THE UNIVERSITY OF CHICAGO

KOLMOGOROV TECHNIQUES
IN COMPUTATIONAL COMPLEXITY THEORY

A DISSERTATION SUBMITTED TO

THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES

IN CANDIDACY FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

SOPHIE LAPLANTE

CHICAGO, ILLINOIS

DECEMBER 1997

ACKNOWLEDGEMENTS

I owe a great debt of gratitude to several people and organizations.

For their financial support I would like to thank the National Science and Engineering Research Council of Canada; the Division of Physical Sciences and the Department of Computer Science of the University of Chicago; and the National Science Foundation by way of Lance Fortnow.

I have learned a great deal from all the faculty at the Department of Computer Science , but I would particularly like to thank László Babai, Lance Fortnow, Stuart Kurtz, Ashish Naik, Janos Simon, and Robert Soare, whose classes in complexity theory and related areas comprise the foundations of my knowledge in the area.

I would also like to thank all my friends in Chicago, for all the fun we had together, but also for their continued support. Amber, André, Barry, Elisabetta, Juffi, Julia, John, Kristian, Manya, Marcus, Mark, Max, Peter, thanks for all the good times.

Thanks also to my coauthors, Joan Feigenbaum, Lance Fortnow, Ashish Naik, and John Rogers. Amber Settle helped me through countless research roadblocks by listening to my ideas and discussing them with me. She and Barry Guiduli were kind enough to carefully read through this thesis and offer many comments and suggestions on how to improve it. Their help was invaluable.

My committee was composed of Lance Fortnow, Stuart Kurtz, and Janos Simon. They guided me through all the hurdles, making the whole process as painless as possible for me. Lance Fortnow, my advisor, has always stood up for me, pushed and prodded me forward, believed in me and gave me the means to complete my goals. In his absence in 1996–97, Stuart Kurtz did the same, advised and encouraged me in all matters. To both of you, thank you for your patience and commitment.

Robert Soare has also been a wonderful source of encouragement over the years, for which I am most grateful.

For my parents and family, for setting such a high moral and intellectual standard; and for Dominique who was there through thick and thin; un gros merci.

TABLE OF CONTENTS

ABSTRACT

The subarea of theoretical computing known as computational complexity is devoted to the study, not of whether or not a problem can be solved, but how efficiently a computable problem can be solved.

While complexity theory studies the difficulty of solving all instances of a given problem with a single program, Kolmogorov complexity is a much more local measure of complexity. The Kolmogorov complexity of a string measures the length of the smallest single-purpose program whose sole task is to print out that string.

The interplay between the two areas of research occurs on many levels. Kolmogorov complexity can be used as a proof technique to solve problems in complexity theory, or it can be viewed as an independent and complementary approach to complexity. The results in this thesis illustrate the diversity of relationships between the two.

In our first group of results, we show that rounds of queries cannot always be traded off for advice, in a bare-bones model of polynomially time-bounded self-reduction. We use the incompressibility technique to show lower bounds on number of rounds of queries in the context of nonuniform computation.

A problem *self-reduces* if having the ability to receive answers to other instances of the same problem makes the problem tractable.

We consider a simple querying model where the only query to which we cannot obtain an answer is the instance that we are trying to solve (that would make the problem uninteresting.) By "tractable," we mean computable in polynomial time. We prove that there are problems which are self-reducible in this model, but which cannot withstand the following restriction to the way queries are made. If the queries are required to be made all at once, that is, if we may not look at some answers before making the next batch of queries, then the problem no longer self-reduces. As stated, this had already been shown by Feigenbaum *et al.* [*Computational Complexity*, 4:158–

174, 1994] but we make a stronger claim. By using Kolmogorov complexity, we are able to show that the problem is not self-reducible even when we let "tractable" mean computable in polynomial time in a nonuniform model of computation (namely, with polynomial-length advice.)

In our second group of results, we exploit the relationship between randomness and Kolmogorov complexity. There is an extensive body of research devoted to finding ways of reducing the amount (and quality) of randomness required by probabilistic algorithms. Among the most successful techniques is the use of so-called extractors graphs, whose special properties can be used to extract "good" randomness out of unreliable sources of random bits.

Our use of extractors allows us to improve upon results in Kolmogorov complexity which are already known to have important applications in computational complexity.

In many applications, it is useful to give a bound on the Kolmogorov complexity of a string that we know lies in a relatively small, easily described set. If we know the set has size s, we could describe the string by saying that it is, say, string number i when the set is listed by increasing lexicographical order. We can always do this using $\log s$ bits.

However we run into problems if the set is rather large and we are concerned with the amount of time spent in recovering the string. Sipser [*STOC* 1983] showed that the string can be recognized *with high probability* in polynomial time with $\log s + O(\log n)$ bits. Buhrman and Fortnow removed the use of randomized algorithms but their solution requires $2\log s + O(\log n)$ bits [*STACS* 1997].

Using current knowledge about extractors, we bring this back down to $\log s + \log^{O(1)} n$, at the price of having the result hold only for almost every string in the set. Our results are subject to improvement if better extractor constructions are found.

In our third group of results, Kolmogorov complexity provides us with an entirely novel viewpoint on computational complexity. The traditional way of describing complexity is to require a single program to solve all instances, then ask how well the program fares as it gets larger and larger instances. It is known, from the work on complexity cores [Orponen and Schöning, *Information and Control*, 70, 1986] that, for some problems, a relatively small set of instances can be responsible for the complexity of the problem. We claim that this phenomenon can be tracked down further: we show that irrespective of the problem we are attempting to solve, there are groups of strings that are virtually impossible to distinguish. Any problem which would require that any such pair of strings be distinguished — by producing a different answer on these instances — will necessarily be hard.

We give a Kolmogorov-style definition for what it means to distinguish two strings. Our measure of complexity is the shortest program which produces a different answer on each string. The definition is very compelling by its simplicity, and we give results that establish that, indeed, there are large groups of strings which are indistinguishable from one another.

We also show how indistinguishability can be used to give lower bounds in communication complexity, a model in which the measure of complexity is neither time nor memory, but rather the amount of communication required if two parties are cooperating to solve a problem.

CHAPTER 1

INTRODUCTION

Rosencrantz: (Impatiently) We have been spinning coins together since I don't know when and in all that time, if it is all that time, one hundred and fifty-seven coins spun consecutively have come down heads one hundred and fifty-seven consecutive times, and all you can do is play with your food.

Tom Stoppard, Rosencrantz and Guildenstern are Dead

1.1 "How Hard is this Problem?"

Computational complexity is the study of the inherent difficulty of computationally solving problems. This is a problem of such a fundamental nature that related questions were posed as far back as in Ancient Egypt, Babylon, and Ancient Greece [KP80]. In his *Elements* (Book 7, ca. 300 B.C.), Euclid gave what was arguably the first algorithm, or formal, precise method, for computationally solving a problem.

Euclid's algorithm finds the greatest common divisor given any pair of numbers. How much effort does it take to compute the greatest common divisor of two numbers, using Euclid's algorithm? A classic analysis tells us that the number of steps required to carry out the calculation is proportional to the logarithm of the larger of the two numbers. This certainly places the problem of finding the greatest common divisor in the range of problems we think of as tractable: the number of steps required is proportional to the number of digits in the numbers to which we are applying the algorithm.

A natural question is: can we do any better than Euclid's algorithm? Could we solve the problem using fewer steps if we had a more clever algorithm? Then the question becomes: How do we demonstrate that beyond a certain point, it is impossible to do any better? These are the starting points of computational complexity theory.

With the advent of the computer in the 1940's, and their proliferation in the last four decades, questions about the possibilities, and also the limits, of computing have gained considerable interest, and the theory of computing has grown into a rich and exciting area of research. Early on, in fact, even before the construction of the first computer, mathematicians put forth formal models of what we intuitively think of as "computation" and "computational processes." Soon thereafter, it was shown that there existed easily-stated, natural problems, which were so inherently difficult that they were in fact uncomputable by any computational process. Further study uncovered the rich and vast structure of problems, classified according to how difficult they are to solve.

The subarea of theoretical computing known as computational complexity is devoted to the study, not of whether or not a problem can be solved, but how efficiently a computable problem can be solved. The measure of efficiency can be the amount of time or the amount of memory or disk space required to carry out a computation, or it can be the amount of randomness needed in order to obtain an answer that we are reasonably sure is correct. These are but a few examples of what can be meant by "complexity."

The ultimate goal of complexity theory is to establish a complete classification of problems according to their difficulty. This turns out to be a very challenging task, and in the thirty or so years since its inception, research in this area has been rich, varied, and deep.

While complexity theory studies the difficulty of solving all instances of a given problem with a single program, Kolmogorov complexity is a much more local measure of complexity. The Kolmogorov complexity of a string measures the length of the smallest single-purpose program whose sole task is to print out that string. It may come as a surprise that such a local measure of complexity could be such a valuable tool in the study of complexity. As it turns out, the applications of Kolmogorov complexity to computational complexity are myriad.

The interplay between the two areas of research occurs on many levels. Kolmogorov complexity can be used as a proof technique to solve problems in complexity theory, or it can be viewed as an independent and complementary approach to complexity. The results in this thesis illustrate the diversity of relationships between the two.

The contributions of this thesis fall into three categories.

First, Chapter 3 contains results in which Kolmogorov complexity is used as a proof technique to obtain lower bounds for a specific model of complexity, namely the number of rounds required in the context of a bare-bones variant of self-reducibility.

Second, in Chapter 4, we study the relationship between Kolmogorov complexity and extractors. Extractors are graphs with certain properties which we will describe later; they have been the subject of extensive research, as they are a very effective tool to reduce the amount of randomness required by randomized algorithms. Our results show how they can be used to obtain upper bounds in a time-bounded variant of Kolmogorov complexity. Bounds of this kind are known to apply to several important results in complexity. We also show, using similar ideas, how to "distill" Kolmogorov complexity efficiently.

Finally, in Chapter 5, we give a new Kolmogorov-based measure of the complexity of distinguishing strings. At first glance, this measure may appear to be too localized to yield any information on the more global, standard approach to complexity. However it provides a very different perspective on complexity, and yields very general lower bounds in communication complexity.

1.1.1 An introduction to computational complexity

The ultimate goal of computational complexity is to establish a definitive hierarchical structure, classifying all natural problems according to their complexity. This is of course a very long-term goal, and progress towards this goal consists of many small, often highly specialized steps. One approach is to consider one important problem, prove a lower bound on the amount of resources necessary to solve the problem, and exhibit an optimal algorithm that matches this lower bound.

Another approach is to consider the nature of complexity on a more abstract, general level, and study its structural properties. The research in this thesis falls into the second category.

Before we begin to reason about complexity, we must first settle on a formal model of what we think of as “computation”. In 1937, Alan Turing [Tur37] published a description of such a model, used to this day in theoretical computer science as one of the fundamental models of computation. In fact, Turing formulated this model years before the construction of the first computer (ENIAC, 1945), but well after Charles Babbage's description of a *Calculating Engine* in 1837 [KP80].

Turing's model, the *Turing Machine*, breaks down the process of calculation into elementary steps. The Turing Machine is equipped with a reading head, positioned on a tape which is separated into squares, each of which can contain one of a small number of symbols. The reading head moves over the tape, one square at a time, according to a small set of rules, called the *transition function*. The head can also keep track of where it is in the calculation by recalling its current *state*. The head at any given state must be in one of a small number of states, and the transition function determines what the state is at the next time step, based on the current state and the character read at the current position on the tape. The transition function also determines which direction the head should move next, and what symbol should be written to the current position of the tape. The contents of the tape before starting the machine is the *input*, an encoding of the particular instance of the problem the Turing machine is purported to solve. The *output* is what is on the tape at the end of the computation, an encoding of the solution.

For example, a simple Turing machine for computing the parity of the number of 1's on the tape could work as follows. The head starts in the state *EVEN*. At each step, if a 0 is encountered, remain in the same state, write the 0 back on the tape, and move to the right. If a 1 is encountered, change the state to *ODD* if it was *EVEN*, or to *EVEN* if it was *ODD*, write the 1 back on the tape, and move to the right. When the end of the input is encountered, change states to *REWIND-EVEN* if the state was *EVEN*, or *REWIND-ODD* if the state was *ODD*. Move back to the beginning of the tape, erasing everything on the way. Once the end of the tape is reached, write 0 on the tape if the state was *REWIND-EVEN*, and 1 if it was *REWIND-ODD*, and stop.

How much effort is required to solve a problem when the Turing machine is the model of computation? The *time* is the number of steps from the beginning of the computation to its conclusion. The *space*, or what we think of as amount of *memory*, is the number of squares of tape that are required in addition to the space used for the input. In the simple example described above, the number of steps was proportional to the length of the input, and the amount of space was nil.

As the example above illustrates, the amount of resources required depends on the instance at hand. The approach of complexity theory is to only take into account the rate of growth of the amount of resources required, as a function of the length of the input. In the example, the rate of growth was linear in the length of the input. Whether it is twice the length, or three times the length is deemed much less important than the fact that it is linear, as opposed to, say, logarithmic, or quadratic, and problems which require linear (or logarithmic, or quadratic) time are grouped together as having the same complexity.

The two most famous complexity classes are the classes P and NP. P is the class of languages which can be solved by a program which runs in polynomial time. The class NP is its nondeterministic counterpart, which consists of the problems which can be solved in polynomial time, but where it is allowed to make *nondeterministic moves* in the course of its computation. A nondeterministic move can be thought of as a point in the program where it can choose to go one of two ways; the program is allowed to make an arbitrary choice and hope for the best. The program solves a language if for every positive instance of the problem, there is a sequence of nondeterministic guesses which leads the machine to accept the instance. If the instance is negative, then all sequences of nondeterministic moves must lead to rejection.

The question of whether P is equal to NP is the best known open problem in complexity theory, yet after decades of research, comparatively little progress has been made on this and other fundamental questions of complexity.

One can ask similar questions about other complexity classes. Randomized algorithms have proven in practice to be more efficient than their deterministic counterparts. Is there a fundamental reason why this is the case, or is it possible that there is a way to transform any randomized algorithm into a deterministic one? Although much effort and energy has been put into answering these questions, still very little is known in terms of collapsing or separating complexity classes, and much work remains to be done in the area.

1.1.2 An introduction to Kolmogorov complexity

Kolmogorov complexity turns its attention to the complexity of individual strings, instead of looking at the overall progression of the difficulty of solving larger and larger instances of a problem. The measure of complexity here is also completely different from computational complexity. Instead of considering how much time, or how much memory, or in general, how much resources are needed to solve instances of a problem, in Kolmogorov complexity, it is the length of the program which is measured.

Let us consider a simple example. Which of the following strings appears to be harder to describe: 00000000000000000000, or 10010011001001111000? In the case of the first string, "20 0s" is a good, very short description. In the second case, the curious reader is challenged to discover as short a description for this string.[1] There is no clear pattern, or structure, that could be exploited to give a short description. Strings with this property, whose shortest description consists of the string itself, are said to have high Kolmogorov complexity.

Certainly, the second string looks more "random" than the first, even though the statistically-minded will argue that both are just as likely to occur if a fair coin

[1] The astute reader will have guessed that it was obtained by flipping a coin 20 times.

were tossed 20 times. But in our heart of hearts, we *know* the first is not random. Kolmogorov complexity provides us with a sound basis for this intuition.

This link between randomness and Kolmogorov complexity is not just happenstance. It can easily be shown that a string chosen at random is very likely to have high Kolmogorov complexity. The relationship between randomness and Kolmogorov complexity will be further explored in Chapter 4.

Even though they differ in such important ways, relating Kolmogorov complexity to computational complexity is not as far-fetched as may initially appear. Consider for a moment an algorithm which performs well on most instances, but which on a relatively small number of instances uses a lot of time and/or space. The Kolmogorov complexity of these bad instances must be small: "27th bad instance for algorithm A" is a sufficient description, potentially much shorter than giving the string especially if it is very long.

It turns out to be the case that the following striking fact is true [LV92]: if the inputs to an algorithm A are chosen according to a distribution, known as the universal distribution, which weighs low-Kolmogorov complexity strings much higher than high-complexity strings, then the average-case behavior of A with respect to this distribution has the same order of magnitude as the *worst case* behavior of A.

Another example of how Kolmogorov complexity contributes to computational complexity is the use of the so-called incompressibility method. This very general proof technique lends itself well to proving lower bounds, as we illustrate in the following example, described in the textbook of Li and Vitányi [LV97]. The name "incompressibility method" derives from the fact that strings with maximal Kolmogorov complexity are called "incompressible".

We claim that there must be infinitely many prime numbers. If not, then assume $p_1, ..., p_\ell$ is the list of all primes. Let s be the number represented by a large string of high Kolmogorov complexity, chosen so that $s > 2^{2^\ell}$. Then s can be represented as a sequence of exponents $e_1, ..., e_\ell$, where $s = p_1^{e_1} ... p_\ell^{e_\ell}$, the prime factorization of s. Each of the exponents is smaller than the logarithm of s, therefore they can be described using $\log\log s$ symbols. In toto, this will require no more than

$\ell \log\log s \leq (\log\log s)^2 << \log s$ symbols. But this contradicts our assumption that s is incompressible, since we have just shown it can be represented with far fewer than the $\log s$ symbols. Therefore our assumption that there were a finite number of primes must have been incorrect.

This example represents only the tip of the iceberg when it comes to showing lower bounds using the incompressibility method. Countless more examples are elaborated upon in the textbook by Li and Vitányi [LV97, Chapter 6]. The technique can be used to show lower bounds for a variety of models of computation, including automata and their variants, parallel random access machines, and boolean circuits.

It is worth noting that because of the (highly) non-uniform nature of Kolmogorov complexity, it is very well suited to proving lower bounds on nonuniform models of computation. Nonuniform models are models where the program used to solve an instance can depend on the particular instance, most commonly on its length, such as boolean circuit families, or Turing machines which take "advice". Our lower bound results in Chapter 3 use this to obtain results in a nonuniform model.

In the discussion above, Kolmogorov complexity was viewed as a tool for proving results about standard computational complexity. Kolmogorov complexity itself, however, can be viewed as offering an alternative and complementary view of complexity, one which considers complexity instance by instance, instead of focusing on the rate of growth of complexity. This is perhaps the least explored connection between the two areas, and has the potential of leading to novel and important insight into what makes problems hard to solve.

The notion of instance complexity, introduced by Orponen, Ko, Schöning, and Watanabe [OKSW94] is the foremost example of how an approach derived from Kolmogorov complexity can shed new light on computational complexity. The instance complexity of a string relative to a problem is the length of the shortest program which solves some, perhaps not all instances of the problem, including the given string. In this respect it is an even more nonuniform measure of complexity than the usual circuit or advice models that have been widely studied. Our work in Chapter 5 pushes this approach even further and yields surprisingly general lower bounds for communication complexity.

1.2 Overview of the contributions and previous work

Our main contributions are outlined below.

1.2.1 Self-reductions

In our first group of results, we show that rounds of queries cannot always be traded off for advice, in a bare-bones model of polynomially time-bounded self-reduction. We use the incompressibility technique to show lower bounds on the number of rounds of queries in the context of nonuniform computation.

A problem *self-reduces* if having the ability to receive answers to other instances of the same problem makes the problem tractable. What we mean by "ability to receive answers" and "tractable" may vary according to the model we choose to study.

We consider a simple querying model where the only query to which we may not obtain an answer is the instance that we are trying to solve (that would make the problem uninteresting.) By "tractable," we mean computable in polynomial time. We prove that there are problems which are self-reducible in this model, but which cannot withstand the following restriction to the way queries are made. If the queries are required to be made all at once, that is, if we may not look at some answers before making the next batch of queries, then the problem no longer self-reduces. As stated, this had already been shown by [FFLS94] but we make a stronger claim. By using Kolmogorov complexity, we are able to show that the problem is not self-reducible even when we let "tractable" mean computable in polynomial time in a nonuniform model of computation (namely, with polynomial-length advice.)

1.2.2 Extractors

In our second group of results, we exploit the relationship between randomness and Kolmogorov complexity. There is an extensive body of research devoted to finding ways of reducing the amount (and quality) of randomness required by probabilistic algorithms. Among the most successful techniques is the use of so-called extractors.

Our use of extractors allows us to improve upon results in Kolmogorov complexity which are already known to have important applications in computational complexity.

In many applications, it is useful to give a bound on the Kolmogorov complexity of a string that we know lies in a relatively small, easy-to-describe set. If we know the set has size s, we could describe the string by saying that it is, say, string number i when the set is listed by increasing lexicographical order. We can always do this using $\log s$ bits.

However we run into problem if the set is rather large and we are concerned with the amount of time spent in recovering the string. Sipser [Sip83] showed that the string can be recognized *with high probability* in polynomial time with $\log s + O(\log n)$ bits. Buhrman and Fortnow removed the use of randomized algorithms but their solution requires $2 \log s + O(\log n)$ bits [BF97].

Using current knowledge about extractors, we bring this back down to $\log s + \log^{O(1)}(n/\varepsilon)$, at the price of having the result hold only for almost every string in the set. Our results are subject to improvement if better extractor constructions are found (up to a theoretical limit of $\log s + O(\log(n/\varepsilon))$).

1.2.3 Indistinguishability

In our third group of results, Kolmogorov complexity provides us with an entirely novel viewpoint on computational complexity. The traditional way of describing complexity is to require a single program to solve all instances, then ask how well the program fares as it gets larger and larger instances. It is known, from the work on complexity cores [OS86] that for some problems, a relatively small set of instances can be responsible for the complexity of the problem. We claim that this phenomenon can be tracked down further: we show that irrespective of the problem we are attempting to solve, there are groups of strings that are virtually impossible to distinguish. Any problem which would require that any such pair of strings be distinguished — by producing a different answer on these instances — will necessarily be hard.

We give a Kolmogorov-style definition for what it means to distinguish two strings. Our measure of complexity is the shortest program which produces a different answer on each string. The definition is very compelling by its simplicity, and we give results that establish that, indeed, there are large groups of strings which are indistinguishable from one another.

We also show how indistinguishability can be used to give lower bounds in communication complexity, a model in which the measure of complexity is neither time nor memory, but rather the amount of communication required if two parties are cooperating to solve a problem.

1.2.4 Collaboration and publication status

The work presented in Chapter 3 on adaptiveness and self-reductions was done in collaboration with Lance Fortnow. It has been published as a University of Chicago technical report [FL94], and is also included in a paper which appeared in the proceedings of the Conference on Complexity, and will appear in the journal Complexity [FFLN96, FFLNar]. Further results described in Chapter 3 were inspired by discussions with Lance Fortnow and Ashish Naik; these do not appear in any other publication.

The results in Chapter 4 on extractors was done in collaboration with Lance Fortnow, and has appeared in a University of Chicago technical report [FL96].

Some of the work in Chapter 5 was done with John Rogers. Most of the results have been published in a University of Chicago technical report [LR96].

CHAPTER 2

PRELIMINARIES

This chapter presents the formal definitions for all of the basic concepts needed throughout this thesis. We begin by presenting the conventions we use for notation, then the definitions and some basic facts about computational complexity, Kolmogorov complexity, and instance complexity. Further definitions which are used only in one chapter can be found in the relevant chapter.

2.1 Notational conventions

We use the following notation throughout the thesis.

- x, y, z are used for finite strings over the alphabet $\Sigma = \{0, 1\}$.
- ϵ is the empty string; ε is used for small positive reals.
- n is for integer values and often is the length of an instance.
- i, j, k are integer indices.
- Φ is a partial computable function, and M_Φ is a Turing machine that computes it.
- p is a string which represents a program, or in some contexts it represents a polynomial.
- $t(\cdot)$ is an integer-to-integer function, usually a time bound.
- $a(\cdot)$ is the length of the advice string in nonuniform computations.

- $q(\cdot)$ is a bound on the number of queries in an oracle computation. q may also stand for a polynomial.

- r is used for random strings.

- Vertical bars are used for set cardinality as well as string length, and the meaning should be clear from the context.

- Strings are sometimes interpreted as the integers they represent.

2.2 Complexity

A few basic notions from computational complexity are used throughout this thesis.

The basic model of computation is the Turing Machine. In most cases we assume a Universal Turing machine, whose input is a pair of strings p, x, where p is the program, and x is the input. When discussing time-bounded computation we assume that the underlying model is a clocked Universal Turing Machine, which stops after a prespecified number of steps, usually given by a function t of the size of the input.

A *language* is a set of strings of finite length. Languages are most often denoted by the uppercase letters $A, B, \ldots$. A string x is an *instance* of language A when we intend to compute whether or not it is one of the strings in the language.

A *complexity class* is a set of languages. Complexity classes group together languages which can be solved using a similar amount of resources. The best-known complexity classes are P and NP, which we will describe below.

Most often, a complexity class is defined by specifying a model of computation, and specifying a bound for one of the computational resources. Any language which can be solved within these parameters is in the class, and every other language is not in the class.

The most frequently used resources are time and space. The time used by a Turing machine is the number of steps it takes before halting with its output on the tape.

Perhaps the most widely accepted definition of what it means for a problem to be tractable is that it can be solved by a Turing machine which requires a polynomial number of steps in the size of the instance, for any instance of the problem. This class is called P, the class of problems decidable in polynomial time.

Definition 1. *For any time bound function t, the class $DTIME[t]$ is the class of all languages L such that L is computable in time t. P is the class of all languages computable in polynomial time by a Turing machine, that is, $P = \bigcup_{c \in N} DTIME[n^c]$.*

A nondeterministic Turing machine is one which is allowed to make *nondeterministic moves* in the course of its computation. Informally, a nondeterministic move is one where the program is offered an alternative, and it picks one and hopes for the best. In formal terms, a nondeterministic Turing machine is one where the transition function is not one-to-one, but rather one-to-many. A nondeterministic Turing machine accepts its input if there is at least one computation path which on that input leads to an accepting state. If there is no path which leads to an accepting state, then it rejects. A language is computable or decidable by a nondeterministic Turing machine if on all instances in the language, the machine accepts, and rejects otherwise.

Definition 2. *For any time bound function t, the class $NTIME[t]$ is the class of all languages L such that L is computable in time t by a nondeterministic Turing machine. NP is the class of all languages computable in nondeterministic polynomial time, that is, $NP = \bigcup_{c \in N} NTIME[n^c]$.*

Turing machines may also be equipped with an additional resource, called *advice*. This is a device which allows Turing machines to be *nonuniform*. By uniformity, it is generally meant that a single machine, with a fixed definition, can handle any instance of a problem. By contrast, a nonuniform model allows the computational device to depend in some way on the instance that it is trying to solve. In the case of Turing machines which take advice, a single program, or Turing machine, is used, but depending on the length of the input, it can rely on getting some additional information in the form of an *advice string*.

Formally, a Turing machine which takes advice is equipped with an additional tape, and it is assumed that an advice string which depends only on the length of the input is written on the tape before the computation begins. If the length of the advice string is bounded by a function a, then the computation is said to take $a(n)$ advice. The class of languages which are computable in polynomial time with polynomial advice is written $P/poly$.

2.3 Kolmogorov complexity

Kolmogorov complexity, as we described in Section 1.1.2, was introduced as a measure of the randomness contained in a string. The notion formalizes the intuition that the string 10010011001001111000 is "more random" than 00000000000000000000. The Kolmogorov complexity of a string x is defined to be the length of the shortest program which prints out x.

To make this definition precise, we must specify what we mean by *program*. Fix a partial computable function Φ. Assume that Φ takes two arguments (if not, use a standard pairing function.) The first argument is taken to be a program and the second, the input for that program.

Definition 3. *Fix a partial computable (also called partial recursive) function Φ. For any pair of strings $x, y \in \{0,1\}^*$, the Kolmogorov complexity of x relative to y with respect to Φ is defined as*

$$C_\Phi(x|y) = Min\{|p| : \Phi(p, y) = x\}.$$

The invariance theorem stated below allows us to abstract away from a particular partial computable function Φ. It states that there is a so-called *universal function* which captures the complexity of strings with respect to an arbitrary, fixed, partial computable function, up to an additive constant.

Proposition 2.1 (Invariance). *There is a universal function Φ_0 such that for any partial computable function Φ, there is a constant c_Φ, for which*

$$C_{\Phi_0}(x|y) \leq C_\Phi(x|y) + c_\Phi, \text{ for all } x, y.$$

For the remainder of this discussion, we fix Φ_0 and write $C(x|y)$ instead of $C_{\Phi_0}(x|y)$. When $y = \epsilon$, the empty string, we simply write $C(x)$.

Proposition 2.2. *For every n, there is a string x of length n with $C(x) \geq n$.*

This can be shown using a simple pigeonhole argument. Fix n. There are 2^n strings x which have length n. On the other hand there are only $2^{n-1} + 1$ programs of length less than n. Each program can only print out a single string, so there must be at least one string of length n which none of these programs prints out.

A string x with this property is called *incompressible*. There are countless examples of proofs which use the existence of incompressible strings at the center of their arguments. Many can be found in the invaluable reference by Li and Vitányí [LV97, Chapter 6]. Several applications include fundamental results in complexity theory, such as the celebrated Håstad Lemma, used to prove that computing parity with a constant-depth circuit requires exponentially many gates [Yao85, Hås89, FL95],[LV97, Section 6.12].

There are also strings of high Kolmogorov complexity in any large set. The following proposition formalizes this idea.

Proposition 2.3. *For every $y \in \{0,1\}^*$ and every $A \subseteq \{0,1\}^*$ with $|A| = m$, there is a string $x \in A$ such that $C(x|y) \geq \log m$.*

In fact, the following more general statement also holds.

Proposition 2.4. *For any positive integer c, any y, and any $A \subseteq \{0,1\}^*$ with $|A| = m$, the number of strings $x \in A$ with $C(x|y) \geq \log m - c$ is at least $m(1 - 2^{-c})$.*

2.4 Time-bounded Kolmogorov complexity

In standard Kolmogorov complexity, the amount of time or space that it takes to print out a string is unimportant. However it is useful to formulate Kolmogorov complexity of strings in terms of resource-bounded computation. We will only consider time-bounded computation in this thesis, but space-bounded Kolmogorov complexity can be defined analogously.

2.4.1 Printing complexity

The *time-bounded Kolmogorov complexity* of a string x relative to a string y, written $C^t(x|y)$, is the length of the shortest program p which on input y, prints out the string x in time bounded by $t(|p| + |x| + |y|)$. When y is the empty string, we write $C^t(x)$.

Definition 4. *Fix a partial computable (also called partial recursive) function Φ, which is computed by a Turing machine M_Φ. Also fix a time bound t. For any pair of strings $x, y \in \{0,1\}^*$, the t-time bounded Kolmogorov complexity of x relative to y with respect to Φ is defined as*

$$C_\Phi(x|y) = Min\{|p| : M_\Phi(p, y) = x \text{ and } M_\Phi \text{ halts in fewer than } t(|p| + |y|) \text{ steps}\}.$$

As for unbounded Kolmogorov complexity, it is possible to state an invariance theorem, and incompressibility results. They are completely analogous to the ones in the previous sections.

2.4.2 Distinguishing complexity

In the context of time-bounded computation, it makes sense to make the distinction between programs that print out a string x and those that accept exactly the string x. The latter is the object of CD complexity, or distinguishing complexity. In the case of standard Kolmogorov complexity, the two measures would coincide, since, time being of no object, one could run through all possible strings until one is found on which the program accepts, and print that one out. This is of course not necessarily possible if time is constrained.

Definition 5. *Fix a universal function Φ which is computed by a Turing machine M_Φ, equipped with a clock so that it runs in time bounded by $t(|p| + |x| + |y|)$, and whose output is 0 or 1. For any pair of strings $x, y \in \{0,1\}^*$, the t-time-bounded Kolmogorov distinguishing complexity of x relative to y is defined as*

$$CD^t_\Phi(x|y) = Min\{|p| : \Phi(p, y, z) = 1 \textit{ iff } z = x\}.$$

The *nondeterministic time-bounded Kolmogorov distinguishing complexity* of a string x relative to a string y, written $CND^t(x|y)$, is the length of the shortest program p which has the following behavior. If $z = x$, then there is a witness w of length bounded by $t(|x|)$ such that p accepts on input y, z, w. If $z \neq x$, then no such witness can cause p to accept. The time taken on all inputs y, z, w must be bounded by $t(|p| + |y| + |z| + |w|)$. When y is the empty string, we write $CND^t(x)$.

2.5 Instance complexity

Instance complexity [OKSW94] studies the complexity of solving specific instances of a language A. The measure used is the length of the shortest "special-purpose" program which determines correctly whether the given instance is, or is not, in A. By "special-purpose," we mean that the only programs considered are the ones which always output one of 0,1, or $\perp$, and are *consistent* with A, meaning when the output is not $\perp$ ("I don't know!"), the answer must be correct.

Definition 6. *[OKSW94] Fix a universal Turing machine Φ with output symbols $\{0, 1, \perp\}$. Fix also a computable (also called recursive) set A.*

1. *A Φ-program p is* consistent *with A if for all x, when $\Phi(p, x) \neq \perp$ then in fact $\Phi(p, x) = \chi_A(x)$.*

2. *The* instance complexity *of x relative to the set A, written $ic(x : A)$, is the length of the shortest program p consistent with A and for which $\Phi(p, x) \neq \perp$.*

As with Kolmogorov complexity, one can define in the usual way resource-bounded instance complexity. If the underlying Turing machines run in time bounded by $t(|x|)$ for every instance x then the instance complexity is denoted $ic^t(x : A)$.

Instance complexity captures in a very compelling and elegant way a very general notion of nonuniformity. Turing machines which take advice can be thought of as a set of consistent programs for A, each of which solves exactly all the instances of a given length. The following result of Orponen, Ko, Schöning and Watanabe establishes a very clear relationship between classes based on instance complexity and the "standard" advice classes.

Definition 7. *The class $IC[f(n), t(n)]$ is the set of languages A such that for every instance x of A, $ic^t(x : A) \leq f(|x|)$.*

Theorem 1 ([OKSW94]). *The class $IC[log, poly]$ lies strictly between P/log and $P/poly$.*

In other words, logarithmic advice is strictly weaker than logarithmic length programs, which in turn is strictly weaker than polynomial advice.

CHAPTER 3

ADAPTIVENESS VS. ADVICE IN COHERENCE

3.1 Self-reductions

Self-reducibility is a structural property which can be thought of as a measure the amount of redundant information in a set. The amount of information about a particular instance is contained in other instances, and more importantly, how easy it is to use this information to solve a given instance, yields useful information on the structure of the set.

A set is self-reducible if its characteristic function can be computed easily when access is given to an oracle which can compute any instance of the same set. Of course this only makes sense if access to this oracle is restricted in some way, and several variants of self-reducibility have been put forth and studied extensively.

3.1.1 An overview of self-reductions

In the simplest scenario, the only restriction made on the access to the oracle is that to solve an instance x of a language L, any query can be made except x itself. When a set is self-reducible in this sense, it is said to be *coherent*. This definition was put forth by A.C.-C. Yao [Yao90] in an attempt to capture the weakest form of self-reducibility.

One can consider additional restrictions, such as requiring that the queries be made in a limited number of *rounds*. The queries in any given round can only depend on the answers given to queries in previous rounds. In particular, if all the queries are made in a single round (in parallel), then the reduction is said to be *non-adaptive*.

Another variant of self-reductions is to provide the algorithm with a string of advice that can be used to help solve any instance of a given length. A reduction which uses such advice is called *non-uniform*.

In the presence of all these variants, it is important to understand the relationships among them. What we would really like to know is whether or not they capture different classes of sets, or if in some cases, weakening the access restrictions does not buy us any additional computational ability.

3.2 Contributions

We begin by providing the necessary formal definitions, then state and prove our main results.

3.2.1 Definitions and preliminaries

Informally, a function is *coherent* if any instance can be computed efficiently provided one is given access to an oracle for the function which can be queried on any other instance.

We say that a computation *takes polynomial advice* if $h : N \mapsto \{0,1\}^*$ is such that, for some polynomial $a(n)$, $|h(n)| \leq a(n)$, and, for each x, the computation is provided with $h(|x|)$ as auxiliary input. The function h is called the *advice function*, and $h(n)$ is called the *advice string* for length n. Our definition of computations that take advice is based on the original definition by Karp and Lipton [KL80].

Definition 8. *A function f is said to be* **coherent** *if, for every $\varepsilon > 0$, there is a probabilistic polynomial-time Turing machine E called the* **examiner** *that, on input x, can query an f-oracle multiple times to obtain the value of f at any point except x itself. The machine E computes $f(x)$ with probability at least $1 - \varepsilon$. We say that f is* **nonadaptively weakly coherent** *if E takes polynomial advice and makes only nonadaptive queries to its oracle.*

We can reduce the probability of error ε to exponentially small by the usual trick to running the algorithms for a large number of parallel rounds and taking the plurality of the answers.

3.2.2 Adaptiveness vs. advice for languages

The first main contribution of this chapter establishes that there are functions which are coherent but for which adaptiveness is required, even if polynomial advice is provided.

Theorem 2. *There exists a function $f : \{0,1\}^* \longrightarrow \{0,1\}$ that is adaptively coherent but not nonadaptively weakly coherent.*

We will use the following proposition, which states that if an examiner is allowed advice, we may assume that it is deterministic.

Proposition 3.1. *If E is a nonadaptive examiner that uses an advice string of length $s(n)$, then E can be replaced by a deterministic nonadaptive examiner that uses an advice string of length $s(n)+p(n)$, where $p(n)$ is a polynomial.*

This follows from a standard argument using Chernoff bounds, whereby a polynomial number of "good" coin flips can be encoded into the advice string.

Define $t(m)$ to be a tower of m 2's, i.e., $t(0) = 1$ and, for all $m \geq 1$, $t(m) = 2^{t(m-1)}$. Consider a function f that is zero on all inputs except those of length $n = t(m)$ for some natural number m. Such integers $n = t(m)$ are called *allowed lengths*. The restriction of f to strings of an allowed length n, denoted by f_n, is defined using the following scheme.

To simplify notation, we interpret n-bit strings as integers in the range $[2^n, 2^{n+1}-1]$. The domain $[2^n, 2^{n+1}-1]$ is divided into $B_n = 2^{n/4}$ equal "blocks," so that the ith block spans the interval $[2^n + (i-1)\frac{2^n}{B_n}, 2^n + i\frac{2^n}{B_n} - 1]$. The first value in the ith block is called the *boundary value* for block i and is denoted by

$BV_{n,i} = 2^n + (i-1)\frac{2^n}{B_n}$. Let $s_{n,1}, \ldots, s_{n,B_n}$ be arbitrary but fixed values between 1 and $\frac{2^n}{B_n}$.

On each block, define $f_n|_{[BV_{n,i}, BV_{n,i+1}-1]} = f_{n,i}$ as follows:

$$f_{n,i}(x) = \begin{cases} s_{n,i} \ mod\ 2 & \text{if } x = BV_{n,i} \\ 0 & \text{if } x < BV_{n,i} + s_{n,i} \\ 1 & \text{if } x \geq BV_{n,i} + s_{n,i} \end{cases}$$

We call $s_{n,i}$ the "step" of the ith block for length n. Note that each possible set of steps $\mathcal{S} = \{s_{n,i} | m \in N, n = t(m), 1 \leq i \leq B_n\}$ completely determines a function f. Let $\mathcal{F}$ denote the set of functions each defined by a set of steps as described above.

We first prove that all functions $f \in \mathcal{F}$ are adaptively coherent; then we prove that there exists a function $h \in \mathcal{F}$ that has no nonadaptive examiner even when provided with polynomial advice.

Lemma 3.1. *All functions $f \in \mathcal{F}$ are adaptively coherent.*

We shall exhibit an adaptive examiner for f. Given an input x whose length n is not an allowed length, the examiner always returns zero. Consider $n = t(m)$ for some m. There are three possibilities:

1. If $x = BV_{n,i}$ for some i, then the examiner uses binary search to find the value of $s_{n,i}$ and outputs the parity of this value.

2. If x lies in the interval $[BV_{n,i} + 1, BV_{n,i+1} - 1]$ and oracle queries using its two immediate neighbors as inputs produce the same value, then the examiner outputs this value.

3. If the values of f on the neighbors of x differ, then x is either $BV_{n,i} + s_{n,i} - 1$ or $BV_{n,i} + s_{n,i}$. A query to $BV_{n,i}$ distinguishes these two cases, and the examiner returns zero in the first case and one in the second. □

Next, we define the function $h \in \mathcal{F}$ as follows. For each allowed length n, let s_n denote an incompressible string of length $B_n \log \frac{2^n}{B_n} = \frac{3n}{4} B_n$. Suppose that

$s_n = s_{n,1} \cdot s_{n,2} \cdots s_{n,B_n}$ where, for all i, $s_{n,i} \in \{0,1\}^{\frac{3n}{4}}$. Let h_n be the function in $\mathcal{F}$ obtained by taking the strings $s_{n,1} \ldots, s_{n,B_n}$ as the steps of length n. Since h_n can be fully constructed from s_n and vice-versa, we write that $C(h_n|n) \geq C(h_n|\ |s_n|) \geq |s_n| = B_n \log \frac{2^n}{B_n}$.

Lemma 3.2. *The function h is not nonadaptively coherent, even with an examiner that takes polynomial advice.*

Assume, by way of contradiction, that there exists a deterministic polynomial-time examiner E for h.

Note that, for sufficiently large n, because h is nonzero only on allowed lengths, we can assume that the only non-trivial queries made by E on inputs of length n must also have length n. The next allowed length is exponential in n, and therefore queries of that length cannot be constructed in the time available. We are also able to assume that the values of the function at all the smaller allowed lengths are encoded into the examiner's advice string.

Let E compute h using an advice string of length bounded by a polynomial $a(n)$ and making at most $q(n)$ nonadaptive queries for some polynomial q. To simplify notation, we make the following definition for $\tilde{s}_{n,i}$, which is $s_{n,i}$ with the last bit set to 0:

$$\tilde{s}_{n,i} = \begin{cases} s_{n,i} & \text{if } s_{n,i} \text{ is even} \\ s_{n,i} - 1 & \text{otherwise} \end{cases}$$

We focus on the behavior of E on all the boundary values, i.e., on inputs $x = BV_{n,i}$ for $1 \leq i \leq B_n$ (where $n = t(m)$ for some m). We show that, if E queries too many inputs of the form $BV_{n,i} + \tilde{s}_{n,i}$, we can shorten the description of the function by encoding $\tilde{s}_{n,i}$ with an index to one of E's (polynomially many) queries. On the other hand, if the examiner does not query enough inputs of the form $BV_{n,i} + \tilde{s}_{n,i}$, we can shorten the description of the function by running the examiner with partial information about the last bit of some of the steps $s_{n,i}$. Since the description of h is chosen to be incompressible, we get a contradiction.

Let α be the number of intervals i for which E queries $BV_{n,i} + \tilde{s}_{n,i}$ when its input is one of the boundary values $BV_{n,j}$ for some j.

Claim 3.1. $\alpha \leq \frac{a(n)+c}{\frac{n}{2}-\log q(n)-1}$.

To prove this claim, consider the following description of h_n.

- A list of the values of all the $B_n - \alpha$ steps $s_{n,i}$ for which $BV_{n,i} + \tilde{s}_{n,i}$ does not get queried by E on any of the boundary values.
- For the remaining α steps $s_{n,i}$, a triple consisting of an index to a block j in which E, on input $BV_{n,j}$, queries $BV_{n,i} + \tilde{s}_{n,i}$, the index to the query itself, and the last bit of $s_{n,i}$.
- E's algorithm and advice string.

Note that a program that encodes the above description can be used to construct h_n. Thus,

$$\begin{aligned} C(h_n|n) \;\leq\; & \alpha(\log B_n + \log q(n) + 1) \\ & +(B_n - \alpha)(\log \frac{2^n}{B_n}) + a(n) + c. \end{aligned}$$

Using the fact that $C(h_n|n) \geq B_n \log \frac{2^n}{B_n}$, simple algebraic manipulation yields the claimed upper bound on α, concluding the proof of Claim 3.1. □

Claim 3.2. $\alpha \geq 2^{n/4} - a(n) - c$.

To get the claimed lower bound on α, consider the following description of h_n: For the $B_n - \alpha$ blocks i such that $BV_{n,i} + \tilde{s}_{n,i}$ is never queried, encode all but the last bit of each of these strings, and encode the remaining α strings in their entirety. The last bit of each of the $B_n - \alpha$ values can be constructed by simulating the examiner E on all B_n boundary values. Note that, since the last bits of the $B_n - \alpha$ values are never queried by E, with all the above information, E can be simulated unambiguously. Using this description of h_n, we get the following upper bound on $C(h_n \mid n)$.

$$\begin{aligned} C(h_n|n) &\leq (B_n - \alpha)(\log \frac{2^n}{B_n} - 1) + \alpha(\log \frac{2^n}{B_n}) + a(n) + c \\ &\leq B_n \log \frac{2^n}{B_n} - (B_n - \alpha) + a(n) + c. \end{aligned}$$

Using the fact that $C(h_n|n) \geq B_n \log \frac{2^n}{B_n}$, simple algebraic manipulation now yields the bound on α. $\square$

Clearly, if n is large enough, Claims 3.1 and 3.2 cannot simultaneously hold, and so a nonadaptive examiner cannot exist for h, even when allowed polynomial advice. With this we have proven Lemma 3.2 and Theorem 2. $\square$

3.2.3 Rounds vs. advice for functions

Our second contribution is to show that there are functions that are coherent using a $k+1$-round reduction, but for which there is no k-round examiner, even with polynomial advice.

This refines the previous result in that it shows that one round of queries cannot be traded off for polynomial-length advice, whereas the proof above shows that one cannot go from a polynomial number of rounds to a single round in exchange for polynomial advice. We only prove the second result holds for functions and it is an open problem whether or not it holds for languages as well.

Theorem 3. *For every constant k, there is a function which is $(k+1)$-round-coherent but not k-round coherent, even with polynomial advice.*

Proof. For every allowed string length n (n = tower of twos) take an incompressible string σ of length $\log((2^n-1)!)$, representing a cycle through the 2^n strings of length n.

The successor of x in the cycle is denoted $s(x)$. The idea of the construction is to force an examiner to go through k successors of x to get the full value of the function on that input.

For every other (non-allowed) string length, the value of the function on any input of that length is 0. This way we may assume that the only queries made by the examiner have the same string length: the examiner does not have enough time in polynomial time to formulate a query at the next allowed length; furthermore the value of the function at all the smaller allowed lengths can be encoded into the polynomial length advice string; finally, the answer to any query at a non-allowed length is 0.

The inputs will not actually be x as above but $x.0$ and $x.1$, where $a.b$ denotes concatenation. The notation $[z]_i$ is used to mean the ith "piece" of the string z, where z is viewed as being split into k substrings of (roughly) equal length. The function f is defined on allowed lengths to be:

$$\begin{aligned} f(x.1) &= s(x) \\ f(x.0) &= s(x), [s(s(x))]_1, ..., [s^{k+1}(x)]_k \end{aligned}$$

The idea is that each successor address is broken into k equal-size pieces and divided up among the k inputs as "proof" that the query was made to this successor.

Clearly f is $(k+1)$-round coherent: on $x.1$ one query suffices (namely $x.0$); on $x.0$, query $x.1$ to get the first successor, and so on.

We show there is no k-round protocol by contradiction. We assume that there is a k-round protocol and then show that the original string can be compressed by more than any polynomial. This implies that even with poly-length advice, there is no k-round examiner.

For every x, one of 2 cases must occur when the examiner computes $f(x.0)$. To avoid excessive notation, when referring to a query made to either $z.0$ or $z.1$, we simply write z. In all cases it does not matter if the trailing bit is 0 or 1.

Case 1: Some $s^i(x)$ is not queried. Then the examiner produces as part of the output the ith piece of $s^{i+1}(x)$ and this could be omitted in the description of the original string.

Case 2: All of the k successors are queried, therefore there is a j for which $s^j(x)$ is queried in a round $i < j$. Choose i to be the minimal round at which this occurs, and j to be minimal among the $s^j(x), j > i$ queried at round i. By minimality, $s^{j-1}(x).1$ cannot have been queried before this round; so the value of the function on $s^{j-1}(x)$ could be omitted from the description of the original string. Note that we must assume that the location of s^{j-1} is known, but this assumption can be made without loss of generality.

It still remains to be shown that a large number of these potential savings can indeed be realized.

Claim 3.3. *Cases 1 and 2 above cover $\Omega(2^n)$ many values.*

Proof. Consider only the x whose position in the cycle (those i such that $s^{(i)}(0^n) = x$) is congruent to 0 modulo $k + 1$. Each such x produces an occurrence of Case 1 or Case 2 which is distinct from any other occurrence due to a different $y = s^{(j)}(0^n)$ for which $j \equiv 0 \pmod{k+1}$. Therefore the number of distinct values targeted for omission is at least $\frac{2^n}{k+1}$. □

Claim 3.4. *At least $\frac{\Omega(2^n)}{n^{O(1)}}$ of the values covered by Claim 3.3 can be omitted from the description of the original string σ.*

Proof. Since at most $n^{O(1)}$ many queries may be required in order to omit a piece of information from the description of the original string, the process can be repeated at least $\frac{2^{\Omega(n)}}{n^{O(1)}}$ many times before all potential savings are exhausted. □

In light of Claims 3.3 and 3.4, we need to check that the original string can be reconstructed using the examiner's algorithm, the advice, and a stream of additional information whose length is small compared to the length of σ. Details are relatively straightforward, and a sketch follows.

The auxiliary stream consists of 2 parts. One is a stream of answers to queries that E makes on $x_1.0, x_2.0, \ldots$ (an enumeration of all strings of length n) which have not already been given or reconstructed; the other is a stream of indices to queries which correspond to an occurrence of Case 2.

Since one of Cases 1, 2 must occur, the reconstruction algorithm will assume Case 1 occurs and if it does not, only then does it consider Case 2 and get the corresponding index from the stream.

Thus the length of the first stream is $n(2^n-1) - \frac{n}{4}\frac{2^n}{k+1}$ and the second is $\log q(n)\frac{2^n}{k+1}$, where q is the polynomial that bounds the number of queries made by E.

We have chosen the string σ such that

$$C(\sigma) \geq \log(2^n-1)!.$$

Using Stirling's approximation,

$$\begin{aligned}
C(\sigma) &\geq \log\left(\sqrt{2\pi(2^n-1)}(\frac{2^n-1}{e})^{(2^n-1)}\right) \\
&\geq \log\left(\sqrt{2\pi}\right) + \frac{n-1}{2} + 2^n - 1\log\frac{2^{n-1}}{e} \\
&\geq \log\left(\sqrt{2\pi}\right) + \frac{n-1}{2} + (2^n-1)(n-1-\log e) \\
&\geq (2^n-1)(n-1-\log e)
\end{aligned}$$

On the other hand, our argument above shows that

$$\begin{aligned}
C(\sigma) &\leq n(2^n-1) - \frac{n}{4}\frac{2^n}{k+1} + \log q(n)\frac{2^n}{k+1} + c + a(n) \\
&= n(2^n-1) - \frac{2^n}{k+1}\Big(\frac{n}{4} - \log q(n)\Big) + c + a(n)
\end{aligned}$$

where c represents the length of the examiner and the program to simulate the examiner, and $a(n)$ is the polynomial that bounds the length of the advice string.

Therefore it must be the case that

$$(2^n-1)(n-1-\log e) \leq n(2^n-1) - \frac{2^n}{k+1}\Big(\frac{n}{4} - \log q(n)\Big) + c + a(n)$$

which is a contradiction for large enough n.

$\square$

3.3 Open questions

There are still several unanswered questions about adaptiveness in self-reductions. In Theorem 2 we have shown that for languages, adaptiveness cannot be traded off for advice. Theorem 3 states that not a single round of queries can be traded off for advice, but this only holds for functions. It would be useful to see if these results can be extended to show that not a single round of queries can be traded off for advice, for languages instead of functions.

Another challenging question is whether an analogue of these results holds for random-self-reductions. In this model of self-reducibility, additional constraints are made on the way queries are made so that the distribution of the queries made does not reveal the instance at hand. It is already known that there are adaptive random-self-reducible problems which are not nonadaptively random-self-reducible [FFLS94], but a proof of a similar statement in the presence of advice remains elusive.

CHAPTER 4

EXTRACTORS AND DISTINGUISHING COMPLEXITY

4.1 Preliminaries

4.1.1 Distinguishing Complexity

Distinguishing complexity, a variant of time-bounded Kolmogorov complexity, was put forth in 1983 by Sipser [Sip83], who used it to show that BPP is in the polynomial hierarchy. Although distinguishing, or CD, complexity has been the object of much research since then, only recently has it become apparent what central a role it plays in computational complexity.

Recall from Chapter 2 that the CD complexity of a string, given a time bound t, is the length of the smallest program which can recognize that string, and that string only, in time t.

As a recent paper of Buhrman and Fortnow has established [BF97], techniques similar to those used by Sipser can be used to show many important results in complexity. Applications include questions about computing and isolating satisfying assignments. They also give illuminating new proofs of oracle results showing that there are relativized worlds where $EXP = NEXP$ but where accepting paths in a $NEXP$ machine cannot always be found in exponential time, a result originally proven by Impagliazzo and Tardos [IT89]. Another important and widely-used consequence is the theorem of Valiant and Vazirani [VV86], which gives a randomized reduction of satisfiability to unique satisfiability.

All of these applications can be boiled down to the following question: How much information is needed to distinguish a given string from among all the strings in a given set?

In the standard Kolmogorov setting, this question can be answered by the following fundamental lemma.

Lemma 4.1. *Let A be a computable set. For all n and for all $x \in A \cap \Sigma^n$,*

$$C(x) \leq \log |A \cap \Sigma^n| + O(\log n)$$

The reason that this is true is of course that the rank (with respect to standard lexicographic ordering) of the string x among all the strings in A of length n is enough to distinguish it from all other strings in that set, of that length.

Unfortunately, this is likely to hold only when no limits are imposed on the amount of time required to recover x. If all the strings in the set A must be enumerated until the desired string is found, this may take exponential time. Can we do better?

Sipser was the first to address this question. He gave the following answer, which requires the use of a polynomially-long random string:

Theorem 4 (Sipser). *For every set $A \in \mathcal{P}$, there is a polynomial p and a constant c such that for every n and for most r of length $p(n)$, and for every $x \in A \cap \Sigma^n$,*

$$CD^p(x|r) \leq \log(|A \cap \Sigma^n|) + c \log n.$$

Buhrman and Fortnow showed how to eliminate r at the cost of doubling the complexity.

Theorem 5 (Buhrman-Fortnow). *For any set $A \in \mathcal{P}$, there is a polynomial p and a constant c such that for all strings $x \in A \cap \Sigma^n$,*

$$CD^p(x) \leq 2 \log(|A \cap \Sigma^n|) + c \log n.$$

In this chapter, we extend the work of Buhrman and Fortnow [BF97] and obtain nearly the bound of Sipser [Sip83] without the random string used by Sipser. However, our result only works for most strings in A.

4.1.2 Extractors

An *extractor* is a bipartite graph whose properties are used to extract "good" randomness from a "bad" random distribution, using only a small number of truly random bits in the process. Here, "good" means an almost-uniform distribution.

A lot of research has gone into finding good constructions for extractor graphs. These graphs have been shown to be centrally important in reducing and even eliminating the dependency on randomness of randomized algorithms. We use this technical machinery for a completely different purpose, that is, improving the bounds in Sipser-style lemmas described in the previous section.

An extractor graph has two color classes, which we designate as the left and right parts of the graph. The left side has $N = 2^n$ vertices, each of which is thought of as a point in a sample space associated with a fixed (arbitrary) distribution. The right side has $M = 2^m$ vertices, typically much fewer than the left side. The goal is to produce a good distribution on the vertices on the right by choosing a vertex on the left according to the associated distribution, and choosing one of the edges adjacent with this vertex using a uniform, independent distribution.

It is clear that the bigger the right side and the smaller the maximum degree of any vertex on the left, the better, since this would mean producing a lot of randomness from the original distribution without spending too many truly random bits.

It is known that a random graph with appropriately-chosen parameters is, with high probability, an extractor. What is more, it is easy to prove that if a graph is random in the sense of Kolmogorov complexity, it is still the case that the resulting graph is an extractor. This is not surprising since proofs based on simple probabilistic arguments often are easily expressed as Kolmogorov-based proofs.

The connection to Kolmogorov complexity, however, does not end there. In this chapter we show how the extraction of randomness achieved by an extractor can be expressed in terms of time-bounded Kolmogorov complexity. The use of known constructions for extractors also allows us to extend and improve results on the CD complexity of strings in small, feasibly computable sets.

4.1.3 Definitions

Before moving on to the statement and proofs of our results, we start by giving the formal background. The notions of Kolmogorov complexity that we use in this chapter are described in Chapter 2; here we give the definitions pertaining to extractors.

The definitions presented here are inspired by the presentation given in Nisan's excellent survey paper [Nis96].

An extractor can be thought of as a bipartite graph, whose first color class is larger than the second color class. By convention we think of the first color class as being on the left, and the second on the right. The vertices on the left side are all the strings of length n, so the first color class can be equated with the set $[N]$, where $N = 2^n$. Likewise, the vertices on the right side of the graph are labeled by strings of length $m \leq n$, so we let $M = 2^m$ and $[M]$ is equated with the vertices in the second color class.

On distributions

We will choose a node on the left side of the graph at random according to a distribution X.

The *min-entropy* of a distribution X over $[N]$ can be thought of as a measure of the randomness present in a string x chosen according to X. The min-entropy of D is defined to be $\min\{-\log_2 X(x) | x \in [N]\}$.

A distribution Y is said to be $\varepsilon-$close to X if both distributions are over the same space (say $[M]$) and for any $S \subseteq [M]$, $|X(S) - Y(S)| \leq \varepsilon$.

Extractors

A bipartite graph G with (independent) vertex sets $[N]$ and $[M]$, $N = 2^n, M = 2^m$, and for which the degree of all the vertices in the first color class is bounded by $D = 2^d$ is an (ε, k) *extractor* if given any distribution X on the N vertices whose min-entropy is at least k, the result of choosing an x according to this distribution and a random neighbor y of x in the graph is ε-close to the uniform distribution over $[M]$. We will also say that G is an extractor with parameters $(n, k, d, m, \varepsilon)$. In our setting, the distribution X will be the uniform distribution over a subset $A \subseteq [N]$, so k will be $\log |A|$.

$\Gamma(x)$ denotes the set of neighbors of x in G when x is a vertex on the left side of the graph. The number of edges originating at some vertex x on the left side of the graph is called the *outdegree* of x, whereas the number of edges incident with a vertex y on the right side of the graph is called the *indegree* of y. Edges incident to x are labeled 1 through the outdegree of x, and $G(x, r)$ represents the rth neighbor of x in the graph, where multiple edges are allowed. When y is a vertex on the right side of the graph, $\Gamma_A^{-1}(y)$ is the subset of preimages of y which lie in A. The notation extends to sets in the natural way.

The results we state are subject to improvement if better explicit extractor constructions are found. We have stated our results in general terms so that new results on extractors will be immediately applicable.

The current best known explicit constructions for extractors are due to Ta-Shma and Zuckerman [Ta-96, Zuc96]. They are described by the theorems below.

Theorem 6. (Ta-Shma) *There is an explicit construction which for every $n, \varepsilon = \varepsilon(n)$ and every $m = m(n) \leq n$ yields an extractor with parameters*

$$(n, m, \log^{O(1)}(n/\varepsilon), m, \varepsilon).$$

Theorem 7. (Zuckerman) *There is an explicit construction which for every constant c, every $n, \alpha = \alpha(n), \delta = \delta(n)$ yields an extractor with parameters*

$$(n, \delta n, c_\alpha \log(n/\varepsilon), (\delta - \alpha)n, \varepsilon),$$

where $c_\alpha = \frac{c \log(1/\alpha)}{\alpha}$.

It is useful to compare these constructions to the current lower bound on extractors, due to Nisan and Zuckerman [NZ96].

Theorem 8. (Nisan-Zuckerman) *There is a constant c such that for all $n, m, k \leq n-1, \varepsilon < 1/2$, if there is an extractor whose parameters are*

$$(n, k, d, m, \varepsilon),$$

then it must be the case that $d \geq \min\{m, c \log(n/\varepsilon)\}$.

This lower bound also gives a good indication as to the limits of the techniques described in this chapter.

4.2 Complexity bounds on instances of tractable sets

Our theorems improve upon the results of Sipser [Sip83] and Buhrman and Fortnow [BF97] in the sense that our bounds are stronger and the strings we obtain have high mutual information. The price we pay for these improvements is that our bounds apply only to "most" strings, not to all strings as in [Sip83, BF97].

Using the extractor construction of Ta-Shma [Ta-96] we get the following theorem.

Theorem 9. *For any set $A \in \mathcal{P}$, $\varepsilon = \varepsilon(n)$, there is a polynomial p such that for all n and for all but a 2ε fraction of the $x \in A \cap \Sigma^n$,*

$$CD^p(x) \leq \log|A \cap \Sigma^n| + \log^{O(1)}(n/\varepsilon).$$

We give a few extensions of this result. In one extension, we consider sets in $\mathcal{NP}$ and find short strings whose CD complexity is close to the nondeterministic CD complexity of the original string. We also give a randomized version of these theorems, stating that the shorter string can be chosen at random and the probability of getting a short string which encodes as much information as the original string is bounded away from 1/2.

Buhrman and Fortnow also bound the nondeterministic CD complexity of strings in sets in time-bounded nondeterministic classes. We state their result here and use it in the proof of one of our theorems.

Theorem 10. **(Buhrman-Fortnow)** *For any set $A \in \mathrm{NTIME}[t(n)]$, there is a polynomial p and a constant c such that for all strings $x \in A \cap \Sigma^n$,*

$$CND^{pt}(x) \leq 2\log(|A \cap \Sigma^n|) + c\log n.$$

4.2.1 Extracting CD complexity for sets in $\mathcal{P}$

Theorem 9 follows immediately from the following more general result:

Theorem 11. *Fix a set A in $\mathcal{P}$, a polynomial $q(n)$ and $\varepsilon = \varepsilon(n)$. Then there is a polynomial $p(n)$ such that for all n and for all but a 2ε fraction of the $x \in A \cap \Sigma^n$, there is a y such that*

1. $|y| = \log|A \cap \Sigma^n|$
2. $C^p(y|x) \leq \log D + O(1)$
3. $CD^p(x) \leq C^q(y) + 3\log D + c\log n + O(1)$,

where the value D depends on the construction of explicit extractor graphs, and currently the best known is $2^{\log^{O(1)}(n/\varepsilon)}$.

For the remainder of this section, we fix n and we let $S = |A \cap \Sigma^n|$. We use an (ε, S) extractor G, for which $M = S$ and $d = O(\log^{O(1)}(n/\varepsilon))$ [Ta-96]. In our setting, we will think of the set $A \cap \Sigma^n$ as defining a distribution of min-entropy $\log S$. The string x represents an element of $A \cap \Sigma^n$ and y is one of its neighbors in the graph G. Hence, y has length $\log S$; computing y from x requires knowing only a short ("random") string of length $\log D$; and as we will see, y together with some short additional distinguishing information will suffice to distinguish the string x (in the sense of CD complexity).

The following lemmas are at the heart of all the probability arguments. They allow us to upper-bound the number of "bad" elements in $A \cap \Sigma^n$, where "bad" means the strings x for which Theorem 11 will not apply.

In order to get a short description for x, we need to find a string y in its range which has small indegree.

In Lemma 4.2, we use the properties of the extractor to obtain an upper bound on the number of y which have large indegree, where "large" is parameterized by the value w_0.

Lemma 4.3 gives an upper bound on the number of x on the left side of the extractor whose neighbors all lie within a small subset of the right side of the graph. When applied to the set from Lemma 4.2, these x are the "bad" x to which the theorem will not apply.

Lemma 4.2. *Consider the restriction of the extractor to the set of edges originating in $A \cap \Sigma^n$. Let w_0 be an indegree threshold, $D < w_0 \leq DS$, and Y be a subset of vertices on the right hand side of the extractor graph. If $\forall y \in Y, w(y) > w_0$, then*

$$|Y| \leq \frac{\varepsilon DS}{w_0 - D}$$

Proof. This is just a simple counting argument. Let Y be the set of vertices whose indegree (in the restricted graph) exceeds w_0. Because the graph is an extractor, it must be the case that

$$\frac{w(Y)}{DS} - \frac{|Y|}{S} \leq \varepsilon.$$

Since $w(Y) \geq w_0|Y|$, rearrangement gives the desired inequality. □

Lemma 4.3. *If Y is a set on the right side of the graph containing at most $(1-\varepsilon)S$ vertices, then*

$$|\{x \in A \cap \Sigma^n : \Gamma(x) \subseteq Y\}| \leq \varepsilon S + |Y|$$

Proof. Once again this is a simple counting argument. The fact that the graph is an extractor tells us that $w(Y) \leq \varepsilon DS + D|Y|$. Each x for which $\Gamma(x) \subseteq Y$ contributes D edges to the indegree of Y, therefore the total number of such x cannot exceed $\varepsilon S + |Y|$ as claimed. □

To conclude, we give the proof of Theorem 11.

Proof. Let A be a set in $\mathcal{P}$ and ε, n be given as in the statement of the theorem. By Lemma 4.2, applied with $w_0 = 2D$ and Lemma 4.3 with Y as in the hypothesis of Lemma 4.2, the size of the subset $B \subseteq A \cap \Sigma^n$ such that $\forall x \in B$, $\forall y \in \Gamma(x)$, y has indegree at least w_0 can have size at most $2\varepsilon S$. Therefore for all but $2\varepsilon S$ of the x in $A \cap \Sigma^n$, there is a y in its range whose indegree is at most $2D$. For each such x, let r_x be the label of one of the edges in G which connects x to such a y.

We need to verify 3 properties for each of these pairs x, y.

1. $|y| = \log S$: This is simply by choice of the extractor G.
2. $C(y|x) \leq \log D + O(1)$: $y = G(x, r_x)$ for some $r_x \in \Sigma^d$, so the algorithm to print y will contain an encoding of r_x, and on input x computes $G(x, r_x)$ and outputs the result.
3. $CD(x) \leq C(y) + 3\log D + c\log n + O(1)$: The program to recognize x will contain an encoding for an r_x and y for which $G(x, r_x) = y$ and the indegree of y is at most $2D$. It must also contain a distinguishing program p which recognizes x, r_x among the $2D$ edges originating in A that are adjacent to y. The length of p is bounded by $2\log(2D) + c\log(n + \log D)$, as given by Theorem 5. The algorithm follows:
   On input z:
   (1) Check if z is in A. If not, output 0 and terminate.
   (2) Check if $G(z, r_x) = y$. If not, output 0 and terminate.
   (3) If $p(z) = 1$, then $z = x$. Output 1 and terminate. Otherwise output 0.

So the program needs an encoding of y, r, and the distinguishing program p, for a total length of $C(y) + \log(D) + 2\log(D) + c\log(n + \log D) + O(1)$.

$\square$

4.2.2 Extracting CND complexity for sets in $\mathcal{NP}$

Using a slight variant of the proof of Theorem 11, we can get the following result about CND complexity. It is stronger in that it applies to sets in $\mathcal{NP}$, and although the bound on the complexity of x is for CND instead of CD complexity, the gain is that the bound is smaller by a term of $\log D$ and it involves only $CD(y)$ instead of $C(y)$.

Theorem 12. *Fix a set A in $\mathcal{NP}$, a polynomial $q(n)$, and $\varepsilon = \varepsilon(n)$. Then there is a polynomial $p(n)$ such that for every n and for all but a 2ε fraction of the $x \in A \cap \Sigma^n$, there is a y such that*

1. $|y| = \log |A \cap \Sigma^n|$

2. $C^p(y|x) \leq \log D + O(1)$

3. $CND^p(x) \leq CD^q(y) + 2\log D + c\log(n + \log D) + O(1)$.

The proof is essentially the same as that of Theorem 11. To obtain property 3, we need only guess y, and verify our guess using a distinguishing program for y whose length is bounded by $CD^q(y)$. Likewise, we can simply guess r and omit its encoding, and use the distinguishing program p to verify our guess for r.

4.2.3 Randomly extracting CD complexity

Another trade-off we obtain to save a $\log D$ term is to choose a counterpart y to a string x in a set in $\mathcal{P}$ at random. We will only require that for most x, at least half of the edges from x map to a "good" y. Although this comes at the cost of only applying to "most" strings x, this improves upon the result of Sipser [Sip83] by reducing the length of the random string from $n^{O(1)}$ to $\log^{O(1)}(n/\varepsilon)$.

Theorem 13. *Fix a set A in $\mathcal{P}$, a polynomial $q(n)$, and $\varepsilon = \varepsilon(n)$. Then there is a polynomial $p(n)$ such that for every n and for all but a 4ε fraction of the $x \in A \cap \Sigma^n$, and at least half of the strings r of length D, there is a y such that:*

1. $|y| = \log |A \cap \Sigma^n|$
2. $C^p(y|x, r) \leq O(1)$
3. $CD^p(x|r) \leq C^q(y) + 2\log D + 2\log(n + \log D) + O(1)$.

The proof is similar to that of Theorem 11; it requires only a slight modification to the counting argument.

4.3 Extracting random strings

In the previous section, we used the fact that the strings examined were in a small set of bounded complexity, and we showed the existence of strings for which the mutual information was roughly the CD complexity of the original string. Here we use extractor techniques to a achieve a slightly different goal. We obtain an *incompressible* string whose length is close to the CD complexity of x and which can be computed from x using only $\log(n/\varepsilon)$ bits.

In the case of unbounded Kolmogorov complexity, it is easy to see that the following proposition is true.

Proposition 4.1. [LV97, Ex. 2.1.5, p. 102] *For any string x of length n, there is a y such that:*

1. $|y| = C(x)$

2. $C(y|x) \leq \log(n)$

3. $C(y) > |y| - O(1)$

Namely, y is a minimal-length program for x, and can be obtained from x by dovetailing, given the value of $C(x)$. In the time-bounded setting however, this argument fails, since dovetailing would take too much time. Our use of extractors is far afield from the above approach, yet it yields results surprisingly close to Proposition 4.1. (Non-explicit extractors actually allow us to give an alternate proof of Proposition 4.1, although this is more an artefact than a useful new proof.)

Theorem 14. *Fix a polynomial $q(n)$ and $\varepsilon = \varepsilon(n)$, and let $c > \log\left(\frac{1}{1-\varepsilon}\right)$ for large enough n. Then there exists a polynomial $p(n)$ such that for any string x of length n, there is a string y such that:*

1. $|y| = CND^p(x)/2 - O(\log n)$

2. $C^p(y|x) \leq \log^{O(1)}(n/\varepsilon)$

3. $C^q(y) > |y| - O(1)$.

Instead of giving the proof of Theorem 14, we prove the result in the following more general form, which may be improved as explicit extractor constructions are improved.

Theorem 15. *Fix a polynomial $q(n)$ and $\varepsilon = \varepsilon(n)$, and let $c > \log\left(\frac{1}{1-\varepsilon}\right)$ for large enough n. Then there exists a polynomial $p(n)$ such that for any string x, there is a string y such that:*

1. $|y| = m$

2. $C^p(y|x) \leq d + O(1)$

3. $C^q(y) > |y| - O(1)$,

provided there is an explicit extractor with parameters

$$(n, k, d, m, \varepsilon)$$

where $k = \frac{CND^{pD}(x) - O(\log n)}{2}$.

Theorem 14 follows by applying Theorem 15 with parameters obtained from Ta-Shma's extractor [Ta-96].

Proof. (Sketch) The idea of the proof is to consider a family of extractors parameterized by $n, k, m(k)$, and look at what happens in the extractor $n, k, m(k)$ to strings x of length n.

For fixed n, m, we let

$$A_{n,m} = \{x | \Gamma(x) \subseteq C[q(n), m - c]\}.$$

The fact that G is an extractor prohibits the set $A_{n,m}$ from being large, as we see now. If $|A_{n,m}| > 2^k$, then by the properties of the extractor graph, the weight on $C[q(n), m - c]$ induced by $A_{n,m}$ must be close to uniform, namely:

$$\frac{w(C[q(n), m - c])}{D|A_{n,m}|} \leq \frac{C[q(n), m - c]}{2^m} + \varepsilon.$$

Using the fact that $w(C[q(n), m - c]) = D|A_{n,m}|$ by definition of $A_{n,m}$ and that $|C[q(n), m - c]| \leq 2^{m-c}$, we get that

$$1 \leq 2^{-c} + \varepsilon.$$

However, we have chosen $c > \log\left(\frac{1}{1-\varepsilon}\right)$ precisely to eliminate this possibility. Hence we must conclude that $|A_{n,m}| \leq 2^k$.

Now we may apply Theorem 10 to conclude that all strings $x \in A_{n,m}$ must have small CND complexity. First, notice that verifying membership in $A_{n,m}$ is in NTIME$[Dp]$ for some polynomial p, since it suffices to guess, for each neighbor y of x in $G_{n,m}$ a program of length $m - c$ which prints out y. Hence, for every $x \in A_{n,m}$,

$$CND^{Dp}(x) \leq 2 \log(|A_{n,m}|) + 2 \log n + O(1).$$

Now consider x with respect to the extractor $G_{n,k,m(k)}$, where

$$k = \frac{1}{2}(CND^{D_p}(x) - 2\log n - O(1) - 1)$$

and m is maximal for this k. By the observation above, it must be the case that $x \notin A_{n,m}$. Therefore there must be a y not in $C[q(n), m-c]$ to which x is mapped under $G_{n,k,m}$. It is easy to verify that y satisfies the properties claimed in the statement of the theorem. □

4.4 Extensions

It makes sense to state the results above in a yet more general form. Instead of requiring that the reference set A be in $\mathcal{P}$, all results carry over to the setting where the instance complexity of all instances in A is small, adding the appropriate (small) term to the bounds we obtain.

4.5 Open questions

Two main areas stand out as requiring further investigation. The first is to improve current best known bounds on the CD complexity of instances in tractable sets. Can extractors be exploited in a more sophisticated way to yield better results? Are there other techniques that would allow us to come closer to the optimal $\log |A^{=n}|$ bound for CD complexity of instances in A? The second is to obtain complexity-theoretic consequences of the improved bounds presented in this chapter.

CHAPTER 5

INDISTINGUISHABILITY

How hard is it to decide an instance of a language A? This question has always been at the center of complexity theory. Complexity cores, defined by Lynch [Lyn75] and studied by several other authors [OS86, BD87, etc.] provided complexity theorists with a new angle: Are there particular instances which make a language hard to decide, independent of the algorithm chosen to solve it? Instance complexity was defined [OKSW94] and studied [BO96, FK96, Kum96] with a somewhat similar intention: quantifying the hardness of individual instances with respect to a set, independently of the algorithm chosen to solve it.

We take the approach of instance complexity a step further, and present what we hope will provide a new perspective on the hardness of deciding languages. We tackle the problem of hardness in a way that is *language independent*. As in instance complexity, we quantify how hard it is to distinguish two strings, and prove that there are in fact infinitely many *pairs* of strings, even *sets* of strings, that are *indistinguishable*. Therefore whenever one of these strings is in an "easy" language, then so must be all of the strings that are indistinguishable from it.

Two strings are *distinguished* by a program if the program always outputs one of $\{0, 1, \perp\}$, and it outputs 1 on one of the strings and 0 on the other. Two strings are k-indistinguishable if no program of length at most k can distinguish them.

In this chapter, we introduce the formal definitions for indistinguishability, and prove some basic properties. Indistinguishability is a compelling notion, and we illustrate this by showing that our framework provides a "dual" perspective on nondeterministic communication complexity. We do this by giving tight upper and almost tight lower bounds on nondeterministic communication complexity in terms of distinguishability.

The definition of indistinguishability has the same flavor as Kolmogorov complexity and instance complexity in that the focus is on the *length of the shortest program* which attempts to distinguish strings from one another. In Section 5.1 we introduce distinguishability and give an invariance theorem for distinguishability. Section 5.2 contains some basic results that provide motivation for studying indistinguishability. Section 5.3 links distinguishability with instance complexity and CD complexity. Finally, Section 5.4 establishes a relationship between communication complexity and distinguishability.

5.1 Indistinguishable pairs

In this section we give the definition of distinguishability, and we prove some basic properties about distinguishable strings.

5.1.1 Distinguishability

In order for our notion of indistinguishability to be robust, the programs we consider are ones realized by a fixed partial recursive function Φ. When we write $p(x)$ for a program p and input x, we mean the output of M_Φ on input $\langle p, x\rangle$, where $\langle\cdot,\cdot\rangle$ is a standard pairing function.

The output alphabet of Φ is $\{0, 1, \perp\}$, and the programs we consider always output one of $\{0, 1, \perp\}$. We mean $\perp$ to represent "I don't know"; 0 and 1 might represent the characteristic function of some unspecified set. We say that an algorithm *distinguishes* two strings if it outputs 0 on one string and 1 on the other. If Φ runs in time bounded by the function $t(|p| + |x|)$, then we say the strings are (k, t)-distinguishable. A formal definition follows.

Definition 9. *Two strings $x \neq y$ are (Φ, t, k)-distinguishable if there is a t-time-bounded program p for Φ of length k such that $p(x) \neq \perp, p(y) \neq \perp, p(x) \neq p(y)$.*

In the time-unbounded case, we write x and y are (Φ, k)-distinguishable.

Definition 10. $\mathrm{Dist}^t_\Phi(x,y)$ *is the smallest* k *such that* x, y *are* (Φ, t, k)*-distinguishable. If* $x = y$*, then* $\mathrm{Dist}^t_\Phi(x,y) = \infty$*.*

If we allow the distinguishing programs to access an oracle A, then we write $\mathrm{Dist}^{t,A}_\Phi(x,y)$, and if we also allow a common auxiliary input string w, we write $\mathrm{Dist}^t_\Phi(x,y|w)$.

5.1.2 Invariance

We now show that the notion of indistinguishability is independent of the choice of Φ.

Theorem 16. *There is a universal partial recursive function* Φ_0 *such that for any partial recursive* Φ*, time bound* t*, and pair of strings* x, y*,*

$$\mathrm{Dist}^{\tilde{t}}_{\Phi_0}(x,y) \leq \mathrm{Dist}^t_\Phi(x,y) + c'$$

where $\tilde{t} \leq ct(n)\log(t(n))$*, and* c, c' *are constants depending only on* Φ*.*

This is a standard argument, based on efficient simulation of multi-tape Turing machines by a two-tape Turing machine [LV97, page 460]. Let $M_1, M_2, \ldots$ be a standard enumeration of multitape Turing machines, and let $\Phi_1, \Phi_2, \ldots$ be the corresponding partial recursive functions. Using the standard simulator U, we know that for each partial recursive function Φ_i, $U(\langle\langle i, \cdot\rangle, \cdot\rangle)$ computes the same function as $\Phi_i(\langle \cdot, \cdot\rangle)$, and if $t(n)$ is a bound on the running time of $\Phi_i(\langle p, \cdot\rangle)$, the running time of $U(\langle\langle i, p\rangle, \cdot\rangle)$ is bounded by $\tilde{t}(n) = ct(n)\log(t(n))$, where c is a constant depending only on i. Furthermore, if we let Φ_0 be the partial function computed by the Turing machine U,

$$\mathrm{Dist}^{\tilde{t}}_{\Phi_0}(x,y) \leq \mathrm{Dist}^t_{\Phi_i}(x,y) + c',$$

where c' also depends only on i. This is because a Φ_i-program p which distinguishes x, y can be rewritten into a Φ_0-program to distinguish the same x, y, namely $\langle i, p\rangle$. $\square$

Given the invariance of Dist under the choice of Φ, we can write that x and y are (t,k)-distinguishable, or simply k-distinguishable in the time-unbounded case. Also we write $\mathrm{Dist}^t(x,y)$ to mean $\mathrm{Dist}^t_\Phi(x,y)$ for a fixed universal partial recursive function Φ.

5.1.3 Indistinguishable pairs of strings

It is easy to see that for any $x \neq y, |x| = |y| = n, \mathrm{Dist}^t_\Phi(x,y) \leq \log(n) + c$, since a program need only specify a bit position in which the two strings differ.

The following lemma establishes that for every string length, there are pairs of distinct strings x, y of length n such that $\mathrm{Dist}^t_\Phi(x,y) \geq \log(n) - 1$.

Lemma 5.1. $\forall n, k, 0 < k < \log(n)$, *there are at least* $2^{2n-2^{k+1}} - 2^{n-1}$ *pairs* $x \neq y, |x| = |y| = n$ *such that* $\mathrm{Dist}^t_\Phi(x,y) \geq k$

Proof. To each input x of length n corresponds a list of the output of all the different programs $p \in P^{\leq k}$, i.e, $(p_1(x), \ldots, p_{N_k}(x))$. We call this list the *signature* of x (under $P^{\leq k}$).

We say x's signature is *compatible* with a binary string, or *tag*, $w = w_1 \cdots w_{N_k} \in \{0,1\}^{N_k}$ if whenever $p_i(x) \neq \perp, p_i(x) = w_i$. Notice that any 2 inputs x, y whose signatures are both compatible with the same tag w have $\mathrm{Dist}^t_\Phi(x,y) \geq k$. When this happens, we say that x and y *collide* on w. We may assume without loss of generality that p_i never outputs $\perp$; this can only decrease the possible number of collisions.

From this point we use a counting argument to complete the proof. Since there are only N_k programs, there are only 2^{N_k} tags in $\{0,1\}^{N_k}$ and 2^n inputs competing for them. Even if we assume each signature is compatible with only one tag, the best case scenario (minimizing collisions) occurs when there is an equal number of collisions for each signature. Hence the number of collisions is bounded below by the following expression.

$$2^{N_k}\binom{2^{n-N_k}}{2} \quad = \quad 2^{N_k-1}2^{n-N_k}(2^{n-N_k}-1)$$

$$
\begin{aligned}
&= 2^{n-1}(2^{n-N_k}-1)\\
&= 2^{2n-N_k-1}-2^{n-1}\\
&= 2^{2n-2^{k+1}}-2^{n-1}
\end{aligned}
$$

$\square$

When $k = \log(n) - 1$, the number of pairs is at least 2^{n-1}; when $k = \log(n) - 2$, the number is $2^{\frac{3n}{2}} - 2^{n-1}$, and when $k = c$ for a constant c, i.e., when we consider only constant-size programs, the number is $\frac{1}{2^{c+1}}2^{2n} - 2^{n-1}$, a considerable fraction of the total number of pairs of distinct strings of length n: $2^{2n} - 2^n$.

5.2 Indistinguishable tangles

From the proof of Lemma 5.1, we get not only that there are many pairs of indistinguishable strings, but that there must be relatively large sets of strings compatible with a single tag w. By the pigeonhole principle, it is easy to see that there are always such "tangles" of size $2^{n-N_k} = 2^{n-2^{k+1}+1}$.

Definition 11. *A* (Φ, t, k)-tangle T *is a set of strings with the property that* $\forall x, y \in T, \mathrm{Dist}^t_\Phi(x, y) > k$. *We say* T *is a* (t, k)*-tangle when* Φ *is a fixed universal partial recursive function.*

We can say more about the size and distribution of the tangles if we relate them to the Kolmogorov complexity of the strings involved. We prove in the next lemma that any string of high Kolmogorov complexity must be involved in a large tangle.

Lemma 5.2. $\forall n, k, 0 < k < \log(n), |x| = n$, *there is a* (t, k)*-tangle* T_x *including* x *with*

$$|T_x| \geq 2^{C(x)-2^{k+1}-1-\log(k)-O(\log(n))} = \frac{2^{C(x)-2^k}}{n^{O(1)}}$$

Proof. We bound the Kolmogorov complexity of a string x by encoding the following information:

- the values k and n
- a tag w compatible with x's signature under $P^{\leq k}$
- its index among inputs whose signatures are compatible with w.

We must also encode the lengths of the segments so they can be decoded correctly. For this we need an additional term of $O(\log(n))$ to encode the length of k and n. (Note that the length of w depends only on k so it is not necessary to encode any further information.)

As in the proof of Lemma 5.1, $P^{\leq k}$ is the set of programs of length at most k. Let N_k represent $|P^{\leq k}| = 2^{k+1} - 1$.

We get the following bound on $C(x)$.

$$\begin{aligned} C(x) &\leq N_k + \log(|T_x|) + \log(k) + O(\log(n)) \\ \log(|T_x|) &\geq C(x) - N_k - \log(k) - O(\log(n)) \\ &= C(x) - 2^{k+1} - 1 - \log(k) - O(\log(n)) \end{aligned}$$

So $|T_x| \geq 2^{C(x) - 2^{k+1} - 1 - \log(k) - O(\log(n))}$, as claimed. □

To put this in perspective, it in interesting to look at the size of tangles associated with Kolmogorov-incompressible strings. If we allow programs of length up to $\log(n) - c$ for a constant $c > 1$, then the size is at least (roughly) $2^{n/4}$. If programs have length bounded by a constant c, then the size goes all the way up to $\frac{2^n}{q(n)}$, where q is a polynomial.

We can get a similar result for time-bounded computation using a result of Buhrman and Fortnow [BF97]. Their result, stated in Chapter 4, page 34 as Theorem 5, allows us to bound the CD complexity of instances in tractable sets of bounded size. Using their result we obtain the following analogue of Lemma 5.2 for polynomial-time CD complexity.

Lemma 5.3. *For every $n, k, 0 < k < \log(n), |x| = n$, and polynomial q, there is a polynomial p and a (q, k)-tangle T_x including x with*

$$|T_x| \geq 2^{\frac{CD^{2^k p}(x)}{2} - 2^k - \log(k) - O(\log(n))}$$

The proof is analogous to that of Lemma 5.2, but instead of including x's index into the tangle, we include its distinguishing program as given by Theorem 5.

5.3 Instance complexity and CD complexity

This section presents observations establishing the connection between indistinguishability on the one hand, and instance complexity and CD complexity, on the other.

First we observe that if either x or y has low distinguishing (CD) complexity, then the CD program may be used as a distinguisher for x and y.

Proposition 5.1. *For any x, y, and time bound t,*

$$\mathrm{Dist}^t(x, y) \leq \min\{CD^t(x), CD^t(x, y)\}.$$

In particular, it follows from Theorem 5 that if A is in P then for any x, y in A, $\mathrm{Dist}^p(x, y) \leq 2\log(|A \cap \Sigma^n|) + O(\log n)$.

Next, we show that strings can be easily distinguished if there is an easy set which contains exactly one of x, y.

Proposition 5.2. *If $x \in A$ and $y \notin A$ then*

$$\mathrm{Dist}(x, y) \leq \mathrm{ic}(x : A) + \mathrm{ic}(y : A) + O(1)$$

Proof. The proof is straightforward. Let p_0 and p_1 be programs consistent with A of length $\mathrm{ic}(x : A)$ and $\mathrm{ic}(y : A)$, respectively, and such that $p_0(x), p_1(y) \neq \perp$. Patch the programs together to distinguish x and y as follows: on input z, simulate $p_0(z)$ and $p_1(z)$. Output the following value:

$$\begin{array}{ll} 0 & \text{if } p_0(z) = 0 \text{ or } p_1(z) = 0 \\ 1 & \text{if } p_0(z) = 1 \text{ or } p_1(z) = 1 \\ \perp & \text{otherwise} \end{array}$$

The first two cases are mutually exclusive because p_0 and p_1 are consistent with A, and the resulting program correctly distinguishes x from y. $\square$

The instance complexity relative to a set A can also be bounded by distinguishers of instances in and out of the set, as we show below.

Proposition 5.3. *For every set A, time bound t, and instance z of length n,*

$$\mathrm{ic}^{2^d t}(z : A) \leq 2^{d+1},$$

where $d = \max_{x \in A \cap \Sigma^n, y \notin A \cap \Sigma^n} \{\mathrm{Dist}^t(x, y)\}$.

Proof. To solve instance z correctly, run z on all distinguishing programs of length at most d. Then compare all of these answers to a hard-coded "cheat-sheet" containing all the correct answers for z. If the answers all agree, conclude z is in the language; otherwise output $\perp$. This program must halt and output the correct answer on z. On all other inputs z', if the program does not output $\perp$, then z' must be in A by maximality of d, so the program is consistent with A as required. $\square$

5.4 Nondeterministic communication complexity

The framework of distinguishability applies naturally and elegantly to the notion of nondeterministic communication complexity. First we give a brief overview of the model, then we provide a connection between the two models. The connection takes the form of an upper bound and a lower bound on the communication complexity, in fact these bounds are tight in the sense that there are functions which meet both the upper and lower bounds we provide.

5.4.1 Communication complexity model

First we present the model of nondeterministic communication complexity, introduced by Lipton and Sedgewick [LS81]. We recommend the survey paper by Lovász [Lov90] for more background on communication complexity.

The setting is the following: 2 players, A and B, want to evaluate a boolean function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ on 2 arguments, x and y. A has x and B has y. They obtain help from a "divine source" in the form of a witness p drawn from a predetermined set of witnesses P. If $f(x,y) = 1$, then there is a p on which both A and B output 1. If not, there is no witness p that can cause both A and B to output 1. The goal is to minimize the length of p, which we may assume to be $\log |P|$. This length is the nondeterministic communication complexity for 1-instances of f, denoted $\kappa_1(f)$. The analogue for the 0-instances is denoted κ_0, where the witness set must have the property that a witness causes both parties to output 0 if and only if $f(x,y) = 0$.

We will associate with a communication problem f a matrix M_f whose rows are labeled by A's input, typically denoted by x, columns are labeled by B's input, typically y, and whose value in position x, y is the value $f(x,y)$.

5.4.2 A connection with distinguishability

It has long been known that the nondeterministic communication complexity has a precise combinatorial characterization in terms of homogeneous rectangle covers. A *rectangle* is a subset of the rows and columns of M_f. A *homogeneous rectangle* is a rectangle in which all the values $f(x,y)$ for which x is a row in the rectangle and y is a column in the rectangle share the same value. An all-1-rectangle is a homogeneous rectangle where all the values are 1, and all-0-rectangles are defined analogously. It is not hard to see that a family of all-0 rectangles which cover all the 0's in the communication matrix corresponds to a protocol for the 0 instances of f and vice versa. The same holds true for all-1 rectangle covers and protocols for 1-instances of f.

Distinguishers offer a dual viewpoint for nondeterministic communication complexity. We start by making a simple observation about how protocols can be viewed as distinguishers for inputs corresponding to 0 instances and those corresponding to 1-instances. Theorem 17 establishes that this connection goes both ways.

Claim 5.1. *If $f(x,y) \neq f(x,y')$ then either one of the 0- or 1-witnesses, together with B's strategy, serves as a distinguisher for y and y'.*

The triples x, y, y' can be thought of as obstacles to rectangles, and each obstacle must be countered with an appropriate distinguisher. In this sense, distinguishers provide us with a dual view of protocols.

Let P_f^A be a set of distinguishers which has the following property. For each x, and each pair y, y' such that $f(x,y) \neq f(x,y')$, there is a distinguisher $p \in P_f^A$ which distinguishes y from y' given the communication matrix M_f. Analogously, let P_f^B be a set of distinguishers for all the pairs x, x' for which there is a y such that $f(x,y) \neq f(x',y)$. Notice that if the length of the longest program in $P_f^A \cup P_f^B$ is d, then $|P_f^A \cup P_f^B|$ is at most 2^d.

Theorem 17. *Let P_f^A, P_f^B be as above, and denote by d the length of the longest distinguisher in $P_f^A \cup P_f^B$. Then*

$$d \leq \kappa_0, \kappa_1 \leq 2^{d+1}.$$

Proof. The first inequality follows directly from Claim 5.1.

To show the second inequality, we exhibit a protocol derived from $P_f^A \cup P_f^B$. We give the proof for 1-instances; the proof for 0 instances is analogous.

Fix a pair of inputs x, y for which $f(x,y) = 1$. To convince A that $f(x,y) = 1$, it is enough to convince her that B's input is none of the inputs y' for which $f(x,y') = 0$. Likewise, to convince B that $f(x,y) = 1$, it is enough to establish that x is not any x' for which $f(x',y) = 0$.

The first part of the witness is an encoding of the distinguishers in P_f^A which differentiate y from all y' for which $f(x, y') = 0$. The second part of the witness is an encoding of the distinguishers in P_f^A which differentiate y from all y' for which $f(x, y') = 0$.

Both accept if they agree with all that has been put forth by the prover, namely, A checks x against all the distinguishers in the second part of the witness; B checks y against all the distinguishers in the first part of the witness. A also checks that the values of y not eliminated by the prover's distinguishers force f to 1; likewise B checks that all values of x not evaluated by the prover's distinguishers force f to 1.

Both parts of the witness can be encoded as a $\{0, 1\}$ vector, where each position set to 1 indicate that the corresponding $p \in P_f^A$ or P_f^B must be checked against B's or A's input. Since there are at most 2^d programs of length $\leq d$, the witnesses have length $\leq 2^{d+1}$, which completes the proof of the theorem.

□

It should be noted that for the identity matrix (i.e., the communication matrix associated with the characteristic function $\chi_=$ of equality), it is known that $\kappa_0(\chi_=) = \log n$, and $\kappa_1(\chi_=) = n$. Clearly in this case $d = \log n$, so we have for this function that

$$\log n = d = \kappa_0(\chi_=)$$

and

$$\kappa_1(\chi_=) = 2^d = n.$$

In this sense our "dual" view of nondeterministic communication complexity is optimal (up to a factor of 2 in the upper bound).

5.5 Open questions

Much work remains to be done on the subject of indistinguishability, as we have only begun to scratch the surface of possible applications for this new framework. It is likely that indistinguishability techniques can be applied to obtain new or improved lower bounds for specific problems in communication complexity. Also worth pursuing is whether similar results to those presented in Section 5.4 can be shown for models of computation other than nondeterministic communication complexity. Nonuniform models of computation such as circuits appear to be most likely to lend themselves to analysis within this framework.

Much work also needs to be done in laying out further groundwork on indistinguishability itself. Can our results be refined to give better bounds on the size of tangles? Can we give constructions of sets which, under reasonable complexity theoretic assumptions, contain a large number of indistinguishable pairs? Can we improve our understanding of the relationship between indistinguishability and other, better known measures of complexity? Perhaps the answer to these questions will shed further light on the fundamental questions of computational complexity.

REFERENCES

[BD87] R.V. Book and D.-Z. Du. The existence and density of generalized complexity cores. *Journal of the ACM*, 34(3):718–730, July 1987.

[BF97] H. Buhrman and L. Fortnow. Resource-bounded Kolmogorov complexity revisited. In *14th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1200 of *Lecture Notes in COmputer Science*, Lübeck, Germany, 1997. Springer.

[BO96] H. Buhrman and P. Orponen. Random strings make hard instances. *Journal of Computer and System Sciences*, 53(2):261–266, October 1996.

[FFLN96] J. Feigenbaum, L. Fortnow, S. Laplante, and A. Naik. On coherence, random-self-reducibility and self-correction. In *Proceedings, Eleventh Annual IEEE Conference on Computational Complexity*, pages 59–67, 1996.

[FFLNar] J. Feigenbaum, L. Fortnow, S. Laplante, and A. Naik. On coherence, random-self-reducibility and self-correction. *Computational Complexity*, to appear.

[FFLS94] J. Feigenbaum, L. Fortnow, C. Lund, and D. Spielman. The power of adaptiveness and additional queries in random-self-reductions. *Computational Complexity*, 4:158–174, 1994.

[FK96] L. Fortnow and M. Kummer. Resource-bounded instance complexity. *Theoretical Computer Science*, 161(1–2):123–140, July 1996.

[FL94] L. Fortnow and S. Laplante. A note on adaptiveness and advice in coherence. Technical Report 94–19, University of Chicago Department of Computer Science, 1994.

[FL95] L. Fortnow and S. Laplante. Circuit lower bounds *à la* Kolmogorov. *Information and Computation*, 123(1):121–126, November 1995.

[FL96] L. Fortnow and S. Laplante. Extractors for Kolmogorov complexity. Technical Report 96–25, University of Chicago Department of Computer Science, 1996.

[Hås89] J. Håstad. Almost optimal lower bounds for small depth circuits. In S. Micali, editor, *Randomness and Computation*, pages 143–170. 1989.

[IT89] R. Impagliazzo and G. Tardos. Decision versus search problems in super-polynomial time. In *30th Annual Symposium on Foundations of Computer Science*, pages 222–227, 1989.

[KL80] R. Karp and R. Lipton. Some connections between nonuniform and uniform complexity classes. In *Conference Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, pages 302–309, 1980. An extended version has also appeared as: Turing machines that take advice, *L'Enseignement Mathématique*, 2nd series, 28:191–209, 1982.

[KP80] D. Knuth and L. Pardo. The early development of programming languages. In *A History of Computing in the Twentieth Century*, pages 197–273. Academic Press, 1980.

[Kum96] M. Kummer. Kolmogorov complexity and instance complexity of recursively enumerable sets. *SIAM Journal on Computing*, 25(6):1123–1143, December 1996.

[Lov90] L. Lovász. Communication complexity: A survey. In *Paths, Flows, and VLSI-layout*, pages 235–266. Springer-Verlag, 1990.

[LR96] S. Laplante and J. Rogers. Indistinguishability. Technical Report 96–26, University of Chicago Department of Computer Science, 1996.

[LS81] R. Lipton and R. Sedgewick. Lower bounds for VLSI. In *iConference Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computation*, pages 300–307, 1981.

[LV92] M. Li and P. Vitányi. Average case complexity under the universal distribution equals worst-case complexity. *Information Processing Letters*, 42(3):145–149, May 1992.

[LV97] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Verlag, second edition, 1997.

[Lyn75] N. Lynch. On reducibility to complex or sparse sets. *Journal of the ACM*, 22:341–345, July 1975.

[Nis96] N. Nisan. Extracting randomness: How and why (a survey). In *Proceedings of the 11th IEEE Conference on Computational Complexity*, pages 44–58, 1996.

[NZ96] N. Nisan and D. Zuckerman. Randomness is linear in space. *Journal of Computer and System Sciences*, 52(1):43–52, February 1996.

[OKSW94] P. Orponen, K.-I Ko, U. Schöning, and O. Watanabe. Instance complexity. *Journal of the ACM*, 41(1):96–121, January 1994.

[OS86] P. Orponen and U. Schöning. The density and complexity of polynomial cores for intractable sets. *Information and Control*, 70:54–68, July 1986.

[Sip83] M. Sipser. A complexity theoretic approach to randomness. In *Proceedings of the 15th ACM Symposium on the Theory of Computing*, pages 330–335, 1983.

[Ta-96] A. Ta-Shma. On extracting randomness from weak random sources. In *Proceedings of the 28th ACM Symposium on the Theory of Computing*, pages 276–285, 1996.

[Tur37] A. Turing. On computable numbers, with an application to the entscheidungspoblem. In *Proceedings of the London Mathematical Society*, volume 42, pages 230–265, 1937. Correction: 43:544–546 (1937).

[VV86] L. Valiant and V. Vazirani. NP is as easy as detecting unique solutions. *Theoretical Computer Science*, 47(1):85–93, 1986.

[Yao85] A. C.-C. Yao. Separating the polynomial hierarchy by oracles. In *Proceedings, 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 1–10, 1985.

[Yao90] A. Yao. Coherent functions and program checkers. In *Proceedings of 22nd ACM Symposium on Theory of Computing*, pages 84–94, 1990.

[Zuc96] D. Zuckerman. Randomness-optimal sampling, extractors, and constructive leader election. In *Proceedings of the 28th ACM Symposium on the Theory of Computing*, pages 286–295, 1996.