Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Selected Topics from Advice Complexity

Diploma thesis

2014 **Michal Petrucha**

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Selected Topics from Advice Complexity

Diploma thesis

Study program:	Computer Science
Field of study:	2508 Computer Science
Department:	Department of Computer Science
Supervisor:	RNDr. Michal Forišek, PhD.

Bratislava, 2014 **Michal Petrucha**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Michal Petrucha
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický

Názov: Selected topics from advice complexity
Vybrané témy z oblasti poradnej zložitosti

Cieľ: Táto diplomová práca má dva hlavné ciele. Prvým je príprava zrozumiteľného prehľadu aktuálnych výsledkov z oblasti poradnej zložitosti (advice complexity) s dôrazom na nájdenie opakujúcich sa techník riešenia problémov. Druhým cieľom je následné skúmanie vybraných online optimalizačných problémov z hľadiska poradnej zložitosti. Očakávané výsledky by mali obsahovať dôkazy dolných a horných odhadov množstva dodatočnej informácie potrebného na optimálne vyriešenie daného online problému, ako aj skúmanie vzťahu medzi množstvom dostupnej dodatočnej informácie a kvalitou riešenia, ktorú vieme zaručiť. Ako možné ďalšie ciele odporúčame napríklad uvažovanie o online algoritmoch využívajúcich aj náhodné čísla, či skúmanie problémov ktoré sú len čiastočne online.

Vedúci: RNDr. Michal Forišek, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 28.11.2012

Dátum schválenia: 28.11.2012 prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Michal Petrucha
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	9.2.1. Computer Science, Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English

Title: Selected topics from advice complexity

Aim: This thesis has two main goals. The first goal is to create a comprehensive overview of the recent results in the field of advice complexity, with a focus on patterns in those results. The second goal is a theoretical analysis of selected online optimization problems from the point of view of advice complexity. Possible results include determining lower and upper bounds on the amount of advice necessary to solve an instance optimally and analyzing the trade-off between the amount of advice available and the competitive ratio we can guarantee. Optionally, it is also possible to consider randomized online algorithms and/or analyze advice complexity for problems that are only partially online.

Supervisor:	RNDr. Michal Forišek, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	doc. RNDr. Daniel Olejár, PhD.
Assigned:	28.11.2012
Approved:	28.11.2012

prof. RNDr. Branislav Rovan, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Abstrakt

Táto práca sa venuje výpočtovým modelom s radou, pomerne novému modelu určenému na meranie obtiažnosti online problémov. Najprv poskytujeme prehľad obvyklých metód používaných na analýzu poradnej zložitosti online problémov. V ďalšej časti ukazujeme nový horný aj dolný odhad na poradnú zložitosť problému alokácie disjunktných ciest pre takmer optimálne riešenia. V poslednej časti práce zavádzame nový model offline výpočtov s radou a poskytujeme základy pre ďalší výskum v tejto oblasti.

Kľúčové slová: online problém, poradná zložitosť, kompetitívna analýza, alokácia disjunktných ciest, subset sum

Abstract

This work focuses on computation with advice, a relatively new model for measuring the complexity of online problems. First, we give an overview of common methods of analysis of the advice complexity of online problems. Then, we show new upper and lower bounds on the advice complexity of near-optimal solutions of the disjoint path allocation problem. Finally, we introduce a model of offline computation with advice and lay out a basic framework for future research in this area.

Key words: online problem, advice complexity, competitive analysis, disjoint path allocation, subset sum

Acknowledgements

First of all, I want to thank my supervisor, Michal Forišek, for his guidance and patience. I also want to thank Alena Bachratá for helpful conversations which helped me move forward when I was stuck. Most of all, however, I want to thank Ivana Kellyérová for her enormous support during the last few weeks.

Contents

Introduction

Online problems often appear in real-world application both inside and outside of the field of computer science. Until recently, the standard way of estimating the difficulty of online problems was competitive analysis.

A new model for analyzing online problems has been introduced in the last decade, which measures the amount of additional information required for an online algorithm to perform better. This model is referred to as the model of advice complexity. This is the model this thesis focuses on.

In the first chapter we give an informal introduction to the world of online computation and explain the concept of online computation with advice.

The second chapter summarizes techniques which make it possible to analyze online problems from the point of view of advice complexity in a standard way. We also demonstrate how these techniques have been applied in recent results for multiple online problems.

In the third chapter we focus on the problem of disjoint path allocation. We summarize latest findings concerning this problem and complement them with our own improvements.

Finally, in the last chapter we extend the model of computation with advice to offline problems. We define a new formal computational model of Turing machines with advice and offer initial crude results about this model as a basis for future research in this area.

Chapter 1

Prerequisites

This chapter introduces the basic concepts studied in this thesis. After explaining the motivation behind these concepts, we proceed to define them in a formal manner.

1.1 Online Problems

One of the countless ways to categorize algorithmic problems is into *offline* and *online problems*. Offline problems are those where the algorighm can access the whole input instance before yielding the output. On the other hand, the instance of an online problem is revealed to the algorithm in smaller pieces and after each piece a partial solution has to be produced. This partial solution cannot be changed later.

A slightly different way of looking at online algorithms is that the algorithm waits for an input query, processes it and outputs an answer to this query immediately. Then it waits for another query and repeats the process until there is nothing more to do.

Solving a problem online is obviously more difficult than solving the same instance knowing the whole input at once. For many problems it is even impossible to compute the optimal partial solutions without the knowledge of the rest of the input sequence. Therefore we define a *competitive ratio* of

an algorithm, which is the quotient of the cost of the solution produced by the online algorithm and the cost of the optimal solution. An optimal solution is one produced by an optimal offline algorithm. Since the competitive ratio can depend on the input instance, we study the worst-case competitive ratio an algorithm achieves.

We may consider randomized online algorithms as well. In this case we examine the expected competitive ratio.

Let us describe a few examples of simple online problems to give a better idea of what they are about. A very simple online problem is ski rental. Suppose we are going to take an unknown number of ski trips and we do not own a pair of skis. Renting a pair of skis for a single trip costs 1, buying one costs s. The input consists of a sequence of queries "take a ski trip" and after each query an answer is expected that is either "rent", "buy" or "use skis already bought". In [Kar92] it is proved that to minimize the competitive ratio the algorithm needs to rent for the first $s-1$ rounds and then buy a pair of skis; this way, the competitive ratio is $\frac{2s-1}{s} \approx 2$.

Another classic online problem is the paging problem. Assume a two-level memory divided into uniform, fixed-size pages. Let k be the number of pages that can fit within the fast memory. The input consists of n queries, each specifying a page we want to access. This page needs to be loaded into the fast memory thus replacing a page called a victim (unless it is there already). The goal is to minimize the number of page faults, i.e. the number of times we need to load a page from the slow memory into the fast level.

In [ST85] the authors show that for any deterministic online algorithm solving the paging problem it is possible to construct an instance using $k+1$ pages where the online algorithm will produce a page fault on each request by always choosing the page that is not in the fast memory. However, an offline algorithm can decrease the number of page faults by at least a factor of k, therefore the competitive ratio of any online paging algorithm is at least k.

In addition, in [ACN96] the authors describe a randomized online algo-

rithm for the paging problem whose competitive ratio is H_k.

1.2 Advice Complexity

In the previous section we showed that there are problems which cannot be solved optimally by a deterministic online algorithm. This means that having access to the whole of the input sequence can help the algorithm to provide better partial results. However, sometimes it may not be necessary to access the whole input sequence in order to compute the optimal solution, in some cases a significantly smaller amount of information is required.

That is why a computational model of *online algorithms with advice* has been introduced in [DKP08]. In this model, the online algorithm is assisted by an oracle with access to the entire input sequence. The oracle has unlimited computational power and provides the online algorithm with information about the input sequence that it requires. We define the *advice complexity* of an online algorithm as the minimal number of bits it needs to read from the oracle in order to solve the problem optimally. The advice complexity of an online problem is then defined as the lowest advice complexity of online algorithms solving it.

There have been multiple formal definitions of this model with various drawbacks. [DKP08] contains a definition in which the online algorithm has access to a finite binary advice tape. That means, however, that additional information can be encoded into the length of the advice tape. In [Eme+09] the authors define a slightly different model where the online algorithm receives the same amount of information in each round. This makes it impossible to use a sublinear amount of advice.

The model used in this thesis has been defined in [Böc+09]; this model uses an infinite advice tape and we measure the number of bits the algorithm accesses. The following sequence of events can therefore be imagined: before we start feeding an online algorithm A with the input, first we give the entire input instance to an oracle which produces a binary string φ. This binary

string is then written at the beginning of an infinite advice tape which can be accessed by A throughout the whole computation.

This model of algorithms with advice suggests a similarity with the model of randomized algorithms. Common definitions of randomized algorithms use a tape filled with random characters from a certain alphabet, often simply with random bits. Our model of algorithms with advice can therefore be looked at as a special case of randomized algorithms, in which the oracle fills the random tape with the string which leads to the best outcome.

To demonstrate the power of advice, we show the amount of advice required to solve the two aforementioned online problems optimally. The ski rental problem is trivial to solve using a single bit of advice – this bit tells the algorithm whether there will be at least s queries. The online algorithm reads this before answering the first query and it knows immediately whether to buy a pair of skis or just rent them on each trip.

The paging problem is slightly more complex to solve optimally using advice. Following the proof in [DKP09], this can be done using n bits of advice. The oracle calculates one optimal solution to the input instance and assigns a single bit to each request. This bit indicates whether the page will be accessed again before it is replaced by another one in the optimal solution, such pages are called active; if the page will not be accessed again, it is passive. The online algorithm then just picks a passive page as the victim on each page fault.

Thus far we only covered the amount of advice required to obtain the optimal solution using an online algorithm. However, it is also useful to examine the amount of advice required to achieve a certain competitive ratio and the tradeoff between these two. In this thesis we will study this aspect as well.

Another possible area of research is the amount of advice required to solve a *partially online problem*. This is a special case of an online problem where only a part of the input instance is served in pieces and at some point the whole rest of the input is served in a single piece.

Taking the previous notion one step further, it also makes sense to apply the concept of advice to offline problems. In that case, we no longer study the competitive ratio. Instead, we can use advice to help an algorithm achieve better efficiency, mainly in terms of its time complexity, especially for known hard problems, such as NP-complete problems. This direction of research is explored further in the last chapter of this thesis.

1.3 Adversaries

When proving lower bounds on the competitiveness of an online problem, it is often useful to model instances on which an online algorithm computes the worst solution. The concept of an *adversary*, denoted by Adv, does precisely that.

A computation of an online algorithm can be thought of as a game in which there are two players: the online algorithm, trying to compute the best solution possible, and an adversary which tries to coerce the algorithm into making as bad decisions as possible by using information about the decisions of the algorithm to construct an instance that is as difficult for the algorithm to solve as possible.

For deterministic online algorithms, informally, the two entities take turns – the adversary submits the first part of the input and the online algorithm provides its first result. Then, the adversary can decide how best to construct the next part of the input instance in order to keep the cost of the solution as far from the optimum as possible.

More formally, we define Adv as an offline algorithm with knowledge of how an algorithm A works in the sense that Adv is able to simulate A, making it possible to anticipate every reaction A makes. The output of Adv is then an instance which is used as the input for A.

If we can show that given an online problem $\mathcal{P}$, there is an adversary Adv such that for every algorithm A, Adv is able to construct an instance for which A fails to be c-competitive, that means there is no c-competitive

algorithm for $\mathcal{P}$.

For randomized online algorithms, there are multiple definitions of adversaries [Ben+94]: the oblivious adversary, the adaptive online adversary and the adaptive offline adversary. The oblivious adversary works in the same way as described for offline algorithms – it can only simulate A without any information about the random data based on which A may make decisions. In the adaptive online model, A and Adv play the game described earlier and Adv creates the input for A in an online fashion. In other words, Adv knows the results of the previous decisions of A when constructing the next piece of input. Finally, the offline adaptive adversary is omniscient – it has full information about the source of randomness based on which A makes its decisions.

When dealing with algorithms with advice, we need to consider whether to allow an adversary to access the advice or not. In this thesis, we follow the model from [Kom12], which gives Adv full information about the advice string corresponding to an instance it creates.

The rationale is that we usually show the existence of an adversary for online algorithms using at most $b(n)$ bits of advice as a way of proving a lower bound of $b(n)$ bits on the advice complexity. This can be done by showing that for each pair (A, O), where A is an online algorithm and O is an oracle which computes the advice string for A, there is an adversary Adv which forces A to fail some criterion, e.g. optimality, or competitiveness.

We can thus assume when constructing Adv that the advice does not exceed $b(n)$ bits. Since we do not impose any restrictions on the computational power of A, Adv, or O, Adv can easily simulate the algorithm A it is working against with all of the $2^{b(n)}$ possible advice strings and find out which one leads to the best outcome. We can then simply assume that Adv knows which advice string is the best one for a given instance.

The previous idea suggests a slightly different approach. By choosing a fixed advice string φ, an online algorithm becomes fully deterministic. Thus an algorithm with b bits of advice can be viewed as a collection of 2^b

deterministic algorithms. Showing that for any collection of 2^b deterministic algorithms, there is an adversary which forces each of them to compute a bad output is therefore equivalent to showing that for each algorithm with b bits of advice, there is such an adversary.

1.4 Formal Definitions and Notations

Having described the basic concepts in informal terms, let us now proceed to formally define the model we are working with.

Definition 1.1 (Online Algorithm)**.** *Let $I = (x_1, \ldots, x_n)$ be an input sequence of an online problem. An* online algorithm *A computes the output sequence $A(I) = (y_1, \ldots, y_n)$ such that $y_i = f(x_1, \ldots, x_i)$ for some function f. We denote the cost of the solution computed by A as $C(A(I))$.*

An optimal solution for I will be denoted by $Opt(I)$. By optimal solution we mean one which can be computed by an offline algorithm with unbounded computational power, such that, in the case of maximization problems, it maximizes the cost. We will use $E[X]$ to denote the expected value of a random variable X.

Definition 1.2 (Competitive Ratio)**.** *Consider an optimization problem in which the goal is to maximize the cost of a solution. An algorithm A is c-competitive if there is a constant α such that for each instance I we have $C(A(I)) \geq C(Opt(I))/c - \alpha$. If $\alpha = 0$, we say that A is strictly c-competitive. The* competitive ratio *of A is the smallest c such that A is c-competitive.*

For minimization problems, competitiveness is defined analogously, only the inequality changes to $C(A(I)) \leq c \cdot C(Opt(I)) + \alpha$.

The previous definition can easily be extended to randomized algorithms. For each instance I we require $E[C(A(I))] \geq C(Opt(I))/c - \alpha$. We say that the expected competitive ratio of A is the smallest value of c satisfying the above inequality.

We shall now extend the above definitions to include advice.

Definition 1.3 (Online Algorithm with Advice)**.** *Consider an input sequence* $I = (x_1, \ldots, x_n)$ *and an infinite binary string* φ*. An* online algorithm A with advice *computes the sequence* $A^{\varphi}(I) = (y_1, \ldots, y_n)$ *if* $y_i = f(\varphi, x_1, \ldots, x_i)$*. We call* φ *the* advice string*.*

As stated earlier, the computation of A can be interpreted as a series of turns, where in the i-th turn the algorithm reads x_i and yields y_i using all the information read so far and possibly some additional bits from the advice string φ. It is worth noting that the definition does not restrict the computational power of A.

Definition 1.4 (Advice Complexity)**.** *The* advice complexity *of an algorithm* A *is a function* s *such that* $s(n)$ *is the smallest value such that for each input sequence of size* n *there is an advice string* φ *such that the algorithm* A *examines at most the first* $s(n)$ *bits of* φ*. The advice complexity of an online problem is the smallest advice complexity of an online algorithm which computes an optimal solution for each instance.*

Definition 1.5. *An online algorithm with advice* A *is* c*-competitive if there is a constant* α *such that for every* $n \in \mathbb{N}$ *and for every instance* I *of size at most* n *there is an advice string* φ *for which* $C(A^{\varphi}(I)) \geq C(Opt(I))/c - \alpha$ *holds.*

Throughout this thesis we use $\log x$ to denote the binary logarithm of x.

Chapter 2

Known Results and Related Work

The aim of this chapter is to provide an overview of techniques commonly used for analysis of the advice complexity of online problems. After describing the most successful techniques, we demonstrate some of their applications on the online versions of graph coloring and the problem of finding the maximum clique in a graph.

2.1 Common Analysis and Proof Techniques

Despite the fact that the computational model of online algorithms with advice has been only concieved a few years ago it is already possible to notice the emergence of common techniques to analyze online problems and find lower and upper bounds for their advice complexity.

To find an upper bound the most straightforward method is, same as with other complexity metrics, to find an algorithm which solves the problem and then determine its advice complexity. Any optimal algorithm cannot then have any worse advice complexity. While this method is obvious, it is often the most demonstrative one.

Proving lower bounds is usually significantly more difficult. Instead of

showing an algorithm which does not need more than a certain amount of advice, to prove that b is a lower bound, we need to show that any algorithm with a certain guarantee on the competitive ratio cannot achieve this without reading at least b bits.

2.1.1 Common Prefix

Probably the most basic approach to finding the lower bound on the advice complexity of a particular online problem is to find a set of instances with the following properties:

(i) for a given non-negative integer k the prefixes $(x_1^{(i)}, \ldots, x_k^{(i)})$ of instances $I^{(i)}$ are equal, i.e., for two instances $I^{(i)} \neq I^{(j)}$, for each l such that $1 \leq l \leq k$, the members $x_l^{(i)}$ and $x_l^{(j)}$ are equal

(ii) for each pair of instances $I^{(i)} \neq I^{(j)}$ there are no optimal solutions $Opt(I^{(i)}) = (y_1^{(i)}, \ldots, y_{n_i}^{(i)})$, $Opt(I^{(j)}) = (y_1^{(j)}, \ldots, y_{n_j}^{(j)})$ such that

$$(y_1^{(i)}, \ldots, y_k^{(i)}) = (y_1^{(j)}, \ldots, y_k^{(j)})$$

In other words, we find a set of instances such that the algorithm cannot possibly distinguish the prefixes of these instances, however, for each instance a unique solution needs to be yielded in the prefix already. To achieve this, the advice string must necessarily be used. If the size of this set of instances is m, at least $\log m$ advice bits need to be accessed which gives a lower bound on the advice complexity of the problem.

This technique is used in various proofs in [FKS12] and in [Kom12] to prove a lower bound on the advice complexity of disjoint path allocation.

It is possible to generalize this technique to show lower bounds not only on the advice complexity of an optimal solution, but also to show lower bounds for c-competitive algorithms for a given constant c.

In this case, it is useful to look at an online algorithm with b bits of advice as a collection of 2^b deterministic online algorithms with different strategies.

If the problem in question has the property that a strategy (sequence of decisions on the common prefix) leading to an optimal solution for a particular instance I also leads to a competitive solution for a set of similar instances, we can estimate an upper bound on the number of such similar instances, let us denote this by s. A lower bound on the number of required strategies is then obtained as m/s, which means that $\log \frac{m}{s}$ is a lower bound on the number of advice bits.

2.1.2 Reduction to String Guessing

In [Böc+12], the authors use reductions to a simpler problem that is easier to analyze as a method to prove lower bounds. Specifically, they picked the string guessing problem in two variants.

Definition 2.1 (String Guessing with Known History)**.** *The string guessing problem with known history over an alphabet Σ of size $q \geq 2$ (denoted as q-*SGKH*) is defined as follows. The input instance $I = (n, d_1, \ldots, d_n)$ consists of an integer n specifying the length of the instance and a sequence of n characters, where $d_i \in \Sigma, 1 \leq i \leq n$. Let A be an online algorithm that solves q-*SGKH*, then $A(I) = (y_1, \ldots, y_n, \_)$, where $y_i \in \Sigma$. We define the cost of a solution as the Hamming distance between the sequence $(y_1, \ldots, y_n)$ and the sequence $(d_1, \ldots, d_n)$, i.e. the number of wrongly guessed characters.*

Definition 2.2 (String Guessing with Unknown History)**.** *The string guessing problem with unknown history over an alphabet Σ of size $q \geq 2$ (denoted as q-*SGUH*) is defined as follows. The input instance $I = (n, ?_2, \ldots, ?_n, d)$ consists of an integer n specifying the length of the instance, $n-1$ queries without additional information and a string $d = d_1 d_2 \ldots d_n$, where $d_i \in \Sigma, 1 \leq i \leq n$. Let A be an online algorithm that solves q-*SGUH*, then $A(I) = (y_1, \ldots, y_n, \_)$, where $y_i \in \Sigma$. We define the cost of a solution as the Hamming distance between the sequence $(y_1, \ldots, y_n)$ and the sequence $(d_1, \ldots, d_n)$.*

Both q-SGKH and q-SGUH consist of $n+1$ queries where for the first n queries the algorithm is expected to guess a single character of the instance; for the last query no meaningful response is expected, its purpose is only to reveal the input string to allow an offline algorithm to guess the whole string correctly. The only difference between the two variants is that in q-SGKH it is revealed whether the algorithm guessed correctly after each guess and in q-SGUH this is revealed in the last turn.

For the sake of simplicity, we may sometimes speak about the input string $d = d_1 d_2 \dots d_n$ instead of the corresponding input instance $I = (n, d_1, \dots, d_n)$ in the case of q-SGKH or $I = (n, ?_2, \dots, ?_n, d)$ in the case of q-SGUH.

It is easy to observe the following relationship between bounds for the two variants of the string guessing problem.

Observation 2.3. *Any upper bound on the advice complexity of* q-SGUH *is also an upper bound on the advice complexity of* q-SGKH *– any algorithm that solves* q-SGUH *can be used to solve* q-SGKH *as well, simply ignoring the characters provided in each query. Similarly, any lower bound for* q-SGKH *is also a lower bound for* q-SGUH*.*

With this in mind, bounds on the advice necessary to achieve optimality for both veriants have been shown.

Theorem 2.4 ([Böc+12])**.** *The advice complexity of* q-SGUH *is at most* $\lceil n \log q \rceil$*.*

Proof. We prove this theorem by describing an algorithm A using $\lceil n \log q \rceil$ bits of advice which solves both q-SGKH and q-SGUH.

The total number of strings of length n is q^n. These can be sorted in a lexicographic order in which each instance has a position. To encode this position, $\lceil n \log q \rceil$ bits are required.

Therefore, after receiving the number n in the first query, A reads the position m of the string from the advice string and enumerates the first m strings of length n in lexicographic order until it finds the correct one. Then it just yields one character from the string per query. □

Theorem 2.5 ([Böc+12])**.** *The advice complexity of* q-SGKH *is at least* $\lceil n \log q \rceil$.

Proof. We prove this by contradiction. Suppose there is an algorithm A which solves q-SGKH using m bits of advice, $m < \lceil n \log q \rceil$. The total number of instances of length n is q^n. However, using m bits of advice it is possible to only encode $2^m \leq 2^{\lceil n \log q \rceil - 1} < 2^{n \log q} = q^n$ different values. Therefore, there are two input strings d, d' where the same m-bit advice string φ leads to the optimal solution.

Consider the first position i at which strings d and d' differ, i.e., $d_i \neq d'_i$. Since A gives the optimal result for the input string d, in the i-th turn it emits d_i. However, since up until the i-th turn, the input is the same for d' as well and since the advice string is also the same, A is in exactly the same state in the i-th turn when processing d' as it is when processing d. Therefore, for the input string d', A outputs d_i in the i-th turn as well. This contradicts the assumption that A provides an optimal solution for d'. □

The following corollary follows from the previous two theorems and observation 2.3.

Corollary 2.6. *The advice complexity of both* q-SGKH *and* q-SGUH *is* $\lceil n \log q \rceil$.

The following lower bounds on the number of advice bits required to guarantee that an algorithm guesses at least a certain amount of characters right have been established.

Theorem 2.7 ([Böc+12])**.** *To guarantee that an online algorithm* A *guesses at least* αn *characters right for an instance of either* q-SGUH *or* q-SGKH *of length* n*, where* $\frac{1}{q} \leq \alpha < 1$*,* A *needs to access at least*

$$\left(1 + (1-\alpha)\log_q\left(\frac{1-\alpha}{q-1}\right) + \alpha \log_q \alpha\right) n \log q = (1 - H_q(1-\alpha)) n \log q$$

advice bits, where H_q *is the* q*-ary entropy function defined as*

$$H_q(p) = p \log_q(q-1) - x \log_q x - (1-x)\log_q(1-x)$$

for any $q \in \mathbb{N}^{\geq 2}, 0 \leq p \leq 1$.

Even though thanks to observation 2.3 it would suffice to show this bound for q-SGKH, it has been proved for each problem independently as both proofs are interesting in their own right.

The proof for q-SGUH uses the common prefix technique described in the previous subsection. This is possible thanks to the fact that all instances of length n are identical except for the very last query.

In this problem, each strategy is in fact one hard-coded string of length n that an algorithm outputs for each instance. Since the output is allowed to differ in at most $(1 - \alpha)n$ characters, all instances for which a strategy is acceptable have a Hamming distance of at most $(1 - \alpha)n$ from the guessed character. The lower bound is then obtained by estimating the number of strings of length n within the appropriate Hamming distance.

For q-SGKH, however, the common prefix technique is no longer applicable, because an algorithm receives information about the correctness of its guess after each round. Even though this information does not correlate with the rest of the instance in any way, an algorithm may make different decisions based on the correctness of its previous guesses.

The formal proof of this bound is therefore significantly more complicated than in the q-SGUH problem and involves representing each computation as a walk through a complete rooted q-ary tree of depth n and estimating the number of instances in each subtree for which an adversary is able to enforce at most e errors. This number of instances turns out to be the same as in the case of q-SGUH, which leads to the same lower bound.

These results have been used in [Böc+12] to establish lower bounds on the advice required to attain a certain competitive ratio for the online version of the maximum clique problem and the online set cover problem.

2.1.3 Partition Tree

A generalization of the common prefix technique has been introduced in [Bar+14]. It is not always possible to isolate enough instances which all have

the same prefix of sufficient length. However, it may be possible to find a set of instances such that certain pairs of instances share common prefix of some length (where the length may differ for each pair) and, again, require different handling on this prefix.

The technique is formalized by organizing instances into a tree based on their common prefixes.

Definition 2.8 (Partition Tree)**.** *Consider an online problem and a set of instances $\mathcal{I}$ for this problem. We define a* partition tree *$T(\mathcal{I})$ of $\mathcal{I}$ as a labeled rooted tree with the following properties:*

(i) Each vertex v of $T(\mathcal{I})$ is labeled by a non-empty set of instances $\mathcal{I}_v \subseteq \mathcal{I}$ and by a natural number k_v, such that any two instances $I_1, I_2 \in \mathcal{I}_v$ have a common prefix of length at least k_v.

(ii) For each non-leaf vertex v, the instance sets of its children form a partition of $\mathcal{I}_v$. For each child w of v, $k_w \geq k_v$.

(iii) The instance set of the root of $T(\mathcal{I})$ is $\mathcal{I}$.

These properties ensure that if an algorithm processes two instances from the instance set of a single vertex v, the algorithm cannot distinguish these instances based on their prefix of length k_v. If we combine this property with an additional one, that for any two instances belonging to different children of v, different outputs for the common prefix are required, these outputs are only determined by the advice, as the following lemma states.

Lemma 2.9 ([Bar+14])**.** *Let $\mathcal{I}$ be a set of instances for an online problem and let $T(\mathcal{I})$ be a partition tree of $\mathcal{I}$. Let v_1, v_2 be two different vertices of $T(\mathcal{I})$ such that neither is an ancestor of the other and let v be the lowest common ancestor of v_1 and v_2. Let $I_1 \in \mathcal{I}_{v_1}$ and $I_2 \in \mathcal{I}_{v_2}$ and let $OPT(I)$ denote the set of optimal output sequences for any instance I.*

If, for all $\pi_1 \in OPT(I_1), \pi_2 \in OPT(I_2)$, π_1 and π_2 differ in the first k_v elements, then any optimal algorithm needs a different advice string for each of the two instances I_1 and I_2.

If we take, for instance, a partition tree satisfying the prerequisite of lemma 2.9, and apply the lemma to its leaves, we can see that each leaf requires a unique advice string. This observation leads to the following theorem.

Theorem 2.10 ([Bar+14])**.** *Let $\mathcal{I}$ be a subset of the set of all instances of an online problem $\mathcal{P}$ and let $T(\mathcal{I})$ be a partition tree of $\mathcal{I}$ satisfying the prerequisite of lemma 2.9.*

Then, any optimal online algorithm for $\mathcal{P}$ needs to read at least $\log m$ bits of advice, where m is the number of leaves of $T(\mathcal{I})$.

While the prerequisite of lemma 2.9 might appear to be rather difficult to prove, it is easily satisfied if we create a set of instances $\mathcal{I}$ orgainzed in a partition tree $T(\mathcal{I})$, such that every leaf of $T(\mathcal{I})$ contains only one instance, each instance has only one optimal output sequence and an optimal sequence for some instance from $\mathcal{I}$ is not optimal for any other instance in $\mathcal{I}$.

If a tree satisfies these three conditions, all that is left to show for lemma 2.9 to hold is that for every pair of instances from $\mathcal{I}$, the output sequences differ in the first k items, where k is the length of their common prefix.

2.2 Selected Known Results

This section gives an overview of selected known results in advice complexity which we use to demonstrate the techniques described in the previous section. First we show some applications of the common prefix technique on a few online graph coloring results and then we show an application of the string guessing problem for the maximum clique problem.

2.2.1 Graph Coloring

Graph coloring is a classic, well-known computational problem. Its offline version is one of the original 21 NP-complete problems published by Karp

[Kar72]. It comes as no surprise, then, that for the most genral version of this problem, online algorithms are unable to perform well [HS94].

An online graph coloring algorithm works roughly as follows. In each round, a single vertex of the input graph is revealed to the algorithm, which in turn has to assign a color to this vertex. More precisely, assuming the vertices of a graph are ordered in a sequence, in t-th turn the algorithm has the knowledge of the subgraph induced by the first t vertices in this sequence. That means, each edge is revealed as soon as both of its ending vertices are known.

In the offline version of graph coloring, making certain assumptions about the input graph may dramatically reduce the difficulty of the problem. For instance, if we assume the graph is bipartite, the difficulty drops from NP-hard to a basic polynomial graph exploration algorithm.

This property carries over to online graph coloring as well. The difficulty of this problem depends greatly on any assumptions we make about the input instance, e.g. restrictions on the class of graphs, such as trees, bipartite graphs, cycles or a relationship between the number of vertices and the number of edges, or the order in which their vertices are revealed to the online algorithm. All these assumptions provide the algorithm with additional information. This means that by comparing the advice required to solve these special cases to the advice complexity of the general case we can quantify the amount of information provided by a particular set of assumptions.

The order in which vertices are revealed is referred to as the *presentation order*. In the most general case, the vertices will appear in a fully arbitrary order. We can restrict this to a connected presentation order, which means that in each turn the vertex currently revealed is connected to at least one vertex revealed previously. This can be restricted even further to the order in which a depth-first search (DFS) or a breadth-first search (BFS) will visit vertices. Another common presentation order is when the sequence of vertices is sorted by their degrees.

Definition 2.11. *In* OnlineColoring *the instance is an undirected graph*

$G = (V, E)$ with $V = \{1, 2, \ldots, n\}$. This graph is presented to an online algorithm in turns: In the k-th turn the online algorithm receives the graph $G_k = G[\{1, 2, \ldots, k\}]$, i.e., a subgraph of G induced by the vertex set $\{1, 2, \ldots, k\}$. As its reply, the online algorithm must return a positive integer: the color it wants to assign to vertex k. The goal is to produce an optimal coloring of G – the online algorithm must assign distinct integers to adjacent vertices, and the largest integer used must be as small as possible.

When talking about a variant of OnlineColoring, we always need to specify the class of graphs it is restricted to and the presentation order. We denote this by OnlineColoring(X, Y) where X is the class of graphs G will belong to and Y is the presentation order. For the class of graphs we will use its common name (e.g., "bipartite", "planar") with the special class called "any" meaning that there is no restriction on G at all. For the presentation order we will use "connected", "BFS", "DFS" and "max-degree" with meanings as discussed earlier and, again, "any" with the meaning that the vertices may be presented in a fully arbitrary order. For instance, OnlineColoring(bipartite, connected) denotes that the problem is restricted to bipartite graphs and their vertices are revealed in a connected order. As a special case, OnlineColoring(any, any) denotes the most general version of the problem where no assumptions are made at all.

The value of n is not known to the online algorithm beforehand. The reason for this is that it would provide the algorithm with additional information about the input instance which may (and in some cases does) affect the advice complexity of the problem.

This problem has been studied in [FKS12; Her13]. We reproduce some of the results below.

General Graphs

The following asymptotically tight estimates on the advice complexity of the most general case of online graph coloring have been established.

Theorem 2.12 ([FKS12])**.** *There is an online algorithm with advice which solves* OnlineColoring(any, any) *using* $n \log n - n \log \log n + O(n)$ *bits of advice.*

The general idea is to encode the position of an optimal coloring in a lexicographically sorted list of all partitions of the set of vertices on the advice tape.

Theorem 2.13 ([FKS12])**.** *The* OnlineColoring(any, BFS) *problem has an advice complexity of at least* $n \log n - n \log \log n + O(n)$*.*

Proof outline. The proof of this theorem uses the common prefix technique described in section 2.1.1. We will not reproduce the details as they are relatively complicated. For a full proof, refer to the original paper. □

These results are crucial in order to quantify how much a restriction on the class of graphs simplifies the coloring problem by means of advice complexity.

Bipartite Graphs

As a reminder, bipartite graphs are those that can be colored using two colors.

Theorem 2.14 ([FKS12])**.** *There is an optimal deterministic online algorithm for* OnlineColoring(bipartite, connected) *without advice.*

Proof. The algorithm for an optimal coloring is trivial. For the first vertex it picks an arbitrary color and afterwards, for each vertex there is at least one neighbor whose color has already been assigned. Therefore the algorithm just picks the other color. □

This result shows that for bipartite graphs it does not really make any sense to analyze any of the connected presentation orders. However, for presentation orders without any restrictions this class of graphs is still interesting from the point of view of advice complexity.

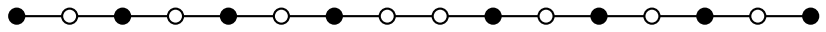

Figure 2.1: *Example of an independent set on a path with $n = 16$ vertices for $x = 4$. Vertices from $P_x \cup Q_x$ are filled.*

Paths

Paths are a subclass of bipartite graphs, therefore it is only interesting to analyze the most general presentation order.

Theorem 2.15 ([FKS12])**.** *The* OnlineColoring(path, any) *problem has an advice complexity of* $\lceil \frac{n}{2} \rceil - 1$.

Proof. For the upper bound, consider an algorithm A which selects an arbitrary color for the first vertex and then reads one bit of advice for every isolated vertex in the input, which is interpreted as the color. For each vertex u connected to some already processed vertex v, A needs to output the color opposite to that of v.

It is easy to see that on a path, at most $\lceil \frac{n}{2} \rceil$ vertices can be selected this way, since the selected vertices have to form an independent set.

To show a lower bound of $\lfloor \frac{n}{2} \rfloor - 1$, we use the common prefix technique. Assume n is even and let us denote the vertices $v_1, \ldots, v_n$ according to their order on the path. Note that this notation does not correlate with the presentation order.

For any $1 \leq x \leq n/2$, consider two sets of vertices $P_x = \{v_{2i-1} \mid 1 \leq i \leq x\}$ and $Q_x = \{v_{2i} \mid x+1 \leq i \leq n/2\}$. The set $P_x \cup Q_x$ forms an independent set such that vertices from P_x have to share one color while all vertices from Q_x need to have the other color. An example is shown in figure 2.1.

Consider the set of all strings of the form $\{\mathtt{p}\} \cdot \{\mathtt{p}, \mathtt{q}\}^{n/2-1}$. For each such string we can now create an instance. Let x be the number of $\mathtt{p}$ characters in a given string w. An instance can be created such that for each $\mathtt{p}$ character, a vertex from P_x is selected and for each $\mathtt{q}$, a vertex from Q_x is chosen. This sequence of vertices forms the prefix of an instance, which is also an independent set.

For every such instance, an optimal algorithm needs to assign one color for every vertex from P_x and the other color for all vertices from Q_x, while the prefix of length $n/2$ looks the same for each instance. The number of different strings is $2^{n/2-1}$, which gives a lower bound of $n/2-1$ on the number of advice bits.

If we also consider odd n, the above proof implies a lower bound of $\lfloor n/2 \rfloor - 1$ bits. An additional bit can be forced, however, this requires a more detailed analysis which can be found in [FKS12]. □

2.2.2 Maximum Clique

The problem of finding the maximum clique in a graph is another example of an NP-complete problem, which is also part of the original 21 problems published by Karp [Kar72]. Similar to the graph coloring problem, in the online version of maximum clique, the vertices of an input graph are revealed to an algorithm one by one and the algorithm needs to decide whether to select a vertex into its solution or not.

We use this problem to show an example of a proof by reduction to string guessing, as discussed in section 2.1.2, offered in [Böc+12]. The authors of this paper use a cost function which allows the algorithm to produce a graph that is not a clique with a penalty for every selected vertex that is not part of the maximal clique in a graph induced by the vertices selected by A. This is to avoid pathologic edge cases where A cannot select any vertex after accepting an isolated vertex at the beginning.

The problem is formally defined as follows.

Definition 2.16 (MaxClique)**.** *In* MaxClique*, the input is an undirected graph $G = (V, E)$ with $V = \{1, \ldots, n\}$ and the goal is to select a clique $C \subseteq V$ in G. The graph is presented to an online algorithm A in turns: in the k-th turn, the online algorithm receives the graph $G_k = [\{1, \ldots, k\}]$, i.e., a subgraph of G induced by the vertex set $\{1, \ldots, k\}$. In each turn, A has to decide whether $i \in C$ or not.*

Let $A(I)$ be the set of vertices selected by A and let $C_{A(I)}$ be a maximum clique in the graph $G_{A(I)}$. The cost function is defined by $C(A(I)) = |C_{A(I)}|^2/|A(I)|$.

If an algorithm selects a clique, the value of the cost function thus defined is as we would expect – the number of vertices selected. This holds for the optimal solution as well, which means the competitive ratio of an algorithm A can be expressed as

$$c = \frac{C(Opt(I))}{C(A(I))} = \frac{|A(I)|}{|C_{A(I)}|} \cdot \frac{|C_{opt}|}{|C_{A(I)}|},$$

where C_{opt} denotes the maximum clique in G. The first ratio can be interpreted as a measure of how many wrong vertices A has selected and the second one measures how many correct vertices A has rejected.

The following lower bound for MaxClique has been proved. Recall that H_n is defined in theorem 2.7 as the n-ary entropy function; as a special case, $H_2(x)$ is commonly denoted by $H(x)$.

Theorem 2.17 ([Böc+12])**.** *Any $(c-\varepsilon)$-competitive algorithm A for* MaxClique *needs at least*

$$(1 + (c-1)\log(c-1) + (2-c)\log(2-c))\frac{n-2}{2} = (1 - H(c-1))\frac{n-2}{2}$$

bits of advice for any $1 < c \leq 3/2$ and $\varepsilon > 0$.

Proof outline. The full proof of theorem 2.17 consists of many nontrivial steps, most of which are not interesting for our purpose of deminstrating the string guessing reduction. Thus we only focus on the first part, where we show how MaxClique can be used to solve 2-SGKH; the rest of the proof can be found in [Böc+12].

We consider the following set of instances such that every instance corresponds to a binary string. Let $b = b_1 b_2 \dots b_{n'}$ be a binary string of length n'. We construct a graph $G_b = (V_b, E_b)$ corresponding to b, with $n = 2n' + 2$ vertices. Let

$$V_b = \{v_{i,j} \mid 1 \leq i \leq n', 0 \leq j \leq 1\},$$

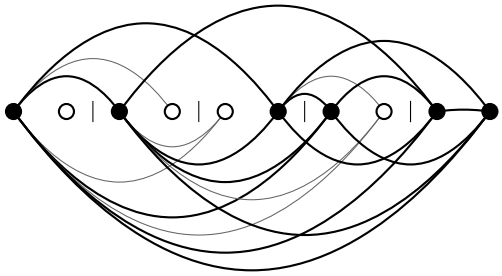

Figure 2.2: *Example of the graph G_{0010}. Maximum clique is highlighted in black.*

and let $V'_b = \{v_{i,b_i} \mid 1 \leq i \leq n'\}$ be a subset of V_b selected by the string b. The set of edges is chosen as

$$E_b = \{\{v_{i,b_i}, v_{j,k}\} \mid 1 \leq i < j \leq n', 0 \leq k \leq 1\} \\ \cup \{\{v, v_{n'+1,0}\}, \{v, v_{n'+1,1}\} \mid v \in V'_b\} \cup \{\{v_{n'+1,0}, v_{n'+1,1}\}\}.$$

The vertices are presented according to the lexicographic order of their indices. An example of such a graph is presented in figure 2.2.

The graph G_b is constructed in such a way that V'_b, together with vertices $v_{n'+1,0}$ and $v_{n'+1,1}$, forms the only maximum clique. In addition, for every $1 \leq i \leq n'$, the two vertices $v_{i,0}$ and $v_{i,1}$ are indistinguishable at the time of their presentation to an algorithm, since both of them are connected to the same set of preceding vertices.

An algorithm A' for solving MaxClique can be used as an oracle for the 2-SGKH problem in the following way. For every time step of the 2-SGKH computation, A simulates two time steps of a computation of A'. For the i-th character of the string guessing problem, A submits the vertices $v_{i,0}$ and $v_{i,1}$ to A' and guesses 0 if A' selects $v_{i,0}$, otherwise it guesses 1.

A constructs the graph G_b in an online fashion based on the information about the string b it is guessing. After reading the $i+1$-th query, A knows the value of b_i, which means it can decide which one of the vertices $v_{i,0}, v_{i,1}$ belongs to V'_b. This way, A is able to build the set V'_b online and reveal edges from G_b to A' in a consistent manner.

A careful analysis of this algorithm shows that assuming there is no algorithm for 2-SGKH which guesses more than $\alpha n'$ characters correctly using b advice bits, for every algorithm A for MaxClique using at most b bits of advice,

$$C(A(G_b)) \leq \frac{(\alpha n + 2 + (1-\alpha)n')^2}{\alpha n' + 2 + 2(1-\alpha)n'}.$$

The next step of the proof is to show that for each pair of wrongly guessed vertices $v_{i,0}, v_{i,1}$, the cost function is maximized if an algorithm selects both vertices. The theorem then follows from these facts and from theorem 2.7. □

Chapter 3

Disjoint Path Allocation

Disjoint path allocation is a well-studied specialization of the more general problem of call admission in arbitrary networks. In the general case, a dispatcher needs to decide which calls to admit based on the topology of the network, capacities of its edges, and the bandwidth and duration of each call.

In the case of disjoint path allocation, we restrict ourselves to a path on $L + 1$ vertices where all edges have the same capacity, which is equal to the bandwidth of each call. In addition, each call has an unlimited duration.

In this chapter we build on the results published in [Bar+14], therefore we use the same definition of the problem as in the aforementioned article.

Definition 3.1 (DPA)**.** *The* disjoint path allocation problem (DPA) *is the following maximization problem on a path $P = (v_0, \ldots, v_L)$. First, the value of L is revealed. Then n requests of the form (i_k, j_k) follow, where each such pair denotes the subpath of P from v_{i_k} to v_{j_k}. For each pair an algorithm decides whether to admit or deny the request. All admitted requests must be pairwise edge-disjoint. The goal is to maximize the number of admitted requests.*

This problem can be looked at as a series of call requests where each call has infinite duration and each edge can accommodate at most one call. Note that the number of requests is not known in advance, only the length of the path.

3.1 Competitiveness without Advice

Before we delve into the area of advice complexity, we focus on an analysis of deterministic online algorithms without advice.

Section 13.5 of [BE98] presents a proof that no deterministic algorithm can guarantee a competitive ratio better than linear in the number of vertices when restricted to strict competitiveness. We reproduce the proof below.

Theorem 3.2 ([BE98])**.** *On a path on $L + 1$ vertices, any deterministic online algorithm A has a competitive ratio of at least L. Specifically, there exists either an input instance I_1 where $C(Opt(I_1)) = L$ and $C(A(I_1)) = 1$ or an instance I_2 for which $C(Opt(I_2)) = 1$ and $C(A(I_2)) = 0$.*

Proof. We prove the theorem using an adversary Adv. Consider an algorithm A. The adversary reveals L and issues as the first query $(0, L)$. If A rejects this query, Adv terminates the input instance, which leads to the second case in the theorem and it means A is not competitive.

If A admits the first query, Adv follows up with L requests: $(0, 1), (1, 2), \ldots, (L - 1, L)$. Since A has already admitted a request spanning the whole path P, it cannot admit any of these following requests, while the optimal solution is to reject the first request and admit all of the following L requests. This leads to the first case and means that the competitive ratio of A is at least L. □

The proof of theorem 3.2 might appear to rely on a pathologic edge case made possible by the definition of strict competitiveness: it leans on the fact that each algorithm that denies the first request can be made non-competitive and setting the parameter α from definition 1.2 to a value of only 1 would eliminate this.

Indeed, Komm described in [Kom12] an algorithm that achieves a competitive ratio of $\lceil \frac{L}{\alpha+1} \rceil$, which seems to indicate that relaxing the definition of competitiveness might lead to better results. However, he also proved that the competitive ratio of any deterministic algorithm is at least linear in the number of requests.

Since we study DPA primarily with respect to the length of the communication network, we complement this with a lower bound on the competitive ratio, which is one of our new results.

Theorem 3.3. *Consider an arbitrary value of α in the definition of competitiveness. On a path on $L+1$ vertices, any deterministic online algorithm has a competitive ratio of at least $\frac{\lfloor\sqrt{L}\rfloor}{\alpha+1}$.*

Proof. Let A be a c-competitive deterministic online algorithm for DPA, let α be a positive constant such that $C(A(I)) \geq \frac{C(Opt(I))}{c} - \alpha$. We use an adversary Adv to prove the bound.

Let $k := \lfloor\sqrt{L}\rfloor$. Adv starts by issuing non-overlapping requests of length k: $(0,k),(k,2k),\ldots,$ until either A admits a request, or Adv submits the request $(k(k-1),k^2)$.

In the former case, let $(ik,(i+1)k)$ be the first (and only) request admitted by A. Adv then submits the following k requests and terminates the input: $(ik,ik+1),(ik+1,ik+2),\ldots,((i+1)k-1,(i+1)k)$. Each of these requests overlaps the single admitted request, therefore A has to deny all of them.

The optimal solution for this instance is to admit the first i requests of length k, deny the $i+1$-th request and admit all of the following k requests of length 1, which means $C(Opt(I)) = i+k$, while $C(A(I)) = 1$. Since A is c-competitive, the following inequalities hold.

$$1 \geq \frac{k+i}{c} - \alpha$$
$$c \geq \frac{k+i}{\alpha+1} \geq \frac{k}{\alpha+1} = \frac{\lfloor\sqrt{L}\rfloor}{\alpha+1}$$

In the latter case, Adv terminates the input after request $(k(k-1),k^2)$. The optimal solution of this instance is to admit all k requests, while A rejects everything, which results in these inequalities:

$$0 \geq \frac{k}{c} - \alpha$$
$$c \geq \frac{k}{\alpha} \geq \frac{\lfloor\sqrt{L}\rfloor}{\alpha+1}$$

□

This result indicates that even though relaxing the condition of competitiveness does make it possible to obtain a better competitive ratio with a deterministic algorithm, it still leaves a significant gap between the optimal solution and deterministic online algorithms. Therefore in the rest of this chapter we will adhere to the strict definition, unless noted otherwise.

3.2 Advice Complexity of Optimal Solution

The first result regarding the advice complexity of DPA has been published in [Kom12] and it states that the minimum amount of advice required to achieve optimality is $L/2$ bits. The proof of this bound uses the common prefix technique described in section 2.1.1: the common prefix consists of $L/2$ requests of length 2 and then in each instance, a different subset of these two-edge paths is chosen and for each of them two single-edge requests are issued. We generalize this bound for competitive algorithms in theorem 3.12.

This bound has since been improved in [Bar+14] to $L-1$ bits. The new bound is tight – an optimal algorithm using exactly $L-1$ bits has been published as well. We reproduce the algorithm below, since multiple competitive algorithms discussed later on are modified versions of this particular optimal algorithm.

Theorem 3.4 ([Bar+14])**.** *There is an online algorithm which guarantees an optimal solution using $L-1$ bits of advice.*

Proof. The algorithm A works as follows. After obtaining the length L, A reads $L-1$ bits from the advice string with the following meaning: the i-th bit (denoted by b_i, for $i \in \{1, \ldots, L-1\}$ indicates whether A should accept a request starting in vertex v_i. We always set b_0 to 1.

Then, whenever A processes a request (i, j) that does not conflict with any already admitted request, A accepts it iff $b_i = 1$ and $b_k = 0$ for all $i < k < j$. □

The proof of the lower bound uses the partition tree technique from sec-

tion 2.1.3. Since the technique used in this proof might be used to improve our bound proposed in section 3.3.2, we summarize it below.

Theorem 3.5 ([Bar+14])**.** *Any optimal online algorithm for DPA needs to read at least* $L - 1$ *bits of advice.*

Proof. We construct a set $\mathcal{I}$ of instances which can be organized in a partition tree $T(\mathcal{I})$ in a way that they satisfy the conditions described at the end of section 2.1.3.

Each instance I corresponds to a binary string b of length $L + 1$, where $b = b_0 b_1 \dots b_L$ and $b_0 = b_L = 1$. The i-th bit of b is a label for the i-th vertex of the path in I. All vertices labeled by 1 are the end points of two requests that are supposed to be accepted except for v_0 and v_L, each of which is the end point of only a single request.

Instance I consists of L phases numbered from L down to 1. In phase p, all requests of length p are asked from left to right, with some exceptions, as required by the bit vector associated with I. Specifically, if a request (i, j) is supposed to be admitted, i.e. $b_i = b_j = 1$ and $b_k = 0$ for all $i < k < j$, the request (i, j) is the last one on the subpath v_i, v_j and in all subsequent phases, requests on this subpath are omitted. Figure 3.1 shows an example of an instance constructed in this way.

Now we need to show that this set of instances has the required properties. We show by contradiction that for each instance from $\mathcal{I}$, the solution described in the previous two paragraphs is the only optimal solution. Let us denote by $Opt(I)$ the expected solution and assume there is a solution $Opt'(I)$ such that $C(Opt'(I)) \geq C(Opt(I))$ which differs from $Opt(I)$ in at least one answer. We know the expected solution is indeed optimal, as the binary string associated with each instance is precisely the one used by the optimal algorithm presented above. There are two possibilities how this can happen. Either $Opt'(I)$ rejects a request (i, j) admitted by $Opt(I)$, in which case there are no further requests on subpath v_i, v_j, which means $Opt'(I)$ admits one less request than $Opt(I)$, which means its cost is lower than that of $Opt(I)$. Otherwise, $Opt'(I)$ needs to admit a request (i, j) not admitted

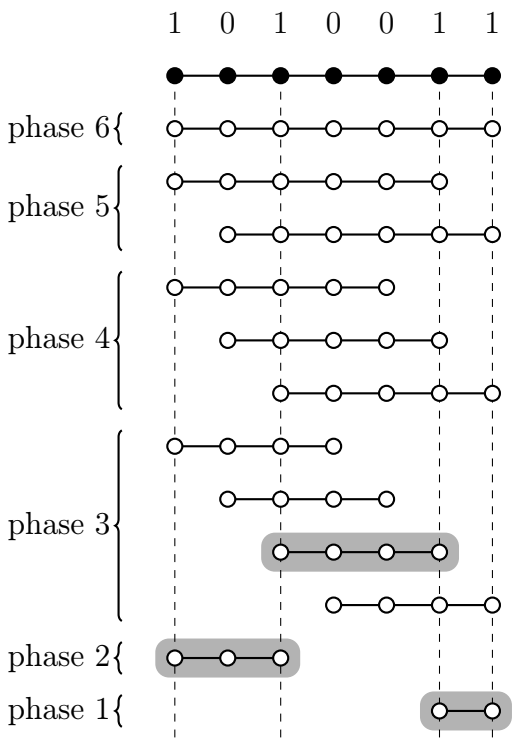

Figure 3.1: *Example of an input instance for the string* 1010011*. The optimal solution is highlighted in gray.*

by $Opt(I)$. However, by construction of I we know that there is at least one 1 bit between i and j, which means $Opt'(I)$ cannot admit at least two other requests admitted by $Opt(I)$, which, again, leads to a contradiction.

This also implies that each output sequence is optimal for only one instance from $\mathcal{I}$.

The only property left to show is that for each two instances, the optimal outputs differ in the common prefix of the two instances. Let $I_1, I_2 \in \mathcal{I}$ be two different instances, let p be the first phase in which they differ, without loss of generality let $r_p = (i, i+p)$ be a request which appears in I_1, but not in I_2. Since phase $p+1$ is identical in the two instances, both contain the request $r_{p+1} = (i, i+p+1)$. Since r_p appears in I_1, r_{p+1} is not admitted by $Opt(I_1)$, however, it is admitted by $Opt(I_2)$. Thus the optimal output sequences for I_1 and I_2 differ in phase $p+1$ already.

Since it is easy to organize all 2^{L-1} instances into a partition tree based on their common prefixes and each instance gets its own leaf, the prerequisite of lemma 2.9 is satisfied and the number of bits required is at least $L-1$. □

3.3 Bounds for Constant Competitiveness

3.3.1 Upper Bound for Small c

An upper bound on the amount of advice required for c-competitiveness for c close to 1 has been published in [Bar+14] as a modification of the optimal algorithm. The original paper only contains a sketch of a proof; we describe the modified algorithm in detail below.

Theorem 3.6 ([Bar+14])**.** *For each $c = k/(k-1)$, where k is an integer greater than 1, there is a c-competitive algorithm for DPA that uses*

$$\left\lceil \log \frac{c}{c-1} \right\rceil + L - 1 - \left\lfloor \left\lfloor (L-2)\frac{c-1}{c} \right\rfloor \cdot (2 - \log 3) \right\rfloor$$

bits of advice.

Proof. Let $\varphi = b_1 \dots b_{L-1}$ be an advice string for the optimal algorithm from the proof of theorem 3.4 leading to an optimal solution. We modify the advice string by adding the value of every k-th bit to the bit immediately preceding it and omitting the bit itself from the sequence. This way, we replace some pairs of successive bits with ternary numbers. For example, consider the following sequence: $(1, 0, 1, 1, 0, 1, 0, 0, 1, 0)$. For $k = 4$, we obtain the following sequence: $(1, 0, 1 + 1, 0, 1, 0 + 0, 1, 0) = (1, 0, 2, 0, 1, 0, 1, 0)$. Assuming the positions of ternary numbers in the sequence are known beforehand, it is possible to encode the modified sequence using $L - 1 - 2p + \lceil p \log 3 \rceil$ bits, where p is the number of added pairs.

Given a sequence thus shortened, it is possible to reconstruct most of the original sequence φ. Each sum of two bits needs to be replaced by a pair of bits. In cases where the sum is 0 or 2, the original pair of bits is unambiguous, however, if the sum is 1, there are two possibilities. In this case, our algorithm will always assume that the original pair of bits was $0, 1$. With this reconstructed sequence φ' it is now possible to simulate the optimal algorithm.

Clearly, using φ' as the advice instead of φ can lead to the algorithm rejecting some requests which would be admitted in an optimal solution. This can happen only in case a pair of bits was $1, 0$ before adding them; in this case, the algorithm would expect a request starting at the second position, which might not arrive. It will, however, not block any requests starting before the first of the two bits. Therefore, if e is the number of pairs reconstructed incorrectly, the cost of a solution produced by this algorithm will be at least $C(Opt(I)) - e$.

However, simply selecting bits $b_{k+1}, b_{2k+1}, \dots$ does not lead to the required competitive ratio. For example, let $k = 3$ and assume the original advice string φ is 010010010. By selecting every 3-rd bit, the compressed string becomes 010101 and the reconstructed φ' becomes 001001001. Now, if the only requests the instance contains are those starting on positions 2, 5, and 8, our algorithm would admit no request and thus fail to be competitive.

The solution is to consider all strategies for choosing the pairs of bits to add of the following form: for each $1 \leq i \leq k$, the i-th strategy is to choose b_{i+ak}, b_{i+ak+1} for all integers a such that $0 \leq a \leq \left\lceil \frac{L-2}{k} - 1 \right\rceil$ as the pairs to add. In each strategy, the number of pairs is $p \geq \left\lfloor \frac{L-2}{k} \right\rfloor$. Of all such strategies, the one with the smallest number of errors is chosen, and its number i is encoded in the advice string.

This way of choosing strategies ensures that for each bit b_j, exactly two strategies are considered where b_j is part of an added pair, once in the position of the first bit and once in the position of the second bit (with the exception of the first and last bit, of course). Thanks to this fact, if we sum all encoding errors over all strategies, each bit can contribute to this sum at most once, and in addition, only 1 bits can contribute at all. From this and from the fact that the cost of the optimal solution is at least the number of 1 bits (possibly +1 if the optimal solution accepts a request starting in v_0) follows that $C(Opt(I)) \geq ke$, where e is the number of errors in the best strategy.

The competitive ratio of this solution is then obtained as follows.

$$\frac{C(Opt(I))}{C(Opt(I)) - e} \leq \frac{C(Opt(I))}{C(Opt(I)) - \frac{C(Opt(I))}{k}} = \frac{k}{k-1}$$

The amount of advice required is therefore $\lceil \log k \rceil$ bits to encode the number of strategy used, and $L - 1 - 2p + \lceil p \log 3 \rceil$, where the value of p is described above, which matches the theorem. □

We present a new result which simplifies the above algorithm and requires less advice for the same competitive ratios.

Theorem 3.7. *For each $c = (p+q)/q$, where p, q are positive integers, there is a c-competitive algorithm for DPA which uses*

$$\lceil \log(p+q) \rceil + L - 1 - \left\lfloor (L-1)\frac{p}{p+q} \right\rfloor$$

bits of advice.

Proof. The algorithm works in a similar fashion to the one presented in the proof of theorem 3.7. However, instead of adding certain bits to the previous ones, it simply leaves them out of the advice string.

More precisely, given an advice string φ leading to an optimal solution, we split φ into alternating blocks of lengths p and q bits (with a possible exception of the first and last blocks, which may be shorter). We retain blocks of length q and we leave out blocks of length p, thus removing s bits, where $s \geq \lfloor (L-1)p/(p+q) \rfloor$.

Again, our algorithm A first reconstructs an approximation φ' of the original string φ, by filling all the gaps in φ' with zeroes and then simulates the optimal algorithm with φ'.

Some of the omitted bits will have been ones; for every such bit, $C(A(I))$ will decrease by 1 compared to the optimal solution. In order to guarantee the expected competitive ratio, again, we need to consider $p+q$ possible strategies based on whether the first block is retained or omitted and setting its length to some nonnegative integer less than or equal to q or p respectively, and choose the best strategy.

Let us denote the number of errors (i.e. bits whose value in φ is 1 and in φ' it is 0) in the i-th strategy as e_i. We already know that if we choose the i-th strategy, $C(A(I)) \geq C(Opt(I)) - e_i$. Every 1 bit contributes to the error count of p strategies, which, combined with the fact that $C(Opt(I))$ is at least the number of 1 bits, gives $p \cdot C(Opt(I)) \geq \sum_{i=1}^{p+q} e_i \geq (p+q)\overline{e}$, where $\overline{e}$ is the error count of the best strategy, i.e. the lowest of all e_i.

The competitive ratio is therefore obtained from the following inequalities.

$$\frac{C(Opt(I))}{C(Opt(I)) - \overline{e}} \leq \frac{C(Opt(I))}{C(Opt(I)) - \frac{p \cdot C(Opt(I))}{p+q}} = \frac{p+q}{q}$$

The advice string consists of a binary encoding of the number of chosen strategy taking $\lceil \log(p+q) \rceil$, bits followed by the shortened string from an optimal solution taking $L - 1 - s$ bits. □

Other upper bounds were shown in [Bar+14] as well, such that each of them is the best for a certain interval of competitive ratios.

Theorem 3.8 ([Bar+14], Theorem 6)**.** *There is a c-competitive online algorithm, for $c = 2\sqrt{k}$ with $k \in \mathbb{N}$, that uses at most*

$$\left\lceil \left\lceil \frac{4L}{c^2} \right\rceil \log 3 \right\rceil$$

bits of advice.

Theorem 3.9 ([Bar+14], Theorem 7)**.** *There is a c-competitive online algorithm, for $c \in \mathbb{N}$, that uses at most*

$$\left\lceil \frac{2(L-1)}{c} \right\rceil$$

bits of advice.

Theorem 3.10 ([Bar+14], Theorem 8)**.** *There is a c-competitive online algorithm, for $c = 4 \log k$ with $k \in \mathbb{N}^{\geq 2}$, that uses at most*

$$\left\lceil \frac{L}{2^{c/4}} \cdot \left(\frac{c}{2} + \lceil \log c \rceil + 0.33 \right) \right\rceil$$

bits of advice.

Taking into account only the bounds from [Bar+14], the bound from theorem 3.6 is the best for the interval $1 \leq c \leq 3$; for $3 \leq c \leq 2\sqrt{3}$, theorem 3.9 takes over; for $2\sqrt{3} \leq c \leq 64$, theorem 3.8 is even better; and the rest, $c > 64$, is covered by theorem 3.10.

Our bound from theorem 3.7 is obviously better than the one from theorem 3.6, which makes it the best one for $1 \leq c \leq 3$. However, it also outperforms both bounds from theorems 3.8 and 3.9 for $c \leq 4 \log 3 \approx 6.34$, which makes theorem 3.9 obsolete.

3.3.2 Lower Bound for Small c

We obtain a lower bound by applying the generalized common prefix technique from section 2.1.1. First, we isolate a set of instances with these two properties: 1. all instances share the same prefix, and 2. for any two instances,

the decisions of an online algorithm on the common prefix must be different in order to obtain an optimal solution. We use the same set as described in section 3.2 for the first lower bound on the advice required by an optimal solution.

Next, we observe that the decisions leading to an optimal solution for an instance I are sufficient to obtain a good enough solution (i.e. one whose cost fits within the range allowed by a given competitive ratio) for a set of "similar" instances C_I and compute an upper bound on the number of such instances. Finally, a lower bound on the length of the advice string is obtained as the binary logarithm of a fraction of the number of instances and the upper bound of $|C_I|$.

In order to estimate the size of C_I, we will use the following lemma. Recall that $H(x)$ is the binary entropy function as defined in theorem 2.7.

Lemma 3.11 ([FG06])**.** *Let $n \geq 1$ and $0 < q \leq 1/2$. Then*

$$\sum_{i=0}^{\lfloor qn \rfloor} \binom{n}{i} \leq 2^{n \cdot H(q)}.$$

A more straightforward way to write the above bound is

$$2^{n \cdot H(q)} = \left(\frac{1}{q}\right)^{nq} \cdot \left(\frac{1}{1-q}\right)^{n(1-q)}.$$

This allows us to state the following lower bound, which is an original result of our work.

Theorem 3.12. *Any online algorithm for DPA which guarantees a competitive ratio $1 < c \leq \frac{4}{3}$ needs to read at least*

$$L\left(\frac{1}{2} + \left(1 - \frac{1}{c}\right)\log\left(2 - \frac{2}{c}\right) + \frac{1}{2}\left(\frac{2}{c} - 1\right)\log\left(\frac{2}{c} - 1\right)\right)$$

bits of advice.

Proof. Let $L = 2k$ for some positive integer k. Consider the following set $\mathcal{I}$ of instances consisting of two phases. The first phase of each instance consists

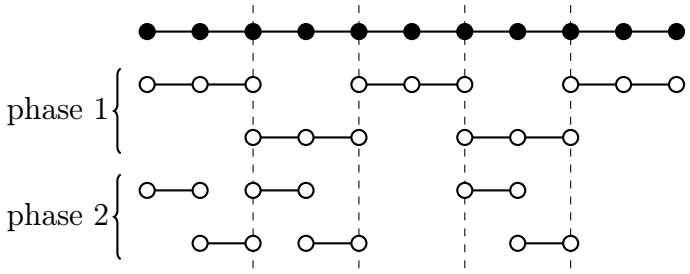

Figure 3.2: *Input instance corresponding to the string* 11010.

of requests $(2i, 2i+2)$ for all $0 \leq i < k$. The second phase is unique for each instance and consists of pairs of requests $(2i, 2i+1), (2i+1, 2i+2)$, where i is chosen from some subset of $\{0, \ldots, k-1\}$.

Each instance corresponds to a binary string of length k: if the i-th bit in the binary string is 1, the second phase contains requests $(2i, 2i+1), (2i+1, 2i+2)$, otherwise, it contains no request on this subpath. For an example, refer to figure 3.2.

The optimal solution for each instance is to admit all requests in phase 2 and those in phase 1 that do not overlap any requests in phase 2. If we look at the binary representation of an instance, that means the optimal solution admits one request for every zero bit (phase 1) and two requests for every one bit (phase 2).

A crucial observation is that for each mistake an online algorithm A makes in phase 1, the cost of its solution decreases by at least one: if A admits a phase 1 request corresponding to a one bit, it will not be able to accept any of the two phase 2 requests on this subpath, and if it rejects a phase 1 request corresponding to a zero bit, there will not be any further requests in phase 2 for this subpath.

Moreover, for each instance $I \in \mathcal{I}$, the equality $C(Opt(I)) = pL$ holds for some $p \in [1/2, 1]$. That means, if we allow A to make e mistakes in phase 1, its competitive ratio is described by the inequality

$$\frac{pL}{pL - e} \leq c.$$

Solving this inequality for e gives us an upper bound on the number of errors:

$$e \leq qL\frac{c-1}{c} \leq L\frac{c-1}{c}. \tag{3.1}$$

Since the first k queries are the same in all instances, all phase 1 decisions an online algorithm makes only depend on the advice string. Each sequence of decisions in phase 1 is optimal for one instance $I \in \mathcal{I}$ with its corresponding bit vector B. In addition, if we allow at most e errors in phase 1, the same sequence of decisions is acceptable for an instance I' with a bit string B' if $Ham(B, B') \leq e$, where Ham denotes the Hamming distance. The number of such acceptable instances is then $Vol_2(k, e) = \sum_{i=0}^{e} \binom{k}{i}$, which is the volume of a binary Hamming ball of radius e around a string of length k.

We already have an upper bound on e from (3.1). If we substitute this into the bound on $Vol_2(n, qn)$ from lemma 3.11, we obtain an upper bound on the number of instances from $\mathcal{I}$ that can be served with a single advice string in order to achieve c-competitiveness. For simplicity, we use q to denote the expression $2(c-1)/c$.

$$\begin{aligned} Vol_2(k, e) &\leq Vol_2\left(k, 2k\frac{c-1}{c}\right) \\ &= \sum_{i=0}^{\lfloor qk \rfloor} \binom{k}{i} \\ &\leq \left(\frac{1}{q}\right)^{kq} \left(\frac{1}{1-q}\right)^{k(1-q)} \end{aligned} \tag{3.2}$$

If we denote the length of advice strings by b, we know that there are at most 2^b possible advice strings, whereas there are 2^k instances in $\mathcal{I}$. (3.2) gives us an upper bound on the fraction of these two numbers, which we can solve for b and thus obtain the lower bound on the amount of advice required

for c-competitiveness.

$$\frac{2^k}{2^b} \leq \left(\frac{1}{q}\right)^{kq} \left(\frac{1}{1-q}\right)^{k(1-q)}$$

$$k - b \leq kq(-\log q) - k(1-q)\log(1-q)$$

$$k - b \leq -2k\frac{c-1}{c}\log\frac{2(c-1)}{c} - k\left(1 - \frac{2(c-1)}{c}\right)\log\left(1 - \frac{2(c-1)}{c}\right)$$

$$b \geq k + 2k\left(1 - \frac{1}{c}\right)\log\left(2 - \frac{2}{c}\right) + k\left(\frac{2}{c} - 1\right)\log\left(\frac{2}{c} - 1\right)$$

In order for (3.2) to hold, q cannot exceed $1/2$, which gives us the restriction on c for which this bound holds.

$$q \leq \frac{1}{2}$$

$$2\frac{c-1}{c} \leq \frac{1}{2}$$

$$c \leq \frac{4}{3}$$

□

This lower bound is certainly not tight. The bound is a linear function where for $c = 4/3$, the coefficient is 0 and for $c \to 1$, the coefficient approaches $1/2$. See figure 3.3 for a plot of the coefficient with respect to the competitive ratio. However, from theorem 3.5 we know that to achieve optimality, i.e. $c = 1$, we need $L - 1$ bits.

Even though we have not succeeded in proving a better lower bound, we believe it is possible to obtain a tighter bound by generalizing the proof of theorem 3.5. It should be possible to use the same partition tree $T(\mathcal{I})$ as in the aforementioned proof and combine it with the technique used to prove a lower bound for the string guessing problem with known history, as discussed in section 2.1.2.

Each computation of an algorithm in an instance $I \in \mathcal{I}$ can be viewed as a path from the root of $T(\mathcal{I})$. We would need to calculate for each vertex of $T(\mathcal{I})$ the number of instances for which an adversary can force an algorithm A to miss at most e requests.

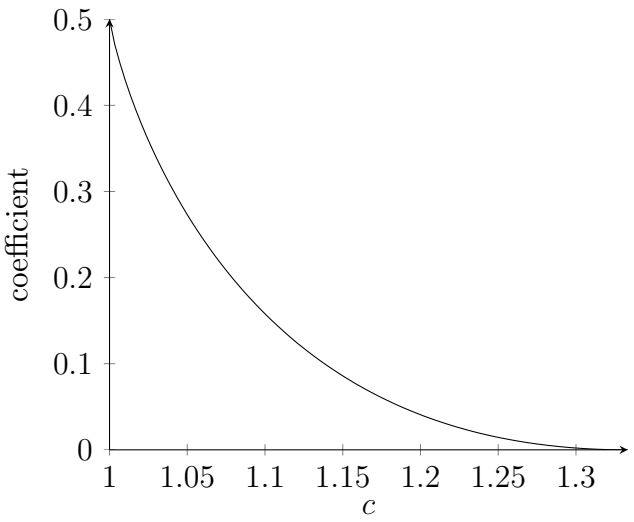

Figure 3.3: *Plot showing the relationship between the competitive ratio and the coefficient in the lower bound in theorem 3.12.*

Unfortunately, due to the complex structure of $T(\mathcal{I})$, we have not managed to obtain such an estimate.

Chapter 4

Offline Algorithms with Advice

The model of algorithms with advice appears to be a useful tool to give a quantitative measure on the difficulty of online problems, which is more fine grained than what competitive analysis alone offers. A new question arises: Is it possible to use advice complexity to quantify the difficulty of hard offline problems?

The concept of a Turing machine with advice is nothing new – it has been introduced in [KL82]. However, the motivation behind the Karp-Lipton model of advice differs significantly from ours – their model is closely tied to Boolean circuits and it requires that given a Turing machine A, for every number $n \in \mathbb{N}$ there be some advice string α_n such that for every input x of size n, $A(x, \alpha_n)$ gives the correct answer.

Our interest lies somewhere else: We want to find out what amount of additional information about a specific input instance can help an algorithm – or a Turing machine – to perform better in terms of time or space complexity. In other words, while Karp and Lipton allow only a single advice string for a given size of input, we allow an advice string specially tailored to each input instance.

The purpose of this chapter is not to give any conclusive results. Instead, our intention is to lay out the groundwork for future research in this area.

4.1 Formal Definition of the Model

There is a wide collection of computational models from which we can choose one to extend with advice. We pick Turing machines due to their status of the de-facto standard computational model.

Definition 4.1 (Turing Machine with Advice)**.** *A* Turing machine with advice *is a deterministic Turing machine A with alphabet $\Sigma = \{0, 1\}$, two read-only tapes and one read-write tape such that one read-only tape contains the input word and the other one contains an infinite binary advice string. A accepts an input word x if there exists an advice string φ such that when the input tape contains x and the advice tape contains φ, A terminates in an accepting state.*

A Turing machine with advice and output *has an additional write-only output tape. Let S be some subset of Σ^* and let $f : S \to \Sigma^*$ be some function. Turing machine A computes the function f if for each $x \in S$ there is an advice string φ such that if the input tape contains x and the advice tape contains φ, A writes $f(x)$ to the output tape and terminates in an accepting state.*

The same considerations as in the case of online algorithms apply here as well. We define the model to use an infinite advice tape in order to avoid giving away any additional information in the form of the length of advice.

This definition resembles nondeterministic Turing machines and in a sense it is equivalent – it is easy to see that one computational step of a nondeterministic Turing machine A with k possible outcomes can be simulated by $\lceil \log k \rceil$ computational steps of a Turing machine B with advice. In each step, B reads one bit of advice and after reading all $\lceil \log k \rceil$ bits, decodes the number k indicating which decision to take. Conversely, we can easily simulate a Turing machine B with advice using a nondeterministic Turing machine A: A can simply nondeterministically fill one tape used to emulate the advice tape with a string of nondeterministically chosen length and then proceed to simulate B without any further changes. The key difference is, as we will

see in a little while, that the model with advice gives us fine-grained control over how much information the machine can obtain nondeterministically.

The definition of a Turing machine with output is somewhat peculiar. Since the result of a computation of a given machine A with a given input x can vary depending on the advice string, it is practically impossible to determine what function A computes. Instead, we choose the opposite direction. In our work we concentrate on problems for which we know what the expected output looks like, i.e. we already have a function f and we want to find a machine such that it can reach the correct output if it receives correct advice.

It is easy to modify the definition of a machine with output to make it better suited for analysis of optimization algorithms. Instead of requiring that the machine computes the correct output, we can define a cost function whose value is ∞ for invalid outputs (or $-\infty$, depending on whether we talk about a minimization or maximization problem) and define the output as the outcome with the minimal (or maximal) cost.

Let us now define the complexity measure which lead to the conception of this computational model in the first place.

Definition 4.2 (Offline Advice Complexity)**.** *The* advice complexity *of Turing machine A with advice is a function $b : \mathbb{N} \to \mathbb{N}$ such that for every input x such that $|x| = n$, A accesses at most $b(n)$ positions on the advice tape.*

This complexity measure is analogous to the measure of space complexity in nondeterministic Turing machines [AB09], except we are only interested in the advice tape; on the work tape, A can use as much space as it needs to. Also note that while the space complexity of a Turing machine is usually defined in a way that allows any multiplicative constant, this is not the case of advice complexity. The reason is that in the case of traditional space complexity, the linear tape compression theorem holds [SHL65], and even though it is possible to represent multiple advice symbols in one position by extending the advice alphabet in a similar way, we are specifically interested in the amount of advice information in terms of the number of bits.

4.2 Related Work

As we have indicated earlier, a different model of Turing machines with advice by Karp and Lipton has been around for more than 30 years. This model has been introduced as a means to provide complexity measures for languages accepted by Boolean circuits. Using this model, various complexity classes of languages have been analyzed and included in known complexity hierarchies, such as **P/poly** or **L/poly**, with important implications on the polynomial hierarchy.

This area of research is, however, very distant from ours. While the purpose of the Karp-Lipton model is to analyze complexity classes, the purpose of our model is to analyze individual problems.

Our research has not revealed any paper focusing on a model equivalent to the one we introduced. This correlates with the fact that thus far, research in this area has been very limited.

One particular area in which similar research has been done is cryptanalysis of the RSA cryptosystem, where Coppersmith has shown that it is possible to find the factors of $N = PQ$ given the high order $1/4 \log N$ bits of P [Cop96]. His method works by reducing the factoring problem to the problem of finding a root of a bivariate integer polynomial. This problem can in turn be solved by constructing a basis of an appropriate lattice and running a basis reduction algorithm in order to obtain a basis consisting of short vectors. These short vectors are then used as candidates for a solution.

In terms of our model of advice complexity, the $1/4 \log N$ bits can be interpreted as advice corresponding to the input N. This translates in an upper bound on the advice complexity of the factoring problem for inputs N such that N is a product of two primes: for any input of length n (i.e. the binary representation of N consists of n bits), $n/4$ bits of advice are sufficient to obtain a solution in polynomial time.

4.3 Subset Sum Problem

We devote the rest of this chapter to an analysis of the *subset sum* problem, which is another example of an NP-hard problem [GJ79]. This problem can be formulated as the following 0-1 integer programming problem.

Definition 4.3 (Subset Sum)**.** *Given a vector* $\overrightarrow{a} = (a_1, \ldots, a_n)$ *of positive integers and a positive integer* M*, find a feasible solution to the 0-1 integer programming problem*

$$\sum_{i=1}^{n} a_i x_i = M; \quad x_i \in \{0, 1\} \text{ for all } i \tag{4.1}$$

This problem is of interest because of its applications in cryptography: multiple asymmetric cryptographic schemes have been proposed based on this problem [MH78; CR88]. These are usually referred to as knapsack-based cryptosystems, since subset sum is a special case of the 0-1 knapsack problem. In these schemes, the public key generally consists of a set of weights $\{a_i \mid 1 \leq i \leq n\}$ and ciphertext is obtained from a plaintext binary message $(b_1, \ldots, b_n)$ as $\sum_{i=1}^{n} a_i b_i$. The private key then consists of additional information which makes it possible to solve instances of the subset sum problem with the given sequence of a_i in polynomial time.

First we present some rudimentary properties of the advice required to solve subset sum and other NP-hard problems in polynomial time and then we look at a more sophisticated algorithm which applies to a specific class of subset sum instances.

4.3.1 General Results

A nondeterministic algorithm for the subset sum problem might guess the values of variables x_i in (4.1) and verify that the guess is correct. This suggests a trivial algorithm with advice: it simply reads the values of x_i from the advice tape, writes them to the output and finishes. Obviously, there is nothing interesting about this algorithm, since it reads the whole output from the advice.

A slightly better algorithm is obtained if we do not read the whole output from advice, but instead stop after reading the k-th bit, i.e. A now knows the values of the first k variables x_i. Then, our algorithm can find the correct values for the remaining $n-k$ varibles x_i by exhaustive search, which takes $O(2^{n-k})$ time. This observation leads to the following claim.

Theorem 4.4. *The subset sum problem can be solved in polynomial time with $n - O(\log n)$ bits of advice.*

Proof. Let $f(n) \leq c \log n$ for some c, let $k = n - c \log n$. A reads k bits of advice and interprets them as the values of $x_1, \ldots, x_k$ from (4.1). A then performs an exhaustive search for the remaining $c \log n$ values $x_{k+1}, \ldots, x_n$, which takes $O(P(n) \cdot 2^{c \log n}) = O(P(n) \cdot n^c)$, where $P(n)$ is a polynomial bound on the time required to verify the correctness of a single vector $(x_1, \ldots, x_n)$. □

This result can be generalized to any problem from NP. This class of problems can be characterized as those where for each input instance, a polynomial-length certificate exists such that the correctness of the certificate can be verified in polynomial time [Cor+09]. If the upper bound on the length of the certificate for a given problem $\mathcal{P}$ is $c(n)$, the sufficient advice to solve $\mathcal{P}$ is $c(n) - O(\log n)$.

We can establish a lower bound on the amount of advice for any NP-hard problem in a similar way: If we can solve some NP-hard problem using $O(\log n)$ bits of advice, it is easy to perform an exhaustive search in polynomial time for the right advice string using a deterministic Turing machine.

Theorem 4.5. *For any NP-hard problem $\mathcal{P}$, the amount of advice required to solve $\mathcal{P}$ is $\omega(\log n)$, unless $P = NP$.*

4.3.2 Lattice-Based Algorithm for Low-Density Subset Sum

Lagarias and Odlyzko proposed in [LO85] an algorithm called SV (Short Vector) which solves almost all instances of subset sum with a low density. The density of a vector $\overrightarrow{a}$ of n elements is defined as

$$d(\overrightarrow{a}) = \frac{n}{\log \max_i a_i}.$$

Informally, instances with a low density are those consisting of elements significantly larger than 2^n.

We formulate the following observation which clarifies the importance of this restricted version of the subset sum problem.

Observation 4.6. *If there is an algorithm A for solving all instances $(\overrightarrow{a}, M)$ of subset sum such that $d(a) < c$ for some $c \in \mathbb{R}^+$ in polynomial time, we can solve any instance of subset sum in polynomial time, regardless of density.*

Proof. Assume we can solve all instances with density lower than c for some $c > 0$. Consider an instance $(\overrightarrow{a}, M)$ such that $d(\overrightarrow{a}) > c$. Let $e = n/c - \log \max_i a_i$. From $d(\overrightarrow{a}) > c$ follows that $e > 0$.

If we multiply both $\overrightarrow{a}$ and M by $2^{\lceil e \rceil + 1}$, we obtain an instance $(\overrightarrow{a}', M')$ with identical solutions to those of $(\overrightarrow{a}, M)$.

For the density of $\overrightarrow{a}'$, the bound

$$\begin{aligned} d(\overrightarrow{a}') &= \frac{n}{\log\left(2^{\lceil e \rceil + 1} \max_i a_i\right)} \leq \frac{n}{e + 1 + \log \max_i a_i} \\ &= \frac{n}{1 + \frac{n}{c} - \log \max_i a_i + \log \max_i a_i} = \frac{cn}{c+n} = c \cdot \left(1 - \frac{c}{n+c}\right) < c \end{aligned}$$

holds, which means $(\overrightarrow{a}', M')$ can be solved in polynomial time in the size of the new instance.

To be entirely correct, we note that the number e is polynomial in n and the size of a binary representation of $(\overrightarrow{a}', M')$ is e times larger than the

binary representation of $(\overrightarrow{a}, M)$, thus this transformation keeps the size of the input polynomial. □

This observation implies that restricting the density does not make the problem significantly easier to solve than it is in the general case. Nevertheless, it does not mean we cannot use any special properties of low-density instances to our advantage.

In the rest of this section, we focus on an analysis of the Lagarias-Odlyzko algorithm. The ultimate goal of this analysis is to better understand the class of instances for which this algorithm fails to find a solution. We believe this should, in theory, make it possible to use less advice for low-density instances, since we can tailor the advice string specifically to the failing instances.

Unfortunately, we have not managed to obtain a sufficient characterization of this class. We describe the results of our statistical analysis of this algorithm and draw some conclusions in hopes they will be useful for future research.

The SV algorithm reduces the problem of finding a solution for (4.1) to the problem of finding the shortest vector in a $n+1$-dimensional lattice with Euclidean norm. This problem is known to be NP-hard for the supremum norm and this is conjectured to hold for the Euclidean norm as well [Emd81]. Nevertheless, a polynomial algorithm for finding short (albeit not necessarily the sortest) vectors in a lattice is due to Lenstra, Lenstra and Lovász [LLL82], commonly referred to as LLL.

The output of the LLL algorithm is a y-reduced basis, where $1/4 \leq y < 1$ is a parameter whose value, unless noted otherwise, is usually $3/4$, containing at least one relatively short vector.

Theorem 4.7 ([LLL82])**.** *Let $[\overrightarrow{v_1}, \ldots, \overrightarrow{v_n}]$ be a y-reduced basis of a lattice L. Then*

$$|\overrightarrow{v_1}|^2 \leq \left(\frac{4}{4y-1}\right)^{n-1} \min_{\overrightarrow{x} \in L, \overrightarrow{x} \neq \overrightarrow{0}} |\overrightarrow{x}|^2, \tag{4.2}$$

where $|\overrightarrow{x}|$ denotes the Euclidean norm of $\overrightarrow{x}$.

Specifically, for $y = 3/4$,

$$|\overrightarrow{v_1}|^2 \leq 2^{n-1} \min_{\overrightarrow{x} \in L, \overrightarrow{x} \neq \overrightarrow{0}} |\overrightarrow{x}|^2. \tag{4.3}$$

The SV algorithm works as follows.

Algorithm 4.8 (SV, [LO85])**.** Let $\overrightarrow{a} = (a_1, \ldots, a_n), M$ be the input.

1. Use the following vectors as a basis $[\overrightarrow{b_1}, \ldots, \overrightarrow{b_{n+1}}]$ of an $n+1$-dimensional integer lattice $L(\overrightarrow{a}, M)$:

$$\begin{aligned} \overrightarrow{b_1} &= (1, 0, \ldots, 0, -a_1) \\ \overrightarrow{b_2} &= (0, 1, \ldots, 0, -a_2) \\ &\vdots \\ \overrightarrow{b_n} &= (0, 0, \ldots, 0, -a_n) \\ \overrightarrow{b_{n+1}} &= (0, 0, \ldots, 0, M). \end{aligned} \tag{4.4}$$

2. Find a reduced basis $[\overrightarrow{b_1}^*, \ldots, \overrightarrow{b_{n+1}}^*]$ of $L(\overrightarrow{a}, M)$ using the LLL algorithm.

3. Check if any $\overrightarrow{b_i}^* = (b^*_{i,1}, \ldots, b^*_{i,n+1})$ has all $b^*_{i,j} \in \{0, \lambda\}$ for some fixed λ for $1 \leq j \leq n$. For any such $\overrightarrow{b_i}^*$, check if $x_j = \frac{1}{\lambda} b^*_{i,j}$ gives a solution to (4.1) and if so, halt, otherwise continue.

4. Repeat steps 1 through 3 with M replaced by $M' = \sum_{i=1}^n a_i - M$, then halt.

The intuition behind this algorithm is that if a vector $\overrightarrow{e}$ is a solution to (4.1), then the vector $\sum_{i=1}^n e_i \overrightarrow{b_i}$ has $-M$ on the last position and $\overrightarrow{e}$ on the first n positions; adding the last vector eliminates the last position. Since the solution vector is binary, it should presumably be one of the shortest vectors in $L(\overrightarrow{a}, M)$.

The SV algorithm has been analyzed thoroughly and it has been shown that the solution to (4.1) is indeed the shortest nonzero vector in the lattice

spanned by vectors b_i from (4.4) for almost all instances such that $d(\overrightarrow{a}) < 0.645\ldots$ [LO85; Cos+92]. As stated earlier, though, the LLL algorithm does not guarantee finding the shortest vector, which further decreases the chance of success of SV.

We have implemented the SV algorithm using an implementation of LLL from the NTL library [Sho]. We then tested this implementation on randomly generated instances of subset sum with varying parameters. We have analyzed the results to find characteristics of those instances that the SV algorithm cannot solve.

In fact, Lagarias and Odlyzko state one possible reason for failure of the SV algorithm in item (4) in the discussion at the end of [LO85]. If the vector $\overrightarrow{a}$ contains many linear dependencies

$$\sum_{i=0}^{n} a_i \lambda_i = 0$$

with $|\overrightarrow{\lambda}|$ small, the lattice $L(\overrightarrow{a}, M)$ will contain the short vector $(\lambda_1, \ldots, \lambda_n, 0)$ for all M. If there are a lot of these small linear dependencies, the lattice reduction algorithm may find those instead of the sought solution vector.

The two important parameters we varied are the dimension n and the density of generated instances. For dimensions, we considered $10, 20, \ldots, 70$, and for densities, we tried $0.3, 0.4, 0.5, 0.6$, and 0.645.

For each combination of these parameters, we generated 100 random vectors $\overrightarrow{a}$ and for each such vector, we sampled 100 random nontrivial subsets (i.e., neither the whole $\overrightarrow{a}$, nor the empty set).

Table 4.1 shows the percentages of random instances solved by the SV algorithm for all combinations of density and dimension. This table clearly shows that the critical density above which the SV algorithm has a very low chance of success decreases rapidly with increasing dimension.

The original article [LO85] contains a proof that if the density is below $0.645\ldots$, the probability of $L(\overrightarrow{a}, M)$ containing a vector shorter than the solution to the subset sum instance approaches zero for $n \to \infty$. This means

Table 4.1: *Percentages of successfully solved random instances for all combinations of density and dimension.*

		density				
		0.300	**0.400**	**0.500**	**0.600**	**0.645**
dimension	**10**	100.00	100.00	100.00	100.00	100.00
	20	100.00	100.00	99.59	92.68	87.61
	30	100.00	97.31	68.19	30.78	18.69
	40	99.51	60.82	9.79	1.14	0.89
	50	81.54	6.03	0.28	0.05	0.02
	60	17.46	0.14	0.00	0.00	0.00
	70	0.77	0.00	0.00	0.00	0.00

that the inability of the SV algorithm to solve high-dimension instances can only be due to LLL not finding the shortest vector.

Indeed, according to theorem 4.7, the upper bound on the lengths of short vectors found by LLL, compared to the shortest vector in the lattice, grows exponentially in the dimension. In other words, the guarantee of the shortness of vectors found by LLL becomes exponentially worse for large n.

A cursory look at some of the reduced bases computed by LLL for the unsolved high-dimension instances rules out the small linear dependency argument stated above. There appear to be very few vectors whose last component is equal to zero.

This observation suggests a possible method to improve the success rate of high-dimension instances. If a good enough estimate of the function of density for which a solution can be found with high probability, depending on n, can be found, and if this function decreases only polynomially, it may be possible to obtain a solution with high probability at a polynomial increase in time complexity by using the reduction described in observation 4.6.

Conclusion

In this thesis we accomplished three separate goals. First, we summarized the most successful methods of analysis of online problems which have been used to obtain better insight on various online problems recently.

Then, we improved an upper bound on the advice required for competitive solutions of the disjoint path allocation problem and we established a lower bound on the advice required for near-optimal solutions as well.

In the final chapter, we introduced a new model of algorithms with advice which makes it possible to study offline problems as well. We offered some rudimentary results which apply to the subset sum problem and other NP problems in general. We showed how the general case of subset sum can be reduced to a class of instances with low density and performed statistical analysis on an algorithm designed specifically for these low-density instances.

We suggest multiple ways in which to extend our research. We note that our lower bound on the advice complexity of competitive solutions for disjoint path allocation is certainly not tight and offer a possible way to improve it.

The bigger area, however, is the newly-introduced model of offline computation with advice in which there has been minimal research thus far and which allows us to look at many problems that are considered to be well-known in a new light.

Bibliography

[AB09] S. Arora and B. Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[ACN96] D. Achlioptas, M. Chrobak, and J. Noga. "Competitive analysis of randomized paging algorithms". In: *Algorithms — ESA '96*. Vol. 1136. Springer Berlin Heidelberg, 1996, pp. 419–430. ISBN: 978-3-540-61680-1.

[Bar+14] K. Barhum, H.-J. Böckenhauer, M. Forišek, H. Gebauer, J. Hromkovič, S. Krug, J. Smula, and B. Steffen. "On the Power of Advice and Randomization for the Disjoint Path Allocation Problem". In: *SOFSEM 2014: Theory and Practice of Computer Science*. Vol. 8327. Springer International Publishing, 2014, pp. 89–101. ISBN: 978-3-319-04297-8.

[BE98] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. New York, NY, USA: Cambridge University Press, 1998. ISBN: 0-521-56392-5.

[Ben+94] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. "On the power of randomization in on-line algorithms". In: *Algorithmica* 11.1 (1994), pp. 2–14. ISSN: 0178-4617.

[Böc+09] H.-J. Böckenhauer, D. Komm, R. Královič, R. Královič, and T. Mömke. "On the Advice Complexity of Online Problems". In: *Proceedings of the 20th International Symposium on Algorithms and Computation*. Honolulu, Hawaii: Springer-Verlag, 2009, pp. 331–340. ISBN: 978-3-642-10630-9.

[Böc+12] H. Böckenhauer, J. Hromkovič, D. Komm, S. Krug, J. Smula, and A. Sprock. "The String Guessing Problem as a Method to Prove Lower Bounds on the Advice Complexity". In: *Electronic Colloquium on Computational Complexity*. 2012.

[Cop96] D. Coppersmith. "Finding a Small Root of a Bivariate Integer Equation; Factoring with High Bits Known". In: *Advances in cryptology — EUROCRYPT'96*. Springer. 1996, pp. 178–189.

[Cor+09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[Cos+92] M. J. Coster, A. Joux, B. A. LaMacchia, A. M. Odlyzko, C.-P. Schnorr, and J. Stern. "Improved low-density subset sum algorithms". In: *Computational Complexity* 2.2 (1992), pp. 111–128.

[CR88] B. Chor and R. L. Rivest. "A knapsack-type public key cryptosystem based on arithmetic in finite fields". In: *IEEE Transactions on Information Theory* 34.5 (1988), pp. 901–909.

[DKP08] S. Dobrev, R. Královič, and D. Pardubská. "How much information about the future is needed?" In: *Proceedings of the 34th conference on Current trends in theory and practice of computer science*. Nový Smokovec, Slovakia: Springer-Verlag, 2008, pp. 247–258. ISBN: 3-540-77565-X, 978-3-540-77565-2.

[DKP09] S. Dobrev, R. Královič, and D. Pardubská. "Measuring the problem-relevant information in input". In: *RAIRO - Theoretical Informatics and Applications* 43 (03 2009), pp. 585–613. ISSN: 1290-385X.

[Emd81] P. van Emde Boas. *Another NP-complete partition problem and the complexity of computing short vectors in a lattice*. Tech. rep. TR 81-04. Mathematics Department, University of Amsterdam, 1981.

[Eme+09] Y. Emek, P. Fraigniaud, A. Korman, and A. Rosén. "Online Computation with Advice". In: *Proceedings of the 36th International Colloquium on Automata, Languages and Programming: Part I*. Rhodes, Greece: Springer-Verlag, 2009, pp. 427–438. ISBN: 978-3-642-02926-4.

[FG06] J. Flum and M. Grohe. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN: 3540299521.

[FKS12] M. Forišek, L. Keller, and M. Steinová. "Advice Complexity of Online Coloring for Paths". In: *Language and Automata Theory and Applications (LATA 2012)*. 2012, pp. 228–239. ISBN: 978-3-642-28331-4.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979. ISBN: 0716710447.

[Her13] P. Herman. "Výskum online problémov z hľadiska advice complexity". Supervised by Michal Forišek. MA thesis. Comenius University in Bratislava, 2013.

[HS94] M. M. Halldórsson and M. Szegedy. "Lower bounds for on-line graph coloring". In: *Theoretical Computer Science* 130.1 (1994), pp. 163–174. ISSN: 0304-3975.

[Kar72] R. M. Karp. "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations*. Springer US, 1972, pp. 85–103. ISBN: 978-1-4684-2003-6.

[Kar92] R. Karp. "On-line algorithms versus off-line algorithms: How much is it worth to know the future". In: *Proc. IFIP 12th World Computer Congress*. Vol. 1. 992. Madrid, Spain, 1992, p. 1.

[KL82] R. M. Karp and R. J. Lipton. "Turing machines that take advice". In: *Enseignement Mathématique* 28.2 (1982), pp. 191–209.

[Kom12] D. Komm. "Advice and Randomization in Online Computation". PhD thesis. ETH Zurich, 2012.

[LLL82] A. K. Lenstra, H. W. Lenstra, and L. Lovász. "Factoring polynomials with rational coefficients". In: *Mathematische Annalen* 261.4 (1982), pp. 515–534.

[LO85] J. C. Lagarias and A. M. Odlyzko. "Solving low-density subset sum problems". In: *Journal of the ACM* 32.1 (1985), pp. 229–246.

[MH78] R. Merkle and M. E. Hellman. “Hiding information and signatures in trapdoor knapsacks”. In: *IEEE Transactions on Information Theory* 24.5 (1978), pp. 525–530.

[SHL65] R. E. Stearns, J. Hartmanis, and P. M. Lewis II. “Hierarchies of memory limited computations”. In: *Sixth Annual Symposium on Switching Circuit Theory and Logical Design*. 1965, pp. 179–190.

[Sho] V. Shoup. *Number Theory C++ Library (NTL)*. URL: `http://www.shoup.net` (visited on 05/04/2014).

[ST85] D. D. Sleator and R. E. Tarjan. “Amortized efficiency of list update and paging rules”. In: *Communications of the ACM* 28.2 (Feb. 1985), pp. 202–208. ISSN: 0001-0782.