notation $F_i^{\mathcal{A}}$ for the set that a SETCOVER algorithm $\mathcal{A}$ chooses in round i, and the notation $\mathcal{C}_i^{\mathcal{A}}$ for the family of covering sets that have been chosen by $\mathcal{A}$ after round i. Thus we have $\mathcal{C}_i^{\mathcal{A}} := \{F_j^{\mathcal{A}} \mid 1 \leq j \leq i\}$, for $0 \leq i \leq n$, and $\mathcal{C}_n^{\mathcal{A}} =: \mathcal{C}^{\mathcal{A}}$.

Before we give a reduction from the bit string guessing problem to the online set cover problem, let us first make the following observation.

Lemma 5.63. *Consider an online algorithm* $\mathcal{B}$ *for* SETCOVER *that adds a new set to* $\mathcal{C}$ *in some round, although the element requested in this round is already covered. Then there is an online algorithm* $\mathcal{A}$ *for* SETCOVER *that does not have a larger cost than* $\mathcal{B}$ *that only* *adds a set to $\mathcal{C}$ in round i, for all i with $1 \leq i \leq n$, if the current request q_i is not covered yet, and chooses $F_i := F_{i-1}$ otherwise.*

Proof. We determine the output of $\mathcal{A}$ in each round i depending on $\mathcal{B}$'s output in rounds $1, \ldots, i$, starting with round 1 and then traversing all subsequent rounds in ascending order. In the first round, $\mathcal{A}$ outputs the same set as $\mathcal{B}$, i. e., $F_1^{\mathcal{A}} := F_1^{\mathcal{B}}$. For the output $F_i^{\mathcal{A}}$ of $\mathcal{A}$ in a round i, where $1 \leq i \leq n$, we distinguish two cases. If the current request q_i in round i is already contained in a covering set chosen by $\mathcal{A}$ in a preceding round, there is no need to pick a new covering set, so $\mathcal{A}$ chooses $F_i^{\mathcal{A}} := F_{i-1}^{\mathcal{A}}$. If, on the other hand, the request q_i is not covered yet by any set chosen by $\mathcal{A}$ in any of the preceding rounds, then $\mathcal{A}$ has to add a new covering set to $\mathcal{C}_{i-1}^{\mathcal{A}}$. We know that, according to Definition 5.62, at least one set from $\mathcal{C}_i^{\mathcal{B}}$ contains the request q_i. Hence, $\mathcal{A}$ chooses $F_i^{\mathcal{A}} := F_j^{\mathcal{B}}$ for some arbitrary set $F_j^{\mathcal{B}} \in \mathcal{C}_i^{\mathcal{B}}$ containing q_i.

Obviously, in each round i, the set picked by $\mathcal{A}$ is contained in $\mathcal{C}_i^{\mathcal{B}} \subseteq \mathcal{C}^{\mathcal{B}}$, and therefore, $\mathcal{C}^{\mathcal{A}} \subseteq \mathcal{C}^{\mathcal{B}}$. Thus, the cost of $\mathcal{A}$ cannot be larger than the one of $\mathcal{B}$. □

We have now shown that, given an algorithm $\mathcal{B}$ for SETCOVER, we can construct an algorithm $\mathcal{A}$ for SETCOVER whose cost is not larger than the one of $\mathcal{B}$ and that only adds new sets to the family of covering sets if the current request is not covered yet. Note that this property corresponds to the definition of a *lazy* algorithm as defined in Section 2.1). Hence, from now on we can constrain our considerations to lazy SETCOVER algorithms, that only add new sets to $\mathcal{C}$ if they have to.

Recall the notation $[s]^i$ for the prefix of length i of the string $s = s_1 \ldots s_n$, for $0 \leq i \leq n$. To give a reduction from 2-GUESS to SETCOVER, we show that we can use an algorithm $\mathcal{A}$ for SETCOVER to construct an algorithm $\mathcal{B}$ that solves 2-GUESS. To this end, at first we have to transform a given instance $I_{\mathcal{B}} = (n, r_1, \ldots, r_n)$ for 2-GUESS, consisting of the length n of the to-be-guessed string $r = r_1 \ldots r_n$ and n bits r_i revealed in subsequent rounds, into an instance $I_{\mathcal{A}}$ for SETCOVER. Then, we run the algorithm $\mathcal{A}$ for SETCOVER on the instance $I_{\mathcal{A}}$ and transform the output $\mathcal{A}(I_{\mathcal{A}})$ back into an output for 2-GUESS. Note that this transformation is done by the 2-GUESS-algorithm $\mathcal{B}$ in an online fashion.

The transformation we use is the one from Böckenhauer et al. [BHK+14] applied to an alphabet of size 2. As the ground set Q, we choose all possible strings of length at most n, including the empty string ε. For each bit string s of length exactly n, let us define the set F_s, which contains all prefixes of s, including ε and the string s itself. Hence, $F_s := \{[s]^i \mid 0 \leq i \leq n\}$. The set family $\mathcal{F}$ consists of all sets F_s, i. e., $\mathcal{F} := \{F_s \mid s \text{ is a bit string of length } n\}$. Hence, $\mathcal{F}$ consists of 2^n sets, and the ground set Q has a cardinality of $\sum_{j=0}^{n} 2^j = 2^{n+1} - 1$. The pair $(\mathcal{F}, Q)$ is given to $\mathcal{A}$ in round 0 as the first request, followed by $n + 1$ further requests as follows. In each round i with $1 \leq i \leq n + 1$, the request sent to $\mathcal{A}$ is $q_i := [r]^{i-1}$.

The resulting total request sequence is thus $I_{\mathcal{A}} := \big((\mathcal{F}, Q), [r]^0, [r]^1, [r]^2, \ldots, [r]^n\big) = ((\mathcal{F}, Q), \varepsilon, r_1, r_1r_2, \ldots, r_1 \ldots r_n)$.

The algorithm $\mathcal{B}$ simulates $\mathcal{A}$ on this instance $I_{\mathcal{A}}$. In each round i with $1 \leq i \leq n$, when $\mathcal{B}$ is asked to guess the bit r_i, one round of $\mathcal{A}$ is simulated, in which it is asked to cover the string $[r]^{i-1}$. In the last round, which is round $n+1$, when $\mathcal{B}$ does not have to guess another bit but only gets the feedback whether its guess in the previous round was correct, $\mathcal{A}$ is asked to cover the string $[r]^n = r$. During $\mathcal{B}$'s execution, it reads advice bits whenever $\mathcal{A}$ reads advice bits, writing them onto a dummy advice tape as before. Let the generated output of $\mathcal{A}$ be $\mathcal{A}(I_{\mathcal{A}}) := (\varepsilon, F_1, \ldots, F_{n+1})$ with $F_{n+1} = F_r$. In particular, let the output of $\mathcal{A}$ in round i, for $1 \leq i \leq n$, be $F_i = F_s$ for some $F_s \in \mathcal{F}$ with $s = s_1 \ldots s_n$. Then we define the output of $\mathcal{B}$ in round i to be s_i. In round $n+1$, the algorithm $\mathcal{B}$ does not have to output anything.

We now show that the output of any SETCOVER algorithm on the input $I_{\mathcal{A}}$ as constructed above has the following useful property.

Lemma 5.64. *Consider an online algorithm $\mathcal{A}$ for* SETCOVER *and an input sequence $I_{\mathcal{A}} = \big((\mathcal{F}, Q), [r]^0, [r]^1, [r]^2, \ldots, [r]^n\big)$ for $\mathcal{A}$ as constructed above. Let the output of $\mathcal{A}$ on $I_{\mathcal{A}}$ be $\mathcal{A}(I_{\mathcal{A}}) = (\varepsilon, F_1, \ldots, F_{n+1})$. Then the following holds. If, in some round i, the string $[r]^i \notin F_i$, then $[r]^i \notin F_j$ either, for all rounds $j < i$.*

Proof. Towards contradiction, assume that there is a $j < i$ with $[r]^i \in F_j$, although $[r]^i \notin F_i$. The set F_j is the one chosen by $\mathcal{A}$ in round j, in which the request $[r]^{j-1}$ is sent to $\mathcal{A}$. The requests sent to $\mathcal{A}$ in rounds $j+1, \ldots, i$ are $[r]^j, \ldots, [r]^{i-1}$. Since $[r]^i \in F_j$, from the way we constructed the sets $F \in \mathcal{F}$, we know that also $[r]^j, \ldots, [r]^{i-1} \in F_j$. Hence, F_j already contains all requests sent to $\mathcal{A}$ in rounds $j, \ldots, i$, and $\mathcal{A}$ does not have to choose any new sets to cover these requests. Due to Lemma 5.63, we can assume that $\mathcal{A}$ is lazy and only chooses new sets if the current request is not covered. We conclude that $\mathcal{A}$ does not pick any new sets in rounds $j+1, \ldots, i$, but instead chooses $F_i := F_{i-1} := \ldots := F_{j+1} := F_j$. Hence, we have

$$\exists\, j < i\colon [r]^i \in F_j \implies [r]^i \in F_i,$$

which concludes the proof. □

Lemma 5.65. *Consider an input sequence $I_{\mathcal{B}} = (n, r_1, \ldots, r_n)$ for 2-*GUESS *and the input sequence $I_{\mathcal{A}} = ((\mathcal{F}, Q), \varepsilon, r_1, r_1r_2, \ldots, r_1 \ldots r_n)$ for* SETCOVER *constructed from $I_{\mathcal{B}}$ as described above. Furthermore, consider an algorithm $\mathcal{A}$ on the instance $I_{\mathcal{A}}$ and an algorithm $\mathcal{B}$ that we constructed using $\mathcal{A}$ as described above and that we run on $I_{\mathcal{B}}$. Then the following holds. If $\mathcal{B}$ makes an error in round i, for $1 \leq i \leq n$, then $\mathcal{A}$ must add a new set to $\mathcal{C}$ in round $i+1$.*

Proof. Consider a round i in which $\mathcal{B}$ makes an error. Let the output of $\mathcal{A}$ in round i be the set $F_i = F_s$ for some string $s = s_1 \ldots s_n$. Then, $\mathcal{B}$'s output in round i is s_i, and, as this guess is incorrect, $s_i \neq r_i$. Since the set $F_i = F_s$ consists of all prefixes of s and the only string of length i that it contains is $[s]^i$, the set F_i does not contain $[r]^i \neq [s]^i$. Due to Lemma 5.64, we conclude from $[r]^i \notin F_i$ that $[r]^i \notin F_j$, for all $j \leq i$. However, the string $[r]^i$ is the request sent to $\mathcal{A}$ in round $i+1$, and since it is not contained in any of the sets $F_1, \ldots, F_i$, it is not covered yet. Hence, $\mathcal{A}$ must choose a set $F_j \notin \mathcal{C}_i$ with $[r]^i \in F_j$ in round $i+1$ to cover $[r]^i$. □

Corollary 5.66. *Consider an instance $I_\mathcal{B}$ for* 2-GUESS *and an instance $I_\mathcal{A}$ for* SETCOVER *constructed from $I_\mathcal{B}$ as described above. Given an algorithm $\mathcal{A}$ for* SETCOVER *that computes a solution with a cost of $\mathrm{cost}(\mathcal{A}(I_\mathcal{A})) = c$ on $I_\mathcal{A}$, we can construct an algorithm $\mathcal{B}$ for* 2-GUESS *that makes only $c-1$ errors on the instance $I_\mathcal{B}$.*

Proof. In the first round, $\mathcal{A}$ has to pick a new covering set anyway. If $\mathcal{A}$ only picks c covering sets in total, it adds exactly $c-1$ sets in the following rounds $2, \ldots, n+1$, and hence, there are $n-c+1$ rounds $i \geq 2$ in which it does not add any new set to $\mathcal{C}$. Due to Lemma 5.65, if $\mathcal{A}$ does not add a new set in round i, then $\mathcal{B}$ does not make an error in round $i-1$. Thus, $\mathcal{B}$ does not make any errors in $n-c+1$ of the first n rounds (and does not have to make a guess in the last one). Consequently, the total number of errors of $\mathcal{B}$ is $c-1$. □

From this, we can derive the following. Consider a setting with a probabilistic adversary. The adversary specifies a set $\mathcal{I}$ of size $k = 2^n - \tilde{k} \leq 2^n$, with $k = 2^m \cdot \ell$ for some odd number ℓ, of possible input instances for SETCOVER from a set $\mathcal{I}_{\mathrm{all}}$ of all input instances of size $|\mathcal{I}_{\mathrm{all}}| = (2^{n+1}-1)^n$. Then it chooses one instance from $\mathcal{I}$ uniformly at random as the input. In this setting, we can deduce a lower bound on the number of advice bits any SETCOVER algorithm needs to read to achieve a certain expected competitive ratio for both the monolog and the dialog model.

Theorem 5.67. *There is no online algorithm for* SETCOVER *that reads less than $\tilde{k}n - 2\,\hat{\omega}(\tilde{k})$ advice bits in the monolog model and achieves a strict competitive ratio of $1 + \hat{\omega}(2^n - \tilde{k})/(2^n - \tilde{k})$ in expectation.*

Proof. Towards contradiction, assume there is an algorithm $\mathcal{A}$ for SETCOVER that has an expected strict competitive ratio of at most $1 + \hat{\omega}(2^n - \tilde{k})/(2^n - \tilde{k})$, while reading less than $\tilde{k}n - 2\,\hat{\omega}(\tilde{k})$ advice bits. Then we can use this algorithm $\mathcal{A}$ to construct an algorithm $\mathcal{B}$ for 2-GUESS as described in the reduction given above. The resulting algorithm $\mathcal{B}$ reads as many advice bits as $\mathcal{A}$, such that the number of advice bits used by $\mathcal{B}$ is also less than $\tilde{k}n - 2\,\hat{\omega}(\tilde{k})$. Exactly one set is necessary to cover all requests from $I_\mathcal{A}$, namely F_r. Hence, for $\mathcal{A}$ to be able to achieve a strict competitive ratio of $c := 1 + \hat{\omega}(2^n - \tilde{k})/(2^n - \tilde{k})$ as assumed, the family $\mathcal{C}$ of covering sets computed by $\mathcal{A}$ can only have a cardinality of at most c. Therefore, due to Corollary 5.66, the constructed algorithm $\mathcal{B}$ guesses at most $c - 1 = \hat{\omega}(2^n - \tilde{k})/(2^n - \tilde{k})$ bits incorrectly in expectation. The expected number of correctly guessed bits by $\mathcal{B}$ is thus

$$\begin{aligned}
\mathbb{E}[\mathrm{gain}_{\mathcal{B}}] &\geq n - \frac{\hat{\omega}(2^n - \tilde{k})}{2^n - \tilde{k}} \\
&= \frac{n \cdot (2^n - \tilde{k})}{2^n - \tilde{k}} - \frac{\hat{\omega}(2^n) - \tilde{k} \cdot n + \hat{\omega}(\tilde{k})}{2^n - \tilde{k}} && (5.37) \\
&= \frac{n \cdot 2^n - n \cdot \tilde{k} - n \cdot 2^{n-1} + \tilde{k} \cdot n - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}} && (5.38) \\
&= \frac{n \cdot 2^{n-1} - \hat{\omega}(\tilde{k})}{2^n - \tilde{k}},
\end{aligned}$$

where (5.37) follows from Observation 5.8 and (5.38) from Observation 5.4.

This is exactly the expected number of correctly guessed bits of any Ψ-optimal algorithm. However, from Theorem 5.32, we know that any Ψ-optimal algorithm (and therefore also $\mathcal{B}$) needs at least $\tilde{k} \cdot n - 2\,\hat{\omega}(\tilde{k})$ advice bits in the monolog model, which leads to a contradiction. Hence, there cannot be an algorithm for SETCOVER that reads less than $\tilde{k}n - 2\,\hat{\omega}(\tilde{k})$ advice bits and achieves a strict competitive ratio of at most $1 + \hat{\omega}(2^n - \tilde{k})/(2^n - \tilde{k})$ in expectation. □

For the same setting of $k = 2^m \cdot \ell$ possible input instances for SETCOVER that are chosen by the adversary, from which one is chosen uniformly at random as the actual input instance, we can derive the following result for the dialog model.

Theorem 5.68. *There is no algorithm for* SETCOVER *that reads less than* $n - m$ *advice bits in the dialog model and achieves a strict competitive ratio of* $1 + \hat{\omega}(2^n - \tilde{k})/(2^n - \tilde{k})$ *in expectation.*

Proof. This proof is completely analogous to the one of Theorem 5.67. Here, we assume towards contradiction that there is a Ψ-optimal algorithm for SETCOVER reading less than $n - m$ advice bits. With the same argumentations and calculations as in the proof of Theorem 5.67, we obtain a contradiction to Theorem 5.61, which states that each optimal algorithm for 2-GUESS needs at least $n - m$ advice bits in the dialog model. □

The above two theorems demonstrate how lower bounds for 2-GUESS in a probabilistic setting carry over to lower bounds for SETCOVER. It seems promising to explore this technique for other online problems as well, allowing to reason about to which extent information on the future may help an online algorithm in such probabilistic settings.

6

Conclusion

In this dissertation, we investigated the information content and the advice complexity of some selected online problems. To this end, we excessively made use of the very generic bit string guessing problem and the fact that all the examined online problems can somehow be interpreted as the problem of guessing a binary string bit by bit.

For the bit string guessing problem, there already exist lower and upper bounds for the number of advice bits necessary and sufficient to obtain solutions with a specified number of correctly guessed bits. Due to its generic nature, many online problems—also such that have not been considered in this thesis—can easily be re-interpreted as some kind of string guessing. Consequently, it is possible to devise reductions from the bit string guessing problem to a given online problem, and thus to directly infer lower bounds on the advice complexity for the problem at hand from the already known lower bounds for string guessing. This method has already been used before as a helpful tool for lower-bound proofs. There are also recent approaches in which variants of the bit string guessing problem are analyzed and the obtained bounds are transfered to other online problems by giving reductions from these altered problems; this way, bounds for the online bin packing problem [BKLL14b] and a certain class of online graph problems like, for example, independent set and vertex cover [BFKM15] are obtained.

In this thesis, we applied the reduction technique to the 2-server problem on a finite path. Furthermore, we demonstrated that this method can also easily be applied to the disjoint path allocation problem on a path, and we obtained a linear lower bound on the necessary advice bits to achieve competitive ratios $c < 4/3$. On the other hand, we observed that this reduction technique does have its limitations, since it is often possible to obtain better bounds using an approach customized for the problem at hand. We witnessed this in the context of the disjoint path allocation problem, for which we took a different approach not involving a string guessing reduction that yielded a lower bound on the advice complexity covering a much wider range of competitive ratios than only up to $c < 4/3$; namely, not only constant ones, but up to logarithmic in the path length. Furthermore, we investigated two related online problems, the graph searching and the graph exploration problem, which are both online problems with the unusual property that the input sequence given to the algorithm depends on the previous actions of the online algorithm. Since the usual reduction technique is not applicable for this kind of online problems, we demonstrated how this method can be adapted, and applied it to the graph exploration problem. Finally, we introduced a new probabilistic model in which the oracle is less powerful, making it reflect real-world environments more accurately. In this model, we analyzed the bit string guessing problem thoroughly. We investigated two different ways to model the oracle and gave lower and upper bounds on the advice complexity of any online algorithm that achieves the best possible expected number of correctly guessed bits.

Let us give a short overview of possible future research in the areas investigated in this thesis. We only analyzed the k-server problem in a very restricted setting; with only two servers and only on paths. Expanding the ideas presented here to other topologies and to $k \geq 3$ servers could be a next step to obtaining more insight into the advice complexity of the k-server problem. Concerning the graph searching and exploration problem, we already mentioned in the introduction of Chapter 4 that many different versions of searching and exploration problems have been considered in the literature. For sure, the topic has been exhaustively investigated in the classical online setting without advice, but it would certainly be interesting to obtain more results in this area concerning the advice complexity of those problems. Furthermore, in the context of the disjoint path allocation problem, we have seen that the attempted reduction from the string guessing problem suffers from the symmetric cost function of the problem. Devising lower bounds for the string guessing problem with more flexible cost functions might help to overcome such problems and thus make the reduction technique even more universally applicable. A first step into this direction has already been taken by Boyar et al. [BFKM15]. We also mentioned in the beginning of Chapter 3 that the disjoint path allocation is a special case of the call admission problem. It might be interesting to examine a more general setting, in which the calls can have different durations, bandwidths, and profits. Moreover, the model of the probabilistic adversary leaves a lot of room for generalizations. In this thesis, we only studied the number of advice bits necessary and sufficient to achieve optimality for the string guessing problem on binary alphabets, when the actual input string is chosen according to a restricted class of probability distributions. As the string guessing problem can also be used in the probabilistic model to prove lower bounds for other online problems, as shown in Section 5.5 for the online set cover problem, it would be desirable to provide lower bounds in this model for the most general version of the string guessing problem. An interesting continuation of the work that was started in this model would thus be to expand it to larger alphabets, a wider class of probability distributions, and to a trade-off between the number of advice bits and the quality of the given online algorithm. In the end, of course, all the results concerning the string guessing problem in the probabilistic model (as well as in the original deterministic one) are designed to be transfered to other online problems by developing reductions.

Bibliography

[AL08] R. Anitha and R. S. Lekshmi. N-sun decomposition of complete, complete bipartite and some harary graphs. *International Journal of Computational and Mathematical Sciences*, 2(1):33–38, 2008. 73

[Bar14] Kfir Barhum. Tight bounds for the advice complexity of the online minimum steiner tree problem. In *Proceedings of the 40th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2014)*, volume 8327 of *Lecture Notes in Computer Science*, pages 77–88. Springer-Verlag, 2014. 11

[BBF$^+$14] Kfir Barhum, Hans-Joachim Böckenhauer, Michal Forišek, Heidi Gebauer, Juraj Hromkovič, Sacha Krug, Jasmin Smula, and Björn Steffen. On the power of advice and randomization for the disjoint path allocation problem. In *Proceedings of the 40th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2014)*, volume 8327 of *Lecture Notes in Computer Science*, pages 89–101. Springer-Verlag, 2014. 11, 38, 39, 42

[BBH$^+$13] Maria Paola Bianchi, Hans-Joachim Böckenhauer, Juraj Hromkovič, Sacha Krug, and Björn Steffen. On the advice complexity of the online $L(2,1)$-coloring problem on paths and cycles. In *Proceedings of the 19th Annual International Conference on Computing and Combinatorics (COCOON 2013)*, volume 7936 of *Lecture Notes in Computer Science*, pages 53–64. Springer-Verlag, 2013. 11

[BBH$^+$14] Maria Paola Bianchi, Hans-Joachim Böckenhauer, Juraj Hromkovič, Sacha Krug, and Björn Steffen. On the advice complexity of the online $L(2,1)$-coloring problem on paths and cycles. *Theoretical Computer Science*, 554:22–39, 2014. 11

[BBHK12] Maria Paola Bianchi, Hans-Joachim Böckenhauer, Juraj Hromkovič, and Lucia Keller. Online coloring of bipartite graphs with and without advice. In *Proceedings of the 18th Annual International Conference on*

Computing and Combinatorics (COCOON 2012), volume 7434 of *Lecture Notes in Computer Science*, pages 519–530, 2012. 11

[BBMN11] Nikhil Bansal, Niv Buchbinder, Aleksander Mądry, and Joseph Naor. A polylogarithmic-competitive algorithm for the k-server problem (extended abstract). In *Proceedings of the 52nd Annual Symposium on Foundations of Computer Science (FOCS 2011)*, pages 267–276. IEEE Computer Society, 2011. 18

[Ber96] Piotr Berman. On-line searching and navigation. In *Online Algorithms, The State of the Art*, volume 1442 of *Lecture Notes in Computer Science*, pages 232–241. Springer-Verlag, 1996. 58

[BEY98] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998. 3, 10, 11, 17, 18, 37

[BFKM15] Joan Boyar, Lene M. Favrholdt, Christian Kudahl, and Jesper W. Mikkelsen. Advice complexity for a class of online problems. In *Proceedings of the 32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, volume 30 of *LIPIcs*, pages 116–129. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. 149, 150

[BHK$^+$13] Hans-Joachim Böckenhauer, Juraj Hromkovič, Dennis Komm, Sacha Krug, Jasmin Smula, and Andreas Sprock. The string guessing problem as a method to prove lower bounds on the advice complexity. In *Proceedings of the 19th Annual International Conference on Computing and Combinatorics (COCOON 2013)*, volume 7936 of *Lecture Notes in Computer Science*, pages 493–505. Springer-Verlag, 2013. 14

[BHK$^+$14] Hans-Joachim Böckenhauer, Juraj Hromkovič, Dennis Komm, Sacha Krug, Jasmin Smula, and Andreas Sprock. The string guessing problem as a method to prove lower bounds on the advice complexity. *Theoretical Computer Science*, 554:95–108, 2014. 11, 14, 27, 36, 105, 124, 145

[BKK$^+$09] Hans-Joachim Böckenhauer, Dennis Komm, Rastislav Královič, Richard Královič, and Tobias Mömke. On the advice complexity of online problems. In *Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC 2009)*, volume 5878 of *Lecture Notes in Computer Science*, pages 331–340. Springer-Verlag, 2009. 11, 38

[BKKK11] Hans-Joachim Böckenhauer, Dennis Komm, Rastislav Královič, and Richard Královič. On the advice complexity of the k-server problem.

In *Proceedings of the 38th International Colloquium on Automata, Languages and Programming (ICALP 2011)*, volume 6755 of *Lecture Notes in Computer Science*, pages 207–218. Springer-Verlag, 2011. 11, 19

[BKKR12] Hans-Joachim Böckenhauer, Dennis Komm, Richard Královič, and Peter Rossmanith. On the advice complexity of the knapsack problem. In *Proceedings of the 10th Latin American Symposium on Theoretical Informatics (LATIN 2012)*, volume 7256 of *Lecture Notes in Computer Science*, pages 61–72. Springer-Verlag, 2012. 11

[BKLL14a] Joan Boyar, Shahin Kamali, Kim S. Larsen, and Alejandro López-Ortiz. On the list update problem with advice. In *Proceedings of the 8th International Conference on Language and Automata Theory and Applications (LATA 2014)*, volume 8370 of *Lecture Notes in Computer Science*, pages 210–221. Springer-Verlag, 2014. 11

[BKLL14b] Joan Boyar, Shahin Kamali, Kim S. Larsen, and Alejandro López-Ortiz. Online bin packing with advice. In *Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, volume 25 of *LIPIcs*, pages 174–186. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014. 11, 149

[BLS99] Andreas Brandstädt, Van B. Le, and Jeremy P. Spinrad. *Graph Classes: A Survey*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999. 73, 74

[BRS97] Avrim Blum, Prabhakar Raghavan, and Baruch Schieber. Navigating in unfamiliar geometric terrain. *SIAM Journal on Computing*, 26(1):110–137, February 1997. 18

[Car00] Neil L. Carothers. *Real analysis*. Cambridge: Cambridge University Press, 2000. 43

[Che14] Wenbin Chen. Settling the randomized k-sever conjecture on some special metrics. http://arxiv.org/abs/1410.4955, 2014. 18

[Chv79] Vašek Chvátal. The tail of the hypergeometric distribution. *Discrete Mathematics*, 25(3):285–287, 1979. 43

[CKPV91] Marek Chrobak, Howard J. Karloff, Thomas H. Payne, and Sundar Vishwanathan. New results on server problems. *SIAM Journal on Discrete Mathematics*, 4(2):172–181, March 1991. 18, 19, 21

[CL91] Marek Chrobak and Lawrence L. Larmore. An optimal on-line algorithm for k-servers on trees. *SIAM Journal on Computing*, 20(1):144–148, 1991. 18

[DHZ12] Reza Dorrigiv, Meng He, and Norbert Zeh. On the advice complexity of buffer management. In *Proceedings of the 23rd International Symposium on Algorithms and Computation (ISAAC 2012)*, volume 7676 of *Lecture Notes in Computer Science*, pages 136–145. Springer-Verlag, 2012. 11

[DKK12] Stefan Dobrev, Rastislav Královič, and Richard Královič. Independent set with advice: The impact of graph knowledge (extended abstract). In *Proceedings of the 10th International Workshop on Approximation and Online Algorithms (WAOA 2012)*, volume 7846 of *Lecture Notes in Computer Science*, pages 2–15. Springer-Verlag, 2012. 11

[DKM12] Stefan Dobrev, Rastislav Královič, and Euripides Markou. Online graph exploration with advice. In *Proceedings of the 19th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2012)*, volume 7355 of *Lecture Notes in Computer Science*, pages 267–278. Springer-Verlag, 2012. 11, 58, 59

[DKP08] Stefan Dobrev, Rastislav Královič, and Dana Pardubská. How much information about the future is needed? In *Proceedings of the 34th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2008)*, volume 4910 of *Lecture Notes in Computer Science*, pages 247–258. Springer-Verlag, 2008. 11, 88

[Doh15] Jérôme Dohrau. Online makespan scheduling with sublinear advice. In *Proceedings of the 41st International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2015)*, volume 8939 of *Lecture Notes in Computer Science*, pages 177–188. Springer-Verlag, 2015. 11

[DP90] Xiaotie Deng and Christos H. Papadimitriou. Exploring an unknown graph (extended abstract). In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science (FOCS 1990)*, volume 1, pages 355–361. IEEE Computer Society, 1990. 58

[EFKR11] Yuval Emek, Pierre Fraigniaud, Amos Korman, and Adi Rosén. Online computation with advice. *Theoretical Computer Science*, 412(24):2642–2656, 2011. 11, 14, 18

[FIP08] Pierre Fraigniaud, David Ilcinkas, and Andrzej Pelc. Tree exploration with advice. *Information and Computation*, 206(11):1276–1287, 2008. 59

[FKL$^+$91] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator, and Neal E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991. 11

[FKS12] Michal Forišek, Lucia Keller, and Monika Steinová. Advice complexity of online coloring for paths. In *Proceedings of the 6th International Conference on Language and Automata Theory and Applications (LATA 2012)*, pages 228–239, 2012. 11

[FRR94] Amos Fiat, Yuval Rabani, and Yiftach Ravid. Competitive k-server algorithms. *Journal of Computer and Systems Sciences*, 48(3):410–428, 1994. 18

[Ful14] Peter Fulla. Advice complexity of online algorithms. Master's thesis, Comenius University in Bratislava, 2014. To appear. 59, 73, 83

[FW12] Klaus-Tycho Foerster and Roger Wattenhofer. Directed graph exploration. In *Proceedings of the 16th International Conference on Principles Of Distributed Systems (OPODIS 2012)*, volume 7702 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, 2012. 58, 60, 61

[GK10] Subir K. Ghosh and Rolf Klein. Online algorithms for searching and exploration in the plane. *Computer Science Review*, 4(4):189–201, 2010. 58

[GKK+15] Heidi Gebauer, Dennis Komm, Rastislav Královič, Richard Královič, and Jasmin Smula. Disjoint path allocation with sublinear advice. In *Proceedings of the 21st Annual International Conference on Computing and Combinatorics (COCOON 2015)*, 2015. To appear. 11

[GKLO13] Sushmita Gupta, Shahin Kamali, and Alejandro López-Ortiz. On advice complexity of the k-server problem under sparse metrics. In *Proceedings of the 20th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2013)*, volume 8179 of *Lecture Notes in Computer Science*, pages 55–67. Springer-Verlag, 2013. 11, 19, 20

[GKP89] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. A Foundation for Computer Science. Addison-Wesley, 1989. 114

[HKK10] Juraj Hromkovič, Rastislav Královič, and Richard Královič. Information complexity of online problems. In *Proceedings of the 35th International Symposium on Mathematical Foundations of Computer Science (MFCS 2010)*, volume 6281 of *Lecture Notes in Computer Science*, pages 24–36. Springer-Verlag, 2010. 11, 87

[HMSW07] Juraj Hromkovič, Tobias Mömke, Kathleen Steinhöfel, and Peter Widmayer. Job shop scheduling with unit length tasks: bounds and algorithms. *Algorithmic Operations Research*, 2(1):1–14, 2007. 85

[Hro05] Juraj Hromkovič. *Design and Analysis of Randomized Algorithms: Introduction to Design Paradigms*. Springer-Verlag, 2005. 7

[KK11] Dennis Komm and Richard Královič. Advice complexity and barely random algorithms. In *Proceedings of the 37th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2011)*, volume 6543 of *Lecture Notes in Computer Science*, pages 332–343. Springer-Verlag, 2011. 11

[KKM12] Dennis Komm, Richard Královič, and Tobias Mömke. On the advice complexity of the set cover problem. In *Proceedings of the 7th Symposium on Computer Science in Russia (CSR 2012)*, volume 7353 of *Lecture Notes in Computer Science*, pages 241–252. Springer-Verlag, 2012. 11

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 3rd edition, 1997. 125

[Kom12] Dennis Komm. *Advice and Randomization in Online Computation*. PhD thesis, ETH Zurich, 2012. 3, 13, 39

[Kou09] Elias Koutsoupias. The k-server problem. *Computer Science Review*, 3(2):105–118, 2009. 17

[KP94] Bala Kalyanasundaram and Kirk R. Pruhs. Constructing competitive tours from local information. *Theoretical Computer Science*, 130(1):125–138, August 1994. 58, 60

[KP95] Elias Koutsoupias and Christos H. Papadimitriou. On the k-server conjecture. *Journal of the ACM*, 42(5):971–983, September 1995. 18

[Kru15] Sacha Krug. Towards using the history in online computation with advice. *RAIRO Theoretical Informatics and Applications*, 2015. To appear. 14

[LT79] Richard J. Lipton and Robert E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979. 67

[MMO09] Shuichi Miyazaki, Naoyuki Morimoto, and Yasuo Okabe. The online graph exploration problem on restricted graphs. *IEICE Transactions*, 92-D(9):1620–1627, 2009. 58

[MMS88] Mark S. Manasse, Lyle A. McGeoch, and Daniel D. Sleator. Competitive algorithms for on-line problems. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC 1988)*, pages 322–333, 1988. 17, 18

[MMS90] Mark S. Manasse, Lyle A. McGeoch, and Daniel D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11(2):208–230, 1990. 21

[MMS11] Nicole Megow, Kurt Mehlhorn, and Pascal Schweitzer. Online graph exploration: New results on old and new algorithms. In *Proceedings of the 40th International Colloquium on Automata, Languages and Programming (ICALP 2013)*, volume 6756 of *Lecture Notes in Computer Science*, pages 478–489. Springer-Verlag, 2011. 58

[NS07] Nicolas Nisse and David Soguet. Graph searching with advice. In *Proceedings of the 14th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2007)*, volume 4474 of *Lecture Notes in Computer Science*, pages 51–65. Springer-Verlag, 2007. 59

[PY91] Christos H. Papadimitriou and Mihalis Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84(1):127–150, July 1991. 58

[Ric07] John A. Rice. *Mathematical Statistics and Data Analysis*. Duxbury Press, Belmont, CA, 3rd edition, 2007. 43

[Rot06] Ron Roth. *Introduction to Coding Theory*. Cambridge University Press, New York, NY, USA, 2006. 5

[RR11] Marc P. Renault and Adi Rosén. On online algorithms with advice for the k-server problem. In *Proceedings of the 9th International Workshop on Approximation and Online Algorithms (WAOA 2011)*, volume 7164 of *Lecture Notes in Computer Science*, pages 198–210. Springer-Verlag, 2011. 11, 19

[SSU13] Sebastian Seibert, Andreas Sprock, and Walter Unger. Advice complexity of the online coloring problem. In *Proceedings of the 8th International Conference on Algorithms and Complexity (CIAC 2013)*, volume 7878 of *Lecture Notes in Computer Science*, pages 345–357. Springer-Verlag, 2013. 11

[ST85] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985. 10

[Wal01] Walter D. Wallis. *Magic Graphs*. Birkhäuser Verlag, 2001. 73

[Weh14] David Wehner. A new concept in advice complexity of job shop scheduling. In *Proceedings of the 9th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2014)*, volume 8934 of *Lecture Notes in Computer Science*, pages 147–158. Springer-Verlag, 2014. 85, 86

[Weh15] David Wehner. Advice complexity of fine grained job shop scheduling. In *Proceedings of the 9th International Conference on Algorithms and Complexity (CIAC 2015)*, 2015. To appear. 11