Computational Complexity: A Modern Approach

Draft of a book: Dated January 2007
Comments welcome!

Sanjeev Arora and Boaz Barak

Princeton University

complexitybook@gmail.com

This is an Internet draft. Some chapters are more finished than others. References and attributions are very preliminary and we apologize in advance for any omissions (but hope you will nevertheless point them out to us).

Please send us bugs, typos, missing references or general comments to
complexitybook@gmail.com — **Thank You!!**

DRAFT

ii

About this book

Computational complexity theory has developed rapidly in the past three decades. The list of surprising and fundamental results proved since 1990 alone could fill a book: these include new probabilistic definitions of classical complexity classes (**IP** = **PSPACE** and the **PCP** Theorems) and their implications for the field of approximation algorithms; Shor's algorithm to factor integers using a quantum computer; an understanding of why current approaches to the famous **P** versus **NP** will not be successful; a theory of derandomization and pseudorandomness based upon computational hardness; and beautiful constructions of pseudorandom objects such as extractors and expanders.

This book aims to describe such recent achievements of complexity theory in the context of the classical results. It is intended to both serve as a textbook as a reference for self-study. This means it must simultaneously cater to many audiences, and it is carefully designed with that goal. Throughout the book we explain the context in which a certain notion is useful, and *why* things are defined in a certain way. Examples and solved exercises accompany key definitions. We assume essentially no computational background and very minimal mathematical background, which we review in Appendix A.

We have also provided a *web site* for this book at `http://www.cs.princeton.edu/theory/complexity/` with related auxiliary material. This includes web chapters on automata and computability theory, detailed teaching plans for courses based on this book, a draft of all the book's chapters, and links to other online resources covering related topics.

The book is divided into three parts:

Part I: Basic complexity classes. This volume provides a broad introduction to the field. Starting from the definition of Turing machines and the basic notions of computability theory, this volumes covers the basic time and space complexity classes, and also includes a few more modern topics such probabilistic algorithms, interactive proofs and cryptography.

Part II: Lower bounds on concrete computational models. This part describes lower bounds on resources required to solve algorithmic tasks on concrete models such as circuits, decision trees, etc. Such models may seem at first sight very different from Turing machines, but looking deeper one finds interesting interconnections.

Part III: Advanced topics. This part is largely devoted to developments since the late 1980s. It includes average case complexity, derandomization and pseudorandomness, the **PCP** theorem and hardness of approximation, proof complexity and quantum computing.

Almost every chapter in the book can be read in isolation (though we recommend reading Chapters 1, 2 and 7 before reading later chapters). This is important because the book is aimed

at many classes of readers:

- *Physicists, mathematicians, and other scientists.* This group has become increasingly interested in computational complexity theory, especially because of high-profile results such as Shor's algorithm and the recent deterministic test for primality. This intellectually sophisticated group will be able to quickly read through Part I. Progressing on to Parts II and III they can read individual chapters and find almost everything they need to understand current research.

- *Computer scientists (e.g., algorithms designers) who do not work in complexity theory per se.* They may use the book for self-study or even to teach a graduate course or seminar.

- *All those —professors or students— who do research in complexity theory or plan to do so.* They may already know Part I and use the book for Parts II and III, possibly in a seminar or reading course. The coverage of advanced topics there is detailed enough to allow this.

This book can be used as a textbook for several types of courses. We will provide several teaching plans and material for such courses on the book's web site.

- *Undergraduate Theory of Computation Course.* Part I may be suitable for an undergraduate course that is an alternative to the more traditional *Theory of Computation* course currently taught in most computer science departments (and exemplified by Sipser's excellent book with the same name [SIP96]). Such a course would have a greater emphasis on modern topics such as probabilistic algorithms and cryptography. We note that in contrast to Sipser's book, the current book has a quite minimal coverage of computability and no coverage of automata theory, but we provide web-only chapters with more coverage of these topics on the book's web site. The prerequisite mathematical background would be some comfort with mathematical proofs and elementary probability on finite sample spaces, topics that are covered in typical "discrete math"/"math for CS" courses currently offered in most CS departments.

- *Advanced undergraduate/beginning graduate introduction to complexity course.* The book can be used as a text for an introductory complexity course aimed at undergraduate or non-theory graduate students (replacing Papadimitriou's 1994 book [Pap94] that does not contain many recent results). Such a course would probably include many topics from Part I and then a sprinkling from Parts II and III, and assume some background in algorithms and/or the theory of computation.

- *Graduate Complexity course.* The book can serve as a text for a graduate complexity course that prepares graduate students interested in theory to do research in complexity and related areas. Such a course can use parts of Part I to review basic material, and then move on to the advanced topics of Parts II and III. The book contains far more material than can be taught in one term, and we provide on our website several alternative outlines for such a course.

We hope that this book conveys our excitement about this new field and the insights it provides in a host of older disciplines.

DRAFT

Contents

DRAFT

III Advanced topics

15 Average Case Complexity: Levin's Theory

16 Derandomization, Expanders and Extractors

DRAFT

DRAFT

DRAFT

Introduction

> *"As long as a branch of science offers an abundance of problems, so long it is alive; a lack of problems foreshadows extinction or the cessation of independent development."*
> David Hilbert, 1900

> *"The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why?...I (would like to) show that there is no algorithm for multiplication computationally as simple as that for addition, and this proves something of a stumbling block."*
> Alan Cobham, 1964 [Cob64]

The notion of *computation* has existed in some form for thousands of years. In its everyday meaning, this term refers to the process of producing an output from a set of inputs in a finite number of steps. Here are three examples for computational tasks:

- Given two integer numbers, compute their product.

- Given a set of n linear equations over n variables, find a solution if it exists.

- Given a list of acquaintances and a list of containing all pairs of individuals who are not on speaking terms with each other, find the largest set of acquaintances you can invite to a dinner party such that you do not invite any two who are not on speaking terms.

In the first half of the 20th century, the notion of "computation" was made much more precise than the hitherto informal notion of "a person writing numbers on a note pad following certain rules." Many different models of computation were discovered —Turing machines, lambda calculus, cellular automata, pointer machines, bouncing billiards balls, Conway's *Game of life*, etc.— and found to be equivalent. More importantly, they are all *universal*, which means that each is capable of implementing all computations that we can conceive of on any other model (see Chapter 1). The notion of universality motivated the invention of the standard *electronic computer*, which is capable of executing all possible programs. The computer's rapid adoption in society in the subsequent half decade brought computation into every aspect of modern life, and made computational issues important in design, planning, engineering, scientific discovery, and many other human endeavors.

However, computation is not just a practical tool, but also a major scientific concept. Generalizing from models such as cellular automata, scientists have come to view many natural phenomena

as akin to computational processes. The understanding of reproduction in living things was triggered by the discovery of self-reproduction in computational machines. (In fact, a famous article by Pauli predicted the existence of a DNA-like substance in cells almost a decade before Watson and Crick discovered it.) Today, computational models underlie many research areas in biology and neuroscience. Several physics theories such as QED give a description of nature that is very reminiscent of computation, motivating some scientists to even suggest that the entire universe may be viewed as a giant computer (see Lloyd [**?**]). In an interesting twist, such physical theories have been used in the past decade to design a model for *quantum computation*; see Chapter 20.

From 1930s to the 1950s, researchers focused on the theory of *computability* and showed that several interesting computational tasks are *inherently uncomputable*: no computer can solve them without going into infinite loops (i.e., never halting) on certain inputs. Though a beautiful theory, it will not be our focus here. (But, see the texts [SIP96, HMU01, Koz97, **?**].) Instead, we focus on issues of *computational efficiency*. *Computational complexity theory* is concerned with how much computational resources are required to solve a given task. The questions it studies include the following:

1. Many computational tasks involve searching for a solution across a vast space of possibilities (for example, the aforementioned tasks of solving linear equations and finding a maximal set of invitees to a dinner party). Is there an *efficient* search algorithm for all such tasks, or do some tasks inherently require an exhaustive search?

   As we will see in Chapter 2, this is the famous "**P** vs. **NP**" question that is considered the central problem of complexity theory. Computational search tasks of the form above arise in a host of disciplines including the life sciences, social sciences and operations research, and computational complexity has provided strong evidence that many of these tasks are *inherently intractable*.

2. Can algorithms use randomness (i.e., coin tossing) to speed up computation?

   Chapter 7 presents *probabilistic algorithms* and shows several algorithms and techniques that use probability to solve tasks more efficiently. But Chapters 16 and 17 show a surprising recent result giving strong evidence that randomness does *not* help speed up computation, in the sense that any probabilistic algorithm can be replaced with a *deterministic* algorithm (tossing no coins) that is almost as efficient.

3. Can hard problems be solved quicker if we allow the algorithms to err on a small number of inputs, or to only compute an *approximate* solution?

   Average-case complexity and *approximation algorithms* are studied in Chapters 15, 17, 18 and 19. These chapters also show fascinating connections between these questions, the power of randomness, different notions of mathematical proofs, and the theory of error correcting codes.

4. Is there any use for computationally hard problems? For example, can we use them to construct secret codes that are *unbreakable*? (at least in the universe's lifetime).

   Our society increasingly relies on digital cryptography for commerce and security. As described in Chapter 10, these secret codes are built using certain hard computational tasks

such as factoring integers. The security of digital cryptography is intimately related to the **P** vs. **NP** question (see Chapter 2) and average-case complexity (see Chapters 15).

5. Can we use the counterintuitive quantum mechanical properties of our universe to solve hard problems faster?

   Chapter 20 describes the fascinating notion of *quantum computers* that use such properties to speed up certain computations. Although there are many theoretical and practical obstacles to actually building such computers, they have generated tremendous interest in recent years. This is not least due to Shor's algorithm that showed that, if built, quantum computers will be able to factor integers efficiently. (Thus breaking many of the currently used cryptosystems.)

6. Can we generate mathematical proofs automatically? Can we check a mathematical proof by only reading 3 probabilistically chosen letters from it? Do interactive proofs, involving a dialog between prover and verifier, have more power than standard "static" mathematical proofs?

   The notion of *proof*, central to mathematics, turns out to be central to computational complexity as well, and complexity has shed new light on the meaning of mathematical proofs. Whether mathematical proofs can be generated automatically turns out to depend on the **P** vs. **NP** question (see Chapter 2). Chapter 18 describes *probabilistically checkable proofs*. These are surprisingly robust mathematical proofs that can checked by only reading them in very few probabilistically chosen locations. *Interactive proofs* are studied in Chapter 8. Finally, *proof complexity*, a subfield of complexity studying the minimal proof length of various statements, is studied in Chapter 21.

At roughly 40 years of age, Complexity theory is still an infant science. Thus we still do not have complete answers for any of these questions. (In a surprising twist, computational complexity has also been used to provide evidence for the hardness to solve some of the questions of ... computational complexity; see Chapter 22.) Furthermore, many major insights on these questions were only found in recent years.

Meaning of efficiency

Now we explain the notion of *computational efficiency*, using the three examples for computational tasks we mentioned above. We start with the task of multiplying two integers. Consider two different methods (or *algorithms*) to perform this task. The first is *repeated addition*: to compute $a \cdot b$, just add a to itself $b-1$ times. The other is the *grade-school algorithm* illustrated in Figure 1. Though the repeated addition algorithm is perhaps simpler than the grade-school algorithm, we somehow feel that the latter is *better*. Indeed, it is much more efficient. For example, multiplying 577 by 423 using repeated addition requires 422 additions, whereas doing it with the grade-school algorithm requires only 3 additions and 3 multiplications of a number by a single digit.

We will quantify the efficiency of an algorithm by studying the number of *basic operations* it performs as the *size* of the input increases. Here, the *basic operations* are addition and multiplication of single digits. (In other settings, we may wish to throw in division as a basic operation.) The

$$
\begin{array}{ccccccc}
 & & & 5 & 7 & 7 \\
 & & & 4 & 2 & 3 \\
\hline
 & & 1 & 7 & 3 & 1 \\
 & 1 & 1 & 5 & 4 & \\
2 & 3 & 0 & 8 & & \\
\hline
2 & 4 & 4 & 0 & 7 & 1 \\
\end{array}
$$

Figure 1: Grade-school algorithm for multiplication. Illustrated for computing $577 \cdot 423$.

size of the input is the number of digits in the numbers. The number of basic operations used to multiply two n-digit numbers (i.e., numbers between 10^{n-1} and 10^n) is at most $2n^2$ for the grade-school algorithm and at least $n10^{n-1}$ for repeated addition. Phrased this way, the huge difference between the two algorithms is apparent: even for 11-digit numbers, a pocket calculator running the grade-school algorithm would beat the best current supercomputer running the repeated addition algorithm. For slightly larger numbers even a fifth grader with pen and paper would outperform a supercomputer. We see that *the efficiency of an algorithm is to a considerable extent much more important than the technology used to execute it.*

Surprisingly enough, there is an even faster algorithm for multiplication that uses the *Fast Fourier Transform*. It was only discovered some 40 years ago and multiplies two n-digit numbers using $cn \log n$ operations where c is some absolute constant independent of n. (Using asymptotic notation, we call this an $O(n \log n)$-step algorithm; see Chapter 1.)

Similarly, for the task of solving linear equations, the classic *Gaussian elimination* algorithm (named after Gauss but already known in some form to Chinese mathematicians of the first century) uses $O(n^3)$ basic arithmetic operations to solve n equations over n variables. In the late 1960's, Strassen found a more efficient algorithm that uses roughly $O(n^{2.81})$ operations, and the best current algorithm takes $O(n^{2.376})$ operations.

The *dinner party* task also has an interesting story. As in the case of multiplication, there is an obvious and simple inefficient algorithm: try all possible subsets of the n people from the largest to the smallest, and stop when you find a subset that does not include any pair of guests who are not on speaking terms. This algorithm can take as much time as the number of subsets of a group of n people, which is 2^n. This is highly unpractical —an organizer of, say, a 70-person party, would need to plan it at least a thousand years in advance, even if she has a supercomputer at her disposal. Surprisingly, we still do not know of a significantly better algorithm. In fact, as we will see in Chapter 2, we have reasons to suspect that no efficient algorithm *exists* for this task. We will see that it is equivalent to the *independent set* computational problem, which, together with thousands of other important problems, is **NP**-complete. The famous "**P** versus **NP**" question asks whether or not any of these problems has an efficient algorithm.

Proving nonexistence of efficient algorithms

We have seen that sometimes computational tasks have nonintuitive algorithms that are more efficient than algorithms that were known for thousands of years. It would therefore be really

interesting to prove for some computational tasks that the current algorithm is the *best* —in other words, no better algorithms exist. For instance, we could try to prove that the $O(n \log n)$-step algorithm for multiplication can never be improved (thus implying that multiplication is inherently more difficult than addition, which does have an $O(n)$-step algorithm). Or, we could try to prove that there is no algorithm for the dinner party task that takes fewer than $2^{n/10}$ steps.

Since we cannot very well check every one of the infinitely many possible algorithms, the only way to verify that the current algorithm is the best is to *mathematically prove* that there is no better algorithm. This may indeed be possible to do, since computation can be given a mathematically precise model. There are several precedents for proving *impossibility results* in mathematics, such as the independence of Euclid's parallel postulate from the other basic axioms of geometry, or the impossibility of trisecting an arbitrary angle using a compass and straightedge. Such results count among the most interesting, fruitful, and surprising results in mathematics.

Given the above discussion, it is no surprise that mathematical proofs are the main tool of complexity theory, and that this book is filled with theorems, definitions and lemmas. However, we hardly use any fancy mathematics and so the main prerequisite for this book is the ability to read (and perhaps even enjoy!) mathematical proofs. The reader might want to take a quick look at Appendix A, that reviews mathematical proofs and other notions used, and come back to it as needed.

We conclude with another quote from Hilbert's 1900 lecture:

> *Proofs of impossibility were effected by the ancients ... [and] in later mathematics, the question as to the impossibility of certain solutions plays a preminent part. ...*
>
> *In other sciences also one meets old problems which have been settled in a manner most satisfactory and most useful to science by the proof of their impossibility. ... After seeking in vain for the construction of a perpetual motion machine, the relations were investigated which must subsist between the forces of nature if such a machine is to be impossible; and this inverted question led to the discovery of the law of the conservation of energy. ...*
>
> *It is probably this important fact along with other philosophical reasons that gives rise to conviction ... that every definite mathematical problem must necessary be susceptible of an exact settlement, either in the form of an actual answer to the question asked, or by the proof of the impossibility of its solution and therewith the necessary failure of all attempts. ... This conviction... is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason, for in mathematics there is no ignorabimus.*

Conventions: A *whole number* is a number in the set $\mathbb{Z} = \{0, \pm1, \pm2, \ldots\}$. A number denoted by one of the letters i, j, k, ℓ, m, n is always assumed to be whole. If $n \geq 1$, then we denote by $[n]$ the set $\{1, \ldots, n\}$. For a real number x, we denote by $\lceil x \rceil$ the smallest $n \in \mathbb{Z}$ such that $n \geq x$ and by $\lfloor x \rfloor$ the largest $n \in \mathbb{Z}$ such that $n \leq x$. Whenever we use a real number in a context requiring a whole number, the operator $\lceil \ \rceil$ is implied. We denote by $\log x$ the logarithm of x to the base 2. We say that a condition holds for *sufficiently large n* if it holds for every $n \geq N$ for some number N (for example, $2^n > 100n^2$ for sufficiently large n). We use expressions such as $\sum_i f(i)$ (as opposed to, say, $\sum_{i=1}^{n} f(i)$) when the range of values i takes is obvious from the context. If u is a string or vector, then u_i denotes the value of the ith symbol/coordinate of u.

Part I

Basic Complexity Classes

Chapter 1

The computational model —and why it doesn't matter

> "The idea behind digital computers may be explained by saying that these machines
> are intended to carry out any operations which could be done by a human computer.
> The human computer is supposed to be following fixed rules; he has no authority to
> deviate from them in any detail. We may suppose that these rules are supplied in a
> book, which is altered whenever he is put on to a new job. He has also an unlimited
> supply of paper on which he does his calculations."
> Alan Turing, 1950

> "[Turing] has for the first time succeeded in giving an absolute definition of an in-
> teresting epistemological notion, i.e., one not depending on the formalism chosen."
> Kurt Gödel, 1946

The previous chapter gave an informal introduction to computation and *efficient computations* in context of arithmetic. IN this chapter we show a more rigorous and general definition. As mentioned earlier, one of the surprising discoveries of the 1930s was that all known computational models are able to *simulate* each other. Thus the set of *computable* problems does not depend upon the computational model.

In this book we are interested in issues of *computational efficiency*, and therefore in classes of "efficiently computable" problems. Here, at first glance, it seems that we have to be very careful about our choice of a computational model, since even a kid knows that whether or not a new video game program is "efficiently computable" depends upon his computer's hardware. Surprisingly though, we can restrict attention to a single abstract computational model for studying many questions about efficiency—the Turing machine. The reason is that the Turing Machine seems able to *simulate* all physically realizable computational models with very little loss of efficiency. Thus the set of "efficiently computable" problems is at least as large for the Turing Machine as for any other model. (One possible exception is the quantum computer model, but we do not currently know if it is physically realizable.)

The *Turing machine* is a simple embodiment of the age-old intuition that computation consists of applying mechanical rules to manipulate numbers, where the person/machine doing the manipulation is allowed a *scratch pad* on which to write the intermediate results. The Turing Machine can be also viewed as the equivalent of any modern programming language — albeit one with no built-in prohibition about memory size[1]. In fact, this intuitive understanding of computation will suffice for most of the book and most readers can skip many details of the model on a first reading, returning to them later as needed.

The rest of the chapter formally defines the Turing Machine and the notion of *running time*, which is one measure of computational effort. It also presents the important notion of the *universal Turing machine*. Section 1.5 introduces a class of "efficiently computable" problems called **P** (which stands for *Polynomial* time) and discuss its philosophical significance. The section also points out how throughout the book the definition of the Turing Machine and the class **P** will be a starting point for definitions of many other models, including nondeterministic, probabilistic and quantum Turing machines, Boolean circuits, parallel computers, decision trees, and communication games. Some of these models are introduced to study arguably realizable modes of physical computation, while others are mainly used to gain insights on Turing machines.

1.1 Encodings and Languages: Some conventions

Below we specify some of the notations and conventions used throughout this chapter and this book to represent computational problem. We make use of some notions from discrete math such as strings, sets, functions, tuples, and graphs. All of these notions are reviewed in Appendix **??**.

1.1.1 Representing objects as strings

In general we study the complexity of computing a function whose input and output are finite strings of bits. (A string of bits is a finite sequence of zeroes and ones. The set of all strings of length n is denoted by $\{0,1\}^n$, while the set of all strings is denoted by $\{0,1\}^* = \cup_{n \geq 0} \{0,1\}^n$; see Appendix A.) Note that simple encodings can be used to represent general objects—integers, pairs of integers, graphs, vectors, matrices, etc.— as strings of bits. For example, we can represent an integer as a string using the binary expansion (e.g., 34 is represented as 100010) and a graph as its adjacency matrix (i.e., an n vertex graph G is represented by an $n \times n$ 0/1-valued matrix A such that $A_{i,j} = 1$ iff the edge (i,j) is present in G). We will typically avoid dealing explicitly with such low level issues of representation, and will use $\llcorner x \lrcorner$ to denote some canonical (and unspecified) binary representation of the object x. Often we will drop the symbols $\llcorner \lrcorner$ and simply use x to denote both the object and its representation.

Representing pairs and tuples. We use the notation $\langle x, y \rangle$ to denote the ordered pair consisting of x and y. A canonical representation for $\langle x, y \rangle$ can be easily obtained from the representations of x and y. For example, we can first encode $\langle x, y \rangle$ as the string $\llcorner x \lrcorner \circ \# \circ \llcorner y \lrcorner$ over the alphabet

---

[1] Though the assumption of an infinite memory may seem unrealistic at first, in the complexity setting it is of no consequence since we will restrict the machine to use a finite amount of tape cells for any given input (the number allowed will depend upon the input size).

DRAFT

$\{0, 1, \#\}$ (where $\circ$ denotes concatenation) and then use the mapping $0 \mapsto 00, 1 \mapsto 11, \# \mapsto 01$ to convert this into a string of bits. To reduce notational clutter, instead of $\llcorner \langle x, y \rangle \lrcorner$ we use $\langle x, y \rangle$ to denote not only the pair consisting of x and y but also the representation of this pair as a binary string. Similarly, we use $\langle x, y, z \rangle$ to denote both the ordered triple consisting of x, y, z and its representation, and use similar notation for 4-tuples, 5-tuples etc..

1.1.2 Decision problems / languages

An important special case of functions mapping strings to strings is the case of *Boolean* functions, whose output is a single bit. We identify such a function f with the set $L_f = \{x : f(x) = 1\}$ and call such sets *languages* or *decision problems* (we use these terms interchangeably). We identify the computational problem of computing f (i.e., given x compute $f(x)$) with the problem of deciding the language L_f (i.e., given x, decide whether $x \in L_f$).

EXAMPLE 1.1
By representing the possible invitees to a dinner party with the vertices of a graph having an edge between any two people that can't stand one another, the dinner party computational problem from the introduction becomes the problem of finding a maximum sized *independent set* (set of vertices not containing any edges) in a given graph. The corresponding language is:

$$\mathsf{INDSET} = \{\langle G, k \rangle : \exists S \subseteq V(G) \text{ s.t. } |S| \geq k \text{ and } \forall u, v \in S, \overline{uv} \notin E(G)\}$$

An algorithm to solve this language will tell us, on input a graph G and a number k, whether there exists a conflict-free set of invitees, called an *independent set*, of size at least k. It is not immediately clear that such an algorithm can be used to actually find such a set, but we will see this is the case in Chapter 2. For now, let's take it on faith that this is a good formalization of this problem.

1.1.3 Big-Oh notation

As mentioned above, we will typically measure the computational efficiency algorithm as the number of a basic operations it performs as *a function of its input length*. That is, the efficiency of an algorithm can be captured by a function T from the set of natural numbers $\mathbb{N}$ to itself such that $T(n)$ is equal to the maximum number of basic operations that the algorithm performs on inputs of length n. However, this function is sometimes be overly dependant on the low-level details of our definition of a basic operation. For example, the addition algorithm will take about three times more operations if it uses addition of single digit *binary* (i.e., base 2) numbers as a basic operation, as opposed to *decimal* (i.e., base 10) numbers. To help us ignore these low level details and focus on the big picture, the following well known notation is very useful:

DEFINITION 1.2 (BIG-OH NOTATION)
If f, g are two functions from $\mathbb{N}$ to $\mathbb{N}$, then we **(1)** say that $f = O(g)$ if there exists a constant c such that $f(n) \leq c \cdot g(n)$ for every sufficiently large n, **(2)** say that $f = \Omega(g)$ if $g = O(f)$, **(3)** say that $f = \Theta(g)$ is $f = O(g)$ and $g = O(f)$, **(4)** say that $f = o(g)$ if for every $\epsilon > 0$, $f(n) \leq \epsilon \cdot g(n)$ for every sufficiently large n, and **(5)** say that $f = \omega(g)$ if $g = o(f)$.

To emphasize the input parameter, we often write $f(n) = O(g(n))$ instead of $f = O(g)$, and use similar notation for $o, \Omega, \omega, \Theta$.

EXAMPLE 1.3
Here are some examples for use of big-Oh notation:

1. If $f(n) = 100n \log n$ and $g(n) = n^2$ then we have the relations $f = O(g)$, $g = \Omega(f)$, $f = o(g)$, $g = \omega(f)$.

2. If $f(n) = 100n^2 + 24n + 2logn$ and $g(n) = n^2$ then $f = O(g)$. We will often write this relation as $f(n) = O(n^2)$. Note that we also have the relation $g = O(f)$ and hence $f = \Theta(g)$ and $g = \Theta(f)$.

3. If $f(n) = \min\{n, 10^6\}$ and $g(n) = 1$ for every n then $f = O(g)$. We use the notation $f = O(1)$ to denote this. Similarly, if h is a function that tends to infinity with n (i.e., for every c it holds that $h(n) > c$ for n sufficiently large) then we write $h = \omega(1)$.

4. If $f(n) = 2^n$ then for every number $c \in \mathbb{N}$, if $g(n) = n^c$ then $g = o(f)$. We sometimes write this as $2^n = n^{\omega(1)}$. Similarly, we also write $h(n) = n^{O(1)}$ to denote the fact that h is bounded from above by some polynomial. That is, there exist a number $c > 0$ such that for sufficiently large n, $h(n) \leq n^c$.

For more examples and explanations, see any undergraduate algorithms text such as [KT06, CLRS01] or Section 7.1 in Sipser's book [SIP96].

1.2   Modeling computation and efficiency

We start with an informal description of computation. Let f be a function that takes a string of bits (i.e., a member of the set $\{0, 1\}^*$) and outputs, say, either 0 or 1. Informally speaking, an *algorithm* for computing f is a set of mechanical rules, such that by following them we can compute $f(x)$ given any input $x \in \{0, 1\}^*$. The set of rules being followed is fixed (i.e., the same rules must work for all possible inputs) though each rule in this set may be applied arbitrarily many times. Each rule involves one or more of the following "elementary" operations:

1. Read a bit of the input.

2. Read a bit (or possibly a symbol from a slightly larger alphabet, say a digit in the set $\{0, \ldots, 9\}$) from the "scratch pad" or working space we allow the algorithm to use.

   Based on the values read,

3. Write a bit/symbol to the scratch pad.

4. Either stop and output 0 or 1, or choose a new rule from the set that will be applied next.

Finally, the *running time* is the number of these basic operations performed.
Below, we formalize all of these notions.

1.2.1   The Turing Machine

The *k-tape Turing machine* is a concrete realization of the above informal notion, as follows (see Figure 1.1).

Scratch Pad: The scratch pad consists of k tapes. A *tape* is an infinite one-directional line of cells, each of which can hold a symbol from a finite set Γ called the *alphabet* of the machine. Each tape is equipped with a *tape head* that can potentially read or write symbols to the tape one cell at a time. The machine's computation is divided into discrete time steps, and the head can move left or right one cell in each step.

The first tape of the machine is designated as the *input* tape. The machine's head can can only read symbols from that tape, not write them —a so-called read-only head.

The $k - 1$ read-write tapes are called *work tapes* and the last one of them is designated as the *output tape* of the machine, on which it writes its final answer before halting its computation.

Finite set of operations/rules: The machine has a finite set of *states*, denoted Q. The machine contains a "register" that can hold a single element of Q; this is the "state" of the machine at that instant. This state determines its action at the next computational step, which consists of the following: **(1)** read the symbols in the cells directly under the k heads **(2)** for the $k - 1$ read/write tapes replace each symbol with a new symbol (it has the option of not changing the tape by writing down the old symbol again), **(3)** change its register to contain another state from the finite set Q (it has the option not to change its state by choosing the old state again) and **(4)** move each head one cell to the left or to the right.

One can think of the Turing machine as a simplified modern computer, with the machine's tape corresponding to a computer's memory, and the transition function and register corresponding to the computer's central processing unit (CPU). However, it's best to think of Turing machines as simply a formal way to describe algorithms. Even though algorithms are often best described by plain English text, it is sometimes useful to express them by such a formalism in order to argue about them mathematically. (Similarly, one needs to express an algorithm in a programming language in order to execute it on a computer.)

Formal definition.   Formally, a TM M is described by a tuple (Γ, Q, δ) containing:

read only head

Input tape   > 0 0 0 1 1 0 1 0 0 0 1 0 0 0 0

read/write head

Work tape   > 1 1 0 1 0 1 0 0 0 1

read/write head

Output tape   > 0 1

Register q_7
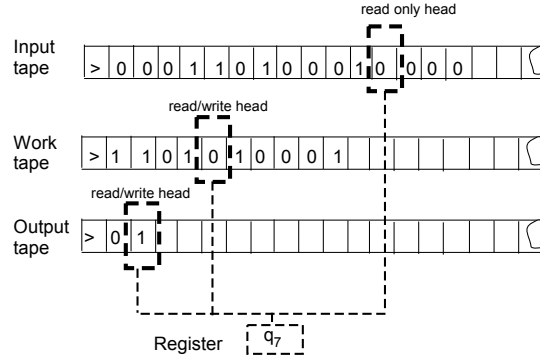
Figure 1.1: A snapshot of the execution of a 3-tape Turing machine M with an input tape, a work tape, and an output tape.

- A set Γ of the symbols that M's tapes can contain. We assume that Γ contains a designated "blank" symbol, denoted $\square$, a designated "start" symbol, denoted $\triangleright$ and the numbers 0 and 1. We call Γ the *alphabet* of M.

- A set Q of possible states M's register can be in. We assume that Q contains a designated start state, denoted q_{start} and a designated halting state, denoted q_{halt}.

- A function $\delta: Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{\mathsf{L}, \mathsf{S}, \mathsf{R}\}^k$ describing the rule M uses in performing each step. This function is called the *transition function* of M (see Figure 1.2.)

IF			THEN			
input symbol read	work/ output tape symbol read	current state	move input head	new work/ output tape symbol	move work/ output tape	new state
⋮	⋮	⋮	⋮	⋮	⋮	⋮
a	b	q	$\longrightarrow$	b'	$\longleftarrow$	q'
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 1.2: The transition function of a two tape TM (i.e., a TM with one input tape and one work/output tape).

If the machine is in state $q \in Q$ and $(\sigma_1, \sigma_2, \ldots, \sigma_k)$ are the symbols currently being read in the k tapes, and $\delta(q, (\sigma_1, \ldots, \sigma_{k+1})) = (q', (\sigma_2', \ldots, \sigma_k'), z)$ where $z \in \{\mathsf{L}, \mathsf{SR}\}^k$ then at the next step the σ symbols in the last $k-1$ tapes will be replaced by the σ' symbols, the machine will be in state

q', and the $k + 1$ heads will move Left, Right or Stay in place, as given by z. (If the machine tries to move left from the leftmost position of a tape then it will stay in place.)

All tapes except for the input are initialized in their first location to the *start* symbol $\triangleright$ and in all other locations to the *blank* symbol $\square$. The input tape contains initially the start symbol $\triangleright$, a finite non-blank string ("the input"), and the rest of its cells are initialized with the blank symbol $\square$. All heads start at the left ends of the tapes and the machine is in the special starting state q_{start}. This is called the *start configuration* of M on input x. Each step of the computation is performed by applying the function δ as described above. The special halting state q_{halt} has the property that once the machine is in q_{halt}, the transition function δ does not allow it to further modify the tape or change states. Clearly, if the machine enters q_{halt} then it has *halted*. In complexity theory we are typically only interested in machines that halt for every input in a finite number of steps.

Now we formalize the notion of running time. As every non-trivial algorithm needs to at least read its entire input, by "quickly" we mean that the number of basic steps we use is small *when considered as a function of the input length*.

DEFINITION 1.4 (COMPUTING A FUNCTION AND RUNNING TIME)
Let $f : \{0,1\}^* \to \{0,1\}^*$ and let $T : \mathbb{N} \to \mathbb{N}$ be some functions, and let M be a Turing machine. We say that M *computes f in $T(n)$-time*[2] if for every $x \in \{0,1\}^*$, if M is initialized to the start configuration on input x, then after at most $T(|x|)$ steps it halts with $f(x)$ written on its output tape.
We say that M *computes f* if it computes f in $T(n)$ time for some function $T : \mathbb{N} \to \mathbb{N}$.

REMARK 1.5 (TIME-CONSTRUCTIBLE FUNCTIONS)
We say that a function $T : \mathbb{N} \to \mathbb{N}$ is *time constructible* if $T(n) \geq n$ and there is a TM M that computes the function $x \mapsto \llcorner T(|x|) \lrcorner$ in time $T(n)$. (As usual, $\llcorner T(|x|) \lrcorner$ denotes the binary representation of the number $T(|x|)$.)

Examples for time-constructible functions are n, $n \log n$, n^2, 2^n. Almost all functions encountered in this book will be time-constructible and, to avoid annoying anomalities, we will restrict our attention to time bounds of this form. (The restriction $T(n) \geq n$ is to allow the algorithm time to read its input.)

EXAMPLE 1.6
Let PAL be the Boolean function defined as follows: for every $x \in \{0,1\}^*$, PAL(x) is equal to 1 if x is a *palindrome* and equal to 0 otherwise. That is, PAL$(x) = 1$ if and only if x reads the same from left to right as from right to left (i.e., $x_1 x_2 \ldots x_n = x_n x_{n-1} \ldots x_1$). We now show a TM M that computes PAL within less than $3n$ steps.

---

[2]Formally we should write "T-time" instead of "$T(n)$-time", but we follow the convention of writing $T(n)$ to emphasize that T is applied to the input length.

Our TM M will use 3 tapes (input, work and output) and the alphabet $\{\triangleright, \square, 0, 1\}$. It operates as follows:

1. Copy the input to the read/write work tape.

2. Move the input head to the beginning of the input.

3. Move the input-tape head to the right while moving the work-tape head to the left. If at any moment the machine observes two different values, it halts and output 0.

4. Halt and output 1.

We now describe the machine more formally: The TM M uses 5 states denoted by $\{q_{\mathsf{start}}, q_{\mathsf{copy}}, q_{\mathsf{right}}, q_{\mathsf{test}}, q_{\mathsf{halt}}\}$. Its transition function is defined as follows:

1. On state q_{start}, move the input-tape head to the right, and move the work-tape head to the right while writing the start symbol $\triangleright$; change the state to q_{copy}. (Unless we mention this explicitly, the function does not change the output tape's contents or head position.)

2. On state q_{copy}:

   • If the symbol read from the input tape is not the blank symbol $\square$ then move both the input-tape and work-tape heads to the right, writing the symbol from the input-tape on the work-tape; stay in the state q_{copy}.

   • If the symbol read from the input tape is the blank symbol $\square$, then move the input-tape head to the left, while keeping the work-tape head in the same place (and not writing anything); change the state to q_{right}.

3. On state q_{right}:

   • If the symbol read from the input tape is not the start symbol $\triangleright$ then move the input-head to the left, keeping the work-tape head in the same place (and not writing anything); stay in the state q_{right}.

   • If the symbol read from the input tape is the start symbol $\triangleright$ then move the input-tape to the right and the work-tape head to the left (not writing anything); change to the state q_{test}.

4. On state q_{test}:

   • If the symbol read from the input-tape is the blank symbol $\square$ and the symbol read from the work-tape is the start symbol $\triangleright$ then write 1 on the output tape and change state to q_{halt}.

   • Otherwise, if the symbols read from the input tape and the work tape are not the same then write 0 on the output tape and change state to q_{halt}.

- Otherwise, if the symbols read from the input tape and the work tape are the same, then move the input-tape head to the right and the work-tape head to the left; stay in the state q_{test}.

As you can see, fully specifying a Turing machine is somewhat tedious and not always very informative. While it is useful to work out one or two examples for yourself (see Exercise 4), in the rest of the book we avoid such overly detailed descriptions and specify TM's in a more high level fashion.

REMARK 1.7
Some texts use as their computational model *single tape* Turing machines, that have one read/write tape that serves as input, work and output tape. This choice does not make any difference for most of this book's results (see Exercise 10). However, Example 1.6 is one exception: it can be shown that such machines require $\Omega(n^2)$ steps to compute the function PAL.

1.2.2 Robustness of our definition.

Most of the specific details of our definition of Turing machines are quite arbitrary. For example, the following simple claims show that restricting the alphabet Γ to be $\{0, 1, \square, \triangleright\}$, restricting the machine to have a single work tape, or allowing the tapes to be infinite in both directions will not have a significant effect on the time to compute functions: (Below we provide only proof sketches for these claims; completing these sketches into full proofs is a very good way to gain intuition on Turing machines, see Exercises 5, 6 and 7.)

CLAIM 1.8
For every $f : \{0,1\}^* \to \{0,1\}$ and time-constructible $T : \mathbb{N} \to \mathbb{N}$, if f is computable in time $T(n)$ by a TM M using alphabet Γ then it is computable in time $4 \log |\Gamma| T(n)$ by a TM $\tilde{M}$ using the alphabet $\{0, 1, \square, \triangleright\}$.

Figure 1.3: We can simulate a machine M using the alphabet $\{\triangleright, \square, \mathsf{a}, \mathsf{b}, \ldots, \mathsf{z}\}$ by a machine M' using $\{\triangleright, \square, 0, 1\}$ via encoding every tape cell of M using 5 cells of M'.

PROOF SKETCH: Let M be a TM with alphabet Γ, k tapes, and state set Q that computes the function f in $T(n)$ time. We will show a TM $\tilde{M}$ computing f with alphabet $\{0, 1, \square, \triangleright\}$, k tapes and a set Q' of states which will be described below. The idea behind the proof is simple: one can encode any member of Γ using $\log |\Gamma|$ bits.[3] Thus, each of $\tilde{M}$'s work tapes will simply encode one

---

[3]Recall our conventions that log is taken to base 2, and non-integer numbers are rounded up when necessary.

of M's tapes: for every cell in M's tape we will have $\log|\Gamma|$ cells in the corresponding tape of $\tilde{M}$ (see Figure 1.3).

To simulate one step of M, the machine $\tilde{M}$ will: **(1)** use $\log|\Gamma|$ steps to read from each tape the $\log|\Gamma|$ bits encoding a symbol of Γ **(2)** use its state register to store the symbols read, **(3)** use M's transition function to compute the symbols M writes and M's new state given this information, **(3)** store this information in its state register, and **(4)** use $\log|\Gamma|$ steps to write the encodings of these symbols on its tapes.

One can verify that this can be carried out if $\tilde{M}$ has access to registers that can store M's state, k symbols in Γ and a counter from 1 to k. Thus, there is such a machine $\tilde{M}$ utilizing no more than $10|Q||\Gamma|^k k$ states. (In general, we can always simulate several registers using one register with a larger state space. For example, we can simulate three registers taking values in the sets A, B and C respectively with one register taking a value in the set $A \times B \times C$ which is of size $|A||B||C|$.)

It is not hard to see that for every input $x \in \{0,1\}^n$, if on input x the TM M outputs $f(x)$ within $T(n)$ steps, then $\tilde{M}$ will output the same value within less than $4\log|\Gamma|T(n)$ steps. ∎

CLAIM 1.9
For every $f : \{0,1\}^* \to \{0,1\}$, time-constructible $T : \mathbb{N} \to \mathbb{N}$, if f is computable in time $T(n)$ by a TM M using k tapes (plus additional input and output tapes) then it is computable in time $5kT(n)^2$ by a TM $\tilde{M}$ using only a single work tape (plus additional input and output tapes).
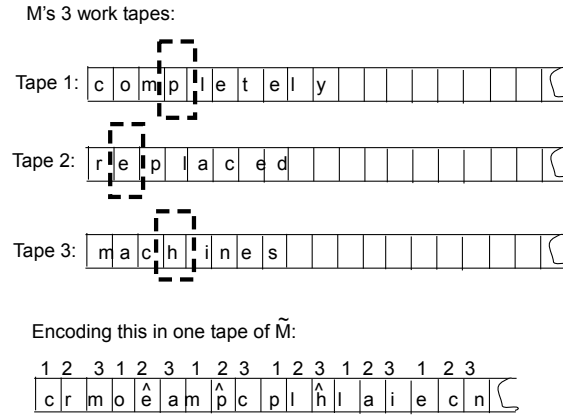
Figure 1.4: Simulating a machine M with 3 work tapes using a machine $\tilde{M}$ with a single work tape (in addition to the input and output tapes).

PROOF SKETCH: Again the idea is simple: the TM $\tilde{M}$ encodes the k tapes of M on a single tape by using locations $1, k+1, 2k+1, \ldots$ to encode the first tape, locations $2, k+2, 2k+2, \ldots$ to encode the second tape etc.. (see Figure 1.4). For every symbol a in M's alphabet, $\tilde{M}$ will contain both the symbol a and the symbol $\hat{a}$. In the encoding of each tape, exactly one symbol will be of the "ˆ type", indicating that the corresponding head of M is positioned in that location (see figure). $\tilde{M}$ uses the input and output tape in the same way M does. To simulate one step of M, the machine $\tilde{M}$ makes two sweeps of its work tape: first it sweeps the tape in the left-to-right direction and

records to its register the k symbols that are marked by ˆ. Then $\tilde{M}$ uses M's transition function to determine the new state, symbols, and head movements and sweeps the tape back in the right-to-left direction to update the encoding accordingly. Clearly, $\tilde{M}$ will have the same output as M. Also, since on n-length inputs M never reaches more than location $T(n)$ of any of its tapes, $\tilde{M}$ will never need to reach more than location $kT(n)$ of its work tape, meaning that for each the at most $T(n)$ steps of M, $\tilde{M}$ performs at most $5kT(n)$ work (sweeping back and forth requires about $2T(n)$ steps, and some additional steps may be needed for updating head movement and book keeping). ■

REMARK 1.10
With a bit of care, one can ensure that the proof of Claim 1.9 yields a TM $\tilde{M}$ with the following property: the head movements of $\tilde{M}$ are independent of the contents of its tapes but only on the input length (i.e., $\tilde{M}$ always performs a sequence of left to right and back sweeps of the same form regardless of what is the input). A machine with this property is called *oblivious* and the fact that every TM can be simulated by an oblivious TM will be useful for us later on (see Exercises 8 and 9 and the proof of Theorem 2.10).

CLAIM 1.11
Define a bidirectional *TM to be a TM whose tapes are infinite in* both *directions. For every $f : \{0,1\}^* \rightarrow \{0,1\}^*$ and time constructible $T : \mathbb{N} \rightarrow \mathbb{N}$, if f is computable in time $T(n)$ by a bidirectional TM M then it is computable in time $4T(n)$ by a standard (unidirectional) TM $\tilde{M}$.*
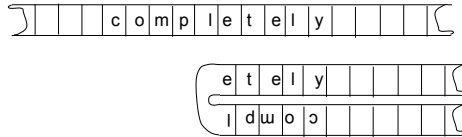
Figure 1.5: To simulate a machine M with alphabet Γ that has tapes infinite in both directions, we use a machine $\tilde{M}$ with alphabet Γ^2 whose tapes encode the "folded" version of M's tapes.

PROOF SKETCH: The idea behind the proof is illustrated in Figure 1.5. If M uses alphabet Γ then $\tilde{M}$ will use the alphabet Γ^2 (i.e., each symbol in $\tilde{M}$'s alphabet corresponds to a pair of symbols in M's alphabet). We encode a tape of M that is infinite in both direction using a standard (infinite in one direction) tape by "folding" it in an arbitrary location, with each location of $\tilde{M}$'s tape encoding two locations of M's tape. At first, $\tilde{M}$ will ignore the second symbol in the cell it reads and act according to M's transition function. However, if this transition function instructs $\tilde{M}$ to go "over the edge" of its tape then instead it will start ignoring the first symbol in each cell and use only the second symbol. When it is in this mode, it will translate left movements into right movements and vice versa. If it needs to go "over the edge" again then it will go back to reading the first symbol of each cell, and translating movements normally. ■

Other changes that will not have a very significant effect include having two or three dimensional tapes, allowing the machine *random access* to its tape, and making the output tape *write only* (see Exercises 11 and 12; also the texts [SIP96, HMU01] contain more examples). In particular none of these modifications will change the class **P** of polynomial-time computable decision problems defined below in Section 1.5.

1.2.3   The expressive power of Turing machines.

When you encounter Turing machines for the first time, it may not be clear that they do indeed fully encapsulate our intuitive notion of computation. It may be useful to work through some simple examples, such as expressing the standard algorithms for addition and multiplication in terms of Turing machines computing the corresponding functions (see Exercise 4). You can also verify that you can simulate a program in your favorite programming language using a Turing machine. (The reverse direction also holds: most programming languages can simulate a Turing machine.)

EXAMPLE 1.12
(This example assumes some background in computing.) We give a hand-wavy proof that Turing machines can simulate any program written in any of the familiar programming languages such as C or Java. First, recall that programs in these programming languages can be translated (the technical term is *compiled*) into an equivalent *machine language* program. This is a sequence of simple instructions to read from memory into one of a finite number of registers, write a register's contents to memory, perform basic arithmetic operations, such as adding two registers, and control instructions that perform actions conditioned on, say, whether a certain register is equal to zero.

All these operations can be easily simulated by a Turing machine. The memory and register can be implemented using the machine's tapes, while the instructions can be encoded by the machine's transition function. For example, it's not hard to show TM's that add or multiply two numbers, or a two-tape TM that, if its first tape contains a number i in binary representation, can move the head of its second tape to the ith location.

Exercise 13 asks you to give a more rigorous proof of such a simulation for a simple tailor-made programming language.

1.3   Machines as strings and the universal Turing machines.

It is almost obvious that a Turing machine can be represented as a string: since we can write the description of any TM M on paper, we can definitely encode this description as a sequence of zeros and ones. Yet this simple observation— that we can treat programs as data— has had far reaching consequences on both the theory and practice of computing. Without it, we would not have had *general purpose* electronic computers, that, rather than fixed to performing one task, can execute arbitrary programs.

DRAFT

Because we will use this notion of representing TM's as strings quite extensively, it may be worth to spell out our representation out a bit more concretely. Since the behavior of a Turing machine is determined by its transition function, we will use the list of all inputs and outputs of this function (which can be easily encoded as a string in $\{0,1\}^*$) as the encoding of the Turing machine.[4] We will also find it convenient to assume that our representation scheme satisfies the following properties:

1. Every string in $\{0,1\}^*$ represents *some* Turing machine.

   This is easy to ensure by mapping strings that are not valid encodings into some canonical trivial TM, such as the TM that immediately halts and outputs zero on any input.

2. Every TM is represented by infinitely many strings.

   This can be ensured by specifying that the representation can end with an arbitrary number of 1's, that are ignored. This has somewhat similar effect as the *comments* of many programming languages (e.g., the `/*...*/` construct in C,C++ and Java) that allows to add superfluous symbols to any program.

If M is a Turing machine, then we use $\llcorner M \lrcorner$ to denotes M's representation as a binary string. If α is a string then we denote the TM that α represents by M_α. As is our convention, we will also often use M to denote both the TM and its representation as a string. Exercise 14 asks you to fully specify a representation scheme for Turing machines with the above properties.

1.3.1 The Universal Turing Machine

It was Turing that first observed that general purpose computers are possible, by showing a *universal* Turing machine that can *simulate* the execution of every other TM M given M's description as input. Of course, since we are so used to having a universal computer on our desktops or even in our pockets, today we take this notion for granted. But it is good to remember why it was once counterintuitive. The parameters of the universal TM are fixed —alphabet size, number of states, and number of tapes. The corresponding parameters for the machine being simulated could be much larger. The reason this is not a hurdle is, of course, the ability to use *encodings*. Even if the universal TM has a very simple alphabet, say $\{0,1\}$, this is sufficient to allow it to represent the other machine's state and and transition table on its tapes, and then follow along in the computation step by step.

Now we state a computationally efficient version of Turing's construction due to Hennie and Stearns [HS66]. To give the essential idea we first prove a slightly relaxed variant where the term $T \log T$ below is replaced with T^2. But since the efficient version is needed a few times in the book, a full proof is also given at the end of the chapter (see Section 1.A).

---

[4]Note that the size of the alphabet, the number of tapes, and the size of the state space can be deduced from the transition function's table. We can also reorder the table to ensure that the special states $q_{\mathsf{start}}, q_{\mathsf{halt}}$ are the first 2 states of the TM. Similarly, we may assume that the symbols $\triangleright, \square, 0, 1$ are the first 4 symbols of the machine's alphabet.

THEOREM 1.13 (EFFICIENT UNIVERSAL TURING MACHINE)
There exists a TM $\mathcal{U}$ such that for every $x, \alpha \in \{0,1\}^*$, $\mathcal{U}(x, \alpha) = M_\alpha(x)$, where M_α denotes the TM represented by α.
Furthermore, if M_α halts on input x within T steps then $\mathcal{U}(x, \alpha)$ halts within $CT \log T$ steps, where C is a number independent of $|x|$ and depending only on M_α's alphabet size, number of tapes, and number of states.

REMARK 1.14
A common exercise in programming courses is to write an *interpreter* for a particular programming language using the same language. (An interpreter takes a program P as input and outputs the result of executing the program P.) Theorem 1.13 can be considered a variant of this exercise.

PROOF: Our universal TM $\mathcal{U}$ is given an input x, α, where α represents some TM M, and needs to output $M(x)$. A crucial observation is that we may assume that M **(1)** has a single work tape (in addition to the input and output tape) and **(2)** uses the alphabet $\{\triangleright, \square, 0, 1\}$. The reason is that $\mathcal{U}$ can transform a representation of every TM M into a representation of an equivalent TM $\tilde{M}$ that satisfies these properties as shown in the proofs of Claims 1.8 and 1.9. Note that these transformations may introduce a quadratic slowdown (i.e., transform M from running in T time to running in $C'T^2$ time where C' depends on M's alphabet size and number of tapes).
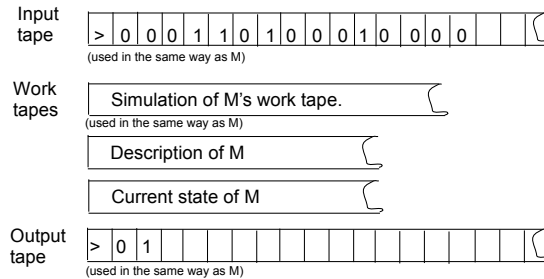
Figure 1.6: The universal TM $\mathcal{U}$ has in addition to the input and output tape, three work tapes. One work tape will have the same contents as the simulated machine M, another tape includes the description M (converted to an equivalent one-work-tape form), and another tape contains the current state of M.

The TM $\mathcal{U}$ uses the alphabet $\{\triangleright, \square, 0, 1\}$ and three work tapes in addition to its input and output tape (see Figure 1.6). $\mathcal{U}$ uses its input tape, output tape, and one of the work tapes in the same way M uses its three tapes. In addition, $\mathcal{U}$ will use its first extra work tape to store the table of values of M's transition function (after applying the transformations of Claims 1.8 and 1.9 as noted above), and its other extra work tape to store the current state of M. To simulate one computational step of M, $\mathcal{U}$ scans the table of M's transition function and the current state to find out the new state, symbols to be written and head movements, which it then executes. We see that each computational step of M is simulated using C steps of $\mathcal{U}$, where C is some number depending on the size of the transition function's table.

DRAFT

This high level description can turned into an exact specification of the TM $\mathcal{U}$, though we leave this to the reader. If you are not sure how this can be done, think first of how you would program these steps in your favorite programming language and then try to transform this into a description of a Turing machine. ∎

REMARK 1.15
It is sometimes useful to consider a variant of the universal TM $\mathcal{U}$ that gets a number t as an extra input (in addition to x and α), and outputs $M_\alpha(x)$ if and only if M_α halts on x within t steps (otherwise outputting some special failure symbol). By adding a counter to $\mathcal{U}$, the proof of Theorem 1.13 can be easily modified to give such a universal TM with the same efficiency.

1.4 Uncomputable functions.

It may seem "obvious" that every function can be computed, given sufficient time. However, this turns out to be false: there exist functions that cannot be computed within any finite number of steps!

THEOREM 1.16
There exists a function $\mathsf{UC} : \{0,1\}^* \to \{0,1\}$ that is not computable by any TM.

PROOF: The function UC is defined as follows: for every $\alpha \in \{0,1\}^*$, let M be the TM represented by α. If on input α, M halts within a finite number of steps and outputs 1 then $\mathsf{UC}(\alpha)$ is equal to 0, otherwise $\mathsf{UC}(\alpha)$ is equal to 1.

Suppose for the sake of contradiction that there exists a TM M such that $M(\alpha) = \mathsf{UC}(\alpha)$ for every $\alpha \in \{0,1\}^*$. Then, in particular, $M(\llcorner M \lrcorner) = \mathsf{UC}(\llcorner M \lrcorner)$. But this is impossible: by the definition of UC, if $\mathsf{UC}(\llcorner M \lrcorner) = 1$ then $M(\llcorner M \lrcorner)$ cannot be equal to 1, and if $\mathsf{UC}(\llcorner M \lrcorner) = 0$ then $M(\llcorner M \lrcorner)$ cannot be equal to 0. This proof technique is called "diagnoalization", see Figure 1.7. ∎

1.4.1 The Halting Problem

One might ask why should we care whether or not the function UC described above is computable— why would anyone want to compute such a contrived function anyway? We now show a more natural uncomputable function. The function HALT takes as input a pair α, x and outputs 1 if and only if the TM M_α represented by α halts on input x within a finite number of steps. This is definitely a function we want to compute: given a computer program and an input we'd certainly like to know if the program is going to enter an infinite loop on this input. Unfortunately, this is not possible, even if we were willing to wait an arbitrary long time:

THEOREM 1.17
HALT is not computable by any TM.

PROOF: Suppose, for the sake of contradiction, that there was a TM M_{HALT} computing HALT. We will use M_{HALT} to show a TM M_{UC} computing UC, contradicting Theorem 1.16.

The TM M_{UC} is simple: on input α, we run $M_{\mathsf{HALT}}(\alpha, \alpha)$. If the result is 0 (meaning that the machine represented by α does not halt on α) then we output 1. Otherwise, we use the universal

	0	1	00	01	10	11	...	α	....
0	Ø1	1	*	0	1	0		$M_0(\alpha)$	
1	1	1	0	1	*	1		...	
00	*	0	10	0	1	*			
01	1	*	0	Ø1	*	0			
...									
α	$M_\alpha(0)$	...						$M_\alpha(\alpha)$ 1-$M_\alpha(\alpha)$	
...									

Figure 1.7: Suppose we order all strings in lexicographic order, and write in a table the value of $M_\alpha(x)$ for all strings α, x, where M_α denotes the TM represented by the string α and we use $\star$ to denote the case that $M_\alpha(x)$ is not a value in $\{0, 1\}$ or that M_α does not halt on input x. Then, UC is defined by "negating" the diagonal of this table, and by its definition it cannot be computed by any TM.

TM $\mathcal{U}$ to compute $M(\alpha)$, where M is the TM represented by α. If $M(\alpha) = 0$ we output 1, and otherwise we output 1. Note that indeed, under the assumption that $M_{\mathsf{HALT}}(\alpha, x)$ outputs within a finite number of steps $\mathsf{HALT}(\alpha, x)$, the TM $M_{\mathsf{UC}}(\alpha)$ will output $\mathsf{UC}(\alpha)$ within a finite number of steps. ∎

REMARK 1.18
The proof technique employed to show Theorem 1.17— namely showing that HALT is uncomputable by showing an algorithm for UC using a hypothetical algorithm for HALT— is called a *reduction*. We will see many reductions in this book, often used (as is the case here) to show that a problem B is at least as hard as a problem A, by showing an algorithm that could solve A given a procedure that solves B.

There are many other examples for interesting uncomputable (also known as *undecidable*) functions, see Exercise 15. There are even uncomputable functions whose formulation has seemingly nothing to do with Turing machines or algorithms. For example, the following problem cannot be solved in finite time by any TM: given a set of polynomial equations with integer coefficients, find out whether these equations have an integer solution (i.e., whether there is an assignment of integers to the variables that satisfies the equations). This is known as the problem of solving Diophantine equations, and in 1900 Hilbert mentioned finding such algorithm to solve it (which he presumed to exist) as one of the top 23 open problems in mathematics.

For more on computability theory, see the chapter notes and the book's website.

1.5 Deterministic time and the class P.

A *complexity class* is a set of functions that can be computed within a given resource. We will now introduce our first complexity classes. For reasons of technical convenience, throughout most of this book we will pay special attention to Boolean functions (that have one bit output), also known as *decision problems* or *languages*. (Recall that we identify a Boolean function f with the language $L_f = \{x : f(x) = 1\}$.)

DEFINITION 1.19 (THE CLASS **DTIME**.)
Let $T : \mathbb{N} \to \mathbb{N}$ be some function. We let **DTIME**$(T(n))$ be the set of all Boolean (one bit output) functions that are computable in $c \cdot T(n)$-time for some constant $c > 0$.

The following class will serve as our rough approximation for the class of decision problems that are efficiently solvable.

DEFINITION 1.20 (THE CLASS **P**)
$\mathbf{P} = \cup_{c \geq 1}\mathbf{DTIME}(n^c)$

Thus, we can phrase the question from the introduction as to whether the dinner party problem has an efficient algorithm as follows: *"Is* INDSET *in* **P***?"*, where INDSET is the language defined in Example 1.6.

1.5.1 On the philosophical importance of P

The class **P** is felt to capture the notion of decision problems with "feasible" decision procedures. Of course, one may argue whether **DTIME**(n^{100}) really represents "feasible" computation in the real world. However, in practice, whenever we show that a problem is in **P**, we usually find an n^3 or n^5 time algorithm (with reasonable constants), and not an n^{100} algorithm. (It has also happened a few times that the first polynomial-time algorithm for a problem had high complexity, say n^{20}, but soon somebody simplified it to say an n^5 algorithm.)

Note that the class **P** is useful only in a certain context. Turing machines are a poor model if one is designing algorithms that must run in a fraction of a second on the latest PC (in which case one must carefully account for fine details about the hardware). However, if the question is whether any subexponential algorithms exist for say INDSET then even an n^{20} algorithm would be a fantastic breakthrough.

P is also a natural class from the viewpoint of a programmer. Suppose undergraduate programmers are asked to invent the definition of an "efficient" computation. Presumably, they would agree that a computation that runs in linear or quadratic time is "efficient." Next, since programmers often write programs that call other programs (or subroutines), they might find it natural to consider a program "efficient" if it performs only "efficient" computations and calls subroutines that are "efficient". The notion of "efficiency" obtained turns out to be exactly the class **P** [Cob64].

1.5.2 Criticisms of P and some efforts to address them

Now we address some possible criticisms of the definition of **P**, and some related complexity classes that address these.

Worst-case exact computation is too strict. The definition of **P** only considers algorithms that compute the function *exactly* on *every* possible input. However, not all possible inputs arise in practice (although it's not always easy to characterize the inputs that do). Chapter 15 gives a theoretical treatment of *average-case complexity* and defines the analogue of **P** in that context. Sometimes, users are willing to settle for *approximate* solutions. Chapter 18 contains a rigorous treatment of the complexity of approximation.

Other physically realizable models. If we were to make contact with an advanced alien civilization, would their class **P** be any different from the class defined here?

Most scientists believe the *Church-Turing (CT) thesis*, which states that every physically realizable computation device— whether it's silicon-based, DNA-based, neuron-based or using some alien technology— can be simulated by a Turing machine. Thus they believe that the set of *computable* problems would be the same for aliens as it is for us. (The CT thesis is not a theorem, merely a belief about the nature of the world.)

However, when it comes to *efficiently* computable problems, the situation is less clear. The **strong form of the CT thesis** says that every physically realizable computation model can be simulated by a TM *with polynomial overhead* (in other words, t steps on the model can be simulated in t^c steps on the TM, where c is a constant that depends upon the model). If true, it implies that the class **P** defined by the aliens will be the same as ours. However, several objections have been made to this strong form.

(a) *Issue of precision:* TM's compute with discrete symbols, whereas physical quantities may be real numbers in $\mathbb{R}$. Thus TM computations may only be able to approximately simulate the real world. Though this issue is not perfectly settled, it seems so far that TMs do not suffer from an inherent handicap. After all, real-life devices suffer from noise, and physical quantities can only be measured up to finite precision. Thus a TM could simulate the real-life device using finite precision. (Note also that we often only care about the *most significant bit* of the result, namely, a 0/1 answer.)

Even so, in Chapter 14 we also consider a modification of the TM model that allows computations in $\mathbb{R}$ as a basic operation. The resulting complexity classes have fascinating connections with the usual complexity classes.

(b) *Use of randomness:* The TM as defined is *deterministic.* If randomness exists in the world, one can conceive of computational models that use a source of random bits (i.e., "coin tosses"). Chapter 7 considers Turing Machines that are allowed to also toss coins, and studies the class **BPP**, that is the analogue of **P** for those machines. (However, we will see in Chapters 16 and 17 the intriguing possibility that randomized computation may be no more powerful than deterministic computation.)

(c) *Use of quantum mechanics:* A more clever computational model might use some of the counterintuitive features of quantum mechanics. In Chapter 20 we define the class **BQP**,

that generalizes **P** in such a way. We will see problems in **BQP** that are currently not known to be in **P**. However, currently it is unclear whether the quantum model is truly physically realizable. Even if it is realizable it currently seems only able to efficiently solve only very few "well-structured" problems that are (presumed to be) not in **P**. Hence insights gained from studying **P** could still be applied to **BQP**.

(d) *Use of other exotic physics, such as string theory.* Though an intriguing possibility, it hasn't yet had the same scrutiny as quantum mechanics.

Decision problems are too limited. Some computational problems are not easily expressed as decision problems. Indeed, we will introduce several classes in the book to capture tasks such as computing non-Boolean functions, solving search problems, approximating optimization problems, interaction, and more. Yet the framework of decision problems turn out to be surprisingly expressive, and we will often use it in this book.

1.5.3 Edmonds' quote

We conclude this section with a quote from Edmonds [Edm65], that in the paper showing a polynomial-time algorithm for the maximum matching problem, explained the meaning of such a result as follows:

> *For practical purposes computational details are vital. However, my purpose is only to show as attractively as I can that there is an efficient algorithm. According to the dictionary, "efficient" means "adequate in operation or performance." This is roughly the meaning I want in the sense that it is conceivable for maximum matching to have no efficient algorithm.*
>
> *...There is an obvious finite algorithm, but that algorithm increases in difficulty exponentially with the size of the graph. It is by no means obvious whether or not there exists an algorithm whose difficulty increases only algebraically with the size of the graph.*
>
> *...When the measure of problem-size is reasonable and when the sizes assume values arbitrarily large, an asymptotic estimate of ... the order of difficulty of an algorithm is theoretically important. It cannot be rigged by making the algorithm artificially difficult for smaller sizes.*
>
> *...One can find many classes of problems, besides maximum matching and its generalizations, which have algorithms of exponential order but seemingly none better ... For practical purposes the difference between algebraic and exponential order is often more crucial than the difference between finite and non-finite.*
>
> *...It would be unfortunate for any rigid criterion to inhibit the practical development of algorithms which are either not known or known not to conform nicely to the criterion. Many of the best algorithmic idea known today would suffer by such theoretical pedantry. ... However, if only to motivate the search for good, practical algorithms, it is important to realize that it is mathematically sensible even to question their existence. For one thing the task can then be described in terms of concrete conjectures.*

WHAT HAVE WE LEARNED?

- There are many equivalent ways to mathematically model computational processes; we use the standard Turing machine formalization.

- Turing machines can be represented as strings. There is a *universal* TM that can emulate (with small overhead) any TM given its representation.

- There exist functions, such as the Halting problem, that cannot be computed by any TM regardless of its running time.

- The class **P** consists of all decision problems that are solvable by Turing machines in polynomial time. We say that problems in **P** are efficiently solvable.

- All low-level choices (number of tapes, alphabet size, etc..) in the definition of Turing machines are immaterial, as they will not change the definition of **P**.

Chapter notes and history

Although certain algorithms have been studied for thousands of years, and some forms of computing devices were designed before the 20th century (most most notably Charles Babbage's difference and analytical engines in the mid 1800's), it seems fair to say that the foundations of modern computer science were only laid in the 1930's.

In 1931, Kurt Gödel shocked the mathematical world by showing that certain true statements about the natural numbers are *inherently unprovable*, thereby shattering an ambitious agenda set in 1900 by David Hilbert to base all of mathematics on solid axiomatic foundations. In 1936, Alonzo Church defined a model of computation called λ-calculus (which years later inspired the programming language LISP) and showed the existence of functions *inherently uncomputable* in this model [Chu36]. A few months later, Alan Turing independently introduced his Turing machines and showed functions inherently uncomputable by such machines [Tur36]. Turing also introduced the idea of the *universal* Turing machine that can be loaded with arbitrary programs. The two models turned out to be equivalent, but in the words of Church himself, Turing machines have "the advantage of making the identification with effectiveness in the ordinary (not explicitly defined) sense evident immediately". The anthology [Dav65] contains many of the seminal papers in the theory of computability. Part II of Sipser's book [SIP96] is a good gentle introduction to this theory, while the books [**?**, HMU01, Koz97] go into a bit more depth. This book's web site also contains some additional coverage of this theory.

During World War II Turing designed mechanical code-breaking devices and played a key role in the effort to crack the German "Enigma" cipher, an achievement that had a decisive effect on the war's progress (see the biographies [Hod83, Lea05]).[5] After World War II, efforts to build electronic universal computers were undertaken in both sides of the Atlantic. A key figure in these

---

[5]Unfortunately, Turing's wartime achievements were kept confidential during his lifetime, and so did not keep him from being forced by British courts to take hormones to "cure" his homosexuality, resulting in his suicide in 1954.

efforts was John von-Neumann, an extremely prolific scientist that was involved in anything from the Manhattan project to founding game theory in economics. To this day essentially all digital computers follow the "von-Neumann architecture" he pioneered while working on the design of the EDVAC, one of the earliest digital computers [vN45].

As computers became more prevalent, the issue of efficiency in computation began to take center stage. Cobham [Cob64] defined the class **P** and suggested it may be a good formalization for efficient computation. A similar suggestion was made by Edmonds ([Edm65], see quote above) in the context of presenting a highly non-trivial polynomial-time algorithm for finding a maximum matching in general graphs. Hartmanis and Stearns [HS65] defined the class **DTIME**$(T(n))$ for every function T, and proved the slightly relaxed version of Theorem 1.13 we showed above (the version we stated and prove below was given by Hennie and Stearns [HS66]). They also coined the name "computational complexity" and proved an interesting "speed-up theorem": if a function f is computable by a TM M in time $T(n)$ then for every constant $c \geq 1$, f is computable by a TM $\tilde{M}$ (possibly with larger state size and alphabet size than M) in time $T(n)/c$. This speed-up theorem is another justification for ignoring constant factors in the definition of **DTIME**$(T(n))$. Blum [Blu67] suggested an axiomatic formalization of complexity theory, that does not explicitly mention Turing machines.

We have omitted a discussion of some of the "bizarre conditions" that may occur when considering time bounds that are not time-constructible, especially "huge" time bounds (i.e., function $T(n)$ that are much larger than exponential in n). For example, there is a non-time constructible function $T : \mathbb{N} \to \mathbb{N}$ such that every function computable in time $T(n)$ can also be computed in the much shorter time $\log T(n)$. However, we will not encounter non time-constructible time bounds in this book.

Exercises

§1 For each of the following pairs of functions f, g determine whether $f = o(g)$, $g = o(f)$ or $f = \Theta(g)$. If $f = o(g)$ then find the first number n such that $f(n) < g(n)$:

(a) $f(n) = n^2$, $g(n) = 2n^2 + 100\sqrt{n}$.

(b) $f(n) = n^{100}$, $g(n) = 2^{n/100}$.

(c) $f(n) = n^{100}$, $g(n) = 2^{n^{1/100}}$.

(d) $f(n) = \sqrt{n}, g(n) = 2^{\sqrt{\log n}}$.

(e) $f(n) = n^{100}, g(n) = 2^{(\log n)^2}$.

(f) $f(n) = 1000n, g(n) = n \log n$.

§2 For each of the following recursively defined functions f, find a closed (non-recursive) expression for a function g such that $f(n) = \Theta(g(n))$.

(Note: below we only supply the recursive rule, you can assume that $f(1) = f(2) = \cdots = f(10) = 1$ and the recursive rule is applied for $n > 10$; in any case regardless how the base case it won't make any difference to the answer - can you see why?)

(a) $f(n) = f(n-1) + 10$.

(b) $f(n) = f(n-1) + n$.

(c) $f(n) = 2f(n-1)$.

(d) $f(n) = f(n/2) + 10$.

(e) $f(n) = f(n/2) + n$.

(f) $f(n) = 2f(n/2) + n$.

(g) $f(n) = 3f(n/2)$.

§3 The MIT museum contains a kinetic sculpture by Arthur Ganson called "Machine with con-
crete" (see Figure 1.8). It consists of 13 gears connected to one another in a series such that
each gear moves 50 times slower than the previous one. The fastest gear is constantly rotated
by an engine at a rate of 212 rotations per minute. The slowest gear is fixed to a block of
concrete and so apparently cannot move at all. How come this machine does not break apart?
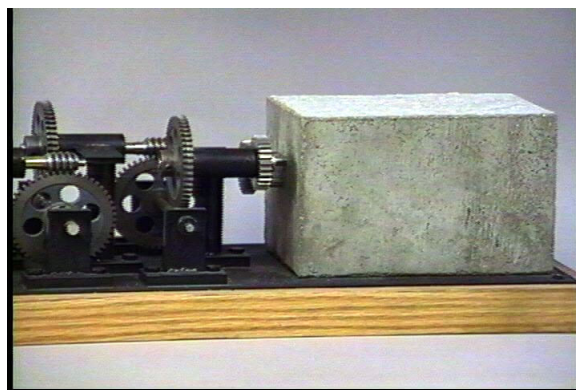
Figure 1.8: Machine with concrete by Arthur Ganson.

§4 Let f be the *addition* function that maps the representation of a pair of numbers x, y to the
representation of the number $x + y$. Let g be the *multiplication* function that maps $\langle x, y \rangle$ to
$\lfloor x \cdot y \rfloor$. Prove that both f and g are computable by writing down a full description (including
the states, alphabet and transition function) of the corresponding Turing machines.

Hint: Follow the gradeschool algorithms.

§5 Complete the proof of Claim 1.8 by writing down explicitly the description of the machine
$\tilde{M}$.

§6 Complete the proof of Claim 1.9.

§7 Complete the proof of Claim 1.11.

§8 Define a TM M to be *oblivious* if its head movement does not depend on the input but only on the input length. That is, M is oblivious if for every input $x \in \{0,1\}^*$ and $i \in \mathbb{N}$, the location of each of M's heads at the ith step of execution on input x is only a function of $|x|$ and i. Show that for every time-constructible $T : \mathbb{N} \to \mathbb{N}$, if $L \in \mathbf{DTIME}(T(n))$ then there is an oblivious TM that decides L in time $O(T(n)^2)$. Furthermore, show that there is such a TM that uses only *two tapes*: one input tape and one work/output tape.

Hint: Use the proof of Claim 1.9.

§9 Show that for every time-constructible $T : \mathbb{N} \to \mathbb{N}$, if $L \in \mathbf{DTIME}(T(n))$ then there is an oblivious TM that decides L in time $O(T(n) \log T(n))$.

Hint: show that the universal TM $\mathcal{U}$ obtained by the proof of Theorem 1.13 can be tweaked to be oblivious.

§10 Define a *single-tape* Turing machine to be a TM that has only one read/write tape, that is used as input, work and output tape. Show that for every (time-constructible) $T : \mathbb{N} \to \mathbb{N}$ and $f \in \mathbf{DTIME}(T(n))$, f can be computed in $O(T(n)^2)$ steps by a single-tape TM.

§11 Define a *two dimensional* Turing machine to be a TM where each of its tapes is an infinite grid (and the machine can move not only Left and Right but also Up and Down). Show that for every (time-constructible) $T : \mathbb{N} \to \mathbb{N}$ and every Boolean function f, if g can be computed in time $T(n)$ using a two-dimensional TM then $f \in \mathbf{DTIME}(T(n)^2)$.

§12 Define a *RAM Turing machine* to be a Turing machine that has *random access memory*. We formalize this as follows: the machine has additional two symbol on its alphabet we denote by R and W and an additional state we denote by q_{access}. We also assume that the machine has an infinite array A that is initialized to all blanks. Whenever the machine enters q_{access}, if its address tape contains $\llcorner i \lrcorner$R (where $\llcorner i \lrcorner$ denotes the binary representation of i) then the value $A[i]$ is written in the cell next to the R symbol. If its tape contains $\llcorner i \lrcorner$Wσ (where σ is some symbol in the machine's alphabet) then $A[i]$ is set to the value σ.

Show that if a Boolean function f is computable within time $T(n)$ (for some time-constructible T) by a RAM TM, then it is in $\mathbf{DTIME}(T(n)^2)$.

§13 Consider the following simple programming language. It has a single infinite array A of elements in $\{0, 1, \square\}$ (initialized to $\square$) and a single integer variable i. A program in this language contains a sequence of lines of the following form:

label : If A[i] equals σ then *cmds*

Where $\sigma \in \{0, 1, \square\}$ and *cmds* is a list of one or more of the following commands: **(1)** Set A[i] to τ where $\tau \in \{0, 1, \square\}$, **(2)** Goto *label*, **(3)** Increment i by one, **(4)** Decrement i by one, and **(5)** Output b and halt. where $b \in \{0, 1\}$. A program is executed on an input $x \in \{0, 1\}^n$ by placing the ith bit of x in A[i] and then running the program following the obvious semantics.

Prove that for every functions $f : \{0,1\}^* \to \{0,1\}$ and (time constructible) $T : \mathbb{N} \to \mathbb{N}$, if f is computable in time $T(n)$ by a program in this language, then $f \in \mathbf{DTIME}(T(n))$.

§14 Give a full specification of a representation scheme of Turing machines as binary string strings. That is, show a procedure that transforms any TM M (e.g., the TM computing the function PAL described in Example 1.6) into a binary string $\llcorner M \lrcorner$. It should be possible to recover M from $\llcorner M \lrcorner$, or at least recover a functionally equivalent TM (i.e., a TM $\tilde{M}$ computing the same function as M with the same running time).

§15 A *partial* function from $\{0,1\}^*$ to $\{0,1\}^*$ is a function that is not necessarily defined on all its inputs. We say that a TM M computes a partial function f if for every x on which f is defined, $M(x) = f(x)$ and for every x on which f is not defined M gets into an infinite loop when executed on input x. If $\mathcal{S}$ is a set of partial functions, we define $f_{\mathcal{S}}$ to be the Boolean function that on input α outputs 1 iff M_α computes a partial function in $\mathcal{S}$. *Rice's Theorem* says that for every non-trivial $\mathcal{S}$ (a set that is not the empty set nor the set of all partial functions), the $f_{\mathcal{S}}$ is not computable.

(a) Show that Rice's Theorem yields an alternative proof for Theorem 1.17 by showing that the function HALT is not computable.

(b) Prove Rice's Theorem.

> HALT$_x$ thereby deriving Rice's Theorem from Theorem 1.17.
> M_α halts on input x. Then reduce computing HALT to computing
> $f_{\mathcal{S}}$ can compute the function HALT$_x$ which outputs 1 on input α iff
> that is not in $\mathcal{S}$. Use this to show that an algorithm to compute
> is in $\mathcal{S}$ there is some function f that is defined on some input x
> assume that the empty function $\emptyset$ (that is not defined on any input)
> **Hint:** By possible changing from $\mathcal{S}$ to its complement, we may

§16 Prove that the following languages/decision problems on graphs are in **P**: (You may pick either the adjacency matrix or adjacency list representation for graphs; it will not make a difference. Can you see why?)

(a) CONNECTED — the set of all connected graphs. That is, $G \in$ CONNECTED if every pair of vertices u, v in G are connected by a path.

(b) TRIANGLEFREE — the set of all graphs that do not contain a triangle (i.e., a triplet u, v, w of connected distinct vertices.

(c) BIPARTITE — the set of all bipartite graphs. That is, $G \in$ BIPARTITE if the vertices of G can be partitioned to two sets A, B such that all edges in G are from a vertex in A to a vertex in B (there is no edge between two members of A or two members of B).

(d) TREE — the set of all trees. A graph is a *tree* if it is connected and contains no cycles. Equivalently, a graph G is a tree if every two distinct vertices u, v in G are connected by exactly one simple path (a path is simple if it has no repeated vertices).

§17 Recall that normally we assume that numbers are represented as string using the *binary* basis. That is, a number n is represented by the sequence $x_0, x_1, \ldots, x_{\log n}$ such that $n = \sum_{i=0}^{n} x_i 2^i$. However, we could have used other encoding schemes. If $n \in \mathbb{N}$ and $b \geq 2$, then *the representation of n in base b*, denoted by $\llcorner n \lrcorner_b$ is obtained as follows: first represent n as a sequence of digits in $\{0, \ldots, b-1\}$, and then replace each digit by a sequence of zeroes and ones. The *unary* representation of n, denoted by $\llcorner n \lrcorner_{\text{unary}}$ is the string 1^n (i.e., a sequence of n ones).

(a) Show that choosing a different base of representation will make no difference to the class **P**. That is, show that for every subset S of the natural numbers, if we define $L_S^b = \{ \llcorner n \lrcorner_b : n \in S \}$ then for every $b \geq 2$, $L_S^b \in \mathbf{P}$ iff $L_S^2 \in \mathbf{P}$.

(b) Show that choosing the unary representation make make a difference by showing that the following language is in **P**:

$$\text{UNARYFACTORING} = \{ \langle \llcorner n \lrcorner_{\text{unary}}, \llcorner \ell \lrcorner_{\text{unary}}, \llcorner k \lrcorner_{\text{unary}} \rangle : \text{there is } j \in (\ell, k) \text{ dividing } n \}$$

It is not known to be in **P** if we choose the binary representation (see Chapters 10 and 20). In Chapter 3 we will see that there is a problem that is *proven* to be in **P** when choosing the unary representation but not in **P** when using the binary representation.

1.A Proof of Theorem 1.13: Universal Simulation in $O(T \log T)$-time

We now show how to prove Theorem 1.13 as stated. That is, we show a universal TM $\mathcal{U}$ such that given an input x and a description of a TM M that halts on x within T steps, $\mathcal{U}$ outputs $M(x)$ within $O(T \log T)$ time (where the constants hidden in the O notation may depend on the parameters of the TM M being simulated).

The general structure of $\mathcal{U}$ will be as in Section 1.3.1, using the input and output tape as M does, and with extra work tapes to store M's transition table and current state. We will also have another "scratch" work tape to assist in certain computation. The main obstacle we need to overcome is that we cannot use Claim 1.9 to reduce the number of M's work tapes to one, as that claim introduces too much of overhead in the simulation. Therefore, we need to show a different way to encode all of M's work tapes using one tape of $\mathcal{U}$.

Let k be the number of tapes that M uses and Γ its alphabet. Following the proof of Claim 1.8, we may assume that $\mathcal{U}$ uses the alphabet Γ^k (as this can be simulated with a overhead depending only on $|\Gamma|$). Thus we can encode in each cell of $\mathcal{U}$'s work tape k symbols of Γ, each corresponding to a symbol from one of M's tapes. However, we still have to deal with the fact that M has k read/write heads that can each move independently to the left or right, whereas $\mathcal{U}$'s work tape only has a single head. Paraphrasing the famous saying, our strategy to handle this can be summarized as follows:

"If Muhammad will not come to the mountain then the mountain will go to Muhammad".

That is, since we can not move $\mathcal{U}$'s read/write head in different directions at once, we simply move the tape "under" the head. To be more specific, since we consider $\mathcal{U}$'s alphabet to be Γ^k, we can think of $\mathcal{U}$'s main work tape not as a single tape but rather k *parallel tapes*; that is, we can think of $\mathcal{U}$ as having k tapes with the property that in each step either all their read/write heads go in unison one location to the left or they all go one location to the right (see Figure 1.9).

To simulate a single step of M we shift all the non-blank symbols in each of these parallel tapes until the head's position in these parallel tapes corresponds to the heads' positions of M's k tapes. For example, if $k = 3$ and in some particular step M's transition function specifies the movements L, R, R then $\mathcal{U}$ will shift all the non-blank entries of its first parallel tape one cell to the right, and shift the non-blank entries of its second and third tapes one cell to the left. ($\mathcal{U}$ can easily perform these shifts using the additional "scratch" work tape.)
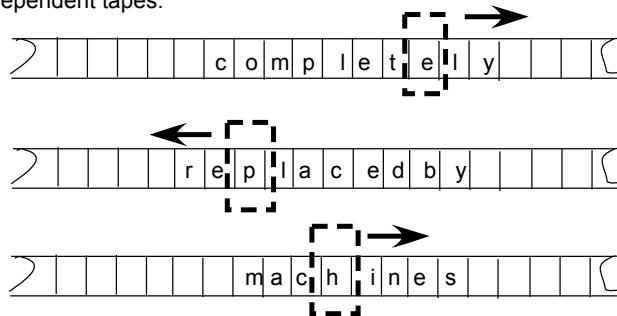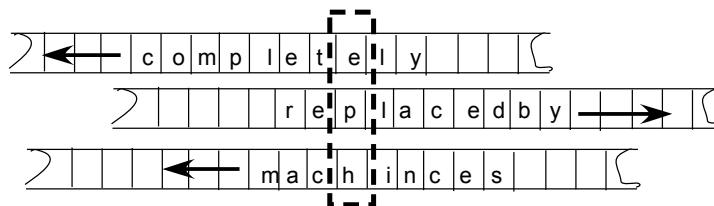
Figure 1.9: Packing k tapes of M into one tape of $\mathcal{U}$. We consider $\mathcal{U}$'s single work tape to be composed of k parallel tapes, whose heads move in unison, and hence we shift the contents of these tapes to simulate independent head movement.

The approach above is still not good enough to get $O(T \log T)$-time simulation. The reason is that there may be as much as T non-blank symbols in each tape, and so each shift operation may cost $\mathcal{U}$ at least T operations per each step of M. Our approach to deal with this is to create "buffer zones": rather than having each of $\mathcal{U}$'s parallel tapes correspond exactly to a tape of M, we add a special kind of blank symbol $\boxtimes$ to the alphabet of $\mathcal{U}$'s parallel tapes with the semantics that this symbol is ignored in the simulation. For example, if the non-blank contents of M's tape are 010 then this can be encoded in the corresponding parallel tape of $\mathcal{U}$ not just by 010 but also by $0\boxtimes01$ or $0\boxtimes\boxtimes1\boxtimes0$ and so on.
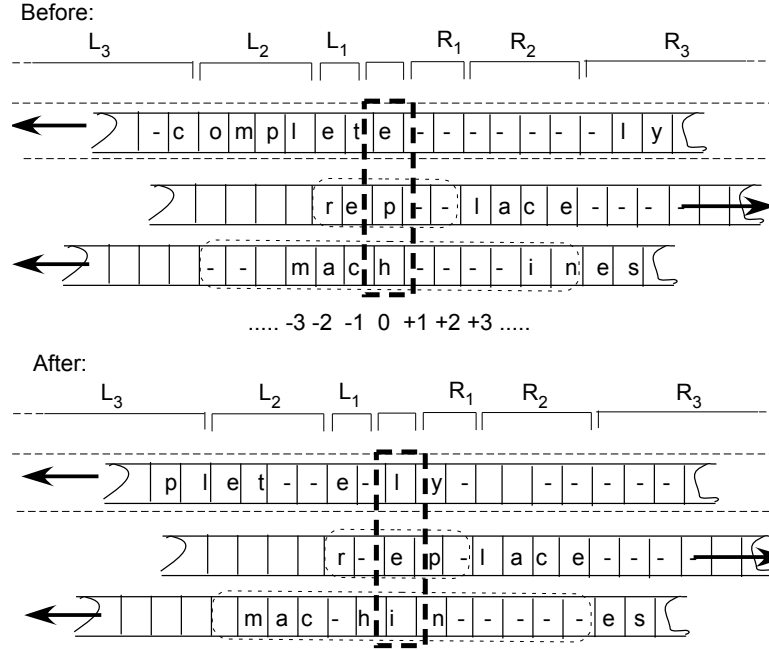
Figure 1.10: Performing a shift of the parallel tapes. The left shift of the first tape involves zones $L_1, R_1, L_2, R_2, L_3, R_3$, the right shift of the second tape involves only L_1, R_1, while the left shift of the third tape involves zones L_1, R_1, L_2, R_2. We maintain the invariant that each zone is either empty, half-full or full. Note that – denotes ⊠.

For convenience, we think of $\mathcal{U}$'s parallel tapes as infinite in both the left and right directions (this can be easily simulated with minimal overhead, see Claim 1.11). Thus, we index their locations by $0, \pm 1, \pm 2, \ldots$. Normally we keep $\mathcal{U}$'s head on location 0 of these parallel tapes. We will only move it temporarily to perform a shift when, following our general approach, we simulate a left head movement by shifting the tape to the right and vice versa. At the end of the shift we return the head to location 0.

We split the tapes into zones $L_0, R_0, L_1, R_1, \ldots$ (we'll only need to go up to $L_{\log T+1}, R_{\log T+1}$) where zone L_i contains the 2^i cells in the interval $[2^i..2^{i+1} - 1]$ and zone R_i contains the cells in the interval $[-2^{i+1} + 1.. - 2^i]$ (location 0 is not in any zone). We shall always maintain the following invariants:

- Each of the zones is either *empty*, *full*, or *half-full* with non-⊠ symbols. That is, the number of symbols in zone L_i that are not ⊠ is either $0, 2^{i-1}$, or 2^i and the same holds for R_i. (We treat the ordinary □ symbol the same as any other symbol in Γ and in particular a zone full of □'s is considered full.)

  We assume that initially all the zones are half-full. We can ensure this by filling half of each zone with ⊠ symbols in the first time we encounter it.

- The total number of non-⊠ symbols in $L_i \cup R_i$ is 2^i. That is, if L_i is full then R_i is empty and vice versa.

- Location 0 always contains a non-$\boxtimes$ symbol.

The advantage in setting up these zones is that now when performing the shifts, we do not always have to move the entire tape, but by using the "buffer zones" made up of $\boxtimes$ symbols, we can restrict ourselves to only using some of the zones. We illustrate this by showing how $\mathcal{U}$ performs a left shift on the first of its parallel tapes (see Figure 1.10):

1. $\mathcal{U}$ finds the smallest i such that R_i is not empty. Note that this is also the smallest i such that L_i is not full.

2. $\mathcal{U}$ puts the leftmost non-$\boxtimes$ symbol of R_i in position 0 and shifts the remaining leftmost $2^{i-1}-1$ non-$\boxtimes$ symbols from R_i into the zones $R_0, \ldots, R_{i-1}$ filling up exactly half the symbols of each zone. Note that there is room to perform this since all the zones $R_0, \ldots, R_{i-1}$ were empty and that indeed $2^{i-1} = \sum_{j=0}^{i-2} 2^j + 1$.

3. $\mathcal{U}$ performs the symmetric operation to the left of position 0: it shifts into L_i the 2^{i-1} leftmost symbols in the zones $L_{i-1}, \ldots, L_1$ and reorganizes $L_{i-1}, \ldots, L_i$ such that the remaining $\sum_{j=1}^{i-1} 2^j - 2^{i-1} = 2^{i-1} - 1$ symbols, plus the symbol that was originally in position 0 (modified appropriately according to M's transition function) take up exactly half of each of the zones $L_{i-1}, \ldots, L_i$.

4. Note that at the end of the shift, all of the zones $L_0, R_0, \ldots, L_{i-1}, R_{i-1}$ are half-full and so we haven't violated our invariant.

Performing such a shift costs $O(\sum_{j=1}^{i} 2^j) = O(2^i)$ operations. However, once we do this, we will not touch L_i again until we perform at least 2^{i-1} shifts (since now the zones $L_0, R_0, \ldots, L_{i-1}, R_{i-1}$ are half-full). Thus, when simulating T steps of M, we perform a shift involving L_i and R_i during the simulation of at most a $\frac{1}{2^{i-1}}$ fraction of these steps. Thus, the total number of operations used by these shifts is when simulating T steps is

$$O\left(\sum_{i=1}^{\log T + 1} \frac{T}{2^{i-1}} 2^i \right) = O(T \log T).$$

■

DRAFT

Chapter 2

NP and NP completeness

> "(if $\phi(n) \approx Kn^2$)[a] then this would have consequences of the greatest magnitude. That is to say, it would clearly indicate that, despite the unsolvability of the (Hilbert) Entscheidungsproblem, the mental effort of the mathematician in the case of the yes-or-no questions would be completely replaced by machines.... (this) seems to me, however, within the realm of possibility."
> Kurt Gödel in a letter to John von Neumann, 1956

> ---
> [a]In modern terminology, if SAT has a quadratic time algorithm

> "I conjecture that there is no good algorithm for the traveling salesman problem. My reasons are the same as for any mathematical conjecture: (1) It is a legitimate mathematical possibility, and (2) I do not know."
> Jack Edmonds, 1966

> "In this paper we give theorems that suggest, but do not imply, that these problems, as well as many others, will remain intractable perpetually."
> Richard Karp, 1972

If you have ever attempted a crossword puzzle, you know that there is often a big difference between solving a problem from scratch and verifying a given solution. In the previous chapter we encountered **P**, the class of decision problems that can be efficiently solved. In this chapter, we define the complexity class **NP** that aims to capture the set of problems whose solutions can be efficiently *verified*. The famous **P** versus **NP** question asks whether or not the two are the same. The resolution of this conjecture will be of great practical, scientific and philosophical interest; see Section 2.7.

This chapter also introduces **NP**-*completeness*, an important class of computational problems that are in **P** if and only if **P** = **NP**. Notions such as *reductions* and *completeness* encountered in this study motivate many other definitions encountered later in the book.

2.1 The class NP

As mentioned above, the complexity class **NP** will serve as our formal model for the class of
problems having efficiently verifiable solutions: a decision problem / language is in **NP** if given an
input x, we can easily verify that x is a YES instance of the problem (or equivalently, x is in the
language) *if* we are given the polynomial-size *solution* for x, that *certifies* this fact. We will give
several equivalent definitions for **NP**. The first one is as follows:

DEFINITION 2.1 (THE CLASS **NP**)
A language $L \subseteq \{0,1\}^*$ is in **NP** if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a
polynomial-time TM M such that for every $x \in \{0,1\}^*$,

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x,u) = 1$$

If $x \in L$ and $u \in \{0,1\}^{p(|x|)}$ satisfy $M(x,u) = 1$ then we call u a *certificate*[1]for x
(with respect to the language L and machine M).

EXAMPLE 2.2
Here are a few examples for decision problems in **NP**:

Independent set: (See Example 1.1 in the previous chapter.) Given a graph G and a number k,
decide if there is a k-size independent subset of G's vertices. The certificate for membership
is the list of k vertices forming an independent set.

Traveling salesperson: Given a set of n nodes, $\binom{n}{2}$ numbers $d_{i,j}$ denoting the distances between
all pairs of nodes, and a number k, decide if there is a closed circuit (i.e., a "salesperson
tour") that visits every node exactly once and has total length at most k. The certificate is
the sequence of nodes in the tour.

Subset sum: Given a list of n numbers $A_1, \ldots, A_n$ and a number T, decide if there is a subset of
the numbers that sums up to T. The certificate is the list of members in this subset.

Linear programming: Given a list of m linear inequalities with rational coefficients over n vari-
ables $u_1, \ldots, u_n$ (a linear inequality has the form $a_1u_1 + a_2u_2 + \ldots + a_nu_n \leq b$ for some
coefficients $a_1, \ldots, a_n, b$), decide if there is an assignment of rational numbers to the variables
$u_1, \ldots, u_n$ that satisfies all the inequalities. The certificate is the assignment.

Integer programming: Given a list of m linear inequalities with rational coefficients over n
variables $u_1, \ldots, u_m$, find out if there is an assignment of *integer* numbers to $u_1, \ldots, u_n$
satisfying the inequalities. The certificate is the assignment.

---

[1]Some texts use the term *witness* instead of certificate.

Graph isomorphism: Given two $n \times n$ adjacency matrices M_1, M_2, decide if M_1 and M_2 define the same graph, up to renaming of vertices. The certificate is the permutation $\pi : [n] \to [n]$ such that M_2 is equal to M_1 after reordering M_1's indices according to π.

Composite numbers: Given a number N decide if N is a composite (i.e., non-prime) number. The certificate is the factorization of N.

Factoring: Given three numbers N, L, U decide if N has a factor M in the interval $[L, U]$. The certificate is the factor M.

Connectivity: Given a graph G and two vertices s, t in G, decide if s is connected to t in G. The certificate is the path from s to t.

2.1.1 Relation between NP and P

We have the following trivial relationships between **NP** and the classes **P** and **DTIME**$(T(n))$ defined in the previous chapter:

CLAIM 2.3
$\mathbf{P} \subseteq \mathbf{NP} \subseteq \bigcup_{c>1} \mathbf{DTIME}(2^{n^c})$.

PROOF: ($\mathbf{P} \subseteq \mathbf{NP}$): Suppose $L \in \mathbf{P}$ is decided in polynomial-time by a TM N. Then $L \in \mathbf{NP}$ since we can take N as the machine M in Definition 2.1 and make $p(x)$ the zero polynomial (in other words, u is an empty string).

($\mathbf{NP} \subseteq \bigcup_{c>1} \mathbf{DTIME}(2^{n^c})$): If $L \in \mathbf{NP}$ and $M, p()$ are as in Definition 2.1 then we can decide L in time $2^{O(p(n))}$ by enumerating all possible u and using M to check whether u is a valid certificate for the input x. The machine accepts iff such a u is ever found. Since $p(n) = O(n^c)$ for some $c > 1$ then this machine runs in $2^{O(n^c)}$ time. Thus the theorem is proven. ∎

At the moment, we do not know of any stronger relation between **NP** and deterministic time classes than the trivial ones stated in Claim 2.3. The question whether or not $\mathbf{P} = \mathbf{NP}$ is considered *the* central open question of complexity theory, and is also an important question in mathematics and science at large (see Section 2.7). Most researchers believe that $\mathbf{P} \neq \mathbf{NP}$ since years of effort has failed to yield efficient algorithms for certain **NP** languages.

EXAMPLE 2.4
Here is the current knowledge regarding the **NP** decision problems mentioned in Example 2.2:
The **Connectivity**, **Composite Numbers** and **Linear programming** problems are known to be in **P**. For connectivity this follows from the simple and well known breadth-first search algorithm (see [KT06, CLRS01]). The composite numbers problem was only recently shown to be in **P** by Agrawal, Kayal and Saxena [?], who gave a beautiful algorithm to solve it. For the linear programming problem this is again highly non-trivial, and follows from the Ellipsoid algorithm of Khachiyan [?] (there are also faster algorithms, following Karmarkar's interior point paradigm [?]).

All the rest of the problems are not known to have a polynomial-time algorithm, although we have no proof that they are not in **P**. The **Independent Set**, **Traveling Salesperson**, **Subset Sum**, and **Integer Programming** problems are known to be **NP**-*complete*, which, as we will see in this chapter, implies that they are not in **P** unless **P** = **NP**. The **Graph Isomorphism** and **Factoring** problems are not known to be either in **P** nor **NP**-complete.

2.1.2   Non-deterministic Turing machines.

The class **NP** can also be defined using a variant of Turing machines called *non-deterministic* Turing machines (abbreviated NDTM). In fact, this was the original definition and the reason for the name **NP**, which stands for *non-deterministic polynomial-time*. The only difference between an NDTM and a standard TM is that an NDTM has *two* transition functions δ_0 and δ_1. In addition the NDTM has a special state we denote by q_{accept}. When an NDTM M computes a function, we envision that at each computational step M makes an arbitrary choice as to which of its two transition functions to apply. We say that M outputs 1 on a given input x if there is *some* sequence of these choices (which we call the *non-deterministic choices* of M) that would make M reach q_{accept} on input x. Otherwise— if *every* sequence of choices makes M halt without reaching q_{accept}— then we say that $M(x) = 0$. We say that M runs in $T(n)$ time if for every input $x \in \{0,1\}^*$ and every sequence of non-deterministic choices, M reaches either the halting state or q_{accept} within $T(|x|)$ steps.

DEFINITION 2.5
For every function $T : \mathbb{N} \to \mathbb{N}$ and $L \subseteq \{0,1\}^*$, we say that $L \in \mathbf{NTIME}(T(n))$ if there is a constant $c > 0$ and a $cT(n)$-time NDTM M such that for every $x \in \{0,1\}^*$, $x \in L \Leftrightarrow M(x) = 1$

The next theorem gives an alternative definition of **NP**, the one that appears in most texts.

THEOREM 2.6
$\mathbf{NP} = \cup_{c \in \mathbb{N}} \mathbf{NTIME}(n^c)$

PROOF: The main idea is that the sequence of nondeterministic choices made by an accepting computation of an NDTM can be viewed as a certificate that the input is in the language, and vice versa.

Suppose $p : \mathbb{N} \to \mathbb{N}$ is a polynomial and L is decided by a NDTM N that runs in time $p(n)$. For every $x \in L$, there is a sequence of nondeterministic choices that makes N reach q_{accept} on input x. We can use this sequence as a *certificate* for x. Notice, this certificate has length $p(|x|)$ and can be verified in polynomial time by a *deterministic* machine, which checks that N would have entered q_{accept} after using these nondeterministic choices. Thus $L \in \mathbf{NP}$ according to Definition 2.1.

Conversely, if $L \in \mathbf{NP}$ according to Definition 2.1, then we describe a polynomial-time NDTM N that decides L. On input x, it uses the ability to make non-deterministic choices to write down a string u of length $p(|x|)$. (Concretely, this can be done by having transition δ_0 correspond to writing a 0 on the tape and transition δ_1 correspond to writing a 1.) Then it runs the deterministic

verifier M of Definition 2.1 to verify that u is a valid certificate for x, and if so, enters q_{accept}. Clearly, N enters q_{accept} on x if and only if a valid certificate exists for x. Since $p(n) = O(n^c)$ for some $c > 1$, we conclude that $L \in \mathbf{NTIME}(n^c)$. ∎

As is the case with deterministic TM's, NDTM's can be easily represented as strings and there exists a *universal* non-deterministic Turing machine, see Exercise 1. (In fact, using non-determinism we can even make the simulation by a universal TM slightly more efficient.)

2.2 Reducibility and NP-completeness

It turns out that the independent set problem is at least as hard as any other language in **NP**: if it has a polynomial-time algorithm then so do all the problems in **NP**. This fascinating property is called **NP**-*hardness*. Since most scientists conjecture that $\mathbf{NP} \neq \mathbf{P}$, the fact that a language is **NP**-hard can be viewed as evidence that it cannot be decided in polynomial time.

How can we prove that a language B is at least as hard as some other language A? The crucial tool we use is the notion of a *reduction* (see Figure 2.1):

DEFINITION 2.7 (REDUCTIONS, **NP**-HARDNESS AND **NP**-COMPLETENESS)
We say that a language $A \subseteq \{0,1\}^*$ is *polynomial-time Karp reducible to a language $B \subseteq \{0,1\}^*$* (sometimes shortened to just "polynomial-time reducible"[2]) denoted by $A \leq_p B$ if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$, $x \in A$ if and only if $f(x) \in B$.
We say that B is **NP**-*hard* if $A \leq_p B$ for every $A \in \mathbf{NP}$. We say that B is **NP**-*complete* if B is **NP**-hard and $B \in \mathbf{NP}$.
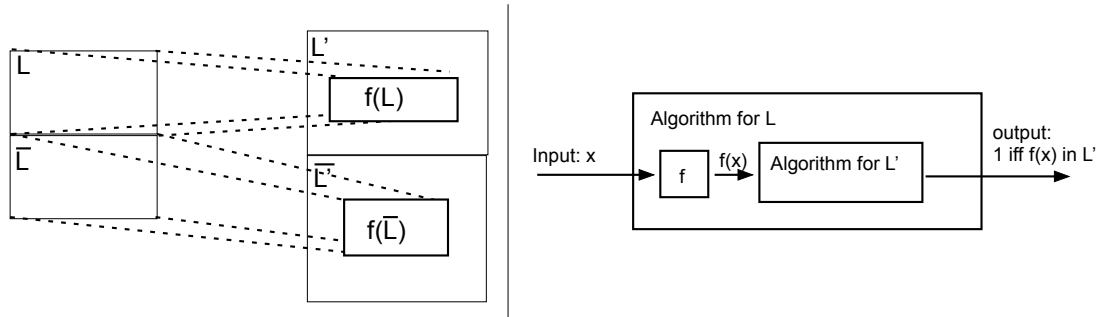
Figure 2.1: A Karp reduction from L to L' is a polynomial-time function f that maps strings in L to strings in L' and strings in $\overline{L} = \{0,1\}^* \setminus L$ to strings in $\overline{L'}$. It can be used to transform a polynomial-time TM M' that decides L' into a polynomial-time TM M for L by setting $M(x) = M'(f(x))$.

Now we observe some properties of polynomial-time reductions. Part 1 of the following Theorem shows that this relation is *transitive*. (Later we will define other notions of reduction, and all will

---

[2]Some texts call this notion "many-to-one reducibility" or "polynomial-time mapping reducibility".

satisfy transitivity.) Part 2 suggests the reason for the term **NP**-hard —namely, an **NP**-hard languages is *at least as hard* as any other **NP** language. Part 3 similarly suggests the reason for the term **NP**-complete: to study the **P** versus **NP** question it suffices to study whether any **NP**-complete problem can be decided in polynomial time.

THEOREM 2.8
1. *(Transitivity) If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.*

2. *If language A is **NP**-hard and $A \in \mathbf{P}$ then $\mathbf{P} = \mathbf{NP}$.*

3. *If language A is **NP**-complete then $A \in \mathbf{P}$ if and only if $\mathbf{P} = \mathbf{NP}$.*

PROOF: The main observation is that if p, q are two functions that have polynomial growth then their composition $p(q(n))$ also has polynomial growth. We prove part 1 and leave the others as simple exercises.

If f_1 is a polynomial-time reduction from A to B and f_2 is a reduction from B to C then the mapping $x \mapsto f_2(f_1(x))$ is a polynomial-time reduction from A to C since $f_2(f_1(x))$ takes polynomial time to compute given x and $f_2(f_1(x)) \in C$ iff $x \in A$. ∎

Do **NP**-complete languages exist? It may not be clear that **NP** should possess a language that is as hard as any other language in the class. However, this does turn out to be the case:

THEOREM 2.9
*The following language is **NP**-complete:*

$$\mathsf{TMSAT} = \{\langle \alpha, x, 1^n, 1^t \rangle : \exists u \in \{0,1\}^n \text{ s.t. } M_\alpha \text{ outputs } 1 \text{ on input } \langle x, u \rangle \text{ within } t \text{ steps}\}$$

where M_α denotes the TM represented by the string α.[3]

Once you internalize the definition of **NP**, the proof of Theorem 2.9 is straightforward and so is left to the reader as Exercise 2. But TMSAT is not a very useful **NP**-complete problem since its definition is intimately tied to the notion of the Turing machine, and hence the fact that it is **NP**-complete does not provide much new insight.

2.3   The Cook-Levin Theorem: Computation is Local

Around 1971, Cook and Levin independently discovered the notion of **NP**-completeness and gave examples of combinatorial **NP**-complete problems whose definition seems to have nothing to do with Turing machines. Soon after, Karp showed that **NP**-completeness occurs widely and many problems of practical interest are **NP**-complete. To date, thousands of computational problems in a variety of disciplines have been found to be **NP**-complete.

---

[3]Recall that 1^k denotes the string consisting of k 1's. it is a common convention in complexity theory to provide a polynomial TM with such an input to allow it to run in time polynomial in k.

2.3.1 Boolean formulae and the CNF form.

Some of the simplest examples of **NP**-complete problems come from propositional logic. A *Boolean formula* over the variables $u_1, \ldots, u_n$ consists of the variables and the logical operators AND ($\wedge$), NOT ($\neg$) and OR ($\vee$); see Appendix A for their definitions. For example, $(a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$ is a Boolean formula that is TRUE if and only if the majority of the variables a, b, c are TRUE. If φ is a Boolean formula over variables $u_1, \ldots, u_n$, and $z \in \{0, 1\}^n$, then $\varphi(z)$ denotes the value of φ when the variables of φ are assigned the values z (where we identify 1 with TRUE and 0 with FALSE). A formula φ is *satisfiable* if there there exists some assignment z such that $\varphi(z)$ is TRUE. Otherwise, we say that φ is *unsatisfiable*.

A Boolean formula over variables $u_1, \ldots, u_n$ is in *CNF form* (shorthand for *Conjunctive Normal Form*) if it is an AND of OR's of variables or their negations. For example, the following is a 3CNF formula:

$$(u_1 \vee \bar{u}_2 \vee u_3) \wedge (u_2 \vee \bar{u}_3 \vee u_4) \wedge (\bar{u}_1 \vee u_3 \vee \bar{u}_4).$$

where $\bar{u}$ denotes the negation of the variable u.

More generally, a CNF formula has the form

$$\bigwedge_i \left(\bigvee_j v_{i_j} \right),$$

where each v_{i_j} is either a variable u_k or to its negation $\neg u_k$. The terms v_{i_j} are called the *literals* of the formula and the terms $(\vee_j v_{i_j})$ are called its *clauses*. A kCNF is a CNF formula in which all clauses contain at most k literals.

2.3.2 The Cook-Levin Theorem

The following theorem provides us with our first natural **NP**-complete problems:

THEOREM 2.10 (COOK-LEVIN THEOREM [Coo71, Lev73])
Let SAT *be the language of all satisfiable CNF formulae and* 3SAT *be the language of all satisfiable 3CNF formulae. Then,*

1. SAT *is* **NP**-*complete.*

2. 3SAT *is* **NP**-*complete.*

REMARK 2.11
An alternative proof of the Cook-Levin theorem, using the notion of *Boolean circuits*, is described in Section 6.7.

Both SAT and 3SAT are clearly in **NP**, since a satisfying assignment can serve as the certificate that a formula is satisfiable. Thus we only need to prove that they are **NP**-hard. We do so by first

proving that SAT is **NP**-hard and then showing that SAT is polynomial-time Karp reducible to 3SAT. This implies that 3SAT is **NP**-hard by the transitivity of polynomial-time reductions. Thus the following lemma is the key to the proof.

LEMMA 2.12
SAT *is* **NP**-*hard.*

Notice, to prove this we have to show how to reduce *every* **NP** language L to SAT, in other words give a polynomial-time transformation that turns any $x \in \{0,1\}^*$ into a CNF formula φ_x such that $x \in L$ iff φ_x is satisfiable. Since we know nothing about the language L except that it is in **NP**, this reduction has to rely just upon the definition of computation, and express it in some way using a Boolean formula.

2.3.3 Warmup: Expressiveness of boolean formulae

As a warmup for the proof of Lemma 2.12 we show how to express various conditions using CNF formulae.

EXAMPLE 2.13
The formula $(a \vee \bar{b}) \wedge (\bar{a} \vee b)$ is in CNF form. It is satisfied by only those values of a, b that are equal. Thus, the formula

$$(x_1 \vee \overline{y}_1) \wedge (\overline{x}_1 \vee y_1) \wedge \cdots \wedge (x_n \vee \overline{y}_n) \wedge (\overline{x}_n \vee y_n)$$

is TRUE if and only if the strings $x, y \in \{0,1\}^n$ are equal to one another.

Thus, though $=$ is not a standard boolean operator like $\vee$ or $\wedge$, we will use it as a convenient shorthand since the formula $\phi_1 = \phi_2$ is equivalent to (in other words, has the same satisfying assignments as) $(\phi_1 \vee \overline{\phi_2}) \wedge (\overline{\phi_1} \vee \phi_2)$.

In fact, CNF formulae of sufficient size can express *every* Boolean condition, as shown by the following simple claim: (this fact is sometimes known as *universality* of the operations AND, OR and NOT)

CLAIM 2.14
For every Boolean function $f : \{0,1\}^{\ell} \to \{0,1\}$ *there is an ℓ-variable CNF formula φ of size $\ell 2^{\ell}$ such that $\varphi(u) = f(u)$ for every $u \in \{0,1\}^{\ell}$, where the size of a CNF formula is defined to be the number of $\wedge/\vee$ symbols it contains.*

PROOF SKETCH: For every $v \in \{0,1\}^{\ell}$, it is not hard to see that there exists a clause C_v such that $C_v(v) = 0$ and $C_v(u) = 1$ for every $u \neq v$. For example, if $v = \langle 1, 1, 0, 1 \rangle$, the corresponding clause is $\overline{u}_1 \vee \overline{u}_2 \vee u_3 \vee \overline{u}_4$.

We let φ be the AND of all the clauses C_v for v such that $f(v) = 0$ (note that φ is indeed of size at most $\ell 2^{\ell}$). Then for every u such that $f(u) = 0$ it holds that $C_u(u) = 0$ and hence $\varphi(u)$ is also equal to 0. On the other hand, if $f(u) = 1$ then $C_v(u) = 1$ for every v such that $f(v) = 0$ and hence $\varphi(u) = 1$. We get that for every u, $\varphi(u) = f(u)$. ∎

In this chapter we will use Claim 2.14 only when the number of variables is some fixed constant.

2.3.4   Proof of Lemma 2.12

Let L be an **NP** language and let M be the polynomial time TM such that that for every $x \in \{0,1\}^*$, $x \in L \Leftrightarrow M(x,u) = 1$ for some $u \in \{0,1\}^{p(|x|)}$, where $p : \mathbb{N} \to \mathbb{N}$ is some polynomial. We show L is polynomial-time Karp reducible to SAT by describing a way to transform in polynomial-time every string $x \in \{0,1\}^*$ into a CNF formula φ_x such that $x \in L$ iff φ_x is satisfiable.

How can we construct such a formula φ_x? By Claim 2.14, the function that maps $u \in \{0,1\}^{p(|x|)}$ to $M(x,u)$ can be expressed as a CNF formula ψ_x (i.e., $\psi_x(u) = M(x,u)$ for every $u \in \{0,1\}^{p(|x|)}$). Thus a string u such that $M(x,u) = 1$ exists if and only if ψ_x is satisfiable. But this is not useful for us, since the size of the formula ψ_x obtained from Claim 2.14 can be as large as $p(|x|)2^{p(|x|)}$. To get a smaller formula we use the fact that M runs in polynomial time, and that each basic step of a Turing machine is highly *local* (in the sense that it examines and changes only a few bits of the machine's tapes).

In the course of the proof we will make the following simplifying assumptions about the TM M: **(1)** M only has two tapes: an input tape and a work/output tape and **(2)** M is an *oblivious* TM in the sense that its head movement does not depend on the contents of its input tape. In particular, this means that M's computation takes the same time for all inputs of size n and for each time step i the location of M's heads at the ith step depends only on i and M's input length.

We can make these assumptions without loss of generality because for every $T(n)$-time TM M there exists a two-tape oblivious TM $\tilde{M}$ computing the same function in $O(T(n)^2)$ time (see Remark 1.10 and Exercise 8 of Chapter 1).[4] Thus in particular, if L is in **NP** then there exists a two-tape oblivious polynomial-time TM M and a polynomial p such that $x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)}$ s.t. $M(x,u) = 1$.

The advantage of assuming that M is oblivious is that for any given input length, we can define functions $\mathtt{inputpos}(i), \mathtt{prev}(i)$ where $\mathtt{inputpos}(i)$ denotes the location of the input tape head at the ith step and $\mathtt{prev}(i)$ denotes the last step before i that M visited the same location on its work tape, see Figure 2.3.[5] These values can be computed in polynomial time by simulating M on, say, the all-zeroes input.

Denote by Q the set of M's possible states and by Γ its alphabet. The *snapshot* of M's execution on some input y at a particular step i is the triple $\langle a, b, q \rangle \in \Gamma \times \Gamma \times Q$ such that a, b are the symbols read by M's heads from the two tapes and q is the state M is in at the ith step (see Figure 2.2). For every $m \in \mathbb{N}$ and $y \in \{0,1\}^m$, the snapshot of M's execution on input y at the ith step depends on **(1)** its state in the $i - 1^{st}$ step and **(2)** the contents of the current cells of its input and work tapes. We write this relation as

$$z_i = F\left(z_{i-1}, z_{\mathtt{prev}(i)}, y_{\mathtt{inputpos}(i)}\right),$$

where $\mathtt{inputpos}(i)$ and $\mathtt{prev}(i)$ are as defined earlier, z_i is the encoding of the ith snapshot as a binary string of some length c, and F is some function (derived from M's transition function) that

---

[4]In fact, with some more effort we even simulate a non-oblivious $T(n)$-time TM by an oblivious TM running in $O(T(n) \log T(n))$-time, see Exercise 9 of Chapter 1. This oblivious machine may have more than two tapes, but the proof below easily generalizes to this case.

[5]If i is the first step that M visits a certain location, then we define $\mathtt{prev}(i) = 1$.
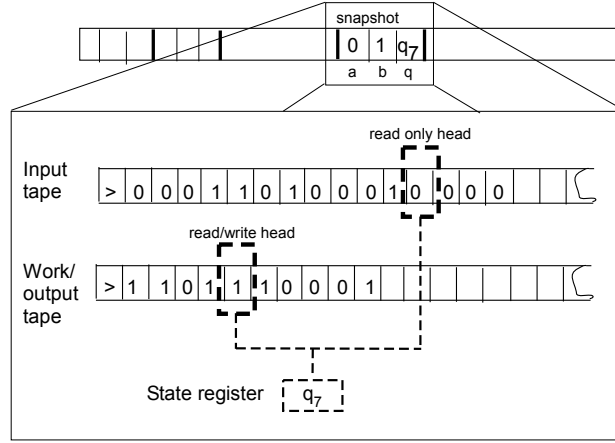
Figure 2.2: A snapshot of a TM contains the current state and symbols read by the TM at a particular step. If at the ith step M reads the symbols $0, 1$ from its tapes and is in the state q_7 then the snapshot of M at the ith step is $\langle 0, 1, q_7 \rangle$.

maps $\{0, 1\}^{2c+1}$ to $\{0, 1\}^c$. (Note that c is a constant depending only on M's state and alphabet size, and independent of the input size.)
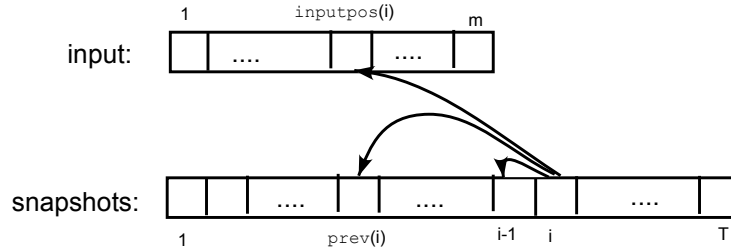
Figure 2.3: The snapshot of M at the ith step depends on its previous state (contained in the snapshot at the $i - 1^{st}$ step), and the symbols read from the input tape, which is in position $\texttt{inputpos}(i)$, and from the work tape, which was last written to in step $\texttt{prev}(i)$.

Let $n \in \mathbb{N}$ and $x \in \{0, 1\}^n$. We need to construct a CNF formula φ_x such that $x \in L \Leftrightarrow \varphi_x \in \mathsf{SAT}$. Recall that $x \in L$ if and only if there exists some $u \in \{0, 1\}^{p(n)}$ such that $M(y) = 1$ where $y = x \circ u$ (with $\circ$ denoting concatenation). Since the sequence of snapshots in M's execution completely determines its outcome, this happens if and only if there exists a string $y \in \{0, 1\}^{n+p(n)}$ and a sequence of strings $z_1, \ldots, z_T \in \{0, 1\}^c$ (where $T = T(n)$ is the number of steps M takes on inputs of length $n + p(n)$) satisfying the following four conditions:

1. The first n bits of y are equal to x.

2. The string z_1 encodes the initial snapshot of M (i.e., the triple $\langle \triangleright, \square, q_{\mathsf{start}} \rangle$ where $\triangleright$ is the start symbol of the input tape, $\square$ is the blank symbol, and q_{start} is the initial state of the TM

M).

3. For every $i \in \{2, .., T\}$, $z_i = F(z_{i-1}, z_{\texttt{inputpos}(i)}, z_{\texttt{prev}(i)})$.

4. The last string z_T encodes a snapshot in which the machine halts and outputs 1.

The formula φ_x will take variables $y \in \{0,1\}^{n+p(n)}$ and $z \in \{0,1\}^{cT}$ and will verify that y, z satisfy the AND of these four conditions. Clearly $x \in L \Leftrightarrow \varphi_x \in \mathsf{SAT}$ and so all that remains is to show that we can express φ_x as a polynomial-sized CNF formula.

Condition 1 can be expressed as a CNF formula of size $4n$ (see Example 2.13). Conditions 2 and 4 each depend on c variables and hence by Claim 2.14 can be expressed by CNF formulae of size $c2^c$. Condition 3, which is an AND of T conditions each depending on at most $3c + 1$ variables, can be expressed as a CNF formula of size at most $T(3c + 1)2^{3c+1}$. Hence the AND of all these conditions can be expressed as a CNF formula of size $d(n + T)$ where d is some constant depending only on M. Moreover, this CNF formula can be computed in time polynomial in the running time of M. ∎

2.3.5 Reducing SAT to 3SAT.

Since both SAT and $\mathsf{3SAT}$ are clearly in **NP**, Lemma 2.12 completes the proof that SAT is **NP**-complete. Thus all that is left to prove Theorem 2.10 is the following lemma:

LEMMA 2.15
$\mathsf{SAT} \leq_p \mathsf{3SAT}$.

PROOF: We will map a CNF formula φ into a 3CNF formula ψ such that ψ is satisfiable if and only if φ is. We demonstrate first the case that φ is a 4CNF. Let C be a clause of φ, say $C = u_1 \vee \overline{u}_2 \vee \overline{u}_3 \vee u_4$. We add a new variable z to the φ and replace C with the pair of clauses $C_1 = u_1 \vee \overline{u}_2 \vee z$ and $C_2 = \overline{u}_3 \vee u_4 \vee \overline{z}$. Clearly, if $u_1 \vee \overline{u}_2 \vee \overline{u}_3 \vee u_4$ is true then there is an assignment to z that satisfies both $u_1 \vee \overline{u}_2 \vee z$ and $\overline{u}_3 \vee u_4 \vee \overline{z}$ and vice versa: if C is false then no matter what value we assign to z either C_1 or C_2 will be false. The same idea can be applied to a general clause of size 4, and in fact can be used to change every clause C of size k (for $k > 3$) into an equivalent pair of clauses C_1 of size $k - 1$ and C_2 of size 3 that depend on the k variables of C and an additional auxiliary variable z. Applying this transformation repeatedly yields a polynomial-time transformation of a CNF formula φ into an equivalent 3CNF formula ψ. ∎

2.3.6 More thoughts on the Cook-Levin theorem

The Cook-Levin theorem is a good example of the power of abstraction. Even though the theorem holds regardless of whether our computational model is the C programming language or the Turing machine, it may have been considerably more difficult to discover in the former context.

Also, it is worth pointing out that the proof actually yields a result that is a bit stronger than the theorem's statement:

1. If we use the efficient simulation of a standard TM by an oblivious TM (see Exercise 9, Chapter 1) then for every $x \in \{0,1\}^*$, the size of the formula φ_x (and the time to compute it) is $O(T \log T)$, where T is the number of steps the machine M takes on input x (see Exercise 10).

2. The reduction f from an **NP**-language L to SAT presented in Lemma 2.12 not only satisfied that $x \in L \Leftrightarrow f(x) \in$ SAT but actually the proof yields an efficient way to transform a certificate for x to a satisfying assignment for $f(x)$ and vice versa. We call a reduction with this property a *Levin* reduction. One can also verify that the proof supplied a one-to-one and onto map between the set of certificates for x and the set of satisfying assignments for $f(x)$, implying that they are of the same size. A reduction with this property is called *parsimonious*. Most of the known **NP**-complete problems (including all the ones mentioned in this chapter) have parsimonious Levin reductions from all the **NP** languages (see Exercise 11). As we will see in this book, this fact is sometimes useful for certain applications.

Why 3SAT? The reader may wonder why is the fact that 3SAT is **NP**-complete so much more interesting than the fact that, say, the language TMSAT of Theorem 2.9 is **NP**-complete. One answer is that 3SAT is useful for proving the **NP**-completeness of other problems: it has very minimal combinatorial structure and thus easy to use in reductions. Another answer has to do with history: propositional logic has had a central role in mathematical logic —in fact it was exclusively the language of classical logic (e.g. in ancient Greece). This historical resonance is one reason why Cook and Levin were interested in 3SAT in the first place. A third answer has to do with practical importance: it is a simple example of *constraint satisfaction problems*, which are ubiquitous in many fields including artificial intelligence.

2.4 The web of reductions

Cook and Levin had to show how *every* **NP** language can be reduced to SAT. To prove the **NP**-completeness of any other language L, we do not need to work as hard: it suffices to reduce SAT or 3SAT to L. Once we know that L is **NP**-complete we can show that an **NP**-language L' is in fact **NP**-complete by reducing L to L'. This approach has been used to build a "web of reductions" and show that thousands of interesting languages are in fact **NP**-complete. We now show the **NP**-completeness of a few problems. More examples appear in the exercises (see Figure 2.4). See the classic book by Garey and Johnson [GJ79] and the Internet site [**?**] for more.

THEOREM 2.16 (INDEPENDENT SET IS **NP**-COMPLETE)
Let INDSET $= \{\langle G, k \rangle : G$ *has independent set of size* $k\}$. *Then* INDSET *is* **NP**-*complete.*

PROOF: Since INDSET is clearly in **NP**, we only need to show that it is **NP**-hard, which we do by reducing 3SAT to INDSET. Let φ be a 3CNF formula on n variables with m clauses. We define a graph G of $7m$ vertices as follows: we associate a cluster of 7 vertices in G with each clause of φ. The vertices in cluster associated with a clause C correspond to the 7 possible partial assignments to the three variables C depends on (we call these *partial assignments*, since they only give values for some of the variables). For example, if C is $\overline{u_2} \vee \overline{u_5} \vee \overline{u_7}$ then the 7 vertices in the cluster associated

$\forall L \in \mathbf{NP}$

Theorem 2.10 (Lemma 2.12)

SAT

Theorem 2.10 (Lemma 2.15)

Theorem 2.18

Theorem 2.17
INTEGERPROG

dHAMPATH

Ex 17

Theorem 2.16

3SAT
Ex 16
Exactone3SAT

HAMPATH

INDSET

Ex 16

TSP

Ex 17

HAMCYCLE

Ex 14

SUBSETSUM

Ex 6

THEOREMS

CLIQUE

VERTEXCOVER

Ex 15

Ex 18

MAXCUT

QUADEQ
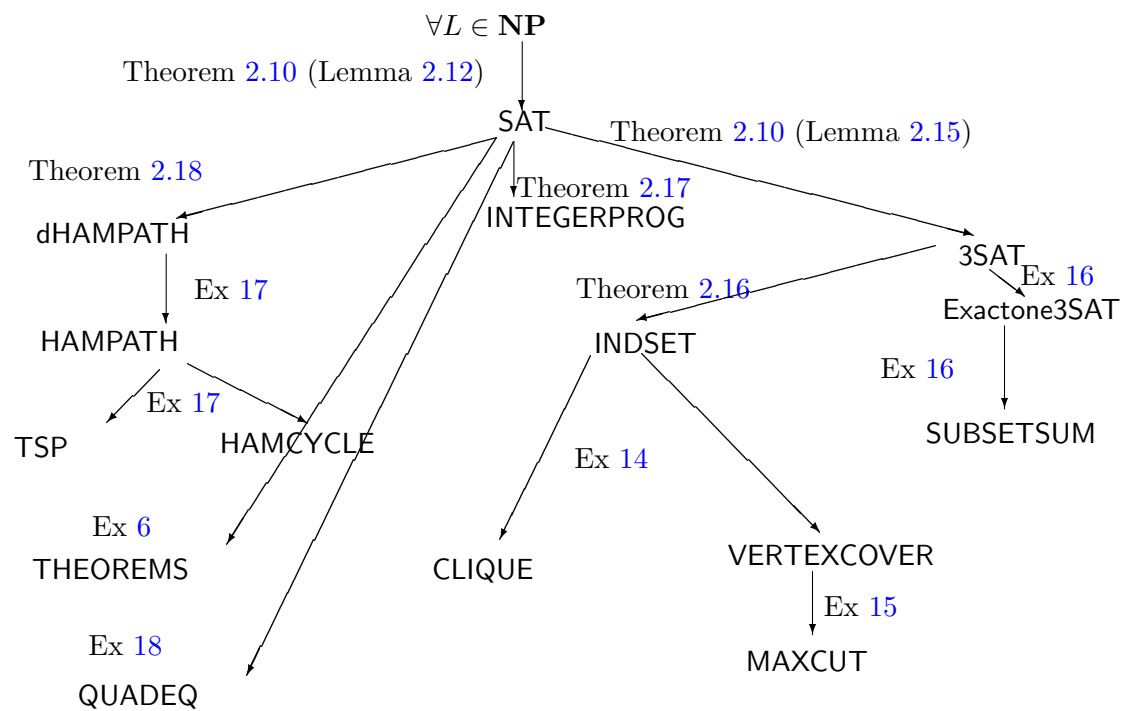
Figure 2.4: Web of reductions between the **NP**-completeness problems described in this chapter and the exercises. Thousands more are known.

with C correspond to all partial assignments of the form $u_1 = a, u_2 = b, u_3 = c$ for a binary vector $\langle a, b, c \rangle \neq \langle 1, 1, 1 \rangle$. (If C depends on less than three variables then we repeat one of the partial assignments and so some of the 7 vertices will correspond to the same assignment.) We put an edge between two vertices of G if they correspond to *inconsistent* partial assignments. Two partial assignments are consistent if they give the same value to all the variables they share. For example, the assignment $u_1 = 1, u_2 = 0, u_3 = 0$ is inconsistent with the assignment $u_3 = 1, u_5 = 0, u_7 = 1$ because they share a variable (u_3) to which they give a different value. In addition, we put edges between every two vertices that are in the same cluster.

Clearly transforming φ into G can be done in polynomial time. We claim that φ is satisfiable if and only if G has a clique of size m. Indeed, suppose that φ has a satisfying assignment u. Define a set S of m vertices as follows: for every clause C of φ put in S the vertex in the cluster associated with C that corresponds to the restriction of u to the variables C depends on. Because we only choose vertices that correspond to restrictions of the assignment u, no two vertices of S correspond to inconsistent assignments and hence S is an independent set of size m.

On the other hand, suppose that G has an independent set S of size m. We will use S to construct a satisfying assignment u for φ. We define u as follows: for every $i \in [n]$, if there is a vertex in S whose partial assignment gives a value a to u_i, then set $u_i = a$; otherwise set $u_i = 0$. This is well defined because S is an independent set, and hence each variable u_i can get at most a single value by assignments corresponding to vertices in S. On the other hand, because we put all the edges within each cluster, S can contain at most a single vertex in each cluster, and hence there is an element of S in every one of the m clusters. Thus, by our definition of u, it satisfies all of φ's clauses. ∎

We see that, surprisingly, the answer to the famous **NP** vs. **P** question depends on the seemingly mundane question of whether one can efficiently plan an optimal dinner party. Here are some more **NP**-completeness results:

THEOREM 2.17 (INTEGER PROGRAMMING IS **NP**-COMPLETE)
We say that a set of linear inequalities with rational coefficients over variables $u_1, \ldots, u_n$ is in IPROG *if there is an assignment of integer numbers in $\{0, 1, 2, \ldots\}$ to $u_1, \ldots, u_n$ that satisfies it. Then,* IPROG *is* **NP**-*complete.*

PROOF: IPROG is clearly in **NP**. To reduce SAT to IPROG note that every CNF formula can be easily expressed as an integer program: first add the constraints $0 \leq u_i \leq 1$ for every i to ensure that the only feasible assignments to the variables are 0 or 1, then express every clause as an inequality. For example, the clause $u_1 \vee \overline{u}_2 \vee \overline{u}_3$ can be expressed as $u_1 + (1 - u_2) + (1 - u_3) \geq 1$. ∎

THEOREM 2.18 (HAMILTONIAN PATH IS **NP**-COMPLETE)
Let dHAMPATH *denote the set of all directed graphs that contain a path visiting all of their vertices exactly once. Then* dHAMPATH *is* **NP**-*complete.*

PROOF: Again, dHAMPATH is clearly in **NP**. To show it's **NP**-complete we show a way to map every CNF formula φ into a graph G such that φ is satisfiable if and only if G has a Hamiltonian path (i.e. path that visits all of G's vertices exactly once).
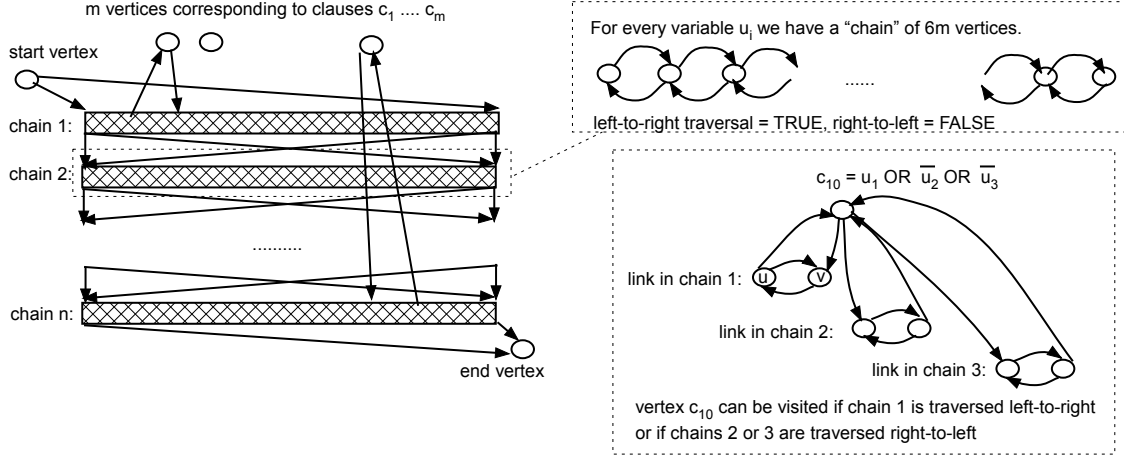
Figure 2.5: Reducing SAT to dHAMPATH. A formula φ with n variables and m clauses is mapped to a graph G that has m vertices corresponding to the clauses and n doubly linked chains, each of length $6m$, corresponding to the variables. Traversing a chain left to right corresponds to setting the variable to TRUE, while traversing it right to left corresponds to setting it to FALSE. Note that in the figure every Hamiltonian path that takes the edge from u to c_{10} must immediately take the edge from c_{10} to v, as otherwise it would get "stuck" the next time it visits v.

The reduction is described in Figure 2.5. The graph G has **(1)** m vertices for each of φ's clauses $c_1, \ldots, c_m$, **(2)** a special starting vertex v_{start} and ending vertex v_{end} and **(3)** n "chains" of $6m$ vertices corresponding to the n variables of φ. A chain is a set of vertices $v_1, \ldots, v_{6m}$ such that for every $i \in [6m-1]$, v_i and v_{i+1} are connected by two edges in both directions.

We put edges from the starting vertex v_{start} to the two extreme points of the first chain. We also put edges from the extreme points of the jth chain to the extreme points to the $j+1^{th}$ chain for every $j \in [n-1]$. We put an edge from the extreme points of the nth chain to the ending vertex v_{end}.

In addition to these edges, for every clause C of φ, we put edges between the chains corresponding to the variables appearing in C and the vertex v_C corresponding to C in the following way: if C contains the literal u_j then we take two neighboring vertices v_i, v_{i+1} in the jth chain and put an edge from v_i to C and from C to v_{i+1}. If C contains the literal $\overline{u}_j$ then we connect these edges in the opposite direction (i.e., v_{i+1} to C and C to v_i). When adding these edges, we never "reuse" a link v_i, v_{i+1} in a particular chain and always keep an unused link between every two used links. We can do this since every chain has $6m$ vertices, which is more than sufficient for this.

($\varphi \in$ SAT $\Rightarrow G \in$ dHAMPATH.) Suppose that φ has a satisfying assignment $u_1, \ldots, u_n$. We will show a path that visits all the vertices of G. The path will start at v_{start}, travel through all the chains in order, and end at v_{end}. For starters, consider the path that travels the jth chain in left-to-right order if $u_j = 1$ and travels it in right-to-left order if $u_j = 0$. This path visits all the vertices except for those corresponding to clauses. Yet, if u is a satisfying assignment then the path can be easily modified to visit all the vertices corresponding to clauses: for each clause C there is at least one literal that is true, and we can use one link on the chain corresponding to that literal to "skip" to the vertex v_C and continue on as before.

($G \in$ dHAMPATH $\Rightarrow \varphi \in$ SAT.) Suppose that G has an Hamiltonian path P. We first note that the path P must start in v_{start} (as it has no incoming edges) and end at v_{end} (as it has no outgoing edges). Furthermore, we claim that P needs to traverse all the chains in order, and within each chain traverse it either in left-to-right order or right-to-left order. This would be immediate if the path did not use the edges from a chain to the vertices corresponding to clauses. The claim holds because if a Hamiltonian path takes the edge $u \to w$, where u is on a chain and w corresponds to a clause, then it must at the next step take the edge $w \to v$ where v is the vertex adjacent to u in the link. Otherwise, the path will get stuck the next time it visits v (see Figure 2.1). Now, define an assignment $u_1, \ldots, u_n$ to φ as follows: $u_j = 1$ if P traverses the jth chain in left-to-right order, and $u_j = 0$ otherwise. It is not hard to see that because P visits all the vertices corresponding to clauses, $u_1, \ldots, u_n$ is a satisfying assignment for φ. ∎

In praise of reductions

Though originally invented as part of the theory of **NP**-completeness, the polynomial-time reduction (together with its first cousin, the randomized polynomial-time reduction defined in Section 7.9) has led to a rich understanding of complexity above and beyond **NP**-completeness. Much of complexity theory and cryptography today (thus, many chapters of this book) consists of using reductions to make connections between disparate complexity theoretic conjectures. Why do complexity theorists excel at reductions but not at actually proving lower bounds on Turing machines? A possible explanation is that humans have evolved to excel at problem solving, and hence are more adept at algorithms (after all, a reduction is merely an algorithm to transform one problem into another) than at proving lower bounds on Turing machines.

Coping with NP hardness.

NP-complete problems turn up in great many applications, from flight scheduling to genome sequencing. What do you do if the problem you need to solve turns out to be **NP**-complete? On the outset, the situation looks bleak: if $\mathbf{P} \neq \mathbf{NP}$ then there simply does not *exist* an efficient algorithm to solve such a problem. However, there may still be some hope: **NP** completeness only means that (assuming $\mathbf{P} \neq \mathbf{NP}$) the problem does not have an algorithm that solves it *exactly* on *every* input. But for many applications, an *approximate* solution on *some* of the inputs might be good enough.

    A case in point is the traveling salesperson problem (TSP), of computing, given a list of pairwise distances between n cities, the shortest route that travels through all of them. Assume that you are indeed in charge of coming up with travel plans for traveling salespersons that need to visit various cities around your country. Does the fact that TSP is **NP**-complete means that you are bound to do a hopelessly suboptimal job? This does not have to be the case.

    First note that you do not need an algorithm that solves the problem on *all* possible lists of pairwise distances. We might model the inputs that actually arise in this case as follows: the n cities are points on a plane, and the distance between a pair of cities is the distance between the corresponding points (we are neglecting here the difference between travel distance and direct/arial distance). It is an easy exercise to verify that not all possible lists of pairwise distances can be

generated in such a way. We call those that do *Euclidean* distances. Another observation is that computing the *exactly* optimal travel plan may not be so crucial. If you could always come up with a travel plan that is at most 1% longer than the optimal, this should be good enough.

It turns out that neither of these observations on its own is sufficient to make the problem tractable. The TSP problem is still **NP** complete even for Euclidean distances. Also if $\mathbf{P} \neq \mathbf{NP}$ then TSP is hard to approximate within any constant factor. However, *combining* the two observations together actually helps: for every ϵ there is a $\text{poly}(n(\log n)^{O(1/\epsilon)})$-time algorithm that given Euclidean distances between n cities comes up with a tour that is at most a factor of $(1 + \epsilon)$ worse than the optimal tour [Aro98].

We see that discovering the problem you encounter is **NP**-complete should not be cause for immediate despair. Rather you should view this as indication that a more careful modeling of the problem is needed, letting the literature on complexity and algorithms guide you as to what features might make the problem more tractable. Alternatives to worst-case exact computation are explored in Chapters 15 and 18, that investigate *average-case complexity* and *approximation algorithms* respectively.

2.5 Decision versus search

We have chosen to define the notion of **NP** using Yes/No problems ("Is the given formula satisfiable?") as opposed to *search* problems ("Find a satisfying assignment to this formula if one exists"). Clearly, the search problem is harder than the corresponding decision problem, and so if $\mathbf{P} \neq \mathbf{NP}$ then neither one can be solved for an **NP**-complete problem. However, it turns out that for **NP**-complete problems they are equivalent in the sense that if the decision problem can be solved (and hence $\mathbf{P} = \mathbf{NP}$) then the search version of any **NP** problem can also be solved in polynomial time.

THEOREM 2.19
Suppose that $\mathbf{P} = \mathbf{NP}$. *Then, for every* **NP** *language* L *there exists a polynomial-time TM* B *that on input* $x \in L$ *outputs a certificate for* x.

That is, if, as per Definition 2.1, $x \in L$ iff $\exists u \in \{0,1\}^{p(|x|)}$ s.t. $M(x, u) = 1$ where p is some polynomial and M is a polynomial-time TM, then on input $x \in L$, $B(x)$ will be a string $u \in \{0,1\}^{p(|x|)}$ satisfying $M(x, B(x)) = 1$.

PROOF: We start by showing the theorem for the case of SAT. In particular we show that given an algorithm A that decides SAT, we can come up with an algorithm B that on input a satisfiable CNF formula φ with n variables, finds a satisfying assignment for φ using $2n + 1$ calls to A and some additional polynomial-time computation.

The algorithm B works as follows: we first use A to check that the input formula φ is satisfiable. If so, we substitute $x_1 = 0$ and $x_1 = 1$ in φ (this transformation, that simplifies and shortens the formula a little, leaving a formula with $n - 1$ variables, can certainly be done in polynomial time) and then use A to decide which of the two is satisfiable (it is possible that both are). Say the first is satisfiable. Then we fix $x_1 = 0$ from now on and continue with the simplified formula. Continuing this way we end up fixing all n variables while ensuring that each intermediate formula is satisfiable. Thus the final assignment to the variables satisfies φ.

To solve the search problem for an arbitrary **NP**-language L, we use the fact that the reduction of Theorem 2.10 from L to SAT is actually a *Levin* reduction. This means that we have a polynomial-time computable function f such that not only $x \in L \Leftrightarrow f(x) \in$ SAT but actually we can map a satisfying assignment of $f(x)$ into a certificate for x. Therefore, we can use the algorithm above to come up with an assignment for $f(x)$ and then map it back into a certificate for x. ∎

REMARK 2.20
The proof above shows that SAT is *downward self-reducible*, which means that given an algorithm that solves SAT on inputs of length smaller than n we can solve SAT on inputs of length n. Using the Cook-Levin reduction, one can show that all **NP**-complete problems have a similar property, though we do not make this formal.

2.6 coNP, EXP and NEXP

Now we define some related complexity classes that are very relevant to the study of the **P** versus **NP** question.

2.6.1 coNP

If $L \subseteq \{0,1\}^*$ is a language, then we denote by $\overline{L}$ the *complement* of L. That is, $\overline{L} = \{0,1\}^* \setminus L$. We make the following definition:

DEFINITION 2.21
$\mathbf{coNP} = \{ L : \overline{L} \in \mathbf{P} \}$.

It is important to note that **coNP** is *not* the complement of the class **NP**. In fact, they have a non-empty intersection, since every language in **P** is in $\mathbf{NP} \cap \mathbf{coNP}$ (see Exercise 19). The following is an example of a **coNP** language: $\overline{\mathsf{SAT}} = \{ \varphi : \varphi \text{ is not satisfiable} \}$. Students sometimes mistakenly convince themselves that $\overline{\mathsf{SAT}}$ is in **NP**. They have the following polynomial time NDTM in mind: on input φ, the machine guesses an assignment. If this assignment does not satisfy φ then it accepts (i.e., goes into q_{accept} and halts) and if it does satisfy φ then the machine halts without accepting. This NDTM does not do the job: indeed it accepts every unsatisfiable φ but in addition it also accepts many satisfiable formulae (i.e., every formula that has a single unsatisfying assignment). That is why pedagogically we prefer the following definition of **coNP** (which is easily shown to be equivalent to the first, see Exercise 20):

DEFINITION 2.22 (**coNP**, ALTERNATIVE DEFINITION)
For every $L \subseteq \{0,1\}^*$, we say that $L \in \mathbf{coNP}$ if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time TM M such that for every $x \in \{0,1\}^*$,

$$x \in L \Leftrightarrow \forall u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x,u) = 1$$

The key fact to note is the use of "$\forall$" in this definition where Definition 2.1 used $\exists$.

We can define **coNP**-completeness in analogy to **NP**-completeness: a language is **coNP**-complete if it is in **coNP** and every **coNP** language is polynomial-time Karp reducible to it.

EXAMPLE 2.23

In classical logic, *tautologies* are true statements. The following language is **coNP**-complete:

TAUTOLOGY $= \{\varphi : \varphi$ is a Boolean formula that is satisfied by every assignment$\}$.

It is clearly in **coNP** by Definition 2.22 and so all we have to show is that for every $L \in$ **coNP**, $L \leq_p$ TAUTOLOGY. But this is easy: just modify the Cook-Levin reduction from $\overline{L}$ (which is in **NP**) to SAT. For every input $x \in \{0, 1\}^*$ that reduction produces a formula φ_x that is satisfiable iff $x \in \overline{L}$. Now consider the formula $\neg \varphi_x$. It is in TAUTOLOGY iff $x \in L$, and this completes the description of the reduction.

It is a simple exercise to check that if **P** = **NP** then **NP** = **coNP** = **P**. Put in the contrapositive, if we can show that **NP** $\neq$ **coNP** then we have shown **P** $\neq$ **NP**. Most researchers believe that **NP** $\neq$ **coNP**. The intuition is almost as strong as for the **P** versus **NP** question: it seems hard to believe that there is any short certificate for certifying that a given formula is a TAUTOLOGY, in other words, to certify that *every* assignment satisfies the formula.

2.6.2 EXP and NEXP

The following two classes are exponential time analogues of **P** and **NP**.

DEFINITION 2.24
EXP $= \cup_{c \geq 0}$**DTIME**(2^{n^c}).
NEXP $= \cup_{c \geq 0}$**NTIME**(2^{n^c}).

Because every problem in **NP** can be solved in exponential time by a brute force search for the certificate, **P** $\subseteq$ **NP** $\subseteq$ **EXP** $\subseteq$ **NEXP**. Is there any point to studying classes involving exponential running times? The following simple result —providing merely a glimpse of the rich web of relations we will be establishing between disparate complexity questions— may be a partial answer.

THEOREM 2.25
If **EXP** $\neq$ **NEXP** *then* **P** $\neq$ **NP**.

PROOF: We prove the contrapositive: assuming **P** = **NP** we show **EXP** = **NEXP**. Suppose $L \in$ **NTIME**(2^{n^c}) and NDTM M decides it. We claim that then the language

$$L_{\text{pad}} = \left\{ \langle x, 1^{2^{|x|^c}} \rangle : x \in L \right\} \tag{1}$$

is in **NP**. Here is an NDTM for L_{pad}: given y, first check if there is a string z such that $y = \langle z, 1^{2^{|z|^c}} \rangle$. If not, output REJECT. If y is of this form, then run M on z for $2^{|z|^c}$ steps and output its answer. Clearly, the running time is polynomial in $|y|$, and hence $L_{\text{pad}} \in$ **NP**. Hence if **P** = **NP** then L_{pad} is in **P**. But if L_{pad} is in **P** then L is in **EXP**: to determine whether an input x is in L, we just pad the input and decide whether it is in L_{pad} using the polynomial-time machine for L_{pad}. ∎

REMARK 2.26

The *padding* technique used in this proof, whereby we transform a language by "padding" every string in a language with a string of (useless) symbols, is also used in several other results in complexity theory. In many settings it can be used to show that equalities between complexity classes "scale up"; that is, if two different type of resources solve the same problems within bound $T(n)$ then this also holds for functions T' larger than T. Viewed contrapositively, padding can be used to show that inequalities between complexity classes involving resurce bound $T'(n)$ "scale down" to resource bound $T(n)$.

Like **P** and **NP**, most of the complexity classes studied later are also contained in both **EXP** and **NEXP**.

2.7 More thoughts about P, NP, and all that

2.7.1 The philosophical importance of NP

At a totally abstract level, the **P** versus **NP** question may be viewed as a question about the power of nondeterminism in the Turing machine model. (Similar questions have been completely answered for simpler models such as finite automata.)

However, the certificate definition of **NP** also suggests that the **P** versus **NP** question captures a widespread phenomenon of some philosophical importance (and a source of great frustration to students): recognizing the correctness of an answer is often easier than coming up with the answer. To give other analogies from life: appreciating a Beethoven sonata is far easier than composing the sonata; verifying the solidity of a design for a suspension bridge is easier (to a civil engineer anyway!) than coming up with a good design; verifying the proof of a theorem is easier than coming up with a proof itself (a fact referred to in Gödel's letter mentioned at the start of the chapter), and so forth. In such cases, coming up with the right answer seems to involve *exhaustive search* over an exponentially large set. The **P** versus **NP** question asks whether exhaustive search can be avoided in general. It seems obvious to most people —and the basis of many false proofs proposed by amateurs— that exhaustive search cannot be avoided: checking that a given salesperson tour (provided by somebody else) has length at most k ought to be a lot easier than coming up with such a tour by oneself. Unfortunately, turning this intuition into a proof has proved difficult.

2.7.2 NP and mathematical proofs

By definition, **NP** is the set of languages where membership has a short certificate. This is reminiscent of another familiar notion, that of a mathematical proof. As noticed in the past century, in principle all of mathematics can be axiomatized, so that proofs are merely formal manipulations of axioms. Thus the correctness of a proof is rather easy to verify —just check that each line follows from the previous lines by applying the axioms. In fact, for most known axiomatic systems (e.g., Peano arithmetic or Zermelo-Fraenkel Set Theory) this verification runs in time *polynomial* in the length of the proof. Thus the following problem is in **NP** for any of the usual axiomatic systems $\mathcal{A}$:

$$\text{THEOREMS} = \{(\varphi, 1^n) : \ \varphi \text{ has a formal proof of length} \leq n \text{ in system } \mathcal{A}\}.$$

In fact, the exercises ask you to prove that this problem is **NP**-complete. Hence the **P** versus **NP** question is a *rephrasing* of Gödel's question (see quote at the beginning of the chapter), which asks whether or not there is a algorithm that finds mathematical proofs in time polynomial in the length of the proof.

Of course, all our students know in their guts that finding correct proofs is far harder than verifying their correctness. So presumably, they believe at an intuitive level that **P** $\neq$ **NP**.

2.7.3   What if P = NP?

If **P** = **NP** —specifically, if an **NP**-complete problem like 3SAT had a very efficient algorithm running in say $O(n^2)$ time— then the world would be mostly a Utopia. Mathematicians could be replaced by efficient theorem-discovering programs (a fact pointed out in Kurt Gödel's 1956 letter and discovered three decades later). In general for every search problem whose answer can be efficiently verified (or has a short certificate of correctness), we will be able to find the correct answer or the short certificate in polynomial time. AI software would be perfect since we could easily do exhaustive searches in a large tree of possibilities. Inventors and engineers would be greatly aided by software packages that can design the perfect part or gizmo for the job at hand. VLSI designers will be able to whip up optimum circuits, with minimum power requirements. Whenever a scientist has some experimental data, she would be able to automatically obtain the simplest theory (under any reasonable measure of simplicity we choose) that best explains these measurements; by the principle of Occam's Razor the simplest explanation is likely to be the right one. Of course, in some cases it took scientists centuries to come up with the simplest theories explaining the known data. This approach can be used to solve also non-scientific problems: one could find the simplest theory that explains, say, the list of books from the New York *Times'* bestseller list. (NB: All these applications will be a consequence of our study of the Polynomial Hierarchy in Chapter 5.)

Somewhat intriguingly, this Utopia would have no need for randomness. As we will later see, if **P** = **NP** then randomized algorithms would buy essentially no efficiency gains over deterministic algorithms; see Chapter 7. (Philosophers should ponder this one.)

This Utopia would also come at one price: there would be no privacy in the digital domain. Any encryption scheme would have a trivial decoding algorithm. There would be no digital cash, no SSL, RSA or PGP (see Chapter 10). We would just have to learn to get along better without these, folks.

This utopian world may seem ridiculous, but the fact that we can't rule it out shows how little we know about computation. Taking the half-full cup point of view, it shows how many wonderful things are still waiting to be discovered.

2.7.4   What if NP = coNP?

If **NP** = **coNP**, the consequences still seem dramatic. Mostly, they have to do with existence of short certificates for statements that do not seem to have any. To give an example, remember the **NP**-complete problem of finding whether or not a set of multivariate polynomials has a common

root, in other words, deciding whether a system of equations of the following type has a solution:

$$f_1(x_1, \ldots, x_n) = 0$$
$$f_2(x_1, \ldots, x_n) = 0$$
$$\vdots$$
$$f_m(x_1, \ldots, x_n) = 0$$

where each f_i is a quadratic polynomial.

If a solution exists, then that solution serves as a *certificate* to this effect (of course, we have to also show that the solution can be described using a polynomial number of bits, which we omit). The problem of deciding that the system does *not* have a solution is of course in **coNP**. Can we give a certificate to the effect that the system does *not* have a solution? Hilbert's Nullstellensatz Theorem seems to do that: it says that the system is infeasible iff there is a sequence of polynomials $g_1, g_2, \ldots, g_m$ such that $\sum_i f_i g_i = 1$, where 1 on the right hand side denotes the constant polynomial 1.

What is happening? Does the Nullstellensatz prove **coNP** = **NP**? No, because the degrees of the g_i's —and hence the number of bits used to represent them— could be exponential in n, m. (And it is simple to construct f_i's for which this is necessary.)

However, if **NP** = **coNP** then there would be some *other* notion of a short certificate to the effect that the system is infeasible. The effect of such a result on mathematics would probably be even greater than the effect of Hilbert's Nullstellensatz. Of course, one can replace Nullstellensatz with any other **coNP** problem in the above discussion.

WHAT HAVE WE LEARNED?

- The class **NP** consists of all the languages for which membership can be certified to a polynomial-time algorithm. It contains many important problems not known to be in **P**. **NP** can also be defined using non-deterministic Turing machines.

- **NP**-complete problems are the hardest problems in **NP**, in the sense that they have a polynomial-time algorithm if and only if **P** =**NP**. Many natural problems that seemingly have nothing to do with Turing machines turn out to be **NP**-complete. One such example is the language 3SAT of satisfiable Boolean formulae in $3CNF$ form.

- If **P** = **NP** then for every search problem for which one can efficiently verify a given solution, one can also efficiently find such a solution from scratch.

Chapter notes and history

In the 1950's, Soviet scientists were aware of the undesirability of using exhaustive or brute force search, which they called *perebor*, for combinatorial problems, and asked the question of whether

certain problems *inherently* require such search (see [Tra84]). In the west the first published description of this issue is by Edmonds [Edm65], in the paper quoted in the previous chapter. However, on both sides of the iron curtain it took some time to realize the right way to formulate the problem and to arrive at the modern definition of the classes **NP** and **P**. Amazingly, in his 1956 letter to von Neumann we quoted above, Gödel essentially asks the question of **P** vs. **NP**, although there is no hint that he realized that one of the particular problems he mentions is **NP**-complete. Unfortunately, von Neumann was very sick at the time, and as far as we know, no further research was done by either on them on this problem, and the letter was only discovered in the 1980's.

In 1971 Cook published his seminal paper defining the notion of **NP**-completeness and showing that SAT is **NP** complete [Coo71]. Soon afterwards, Karp [Kar72] showed that 21 important problems are in fact **NP**-complete, generating tremendous interest in this notion. Meanwhile in the USSR Levin independently defined **NP**-completeness (although he focused on search problems) and showed that a variant of SAT is **NP**-complete. (Levin's paper [Lev73] was published in 1973, but he had been giving talks on his results since 1971, also in those days there was essentially zero communication between eastern and western scientists.) See Sipser's survey [Sip92] for more on the history of **P** and **NP** and a full translation of Gödel's remarkable letter.

The "TSP book" by Lawler et al. [LLKS85] also has a similar chapter, and it traces interest in the Traveling Salesman Problem back to the 19th century. Furthermore, a recently discovered letter by Gauss to Schumacher shows that Gauss was thinking about methods to solve the famous *Euclidean Steiner Tree* problem —today known to be **NP**-hard— in the early 19th century.

As mentioned above, the book by Garey and Johnson [GJ79] and the web site [CK00] contain many more examples of **NP** complete problem. Also, Aaronson [Aar05] surveys various attempts to solve **NP** complete problems via "non-traditional" computing devices.

Even if **NP** $\neq$ **P**, this does not necessarily mean that all of the utopian applications mentioned in Section 2.7.3 are gone. It may be that, say, 3SAT is hard to solve in the worst case on every input but actually very easy on the average, See Chapter 15 for a more detailed study of *average-case* complexity. Also, Impagliazzo [Imp95] has an excellent survey on this topic.

Exercises

§1 Prove the existence of a *non-deterministic Universal TM* (analogously to the deterministic universal TM of Theorem 1.13). That is, prove that there exists a representation scheme of NDTMs, and an NDTM $\mathcal{NU}$ such that for every string α, and input x, $\mathcal{NU}(x, \alpha) = M_\alpha(x)$.

  (a) Prove that there exists such a universal NDTM $\mathcal{NU}$ such that if M_α halts on x within T steps, then $\mathcal{NU}$ halts on x, α within $CT \log T$ steps (where C is a constant depending only on the machine represented by α).

  (b) Prove that there is such a universal NDTM that runs on these inputs for at most Ct steps.

Hint: A simulation in $O(|a|t \log t)$ time can be obtained by a straightforward adaptation of the proof of Theorem 1.13. To do a more efficient simulation, the main idea is to first run a simulation of M without actually reading the contents of the work tapes, but rather simply non-deterministically guessing these contents, and writing those guesses down. Then, go over tape by tape and verify that all guesses were consistent.

§2 Prove Theorem 2.

§3 Let HALT be the Halting language defined in Theorem 1.17. Show that HALT is **NP**-hard. Is it **NP**-complete?

§4 We have defined a relation $\leq_p$ among languages. We noted that it is *reflexive* (that is, $A \leq_p A$ for all languages A) and *transitive* (that is, if $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$). Show that it is not commutative, namely, $A \leq_p B$ need not imply $B \leq_p A$.

§5 Suppose $L_1, L_2 \in$ **NP**. Then is $L_1 \cup L_2$ in **NP**? What about $L_1 \cap L_2$?

§6 Mathematics can be axiomatized using for example the *Zermelo Frankel* system, which has a finite description. Argue at a high level that the following language is **NP**-complete.

$$\{\langle \varphi, 1^n \rangle : \text{ math statement } \varphi \text{ has a proof of size at most } n \text{ in the ZF system}\}.$$

Hint: Why is this language in **NP**? Is boolean satisfiability a mathematical statement?

The question of whether this language is in **P** is essentially the question asked by Gödel in the chapter's initial quote.

§7 Show that **NP** = **coNP** iff 3SAT and TAUTOLOGY are polynomial-time reducible to one another.

§8 Can you give a definition of **NEXP** without using NDTMs, analogous to the definition of **NP** in Definition 2.1? Why or why not?

§9 We say that a language is **NEXP**-complete if it is in **NEXP** and every language in **NEXP** is polynomial-time reducible to it. Describe a **NEXP**-complete language. Prove that if this problem is in **EXP** then **NEXP** = **EXP**.

§10 Show that for every time constructible $T : \mathbb{N} \to \mathbb{N}$, if $L \in$ **NTIME**$(T(n))$ then we can give a polynomial-time Karp reduction from L to 3SAT that transforms instances of size n into 3CNF formulae of size $O(T(n) \log T(n))$. Can you make this reduction also run in $O(T(n) \log T(n))$?

§11 Recall that a reduction f from an **NP**-language L to an **NP**-languages L' is *parsimonious* if the number of certificates of f is equal to the number of certificates of $f(x)$.

   (a) Prove that the reduction from every **NP**-language L to SAT presented in the proof of Lemma 2.12 is parsimonious.

(b) Show a parsimonious reduction from SAT to 3SAT.

§12 The notion of polynomial-time reducibility used in Cook's paper was somewhat different: a language A is *polynomial-time Cook reducible* to a language B if there is a polynomial time TM M that, given an *oracle* for deciding B, can decide A. (An oracle for B is a magical extra tape given to M, such that whenever M writes a string on this tape and goes into a special "invocation" state, then the string —in a single step!—gets overwritten by 1 or 0 depending upon whether the string is or is not in B, see Section **??**)

Show that the notion of cook reducibility is transitive and that 3SAT is Cook-reducible to TAUTOLOGY.

§13 (Berman's Theorem 1978) A language is called *unary* if every string in it is of the form 1^i (the string of i ones) for some $i > 0$. Show that if a unary language is **NP**-complete then **P** = **NP**. (See Exercise 6 of Chapter 6 for a strengthening of this result.)

self reducibility argument of Theorem 2.19.
obtain a polynomial-time algorithm for SAT using the downward
to some string of the form 1^i where $i \lesssim n^c$. Use this observation to
guage L, then this reduction can only map size n instances of 3SAT
Hint: If there is a n^c time reduction from 3SAT to a unary lan-

§14 In the CLIQUE problem we are given an undirected graph G and an integer K and have to decide whether there is a subset S of at least K vertices such that every two distinct vertices $u, v \in S$ have an edge between them (such a subset is called a *clique*). In the VERTEX COVER problem we are given an undirected graph G and an integer K and have to decide whether there is a subset S of at most K vertices such that for every edge $\{i, j\}$ of G, at least one of i or j is in S. Prove that both these problems are **NP**-complete.

Hint: reduce from INDSET.

§15 In the MAX CUT problem we are given an undirected graph G and an integer K and have to decide whether there is a subset of vertices S such that there are at least K edges that have one endpoint in S and one endpoint in $\overline{S}$. Prove that this problem is **NP**-complete.

§16 In the Exactly One 3SAT problem, we are given a 3CNF formula φ and need to decide if there exists a satisfying assignment u for φ such that every clause of φ has exactly one TRUE literal. In the SUBSET SUM problem we are given a list of n numbers $A_1, \ldots, A_n$ and a number T and need to decide whether there exists a subset $S \subseteq [n]$ such that $\sum_{i \in S} A_i = T$ (the problem size is the sum of all the bit representations of all numbers). Prove that both Exactly One3SAT and SUBSET SUM are **NP**-complete.

Hint: For Exactly One 3SAT replace each occurrence of a literal u_i in a clause C by a new variable $z_{i,C}$ and auxiliary variables ensuring that if u_i is TRUE then $z_{i,C}$ is allowed to be either TRUE or FALSE but if u_i is false then $z_{i,C}$ must be FALSE. The approach for the reduction of Exactly One 3SAT to SUBSET SUM is that given a formula φ, we map it to a SUBSET SUM instance by mapping each possible literal u_i to the number $\sum_{j \in S_i} (2n)^j$ where S_i is the set of clauses that the literal u_i satisfies, and setting the target T to be $\sum_{j=1}^m (2n)^j$. An additional trick is required to ensure that the solution to the subset sum instance will not include two literals that correspond to a variable and its negation.

§17 Prove that the language HAMPATH of *undirected* graphs with Hamiltonian paths is **NP**-complete. Prove that the language TSP described in Example 2.2 is **NP**-complete. Prove that the language HAMCYCLE of undirected graphs that contain Hamiltonian cycle (a simple cycle involving all the vertices) is **NP**-complete.

§18 Let quadeq be the language of all satisfiable sets of *quadratic equations* over 0/1 variables (a quadratic equations over $u_1, \ldots, u_n$ has the form form $\sum_{a_{i,j}} u_i u_j = b$), where addition is modulo 2. Show that quadeq is **NP**-complete.

Hint: Reduce from SAT

§19 Prove that $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{coNP}$.

§20 Prove that the Definitions 2.21 and 2.22 do indeed define the same class **coNP**.

§21 Suppose $L_1, L_2 \in \mathbf{NP} \cap \mathbf{coNP}$. Then show that $L_1 \oplus L_2$ is in $\mathbf{NP} \cap \mathbf{coNP}$, where $L_1 \oplus L_2 = \{x : x$ is in exactly one of $L_1, L_2\}$.

§22 Define the language UNARY SUBSET SUM to be the variant of the SUBSET SUM problem of Exercise 16 where all numbers are represented by the *unary* representation (i.e., the number k is represented as 1^k). Show that UNARY SUBSET SUM is in **P**.

Hint: Start with an exponential-time recursive algorithm for SUB-SET SUM, and show that in this case you can make it into a polynomial-time algorithm by storing previously computed values in a table.

§23 Prove that if every *unary* **NP**-language is in **P** then **EXP** = **NEXP**. (A language L is unary if it is a subset of $\{1\}^*$, see Exercise 13.)

DRAFT

Chapter 3

Diagonalization

> *"..the relativized $\mathbf{P} =?\mathbf{NP}$ question has a positive answer for some oracles and a negative answer for other oracles. We feel that this is further evidence of the difficulty of the $\mathbf{P} =?\mathbf{NP}$ question."*
> Baker, Gill, Solovay. [BGS75]

One basic goal in complexity theory is to separate interesting complexity classes. To separate two complexity classes we need to exhibit a machine in one class that gives a different answer on some input from *every* machine in the other class. This chapter describes *diagonalization*, essentially the only general technique known for constructing such a machine. We have already seen diagonalization in Section 1.4, where it was used to show the existence of uncomputable functions. In this chapter, we first use diagonalization to prove *hierarchy theorems*, according to which giving Turing machines more computational resources (such as time, space, and non-determinism) allows them to solve a strictly larger number of problems. We will also use it to show that if $\mathbf{P} \neq \mathbf{NP}$ then there exist problems that are neither in $\mathbf{P}$ nor $\mathbf{NP}$-complete.

Though diagonalization led to some of these early successes of complexity theory, researchers realized in the 1970s that diagonalization alone may not resolve $\mathbf{P}$ versus $\mathbf{NP}$ and other interesting questions; see Section 3.5. Interestingly, the limits of diagonalization are proved using diagonalization.

This last result caused diagonalization to go out of favor for many years. But some recent results (see Section 16.3 for an example) use diagonalization as a key component. Thus future complexity theorists should master this simple idea before going on to anything fancier!

Machines as strings and the universal TM. The one common tool used in all diagonalization proofs is the representation of TMs by strings, such that given a string x a *universal* TM can simulate the machine M_x represented by x with a small (i.e. at most logarithmic) overhead, see Theorems 1.13, **??** and **??**. Recall that we assume that every string x represents some machine and every machine is represented by infinitely many strings. For $i \in \mathbb{N}$, we will also use the notation M_i for the machine represented by the string that is the binary expansion of the number i (ignoring the leading 1).

3.1 Time Hierarchy Theorem

The Time Hierarchy Theorem shows that allowing Turing Machines more computation time strictly increases the class of languages that they can decide. Recall that a function $f : \mathbb{N} \to \mathbb{N}$ is a *time-constructible* function if there is a Turing machine that, given the input 1^n, writes down $1^{f(n)}$ on its tape in $O(f(n))$ time. Usual functions like $n \log n$ or n^2 satisfy this property, and we will restrict attention to running times that are time-constructible.

THEOREM 3.1
If f, g are time-constructible functions satisfying $f(n) \log f(n) = o(g(n))$, then

$$\mathbf{DTIME}(f(n)) \subsetneq \mathbf{DTIME}(g(n)) \tag{1}$$

PROOF: To showcase the essential idea of the proof of Theorem 3.1, we prove the simpler statement $\mathbf{DTIME}(n) \subsetneq \mathbf{DTIME}(n^{1.5})$.

Consider the following Turing Machine D: "*On input x, run for $|x|^{1.4}$ steps the Universal TM $\mathcal{U}$ of Theorem 1.13 to simulate the execution of M_x on x. If M_x outputs an answer in this time, namely, $M_x(x) \in \{0, 1\}$ then output the opposite answer (i.e., output $1 - M_x(x)$). Else output 0.*" Here M_x is the machine represented by the string x.

By definition, D halts within $n^{1.4}$ steps and hence the language L decided by D is in $\mathbf{DTIME}(n^{1.5})$. We claim that $L \notin \mathbf{DTIME}(n)$.

For contradiction's sake assume that some TM M decides L but runs in time cn on inputs of size n. Then every $x \in \{0, 1\}^*$, $M(x) = D(x)$.

The time to simulate M by the universal Turing machine $\mathcal{U}$ on every input x is at most $c'c|x| \log |x|$ for some constant c' (depending on the alphabet size and number of tapes and states of M, but independent of $|x|$). There exists a number n_0 such that for every $n \geq n_0$, $n^{1.4} > c'cn \log n$. Let x be a string representing the machine M of length at least n_0 (there exists such a string since M is represented by infinitely many strings). Then, $D(x)$ will obtain the output $M(x)$ within $|x|^{1.4}$ steps, but by definition of D, we have $D(x) = 1 - M(x) \neq M(x)$. Thus we have derived a contradiction. ■

3.2 Space Hierarchy Theorem

The space hierarchy theorem is completely analogous to the time hierarchy theorem. One restricts attention to *space-constructible* functions, which are functions $f : \mathbb{N} \to \mathbb{N}$ for which there is a machine that, given any n-bit input, constructs $f(n)$ in space $O(f(n))$. The proof of the next theorem is completely analogous to that of Theorem 3.1. (The theorem does not have the $\log f(n)$ factor because the universal machine for space-bounded computation incurs only a constant factor overhead in space; see Theorem **??**.)

THEOREM 3.2
If f, g are space-constructible functions satisfying $f(n) = o(g(n))$, then

$$\mathbf{SPACE}(f(n)) \subsetneq \mathbf{SPACE}(g(n)) \tag{2}$$

3.3 Nondeterministic Time Hierarchy Theorem

The following is the hierarchy theorem for *non-deterministic* Turing machines.

THEOREM 3.3
If f, g are time constructible functions satisfying $f(n + 1) = o(g(n))$, then

$$\mathbf{NTIME}(f(n)) \subsetneq \mathbf{NTIME}(g(n)) \tag{3}$$

PROOF: Again, we just showcase the main idea by proving $\mathbf{NTIME}(n) \subsetneq \mathbf{NTIME}(n^{1.5})$. The technique from the previous section does not directly apply, since it has to determine the answer of a TM in order to flip it. To determine the answer of a nondeterminisitic that runs in $O(n)$ time, we may need to examine as many as $2^{\Omega(n)}$ possible strings of non-deterministic choices. So it is unclear that how the "diagonalizer" machine can determine in $O(n^{1.5})$ (or even $O(n^{100})$) time how to flip this answer. Instead we introduce a technique called *lazy* diagonalization, which is only guaranteed to flip the answer on some input in a fairly large range.

For every $i \in \mathbb{N}$ we denote by M_i the non-deterministic TM represented by i's binary expansion according to the universal NDTM $\mathcal{NU}$ (see Theorem **??**). We define the function $f : \mathbb{N} \to \mathbb{N}$ as follows: $f(1) = 2$ and $f(i + 1) = 2^{f(i)^{1.2}}$. Note that given n, we can can easily find in $O(n^{1.5})$ time the number i such that n is sandwiched between $f(i)$ and $f(i + 1)$. Our diagonalizing machine D will try to flip the answer of M_i on *some* input in the set $\{1^n : f(i) < n \leq f(i + 1)\}$. It is defined as follows:

"On input x, if $x \notin 1^*$, reject. If $x = 1^n$, then compute i such that $f(i) < n \leq f(i + 1)$ and

1. *If $f(i) < n < f(i + 1)$ then simulate M_i on input 1^{n+1} using nondeterminism in $n^{1.1}$ time and output its answer. (If the simulation takes more than that then halt and accept.)*

2. *If $n = f(i + 1)$, accept 1^n iff M_i rejects $1^{f(i)+1}$ in $(f(i) + 1)^{1.5}$ time.*"

Note that Part 2 requires going through all possible $\exp((f(i) + 1)^{1.1})$ branches of M_i on input $1^{f(i)+1}$, but that is fine since the input size $f(i + 1)$ is $2^{f(i)^{1.2}}$. We conclude that NDTM D runs in $O(n^{1.5})$ time. Let L be the language decided by D. We claim that $L \notin \mathbf{NTIME}(n)$.

Indeed, suppose for the sake of contradiction that L is decided by an NDTM M running in cn steps (for some constant c). Since each NDTM is represented by infinitely many strings, we can find i large enough such that $M = M_i$ and on inputs of length $n \geq f(i)$, M_i can be simulated in less than $n^{1.1}$ steps. Thus the two steps in the description of D ensure respectively that

$$\text{If } f(i) < n < f(i + 1), \quad \text{then } D(1^n) = M_i(1^{n+1}) \tag{4}$$
$$D(1^{f(i+1)}) \neq M_i(1^{f(i)+1}) \tag{5}$$

see Figure 3.1.

By our assumption M_i and D agree on all inputs 1^n for $n \in (f(i), f(i + 1)]$. Together with (4), this implies that $D(1^{f(i+1)}) = M_i(1^{f(i)+1})$, contradicting (5). ∎
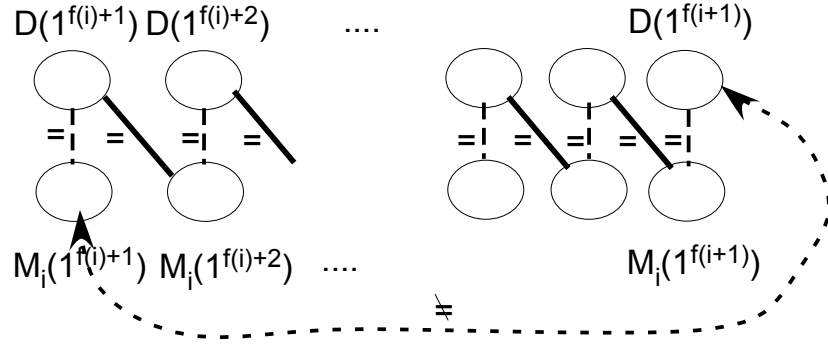
Figure 3.1: The values of D and M_i on inputs 1^n for $n \in (f(i), f(i+1)]$. Full lines denote equality by the design of D, dashed lines denote equality by the assumption that $D(x) = M_i(x)$ for every x, and the dashed arrow denotes inequality by the design of D. Note that together all these relations lead to contradiction.

3.4 Ladner's Theorem: Existence of NP-intermediate problems.

One of the striking aspects of **NP**-completeness is the surprisingly large number of **NP**-problems –including some that were studied for many decades— that turned out to be **NP**-complete. This phenomenon suggests a bold conjecture: every problem in **NP** is either in **P** or **NP** complete. We show that if $\mathbf{P} \neq \mathbf{NP}$ then this is false —there is a language $L \in \mathbf{NP} \setminus \mathbf{P}$ that is not **NP**-complete. (If $\mathbf{P} = \mathbf{NP}$ then the conjecture is trivially true but uninteresting.) The rest of this section proves this.

THEOREM 3.4 (LADNER'S THEOREM [**?**])
Suppose that $\mathbf{P} \neq \mathbf{NP}$. *Then there exists a language* $L \in \mathbf{NP} \setminus \mathbf{P}$ *that is not* **NP**-*complete.*

PROOF: If $\mathbf{P} \neq \mathbf{NP}$ then we know at least one language in $\mathbf{NP} \setminus \mathbf{P}$: namely, the **NP**-complete language SAT. Consider the language SAT_H of length n satisfiable formulae that are padded with $n^{H(n)}$ 1's for some polynomial-time computable function $H : \mathbb{N} \to \mathbb{N}$ (i.e., $\mathsf{SAT}_H = \left\{ \psi 01^{n^{H(n)}} : \psi \in \mathsf{SAT} \text{ and } n = |\psi| \right\}$). Consider two possibilities:

(a) $H(n)$ *is at most some constant c for every n.* In this case SAT_H is simply SAT with a polynomial amount of "padding." Thus, SAT_H is also **NP**-complete and is not in **P** if $\mathbf{P} \neq \mathbf{NP}$.

(b) $H(n)$ *tends to infinity with n*, and thus the padding is of superpolynomial size. In this case, we claim that SAT_H cannot be **NP**-complete. Indeed, if there is a $O(n^i)$-time reduction f from SAT to SAT_H then such a reduction reduces the satisfiability of SAT instances of length n to instances of SAT_H of length $O(n^i)$, which must have the form $\psi 01^{|\psi|^{H(|\psi|)}}$, where $|\psi| + |\psi|^{H(|\psi|)} = O(n^i)$, and hence $|\psi| = o(n)$. In other words, we have a polynomial-time reduction from SAT instances of length n to SAT instances of length $o(n)$, which implies SAT can be solved in polynomial time. (The algorithm consists of applying the reduction again and again, reducing the size of the instances each time until the instance is of size $O(1)$ and can be solved in $O(1)$ time by brute force) This is a contradiction to the assumption $\mathbf{P} \neq \mathbf{NP}$.

The proof of the Theorem uses a language SAT_H for a function H that in some senses combines the two cases above. This function tends to infinity with n, so that SAT_H is not **NP**-complete as in Case (b), but grows slowly enough to assure $\mathsf{SAT}_H \notin \mathbf{P}$ as in Case (a). Function H is defined as follows:

$H(n)$ is the smallest number $i < \log \log n$ such that for every $x \in \{0,1\}^*$ with $|x| \leq \log n$,

M_i halts on x within $i|x|^i$ steps and M_i outputs 1 iff $x \in \mathsf{SAT}_H$

where M_i is the machine represented by the binary expansion of i according to the representation scheme of the universal Turing machine $\mathcal{U}$ of Theorem 1.13. If there is no such i then we let $H(n) = \log \log n$.

Notice, this is implicitly a recursive definition since the definition of H depends on SAT_H, but a moment's thought shows that H is well-defined since $H(n)$ determines membership in SAT_H of strings whose length is greater than n, and the definition of $H(n)$ only relies upon checking the status of strings of length at most $\log n$.

There is a trivial algorithm to compute $H(n)$ in $O(n^3)$ time. After all, we only need to **(1)** compute $H(k)$ for every $k \leq \log n$, **(2)** simulate at most $\log \log n$ machines for every input of length at most $\log n$ for $\log \log n (\log n)^{\log \log n} = o(n)$ steps, and **(3)** compute SAT on all the inputs of length at most $\log n$.

Now we have the following two claims.

CLAIM 1: SAT_H *is not in* **P**. Suppose, for the sake of contradiction, that there is a machine M solving SAT_H in at most cn^c steps. Since M is represented by infinitely many strings, there is a number $i > c$ such that $M = M_i$. By the definition of $H(n)$ this implies that for $n > 2^{2^i}$, $H(n) \leq i$. But this means that for all sufficiently large input lengths, SAT_H is simply the language SAT padded with a polynomial (i.e., n^i) number of 1's, and so cannot be in **P** unless $\mathbf{P} = \mathbf{NP}$.

CLAIM 2: SAT_H *is not* **NP**-*complete*. As in Case (b), it suffices to show that $H(n)$ tends to infinity with n. We prove the equivalent statement that for every integer i, there are only finitely many n's such that $H(n) = i$: since $\mathsf{SAT}_H \notin \mathbf{P}$, for each i we know that there is an input x such that given $i|x|^i$ time, M_i gives the incorrect answer to whether or not $x \in \mathsf{SAT}_H$. Then the definition of H ensures that for every $n > 2^{|x|}$, $H(x) \neq i$.

■

REMARK 3.5
We do not know of a natural decision problem that, assuming $\mathbf{NP} \neq \mathbf{P}$, is proven to be in $\mathbf{NP} \setminus \mathbf{P}$ but not **NP**-complete, and there are remarkably few candidates for such languages. However, there are a few fascinating examples for languages not known to be either in **P** nor **NP**-complete. Two such examples are the *Factoring* and *Graph isomorphism* languages (see Example 2.2). No polynomial-time algorithm is currently known for these languages, and there is some evidence that they are not **NP** complete (see Chapter 8).

3.5   Oracle machines and the limits of diagonalization?

Quantifying the limits of "diagonalization" is not easy. Certainly, the diagonalization in Sections 3.3 and 3.4 seems more clever than the one in Section 3.1 or the one that proves the undecidability of the halting problem.

For concreteness, let us say that "diagonalization" is any technique that relies upon the following properties of Turing machines:

I The existence of an effective representation of Turing machines by strings.

II The ability of one TM to simulate any another without much overhead in running time or space.

Any argument that only uses these facts is treating machines as *blackboxes*: the machine's internal workings do not matter. We now show a general way to define a variant of Turing Machines called *oracle Turing Machines* that still satisfy the above two properties. However, one way of defining the variants results in TMs for which $\mathbf{P} = \mathbf{NP}$, whereas the other way results in TMs for which $\mathbf{P} \neq \mathbf{NP}$. We conclude that to resolve $\mathbf{P}$ versus $\mathbf{NP}$ we need to use some other property besides the above two.

Oracle machines will be used elsewhere in this book in other contexts. These are machines that are given access to an "oracle" that can magically solve the decision problem for some language $O \subseteq \{0,1\}^*$. The machine has a special *oracle tape* on which it can write a string $q \in \{0,1\}^*$ on a and in one step gets an answer to a query of the form "Is q in O?". This can be repeated arbitrarily often with different queries. If O is a difficult language that cannot be decided in polynomial time then this oracle gives an added power to the TM.

DEFINITION 3.6 (ORACLE TURING MACHINES)
An *oracle* Turing machine is a TM M that has a special read/write tape we call M's *oracle tape* and three special states $q_{\mathsf{query}}, q_{\mathsf{yes}}, q_{\mathsf{no}}$. To execute M, we specify in addition to the input a language $O \subseteq \{0,1\}^*$ that is used as the *oracle* for M. Whenever during the execution M enters the state q_{query}, the machine moves into the state q_{yes} if $q \in O$ and q_{no} if $q \notin O$, where q denotes the contents of the special oracle tape. Note that, regardless of the choice of O, a membership query to O counts only as a single computational step. If M is an oracle machine, $O \subseteq \{0,1\}^*$ a language, and $x \in \{0,1\}^*$, then we denote the output of M on input x and with oracle O by $M^O(x)$.

Nondeterministic oracle TMs are defined similarly.

DEFINITION 3.7
For every $O \subseteq \{0,1\}^*$, $\mathbf{P}^O$ is the set of languages decided by a polynomial-time deterministic TM with oracle access to O and $\mathbf{NP}^O$ is the set of languages decided by a polynomial-time nondeterministic TM with oracle access to O.

To illustrate these definitions we show a few simple claims.

CLAIM 3.8
1. Let $\overline{\mathsf{SAT}}$ *denote the language of* unsatisfiable *formulae. Then* $\overline{\mathsf{SAT}} \in \mathbf{P}^{\mathsf{SAT}}$.

2. Let $O \in \mathbf{P}$. Then $\mathbf{P}^O = \mathbf{P}$.

3. Let EXPCOM *be the following language*

$$\left\{ \langle M, x, 1^n \rangle : M \text{ outputs 1 on } x \text{ within } 2^n \text{ steps} \right\}.$$

Then $\mathbf{P}^{\mathsf{EXPCOM}} = \mathbf{NP}^{\mathsf{EXPCOM}} = \mathbf{EXP}$. *(Recall that* $\mathbf{EXP} = \cup_c \mathbf{DTIME}(2^{n^c})$.)

PROOF:

1. Given oracle access to SAT, to decide whether a formula φ is in $\overline{\mathsf{SAT}}$, the machine asks the oracle if $\varphi \in \mathsf{SAT}$, and then gives the opposite answer as its output.

2. Allowing an oracle can only help compute more languages and so $\mathbf{P} \subseteq \mathbf{P}^O$. If $O \in \mathbf{P}$ then it is redundant as an oracle, since we can transform any polynomial-time oracle TM using O into a standard (no oracle) by simply replacing each oracle call with the computation of O. Thus $\mathbf{P}^O \subseteq \mathbf{P}$.

3. Clearly, an oracle to EXPCOM allows one to perform an exponential-time computation at the cost of one call, and so $\mathbf{EXP} \subseteq \mathbf{P}^{\mathsf{EXPCOM}}$. On the other hand, if M is a non-deterministic polynomial-time oracle TM, we can simulate its execution with a EXPCOM oracle in exponential time: such time suffices both to enumerate all of M's non-deterministic choices and to answer the EXPCOM oracle queries. Thus, $\mathbf{EXP} \subseteq \mathbf{P}^{\mathsf{EXPCOM}} \subseteq \mathbf{NP}^{\mathsf{EXPCOM}} \subseteq \mathbf{EXP}$.

∎

The key fact to note about oracle TMs is the following: *Regardless of what oracle O is, the set of all oracle TM's with access to oracle O satisfy Properties I and II above.* The reason is that we can represent TMs with oracle O as strings, and we have a universal TM $\mathcal{OU}$ that, using access to O, can simulate every such machine with logarithmic overhead, just as Theorem 1.13 shows for non-oracle machines. Indeed, we can prove this in exactly the same way of Theorem 1.13, except that whenever in the simulation M makes an oracle query, $\mathcal{OU}$ forwards the query to its own oracle.

Thus any result about TMs or complexity classes that uses only Properties I and II above also holds for the set of all TMs with oracle O. Such results are called *relativizing* results.

All of the results on universal Turing machines and the diagonalizations results in this chapter are of this type.

The next theorem implies that whichever of $\mathbf{P} = \mathbf{NP}$ or $\mathbf{P} \neq \mathbf{NP}$ is true, it cannot be a relativizing result.

THEOREM 3.9 (BAKER, GILL, SOLOVAY [BGS75])
There exist oracles A, B *such that* $\mathbf{P}^A = \mathbf{NP}^A$ *and* $\mathbf{P}^B \neq \mathbf{NP}^B$.

PROOF: As seen in Claim 3.8, we can use $A = \mathsf{EXPCOM}$. Now we construct B. For any language B, let U_B be the unary language

$$U_B = \left\{ 1^n : \text{some string of length } n \text{ is in } B \right\}.$$

For every oracle B, the language U_B is clearly in $\mathbf{NP}^B$, since a non-deterministic TM can make a non-deterministic guess for the string $x \in \{0,1\}^n$ such that $x \in B$. Below we construct an oracle B such that $U_B \notin \mathbf{P}^B$, implying that $\mathbf{P}^B \neq \mathbf{NP}^B$.

Construction of B: For every i, we let M_i be the oracle TM represented by the binary expansion of i. We construct B in stages, where stage i ensures that M_i^B does not decide U_B in $2^n/10$ time. Initially we let B be empty, and gradually add strings to it. Each stage determines the status (i.e., whether or not they will ultimately be in B) of a finite number of strings.

Stage i: So far, we have declared for a finite number of strings whether or not they are in B. Choose n large enough so that it exceeds the length of any such string, and run M_i on input 1^n for $2^n/10$ steps. Whenever it queries the oracle about strings whose status has been determined, we answer consistently. When it queries strings whose status is undetermined, we declare that the string is not in B. Note that until this point, we have not declared that B has any string of length n. Now we make sure that if M_i halts within $2^n/10$ steps then its answer on 1^n is incorrect. If M_i accepts, we declare that all strings of length n are not in B, thus ensuring $1^n \notin B_u$. If M_i rejects, we pick a string of length n that it has not queried (such a string exists because M_i made at most $2^n/10$ queries) and declare that it is in B, thus ensuring $1^n \in B_u$. In either case, the answer of M_i is incorrect. Our construction ensures that U_B is not in $\mathbf{P}^B$ (and in fact not in $\mathbf{DTIME}^B(f(n))$ for every $f(n) = o(2^n)$). ∎

Let us now answer our original question: Can diagonalization or any simulation method resolve $\mathbf{P}$ vs $\mathbf{NP}$? Answer: Possibly, but it has to use some fact about TMs that does not hold in presence of oracles. Such facts are termed *nonrelativizing* and we will later see examples of such facts. However, a simple one was already encountered in Chapter **??**: the Cook-Levin theorem! It is not true for a general oracle A that every language $L \in \mathbf{NP}^A$ is polynomial-time reducible to 3SAT (see Exercise 6). Note however that nonrelativizing facts are necessary, not sufficient. It is an open question how to use known nonrelativizing facts in resolving $\mathbf{P}$ vs $\mathbf{NP}$ (and many interesting complexity theoretic conjectures).

Whenever we prove a complexity-theoretic fact, it is useful to check whether or not it can be proved using relativizing techniques. The reader should check that Savitch's theorem (Corollary **??**) and Theorem 4.18 do relativize.

Later in the book we see other attempts to separate complexity classes, and we will also try to quantify —using complexity theory itself!—why they do not work for the $\mathbf{P}$ versus $\mathbf{NP}$ question.

WHAT HAVE WE LEARNED?

- Diagonalization uses the representation of Turing machines as strings to separate complexity classes.

- We can use it to show that giving a TM more of the same type of resource (time, non-determinism, space) allows it to solve more problems, and to show that, assuming $\mathbf{NP} \neq \mathbf{P}$, $\mathbf{NP}$ has problems neither in $\mathbf{P}$ nor $\mathbf{NP}$-complete.

- Results proven solely using diagonalization *relativize* in the sense that they hold also for TM's with oracle access to O, for every oracle $O \subseteq \{0,1\}^*$. We can use this to show the limitations of such methods. In particular, relativizing methods alone cannot resolve the $\mathbf{P}$ vs. $\mathbf{NP}$ question.

Chapter notes and history

Georg Cantor invented diagonalization in the 19th century to show that the set of real numbers is uncountable. Kurt Gödel used a similar technique in his proof of the *Incompleteness Theorem.* Computer science undergraduates often encounter diagonalization when they are taught the undecidabilty of the *Halting Problem.*

The time hierarchy theorem is from Hartmanis and Stearns' pioneering paper [HS65]. The space hierarchy theorem is from Stearns, Hartmanis, and Lewis [SHL65]. The nondeterministic time hierarchy theorem is from Cook [Coo73], though the simple proof given here is essentially from [Zak83]. A similar proof works for other complexity classes such as the (levels of the) polynomial hierarchy discussed in the next chapter. Ladner's theorem is from [?] but the proof here is due to an unpublished manuscript by Impagliazzo. The notion of relativizations of the **P** versus **NP** question is from Baker, Gill, and Solovay [BGS75], though the authors of that paper note that other researchers independently discovered some of their ideas. The notion of relativization is related to similar ideas in logic (such as *independence results*) and recursive function theory.

The notion of oracle Turing machines can be used to study interrelationships of complexity classes. In fact, Cook [Coo71] defined **NP**-completeness using oracle machines. A subfield of complexity theory called *structural complexity* has carried out a detailed study of oracle machines and classes defined using them; see [].

Whether or not the Cook-Levin theorem is a nonrelativizing fact depends upon how you formalize the question. There is a way to allow the 3SAT instance to "query" the oracle, and then the Cook-Levin theorem does relativize. However, it seems safe to say that any result that uses the locality of computation is looking at the internal workings of the machine and hence is potentially nonrelativizing.

The term *superiority* introduced in the exercises does not appear in the literature but the concept does. In particular, ??? have shown the limitations of relativizing techniques in resolving certain similar open questions.

Exercises

§1 Show that the following language is undecidable:

$$\left\{ \llcorner M \lrcorner :\ M \text{ is a machine that runs in } 100n^2 + 200 \text{ time} \right\}.$$

§2 Show that **SPACE**$(n) \neq$ **NP**. (Note that we do not know if either class is contained in the other.)

§3 Show that there is a language $B \in$ **EXP** such that $\mathbf{NP}^B \neq \mathbf{P}^B$.

§4 Say that a class C_1 is *superior to* a class C_2 if there is a machine M_1 in class C_1 such that for every machine M_2 in class C_2 and every large enough n, there is an input of size between n and n^2 on which M_1 and M_2 answer differently.

   (a) Is **DTIME**$(n^{1.1})$ superior to **DTIME**(n)?

   (b) Is **NTIME**$(n^{1.1})$ superior to **NTIME**(n)?

§5 Show that there exists a function that is not time-constructible.

§6 Show that there is an oracle A and a language $L \in \mathbf{NP}^A$ such that L is not polynomial-time reducible to 3SAT even when the machine computing the reduction is allowed access to A.

§7 Suppose we pick a random language B, by deciding for each string independently and with probability $1/2$ whether or not it is in B. Show that with high probability $\mathbf{P}^B \neq \mathbf{NP}^B$. (To give an answer that is formally correct you may need to know elementary measure theory.)

Chapter 4

Space complexity

> *"(our) construction... also suggests that what makes "games" harder than "puzzles"*
> *(e.g. NP-complete problems) is the fact that the initiative ("the move") can shift*
> *back and forth between the players."*
> Shimon Even and Robert Tarjan, 1976

In this chapter we will study the memory requirements of computational tasks. To do this we define *space-bounded computation,* which has to be performed by the TM using a restricted number of tape cells, the number being a function of the input size. We also study *nondeterministic space-bounded TMs.* As in the chapter on **NP**, our goal in introducing a complexity class is to "capture" interesting computational phenomena— in other words, identify an interesting set of computational problems that lie in the complexity class and are *complete* for it. One phenomenon we will "capture" this way (see Section 4.3.2) concerns computation of winning strategies in 2-person games, which seems inherently different from (and possibly more difficult than) solving **NP** problems such as SAT, as alluded to in the above quote. The formal definition of deterministic and non-deterministic space bounded computation is as follows (see also Figure 4.1):
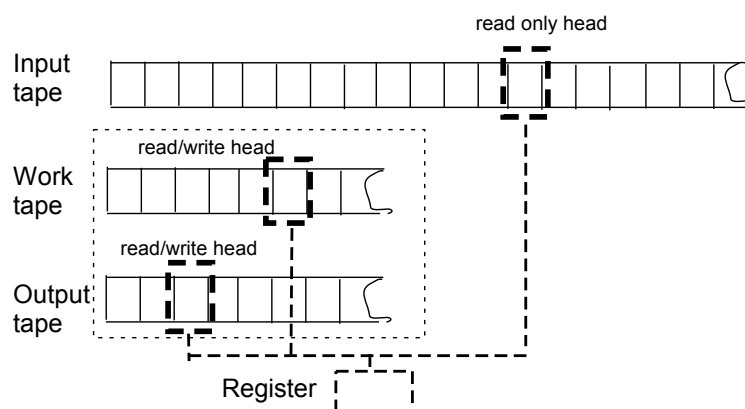
Figure 4.1: Space bounded computation. Only cells used in the read/write tapes count toward the space bound.

DEFINITION 4.1 (SPACE-BOUNDED COMPUTATION.)

Let $S : \mathbb{N} \to \mathbb{N}$ and $L \subseteq \{0,1\}^*$. We say that $L \in \mathbf{SPACE}(s(n))$ (resp. $L \in \mathbf{NSPACE}(s(n))$) if there is a constant c and TM (resp. NDTM) M deciding L such that on every input $x \in \{0,1\}^*$, the total number of locations that are at some point non-blank during M's execution on x is at most $c \cdot s(|x|)$. (Non-blank locations in the read-only input tape do not count.)

As in our definitions of all nondeterministic complexity classes, we require all branches of nondeterministic machines to always halt.

REMARK 4.2

Analogously to time complexity, we will restrict our attention to space bounds $S : \mathbb{N} \to \mathbb{N}$ that are *space-constructible* functions, by which we mean that there is a TM that computes $S(n)$ in $O(S(n))$ space when given 1^n as input. (Intuitively, if S is space-constructible, then the machine "knows" the space bound it is operating under.) This is a very mild restriction since functions of interest, including $\log n, n$ and 2^n, are space-constructible.

Also, realize that since the work tape is separated from the input tape, it makes sense to consider space-bounded machines that use space less than the input length, namely, $S(n) < n$. (This is in contrast to time-bounded computation, where $\mathbf{DTIME}(T(n))$ for $T(n) < n$ does not make much sense since the TM does not have enough time to read the entire input.) We will assume however that $S(n) > \log n$ since the work tape has length n, and we would like the machine to at least be able to "remember" the index of the cell of the input tape that it is currently reading. (One of the exercises explores classes that result when $S(n) \ll \log n$.)

Note that $\mathbf{DTIME}(S(n)) \subseteq \mathbf{SPACE}(S(n))$ since a TM can access only one tape cell per step. Also, notice that space can be *reused*: a cell on the work tape can be overwritten an arbitrary number of times. A space $S(n)$ machine can easily run for as much as $2^{\Omega(S(n))}$ steps —think for example of the machine that uses its work tape of size $S(n)$ to maintain a counter which it increments from 1 to $2^{S(n)-1}$. The next easy theorem (whose proof appears a little later) shows that this is tight in the sense that any languages in $\mathbf{SPACE}(S(n))$ (and even $\mathbf{NSPACE}(S(n))$) is in $\mathbf{DTIME}(2^{O(S(n))})$. Surprisingly enough, up to logarithmic terms, this theorem contains the only relationships we know between the power of space-bounded and time-bounded computation. Improving this would be a major result.

THEOREM 4.3

For every space constructible $S : \mathbb{N} \to \mathbb{N}$,

$$\mathbf{DTIME}(S(n)) \subseteq \mathbf{SPACE}(S(n)) \subseteq \mathbf{NSPACE}(S(n)) \subseteq \mathbf{DTIME}(2^{O(S(n))})$$

4.1 Configuration graphs.

To prove Theorem 4.3 we use the notion of a *configuration graph* of a Turing machine. This notion will also be quite useful for us later in this chapter and the book. Let M be a (deterministic or
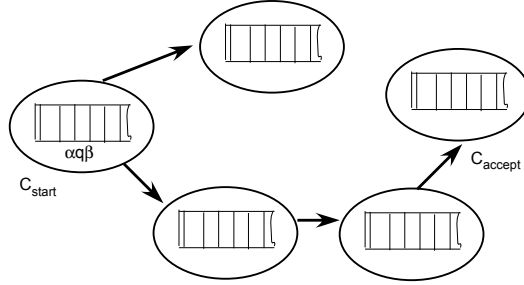
Figure 4.2: The configuration graph $G_{M,x}$ is the graph of all configurations of M's execution on x where there is an edge from a configuration C to a configuration C' if C' can be obtained from C in one step. It has out-degree one if M is deterministic and out-degree at most two if M is non-deterministic.

non-deterministic) TM. A *configuration* of a TM M consists of the contents of all non-blank entries of M's tapes, along with its state and head position, at a particular point in its execution. For every TM M and input $x \in \{0,1\}^*$, the *configuration graph of M on input x*, denoted $G_{M,x}$, is a directed graph whose nodes correspond to possible configurations that M can reach from the starting configuration C_{start}^x (where the input tape is initialized to contain x). The graph has a directed edge from a configuration C to a configuration C' if C' can be reached from C in one step according to M's transition function (see Figure 4.2). Note that if M is deterministic then the graph has out-degree one, and if M is non-deterministic then it has an out-degree at most two. Also note that we can assume that M's computation on x does not repeat the same configuration twice (as otherwise it will enter into an infinite loop) and hence that the graph is a directed acyclic graph (DAG). By modifying M to erase all its work tapes before halting, we can assume that there is only a single configuration C_{accept} on which M halts and outputs 1. This means that M accepts the input x iff there exists a (directed) path in $G_{M,x}$ from C_{start} to C_{accept}. We will use the following simple claim about configuration graphs:

CLAIM 4.4
Let $G_{M,x}$ be the configuration graph of a space-$S(n)$ machine M on some input x of length n. Then,

1. Every vertex in $G_{M,x}$ can be described using $cS(n)$ bits for some constant c (depending on M's alphabet size and number of tapes) and in particular, $G_{M,x}$ has at most $2^{cS(n)}$ nodes.

2. There is an $O(S(n))$-size CNF formula $\varphi_{M,x}$ such that for every two strings C, C', $\varphi_{M,x}(C, C') = 1$ if and only if C, C' encode two neighboring configuration in $G_{M,x}$.

PROOF SKETCH: Part 1 follows from observing that a configuration is completely described by giving the contents of all work tapes, the position of the head, and the state that the TM is in (see Section 1.2.1). We can encode a configuration by first encoding the snapshot (i.e., state and current symbol read by all tapes) and then encoding in sequence the non-blank contents of all the work-tape, inserting a special "marker" symbol, to denote the locations of the heads.

Part 2 follows using similar ideas as in the proof of the Cook-Levin theorem (Theorem 2.10). There we showed that deciding whether two configurations are neighboring can be expressed as the AND of many checks, each depending on only a constant number of bits, where such checks can be expressed by constant-sized CNF formulae by Claim 2.14. ∎

Now we can prove Theorem 4.3.

PROOF OF THEOREM 4.3: Clearly $\mathbf{SPACE}(S(n)) \subseteq \mathbf{NSPACE}(S(n))$ and so we just need to show $\mathbf{NSPACE}(S(n)) \subseteq \mathbf{DTIME}(2^{O(S(n))})$. By enumerating over all possible configurations we can construct the graph $G_{M,x}$ in $2^{O(S(n))}$-time and check whether C_{start} is connected to C_{accept} in $G_{M,x}$ using the standard (linear in the size of the graph) breadth-first search algorithm for connectivity (e.g., see [**?**]). ∎

We also note that there exists a universal TM for space bounded computation analogously to Theorems 1.13 and **??** for deterministic and non-deterministic time bounded computation, see Section **??** below.

4.2 Some space complexity classes.

Now we define some complexity classes, where $\mathbf{PSPACE}, \mathbf{NPSPACE}$ are analogs of $\mathbf{P}$ and $\mathbf{NP}$ respectively.

DEFINITION 4.5
$\mathbf{PSPACE} = \cup_{c>0}\mathbf{SPACE}(n^c)$
$\mathbf{NPSPACE} = \cup_{c>0}\mathbf{NSPACE}(n^c)$
$\mathbf{L} = \mathbf{SPACE}(\log n)$
$\mathbf{NL} = \mathbf{NSPACE}(\log n)$

EXAMPLE 4.6
We show how $\mathsf{3SAT} \in \mathbf{PSPACE}$ by describing a TM that decides $\mathsf{3SAT}$ in linear space (that is, $O(n)$ space, where n is the size of the $\mathsf{3SAT}$ instance). The machine just uses the linear space to cycle through all 2^k assignments in order, where k is the number of variables. Note that once an assignment has been checked it can be erased from the worktape, and the worktape then reused to check the next assignment. A similar idea of cycling through all potential certificates applies to any $\mathbf{NP}$ language, so in fact $\mathbf{NP} \subseteq \mathbf{PSPACE}$.

EXAMPLE 4.7
The reader should check (using the gradeschool method for arithmetic) that the following languages are in $\mathbf{L}$:

$$\mathsf{EVEN} = \{x :\ x \text{ has an even number of 1s}\}.$$

$$\mathsf{MULT} = \{(\llcorner n \lrcorner,\ \llcorner m \lrcorner,\ \llcorner nm \lrcorner) : n \in \mathbb{N}\}.$$

Its seems difficult to conceive of any complicated computations apart from arithmetic that use only $O(\log n)$ space. Nevertheless, we cannot currently even rule out that $\mathsf{3SAT} \in \mathbf{L}$ (in other words —see the exercises— it is open whether $\mathbf{NP} \neq \mathbf{L}$). Space-bounded computations with space $S(n) \ll n$ seem relevant to computational problems such as *web crawling*. The world-wide-web may be viewed crudely as a directed graph, whose nodes are webpages and edges are hyperlinks. Webcrawlers seek to explore this graph for all kinds of information. The following problem PATH is natural in this context:

$$\mathsf{PATH} = \{\langle G, s, t \rangle : \ G \text{ is a directed graph in which there is a path from } s \text{ to } t\} \tag{1}$$

We claim that $\mathsf{PATH} \in \mathbf{NL}$. The reason is that a nondeterministic machine can take a "non-deterministic walk" starting at s, always maintaining the index of the vertex it is at, and using nondeterminism to select a neighbor of this vertex to go to next. The machine accepts iff the walk ends at t in at most n steps, where n is the number of nodes. If the nondeterministic walk has run for n steps already and t has not been encountered, the machine rejects. The work tape only needs to hold $O(\log n)$ bits of information at any step, namely, the number of steps that the walk has run for, and the identity of the current vertex.

Is PATH in $\mathbf{L}$ as well? This is an open problem, which, as we will shortly see, is equivalent to whether or not $\mathbf{L} = \mathbf{NL}$. That is, PATH captures the "essence" of $\mathbf{NL}$ just as $\mathsf{3SAT}$ captures the "essence" of $\mathbf{NP}$. (Formally, we will show that PATH is $\mathbf{NL}$-complete.) A recent surprising result shows that the restriction of PATH to *undirected* graphs is in $\mathbf{L}$; see Chapters 7 and 16.

4.3 PSPACE completeness

As already indicated, we do not know if $\mathbf{P} \neq \mathbf{PSPACE}$, though we strongly believe that the answer is YES. Notice, $\mathbf{P} = \mathbf{PSPACE}$ implies $\mathbf{P} = \mathbf{NP}$. Since complete problems can help capture the essence of a complexity class, we now present some complete problems for $\mathbf{PSPACE}$.

DEFINITION 4.8
A language A is $\mathbf{PSPACE}$-*hard* if for every $L \in \mathbf{PSPACE}$, $L \leq_p A$. If in addition $A \in \mathbf{PSPACE}$ then A is $\mathbf{PSPACE}$-*complete.*

Using our observations about polynomial-time reductions from Chapter **??** we see that if any $\mathbf{PSPACE}$-complete language is in $\mathbf{P}$ then so is every other language in $\mathbf{PSPACE}$. Viewed con-trapostively, if $\mathbf{PSPACE} \neq \mathbf{P}$ then a $\mathbf{PSPACE}$-complete language is not in $\mathbf{P}$. Intuitively, a $\mathbf{PSPACE}$-complete language is the "most difficult" problem of $\mathbf{PSPACE}$. Just as $\mathbf{NP}$ trivially contains $\mathbf{NP}$-complete problems, so does $\mathbf{PSPACE}$. The following is one (Exercise 3):

$$\mathsf{SPACETM} = \{\langle M, w, 1^n \rangle : \mathrm{DTM} \ M \text{ accepts } w \text{ in space } n\}. \tag{2}$$

Now we see some more interesting $\mathbf{PSPACE}$-complete problems. We use the notion of a *quantified boolean formula*, which is a boolean formula in which variables are *quantified* using $\exists$

and $\forall$ which have the usual meaning "there exists" and "for all" respectively. It is customary to also specify the universe over which these signs should be interpreted, but in our case the universe will always be the truth values $\{0, 1\}$. Thus a *quantified boolean formula* has the form $Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \varphi(x_1, x_2, \ldots, x_n)$ where each Q_i is one of the two quantifiers $\forall$ or $\exists$ and φ is an (unquantified) boolean formula[1].

If all variables in the formula are quantified (in other words, there are no free variables) then a moment's thought shows that such a formula is either *true* or *false* —there is no "middle ground". We illustrate the notion of truth by an example.

EXAMPLE 4.9
Consider the formula $\forall x \exists y \ (x \wedge y) \vee (\overline{x} \wedge \overline{y})$ where $\forall$ and $\exists$ quantify over the universe $\{0, 1\}$. Some reflection shows that this is saying "for every $x \in \{0, 1\}$ there is a $y \in \{0, 1\}$ that is different from it", which we can also informally represent as $\forall x \exists y (x \neq y)$. This formula is *true*. (Note: the symbols $=$ and $\neq$ are not logical symbols per se, but are used as informal shorthand to make the formula more readable.)

However, switching the second quantifier to $\forall$ gives $\forall x \forall y \ (x \wedge y) \vee (\overline{x} \wedge \overline{y})$, which is *false*.

EXAMPLE 4.10
Recall that the SAT problem is to decide, given a Boolean formula φ that has n free variables $x_1, \ldots, x_n$, whether or not φ has a satisfying assignment $x_1, \ldots, x_n \in \{0, 1\}^n$ such that $\varphi(x_1, \ldots, x_n)$ is true. An equivalent way to phrase this problem is to ask whether the *quantified* Boolean formula $\psi = \exists x_1, \ldots, x_n \varphi(x_1, \ldots, x_n)$ is true.

The reader should also verify that the *negation* of the formula $Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \varphi(x_1, x_2, \ldots, x_n)$ is the same as

$$Q'_1 x_1 Q'_2 x_2 \cdots Q'_n x_n \neg \varphi(x_1, x_2, \ldots, x_n),$$

where Q'_i is $\exists$ if Q_i was $\forall$ and vice versa. The switch of $\exists$ to $\forall$ in case of SAT gives instances of TAUTOLOGY, the **coNP**-complete language encountered in Chapter **??**.

We define the language TQBF to be the set of quantified boolean formulae that are true.

---

[1] We are restricting attention to quantified boolean formulae which are in *prenex normal form*, i.e., all quantifiers appear to the left. However, this is without loss of generality since we can transform a general formula into an equivalent formula in prenex form in polynomial time using identities such as $p \vee \exists x \varphi(x) = \exists x p \vee \varphi(x)$ and $\neg \forall x \phi(x) = \exists x \neg \phi(x)$. Also note that unlike in the case of the SAT and 3SAT problems, we do not require that the inner unquantified formula φ is in CNF or 3CNF form. However this choice is also not important, since using auxiliary variables in a similar way to the proof of the Cook-Levin theorem, we can in polynomial-time transform a general quantified Boolean formula to an equivalent formula where the inner unquantified formula is in 3CNF form.

THEOREM 4.11
TQBF *is* **PSPACE**-*complete.*

PROOF: First we show that TQBF $\in$ **PSPACE**. Let

$$\psi = Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \varphi(x_1, x_2, \ldots, x_n) \tag{3}$$

be a quantified Boolean formula with n variables, where we denote the size of φ by m. We show a simple recursive algorithm A that can decide the truth of ψ in $O(n+m)$ space. We will solve the slightly more general case where, in addition to variables and their negations, φ may also include the constants 0 (i.e., "false") and 1 (i.e., "true"). If $n = 0$ (there are no variables) then the formula contains only constants and can be evaluated in $O(m)$ time and space. Let $n > 0$ and let ψ be as in (3). For $b \in \{0, 1\}$, denote by $\psi_{\restriction x_1 = b}$ the modification of ψ where the first quantifier Q_1 is dropped and all occurrences of x_1 are replaced with the constant b. Algorithm A will work as follows: if $Q_1 = \exists$ then output 1 iff *at least one* of $A(\psi_{\restriction x_1 = 0})$ and $A(\psi_{\restriction x_1 = 1})$ returns 1. If $Q_1 = \forall$ then output 1 iff *both* $A(\psi_{\restriction x_1 = 0})$ and $A(\psi_{\restriction x_1 = 1})$. By the definition of $\exists$ and $\forall$, it is clear that A does indeed return the correct answer on any formula ψ.

Let $s_{n,m}$ denote the space A uses on formulas with n variables and description size m. The crucial point is —and here we use the fact that space can be *reused*—that both recursive computations $A(\psi_{\restriction x_1 = 0})$ and $A(\psi_{\restriction x_1 = 1})$ can run in the same space. Specifically, after computing $A(\psi_{\restriction x_1 = 0})$, the algorithm A needs to retain only the single bit of output from that computation, and can *reuse* the rest of the space for the computation of $A(\psi_{\restriction x_1 = 1})$. Thus, assuming that A uses $O(m)$ space to write $\psi \restriction x_1 = b$ for its recursive calls, we'll get that $s_{n,m} = s_{n-1,m} + O(m)$ yielding $s_{n,m} = O(n \cdot m)$.
[2]

We now show that $L \leq_p$ TQBF for every $L \in$ **PSPACE**. Let M be a machine that decides L in $S(n)$ space and let $x \in \{0, 1\}^n$. We show how to construct a quantified Boolean formula ψ of size $O(S(n)^2)$ that is true iff M accepts x. Recall that by Claim 4.4, there is a Boolean formula $\varphi_{M,x}$ such that for every two strings $C, C' \in \{0, 1\}^m$ (where $m = O(S(n))$ is the number of bits require to encode a configuration of M), $\varphi_M(C, C') = 1$ iff C and C' are valid encodings of two adjacent configurations in the configuration graph $G_{M,x}$. We will use $\varphi_{M,x}$ to come up with a polynomial-sized quantified Boolean formula ψ' that has polynomially many Boolean variables bound by quantifiers and additional $2m$ unquantified Boolean variables $C_1, \ldots, C_m, C'_1, \ldots, C'_m$ (or, equivalently, two variables C, C' over $\{0, 1\}^m$) such that for every $C, C' \in \{0, 1\}^m$, $\psi(C, C')$ is true iff C has a directed path to C' in $G_{M,x}$. By plugging in the values C_{start} and C_{accept} to ψ' we get a quantified Boolean formula ψ that is true iff M accepts x.

We define the formula ψ' inductively. We let $\psi_i(C, C')$ be true if and only if there is a path of length at most 2^i from C to C' in $G_{M,x}$. Note that $\psi' = \psi_m$ and $\psi_0 = \varphi_{M,x}$. The crucial observation is that there is a path of length at most 2^i from C to C' if and only if there is a configuration C''

---

[2]The above analysis already suffices to show that TQBF is in **PSPACE**. However, we can actually show that the algorithm runs in linear space, specifically, $O(m+n)$ space. Note that algorithm always works with restrictions of the same formula ψ. So it can keep a global partial assignment array that for each variable x_i will contain either $0, 1$ or 'q' (if it's quantified and not assigned any value). Algorithm A will use this global space for its operation, where in each call it will find the first quantified variable, set it to 0 and make the recursive call, then set it to 1 and make the recursive call, and then set it back to 'q'. We see that A's space usage is given by the equation $s_{n,m} = s_{n-1,m} + O(1)$ and hence it uses $O(n+m)$ space.

with such that there are paths of length at most 2^{i-1} path from C to C'' and from C'' to C'. Thus the following formula suggests itself: $\psi_i(C, C') = \exists C'' \ \psi_{i-1}(C, C') \wedge \psi_{i-1}(C'', C)$.

However, this formula is no good. It implies that ψ_i's is twice the size of ψ_{i-1}, and a simple induction shows that ψ_m has size about 2^m, which is too large. Instead, we use additional quantified variables to save on description size, using the following more succinct definition for $\psi_i(C, C')$:

$$\exists C'' \forall D^1 \forall D^2 \big((D^1 = C \wedge D^2 = C') \vee (D^1 = C' \wedge D^2 = C'') \big) \Rightarrow \psi_{i-1}(D^1, D^2)$$

(Here, as in Example 4.9, $=$ and $\Rightarrow$ are convenient shorthands, and can be replaced by appropriate combinations of the standard Boolean operations $\wedge$ and $\neg$.) Note that $size(\psi_i) \leq size(\psi_{i-1}) + O(m)$ and hence $size(\psi_m) \leq O(m^2)$. We leave it to the reader to verify that the two definitions of ψ_i are indeed logically equivalent. As noted above we can convert the final formula to prenex form in polynomial time. ■

4.3.1 Savitch's theorem.

The astute reader may notice that because the above proof uses the notion of a configuration graph and does not require this graph to have out-degree one, it actually yields a stronger statement: that TQBF is not just hard for **PSPACE** but in fact even for **NPSPACE**!. Since TQBF $\in$ **PSPACE** this implies that **PSPACE = NSPACE**, which is quite surprising since our intuition is that the corresponding classes for time (**P** and **NP**) are different. In fact, using the ideas of the above proof, one can obtain the following theorem:

THEOREM 4.12 (SAVITCH [SAV70])
For any space-constructible $S : \mathbb{N} \to \mathbb{N}$ with $S(n) \geq \log n$, **NSPACE**$(S(n)) \subseteq$ **SPACE**$(S(n)^2)$

We remark that the running time of the algorithm obtained from this theorem can be as high as $2^{\Omega(s(n)^2)}$.

PROOF: The proof closely follows the proof that TQBF is **PSPACE**-complete. Let $L \in$ **NSPACE**$(S(n))$ be a language decided by a TM M such that for every $x \in \{0,1\}^n$, the configuration graph $G = G_{M,x}$ has at most $M = 2^{O(S(n))}$ vertices. We describe a recursive procedure REACH?(u, v, i) that returns "YES" if there is a path from u to v of length at most 2^i and "NO" otherwise. Note that REACH?$(s, t, \lceil \log M \rceil)$ is "YES" iff t is reachable from s. Again, the main observation is that there is a path from u to v of length at most 2^i iff there's a vertex z with paths from u to z and from z to v of lengths at most 2^{i-1}. Thus, on input u, v, i, REACH? will enumerate over all vertices z (at a cost of $O(\log M)$ space) and output "YES" if it finds one z such that REACH?$(u, z, i-1)$="YES" and REACH?$(z, v, i-1)$="YES". Once again, the crucial observation is that although the algorithm makes n recursive invocations, it can reuse the space in each of these invocations. Thus, if we let $s_{M,i}$ be the space complexity of REACH?(u, v, i) on an M-vertex graph we get that $s_{M,i} = s_{M,i-1} + O(\log M)$ and thus $s_{M,\log M} = O(\log^2 M) = O(S(n)^2)$. ■

4.3.2 The essence of PSPACE: optimum strategies for game-playing.

Recall that the central feature of **NP**-complete problems is that a yes answer has a short certificate. The analogous unifying concept for **PSPACE**-complete problems seems to be that of a winning

strategy for a 2-player game with perfect information. A good example of such a game is Chess: two players alternately make moves, and the moves are made on a board visible to both. Thus moves have no hidden side effects; hence the term "perfect information." What does it mean for a player to have a "winning strategy?" The first player has a winning strategy iff there is a 1st move for player 1 such that for every possible 1st move of player 2 there is a 2nd move of player 1 such that.... (and so on) such that at the end player 1 wins. Thus deciding whether or not the first player has a winning strategy seems to require searching the tree of all possible moves. This is reminiscent of **NP**, for which we also seem to require exponential search. But the crucial difference is the lack of a short "certificate" for the statement "Player 1 has a winning strategy," since the only certificate we can think of is the winning strategy itself, which as noticed, requires exponentially many bits to even *describe*. Thus we seem to be dealing with a fundamentally different phenomenon than the one captured by **NP**.

The interplay of existential and universal quantifiers in the description of the the winning strategy motivates us to invent the following game.

EXAMPLE 4.13 (THE QBF GAME)
The "board" for the QBF game is a Boolean formula φ whose free variables are $x_1, x_2, \ldots, x_{2n}$. The two players alternately make moves, which involve picking values for $x_1, x_2, \ldots$, in order. Thus player 1 will pick values for the odd-numbered variables $x_1, x_3, x_5, \ldots$ (in that order) and player 2 will pick values for the even-numbered variables $x_2, x_4, x_6, \ldots,$. We say player 1 wins iff at the end φ becomes true.

Clearly, player 1 has a winning strategy iff

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \cdots \forall x_{2n} \varphi(x_1, x_2, \ldots, x_{2n}),$$

namely, iff this quantified boolean formula is true.

Thus deciding whether player 1 has a winning strategy for a given board in the QBF game is **PSPACE**-complete.

At this point, the reader is probably thinking of familiar games such as Chess, Go, Checkers etc. and wondering whether complexity theory may help differentiate between them—for example, to justify the common intuition that Go is more difficult than Chess. Unfortunately, formalizing these issues in terms of asymptotic complexity is tricky because these are finite games, and as far as the existence of a winning strategy is concerned, there are at most three choices: Player 1 has has a winning strategy, Player 2 does, or neither does (they can play to a draw). However, one can study generalizations of these games to an $n \times n$ board where n is arbitrarily large —this may involve stretching the rules of the game since the definition of chess seems tailored to an 8×8 board— and then complexity theory can indeed by applied. For most common games, including chess, determining which player has a winning strategy in the $n \times n$ version is **PSPACE**-complete (see [?]or [?]). Note that if $\mathbf{NP} \neq \mathbf{PSPACE}$ then in general there is no short certificate for exhibiting that either player in the TQBF game has a winning strategy, which is alluded to in Evens and Tarjan's quote at the start of the chapter.

Proving **PSPACE**-completeness of games may seem like a frivolous pursuit, but similar ideas lead to **PSPACE**-completeness of some practical problems. Usually, these involve repeated moves that are in turn counteracted by an adversary. For instance, many computational problems of robotics are **PSPACE**-complete: the "player" is the robot and the "adversary" is the environment. (Treating the environment as an adversary may appear unduly pessimistic; but unfortunately even assuming a benign or "indifferent" environment still leaves us with a **PSPACE**-complete problem; see the Chapter notes.)

4.4 NL completeness

Now we consider problems that form the "essence" of non-deterministic logarithmic space computation, in other words, problems that are *complete* for **NL**. What kind of reduction should we use? We cannot use the polynomial-time reduction since **NL** $\subseteq$ **P**. Thus every language in **NL** is polynomial-time reducible to the trivial language $\{1\}$ (reduction: "decide using polynomial time whether or not the input is in the **NL** language, and then map to 1 or 0 accordingly"). Intuitively, such trivial languages should not be the "hardest" languages of **NL**.

When choosing the type of reduction to define completeness for a complexity class, we must keep in mind the complexity phenomenon we seek to understand. In this case, the complexity question is whether or not **NL** = **L**. The reduction should not be more powerful than the weaker class, which is **L**. For this reason we use *logspace* reductions —for further, justification, see part (b) of Lemma 4.15 below). To define such reductions we must tackle the tricky issue that a reduction typically maps instances of size n to instances of size at least n, and so a logspace machine computing such a reduction does not have even the memory to write down its output. The way out is to require that the reduction should be able to compute any desired bit of the output in logarithmic space. In other words, if the reduction were given a separate output tape, it could in principle write out the entire new instance by first computing the first bit, then the second bit, and so on. (Many texts define such reductions using a "write-once" output tape.) The formal definition is as follows.

DEFINITION 4.14 (LOGSPACE REDUCTION)
Let $f : \{0,1\}^* \to \{0,1\}^*$ be a polynomially-bounded function (i.e., there's a constant $c > 0$ such that $f(x) \leq |x|^c$ for every $x \in \{0,1\}^*$). We say that f is *implicitly logspace computable*, if the languages $L_f = \{\langle x, i \rangle \mid f(x)_i = 1\}$ and $L'_f = \{\langle x, i \rangle \mid i \leq |f(x)|\}$ are in **L**.

Informally, we can think of a *single* $O(\log |x|)$-space machine that given input (x, i) outputs $f(x)|_i$ provided $i \leq |f(x)|$.

Language A is *logspace reducible* to language B, denoted $A \leq_l B$, if there is a function $f : \{0,1\}^* \to \{0,1\}^*$ that is implicitly logspace computable and $x \in A$ iff $f(x) \in B$ for every $x \in \{0,1\}^*$.

Logspace reducibility satisfies usual properties one expects.

LEMMA 4.15
(a) If $A \leq_l B$ and $B \leq_l C$ then $A \leq_l C$. (b) If $A \leq_l B$ and $B \in$ **L** then $A \in$ **L**.

PROOF: We prove that if f, g are two functions that are logspace implicitly computable, then so is the function h where $h(x) = g(f(x))$. Then part (a) of the Lemma follows by letting f be the

reduction from A to B and g be the reduction from B to C. Part (b) follows by letting f be the reduction from A to B and g be the characteristic function of B (i.e. $g(y) = 1$ iff $y \in B$).

So let M_f, M_g be the logspace machines that compute the mappings $x, i \mapsto f(x)_i$ and $y, j \mapsto g(y)_j$ respectively. We construct a machine M_h that computes the mapping $x, j \mapsto g(f(x))_j$, in other words, given input x, j outputs $g(f(x))_j$ provided $j \leq |g(f(x))|$. Machine M_h will pretend that it has an additional (fictitious) input tape on which $f(x)$ is written, and it is merely simulating M_g on this input (see Figure 4.3). Of course, the true input tape has x, j written on it. To maintain its fiction, M_h always maintains on its worktape the index, say i, of the cell on the fictitious tape that M_g is currently reading; this requires only $\log |f(x)|$ space. To compute for one step, M_g needs to know the contents of this cell, in other words, $f(x)|_i$. At this point M_h temporarily suspends its simulation of M_g (copying the contents of M_g's worktape to a safe place on its own worktape) and invokes M_f on inputs x, i to get $f(x)|_i$. Then it resumes its simulation of M_g using this bit. The total space M_h uses is $O(\log |g(f(x))| + s(|x|) + s'(|f(x)|)) = O(\log |x|)$. ∎
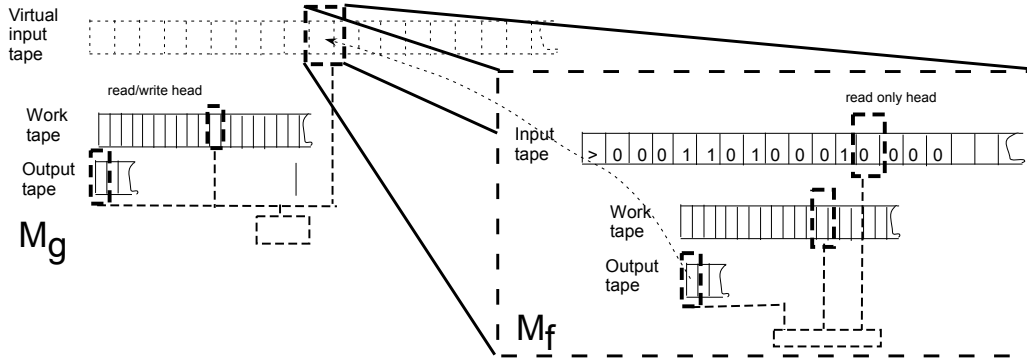
Figure 4.3: Composition of two implicitly logspace computable functions f, g. The machine M_g uses calls to f to implement a "virtual input tape". The overall space used is the space of M_f + the space of M_g + $O(\log |f(x)|) = O(log|x|)$.

We say that A is **NL**-*complete* if it is in **NL** and for every $B \in$ **NL**, $A \leq_l B$. Note that an **NL**-complete language is in **L** iff **NL** =**L**.

THEOREM 4.16
PATH *is* **NL**-*complete*.

PROOF: We have already seen that PATH is in **NL**. Let L be any language in **NL** and M be a machine that decides it in space $O(\log n)$. We describe a logspace implicitly computable function f that reduces L to PATH. For any input x of size n, $f(x)$ will be the configuration graph $G_{M,x}$ whose nodes are all possible $2^{O(\log n)}$ configurations of the machine on input x, along with the start configuration C_{start} and the accepting configuration C_{acc}. In this graph there is a path from C_{start} to C_{acc} iff M accepts x. The graph is represented as usual by an *adjacency matrix* that contain 1 in the $\langle C, C' \rangle^{th}$ position (i.e., in the Cth row and C'^{th} column if we identify the configurations with numbers between 0 and $2^{O(\log n)}$) iff there's an edge C from C' in $G_{M,x}$. To finish the proof we need to show that this adjacency matrix can be computed by a logspace reduction. This is easy

since given a $\langle C, C' \rangle$ a deterministic machine can in space $O(|C| + |C'|) = O(\log |x|)$ examine C, C' and check whether C' is one of the (at most two) configurations that can follow C according to the transition function of M. ∎

4.4.1 Certificate definition of NL: read-once certificates

In Chapter 2 we gave two equivalent definitions of **NP**— one using non-deterministic TM's and another using the notion of a *certificate*. The idea was that the nondeterministic choices of the NDTM that lead it to accept can be viewed as a "certificate" that the input is in the language, and vice versa. We can give a certificate-based definition also for **NL**, but only after addressing one tricky issue: a certificate may be polynomially long, and a logspace machine does not have the space to store it. Thus, the certificate-based definition of **NL** assumes that the logspace machine on a separate read-only tape. Furthermore, on each step of the machine the machine's head on that tape can either stay in place or move to the right. In particular, it cannot reread any bit to the left of where the head currently is. (For this reason the this kind of tape is called "*read once*".) It is easily seen that the following is an alternative definition of **NL** (see also Figure 4.4):
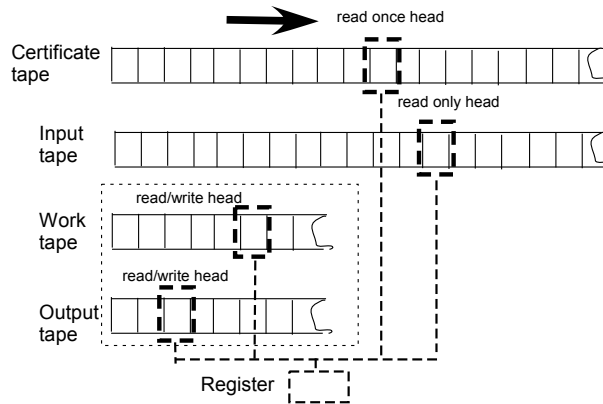
Figure 4.4: Certificate view of **NL**. The certificate for input x is placed on a special "read-once" tape on which the machine's head can never move to the left.

DEFINITION 4.17 (**NL**- ALTERNATIVE DEFINITION.)
A language L is in **NL** if there exists a deterministic TM M and a with an additional special read-once input tape polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$,

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

where by $M(x, u)$ we denote the output of M where x is placed on its input tape and u is placed on its special read-once tape, and M uses at most $O(\log |x|)$ space on its read/write tapes for every input x.

4.4.2   **NL = coNL**

Consider the problem $\overline{\mathsf{PATH}}$, i.e., the complement of PATH. A decision procedure for this language must accept when there is no path from s to t in the graph. Unlike in the case of PATH, there is no natural certificate for the *non-existence* of a path from s to t and thus it seemed "obvious" to researchers that $\overline{\mathsf{PATH}} \notin \mathbf{NL}$, until the discovery of the following theorem in the 1980s proved them wrong.

THEOREM 4.18 (IMMERMAN-SZLEPCSENYI)
$\overline{\mathsf{PATH}} \in \mathbf{NL}$.

PROOF: As we saw in Section 4.4.1, we need to show an $O(\log n)$-space algorithm A such that for every n-vertex graph G and vertices s and t, there exists a polynomial certificate u such that $A(\langle G, s, t \rangle, u) = 1$ if and only if t is not reachable from u in G, where A has only read-once access to u.

What can we certify to an $O(\log n)$-space algorithm? Let C_i be the set of vertices that are reachable from s in G within at most i steps. For every $i \in [n]$ and vertex v, we can easily certify that v is in C_i. The certificate simply contains the labels $v_0, v_1, \ldots, v_k$ of the vertices along the path from s to v (we can assume without loss of generality that vertices are labeled by the numbers 1 to n and hence the labels can be described by $\log n$ bit strings). The algorithm can check the certificate using read-once access by verifying that **(1)** $v_0 = s$, **(2)** for $j > 0$, there is an edge from v_{j-1} to v_j, **(3)** $v_k = v$ and (using a counter) that **(4)** the path ends within at most i steps. Note that the certificate is indeed of size at most polynomial in n.

Our algorithm uses the following two procedures:

1. Procedure to certify that a vertex v is not in C_i given the size of C_i.

2. Procedure to certify that $|C_i| = c$ for some number c, given the size of C_{i-1}.

Since $C_0 = \{s\}$ and C_n contains all the vertices reachable from s, we can apply the second procedure iteratively to learn the sizes of the sets $C_1, \ldots, C_n$, and then use the first procedure to certify that $t \notin C_n$.

Certifying that v is not in C_i, given $|C_i|$. The certificate is simply the list of certificates that u is in C_i for every $u \in C_i$ sorted in ascending order of labels (recall that we identify labels with numbers in $[n]$). The algorithm checks that **(1)** each certificate is valid, **(2)** the label of a vertex u for which a certificate is given is indeed larger than the label of the previous vertex, **(3)** no certificate is provided for v, and **(4)** the total number of certificates provided is exactly $|C_i|$. If $v \notin C_i$ then the algorithm will accept the above certificate, but if $v \in C_i$ there will not exist $|C_i|$ certificates that vertices $u_1 < u_2 < \ldots < u_{|C_i|}$ are in C_i where $u_j \neq v$ for every j.

Certifying that v is not in C_i, given $|C_{i-1}|$. Before showing how we certify that $|C_i| = c$ given $|C_{i-1}|$, we show how to certify that $v \notin C_i$ with this information. This is very similar to the above procedure: the certificate is the list of $|C_{i-1}|$ certificates that $u \in C_{i-1}$ for every $u \in C_{i-1}$ in ascending order. The algorithm checks everything as before except that in step **(3)** it verifies that no certificate is given for v *or for a neighbor of v*. Since $v \in C_i$ if and only if there exists $u \in C_{i-1}$ such that $u = v$ or u is a neighbor of v in G, the procedure will not accept a false certificate by the same reasons as above.

Certifying that $|C_i| = c$ **given** $|C_{i-1}|$.   For every vertex v, if $v \in C_i$ then there is a certificate for this fact, and by the above procedure, given $|C_{i-1}|$, if $v \notin C_i$ then there is a certificate for this fact as well. The certificate that $|C_i| = c$ will consist of n certificates for each of the vertices 1 to n in ascending order. For every vertex u, there will be an appropriate certificate depending on whether $u \in C_i$ or not. The algorithm will verify all the certificate and count the number of certificate that a vertex is in C_i. It accepts if this count is equal to c. $\blacksquare$

Using the notion of the configuration graph we can modify the proof of Theorem 4.18 to prove the following:

COROLLARY 4.19
For every space constructible $S(n) > \log n$, **NSPACE**$(S(n)) = $ **coNSPACE**$(S(n))$.

Our understanding of space-bounded complexity.
The following is our understanding of space-bounded complexity.

$$\textbf{DTIME}(s(n)) \subseteq \textbf{SPACE}(s(n)) \subseteq \textbf{NSPACE}(s(n)) = \textbf{coNSPACE}(s(n)) \subseteq \textbf{DTIME}(2^{O(s(n))}).$$

None of the inclusions are known to be strict though we believe they all are.

Chapter notes and history

The concept of space complexity had already been explored in the 1960s; in particular, Savitch's theorem predates the Cook-Levin theorem. Stockmeyer and Meyer proved the **PSPACE**-completeness of TQBF soon after Cook's paper appeared. A few years later Even and Tarjan pointed out the connection to game-playing and proved the **PSPACE**-completeness of a game called Generalized Hex. Papadimitriou's book gives a detailed account of **PSPACE**-completeness. He also shows **PSPACE**-completeness of several *Games against nature* first defined in [Pap85]. Unlike the TQBF game, where one player is *Existential* and the other *Universal*, here the second player chooses moves randomly. The intention is to model games played against nature—where "nature" could mean not just weather for example, but also large systems such as the stock market that are presumably "indifferent" to the fate of individuals. Papadimitriou gives an alternative characterization **PSPACE** using such games. A stronger result, namely, a characterization of **PSPACE** using interactive proofs, is described in Chapter 8.

Exercises

§1 Show that **SPACE**$(S(n)) = $ **SPACE**(0) when $S(n) = \log \log n$.

§2 Prove the existence of a universal TM for space bounded computation (analogously to the deterministic universal TM of Theorem 1.13). That is, prove that there exists a a TM $\mathcal{SU}$ such that for every string α, and input x, if the TM M_α represented by α halts on x before using t cells of its work tapes then $\mathcal{SU}(\alpha, t, x) = M_\alpha(x)$, and moreover, $\mathcal{SU}$ uses at most Ct cells of its work tapes, where C is a constant depending only on M_α. (Despite the fact

that the bound here is better than the bound of Theorem 1.13, the proof of this statement is actually easier than the proof of Theorem 1.13.)

§3 Prove that the language SPACETM of (2) is **PSPACE**-complete.

§4 Show that the following language is **NL**-complete:

$$\{ \llcorner G \lrcorner : G \text{ is a strongly connected digraph}\} .$$

§5 Show that 2SAT is in **NL**.

§6 Suppose we define **NP**-completeness using logspace reductions instead of polynomial-time reductions. Show (using the proof of the Cook-Levin Theorem) that SAT and 3SAT continue to be **NP**-complete under this new definition. Conclude that SAT $\in$ **L** iff **NP** = **L**.

§7 Show that TQBF is complete for **PSPACE** also under logspace reductions.

§8 Show that in every finite 2-person game with perfect information (by *finite* we mean that there is an *a priori* upperbound n on the number of moves after which the game is over and one of the two players is declared the victor —there are no draws) one of the two players has a winning strategy.

§9 Define **polyL** to be $\cup_{c>0}$**SPACE**$(\log^c n)$. Steve's Class SC (named in honor of Steve Cook) is defined to be the set of languages that can be decided by deterministic machines that run in polynomial time and $\log^c n$ space for some $c > 0$.

It is an open problem whether PATH $\in$ SC. Why does Savitch's Theorem not resolve this question?

Is SC the same as **polyL** $\cap$ **P**?

DRAFT

Chapter 5

The Polynomial Hierarchy and Alternations

> *"..synthesizing circuits is exceedingly difficulty. It is even more difficult to show that a circuit found in this way is the* most *economical one to realize a function. The difficulty springs from the large number of essentially different networks available."*
> Claude Shannon 1949

This chapter discusses the *polynomial hierarchy*, a generalization of **P**, **NP** and **coNP** that tends to crop up in many complexity theoretic investigations (including several chapters of this book). We will provide three equivalent definitions for the polynomial hierarchy, using quantified predicates, alternating Turing machines, and oracle TMs (a fourth definition, using uniform families of circuits, will be given in Chapter 6). We also use the hierarchy to show that solving the SAT problem requires either linear space or super-linear time.

5.1 The classes Σ_2^p and Π_2^p

To understand the need for going beyond nondeterminism, let's recall an **NP** problem, INDSET, for which we *do* have a short certificate of membership:

$$\text{INDSET} = \{\langle G, k \rangle : \text{ graph } G \text{ has an independent set of size } \geq k\}.$$

Consider a slight modification to the above problem, namely, determining the largest independent set in a graph (phrased as a decision problem):

$$\text{EXACT INDSET} = \{\langle G, k \rangle : \text{ the largest independent set in } G \text{ has size exactly } k\}.$$

Now there seems to be no short certificate for membership: $\langle G, k \rangle \in$ EXACT INDSET iff *there exists* an independent set of size k in G and *every other* independent set has size at most k.

Similarly, consider the language MIN-DNF, the decision version of a problem in circuit minimization, a topic of interest in electrical engineering (and referred to in Shannon's paper). We say

that two boolean formulae are *equivalent* if they have the same set of satisfying assignments.

$$\mathsf{MIN-DNF} = \left\{ \ \llcorner\varphi\lrcorner : \ \varphi \text{ is a DNF formula not equivalent to any smaller DNF formula} \right\}.$$
$$= \left\{ \ \llcorner\varphi\lrcorner : \ \forall\psi, |\psi| < |\varphi| , \exists \text{ assignment } s \text{ such that } \varphi(s) \neq \psi(s) \right\}.$$

Again, there is no obvious notion of a certificate of membership. Note that both the above problems are in **PSPACE**, but neither is believed to be **PSPACE**-complete.

It seems that the way to capture such languages is to allow not only an "exists" quantifier (as in Definition 2.1 of **NP**) or only a "for all" quantifier (as Definition 2.22 of **coNP**) but a combination of both quantifiers. This motivates the following definition:

DEFINITION 5.1
The class $\mathbf{\Sigma}_2^p$ is defined to be the set of all languages L for which there exists a polynomial-time TM M and a polynomial q such that

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{q(|x|)} \forall v \in \{0,1\}^{q(|x|)} M(x,u,v) = 1$$

for every $x \in \{0,1\}^*$.

Note that $\mathbf{\Sigma}_2^p$ contains both the classes **NP** and **coNP**.

EXAMPLE 5.2
The language EXACT INDSET above is in $\mathbf{\Sigma}_2^p$, since as we noted above, a pair $\langle G, k \rangle$ is in EXACT INDSET iff

$\exists\ S\ \forall S'$ set S is an independent set of size k in G and

S' is not a independent set of size $\geq k+1$.

We define the class $\mathbf{\Pi}_2^p$ to be the set $\left\{ \overline{L} : L \in sig_2^p \right\}$. It is easy to see that an equivalent definition is that $L \in \mathbf{\Pi}_2^p$ if there is a polynomial-time TM M and a polynomial q such that

$$x \in L \Leftrightarrow \forall u \in \{0,1\}^{q(|x|)} \exists v \in \{0,1\}^{q(|x|)} M(x,u,v) = 1$$

for every $x \in \{0,1\}^*$.

EXAMPLE 5.3
The language EXACT INDSET is also in $\mathbf{\Pi}_2^p$ since a pair $\langle G, k \rangle$ is in EXACT INDSET if *for every S'*, if S' has size at least $k+1$ then it is not an independent set, but *there exists* an independent set S of size k in G. (Exercise 8 shows a finer placement of EXACT INDSET.)

The reader can similarly check that $\mathsf{MIN-DNF}$ is in $\mathbf{\Pi}_2^p$. It is conjectured to be complete for $\mathbf{\Pi}_2^p$.

DRAFT

5.2 The polynomial hierarchy.

The polynomial hierarchy generalizes the definitions of $\mathbf{NP}, \mathbf{coNP}, \mathbf{\Sigma}_2^p, \mathbf{\Pi}_2^p$ to consists all the languages that can be defined via a combination of a polynomial-time computable predicate and a constant number of $\forall/\exists$ quantifiers:

DEFINITION 5.4 (POLYNOMIAL HIERARCHY)
For every $i \geq 1$, a language L is in $\mathbf{\Sigma}_i^p$ if there exists a polynomial-time TM M and a polynomial q such that

$$x \in L \Leftrightarrow \exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \cdots Q_i u_i \in \{0,1\}^{q(|x|)} M(x, u_1, \ldots, u_i) = 1,$$

where Q_i denotes $\forall$ or $\exists$ depending on whether i is even or odd respectively.

We say that L is in $\mathbf{\Pi}_i^p$ if there exists a polynomial-time TM M and a polynomial q such that

$$x \in L \Leftrightarrow \forall u_1 \in \{0,1\}^{q(|x|)} \exists u_2 \in \{0,1\}^{q(|x|)} \cdots Q_i u_i \in \{0,1\}^{q(|x|)} M(x, u_1, \ldots, u_i) = 1,$$

where Q_i denotes $\exists$ or $\forall$ depending on whether i is even or odd respectively.

The *polynomial hierarchy* is the set $\mathbf{PH} = \cup_i \mathbf{\Sigma}_i^p$.

REMARK 5.5
Note that $\mathbf{\Sigma}_1^p = \mathbf{NP}$ and $\mathbf{\Pi}_2^p = \mathbf{coNP}$. More generally, for evert $i \geq 1$, $\mathbf{\Pi}_i^p = \mathbf{co\Sigma}_i^p = \left\{ \overline{L} : L \in \mathbf{\Sigma}_i^p \right\}$. Note also that that $\mathbf{\Sigma}_i^p \subseteq \mathbf{\Pi}_{i+1}^p$, and so we can also define the polynomial hierarchy as $\cup_{i>0} \mathbf{\Pi}_i^p$.

5.2.1 Properties of the polynomial hierarchy.

We believe that $\mathbf{P} \neq \mathbf{NP}$ and $\mathbf{NP} \neq \mathbf{coNP}$. An appealing generalization of these conjectures is that for every i, $\mathbf{\Sigma}_i^p$ is strictly contained in $\mathbf{\Sigma}_{i+1}^p$. This is called the conjecture that the polynomial hierarchy does not collapse, and is used often in complexity theory. If the polynomial hierarchy does collapse this means that there is some i such that $\mathbf{\Sigma}_i^p = \cup_j \mathbf{\Sigma}_j^p = \mathbf{PH}$. In this case we say that the polynomial hierarchy has collapsed to the ith level. The smaller i is, the weaker, and hence more plausible, is the conjecture that $\mathbf{PH}$ does not collapse to the ith level.

THEOREM 5.6
    1. For every $i \geq 1$, if $\mathbf{\Sigma}_i^p = \mathbf{\Pi}_i^p$ then $\mathbf{PH} = \mathbf{\Sigma}_i^p$ (i.e., the hierarchy collapses to the ith level).

    2. If $\mathbf{P} = \mathbf{NP}$ then $\mathbf{PH} = \mathbf{P}$ (i.e., the hierarchy collapses to $\mathbf{P}$).

PROOF: We do the second part; the first part is similar and also easy.

    Assuming $\mathbf{P} = \mathbf{NP}$ we prove by induction on i that $\mathbf{\Sigma}_i^p, \mathbf{\Pi}_i^p \subseteq \mathbf{P}$. Clearly this is true for $i = 1$ since under our assumption $\mathbf{P} = \mathbf{NP} = \mathbf{coNP}$. We assume it is true for $i-1$ and prove it for i. Let

$L \in \mathbf{\Sigma}_i^p$, we will show that $L \in \mathbf{P}$. By definition, there is a polynomial-time M and a polynomial q such that

$$x \in L \Leftrightarrow \exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \cdots Q_i u_i \in \{0,1\}^{q(|x|)} \, M(x, u_1, \ldots, u_i) = 1,$$

where Q_i is $\exists/\forall$ as in Definition 5.4. Define the language L' as follows:

$$u \in L' \Leftrightarrow \exists \forall u_2 \in \{0,1\}^{q(|x|)} \cdots Q_i u_i \in \{0,1\}^{q(|x|)} \, M(u_1, u_2, \ldots, u_i) = 1.$$

Clearly, $L' \in \mathbf{\Pi}_{i-1}^p$ and so under our assumption is in $\mathbf{P}$. This implies that there is a TM M' such that

$$x \in L \Leftrightarrow \exists u_1 \in \{0,1\}^{q(|x|)} \, M'(x, u_1) = 1 \,.$$

But this means $L \in \mathbf{NP}$ and hence under our assumption $L \in \mathbf{P}$. The same idea shows that if $L \in \mathbf{\Pi}_i^p$ then $L \in \mathbf{P}$. $\blacksquare$

5.2.2 Complete problems for levels of PH

For every i, we say that a language L is $\mathbf{\Sigma}_i^p$-complete if $L \in \mathbf{\Sigma}_i^p$ and for every $L' \in \mathbf{\Sigma}_i^p$, $L' \leq_p L$. We define $\mathbf{\Pi}_i^p$-completeness and $\mathbf{PH}$-completeness in the same way. In this section we show that for every $i \in \mathbb{N}$, both $\mathbf{\Sigma}_i^p$ and $\mathbf{\Pi}_i^p$ have complete problems. In contrast the polynomial hierarchy itself is believed not to have a complete problem, as is shown by the following simple claim:

CLAIM 5.7
Suppose that there exists a language L that is $\mathbf{PH}$-complete, then there exists an i such that $\mathbf{PH} = \mathbf{\Sigma}_i^p$ (and hence the hierarchy collapses to its ith level.)

PROOF SKETCH: Since $L \in \mathbf{PH} = \cup_i \mathbf{\Sigma}_i^p$, there exists i such that $L \in \mathbf{\Sigma}_i^p$. Since L is $\mathbf{PH}$-complete, we can reduce every language of $\mathbf{PH}$ to $\mathbf{\Sigma}_i^p$ to L, and thus $\mathbf{PH} \subseteq \mathbf{\Sigma}_i^p$. $\blacksquare$

REMARK 5.8
It is not hard to see that $\mathbf{PH} \subseteq \mathbf{PSPACE}$. A simple corollary of Claim 5.7 is that unless the polynomial hierarchy collapses, $\mathbf{PH} \neq \mathbf{PSPACE}$. Indeed, otherwise the problem TQBF would be $\mathbf{PH}$-complete.

EXAMPLE 5.9
The following are some examples for complete problems for individual levels of the hierarchy:

For every $i \geq 1$, the class $\mathbf{\Sigma}_i^p$ has the following complete problem involving quantified boolean expression with limited number of alternations:

$$\Sigma_i \mathsf{SAT} = \exists u_1 \forall u_2 \exists \cdots Q_i u_i \; \varphi(u_1, u_2, \ldots, u_i) = 1, \tag{1}$$

where φ is a Boolean formula (not necessarily in CNF form, although this does not make much difference), each u_i is a vector of boolean variables, and Q_i is $\forall$ *or* $\exists$ depending on whether i is odd or even. Notice that this is a special case of the TQBF problem defined in Chapter 4. Exercise 1

asks you to prove that $\Sigma_i\mathsf{SAT}$ is indeed $\mathbf{\Sigma}_i^p$-complete. One can similarly define a problem $\Pi_i\mathsf{SAT}$ that is $\mathbf{\Pi}_i^p$-complete.

In the SUCCINCT SET COVER problem we are given a collection $S = \{\varphi_1, \varphi_2, \ldots, \varphi_m\}$ of 3-DNF formulae on n variables, and an integer k. We need to determine whether there is a subset $S' \subseteq \{1, 2, \ldots, m\}$ of size at most K for which $\vee_{i \in S'}\varphi_i$ is a tautology (i.e., evaluates to 1 for every assignment to the variables). Umans showed that SUCCINCT SET COVER is $\mathbf{\Sigma}_2^p$-complete [Uma01].

5.3 Alternating Turing machines

Alternating Turing Machines (ATM), are generalizations of nondeterministic Turing machines. Recall that even though NDTMs are not a realistic computational model, studying them helps us to focus on a natural computational phenomenon, namely, the apparent difference between *guessing* an answer and *verifying* it. ATMs plays a similar role for certain languages for which there is no obvious short *certificate* for membership and hence cannot be characterized using nondeterminism alone.

Alternating TMs are similar to NDTMs in the sense that they have *two* transition functions between which they can choose in each step, but they also have the additional feature that every internal state except q_{accept} and q_{halt} is labeled with either $\exists$ or $\forall$. Similar to the NDTM, an ATM can evolve at every step in two possible ways. Recall that a non-deterministic TM accepts its input if there *exists* some sequence of choices that leads it to the state q_{accept}. In an ATM, the *exists* quantifier over each choice is replaced with the appropriate quantifier according to the labels.

DEFINITION 5.10
Let M be an alternating TM. For a function $T : \mathbb{N} \to \mathbb{N}$, we say that M is an $T(n)$-time ATM if for every input $x \in \{0, 1\}^*$ and for every possible sequence of transition function choices, M will halt after at most $T(|x|)$ steps.

For every $x \in \{0, 1\}^*$, we let $G_{M,x}$ be the configuration graph of x, whose vertices are the configurations of M on input x and there is an edge from configuration C to C' if there C' can be obtained from C in one step using one of the two transition functions (see Section 4.1). Recall that this is a directed acyclic graph. We label some of the nodes in the graph by "ACCEPT" by repeatedly applying the following rules until they cannot be applied anymore:

- The configuration C_{accept} where the machine is in q_{accept} is labeled "ACCEPT".

- If a configuration C is in a state labeled $\exists$ and one of the configurations C' reachable from it in one step is labeled "ACCEPT" then we label C "ACCEPT".

- If a configuration C is in a state labeled $\forall$ and both the configurations C', C'' reachable from it one step is labeled "ACCEPT" then we label C "ACCEPT".

We say that M *accepts* x if at the end of this process the starting configuration C_{start} is labeled "ACCEPT". The *language accepted by* M is the set of all x's such that M accepts x. We denote by $\mathbf{ATIME}(T(n))$ the set of all languages accepted by some $T(n)$-time ATM.

For every $i \in \mathbb{N}$, we define $\mathbf{\Sigma}_i \mathbf{TIME}(T(n))$ (resp. $\mathbf{\Pi}_i \mathbf{TIME}(T(n))$) to be the set of languages accepted by a $T(n)$-time ATM M whose initial state is labeled "$\exists$" (resp. "$\forall$") and on which every input and sequence of choices leads M to change at most $i - 1$ times from states with one label to states with the other label.

The following claim is left as an easy exercise (see Exercise 2):

CLAIM 5.11
For every $i \in \mathbb{N}$,

$$\mathbf{\Sigma}_i^p = \cup_c \mathbf{\Sigma}_i \mathbf{TIME}(n^c)$$
$$\mathbf{\Pi}_i^p = \cup_c \mathbf{\Pi}_i \mathbf{TIME}(n^c)$$

5.3.1 Unlimited number of alternations?

What if we consider polynomial-time alternating Turing machines with no *a priori* bound on the number of quantifiers? We define the class $\mathbf{AP}$ to be $\cup_c \mathbf{ATIME}(n^c)$. We have the following theorem:

THEOREM 5.12
$\mathbf{AP} = \mathbf{PSPACE}$.

PROOF: $\mathbf{PSPACE} \subseteq \mathbf{AP}$ follows since TQBF is trivially in $\mathbf{AP}$ (just "guess" values for each existentially quantified variable using an $\exists$ state and for universally quantified variables using a $\forall$ state) and every $\mathbf{PSPACE}$ language reduces to TQBF.

$\mathbf{AP} \subseteq \mathbf{PSPACE}$ follows using a recursive procedure similar to the one used to show that TQBF $\in \mathbf{PSPACE}$. ■

Similarly, one can consider alternating Turing machines that run in polynomial space. The class of languages accepted by such machines is called $\mathbf{APSPACE}$, and Exercise 6 asks you to prove that $\mathbf{APSPACE} = \mathbf{EXP}$. One can similarly consider alternating logspace machines; the set of languages accepted by them is exactly $\mathbf{P}$.

5.4 Time versus alternations: time-space tradeoffs for SAT.

Despite the fact that SAT is widely believed to require exponential (or at least super-polynomial) time to solve, and to require linear (or at least super-logarithmic) space, we currently have no way to prove these conjectures. In fact, as far as we know, SAT may have both a linear time algorithm and a logarithmic space one. Nevertheless, we *can* prove that SAT does not have an algorithm that runs *simultaneously* in linear time and logarithmic space. In fact, we can prove the following stronger theorem:

THEOREM 5.13 (??)
For every two functions $S, T : \mathbb{N} \to \mathbb{N}$, define $\mathbf{TISP}(T(n), S(n))$ to be the set of languages decided by a TM M that on every input x takes at most $O(T(|x|))$ steps and uses at most $O(S(|x|))$ cells of its read/write tapes. Then, SAT $\notin \mathbf{TISP}(n^{1.1}, n^{0.1})$.

REMARK 5.14
The class $\mathbf{TISP}(T(n), S(n))$ is typically defined with respect to TM's with RAM memory (i.e., TM's that have random access to their tapes; such machines can be defined in a similar way to the definition of oracle TM's in Section 3.5). Theorem 5.13 and its proof carries over for that model as well. We also note that a stronger result is known for both models: for every $c < (\sqrt{5} + 1)/2$, there exists $d > 0$ such that $\mathsf{SAT} \notin \mathbf{TISP}(n^c, n^d)$ and furthermore, d approaches 1 from below as c approaches 1 from above.

PROOF: We will show that

$$\mathbf{NTIME}(n) \not\subseteq \mathbf{TISP}(n^{1.2}, n^{0.2}). \qquad (2)$$

This implies the result for SAT by following the ideas of the proof of the Cook-Levin Theorem (Theorem 2.10). A careful analysis of that proof yields a reduction from the task of deciding membership in an $\mathbf{NTIME}(T(n))$-language to the task deciding whether an $O(T(n) \log T(n))$-sized formula is satisfiable, such that every output bit of this reduction can be computed in polylogarithmic time and space. (See also the proof of Theorem 6.7 for a similar analysis.)   Hence, if $\mathsf{SAT} \in \mathbf{TISP}(n^{1.1}, n^{0.1})$ then $\mathbf{NTIME}(n) \subseteq \mathbf{TISP}(n^{1.1}\text{polylog}(n), n^{0.1}\text{polylog}(n))$. Our main step in proving (2) is the following claim, showing how to replace time with alternations:

CLAIM 5.14.1
$\mathbf{TISP}(n^{12}, n^2) \subseteq \mathbf{\Sigma_2 TIME}(n^8)$.

PROOF: The proof is similar to the proofs of Savitch's Theorem and the $\mathbf{PSPACE}$-completeness of TQBF (Theorems 4.12 and 4.11). Suppose that L is decided by a machine M using n^{12} time and n^2 space. For every $x \in \{0,1\}^*$, consider the configuration graph $G_{M,x}$ of M on input x. Each configuration in this graph can be described by a string of length $O(n^2)$ and x is in L if and only if there is a path of length n^{12} in this graph from the starting configuration C_{start} to an accepting configuration. There is such a path if and only if there *exist n^6 configurations* $C_1, \ldots, C_{n^6}$ (requiring $O(n^8)$ to specify) such that if we let $C_0 = C_{\mathsf{start}}$ then C_{n^6} is accepting and *for every $i \in [n^6]$ the* configuration C_i is computed from C_{i-1} within n^6 steps. Because this condition can be verified in n^6 time, we can we get an $O(n^8)$-time σ_2-TM for deciding membership in L. ■

Our next step will show that, under the assumption that (2) does not hold (and hence $\mathbf{NTIME}(n) \subseteq \mathbf{TISP}(n^{1.2}, n^{0.2})$), we can replace alternations with time:

CLAIM 5.14.2
Suppose that $\mathbf{NTIME}(n) \subseteq \mathbf{DTIME}(n^{1.2})$. *Then* $\mathbf{\Sigma_2 TIME}(n^8) \subseteq \mathbf{NTIME}(n^{9.6})$.

PROOF: Using the characterization of the polynomial hierarchy by alternating machines, L is in $\mathbf{\Sigma_2 TIME}(n^8)$ if and only if there is an $O(n^8)$-time TM M such that

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{c|x|^8} \, \forall v \in \{0,1\}^{d|x|^8} \, M(x, u, v) = 1 \,.$$

for some constants c, d. Yet if $\mathbf{NTIME}(n) \subseteq \mathbf{DTIME}(n^{1.2})$ then by a simple padding argument (a la the proof of Theorem 2.25) we have a deterministic algorithm D that on inputs x, u with $|x| = n$

and $|u| = cn^8$ runs in time $O((n^8)^{1.2}) = O(n^{9.6})$-time and returns 1 if and only if there exists some $v \in \{0,1\}^{dn^8}$ such that $M(x, u, v) = 0$. Thus,

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{c|x|^8} D(x, u) = 0.$$

implying that $L \in \mathbf{NTIME}(n^{9.6})$. ∎

Claims 5.14.1 and 5.14.2 show that the assumption that $\mathbf{NTIME}(n) \subseteq \mathbf{TISP}(n^{1.2}, n^{0.2})$ leads to contradiction: by simple padding it implies that $\mathbf{NTIME}(n^{10}) \subseteq \mathbf{TISP}(n^{12}, n^2)$ which by Claim 5.14.1 implies that $\mathbf{NTIME}(n^{10}) \subseteq \boldsymbol{\Sigma}_2\mathbf{TIME}(n^8)$. But together with Claim 5.14.2 this implies that $\mathbf{NTIME}(n^{10}) \subseteq \mathbf{NTIME}(n^{9.6})$, contradicting the non-deterministic time hierarchy theorem (Theorem 3.3). ∎

5.5   Defining the hierarchy via oracle machines.

Recall the definition of *oracle machines* from Section 3.5. These are machines that are executed with access to a special tape they can use to make queries of the form "is $q \in O$" for some language O. For every $O \subseteq \{0,1\}^*$, oracle TM M and input x, we denote by $M^O(x)$ the output of M on x with access to O as an oracle. We have the following characterization of the polynomial hierarchy:

THEOREM 5.15
For every $i \geq 2$, $\boldsymbol{\Sigma}_i^p = \mathbf{NP}^{\boldsymbol{\Sigma}_{i-1}\mathsf{SAT}}$, where the latter class denotes the set of languages decided by polynomial-time NDTMs with access to the oracle $\boldsymbol{\Sigma}_{i-1}\mathsf{SAT}$.

PROOF: We showcase the idea by proving that $\boldsymbol{\Sigma}_2^p = \mathbf{NP}^{\mathsf{SAT}}$. Suppose that $L \in \boldsymbol{\Sigma}_2^p$. Then, there is a polynomial-time TM M and a polynomial q such that

$$x \in L \Leftrightarrow \exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} M(x, u_1, u_2) = 1$$

yet for every fixed u_1 and x, the statement "for every u_2, $M(x, u_1, u_2) = 1$" is the negation of an **NP**-statement and hence its truth can be determined using an oracle for SAT. We get that there is a simple NDTM N that given oracle access for SAT can decide L: on input x, non-deterministically guess u_1 and use the oracle to decide if $\forall u_2 M(x, u_1, u_2) = 1$. We see that $x \in L$ iff there exists a choice u_1 that makes N accept.

On the other hand, suppose that L is decidable by a polynomial-time NDTM N with oracle access to SAT. Then, x is in L if and only if there exists a sequence of non-deterministic choices and correct oracle answers that makes N accept x. That is, there is a sequence of choices $c_1, \dots, c_m \in \{0,1\}$ and answers to oracle queries $a_1, \dots, a_k \in \{0,1\}$ such that on input x, if the machine N uses the choices $c_1, \dots, c_m$ in its execution and receives a_i as the answer to its ith query, then **(1)** M reaches the accepting state q_{accept} and **(2)** all the answers are correct. Let φ_i denote the ith query that M makes to its oracle when executing on x using choices $c_1, \dots, x_m$ and receiving answers $a_1, \dots, a_k$. Then, the condition **(2)** can be phrased as follows: if $a_i = 1$ then there exists an assignment u_i such that $\varphi_i(u_i) = 1$ and if $a_i = 0$ then for every assignment v_i, $\varphi_i(v_i) = 0$. Thus,

we have that

$$x \in L \Leftrightarrow \exists c_1, \ldots, c_m, a_1, \ldots, a_k, u_1, \ldots, u_k \forall v_1, \ldots, v_k \text{ such that}$$

$$N \text{ accepts } x \text{ using choices } c_1, \ldots, c_m \text{ and answers } a_1, \ldots, a_k \text{ AND}$$

$$\forall i \in [k] \text{ if } a_i = 1 \text{ then } \varphi_i(u_i) = 1 \text{ AND}$$

$$\forall i \in [k] \text{ if } a_i = 0 \text{ then } \varphi_i(v_i) = 0$$

implying that $L \in \Sigma_2^p$. ∎

REMARK 5.16
Because having oracle access to a complete language for a class allows to solve every language in that class, some texts use the class name instead of the complete language in the notation for the oracle. Thus, some texts denote the class $\Sigma_2^p = \mathbf{NP}^{\mathsf{SAT}}$ by $\mathbf{NP}^{\mathbf{NP}}$, the class Σ_3^p by $\mathbf{NP}^{\mathbf{NP}^{\mathbf{NP}}}$ and etc.

WHAT HAVE WE LEARNED?

- The polynomial hierarchy is the set of languages that can be defined via a constant number of alternating quantifiers. It also has equivalent definitions via alternating TMs and oracle TMs. It contains several natural problems that are not known (or believed) to be in $\mathbf{NP}$.

- We conjecture that the hierarchy does not collapse in the sense that each of its levels is distinct from the previous ones.

- We can use the concept of alternations to prove that SAT cannot be solved simultaneously in linear time and sublinear space.

Chapter notes and history

The polynomial hierarchy was formally defined by Stockmeyer [Sto77], though the concept appears in the literature even earlier. For instance, Karp [Kar72] notes that *"a polynomial-bounded version of Kleene's Arithmetic Hierarchy (Rogers 1967) becomes trivial if $\mathbf{P} = \mathbf{NP}$."*

The class $\mathbf{DP}$ was defined by Papadimitriou and Yannakakis [PY82], who used it to characterize the complexity of identifying the facets of a polytope.

The class of complete problems for various levels of $\mathbf{PH}$ is not as rich as it is for $\mathbf{NP}$, but it does contain several interesting ones. See Schaeffer and Umans [SU02a, SU02b] for a recent list. The SUCCINCT SET-COVER problem is from Umans [Uma01], where it is also shown that the following optimization version of MIN-DNF is Σ_2^p-complete:

$$\left\{ \langle \varphi, k \rangle : \ \exists \, \mathrm{DNF}\, \varphi' \text{ of size at most } k, \text{ that is equivalent to DNF } \varphi \right\}.$$

Exercises

§1 Show that the language $\Sigma_i\mathsf{SAT}$ of (1) is complete for $\mathbf{\Sigma}_i^p$ under polynomial time reductions.

<div align="center">

Hint:: Use the **NP**-completeness of SAT.

</div>

§2 Prove Claim 5.11.

§3 Show that if 3SAT is polynomial-time reducible to $\overline{\mathsf{3SAT}}$ then $\mathbf{PH} = \mathbf{NP}$.

§4 Show that $\mathbf{PH}$ has a complete language iff it collapses to some finite level $\mathbf{\Sigma}_i^p$.

§5 Show that the definition of $\mathbf{PH}$ using ATMs coincides with our other definitions.

§6 Show that $\mathbf{APSPACE} = \mathbf{EXP}$.

<div align="center">

similar to those in the proof of Theorem 5.13.

Hint: The nontrivial direction $\mathbf{EXP} \subseteq \mathbf{APSPACE}$ uses ideas

</div>

§7 Show that $\mathbf{\Sigma}_2^p = \mathbf{NP}^{\mathsf{SAT}}$. Generalize your proof to give a characterization of $\mathbf{PH}$ in terms of oracle Turing machines.

§8 The class $\mathbf{DP}$ is defined as the set of languages L for which there are two languages $L_1 \in \mathbf{NP}, L_2 \in \mathbf{coNP}$ such that $L = L_1 \cap L_2$. (Do not confuse $\mathbf{DP}$ with $\mathbf{NP} \cap \mathbf{coNP}$, which may seem superficially similar.) Show that

    (a) EXACT INDSET $\in \mathbf{DP}$.

    (b) Every language in $\mathbf{DP}$ is polynomial-time reducible to EXACT INDSET.

§9 Suppose A is some language such that $\mathbf{P}^A = \mathbf{NP}^A$. Then show that $\mathbf{PH}^A \subseteq \mathbf{P}^A$ (in other words, the proof of Theorem **??** *relativizes*).

§10 Show that SUCCINCT SET-COVER $\in \mathbf{\Sigma}_2^p$.

§11 (Suggested by C. Umans) This problem studies VC-dimension, a concept important in machine learning theory. If $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ is a collection of subsets of a finite set U, the *VC dimension* of $\mathcal{S}$, denoted $VC(S)$, is the size of the largest set $X \subseteq U$ such that for every $X' \subseteq X$, there is an i for which $S_i \cap X = X'$. (We say that X is *shattered* by S.)

A boolean circuit C succinctly represents collection $\mathcal{S}$ if S_i consists of exactly those elements $x \in U$ for which $C(i, x) = 1$. Finally,

$$\text{VC-DIMENSION} = \{\langle C, k \rangle : \ C \text{ represents a collection } \mathcal{S} \text{ s.t. } VC(S) \geq k\}.$$

    (a) Show that VC-DIMENSION $\in \mathbf{\Sigma}_3^p$.

    (b) Show that VC-DIMENSION is $\mathbf{\Sigma}_3^p$-complete.

<div align="center">

your reduction can use the same set multiple times.

Hint: Reduce from Σ_3-3SAT. Also, the collection $\mathcal{S}$ produced by

</div>

Chapter 6

Circuits

> "One might imagine that $\mathbf{P} \neq \mathbf{NP}$, but SAT is tractable in the following sense: for every ℓ there is a very short program that runs in time ℓ^2 and correctly treats all instances of size ℓ. "
> Karp and Lipton, 1982

This chapter investigates a model of computation called a *Boolean circuit*, which is a generalization of Boolean formulae and a rough formalization of the familiar "silicon chip." Here are some motivations for studying it.

First, it is a natural model for *nonuniform* computation, by which we mean that a different "algorithm" is allowed for each input size. By contrast, our standard model thus far was *uniform computation*: the same Turing Machine (or algorithm) solves the problem for inputs of all (infinitely many) sizes. Nonuniform computation crops up often in complexity theory, and also in the rest of this book.

Second, in principle one can separate complexity classes such as $\mathbf{P}$ and $\mathbf{NP}$ by proving *lowerbounds* on circuit size. This chapter outlines why such lowerbounds ought to exist. In the 1980s, researchers felt boolean circuits are mathematically simpler than the Turing Machine, and thus proving circuit lowerbounds may be the right approach to separating complexity classes. Chapter 13 describes the partial successes of this effort and Chapter 22 describes where it is stuck.

This chapter defines the class $\mathbf{P}/\text{poly}$ of languages computable by polynomial-sized boolean circuits and explores its relation to $\mathbf{NP}$. We also encounter some interesting subclasses of $\mathbf{P}/\text{poly}$, including $\mathbf{NC}$, which tries to capture computations that can be efficiently performed on *highly parallel* computers. Finally, we show a (yet another) characterization of the polynomial hierarchy, this time using exponential-sized circuits of constant depth.

6.1   Boolean circuits

A Boolean circuit is a just a diagram showing how to derive an output from an input by a combination of the basic Boolean operations of OR ($\vee$), AND ($\wedge$) and NOT ($\neg$). For example, Figure 6.1 shows a circuit computing the XOR function. Here is the formal definition.
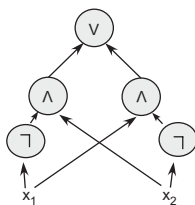
Figure 6.1: A circuit C computing the XOR function (i.e., $C(x_1, x_2) = 1$ iff $x_1 \neq x_2$).

DEFINITION 6.1 (BOOLEAN CIRCUITS)
For every $n, m \in \mathbb{N}$ a *Boolean circuit C with n inputs and m outputs*[1] is a directed acyclic graph. It contains n nodes with no incoming edges; called the *input nodes* and m nodes with no outgoing edges, called the *output nodes*. All other nodes are called *gates* and are labeled with one of $\vee$, $\wedge$ or $\neg$ (in other words, the logical operations OR, AND, and NOT). The $\vee$ and $\wedge$ nodes have fanin (i.e., number of incoming edges) of 2 and the $\neg$ nodes have fanin 1. The *size* of C, denoted by $|C|$, is the number of nodes in it.
The circuit is called a *Boolean formula* if each node has at most one outgoing edge.

The boolean circuit in the above definition implements a function from $\{0,1\}^n$ to $\{0,1\}^m$. This may be clear intuitively to most readers (especially those who have seen circuits in any setting) but here is the proof. Assume that the n input nodes and m output nodes are numbered in some canonical way. Thus each n-bit input can be used to assigned a value in $\{0,1\}$ to each input node. Next, since the graph is acyclic, we can associate an integral *depth* to each node (using breadth-first search, or the so-called *topological sorting* of the graph) such that each node has incoming edges only from nodes of higher depth. Now each node can be assigned a value from $\{0,1\}$ in a unique way as follows. Process the nodes in decreasing order of depth. For each node, examine its incoming edges and the values assigned to the nodes at the other end, and then apply the boolean operation ($\vee, \wedge$, or $\neg$) that this node is labeled with on those values. This gives a value to each node; the values assigned to the m output nodes by this process constitute an m-bit output of the circuit.

For every string $u \in \{0,1\}^n$, we denote by $C(u)$ the output of the circuit C on input u.

We recall that the Boolean operations OR, AND, and NOT form a *universal basis*, by which we mean that every function from $\{0,1\}^n$ to $\{0,1\}^m$ can be implemented by a boolean circuit (in fact, a boolean formula). See Claim 2.14. Furthermore, the "silicon chip" that we all know about is nothing but[2] an implementation of a boolean circuit using a technology called VLSI. Thus if we have a small circuit for a computational task, we can implement it very efficiently as a silicon chip. Of course, the circuit can only solve problems on inputs of a certain size. Nevertheless, this may not be a big restriction in our finite world. For instance, what if a small circuit *exists* that solves

---

[2] Actually, the circuits in silicon chips are not acyclic; in fact the cycles in the circuit are crucial for implementing "memory." However any computation that runs on a silicon chip of size C and finishes in time T can be performed by a boolean circuit of size $O(C \cdot T)$.

3SAT instances of up to say $100,000$ variables? If so, one could imagine a government-financed project akin to the Manhattan project that would try to discover such a small circuit, and then implement it as a silicon chip. This could be used in all kinds of commercial products (recall our earlier depiction of a world in which $\mathbf{P} = \mathbf{NP}$) and in particular would jeopardize every encryption scheme that does not use a huge key. This scenario is hinted at in the quote from Karp and Lipton at the start of the chapter.

As usual, we resort to asymptotic analysis to study the complexity of deciding a language by circuits.

DEFINITION 6.2 (CIRCUIT FAMILIES AND LANGUAGE RECOGNITION)
Let $T : \mathbb{N} \to \mathbb{N}$ be a function. A $T(n)$-*sized circuit family* is a sequence $\{C_n\}_{n\in\mathbb{N}}$ of Boolean circuits, where C_n has n inputs and a single output, such that $|C_n| \leq T(n)$ for every n.

We say that a language L is in $\mathbf{SIZE}(T(n))$ if there exists a $T(n)$-size circuit family $\{C_n\}_{n\in\mathbb{N}}$ such that for every $x \in \{0,1\}^n$, $x \in L \Leftrightarrow C(x) = 1$.

As noted in Claim 2.14, every language is decidable by a circuit family of size $O(n2^n)$, since the circuit for input length n could contain 2^n "hardwired" bits indicating which inputs are in the language. Given an input, the circuit looks up the answer from this table. (The reader may wish to work out an implementation of this circuit.) The following definition formalizes what we can think of as "small" circuits.

DEFINITION 6.3
$\mathbf{P}/\text{poly}$ is the class of languages that are decidable by polynomial-sized circuit families, in other words, $\cup_c \mathbf{SIZE}(n^c)$.

Of course, one can make the same kind of objections to the practicality of $\mathbf{P}/\text{poly}$ as for $\mathbf{P}$: viz., in what sense is a circuit family of size n^{100} practical, even though it has polynomial size. This was answered to some extent in Section 1.5.1. Another answer is that as complexity theorists we hope (eventually) to show that languages such as SAT are not in $\mathbf{P}/\text{poly}$. Thus the result will only be stronger if we allow even such large circuits in the definition of $\mathbf{P}/\text{poly}$.

The class $\mathbf{P}/\text{poly}$ contains $\mathbf{P}$. This is a corollary of Theorem 6.7 that we show below. Can we give a reasonable upperbound on the computational power of $\mathbf{P}/\text{poly}$? Unfortunately not, since it contains even undecidable languages.

EXAMPLE 6.4
Recall that we say that a language L is *unary* if it is a subset of $\{1^n : n \in \mathbb{N}\}$. Every unary language has linear size circuits since the circuit for an input size n only needs to have a single "hardwired" bit indicating whether or not 1^n is in the language. Hence the following unary language has linear size circuits, even though it is undecidable:

$$\{1^n : M_n \text{ outputs } 1 \text{ on input } 1^n\} . \tag{1}$$

where M_n is the machine represented by (the binary expansion of) the number n.

This example suggests that it may be fruitful to consider the restriction to circuits that can actually be built, say using a fairly efficient Turing machine. It will be most useful to formalize this using logspace computations.

Recall that a function $f : \{0,1\}^* \to \{0,1\}^*$ is implicitly logspace computable if the mapping $x, i \mapsto f(x)_i$ can be computed in logarithmic space (see Definition 4.14).

DEFINITION 6.5 (LOGSPACE-UNIFORM CIRCUIT FAMILIES)
A circuit family $\{C_n\}$ is *logspace uniform* if there is an implicitly logspace computable function mapping 1^n to the description of the circuit C_n.

Actually, to make this concrete we need to fix some representation of the circuits as strings. We will assume that the circuit of size N is represented by its $N \times N$ adjacency matrix and in addition an array of size N that gives the labels (gate type and input/output) of each node. This means that $\{C_n\}$ is logspace uniform if and only if the following functions are computable in $O(\log n)$ space:

- $\mathtt{SIZE}(n)$ returns the size m (in binary representation) of the circuit C_n.

- $\mathtt{TYPE}(n, i)$, where $i \in [m]$, returns the label and type of the ith node of C_n. That is it returns one of $\{\vee, \wedge, \neg, NONE\}$ and in addition $\langle OUTPUT, j \rangle$ or $\langle INPUT, j \rangle$ if i is the jth input or output node of C_n.

- $\mathtt{EDGE}(n, i, j)$ returns 1 if there is a directed edge in C_n between the ith node and the jth node.

Note that both the inputs and the outputs of these functions can be encoded using a logarithmic (in $|C_n|$) number of bits. The requirement that they run in $O(\log n)$ space means that we require that $\log |C_n| = O(\log n)$ or in other words that C_n is of size at most polynomial in n.

REMARK 6.6
Exercise 7 asks you to prove that the class of languages decided by such circuits does not change if we use the adjacency list (as opposed to matrix) representation. We will use the matrix representation from now on.

Polynomial circuits that are logspace-uniform correspond to a familiar complexity class:

THEOREM 6.7
A language has logspace-uniform circuits of polynomial size iff it is in **P**.

REMARK 6.8
Note that this implies that $\mathbf{P} \subseteq \mathbf{P}/\mathrm{poly}$.

PROOF SKETCH: The only if part is trivial. The if part follows the proof of the Cook-Levin Theorem (Theorem 2.10). Recall that we can simulate every time $O(T(n))$ TM M by an *oblivious* TM $\tilde{M}$ (whose head movement is independent of its input) running in time $O(T(n)^2)$ (or even $O(T(n) \log T(n))$ if we are more careful). In fact, we can ensure that the movement of the oblivious TM $\tilde{M}$ do not even depend on the contents of its work tape, and so, by simulating $\tilde{M}$ while ignoring

its read/write instructions, we can compute in $O(\log T(n))$ space for every i the position its heads will be at the ith step.[3]

Given this insight, it is fairly straightforward to translate the proof of Theorem 2.10 to prove that every language in **P** has a logspace-uniform circuit family. The idea is that if $L \in \mathbf{P}$ then it is decided by an oblivious TM $\tilde{M}$ of the form above. We will use that to construct a logspace uniform circuit family $\{C_n\}_{n \in \mathbb{N}}$ such that for every $x \in \{0, 1\}^n$, $C_n(x) = \tilde{M}(x)$.

Recall that, as we saw in that proof, the *transcript* of $\tilde{M}$'s execution on input x is the sequence $z_1, \ldots, z_T$ of *snapshots* (machine's state and symbols read by all heads) of the execution at each step in time. Assume that each such z_i is encoded by a string (that needs only to be of constant size). We can compute the string z_i based the previous snapshots z_{i-1} and $z_{i_1}, \ldots, z_{i_k}$ where z_{i_j} denote the last step that $\tilde{M}$'s j^{th} head was in the same position as it is in the ith step. Because these are only a constant number of strings of constant length, we can compute z_i from these previous snapshot using a constant-sized circuit. Also note that, under our assumption above, given the indices i and $i' < i$ we can easily check whether z_i depends on $z_{i'}$.

The composition of all these constant-sized circuits gives rise to a circuit that computes from the input x, the snapshot z_T of the last step of $\tilde{M}$'s execution on x. There is a simple constant-sized circuit that, given z_T outputs 1 if and only if z_T is an accepting snapshot (in which $\tilde{M}$ outputs 1 and halts). Thus, we get a circuit C such that $C(x) = \tilde{M}(x)$ for every $x \in \{0, 1\}^n$.

Because our circuit C is composed of many small (constant-sized) circuits, and determining which small circuit is applied to which nodes can be done in logarithmic space, it is not hard to see that we can find out every individual bit of C's representation in logarithmic space. (In fact, one can show that the functions SIZE, TYPE and EDGE above can be computed using only logarithmic space and polylogarithmic time.) ∎

6.1.1 Turing machines that take advice

There is a way to define $\mathbf{P}/\text{poly}$ using Turing machines that "take advice."

DEFINITION 6.9
Let $T, a : \mathbb{N} \to \mathbb{N}$ be functions. The class of *languages decidable by time-$T(n)$ TM's with $a(n)$ advice*, denoted $\mathbf{DTIME}(T(n))_{/a(n)}$, contains every L such that there exists a sequence $\{\alpha_n\}_{n \in \mathbb{N}}$ of strings with $\alpha_n \in \{0, 1\}^{a(n)}$ and a TM M satisfying

$$M(x, \alpha_n) = 1 \Leftrightarrow x \in L$$

for every $x \in \{0, 1\}^n$, where on input (x, α_n) the machine M runs for at most $O(T(n))$ steps.

EXAMPLE 6.10
Every unary language can be be decided by a polynomial time Turing machine with 1 bit of advice. The advice string for inputs of length n is the single bit indicating whether or not 1^n is in the language. In particular this is true of the language of Example 6.4.

---

[3]In fact, typically the movement pattern is simple enough (for example a sequence of $T(n)$ left to right and back sweeps of the tape) that for every i we can compute this information using only $O(\log T(n))$ space and $\text{polylog} T(n)$ time.

This is an example of a more general phenomenon described in the next theorem.

THEOREM 6.11
$\mathbf{P}/\text{poly} = \cup_{c,d}\mathbf{DTIME}(n^c)/n^d$

PROOF: If $L \in \mathbf{P}/\text{poly}$, we can provide the polynomial-sized description of its circuit family as advice to a Turing machine. When faced with an input of size n, the machine just simulates the circuit for this circuit provided to it.

Conversely, if L is decidable by a polynomial-time Turing machine M with access to an advice family $\{\alpha_n\}_{n\in\mathbb{N}}$ of size $a(n)$ for some polynomial a, then we can use the construction of Theorem 6.7 to construct for every n, a polynomial-sized circuit D_n such that on every $x \in \{0,1\}^n$, $\alpha \in \{0,1\}^{a(n)}$, $D_n(x,\alpha) = M(x,\alpha)$. We let the circuit C_n be the polynomial circuit that maps x to $D_n(x,\alpha_n)$. That is, C_n is equal to the circuit D_n with the string α_n "hardwired" as its second input. ∎

REMARK 6.12
By "hardwiring" an input into a circuit we mean taking a circuit C with two inputs $x \in \{0,1\}^n$, $y \in \{0,1\}^m$ and transforming it into the circuit C_y that for every x returns $C(x,y)$. It is easy to do so while ensuring that the size of C_y is not greater than the size of C. This simple idea is often used in complexity theory.

6.2   Karp-Lipton Theorem

Karp and Lipton formalized the question of whether or not SAT has small circuits as: Is SAT in $\mathbf{P}/\text{poly}$? They showed that the answer is "NO" if the polynomial hierarchy does not collapse.

THEOREM 6.13 (KARP-LIPTON, WITH IMPROVEMENTS BY SIPSER)
If $\mathbf{NP} \subseteq \mathbf{P}/\text{poly}$ then $\mathbf{PH} = \mathbf{\Sigma}_2^p$.

PROOF: To show that $\mathbf{PH} = \mathbf{\Sigma}_2^p$ it is enough to show that $\mathbf{\Pi}_2^p \subseteq \mathbf{\Sigma}_2^p$ and in particular it suffices to show that $\mathbf{\Sigma}_2^p$ contains the $\mathbf{\Pi}_2^p$-complete language $\Pi_2\mathsf{SAT}$ consisting of all true formulae of the form

$$\forall u \in \{0,1\}^n \, \exists v \in \{0,1\}^n \, \varphi(u,v) = 1 \,. \tag{2}$$

where φ is an unquantified Boolean formula.

If $\mathbf{NP} \subseteq \mathbf{P}/\text{poly}$ then there is a polynomial p and a $p(n)$-sized circuit family $\{C_n\}_{n\in\mathbb{N}}$ such that for every Boolean formula φ and $u \in \{0,1\}^n$, $C_n(\varphi,u) = 1$ if and only if there exists $v \in \{0,1\}^n$ such that $\varphi(u,v) = 1$. Yet, using the search to decision reduction of Theorem 2.19, we actually know that there is a $q(n)$-sized circuit family $\{C_n'\}_{n\in\mathbb{N}}$ such that for every such formula φ and $u \in \{0,1\}^n$, if there is a string $v \in \{0,1\}^n$ such that $\varphi(u,v) = 1$ then $C_n'(\varphi,u)$ outputs such a string v. Since C_n' can be described using $10q(n)^2$ bits, this implies that if (2) is true then the following quantified formula is also true:

$$\exists w \in \{0,1\}^{10q(n)^2} \, \forall u \in \{0,1\}^n \, w \text{ describes a circuit } C' \text{ s.t. } \varphi(u,C'(\varphi,u)) = 1 \,. \tag{3}$$

Yet if (2) is false then certainly (regardless of whether $\mathbf{P} = \mathbf{NP}$) the formula (3) is false as well, and hence (3) is actually *equivalent* to (2)! However, since evaluating a circuit on an input can be done in polynomial time, evaluating the truth of (3) can be done in $\mathbf{\Sigma}_2^p$. ∎

Similarly the following theorem can be proven, though we leave the proof as Exercise 3.

THEOREM 6.14 (KARP-LIPTON, ATTRIBUTED TO A. MEYER)
If $\mathbf{EXP} \subseteq \mathbf{P}/\text{poly}$ *then* $\mathbf{EXP} = \mathbf{\Sigma}_2^p$.

Combining the time hierarchy theorem (Theorem 3.1) with the previous theorem implies that if $\mathbf{P} = \mathbf{NP}$ then $\mathbf{EXP} \not\subseteq \mathbf{P}/poly$. Thus upperbounds (in this case, $\mathbf{NP} \subseteq \mathbf{P}$) can potentially be used to prove circuit lowerbounds.

6.3 Circuit lowerbounds

Since $\mathbf{P} \subseteq \mathbf{P}/\text{poly}$, if $\mathbf{NP} \not\subseteq \mathbf{P}/\text{poly}$ then $\mathbf{P} \neq \mathbf{NP}$. The Karp-Lipton theorem gives hope that $\mathbf{NP} \not\subseteq \mathbf{P}/\text{poly}$. Can we resolve $\mathbf{P}$ versus $\mathbf{NP}$ by proving $\mathbf{NP} \not\subseteq \mathbf{P}/\text{poly}$? There is reason to invest hope in this approach as opposed to proving direct lowerbounds on Turing machines. By representing computation using circuits we seem to actually peer into the guts of it rather than treating it as a blackbox. Thus we may be able to get around the limitations of relativizing methods shown in Chapter 3.

Sadly, such hopes have not yet come to pass. After two decades, the best circuit size lowerbound for an $\mathbf{NP}$ language is only $5n$. (However, see Exercise 1 for a better lowerbound for a language in $\mathbf{PH}$.) On the positive side, we have had notable success in proving lowerbounds for more restricted circuit models, as we will see in Chapter 13.

By the way, it is easy to show that for large enough n, almost every boolean function on n variables requires large circuits.

THEOREM 6.15
For $n \geq 100$, *almost all boolean functions on* n *variables require circuits of size at least* $2^n/(10n)$.

PROOF: We use a simple counting argument. There are at most s^{3s} circuits of size s (just count the number of labeled directed graphs, where each node has indegree at most 2). Hence this is an upperbound on the number of functions on n variables with circuits of size s. For $s = 2^n/(10n)$, this number is at most $2^{2^n/10}$, which is miniscule compared 2^{2^n}, the number of boolean functions on n variables. Hence most Boolean functions do not have such small circuits. ∎

REMARK 6.16
Another way to present this result is as showing that with high probability, a *random* function from $\{0,1\}^n$ to $\{0,1\}$ does not have a circuit of size $2^n/10n$. This kind of proof method, showing the existence of an object with certain properties by arguing that a random object has these properties with high probability, is called the *probabilistic method*, and will be repeatedly used in this book.

The problem with the above counting argument is of course, that it does not yield an explicit Boolean function (say an $\mathbf{NP}$ language) that requires large circuits.

6.4 Non-uniform hierarchy theorem

As in the case of deterministic time, non-deterministic time and space bounded machines, Boolean circuits also have a hierarchy theorem. That is, larger circuits can compute strictly more functions than smaller ones:

THEOREM 6.17
For every functions $T, T' : \mathbb{N} \to \mathbb{N}$ with $2^n/(100n) > T'(n) > T(n) > n$ and $T(n) \log T(n) = o(T'(n))$,

$$\mathbf{SIZE}(T(n)) \subsetneq \mathbf{SIZE}(T'(n))$$

PROOF: The diagonalization methods of Chapter 3 do not seem to work for such a function, but nevertheless, we can prove it using the counting argument from above. To show the idea, we prove that $\mathbf{SIZE}(n) \subsetneq \mathbf{SIZE}(n^2)$.

For every ℓ, there is a function $f : \{0,1\}^\ell \to \{0,1\}$ that is not computable by $2^\ell/(10\ell)$-sized circuits. On the other hand, every function from $\{0,1\}^\ell$ to $\{0,1\}$ is computable by a $2^\ell 10\ell$-sized circuit.

Therefore, if we set $\ell = 1.1 \log n$ and let $g : \{0,1\}^n \to \{0,1\}$ be the function that applies f on the first ℓ bits of its input, then

$$g \in \qquad \mathbf{SIZE}(2^\ell 10\ell) = \qquad \mathbf{SIZE}(11n^{1.1} \log n) \subseteq \qquad \mathbf{SIZE}(n^2)$$
$$g \notin \qquad \mathbf{SIZE}(2^\ell/(10\ell)) = \qquad \mathbf{SIZE}(n^{1.1}/(11 \log n)) \supseteq \qquad \mathbf{SIZE}(n)$$

∎

6.5 Finer gradations among circuit classes

There are two reasons why subclasses of $\mathbf{P}/\text{poly}$ are interesting. First, proving lowerbounds for these subclasses may give insight into how to separate $\mathbf{NP}$ from $\mathbf{P}/\text{poly}$. Second, these subclasses correspond to interesting computational models in their own right.

Perhaps the most interesting connection is to *massively parallel computers*. In such a computer one uses simple off-the-shelf microprocessors and links them using an *interconnection network* that allows them to send messages to each other. Usual interconnection networks such as the *hypercube* allows linking n processors such that interprocessor communication is possible —assuming some upperbounds on the total load on the network—in $O(\log n)$ steps. The processors compute in lock-step (for instance, to the ticks of a global clock) and are assumed to do a small amount of computation in each step, say an operation on $O(\log n)$ bits. Thus each processor computers has enough memory to remember its own address in the interconnection network and to write down the address of any other processor, and thus send messages to it. We are purposely omitting many details of the model (Leighton [Lei92] is the standard reference for this topic) since the validity of Theorem 6.24 below does not depend upon them. (Of course, we are only aiming for a loose characterization of parallel computation, not a very precise one.)

6.5.1 Parallel computation and NC

DEFINITION 6.18
A computational task is said to have *efficient parallel algorithms* if inputs of size n can be solved using a parallel computer with $n^{O(1)}$ processors and in time $\log^{O(1)} n$.

EXAMPLE 6.19
Given two n bit numbers x, y we wish to compute $x + y$ fast in parallel. The gradeschool algorithm proceeds from the least significant bit and maintains a *carry bit*. The most significant bit is computed only after n steps. This algorithm does not take advantage of parallelism. A better algorithm called *carry lookahead* assigns each bit position to a separate processor and then uses interprocessor communication to propagate carry bits. It takes $O(n)$ processors and $O(\log n)$ time.

There are also efficient parallel algorithms for integer multiplication and division (the latter is quite nonintuitive and unlike the gradeschool algorithm!).

EXAMPLE 6.20
Many matrix computations can be done efficiently in parallel: these include computing the product, rank, determinant, inverse, etc. (See exercises.)

Some graph theoretic algorithms such as shortest paths and minimum spanning tree also have fast parallel implementations.

But many well-known polynomial-time problems such as minimum matching, maximum flows, and linear programming are not known to have any good parallel implementations and are conjectured not to have any; see our discussion of **P**-completeness below.

Now we relate parallel computation to circuits. The *depth* of a circuit is the length of the longest directed path from an input node to the output node.

DEFINITION 6.21 (NICK'S CLASS OR NC)
A language is in $\mathbf{NC}^i$ if there are constants $c, d > 0$ such that it can be decided by a logspace-uniform family of circuits $\{C_n\}$ where C_n has size $O(n^c)$ and depth $O(\log^d n)$. The class $\mathbf{NC}$ is $\cup_{i \geq 1} \mathbf{NC}^i$.

A related class is the following.

DEFINITION 6.22 ($\mathbf{AC}$)
The class $\mathbf{AC}^i$ is defined similarly to $\mathbf{NC}^i$ except gates are allowed to have unbounded fanin. The class $\mathbf{AC}$ is $\cup_{i \geq 0} \mathbf{AC}^i$.

Since unbounded (but poly(n)) fanin can be simulated using a tree of ORs/ANDs of depth $O(\log n)$, we have $\mathbf{NC}^i \subseteq \mathbf{AC}^i \subseteq \mathbf{NC}^{i+1}$, and the inclusion is known to be strict for $i = 0$ as we will see in Chapter 13. (Notice, $\mathbf{NC}^0$ is extremely limited since the circuit's output depends upon a constant number of input bits, but $\mathbf{AC}^0$ does not suffer from this limitation.)

EXAMPLE 6.23
The language PARITY $= \{x : \ x$ has an odd number of 1s$\}$ is in $\mathbf{NC}^1$. The circuit computing it has the form of a binary tree. The answer appears at the root; the left subtree computes the parity of the first $|x|/2$ bits and the right subtree computes the parity of the remaining bits. The gate at the top computes the parity of these two bits. Clearly, unwrapping the recursion implicit in our description gives a circuit of dept $O(\log n)$.

The classes $\mathbf{AC}$, $\mathbf{NC}$ are important because of the following.

THEOREM 6.24
A language has efficient parallel algorithms iff it is in $\mathbf{NC}$.

PROOF: Suppose a language $L \in \mathbf{NC}$ and is decidable by a circuit family $\{C_n\}$ where C_n has size $N = O(n^c)$ and depth $D = O(\log^d n)$. Take a general purpose parallel computer with N nodes and configure it to decide L as follows. Compute a description of C_n and allocate the role of each circuit node to a distinct processor. (This is done once, and then the computer is ready to compute on any input of length n.) Each processor, after computing the output at its assigned node, sends the resulting bit to every other circuit node that needs it. Assuming the interconnection network delivers all messages in $O(\log N)$ time, the total running time is $O(\log^{d+1} N)$.

The reverse direction is similar, with the circuit having $N \cdot D$ nodes arranged in D layers, and the ith node in the tth layer performs the computation of processor i at time t. The role of the interconnection network is played by the circuit wires. ∎

6.5.2 P-completeness

A major open question in this area is whether $\mathbf{P} = \mathbf{NC}$. We believe that the answer is NO (though we are currently even unable to separate $\mathbf{PH}$ from $\mathbf{NC}^1$). This motivates the theory of $\mathbf{P}$-*completeness*, a study of which problems are likely to be in $\mathbf{NC}$ and which are not.

DEFINITION 6.25
A language is $\mathbf{P}$-*complete* if it is in $\mathbf{P}$ and every language in $\mathbf{P}$ is logspace-reducible to it (as per Definition 4.14).

The following easy theorem is left for the reader as Exercise 12.

THEOREM 6.26
If language L is $\mathbf{P}$-complete then

   1. $L \in \mathbf{NC}$ iff $\mathbf{P} = \mathbf{NC}$.

2. $L \in \mathbf{L}$ iff $\mathbf{P} = \mathbf{L}$. (Where $\mathbf{L}$ is the set languages decidable in logarithmic space, see Definition 4.5.)

The following is a fairly natural **P**-complete language:

THEOREM 6.27
Let CIRCUIT-EVAL *denote the language consisting of all pairs* $\langle C, x \rangle$ *such that* C *is an* n*-inputs single-output circuit and* $x \in \{0, 1\}^n$ *satisfies* $C(x) = 1$. *Then* CIRCUIT-EVAL *is* **P**-*complete.*

PROOF: The language is clearly in **P**. A logspace-reduction from any other language in **P** to this language is implicit in the proof of Theorem 6.7. ■

6.6 Circuits of exponential size

As noted, every language has circuits of size $O(n2^n)$. However, actually finding these circuits may be difficult— sometimes even undecidable. If we place a uniformity condition on the circuits, that is, require them to be efficiently computable then the circuit complexity of some languages could exceed $n2^n$. In fact it is possible to give alternative definitions of some familiar complexity classes, analogous to the definition of **P** in Theorem 6.7.

DEFINITION 6.28 (DC-UNIFORM)
Let $\{\mathcal{C}_n\}_{n \geq 1}$ be a circuit family. We say that it is a *Direct Connect uniform* (DC uniform) family if, given $\langle n, i \rangle$, we can compute in polynomial time the ith but of (the representation of) the circuit C_n. More concretely, we use the adjacency matrix representation and hence a family $\{C_n\}_{n \in \mathbb{N}}$ is DC uniform iff the functions SIZE, TYPE and EDGE defined in Remark **??** are computable in polynomial time.

Note that the circuits may have exponential size, but they have a succinct representation in terms of a TM which can systematically generate any required node of the circuit in polynomial time.

Now we give a (yet another) characterization of the class **PH**, this time as languages computable by uniform circuit families of bounded depth. We leave it as Exercise 13.

THEOREM 6.29
$L \in PH$ *iff* L *can be computed by a DC uniform circuit family* $\{\mathcal{C}_n\}$ *that*

- *uses AND, OR, NOT gates.*

- *has size* $2^{n^{O(1)}}$ *and constant depth (i.e., depth* $O(1)$*).*

- *gates can have unbounded (exponential) fanin.*

- *the NOT gates appear only at the input level.*

If we drop the restriction that the circuits have constant depth, then we obtain exactly **EXP** (see Exercise 14).

6.7 Circuit Satisfiability and an alternative proof of the Cook-Levin Theorem

Boolean circuits can be used to define the following **NP**-complete language:

DEFINITION 6.30
The *circuit satisfiability* language CKT-SAT consists of all (strings representing) circuits with a single output that have a satisfying assignment. That is, a string representing an n-input circuit C is in CKT-SAT iff there exists $u \in \{0,1\}^n$ such that $C(u) = 1$.

CKT-SAT is clearly in **NP** because the satisfying assignment can serve as the certificate. It is also clearly **NP**-hard as every CNF formula is in particular a Boolean circuit. However, CKT-SAT can also be used to give an alternative proof (or, more accurately, a different presentation of the same proof) for the Cook-Levin Theorem by combining the following two lemmas:

LEMMA 6.31
CKT-SAT *is* **NP**-*hard.*

PROOF: Let L be an **NP**-language and let p be a polynomial and M a polynomial-time TM such that $x \in L$ iff $M(x,u) = 1$ for some $u \in \{0,1\}^{p(|x|)}$. We reduce L to CKT-SAT by mapping (in polynomial-time) x to a circuit C_x with $p(|x|)$ inputs and a single output such that $C_x(u) = M(x,u)$ for every $u \in \{0,1\}^{p(|x|)}$. Clearly, $x \in L \Leftrightarrow C_x \in$ CKT-SAT and so this suffices to show that $L \leq_P$ CKT-SAT.

Yet, it is not hard to come up with such a circuit. Indeed, the proof of Theorem 6.7 yields a way to map M, x into the circuit C_x in logarithmic space (which in particular implies polynomial time). ∎

LEMMA 6.32
CKT-SAT $\leq_p$ 3SAT

PROOF: As mentioned above this follows from the Cook-Levin theorem but we give here a direct reduction. If C is a circuit, we map it into a 3CNF formula φ as follows:

For every node v_i of C we will have a corresponding variable z_i in φ. If the node v_i is an AND of the nodes v_j and v_k then we add to φ clauses that are equivalent to the condition "$z_i = (z_j \wedge z_k)$". That is, we add the clauses

$$(\overline{z}_i \vee \overline{z}_j \vee z_k) \wedge (\overline{z}_i \vee z_j \vee \overline{z}_k) \wedge (\overline{z}_i \vee z_j \vee z_k) \wedge (z_i \vee \overline{z}_j \vee \overline{z}_k).$$

Similarly, if v_i is an OR of v_j and v_k then we add clauses equivalent to "$z_i = (z_j \vee z_k)$", and if v_i is the NOT of v_j then we add the clauses $(z_i \vee z_j) \wedge (\overline{z}_i \vee \overline{z}_j)$.

Finally, if v_i is the output node of C then we add the clause z_i to φ. It is not hard to see that the formula φ will be satisfiable if and only if the circuit C is. ∎

WHAT HAVE WE LEARNED?

- Boolean circuits can be used as an alternative computational model to TMs. The class $\mathbf{P}/poly$ of languages decidable by polynomial-sized circuits is a strict superset of $\mathbf{P}$ but does not contain $\mathbf{NP}$ unless the hierarchy collapses.

- Almost every function from $\{0,1\}^n$ to $\{0,1\}$ requires exponential-sized circuits. Finding even one function in $\mathbf{NP}$ with this property would show that $\mathbf{P} \neq \mathbf{NP}$.

- The class $\mathbf{NC}$ of languages decidable by (uniformly constructible) circuits with polylogarithmic depth and polynomial size corresponds to computational tasks that can be efficiently parallelized.

Chapter notes and history

Karp-Lipton theorem is from [KL82]. Karp and Lipton also gave a more general definition of advice that can be used to define the class $\mathcal{C}/a(n)$ for every complexity class $\mathcal{C}$ and function a. However, we do not use this definition here since it does not seem to capture the intuitive notion of advice for classes such as $\mathbf{NP} \cap \mathbf{coNP}$, $\mathbf{BPP}$ and others.

The class of $\mathbf{NC}$ algorithms as well as many related issues in parallel computation are discussed in Leighton [?].

Exercises

§1 [Kannan [Kan82]] Show for every $k > 0$ that $\mathbf{PH}$ contains languages whose circuit complexity is $\Omega(n^k)$.

Hint: Keep in mind the proof of the *existence* of functions with high circuit complexity.

§2 Solve the previous question with $\mathbf{PH}$ replaced by $\mathbf{\Sigma}_2^p$.

§3 ([KL82], attributed to A. Meyer) Show that if $\mathbf{EXP} \subseteq \mathbf{P}/poly$ then $\mathbf{EXP} = \mathbf{\Sigma}_2^p$.

§4 Show that if $\mathbf{P} = \mathbf{NP}$ then there is a language in $\mathbf{EXP}$ that requires circuits of size $2^n/n$.

§5 A language $L \subseteq \{0,1\}^*$ is sparse if there is a polynomial p such that $|L \cap \{0,1\}^n| \leq p(n)$ for every $n \in \mathbb{N}$. Show that every sparse language is in $\mathbf{P}/poly$.

§6 (X's Theorem 19??) Show that if a sparse language is $\mathbf{NP}$-complete then $\mathbf{P} = \mathbf{NP}$. (This is a strengthening of Exercise 13 of Chapter 2.)

Hint: Show a recursive exponential-time algorithm S that on input a u-variable formula ϕ and a string $v \in \{0,1\}^n$ outputs 1 iff ϕ has a satisfying assignment v such that $v < u$ when both are interpreted as the binary representation of a number in $[2^n]$. Use the reduction from SAT to L to prune the recursion tree of S.

§7 Show a logspace implicitly computable function f that maps any n-vertex graph in adjacency matrix representation into the same graph in adjacency list representation. You can think of the adjacency list representation of an n-vertex graph as a sequence of n strings of size $O(n \log n)$ each, where the ith string contains the list of neighbors of the ith vertex in the graph (and is padded with zeros if necessary).

§8 (*Open*) Suppose we make a stronger assumption than $\mathbf{NP} \subseteq \mathbf{P}/\text{poly}$: every language in $\mathbf{NP}$ has linear size circuits. Can we show something stronger than $\mathbf{PH} = \mathbf{\Sigma}_2^p$?

§9 (a) Describe an $\mathbf{NC}$ circuit for the problem of computing the product of two given $n \times n$ matrices A, B.

   (b) Describe an $\mathbf{NC}$ circuit for computing, given an $n \times n$ matrix, the matrix A^n.

   Hint: Use repeated squaring: $A^{2^k} = (A^{2^{k-1}})^2$.

   (c) Conclude that the PATH problem (and hence every $\mathbf{NL}$ language) is in $\mathbf{NC}$.

   Hint: What is the meaning of the (i,j)th entry of A^n?

§10 A *formula* is a circuit in which every node (except the input nodes) has outdegree 1. Show that a language is computable by polynomial-size formulae iff it is in (nonuniform) $\mathbf{NC}^1$.

   Hint: a formula may be viewed — once we exclude the input nodes—as a directed binary tree, and in a binary tree of size m there is always a node whose removal leaves subtrees of size at most $2m/3$ each.

§11 Show that $\mathbf{NC}^1 = \mathbf{L}$. Conclude that $\mathbf{PSPACE} \neq \mathbf{NC}^1$.

§12 Prove Theorem 6.26. That is, prove that if L is $\mathbf{P}$-complete then $L \in \mathbf{NC}$ (resp. $\mathbf{L}$) iff $\mathbf{P} = \mathbf{NC}$ (resp. $\mathbf{L}$).

§13 Prove Theorem 6.29 (that $\mathbf{PH}$ is the set of languages with constant-depth DC uniform circuits).

§14 Show that $\mathbf{EXP}$ is exactly the set of languages with DC uniform circuits of size 2^{n^c} where c is some constant (c may depend upon the language).

§15 Show that if linear programming has a fast parallel algorithm then $\mathbf{P} = \mathbf{NC}$.

   Hint: in your reduction, express the CIRCUIT-EVAL problem as a linear program and use the fact that $x \vee y = 1$ iff $x + y \geq 1$. Be careful; the variables in a linear program are real-valued and not boolean!

Chapter 7

Randomized Computation

> "We do not assume anything about the distribution of the instances of the problem to be solved. Instead we incorporate randomization into the algorithm itself... It may seem at first surprising that employing randomization leads to efficient algorithm. This claim is substantiated by two examples. The first has to do with finding the nearest pair in a set of n points in $\mathbb{R}^k$. The second example is an extremely efficient algorithm for determining whether a number is prime."
> Michael Rabin, 1976

Thus far our standard model of computation has been the deterministic Turing Machine. But everybody who is even a little familiar with computation knows that that real-life computers need not be deterministic since they have built-in "random number generators." In fact these generators are very useful for computer simulation of "random" processes such as nuclear fission or molecular motion in gases or the stock market. This chapter formally studies *probablistic computation*, and complexity classes associated with it.

We should mention right away that it is an open question whether or not the universe has any randomness in it (though quantum mechanics seems to guarantee that it does). Indeed, the output of current "random number generators" is not guaranteed to be truly random, and we will revisit this limitation in Section 7.4.3. For now, assume that true random number generators exist. Then arguably, a realistic model for a real-life computer is a Turing machine with a random number generator, which we call a *Probabilistic Turing Machine* (PTM). It is natural to wonder whether difficult problems like 3SAT are efficiently solvable using a PTM.

We will formally define the class **BPP** of languages decidable by polynomial-time PTMs and discuss its relation to previously studied classes such as **P**/poly and **PH**. One consequence is that if **PH** does not collapse, then 3SAT does not have efficient probabilistic algorithms.

We also show that probabilistic algorithms can be very practical by presenting ways to greatly reduce their error to absolutely minuscule quantities. Thus the class **BPP** (and its sister classes **RP**, **coRP** and **ZPP**) introduced in this chapter are arguably as important as **P** in capturing efficient computation. We will also introduce some related notions such as probabilistic logspace algorithms and probabilistic reductions.

Though at first randomization seems merely a tool to allow simulations of randomized physical processes, the surprising fact is that in the past three decades randomization has led to more efficient —and often simpler—algorithms for problems in a host of other fields—such as combinatorial optimization, algebraic computation, machine learning, and network routing.

In complexity theory too, the role of randomness extends far beyond a study of randomized algorithms and classes such as **BPP**. Entire areas such as cryptography and interactive and probabilistically checkable proofs rely on randomness in an essential way, sometimes to prove results whose statement did not call for randomness at all. The groundwork for studying those areas will be laid in this chapter.

In a later chapter, we will learn something intriguing: to some extent, the power of randomness may be a mirage. If a certain plausible complexity-theoretic conjecture is true (see Chapters 16 and 17), then every probabilistic algorithm can be simulated by a deterministic algorithm (one that does not use any randomness whatsoever) with only polynomial overhead.

Throughout this chapter and the rest of the book, we will use some notions from elementary probability on finite sample spaces; see Appendix A for a quick review.

7.1   Probabilistic Turing machines

We now define probabilistic Turing machines (PTMs). Syntactically, a PTM is no different from a nondeterministic TM: it is a TM with two transition functions δ_0, δ_1. The difference lies in how we interpret the graph of all possible computations: instead of asking whether there *exists* a sequence of choices that makes the TM accept, we ask how large is the *fraction* of choices for which this happens. More precisely, if M is a PTM, then we envision that in every step in the computation, M chooses randomly which one of its transition functions to apply (with probability half applying δ_0 and with probability half applying δ_1). We say that M decides a language if it outputs the right answer with probability at least 2/3.

Notice, the ability to pick (with equal probability) one of δ_0, δ_1 to apply at each step is equivalent to the machine having a "fair coin", which, each time it is tossed, comes up "Heads" or "Tails" with equal probability *regardless of the past history of Heads/Tails.* As mentioned, whether or not such a coin exists is a deep philosophical (or scientific) question.

DEFINITION 7.1 (THE CLASSES **BPTIME** AND **BPP**)
For $T : \mathbb{N} \to \mathbb{N}$ and $L \subseteq \{0,1\}^*$, we say that a PTM M decides L in time $T(n)$, if for every $x \in \{0,1\}^*$, M halts in $T(|x|)$ steps regardless of its random choices, and $\Pr[M(x) = L(x)] \geq 2/3$, where we denote $L(x) = 1$ if $x \in L$ and $L(x) = 0$ if $x \notin L$. We let **BPTIME**$(T(n))$ denote the class of languages decided by PTMs in $O(T(n))$ time and let **BPP** $= \cup_c$**BPTIME**(n^c).

REMARK 7.2
We will see in Section 7.4 that this definition is quite robust. For instance, the "coin" need not be fair. The constant 2/3 is arbitrary in the sense that it can be replaced with any other constant

greater than half without changing the classes $\textbf{BPTIME}(T(n))$ and $\textbf{BPP}$. Instead of requiring the machine to always halt in polynomial time, we could allow it to halt in *expected* polynomial time.

REMARK 7.3
While Definition 7.1 allows the PTM M, given input x, to output a value different from $L(x)$ with positive probability, this probability is only over the random choices that M makes in the computation. In particular for *every* input x, $M(x)$ will output the right value $L(x)$ with probability at least $2/3$. Thus $\textbf{BPP}$, like $\textbf{P}$, is still a class capturing *worst-case* computation.

Since a deterministic TM is a special case of a PTM (where both transition functions are equal), the class $\textbf{BPP}$ clearly contains $\textbf{P}$. As alluded above, under plausible complexity assumptions it holds that $\textbf{BPP} = \textbf{P}$. Nonetheless, as far as we know it may even be that $\textbf{BPP} = \textbf{EXP}$. (Note that $\textbf{BPP} \subseteq \textbf{EXP}$, since given a polynomial-time PTM M and input $x \in \{0,1\}^n$ in time $2^{\text{poly}(n)}$ it is possible to enumerate all possible random choices and compute precisely the probability that $M(x) = 1$.)

An alternative definition.   As we did with $\textbf{NP}$, we can define $\textbf{BPP}$ using deterministic TMs where the "probabilistic choices" to apply at each step can be provided to the TM as an additional input:

DEFINITION 7.4 ($\textbf{BPP}$, ALTERNATIVE DEFINITION)
$\textbf{BPP}$ contains a language L if there exists a polynomial-time TM M and a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$, $\Pr_{r \in_R \{0,1\}^{p(|x|)}}[M(x,r) = L(x)] \geq 2/3$.

7.2   Some examples of PTMs

The following examples demonstrate how randomness can be a useful tool in computation. We will see many more examples in the rest of this book.

7.2.1   Probabilistic Primality Testing

In *primality testing* we are given an integer N and wish to determine whether or not it is prime. Generations of mathematicians have learnt about prime numbers and —before the advent of computers— needed to do primality testing to test various conjectures[1]. Ideally, we want efficient algorithms, which run in time polynomial in the size of N's representation, in other words, $\text{poly}(\log n)$. We knew of no such efficient algorithms[2] until the 1970s, when an effficient probabilistic algorithm was discovered. This was one of the first to demonstrate the power of probabilistic algorithms. In a recent breakthrough, Agrawal, Kayal and Saxena [?] gave a *deterministic* polynomial-time algorithm for primality testing.

---

[1]Though a very fast human computer himself, Gauss used the help of a human supercomputer –an autistic person who excelled at fast calculations—to do primality testing.

[2]In fact, in his letter to von Neumann quoted in Chapter 2, Gödel explicitly mentioned this problem as an example for an interesting problem in $\textbf{NP}$ but not known to be efficiently solvable.

Formally, primality testing consists of checking membership in the language $\mathsf{PRIMES} = \{ \ulcorner N \urcorner : N$ is a prime num
Notice, the corresponding *search* problem of finding the factorization of a given composite number
N seems very different and much more difficult. It is the famous FACTORING problem, whose
conjectured hardness underlies many current cryptosystems. Chapter 20 describes Shor's algorithm
to factors integers in polynomial time in the model of *quantum computers.*

We sketch an algorithm showing that PRIMES is in **BPP** (and in fact in **coRP**). For every
number N, and $A \in [N-1]$, define

$$QR_N(A) = \begin{cases} 0 & gcd(A, N) \neq 1 \\ +1 & \begin{array}{l} A \text{ is a } quadratic\ residue \text{ modulo } N \\ \text{That is, } A = B^2 \pmod{N} \text{ for some } B \text{ with } gcd(B, N) = 1 \end{array} \\ -1 & \text{otherwise} \end{cases}$$

We use the following facts that can be proven using elementary number theory:

- For every odd prime N and $A \in [N-1]$, $QR_N(A) = A^{(N-1)/2} \pmod{N}$.

- For every odd N, A define the *Jacobi symbol* $(\frac{N}{A})$ as $\prod_{i=1}^{k} QR_{P_i}(A)$ where $P_1, \ldots, P_k$ are all
  the (not necessarily distinct) prime factors of N (i.e., $N = \prod_{i=1}^{k} P_i$). Then, the Jacobi symbol
  is computable in time $O(\log A \cdot \log N)$.

- For every odd composite N, $\left| \{ A \in [N-1] : gcd(N, A) = 1 \text{ and } (\frac{N}{A}) = A^{(N-1)/2} \} \right| \leq \frac{1}{2} \left| \{ A \in [N-1] : gcd(N, A) = 1 \} \right|$

Together these facts imply a simple algorithm for testing primality of N (which we can assume
without loss of generality is odd): choose a random $1 \leq A < N$, if $gcd(N, A) > 1$ or $(\frac{N}{A}) \neq A^{(N-1)/2}$
$\pmod{N}$ then output "composite", otherwise output "prime". This algorithm will always output
"prime" is N is prime, but if N is composite will output "composite" with probability at least $1/2$.
(Of course this probability can be amplified by repeating the test a constant number of times.)

7.2.2 Polynomial identity testing

So far we described probabilistic algorithms solving problems that have known deterministic poly-
nomial time algorithms. We now describe a problem for which no such deterministic algorithm is
known:

We are given a polynomial with integer coefficients in an implicit form, and we want to decide
whether this polynomial is in fact identically zero. We will assume we get the polynomial in the
form of an *arithmetic circuit.* This is analogous to the notion of a Boolean circuit, but instead of the
operators $\wedge, \vee$ and $\neg$, we have the operators $+, -$ and $\times$. Formally, an n-variable arithmetic circuit
is a directed acyclic graph with the sources labeled by a variable name from the set $x_1, \ldots, x_n$,
and each non-source node has in-degree two and is labeled by an operator from the set $\{+, -, \times\}$.
There is a single sink in the graph which we call the *output* node. The arithmetic circuit defines
a polynomial from $\mathbb{Z}^n$ to $\mathbb{Z}$ by placing the inputs on the sources and computing the value of
each node using the appropriate operator. We define ZEROP to be the set of arithmetic circuits
that compute the identically zero polynomial. Determining membership in ZEROP is also called

polynomial identity testing, since we can reduce the problem of deciding whether two circuits C, C' compute the same polynomial to ZEROP by constructing the circuit D such that $D(x_1, \ldots, x_n) = C(x_1, \ldots, x_n) - C'(x_1, \ldots, x_n)$.

Since expanding all the terms of a given arithmetic circuit can result in a polynomial with exponentially many monomials, it seems hard to decide membership in ZEROP. Surprisingly, there is in fact a simple and efficient probabilistic algorithm for testing membership in ZEROP. At the heart of this algorithm is the following fact, typically known as the Schwartz-Zippel Lemma, whose proof appears in Appendix A (see Lemma A.25):

LEMMA 7.5
Let $p(x_1, x_2, \ldots, x_m)$ be a polynomial of total degree at most d and S is any finite set of integers. When $a_1, a_2, \ldots, a_m$ are randomly chosen with replacement from S, then

$$\Pr[p(a_1, a_2, \ldots, a_m) \neq 0] \geq 1 - \frac{d}{|S|}.$$

Now it is not hard to see that given a size m circuit C on n variables, it defines a polynomial of degree at most 2^m. This suggests the following simple probabilistic algorithm: choose n numbers $x_1, \ldots, x_n$ from 1 to $10 \cdot 2^m$ (this requires $O(n \cdot m)$ random bits), evaluate the circuit C on $x_1, \ldots, x_n$ to obtain an output y and then accept if $y = 0$, and reject otherwise. Clearly if $C \in$ ZEROP then we always accept. By the lemma, if $C \notin$ ZEROP then we will reject with probability at least $9/10$.

However, there is a problem with this algorithm. Since the degree of the polynomial represented by the circuit can be as high as 2^m, the output y and other intermediate values arising in the computation may be as large as $(10 \cdot 2^m)^{2^m}$ — this is a value that requires exponentially many bits just to describe!

We solve this problem using the technique of *fingerprinting*. The idea is to perform the evaluation of C on $x_1, \ldots, x_n$ modulo a number k that is chosen at random in $[2^{2m}]$. Thus, instead of computing $y = C(x_1, \ldots, x_n)$, we compute the value $y \pmod k$. Clearly, if $y = 0$ then $y \pmod k$ is also equal to 0. On the other hand, we claim that if $y \neq 0$, then with probability at least $\delta = \frac{1}{10m}$, k does not divide y. (This will suffice because we can repeat this procedure $O(1/\delta)$ times to ensure that if $y \neq 0$ then we find this out with probability at lest $9/10$.) Indeed, assume that $y \neq 0$ and let $\mathcal{S} = \{p_1, \ldots, p_\ell\}$ denote set of the distinct prime factors of y. It is sufficient to show that with probability at δ, the number k will be a prime number not in $\mathcal{S}$. Yet, by the prime number theorem, the probability that k is prime is at least $\frac{1}{5m} = 2\delta$. Also, since y can have at most $\log y \leq 5m2^m$ distinct factors, the probability that k is in $\mathcal{S}$ is less than $\frac{5m2^m}{2^{2m}} \ll \frac{1}{10m} = \delta$. Hence by the union bound, with probability at least δ, k will not divide y.

7.2.3 Testing for perfect matching in a bipartite graph.

If $G = (V_1, V_2, E)$ is the bipartite graph where $|V_1| = |V_2|$ and $E \subseteq V_1 \times V_2$ then a *perfect matching* is some $E' \subseteq E$ such that every node appears exactly once among the edges of E'. Alternatively, we may think of it as a permutation σ on the set $\{1, 2, \ldots, n\}$ (where $n = |V_1|$) such that for each $i \in \{1, 2, \ldots, n\}$, the pair $(i, \sigma(i))$ is an edge. Several deterministic algorithms are known for detecting if a perfect matching exists. Here we describe a very simple randomized algorithm (due to Lovász) using the Schwartz-Zippel lemma.

Consider the $n \times n$ matrix X (where $n = |V_1| = |V_2|$) whose (i,j) entry X_{ij} is the variable x_{ij} if $(i,j) \in E$ and 0 otherwise. Recall that the determinant of matrix $det(X)$ is

$$det(X) = \sum_{\sigma \in S_n} (-1)^{\text{sign}(\sigma)} \prod_{i=1}^{n} X_{i,\sigma(i)}, \tag{1}$$

where S_n is the set of all permutations of $\{1, 2, \ldots, n\}$. Note that every permutation is a potential perfect matching, and the corresponding monomial in $det(X)$ is nonzero iff this perfect matching exists in G. Thus the graph has a perfect matching iff $det(X) \neq 0$.

Now observe two things. First, the polynomial in (1) has $|E|$ variables and total degree at most n. Second, even though this polynomial may be of exponential size, for every setting of values to the X_{ij} variables it can be efficiently evaluated, since computing the determinant of a matrix with integer entries is a simple polynomial-time computation (actually, even in $\mathbf{NC}^2$).

This leads us to Lovász's randomized algorithm: pick random values for X_{ij}'s from $[1, \ldots, 2n]$, substitute them in X and compute the determinant. If the determinant is nonzero, output "accept" else output "reject." The advantage of the above algorithm over classical algorithms is that it can be implemented by a randomized $\mathbf{NC}$ circuit, which means (by the ideas of Section 6.5.1) that it has a fast implementation on parallel computers.

7.3 One-sided and zero-sided error: RP, coRP, ZPP

The class $\mathbf{BPP}$ captured what we call probabilistic algorithms with *two sided* error. That is, it allows the machine M to output (with some small probability) both 0 when $x \in L$ and 1 when $x \notin L$. However, many probabilistic algorithms have the property of *one sided* error. For example if $x \notin L$ they will *never* output 1, although they may output 0 when $x \in L$. This is captured by the definition of $\mathbf{RP}$.

DEFINITION 7.6
$\mathbf{RTIME}(t(n))$ contains every language L for which there is a is a probabilistic TM M running in $t(n)$ time such that

$$x \in L \Rightarrow \mathbf{Pr}[M \text{ accepts } x] \geq \frac{2}{3}$$
$$x \notin L \Rightarrow \mathbf{Pr}[M \text{ accepts } x] = 0$$

We define $\mathbf{RP} = \cup_{c>0}\mathbf{RTIME}(n^c)$.

Note that $\mathbf{RP} \subseteq \mathbf{NP}$, since every accepting branch is a "certificate" that the input is in the language. In contrast, we do not know if $\mathbf{BPP} \subseteq \mathbf{NP}$. The class $\mathbf{coRP} = \{L \mid \overline{L} \in \mathbf{RP}\}$ captures one-sided error algorithms with the error in the "other direction" (i.e., may output 1 when $x \notin L$ but will never output 0 if $x \in L$).

For a PTM M, and input x, we define the random variable $T_{M,x}$ to be the running time of M on input x. That is, $\Pr[T_{M,x} = T] = p$ if with probability p over the random choices of M on input x, it will halt within T steps. We say that M has *expected running time* $T(n)$ if the expectation $\mathsf{E}[T_{M,x}]$ is at most $T(|x|)$ for every $x \in \{0,1\}^*$. We now define PTMs that never err (also called "zero error" machines):

DEFINITION 7.7
The class $\mathbf{ZTIME}(T(n))$ contains all the languages L for which there is an expected-time $O(T(n))$ machine that never errs. That is,

$$x \in L \Rightarrow \mathbf{Pr}[M \text{ accepts } x] = 1$$
$$x \notin L \Rightarrow \mathbf{Pr}[M \text{ halts without accepting on } x] = 1$$

We define $\mathbf{ZPP} = \cup_{c>0}\mathbf{ZTIME}(n^c)$.

The next theorem ought to be slightly surprising, since the corresponding statement for nondeterminism is open; i.e., whether or not $\mathbf{P} = \mathbf{NP} \cap \mathbf{coNP}$.

THEOREM 7.8
$\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}$.

We leave the proof of this theorem to the reader (see Exercise 4). To summarize, we have the following relations between the probabilistic complexity classes:

$$\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}$$
$$\mathbf{RP} \subseteq \mathbf{BPP}$$
$$\mathbf{coRP} \subseteq \mathbf{BPP}$$

7.4 The robustness of our definitions

When we defined $\mathbf{P}$ and $\mathbf{NP}$, we argued that our definitions are robust and were likely to be the same for an alien studying the same concepts in a faraway galaxy. Now we address similar issues for probabilistic computation.

7.4.1 Role of precise constants, error reduction.

The choice of the constant $2/3$ seemed pretty arbitrary. We now show that we can replace $2/3$ with any constant larger than $1/2$ and in fact even with $1/2 + n^{-c}$ for a constant $c > 0$.

LEMMA 7.9
For $c > 0$, let $\mathbf{BPP}_{n^{-c}}$ denote the class of languages L for which there is a polynomial-time PTM M satisfying $\Pr[M(x) = L(x)] \geq 1/2 + |x|^{-c}$ for every $x \in \{0,1\}^*$. Then $\mathbf{BPP}_{n^{-c}} = \mathbf{BPP}$.

Since clearly $\mathbf{BPP} \subseteq \mathbf{BPP}_{n^{-c}}$, to prove this lemma we need to show that we can transform a machine with success probability $1/2 + n^{-c}$ into a machine with success probability $2/3$. We do this by proving a much stronger result: we can transform such a machine into a machine with success probability exponentially close to one!

THEOREM 7.10 (ERROR REDUCTION)
Let $L \subseteq \{0,1\}^*$ be a language and suppose that there exists a polynomial-time PTM M such that for every $x \in \{0,1\}^*$, $\Pr[M(x) = L(x)] \geq \frac{1}{2} + |x|^{-c}$.
Then for every constant $d > 0$ there exists a polynomial-time PTM M' such that for every $x \in \{0,1\}^*$, $\Pr[M'(x) = L(x)] \geq 1 - 2^{-|x|^d}$.

PROOF: The machine M' is quite simple: *for every input $x \in \{0,1\}^*$, run $M(x)$ for k times obtaining k outputs $y_1, \ldots, y_k \in \{0,1\}$, where $k = 8|x|^{2d+c}$. If the majority of these values are 1 then accept, otherwise reject.*

To analyze this machine, define for every $i \in [k]$ the random variable X_i to equal 1 if $y_i = L(x)$ and to equal 0 otherwise. Note that $X_1, \ldots, X_k$ are independent Boolean random variables with $\mathsf{E}[X_i] = \Pr[X_i = 1] \geq 1/2 + n^{-c}$ (where $n = |x|$). The Chernoff bound (see Theorem A.18 in Appendix A) implies the following corollary:

COROLLARY 7.11
Let $X_1, \ldots, X_k$ be independent identically distributed Boolean random variables, with $\Pr[X_i = 1] = p$ for every $1 \leq i \leq k$. Let $\delta \in (0,1)$. Then,

$$\Pr\left[\left| \frac{1}{k} \sum_{i=1}^{k} X_i - p \right| > \delta \right] < e^{-\frac{\delta^2}{4} pk}$$

In our case $p = 1/2 + n^{-c}$, and plugging in $\delta = n^{-c}/2$, the probability we output a wrong answer is bounded by

$$\Pr[\frac{1}{n} \sum_{i=1}^{k} X_i \leq 1/2 + n^{-c}/2] \leq e^{-\frac{1}{4n^{-2c}} \frac{1}{2} 8 n^{2c+d}} \leq 2^{-n^d}$$

■

A similar result holds for the class **RP**. In fact, there we can replace the constant $2/3$ with every positive constant, and even with values as low as n^{-c}. That is, we have the following result:

THEOREM 7.12
Let $L \subseteq \{0,1\}^*$ such that there exists a polynomial-time PTM M satisfying for every $x \in \{0,1\}^*$:
(1) If $x \in L$ then $\Pr[M(x) = 1)] \geq n^{-c}$ and (2) if $x \notin L$, then $\Pr[M(x) = 1] = 0$.

Then for every $d > 0$ there exists a polynomial-time PTM M' such that for every $x \in \{0,1\}^*$,
(1) if $x \in L$ then $\Pr[M'(x) = 1] \geq 1 - 2^{-n^d}$ and (2) if $x \notin L$ then $\Pr[M'(x) = 1] = 0$.

These results imply that we can take a probabilistic algorithm that succeeds with quite modest probability and transform it into an algorithm that succeeds with overwhelming probability. In fact, even for moderate values of n an error probability that is of the order of 2^{-n} is so small that for all practical purposes, probabilistic algorithms are just as good as deterministic algorithms.

If the original probabilistic algorithm used m coins, then the error reduction procedure we use (run k independent trials and output the majority answer) takes $O(m \cdot k)$ random coins to reduce the error to a value exponentially small in k. It is somewhat surprising that we can in fact do better, and reduce the error to the same level using only $O(m + k)$ random bits (see Section 7.5).

NOTE 7.13 (THE CHERNOFF BOUND)
The Chernoff bound is extensively used (sometimes under different names) in many areas of computer science and other sciences. A typical scenario is the following: there is a universe $\mathcal{U}$ of objects, a fraction μ of them have a certain property, and we wish to estimate μ. For example, in the proof of Theorem 7.10 the universe was the set of 2^m possible coin tosses of some probabilistic algorithm and we wanted to know how many of them cause the algorithm to accept its input. Another example is that $\mathcal{U}$ may be the set of all the citizens of the United States, and we wish to find out how many of them approve of the current president.
A natural approach to compute the fraction μ is to *sample* n members of the universe independently at random, find out the number k of the sample's members that have the property and to estimate that μ is k/n. Of course, it may be quite possible that 10% of the population supports the president, but in a sample of 1000 we will find 101 and not 100 such people, and so we set our goal only to estimate μ up to an *error* of $\pm\epsilon$ for some $\epsilon > 0$. Similarly, even if only 10% of the population have a certain property, we may be extremely unlucky and select only people having it for our sample, and so we allow a small *probability of failure* δ that our estimate will be off by more than ϵ. The natural question is *how many samples do we need to estimate μ up to an error of $\pm\epsilon$ with probability at least $1 - \delta$?* The Chernoff bound tells us that (considering μ as a constant) this number is $O(\log(1/\delta)/\epsilon^2)$.
This implies that if we sample n elements, then the probability that the number k having the property is $\rho\sqrt{n}$ far from μn decays *exponentially* with ρ: that is, this probability has the famous "bell curve" shape:
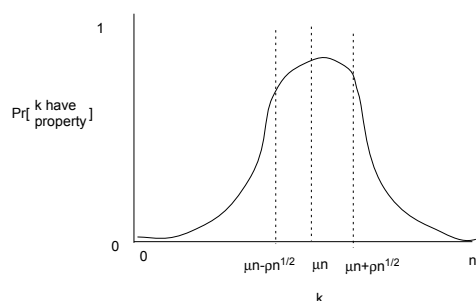
We will use this exponential decay phenomena several times in this book, starting with the proof of Theorem 7.17, showing that $\mathbf{BPP} \subseteq \mathbf{P}/\text{poly}$.

7.4.2 Expected running time versus worst-case running time.

When defining **RTIME**$(T(n))$ and **BPTIME**$(T(n))$ we required the machine to halt in $T(n)$ time regardless of its random choices. We could have used *expected* running time instead, as in the definition of **ZPP** (Definition 7.7). It turns out this yields an equivalent definition: we can add a time counter to a PTM M whose expected running time is $T(n)$ and ensure it always halts after at most $100T(n)$ steps. By Markov's inequality (see Lemma A.10), the probability that M runs for more than this time is at most $1/100$. Thus by halting after $100T(n)$ steps, the acceptance probability is changed by at most $1/100$.

7.4.3 Allowing more general random choices than a fair random coin.

One could conceive of real-life computers that have a "coin" that comes up heads with probability ρ that is not $1/2$. We call such a coin a ρ-coin. Indeed it is conceivable that for a random source based upon quantum mechanics, ρ is an irrational number, such as $1/e$. Could such a coin give probabilistic algorithms new power? The following claim shows that it will not.

LEMMA 7.14
A coin with $\Pr[Heads] = \rho$ *can be simulated by a PTM in expected time* $O(1)$ *provided the ith bit of* ρ *is computable in* $poly(i)$ *time.*

PROOF: Let the binary expansion of ρ be $0.p_1p_2p_3\ldots$. The PTM generates a sequence of random bits $b_1, b_2, \ldots$, one by one, where b_i is generated at step i. If $b_i < p_i$ then the machine outputs "heads" and stops; if $b_i > p_i$ the machine outputs "tails" and halts; otherwise the machine goes to step $i + 1$. Clearly, the machine reaches step $i + 1$ iff $b_j = p_j$ for all $j \leq i$, which happens with probability $1/2^i$. Thus the probability of "heads" is $\sum_i p_i \frac{1}{2^i}$, which is exactly ρ. Furthermore, the expected running time is $\sum_i i^c \cdot \frac{1}{2^i}$. For every constant c this infinite sum is upperbounded by another constant (see Exercise 1). ∎

Conversely, probabilistic algorithms that only have access to ρ-coins do not have less power than standard probabilistic algorithms:

LEMMA 7.15 (VON-NEUMANN)
A coin with $\Pr[Heads] = 1/2$ *can be simulated by a probabilistic TM with access to a stream of* ρ-*biased coins in expected time* $O(\frac{1}{\rho(1-\rho)})$.

PROOF: We construct a TM M that given the ability to toss ρ-coins, outputs a $1/2$-coin. The machine M tosses pairs of coins until the first time it gets two different results one after the other. If these two results were first "heads" and then "tails", M outputs "heads". If these two results were first "tails" and then "heads", M outputs "tails". For each pair, the probability we get two "heads" is ρ^2, the probability we get two "tails" is $(1 - \rho)^2$, the probability we get "head" and then "tails" is $\rho(1 - \rho)$, and the probability we get "tails" and then "head" is $(1 - \rho)\rho$. We see that the probability we halt and output in each step is $2\rho(1 - \rho)$, and that conditioned on this, we do indeed output either "heads" or "tails" with the same probability. Note that we did not need to know ρ to run this simulation. ∎

Weak random sources.   Physicists (and philosophers) are still not completely certain that randomness exists in the world, and even if it does, it is not clear that our computers have access to an endless stream of independent coins. Conceivably, it may be the case that we only have access to a source of *imperfect* randomness, that although unpredictable, does not consist of independent coins. As we will see in Chapter 16, we do know how to simulate probabilistic algorithms designed for perfect independent 1/2-coins even using such a weak random source.

7.5   Randomness efficient error reduction.

In Section 7.4.1 we saw how we can reduce error of probabilistic algorithms by running them several time using independent random bits each time. Ideally, one would like to be frugal with using randomness, because good quality random number generators tend to be slower than the rest of the computer. Surprisingly, the error reduction can be done just as effectively without using truly independent runs, and "recycling" the random bits. Now we outline this idea; a much more general theory will be later presented in Chapter 16.

The main tool we use is *expander* graphs. Expander graphs have played a crucial role in numerous computer science applications, including routing networks, error correcting codes, hardness of approximation and the PCP theorem, derandomization, and more. Expanders can be defined in several roughly equivalent ways. One is that these are graphs where every set of vertices has a very large boundary. That is, for every subset S of vertices, the number of S's neighbors outside S is (up to a constant factor) roughly equal to the number of vertices inside S. (Of course this condition cannot hold if S is too big and already contains almost all of the vertices in the graph.) For example, the n by n grid (where a vertex is a pair (i, j) and is connected to the four neighbors $(i \pm 1, j \pm 1)$) is *not* an expander, as any k by k square (which is a set of size k^2) in this graph only has a boundary of size $O(k)$ (see Figure 7.1).
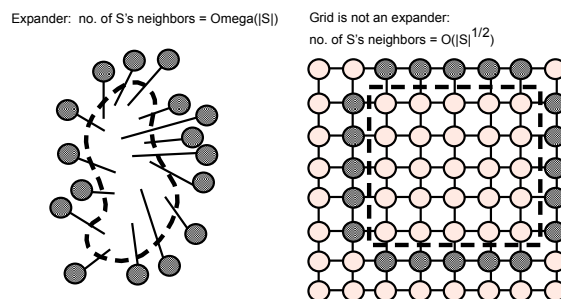
Figure 7.1: In a *combinatorial expander*, every subset S of the vertices that is not too big has at least $\Omega(|S|)$ neighbors outside the set. The grid (and every other planar graph) is not a combinatorial expander as a $k \times k$ square in the grid has only $O(k)$ neighbors outside it.

We will not precisely define expanders now (but see Section 7.B at the end of the chapter). However, an *expander graph family* is a sequence of graphs $\{G_N\}_{N \in \mathbb{N}}$ such for every N, G_N is an N-vertex D-degree graph for some constant D. Deep mathematics (and more recently, simpler mathematics) has been used to construct expander graphs. These constructions yield algorithms

that, given the binary representation of N and an index of a node in G_N, can produce the indices of the D neighbors of this node in poly$(\log N)$ time.

We illustrate the error reduction procedure by showing how we transform an **RP** algorithm that outputs the right answer with probability $1/2$ into an algorithm that outputs the right answer with probability $1 - 2^{-\Omega(k)}$. The idea is simple: let x be an input, and suppose we have an algorithm M using m coins such that if $x \in L$ then $\Pr_{r \in_R \{0,1\}^m}[M(x,r) = 1] \geq 1/2$ and if $x \notin L$ then $M(x,r) = 0$ for every r. Let $N = 2^m$ and let G_N be an N-vertex expander family. We use m coins to select a random vertex v from G_N, and then use $\log Dk$ coins to take a $k-1$-*step random walk* from v on G_N. That is, at each step we choose a random number i in $[D]$ and move from the current vertex to its ith neighbor. Let $v_1, \ldots, v_k$ be the vertices we encounter along this walk (where $v_1 = v$). We can treat these vertices as elements of $\{0,1\}^m$ and run the machine M on input x with all of these coins. If even one of these runs outputs 1, then output 1. Otherwise, output 0. It can be shown that if less than half of the r's cause M to output 0, then the probability that the walk is fully contained in these "bad" r's is exponentially small in k.

We see that what we need to prove is the following theorem:

THEOREM 7.16
Let G be an expander graph of N vertices and B a subset of G's vertices of size at most βN, where $\beta < 1$. Then, the probability that a k-vertex random walk is fully contained in B is at most ρ^k, where $\rho < 1$ is a constant depending only on β (and independent of k).

Theorem 7.16 makes intuitive sense, as in an expander graph a constant fraction of the edges adjacent to vertices of B will have the other vertex in B's complement, and so it seems that at each step we will have a constant probability to leave B. However, its precise formulation and analysis takes some care, and is done at the end of the chapter in Section 7.B.

Intuitively, We postpone the full description of the error reduction procedure and its analysis to Section 7.B.

7.6   **BPP** $\subseteq$ **P**/poly

Now we show that all **BPP** languages have polynomial sized circuits. Together with Theorem **??** this implies that if 3SAT $\in$ **BPP** then **PH** $= \mathbf{\Sigma}_2^p$.

THEOREM 7.17 (ADLEMAN)
BPP $\subseteq$ **P**/poly.

PROOF: Sup-

pose $L \in$ **BPP**, then by the alternative definition of **BPP** and the error reduction procedure of Theorem 7.10, there exists a TM M that on inputs of size n uses m random bits and satisfies

$$x \in L \Rightarrow \mathbf{Pr}_r[M(x,r) \text{ accepts }] \geq 1 - 2^{-(n+2)}$$
$$x \notin L \Rightarrow \mathbf{Pr}_r[M(x,r) \text{ accepts }] \leq 2^{-(n+2)}$$

(Such a machine exists by the error reduction arguments mentioned earlier.)

Say that an $r \in \{0,1\}^m$ is *bad* for an input $x \in \{0,1\}^n$ if $M(x,r)$ is an incorrect answer, otherwise we say its *good* for x. For every x, at most $2 \cdot 2^m/2^{(n+2)}$ values of r are bad for x. Adding over all $x \in \{0,1\}^n$, we conclude that at most $2^n \times 2^m/2^{(n+1)} = 2^m/2$ strings r are bad for *some* x. In other words, at least $2^m - 2^m/2$ choices of r are good for *every* $x \in \{0,1\}^n$. Given a string r_0 that is good for every $x \in \{0,1\}^n$, we can hardwire it to obtain a circuit C (of size at most quadratic in the running time of M) that on input x outputs $M(x, r_0)$. The circuit C will satisfy $C(x) = L(x)$ for every $x \in \{0,1\}^n$. ∎

7.7 BPP is in PH

At first glance, **BPP** seems to have nothing to do with the polynomial hierarchy, so the next theorem is somewhat surprising.

THEOREM 7.18 (SIPSER-GÁCS)
$$\mathbf{BPP} \subseteq \mathbf{\Sigma}_2^p \cap \mathbf{\Pi}_2^p$$

PROOF: It is enough to prove that $\mathbf{BPP} \subseteq \mathbf{\Sigma}_2^p$ because **BPP** is closed under complementation (i.e., $\mathbf{BPP} = \mathbf{coBPP}$).

Suppose $L \in \mathbf{BPP}$. Then by the alternative definition of **BPP** and the error reduction procedure of Theorem 7.10 there exists a polynomial-time deterministic TM M for L that on inputs of length n uses $m = \text{poly}(n)$ random bits and satisfies

$$x \in L \Rightarrow \mathbf{Pr}_r\left[M(x,r) \text{ accepts }\right] \geq 1 - 2^{-n}$$
$$x \notin L \Rightarrow \mathbf{Pr}_r\left[M(x,r) \text{ accepts }\right] \leq 2^{-n}$$

For $x \in \{0,1\}^n$, let S_x denote the set of r's for which M accepts the input pair (x,r). Then either $|S_x| \geq (1 - 2^{-n})2^m$ or $|S_x| \leq 2^{-n}2^m$, depending on whether or not $x \in L$. We will show how to check, using two alternations, which of the two cases is true.
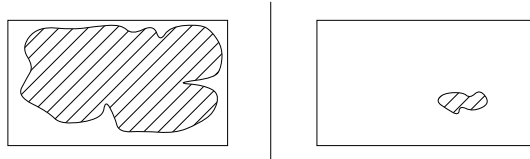
Figure 7.2: There are only two possible sizes for the set of r's such that $M(x,r)$ =Accept: either this set is almost all of $\{0,1\}^m$ or a tiny fraction of $\{0,1\}^m$. In the former case, a few random "shifts" of this set are quite likely to cover all of $\{0,1\}^m$. In the latter case the set's size is so small that a few shifts cannot cover $\{0,1\}^m$

For $k = \frac{m}{n} + 1$, let $U = \{u_1, \ldots, u_k\}$ be a set of k strings in $\{0,1\}^m$. We define G_U to be a graph with vertex set $\{0,1\}^m$ and edges (r,s) for every r, s such that $r = s + u_i$ for some $i \in [k]$

(where $+$ denotes vector addition modulo 2, or equivalently, bitwise XOR). Note that the degree of G_U is k. For a set $S \subseteq \{0,1\}^m$, define $\Gamma_U(S)$ to be all the neighbors of S in the graph G_U. That is, $r \in \Gamma_U(S)$ if there is an $s \in S$ and $i \in [k]$ such that $r = s + u_i$.

Claim 1: For every set $S \subseteq \{0,1\}^m$ with $|S| \leq 2^{m-n}$ and every set U of size k, it holds that $\Gamma_U(S) \neq \{0,1\}^m$. Indeed, since Γ_U has degree k, it holds that $|\Gamma_U(S)| \leq k|S| < 2^m$.

Claim 2: For every set $S \subseteq \{0,1\}^m$ with $|S| \geq (1 - 2^{-n})2^m$ there exists a set U of size k such that $\Gamma_U(S) = \{0,1\}^m$. We show this by the probabilistic method, by proving that for every S, if we choose U at random by taking k random strings $u_1, \ldots, u_k$, then $\Pr[\Gamma_U(S) = \{0,1\}^m] > 0$. Indeed, for $r \in \{0,1\}^m$, let B_r denote the "bad event" that r is not in $\Gamma_U(S)$. Then, $B_r = \cap_{i \in [k]} B_r^i$ where B_r^i is the event that $r \notin S + u_i$, or equivalently, that $r + u_i \notin S$ (using the fact that modulo 2, $a + b = c \Leftrightarrow a = c + b$). Yet, $r + u_i$ is a uniform element in $\{0,1\}^m$, and so it will be in S with probability at least $1 - 2^{-n}$. Since $B_r^1, \ldots, B_r^k$ are independent, the probability that B_r happens is at most $(1 - 2^{-n})^k < 2^{-m}$. By the union bound, the probability that $\Gamma_U(S) \neq \{0,1\}^m$ is bounded by $\sum_{r \in \{0,1\}^m} \Pr[B_r] < 1$.

Together Claims 1 and 2 show $x \in L$ if and only if the following statement is true

$$\exists u_1, \ldots, u_k \in \{0,1\}^m \ \forall r \in \{0,1\}^m \ \bigvee_{i=1}^{k} M(x, r \oplus u_i) \text{accepts}$$

thus showing $L \in \mathbf{\Sigma_2}$. ∎

7.8 State of our knowledge about BPP

We know that $\mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{P}/poly$, and furthermore, that $\mathbf{BPP} \subseteq \mathbf{\Sigma}_2^p \cap \mathbf{\Pi}_2^p$ and so if $\mathbf{NP} = p$ then $\mathbf{BPP} = \mathbf{P}$. As mentioned above, there are complexity-theoretic reasons to strongly believe that $\mathbf{BPP} \subseteq \mathbf{DTIME}(2^\epsilon)$ for every $\epsilon > 0$, and in fact to reasonably suspect that $\mathbf{BPP} = \mathbf{P}$ (see Chapters 16 and 17). However, currently we are not even able to rule out that $\mathbf{BPP} = \mathbf{NEXP}$!

Complete problems for BPP?

Though a very natural class, **BPP** behaves differently in some ways from other classes we have seen. For example, we know of no complete languages for it (under deterministic polynomial time reductions). One reason for this difficulty is that the defining property of **BPTIME** machines is *semantic*, namely, that for *every* string they either accept with probability at least $2/3$ or reject with probability at least $1/3$. Given the description of a Turing machine M, testing whether it has this property is undecidable. By contrast, the defining property of an NDTM is *syntactic:* given a string it is easy to determine if it is a valid encoding of an NDTM. Completeness seems easier to define for syntactically defined classes than for semantically defined ones. For example, consider the following natural attempt at a **BPP**-complete language: $L = \{\langle M, x \rangle : \Pr[M(x) = 1] \geq 2/3\}$. This language is indeed **BPP**-hard but is not known to be in **BPP**. In fact, it is not in any level of the polynomial hierarchy unless the hierarchy collapses. We note that if, as believed, $\mathbf{BPP} = \mathbf{P}$, then **BPP** does have a complete problem. (One can sidestep some of the above issues by using *promise* problems instead of languages, but we will not explore this.)

Does BPTIME have a hierarchy theorem?

Is $\mathbf{BPTIME}(n^c)$ contained in $\mathbf{BPTIME}(n)$ for some $c > 1$? One would imagine not, and this seems as the kind of result we should be able to prove using the tools of Chapter 3. However currently we are even unable to show that $\mathbf{BPTIME}(n^{\log^2 n})$ (say) is not in $\mathbf{BPTIME}(n)$. The standard diagonalization techniques fail, for similar reasons as the ones above. However, recently there has been some progress on obtaining hierarchy theorem for some closely related classes (see notes).

7.9 Randomized reductions

Since we have defined randomized algorithms, it also makes sense to define a notion of randomized reduction between two languages. This proves useful in some complexity settings (e.g., see Chapters 9 and 8).

DEFINITION 7.19
Language A reduces to language B under a randomized polynomial time reduction, denoted $A \leq_r B$, if there is a probabilistic TM M such that for every $x \in \{0,1\}^*$, $\Pr[B(M(x)) = A(x)] \geq 2/3$.

We note that if $A \leq_r B$ and $B \in \mathbf{BPP}$ then $A \in \mathbf{BPP}$. This alerts us to the possibility that we could have defined $\mathbf{NP}$-completeness using randomized reductions instead of deterministic reductions, since arguably $\mathbf{BPP}$ is as good as $\mathbf{P}$ as a formalization of the notion of efficient computation. Recall that the Cook-Levin theorem shows that $\mathbf{NP}$ may be defined as the set $\{L : L \leq_p \mathsf{3SAT}\}$. The following definition is analogous.

DEFINITION 7.20 ($\mathbf{BP} \cdot \mathbf{NP}$)
$\mathbf{BP} \cdot \mathbf{NP} = \{L : L \leq_r \mathsf{3SAT}\}$.

We explore the properties of $\mathbf{BP}\cdot\mathbf{NP}$ in the exercises, including whether or not $\overline{\mathsf{3SAT}} \in \mathbf{BP}\cdot\mathbf{NP}$.

One interesting application of randomized reductions will be shown in Chapter 9, where we present a (variant of a) randomized reduction from $\mathsf{3SAT}$ to the solving special case of $\mathsf{3SAT}$ where we are guaranteed that the formula is either unsatisfiable or has a *single unique* satisfying assignment.

7.10 Randomized space-bounded computation

A PTM is said to work in space $S(n)$ if every branch requires space $O(S(n))$ on inputs of size n and terminates in $2^{O(S(n))}$ time. Recall that the machine has a read-only input tape, and the work space only cell refers only to its read/write work tapes. As a PTM it has two transition functions that are applied with equal probability. The most interesting case is when the work tape has $O(\log n)$ size. The classes $\mathbf{BPL}$ and $\mathbf{RL}$ are the two-sided error and one-sided error probabilistic analogs of the class $\mathbf{L}$ defined in Chapter 4.

DEFINITION 7.21 ([)
The classes **BPL** and **RL**] A language L is in **BPL** if there is an $O(\log n)$-space
probabilistic TM M such that $\Pr[M(x) = L(x)] \geq 2/3$.
A language L is in **RL** if there is an $O(\log n)$-space probabilistic TM M such that
if $x \in L$ then $\Pr[M(x) = 1] \geq 2/3$ and if $x \notin L$ then $\Pr[M(x) = 1] = 0$.

The reader can verify that the error reduction procedure described in Chapter 7 can be imple-
mented with only logarithmic space overhead, and hence also in these definitions the choice of the
precise constant is not significant. We note that $\mathbf{RL} \subseteq \mathbf{NL}$, and thus $\mathbf{RL} \subseteq \mathbf{P}$. The exercises ask
you to show that $\mathbf{BPL} \subseteq \mathbf{P}$ as well.

One famous **RL**-algorithm is the algorithm to solve UPATH: the restriction of the **NL**-complete
PATH problem (see Chapter 4) to undirected graphs. That is, given an n-vertex undirected graph
G and two vertices s and t, determine whether s is connected to t in G.

THEOREM 7.22 ([AKL$^+$79])
UPATH $\in$ **RL**.

The algorithm for UPATH is actually very simple: take a random walk of length n^3 starting
from s. That is, initialize the variable v to the vertex s and in each step choose a random neighbor
u of v, and set $v \leftarrow u$. Accept iff the walk reaches t within n^3 steps. Clearly, if s is not connected to
t then the algorithm will never accept. It can be shown that if s is connected to t then the expected
number of steps it takes for a walk from s to hit t is at most $\frac{4}{27}n^3$ and hence our algorithm will accept
with probability at least $\frac{3}{4}$. We defer the analysis of this algorithm to the end of the chapter at
Section 7.A, where we will prove that a somewhat larger walk suffices to hit t with good probability
(see also Exercise 9).

In Chapter 16 we show a recent *deterministic* logspace algorithm for the same problem. It is
known that **BPL** (and hence also **RL**) is contained in $\mathbf{SPACE}(\log^{3/2} n)$. In Chapter 16 we will
see a somewhat weaker result: a simulation of **BPL** in $\log^2 n$ space and polynomial time.

WHAT HAVE WE LEARNED?

- The class **BPP** consists of languages that can be solved by a probabilistic polynomial-time algorithm. The probability is only over the algorithm's coins and not the choice of input. It is arguably a better formalization of efficient computation than **P**.

- **RP**, **coRP** and **ZPP** are subclasses of **BPP** corresponding to probabilistic algorithms with one-sided and "zero-sided" error.

- Using repetition, we can considerably amplify the success probability of probabilistic algorithms.

- We only know that **P** ⊆ **BPP** ⊆ **EXP**, but we suspect that **BPP** = **P**.

- **BPP** is a subset of both **P**/poly and **PH**. In particular, the latter implies that if **NP** = **P** then **BPP** = **P**.

- Randomness is used in complexity theory in many contexts beyond **BPP**. Two examples are randomized reductions and randomized logspace algorithms, but we will see many more later.

Chapter notes and history

Early researchers realized the power of randomization since their computations —e.g., for design of nuclear weapons— used probabilistic tools such as Monte Carlo simulations. Papers by von Neumann [von61] and de Leeuw et al. [LMSS56] describe probabilistic Turing machines. The definitions of **BPP**, **RP** and **ZPP** are from Gill [Gil77]. (In an earlier conference paper [Gil74], Gill studies similar issues but seems to miss the point that a practical algorithm for deciding a language must feature a *gap* between the acceptance probability in the two cases.)

The algorithm used to show PRIMES is in **coRP** is due to Solovay and Strassen [SS77]. Another primality test from the same era is due to Rabin [Rab80]. Over the years, better tests were proposed. In a recent breakthrough, Agrawal, Kayal and Saxena finally proved that PRIMES ∈ **P**. Both the probabilistic and deterministic primality testing algorithms are described in Shoup's book [**?**].

Lovász's randomized **NC** algorithm [Lov79] for deciding the *existence* of perfect matchings is unsatisfying in the sense that when it outputs "Accept," it gives no clue how to find a matching! Subsequent probabilistic **NC** algorithms can find a perfect matching as well; see [KUW86, MVV87].

BPP ⊆ **P**/poly is from Adelman [Adl78]. **BPP** ⊆ **PH** is due to Sipser [Sip83], and the stronger form **BPP** ⊆ $\Sigma_2^p \cap \Pi_2^p$ is due to P. Gács. Recent work [] shows that **BPP** is contained in classes that are seemingly weaker than $\Sigma_2^p \cap \Pi_2^p$.

Even though a hierarchy theorem for **BPP** seems beyond our reach, there has been some success in showing hierarchy theorems for the seemingly related class **BPP**/1 (i.e., **BPP** with a single bit of nonuniform advice) [Bar02, **?**, **?**].

Readers interested in randomized algorithms are referred to the books by Mitzenmacher and Upfal [MU05] and Motwani and Raghavan [MR95].

STILL A LOT MISSING

Expanders were well-studied for a variety of reasons in the 1970s but their application to pseudorandomness was first described by Ajtai, Komlos, and Szemeredi [AKS87]. Then Cohen-Wigderson [CW89] and Impagliazzo-Zuckerman (1989) showed how to use them to "recycle" random bits as described in Section 7.B.3. The upcoming book by Hoory, Linial and Wigderson (draft available from their web pages) provides an excellent introduction to expander graphs and their applications.

The explicit construction of expanders is due to Reingold, Vadhan and Wigderson [RVW00], although we chose to present it using the replacement product as opposed to the closely related zig-zag product used there. The deterministic logspace algorithm for undirected connectivity is due to Reingold [?].

Exercises

§1 Show that for every $c > 0$, the following infinite sum is finite:

$$\sum_{i \geq 1} \frac{i^c}{2^i}.$$

§2 Show, given input the numbers a, n, p (in binary representation), how to compute $a^n (\bmod p)$ in polynomial time.

Hint: use the binary representation of n and repeated squaring.

§3 Let us study to what extent Claim **??** truly needs the assumption that ρ is efficiently computable. Describe a real number ρ such that given a random coin that comes up "Heads" with probability ρ, a Turing machine can decide an undecidable language in polynomial time.

Hint: think of the real number ρ as an advice string. How can its bits be recovered?

§4 Show that $\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}$.

§5 A nondeterministic circuit has two inputs x, y. We say that it accepts x iff there exists y such that $C(x, y) = 1$. The size of the circuit is measured as a function of $|x|$. Let $\mathbf{NP}/poly$ be the languages that are decided by polynomial size nondeterministic circuits. Show that $BP \cdot \mathbf{NP} \subseteq \mathbf{NP}/poly$.

§6 Show using ideas similar to the Karp-Lipton theorem that if $\overline{3\mathsf{SAT}} \in BP \cdot \mathbf{NP}$ then $\mathbf{PH}$ collapses to $\mathbf{\Sigma}_3^p$. (Combined with above, this shows it is unlikely that $3\mathsf{SAT} \leq_r \overline{3\mathsf{SAT}}$.)

§7 Show that $\mathbf{BPL} \subseteq \mathbf{P}$

Hint: try to compute the probability that the machine ends up
in the accept configuration using either dynamic programming or
matrix multiplication.

§8 Show that the random walk idea for solving connectivity does not work for directed graphs. In other words, describe a directed graph on n vertices and a starting point s such that the expected time to reach t is $\Omega(2^n)$ even though there is a directed path from s to t.

§9 Let G be an n vertex graph where all vertices have the same degree.

(a) We say that a distribution $\mathbf{p}$ over the vertices of G (where $\mathbf{p}_i$ denotes the probability that vertex i is picked by $\mathbf{p}$) is *stable* if when we choose a vertex i according to $\mathbf{p}$ and take a random step from i (i.e., move to a random neighbor j or i) then the resulting distribution is $\mathbf{p}$. Prove that the uniform distribution on G's vertices is stable.

(b) For $\mathbf{p}$ be a distribution over the vertices of G, let $\Delta(\mathbf{p}) = \max_i\{\mathbf{p}_i - 1/n\}$. For every k, denote by $\mathbf{p}^k$ the distribution obtained by choosing a vertex i at random from $\mathbf{p}$ and taking k random steps on G. Prove that if G is connected then there exists k such that $\Delta(\mathbf{p}^k) \leq (1 - n^{-10n})\Delta(\mathbf{p})$. Conclude that

   i. The uniform distribution is the only stable distribution for G.

   ii. For every vertices u, v of G, if we take a sufficiently long random walk starting from u, then with high probability the fraction of times we hit the vertex v is roughly $1/n$. That is, for every $\epsilon > 0$, there exists k such that the k-step random walk from u hits v between $(1-\epsilon)k/n$ and $(1+\epsilon)k/n$ times with probability at least $1 - \epsilon$.

(c) For a vertex u in G, denote by E_u the expected number of steps it takes for a random walk starting from u to reach back u. Show that $E_u \leq 10n^2$.

Hint: consider the infinite random walk starting from u. If $E_u > K$ then by standard tail bounds, u appears in less than a $2/K$ fraction of the places in this walk.

(d) For every two vertices u, v denote by $E_{u,v}$ the expected number of steps it takes for a random walk starting from u to reach v. Show that if u and v are connected by a path of length at most k then $E_{u,v} \leq 100kn^2$. Conclude that for every s and t that are connected in a graph G, the probability that an $1000n^3$ random walk from s does not hit t is at most $1/10$.

Hint: Start with the case $k = 1$ (i.e., u and v are connected by an edge), the case of $k > 1$ can be reduced to this using linearity of expectation. Note that the expectation of a random variable X over $\mathbb{N}$ is equal to $\sum_{m \in \mathbb{N}} \Pr[X \geq m]$ and so it suffices to show that the probability that an ℓn^2-step random walk from u does not hit v decays exponentially with ℓ.

(e) Let G be an n-vertex graph that is not necessarily regular (i.e., each vertex may have different degree). Let G' be the graph obtained by adding a sufficient number of parallel self-loops to each vertex to make G regular. Prove that if a k-step random walk in G' from a vertex s hits a vertex t with probability at least 0.9, then a $10n^2k$-step random walk from s will hit t with probability at least $1/2$.

The following exercises are based on Sections 7.A and 7.B.

§10 Let A be a symmetric stochastic matrix: $A = A^\dagger$ and every row and column of A has non-negative entries summing up to one. Prove that $\|A\| \leq 1$.

Hint: first show that $\|A\|$ is at most say n^2. Then, prove that for every $k \geq 1$, A^k is also stochastic and $\|A^{2k}\mathbf{v}\|_2 \geq \|A^k\mathbf{v}\|_2^2$ using the equality $\langle \mathbf{w}, B\mathbf{z} \rangle = \langle B^\dagger\mathbf{w}, \mathbf{z} \rangle$ and the inequality $\langle \mathbf{w}, \mathbf{z} \rangle \leq \|\mathbf{w}\|_2 \|\mathbf{z}\|_2$.

§11 Let A, B be two symmetric stochastic matrices. Prove that $\lambda(A + B) \leq \lambda(A) + \lambda(B)$.

§12 Let a n, d random graph be an n-vertex graph chosen as follows: choose d random permutations $\pi_1, ldots, \pi_d$ from $[n]$ to $[n]$. Let the the graph G contains an edge (u, v) for every pair u, v such that $v = \pi_i(u)$ for some $1 \leq i \leq d$. Prove that a random n, d graph is an $(n, 2d, \frac{2}{3}d)$ combinatorial expander with probability $1 - o(1)$ (i.e., tending to one with n).

Hint: for every set $S \subseteq n$ with $|S| \leq n/2$ and set $T \subseteq [n]$ with $|T| \geq (1 + \frac{2}{3}p)|S|$, try to bound the probability that $\pi_i(S) \subseteq T$ for every i.

The following two section assume some knowledge of elementary linear algebra (vector spaces and Hilbert spaces); see Appendix A for a quick review.

7.A   Random walks and eigenvalues

In this section we study random walks on (undirected regular) graphs, introducing several important notions such as the spectral gap of a graph's adjacency matrix. As a corollary we obtain the proof of correctness for the random-walk space-efficient algorithm for UPATH of Theorem 7.22. We will see that we can use elementary linear algebra to relate parameters of the graph's adjacency matrix to the behavior of the random walk on that graph.

REMARK 7.23
In this section, we restrict ourselves to *regular* graphs, in which every vertex have the same degree, although the definitions and results can be suitably generalized to general (non-regular) graphs.

7.A.1   Distributions as vectors and the parameter $\lambda(G)$.

Let G be a d-regular n-vertex graph. Let $\mathbf{p}$ be some probability distribution over the vertices of G. We can think of $\mathbf{p}$ as a (column) vector in $\mathbb{R}^n$ where $\mathbf{p}_i$ is the probability that vertex i is obtained by the distribution. Note that the L_1-norm of $\mathbf{p}$ (see Note 7.24), defined as $|\mathbf{p}|_1 = \sum_{i=1}^n |\mathbf{p}_i|$, is equal to 1. (In this case the absolute value is redundant since $\mathbf{p}_i$ is always between 0 and 1.)

Now let $\mathbf{q}$ represent the distribution of the following random variable: choose a vertex i in G according to $\mathbf{p}$, then take a random neighbor of i in G. We can compute $\mathbf{q}$ as a function of $\mathbf{p}$: the probability $\mathbf{q}_j$ that j is chosen is equal to the sum over all j's neighbors i of the probability $\mathbf{p}_i$ that i is chosen times $1/d$ (where $1/d$ is the probability that, conditioned on i being chosen, the walk moves to $\mathbf{q}$). Thus $\mathbf{q} = A\mathbf{p}$, where $A = A(G)$ which is the *normalized adjacency matrix* of G. That is, for every two vertices i, j, $A_{i,j}$ is equal to the number of edges between i and j divided by d. Note that A is a symmetric matrix,[3] where each entry is between 0 and 1, and the sum of entries in each row and column is exactly one (such a matrix is called a symmetric *stochastic* matrix).

Let $\{\mathbf{e}^i\}_{i=1}^n$ be the *standard basis* of $\mathbb{R}^n$ (i.e. $\mathbf{e}^i$ has 1 in the ith coordinate and zero everywhere else). Then, $A^T\mathbf{e}^s$ represents the distribution X_T of taking a T-step random walk from the vertex s. This already suggests that considering the adjacency matrix of a graph G could be very useful in analyzing random walks on G.

DEFINITION 7.25 (THE PARAMETER $\lambda(G)$.)
Denote by $\mathbf{1}$ the vector $(1/n, 1/n, \ldots, 1/n)$ corresponding to the uniform distribution. Denote by $\mathbf{1}^\perp$ the set of vectors perpendicular to $\mathbf{1}$ (i.e., $\mathbf{v} \in \mathbf{1}^\perp$ if $\langle \mathbf{v}, \mathbf{1} \rangle = (1/n) \sum_i \mathbf{v}_i = 0$).
The parameter $\lambda(A)$, denoted also as $\lambda(G)$, is the maximum value of $\|A\mathbf{v}\|_2$ over all vectors $\mathbf{v} \in \mathbf{1}^\perp$ with $\|\mathbf{v}\|_2 = 1$.

---

[3]A matrix A is *symmetric* if $A = A^\dagger$, where $A^\dagger$ denotes the *transpose* of A. That is, $(A^\dagger)_{i,j} = A_{j,i}$ for every i, j.

NOTE 7.24 (L_p NORMS)

A *norm* is a function mapping a vector $\mathbf{v}$ into a real number $\|\mathbf{v}\|$ satisfying
(1) $\|\mathbf{v}\| \geq 0$ with $\|\mathbf{v}\| = 0$ if and only $\mathbf{v}$ is the all zero vector, **(2)** $\|\alpha\mathbf{v}\| = |\alpha| \cdot \|\mathbf{v}\|$ for every $\alpha \in \mathbb{R}$, and **(3)** $\|\mathbf{v} + \mathbf{u}\| \leq \|\mathbf{v}\| + \|\mathbf{u}\|$ for every vector $\mathbf{u}$.
The third inequality implies that for every norm, if we define the *distance* between two vectors $\mathbf{u},\mathbf{v}$ as $\|\mathbf{u} - \mathbf{v}\|$ then this notion of distance satisfies the triangle inequality.

For every $\mathbf{v} \in \mathbb{R}^n$ and number $p \geq 1$, the L_p *norm* of $\mathbf{v}$, denoted $\|\mathbf{v}\|_p$, is equal to $(\sum_{i=1}^n |\mathbf{v}_i|^p)^{1/p}$. One particularly interesting case is $p = 2$, the so-called *Euclidean norm*, in which $\|\mathbf{v}\|_2 = \sqrt{\sum_{i=1}^n \mathbf{v}_i^2} = \sqrt{\langle \mathbf{v}, \mathbf{v}\rangle}$. Another interesting case is $p = 1$, where we use the single bar notation and denote $|\mathbf{v}|_1 = \sum_{i=1}^n |\mathbf{v}_i|$. Another case is $p = \infty$, where we denote $\|\mathbf{v}\|_\infty = \lim_{p\to\infty} \|\mathbf{v}\|_p = \max_{i\in[n]} |\mathbf{v}_i|$.

The *Hölder inequality* says that for every p, q with $\frac{1}{p} + \frac{1}{q} = 1$, $\|\mathbf{u}\|_p \|\mathbf{v}\|_q \geq \sum_{i=1}^n |\mathbf{u}_i\mathbf{v}_i|$. To prove it, note that by simple scaling, it suffices to consider norm one vectors, and so it enough to show that if $\|\mathbf{u}\|_p = \|\mathbf{v}\|_q = 1$ then $\sum_{i=1}^n |\mathbf{u}_i||\mathbf{v}_i| \leq 1$. But $\sum_{i=1}^n |\mathbf{u}_i||\mathbf{v}_i| = \sum_{i=1}^n |\mathbf{u}_i|^{p(1/p)}|\mathbf{v}_i|^{q(1/q)} \leq \sum_{i=1}^n \frac{1}{p}|\mathbf{u}_i|^p + \frac{1}{q}|\mathbf{v}_i|^q = \frac{1}{p} + \frac{1}{q} = 1$, where the last inequality uses the fact that for every $a, b > 0$ and $\alpha \in [0, 1]$, $a^\alpha b^{1-\alpha} \leq \alpha a + (1 - \alpha)b$. This fact is due to the log function being concave— having negative second derivative, implying that $\alpha \log a + (1 - \alpha) \log b \leq \log(\alpha a + (1 - \alpha)b)$.

Setting $p = 1$ and $q = \infty$, the Hölder inequality implies that

$$\|\mathbf{v}\|_2 \leq |\mathbf{v}|_1 \|\mathbf{v}\|_\infty$$

Setting $p = q = 2$, the Hölder inequality becomes the *Cauchy-Schwartz Inequality* stating that $\sum_{i=1}^n |\mathbf{u}_i\mathbf{v}_i| \leq \|\mathbf{u}\|_2 \|\mathbf{v}\|_2$. Setting $\mathbf{u} = (1/\sqrt{n}, 1/\sqrt{n}, \ldots, 1/\sqrt{n})$, we get that

$$|\mathbf{v}|_1/\sqrt{n} = \sum_{i=1}^n \tfrac{1}{\sqrt{n}}|\mathbf{v}_i| \leq \|\mathbf{v}\|_2$$

DRAFT

REMARK 7.26

The value $\lambda(G)$ is often called the *second largest eigenvalue* of G. The reason is that since A is a symmetric matrix, we can find an orthogonal basis of eigenvectors $\mathbf{v}^1, \ldots, \mathbf{v}^n$ with corresponding eigenvalues $\lambda_1, \ldots, \lambda_n$ which we can sort to ensure $|\lambda_1| \geq |\lambda_2| \ldots \geq |\lambda_n|$. Note that $A\mathbf{1} = \mathbf{1}$. Indeed, for every i, $(A\mathbf{1})_i$ is equal to the inner product of the ith row of A and the vector $\mathbf{1}$ which (since the sum of entries in the row is one) is equal to $1/n$. Thus, $\mathbf{1}$ is an *eigenvector* of A with the corresponding eigenvalue equal to 1. One can show that a symmetric stochastic matrix has all eigenvalues with absolute value at most 1 (see Exercise 10) and hence we can assume $\lambda_1 = 1$ and $\mathbf{v}^1 = \mathbf{1}$. Also, because $\mathbf{1}^\perp = \mathrm{Span}\{\mathbf{v}^2, \ldots, \mathbf{v}^n\}$, the value λ above will be maximized by (the normalized version of) $\mathbf{v}^2$, and hence $\lambda(G) = |\lambda_2|$. The quantity $1 - \lambda(G)$ is called the *spectral gap* of the graph. We note that some texts use *un-normalized* adjacency matrices, in which case $\lambda(G)$ is a number between 0 and d and the spectral gap is defined to be $d - \lambda(G)$.

One reason that $\lambda(G)$ is an important parameter is the following lemma:

LEMMA 7.27

For every regular n vertex graph $G = (V, E)$ let $\mathbf{p}$ be any probability distribution over V, then

$$\|A^T \mathbf{p} - \mathbf{1}\|_2 \leq \lambda^T$$

PROOF: By the definition of $\lambda(G)$, $\|A\mathbf{v}\|_2 \leq \lambda\|\mathbf{v}\|_2$ for every $\mathbf{v} \perp \mathbf{1}$. Note that if $\mathbf{v} \perp \mathbf{1}$ then $A\mathbf{v} \perp \mathbf{1}$ since $\langle \mathbf{1}, A\mathbf{v}\rangle = \langle A^\dagger \mathbf{1}, \mathbf{v}\rangle = \langle \mathbf{1}, \mathbf{v}\rangle = 0$ (as $A = A^\dagger$ and $A\mathbf{1} = \mathbf{1}$). Thus A maps the space $\mathbf{1}^\perp$ to itself and since it shrinks any member of this space by at least λ, $\lambda(A^T) \leq \lambda(A)^T$. (In fact, using the eigenvalue definition of λ, it can be shown that $\lambda(A^T) = \lambda(A)$.)

Let $\mathbf{p}$ be some vector. We can break $\mathbf{p}$ into its components in the spaces parallel and orthogonal to $\mathbf{1}$ and express it as $\mathbf{p} = \alpha\mathbf{1} + \mathbf{p}'$ where $\mathbf{p}' \perp \mathbf{1}$ and α is some number. If $\mathbf{p}$ is a probability distribution then $\alpha = 1$ since the sum of coordinates in $\mathbf{p}'$ is zero. Therefore,

$$A^T \mathbf{p} = A^T(\mathbf{1} + \mathbf{p}') = \mathbf{1} + A^T\mathbf{p}'$$

Since $\mathbf{1}$ and $\mathbf{p}'$ are orthogonal, $\|\mathbf{p}\|_2^2 = \|\mathbf{1}\|_2^2 + \|\mathbf{p}'\|_2^2$ and in particular $\|\mathbf{p}'\|_2 \leq \|\mathbf{p}\|_2$. Since $\mathbf{p}$ is a probability vector, $\|\mathbf{p}\|_2 \leq |\mathbf{p}|_1 \cdot 1 \leq 1$ (see Note 7.24). Hence $\|\mathbf{p}'\|_2 \leq 1$ and

$$\|A^T \mathbf{p} - \mathbf{1}\|_2 = \|A^T \mathbf{p}'\|_2 \leq \lambda^T$$

∎

It turns out that every connected graph has a noticeable spectral gap:

LEMMA 7.28

For every d-regular connected G with self-loops at each vertex, $\lambda(G) \leq 1 - \frac{1}{8dn^3}$.

PROOF: Let $\mathbf{u} \perp \mathbf{1}$ be a unit vector and let $\mathbf{v} = A\mathbf{u}$. We'll show that $1 - \|\mathbf{v}\|_2^2 \geq \frac{1}{d4n^3}$ which implies $\|\mathbf{v}\|_2^2 \leq 1 - \frac{1}{d4n^3}$ and hence $\|\mathbf{v}\|_2 \leq 1 - \frac{1}{d8n^3}$.

Since $\|\mathbf{u}\|_2 = 1$, $1 - \|\mathbf{v}\|_2^2 = \|\mathbf{u}\|_2^2 - \|\mathbf{v}\|_2^2$. We claim that this is equal to $\sum_{i,j} A_{i,j}(\mathbf{u}_i - \mathbf{v}_j)^2$ where i, j range from 1 to n. Indeed,

$$\sum_{i,j} A_{i,j}(\mathbf{u}_i - \mathbf{v}_j)^2 = \sum_{i,j} A_{i,j}\mathbf{u}_i^2 - 2\sum_{i,j} A_{i,j}\mathbf{u}_i\mathbf{v}_j + \sum_{i,j} A_{i,j}\mathbf{v}_j^2 =$$

$$\|\mathbf{u}\|_2^2 - 2\langle A\mathbf{u}, \mathbf{v}\rangle + \|\mathbf{v}\|_2^2 = \|\mathbf{u}\|_2^2 - 2\|\mathbf{v}\|_2^2 + \|\mathbf{v}\|_2^2,$$

where these equalities are due to the sum of each row and column in A equalling one, and because $\|\mathbf{v}\|_2^2 = \langle \mathbf{v}, \mathbf{v}\rangle = \langle A\mathbf{u}, \mathbf{v}\rangle = \sum_{i,j} A_{i,j}\mathbf{u}_i\mathbf{v}_j$.

Thus it suffices to show $\sum_{i,j} A_{i,j}(\mathbf{u}_i - \mathbf{v}_j)^2 \geq \frac{1}{d4n^3}$. This is a sum of non-negative terms so it suffices to show that for *some* i, j, $A_{i,j}(\mathbf{u}_i - \mathbf{v}_j)^2 \geq \frac{1}{d4n^3}$. First, because we have all the self-loops, $A_{i,i} \geq 1/d$ for all i, and so we can assume $|\mathbf{u}_i - \mathbf{v}_i| < \frac{1}{2n^{1.5}}$ for every $i \in [n]$, as otherwise we'd be done.

Now sort the coordinates of $\mathbf{u}$ from the largest to the smallest, ensuring that $\mathbf{u}_1 \geq \mathbf{u}_2 \geq \cdots \mathbf{u}_n$. Since $\sum_i \mathbf{u}_i = 0$ it must hold that $\mathbf{u}_1 \geq 0 \geq \mathbf{u}_n$. In fact, since $\mathbf{u}$ is a unit vector, either $\mathbf{u}_1 \geq 1/\sqrt{n}$ or $\mathbf{u}_n \leq 1/\sqrt{n}$ and so $\mathbf{u}_1 - \mathbf{u}_n \geq 1/\sqrt{n}$. One of the $n-1$ differences between consecutive coordinates $\mathbf{u}_i - \mathbf{u}_{i+1}$ must be at least $1/n^{1.5}$ and so there must be an i_0 such that if we let $S = \{1, \ldots, i_0\}$ and $\overline{S} = [n] \setminus S_i$, then for every $i \in S$ and $j \in \overline{S}$, $\mathbf{u}_i - \mathbf{u}_j \geq 1/n^{1.5}$. Since G is connected there exists an edge (i, j) between S and $\overline{S}$. Since $|\mathbf{v}_j - \mathbf{u}_j| \leq \frac{1}{2n^{1.5}}$, for this choice of i, j, $|\mathbf{u}_i - \mathbf{v}_j| \geq |\mathbf{u}_i - \mathbf{u}_j| - \frac{1}{2n^{1.5}} \geq \frac{1}{2n^{1.5}}$. Thus $A_{i,j}(\mathbf{u}_i - \mathbf{v}_j)^2 \geq \frac{1}{d}\frac{1}{4n^3}$. ∎

REMARK 7.29
The proof can be strengthened to show a similar result for every connected non-bipartite graph (not just those with self-loops at every vertex). Note that this condition is essential: if A is the adjacency matrix of a bipartite graph then one can find a vector $\mathbf{v}$ such that $A\mathbf{v} = -\mathbf{v}$.

7.A.2   Analysis of the randomized algorithm for undirected connectivity.

Together, Lemmas 7.27 and 7.28 imply that, at least for regular graphs, if s is connected to t then a sufficiently long random walk from s will hit t in polynomial time with high probability.

COROLLARY 7.30
Let G be a d-regular n-vertex graph with all vertices having a self-loop. Let s be a vertex in G. Let $T > 10dn^3 \log n$ and let X_T denote the distribution of the vertex of the Tth step in a random walk from s. Then, for every j connected to s, $\Pr[X_T = j] > \frac{1}{2n}$.

PROOF: By these Lemmas, if we consider the restriction of an n-vertex graph G to the connected component of s, then for every probability vector $\mathbf{p}$ over this component and $T \geq 10dn^3 \log n$, $\|A^T\mathbf{p} - \mathbf{1}\|_2 < \frac{1}{2n^{1.5}}$ (where $\mathbf{1}$ here is the uniform distribution over this component). Using the relations between the L_1 and L_2 norms (see Note 7.24), $|A^T\mathbf{p} - \mathbf{1}|_1 < \frac{1}{2n}$ and hence every element in the connected component appears in $A^T\mathbf{p}$ with at least $1/n - 1/(2n) \geq 1/(2n)$ probability. ∎

Note that Corollary 7.30 implies that if we repeat the $10dn^3 \log n$ walk for $10n$ times (or equivalently, if we take a walk of length $100dn^4 \log n$) then we will hit t with probability at least 3/4.

7.B Expander graphs.

Expander graphs are extremely useful combinatorial objects, which we will encounter several times in the book. They can be defined in two equivalent ways. At a high level, these two equivalent definitions can be described as follows:

- *Combinatorial definition:* A constant-degree regular graph G is an *expander* if for every subset S of less than half of G's vertices, a constant fraction of the edges touching S are from S to its complement in G. This is the definition alluded to in Section 7.5 (see Figure 7.1).[4]

- *Algebraic expansion:* A constant-degree regular graph G is an *expander* if its parameter $\lambda(G)$ bounded away from 1 by some constant. That is, $\lambda(G) \leq 1 - \epsilon$ for some constant $\epsilon > 0$¿

What do we mean by a constant? By *constant* we refer to a number that is independent of the size of the graph. We will typically talk about graphs that are part of an infinite *family* of graphs, and so by constant we mean a value that is the same for all graphs in the family, regardless of their size.

Below we make the definitions more precise, and show their equivalence. We will then complete the analysis of the randomness efficient error reduction procedure described in Section 7.5.

7.B.1 The Algebraic Definition

.

The Algebraic definition of expanders is as follows:

DEFINITION 7.31 ((n, d, λ)-GRAPHS.)
If G is an n-vertex d-regular G with $\lambda(G) \leq \lambda$ for some number $\lambda < 1$ then we say that G is an (n, d, λ)-graph.
A family of graphs $\{G_n\}_{n \in \mathbb{N}}$ is an *expander graph family* if there are some constants $d \in \mathbb{N}$ and $\lambda < 1$ such that for every n, G_n is an (n, d, λ)-graph.

Explicit constructions. We say that an expander family $\{G_n\}_{n \in \mathbb{N}}$ is *explicit* if there is a polynomial-time algorithm that on input 1^n with $n \in I$ outputs the adjacency matrix of G_n. We say that the family is *strongly explicit* if there is a polynomial-time algorithm that for every $n \in I$ on inputs $\langle n, v, i \rangle$ where $1 \leq v \leq n'$ and $1 \leq i \leq d$ outputs the ith neighbor of v. (Note that the algorithm runs in time polynomial in the its input length which is polylogarithmic in n.)

As we will see below it is not hard to show that expander families exist using the probabilistic method. But this does not yield *explicit* (or very explicit) constructions of such graphs (which, as we saw in Section 7.4.1 are often needed for applications). In fact, there are also several explicit and

---

[4]The careful reader might note that there we said that a graph is an expander if a constant fraction of S's neighboring *vertices* are outside S. However, for constant-degree graphs these two notions are equivalent.

NOTE 7.33 (EXPLICIT CONSTRUCTION OF PSEUDORANDOM OBJECTS)
Expanders are one instance of a recurring theme in complexity theory (and
other areas of math and computer science): it is often the case that a ran-
dom object can be easily proven to satisfy some nice property, but the ap-
plications require an *explicit* object satisfying this property. In our case,
a random d-regular graph is an expander, but to use it for, say, reducing
the error of probabilistic algorithms, we need an *explicit* construction of an
expander family, with an efficient deterministic algorithm to compute the
neighborhood relations. Such explicit constructions can be sometimes hard
to come by, but are often surprisingly useful. For example, in our case the
explicit construction of expander graphs turns out to yield a deterministic
logspace algorithm for undirected connectivity.
We will see another instance of this theme in Chapter 17, which discusses
error correcting codes.

---

strongly explicit constructions of expander graphs known. The smallest λ can be for a d-regular
n-vertex graph is $\Omega(\frac{1}{\sqrt{d}})$ and there are constructions meeting this bound (specifically the bound
is $(1 - o(1))\frac{2\sqrt{d-1}}{d}$ where by $o(1)$ we mean a function that tends to 0 as the number of vertices
grows; graphs meeting this bound are called *Ramanujan graphs*). However, for most applications in
Computer Science, any family with constant d and $\lambda < 1$ will suffice (see also Remark 7.32 below).
Some of these constructions are very simple and efficient, but their analysis is highly non-trivial
and uses relatively deep mathematics.[5] In Chapter 16 we will see a strongly explicit construction
of expanders with elementary analysis. This construction also introduces a tool that is useful to
derandomize the random-walk algorithm for UPATH.

REMARK 7.32
One reason that the particular constants of an expander family are not extremely crucial is that we
can improve the constant λ (make it arbitrarily smaller) at the expense of increasing the degree:
this follows from the fact, observed above in the proof of Lemma 7.27, that $\lambda(G^T) = \lambda(G)^T$, where
G^T denotes the graph obtained by taking the adjacency matrix to the Tth power, or equivalently,
having an edge for every length-T path in G. Thus, we can transform an (n, d, λ) graph into an
(n, d^T, λ^T)-graph for every $T \geq 1$. In Chapter 16 we will see a different transformation called
the *replacement product* to decrease the degree at the expense of increasing λ somewhat (and also
increasing the number of vertices).

---

[5]An example for such an expander is the following 3-regular graph: the vertices are the numbers 1 to $p - 1$ for
some prime p, and each number x is connected to $x + 1, x - 1$ and $x^{-1} \pmod{p}$.

7.B.2 Combinatorial expansion and existence of expanders.

We describe now a combinatorial criteria that is roughly equivalent to Definition 7.31. One advantage of this criteria is that it makes it easy to prove that a non-explicit expander family exists using the probabilistic method. It is also quite useful in several applications.

DEFINITION 7.34 ([)
Combinatorial (edge) expansion] An n-vertex d-regular graph $G = (V, E)$ is called an (n, d, ρ)-*combinatorial expander* if for every subset $S \subseteq V$ with $|S| \leq n/2$, $|E(S, \overline{S})| \geq \rho d |S|$, where for subsets S, T of V, $E(S, T)$ denotes the set of edges (s, t) with $s \in S$ and $t \in T$.

Note that in this case the bigger ρ is the better the expander. We'll loosely use the term expander for any (n, d, ρ)-combinatorial expander with c a positive constant. Using the probabilistic method, one can prove the following theorem: (Exercise 12 asks you to prove a slightly weaker version)

THEOREM 7.35 (EXISTENCE OF EXPANDERS)
Let $\epsilon > 0$ be some constant. Then there exists $d = d(\epsilon)$ and $N \in \mathbb{N}$ such that for every $n > N$ there exists an $(n, d, 1 - \epsilon)$-combinatorial expander.

The following theorem related combinatorial expansion with our previous Definition 7.31

THEOREM 7.36 (COMBINATORIAL AND ALGEBRAIC EXPANSION)
  1. *If G is an (n, d, λ)-graph then it is an $(n, d, (1 - \lambda)/2)$-combinatorial expander.*

  2. *If G is an (n, d, ρ)-combinatorial expander then it is an $(n, d, 1 - \frac{\rho^2}{2})$-graph.*

The first part of Theorem 7.36 follows by plugging $T = \overline{S}$ into the following lemma:

LEMMA 7.37 (EXPANDER MIXING LEMMA)
Let $G = (V, E)$ be an (n, d, λ)-graph. Let $S, T \subseteq V$, then

$$\left| |E(S, T)| - \frac{d}{n}|S||T| \right| \leq \lambda d \sqrt{|S||T|}$$

PROOF: Let $\mathbf{s}$ denote the vector such that $\mathbf{s}_i$ is equal to 1 if $i \in S$ and equal to 0 otherwise, and let $\mathbf{t}$ denote the corresponding vector for the set S. Thinking of $\mathbf{s}$ as a row vector and of $\mathbf{t}$ as a column vector, the Lemma's statement is equivalent to

$$\left| \mathbf{s}A\mathbf{t} - \frac{|S||T|}{n} \right| \leq \lambda \sqrt{|S||T|}, \tag{2}$$

where A is G's normalized adjacency matrix. Yet by Lemma 7.40, we can write A as $(1 - \lambda)J + \lambda C$, where J is the matrix with all entries equal to $1/n$ and C has norm at most one. Hence,

$$\mathbf{s}A\mathbf{t} = (1 - \lambda)\mathbf{s}J\mathbf{t} + \lambda \mathbf{s}C\mathbf{t} \leq \frac{|S||T|}{n} + \lambda \sqrt{|S||T|},$$

where the last inequality follows from $\mathbf{s}J\mathbf{t} = |S||T|/n$ and $\mathbf{s}C\mathbf{t} = \langle \mathbf{s}, C\mathbf{t} \rangle \le \|\mathbf{s}\|_2 \|\mathbf{t}\|_2 = \sqrt{|S||T|}$. ∎

PROOF OF SECOND PART OF THEOREM 7.36.: We prove a slightly relaxed version, replacing the constant 2 with 8. Let $G = (V, E)$ be an n-vertex d-regular graph such that for every subset $S \subseteq V$ with $|S| \le n/2$, there are $\rho|S|$ edges between S and $\overline{S} = V \setminus S$, and let A be G's normalized adjacency matrix.

Let $\lambda = \lambda(G)$. We need to prove that $\lambda \le 1 - \rho^2/8$. Using the fact that λ is the second eigenvalue of A, there exists a vector $\mathbf{u} \perp \mathbf{1}$ such that $A\mathbf{u} = \lambda \mathbf{u}$. Write $\mathbf{u} = \mathbf{v} + \mathbf{w}$ where $\mathbf{v}$ is equal to $\mathbf{u}$ on the coordinates on which $\mathbf{u}$ is positive and equal to 0 otherwise, and $\mathbf{w}$ is equal to $\mathbf{u}$ on the coordinates on which $\mathbf{u}$ is negative, and equal to 0 otherwise. Note that, since $\mathbf{u} \perp \mathbf{1}$, both $\mathbf{v}$ and $\mathbf{w}$ are nonzero. We can assume that $\mathbf{u}$ is nonzero on at most $n/2$ of its coordinates (as otherwise we can take $-\mathbf{u}$ instead of $\mathbf{u}$).

Since $A\mathbf{u} = \lambda \mathbf{u}$ and $\langle \mathbf{v}, \mathbf{w} \rangle = 0$,

$$\langle A\mathbf{v}, \mathbf{v} \rangle + \langle A\mathbf{w}, \mathbf{v} \rangle = \langle A(\mathbf{v} + \mathbf{w}), \mathbf{v} \rangle = \langle A\mathbf{u}, \mathbf{v} \rangle = \langle \lambda(\mathbf{v} + \mathbf{w}), \mathbf{v} \rangle = \lambda \|\mathbf{v}\|_2^2 \,.$$

Since $\langle A\mathbf{w}, \mathbf{v} \rangle$ is negative, we get that $\langle A\mathbf{v}, \mathbf{v} \rangle / \|\mathbf{v}\|_2^2 \ge \lambda$ or

$$1 - \lambda \ge 1 - \frac{\langle A\mathbf{v}, \mathbf{v} \rangle}{\|\mathbf{v}\|_2^2} = \frac{\|\mathbf{v}\|_2^2 - \langle A\mathbf{v}, \mathbf{v} \rangle}{\|\mathbf{v}\|_2^2} = \frac{\sum_{i,j} A_{i,j}(\mathbf{v}_i - \mathbf{v}_j)^2}{2\|\mathbf{v}\|_2^2} \,,$$

where the last equality is due to $\sum_{i,j} A_{i,j}(\mathbf{v}_i - \mathbf{v}_j)^2 = \sum_{i,j} A_{i,j}\mathbf{v}_i^2 - 2\sum_{i,j} A_{i,j}\mathbf{v}_i\mathbf{v}_j + \sum_{i,j} A_{i,j}\mathbf{v}_j^2 = 2\|\mathbf{v}\|_2^2 - 2\langle A\mathbf{v}, \mathbf{v} \rangle$. (We use here the fact that each row and column of A sums to one.) Multiply both numerator and denominator by $\sum_{i,j} A_{i,j}(\mathbf{v}_i^2 + \mathbf{v}_j^2)$. By the Cauchy-Schwartz inequality,[6] we can bound the new numerator as follows:

$$\left(\sum_{i,j} A_{i,j}(\mathbf{v}_i - \mathbf{v}_j)^2 \right) \left(\sum_{i,j} A_{i,j}(\mathbf{v}_i + \mathbf{v}_j)^2 \right) \le \left(\sum_{i,j} A_{i,j}(\mathbf{v}_i - \mathbf{v}_j)(\mathbf{v}_i + \mathbf{v}_j) \right)^2 \,.$$

Hence, using $(a - b)(a + b) = a^2 - b^2$,

$$1 - \lambda \ge \frac{\left(\sum_{i,j} A_{i,j}(\mathbf{v}_i^2 - \mathbf{v}_j^2) \right)^2}{2\|\mathbf{v}\|_2^2 \sum_{i,j} A_{i,j}(\mathbf{v}_i + \mathbf{v}_j)^2} = \frac{\left(\sum_{i,j} A_{i,j}(\mathbf{v}_i^2 - \mathbf{v}_j^2) \right)^2}{2\|\mathbf{v}\|_2^2 \left(\sum_{i,j} A_{i,j}\mathbf{v}_i^2 + 2\sum_{i,j} A_{i,j}\mathbf{v}_i\mathbf{v}_j + \sum_{i,j} A_{i,j}\mathbf{v}_j^2 \right)} =$$

$$\frac{\left(\sum_{i,j} A_{i,j}(\mathbf{v}_i^2 - \mathbf{v}_j^2) \right)^2}{2\|\mathbf{v}\|_2^2 \left(2\|\mathbf{v}\|_2^2 + 2\langle A\mathbf{v}, \mathbf{v} \rangle \right)} \ge \frac{\left(\sum_{i,j} A_{i,j}(\mathbf{v}_i^2 - \mathbf{v}_j^2) \right)^2}{8\|\mathbf{v}\|_2^4} \,,$$

where the last inequality is due to A having matrix norm at most 1, implying $\langle A\mathbf{v}, \mathbf{v} \rangle \le \|\mathbf{v}\|_2^2$. We conclude the proof by showing that

$$\sum_{i,j} A_{i,j}(\mathbf{v}_i^2 - \mathbf{v}_j^2) \ge \rho \|\mathbf{v}\|_2^2 \,, \tag{3}$$

---

[6] The Cauchy-Schwartz inequality is typically stated as saying that for $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, $\sum_i \mathbf{x}_i \mathbf{y}_i \le \sqrt{(\sum_i \mathbf{x}_i^2)(\sum_i \mathbf{y}_i^2)}$. However, it is easily generalized to show that for every non-negative $\mu_1, \ldots, \mu_n$, $\sum_i \mu_i \mathbf{x}_i \mathbf{y}_i \le \sqrt{(\sum_i \mu_i \mathbf{x}_i^2)(\sum_i \mu_i \mathbf{y}_i^2)}$ (this can be proven from the standard Cauchy-Schwartz by multiplying each coordinate of $\mathbf{x}$ and $\mathbf{y}$ by $\sqrt{\mu_i}$. It is this variant that we use here with the $A_{i,j}$'s playing the role of $\mu_1, \ldots, \mu_n$.

which indeed implies that $1 - \lambda \geq \frac{\rho^2 \|\mathbf{v}\|_2^4}{8\|\mathbf{v}\|_2^4} = \frac{\rho^2}{8}$.

To prove (3) sort the coordinates of $\mathbf{v}$ so that $\mathbf{v}_1 \geq \mathbf{v}_2 \geq \cdots \geq \mathbf{v}_n$ (with $\mathbf{v}_i = 0$ for $i > n/2$). Then

$$\sum_{i,j} A_{i,j}(\mathbf{v}_i^2 - \mathbf{v}_j^2) \geq \sum_{i=1}^{n/2} \sum_{j=i+1}^{n} A_{i,j}(\mathbf{v}_i^2 - \mathbf{v}_{i+1}^2) = \sum_{i=1}^{n/2} c_i(\mathbf{v}_i^2 - \mathbf{v}_{i+1}^2),$$

where c_i denotes $\sum_{j>i} A_{i,j}$. But c_i is equal to the number of edges in G from the set $\{k : k \leq i\}$ to its complement, divided by d. Hence, by the expansion of G, $c_i \geq \rho i$, implying (using the fact that $\mathbf{v}_i = 0$ for $i \geq n/2$):

$$\sum_{i,j} A_{i,j}(\mathbf{v}_i^2 - \mathbf{v}_j^2) \geq \sum_{i=1}^{n/2} \rho i(\mathbf{v}_i^2 - \mathbf{v}_{i+1}^2) = \sum_{i=1}^{n/2}(\rho i \mathbf{v}_i^2 - \rho \cdot (i-1)\mathbf{v}_i^2) = \rho\|\mathbf{v}\|_2^2,$$

establishing (3). ∎

7.B.3 Error reduction using expanders.

We now complete the analysis of the randomness efficient error reduction procedure described in Section 7.5. Recall, that this procedure was the following: let $N = 2^m$ where m is the number of coins the randomized algorithm uses. We use $m + O(k)$ random coins to select a k-vertex random walk in an expander graph G_N, and then output 1 if and only if the algorithm outputs 1 when given one of the vertices in the walk as random coins. To show this procedure works we need to show that if the probabilistic algorithm outputs 1 for at least half of the coins, then the probability that all the vertices of the walk correspond to coins on which the algorithm outputs 0 is exponentially small in k. This will be a direct consequence of the following theorem: (think of the set B below as the set of vertices corresponding to coins on which the algorithm outputs 0)

THEOREM 7.38 (EXPANDER WALKS)
Let G be an (N, d, λ) graph, and let $B \subseteq [N]$ be a set with $|B| \leq \beta N$. Let $X_1, \ldots, X_k$ be random variables denoting a $k-1$-step random walk from X_1, where X_1 is chosen uniformly in $[N]$. Then,

$$\underbrace{\Pr[\forall_{1 \leq i \leq k} X_i \in B]}_{(*)} \leq ((1-\lambda)\sqrt{\beta} + \lambda)^{k-1}$$

Note that if λ and β are both constants smaller than 1 then so is the expression $(1-\lambda)\sqrt{\beta} + \lambda$.
PROOF: For $1 \leq i \leq k$, let B_i be the event that $X_i \in B$. Note that the probability $(*)$ we're trying to bound is $\Pr[B_1]\Pr[B_2|B_1]\cdots\Pr[B_k|B_1,\ldots,B_{k-1}]$. Let $\mathbf{p}^i \in \mathbb{R}^N$ be the vector representing the distribution of X_i, conditioned on the events $B_1,\ldots,B_i$. Denote by $\hat{B}$ the following linear transformation from $\mathbb{R}^n$ to $\mathbb{R}^n$: for every $\mathbf{u} \in \mathbb{R}^N$, and $j \in [N]$, $(\hat{B}\mathbf{u})_j = \mathbf{u}_j$ if $j \in B$ and $(\hat{B}\mathbf{u})_j = 0$ otherwise. It's not hard to verify that $\mathbf{p}^1 = \frac{1}{\Pr[B_1]}\hat{B}\mathbf{1}$ (recall that $\mathbf{1} = (1/N,\ldots,1/N)$ is the

vector representing the uniform distribution over $[N]$). Similarly, $\mathbf{p}^2 = \frac{1}{\Pr[B_2|B_1]}\frac{1}{\Pr[B_1]}\hat{B}A\hat{B}\mathbf{1}$ where $A = A(G)$ is the adjacency matrix of G. Since every probability vector $\mathbf{p}$ satisfies $|\mathbf{p}|_1 = 1$,

$$(*) = |(\hat{B}A)^{k-1}\hat{B}\mathbf{1}|_1$$

We bound this norm by showing that

$$\|(\hat{B}A)^{k-1}\hat{B}\mathbf{1}\|_2 \le \frac{((1-\lambda)\sqrt{\beta}+\lambda)^{k-1}}{\sqrt{N}} \tag{4}$$

which suffices since for every $\mathbf{v} \in \mathbb{R}^N$, $|\mathbf{v}|_1 \le \sqrt{N}\|\mathbf{v}\|_2$ (see Note 7.24).

To prove (4), we use the following definition and lemma:

DEFINITION 7.39 (MATRIX NORM)
If A is an m by n matrix, then $\|A\|$ is the maximum number α such that $\|A\mathbf{v}\|_2 \le \alpha\|\mathbf{v}\|_2$ for every $\mathbf{v} \in \mathbb{R}^n$.

Note that if A is a normalized adjacency matrix then $\|A\| = 1$ (as $A\mathbf{1} = \mathbf{1}$ and $\|A\mathbf{v}\|_2 \le \|\mathbf{v}\|_2$ for every $\mathbf{v}$). Also note that the matrix norm satisfies that for every two n by n matrices A, B, $\|A + B\| \le \|A\| + \|B\|$ and $\|AB\| \le \|A\|\|B\|$.

LEMMA 7.40
Let A be a normalized adjacency matrix of an (n, d, λ)-graph G. Let J be the adjacency matrix of the n-clique with self loops (i.e., $J_{i,j} = 1/n$ for every i, j). Then

$$A = (1 - \lambda)J + \lambda C \tag{5}$$

where $\|C\| \le 1$.

Note that for every probability vector $\mathbf{p}$, $J\mathbf{p}$ is the uniform distribution, and so this lemma tells us that in some sense, we can think of a step on a (n, d, λ)-graph as going to the uniform distribution with probability $1 - \lambda$, and to a different distribution with probability λ. This is of course not completely accurate, as a step on a d-regular graph will only go the one of the d neighbors of the current vertex, but we'll see that for the purposes of our analysis, the condition (5) will be just as good.[7]

PROOF OF LEMMA 7.40: Indeed, simply define $C = \frac{1}{\lambda}(A - (1 - \lambda)J)$. We need to prove $\|C\mathbf{v}\|_2 \le \|\mathbf{v}\|_2$ for very $\mathbf{v}$. Decompose $\mathbf{v}$ as $\mathbf{v} = \mathbf{u} + \mathbf{w}$ where u is $\alpha\mathbf{1}$ for some α and $\mathbf{w} \perp \mathbf{1}$, and $\|\mathbf{v}\|_2^2 = \|\mathbf{u}\|_2^2 + \|\mathbf{w}\|_2^2$. Since $A\mathbf{1} = \mathbf{1}$ and $J\mathbf{1} = \mathbf{1}$ we get that $C\mathbf{u} = \frac{1}{\lambda}(\mathbf{u} - (1 - \lambda)\mathbf{u}) = \mathbf{u}$. Now, let $\mathbf{w}' = A\mathbf{w}$. Then $\|\mathbf{w}'\|_2 \le \lambda\|\mathbf{w}\|_2$ and, as we saw in the proof of Lemma 7.27, $\mathbf{w}' \perp \mathbf{1}$. Furthermore, since the sum of the coordinates of $\mathbf{w}$ is zero, $J\mathbf{w} = \mathbf{0}$. We get that $C\mathbf{w} = \frac{1}{\lambda}\mathbf{w}'$. Since $\mathbf{w}' \perp \mathbf{u}$, $\|C\mathbf{w}\|_2^2 = \|\mathbf{u} + \frac{1}{\lambda}\mathbf{w}'\|_2^2 = \|\mathbf{u}\|_2^2 + \|\frac{1}{\lambda}\mathbf{w}'\|_2^2 \le \|\mathbf{u}\|_2^2 + \|\mathbf{w}\|_2^2 = \|\mathbf{w}\|_2^2$. $\blacksquare$

Returning to the proof of Theorem 7.38, we can write $\hat{B}A = \hat{B}\big((1 - \lambda)J + \lambda C\big)$, and hence $\|\hat{B}A\| \le (1 - \lambda)\|\hat{B}J\| + \lambda\|\hat{B}C\|$. Since J's output is always a vector of the form $\alpha\mathbf{1}$, $\|\hat{B}J\| \le \sqrt{\beta}$. Also, because $\hat{B}$ is an operation that merely zeros out some parts of its input, $\|\hat{B}\| \le 1$ implying

---

[7]Algebraically, the reason (5) is not equivalent to going to the uniform distribution in each step with probability $1 - \lambda$ is that C is not necessarily a stochastic matrix, and may have negative entries.

$\|\hat{B}C\| \le 1$. Thus, $\|\hat{B}A\| \le (1-\lambda)\sqrt{\beta}+\lambda$. Since $B\mathbf{1}$ has the value $1/N$ in $|B|$ places, $\|B\mathbf{1}\|_2 = \frac{\sqrt{\beta}}{\sqrt{N}}$, and hence $\|(\hat{B}A)^{k-1}\hat{B}\mathbf{1}\|_2 \le ((1-\lambda)\sqrt{\beta}+\lambda)^{k-1}\frac{\sqrt{\beta}}{\sqrt{N}}$, establishing (4). ∎

One can obtain a similar error reduction procedure for *two-sided error* algorithms by running the algorithm using the k sets of coins obtained from a $k-1$ step random walk and deciding on the output according to the *majority* of the values obtained. The analysis of this procedure is based on the following theorem, whose proof we omit:

THEOREM 7.41 (EXPANDER CHERNOFF BOUND [?])
Let G be an (N, d, λ)-graph and $B \subseteq [N]$ with $|B| = \beta N$. Let $X_1, \ldots, X_k$ be random variables denoting a $k-1$-step random walk in G (where X_1 is chosen uniformly). For every $i \in [k]$, define B_i to be 1 if $X_i \in B$ and 0 otherwise. Then, for every $\delta > 0$,

$$\Pr\left[|\tfrac{\sum_{i=1}^{k} B_i}{k} - \beta| > \delta\right] < 2e^{(1-\lambda)\delta^2 k/60}$$

Chapter 8

Interactive proofs

> "What is intuitively required from a theorem-proving procedure? First, that it is possible to "prove" a true theorem. Second, that it is impossible to "prove" a false theorem. Third, that communicating the proof should be efficient, in the following sense. It does not matter how long must the prover compute during the proving process, but it is essential that the computation required from the verifier is easy."
> Goldwasser, Micali, Rackoff 1985

The standard notion of a mathematical proof follows the certificate definition of **NP**. That is, to prove that a statement is true one provides a sequence of symbols that can be written down in a book or on paper, and a valid sequence exists only for true statements. However, people often use more general ways to convince one another of the validity of statements: they *interact* with one another, with the person verifying the proof (henceforth the *verifier*) asking the person providing it (henceforth the *prover*) for a series of explanations before he is convinced.

It seems natural to try to understand the power of such interactive proofs from the complexity-theoretic perspective. For example, can one prove that a given formula is *not* satisfiable? (recall that is this problem is **coNP**-complete, it's not believed to have a polynomial-sized certificate). The surprising answer is *yes*. Indeed, interactive proofs turned out to have unexpected powers and applications. Beyond their philosophical appeal, interactive proofs led to fundamental insights in cryptographic protocols, the power of approximation algorithms, program checking, and the hardness of famous "elusive" problems (i.e., **NP**-problems not known to be in **P** nor to be **NP**-complete) such as *graph isomorphism* and *approximate shortest lattice vector*.

8.1 Warmup: Interactive proofs with a deterministic verifier

Let us consider what happens when we introduce *interaction* into the **NP** scenario. That is, we'd have an interrogation-style proof system where rather than the prover send a written proof to the verifier, the prover and verifier interact with the verifier asking questions and the prover responding, where at the end the verifier decides whether or not to accept the input. Of course, both verifier and prover can keep state during the interaction, or equivalently, the message a party sends at any

point in the interaction can be a function of all messages sent and received so far. Formally, we make the following definition:

DEFINITION 8.1 (INTERACTION OF DETERMINISTIC FUNCTIONS)
Let $f, g : \{0,1\}^* \to \{0,1\}^*$ be functions. A *k-round interaction* of f and g on input $x \in \{0,1\}^*$, denoted by $\langle f, g \rangle(x)$ is the sequence of the following strings $a_1, \ldots, a_k \in \{0,1\}^*$ defined as follows:

$$
\begin{aligned}
a_1 &= f(x) \\
a_2 &= g(x, a_1) \\
&\ldots \\
a_{2i+1} &= f(x, a_1, \ldots, a_{2i}) \\
a_{2i+2} &= g(x, a_1, \ldots, a_{2i+1})
\end{aligned}
\tag{1}
$$

(Where we consider a suitable encoding of i-tuples of strings to strings.)

The *output* of f (resp. g) at the end of the interaction denoted $\mathsf{out}_f\langle f, g \rangle(x)$ (resp. $\mathsf{out}_g\langle f, g \rangle(x)$) is defined to be $f(x, a_1, \ldots, a_k)$ (resp. $g(x, a_1, \ldots, a_k)$).

DEFINITION 8.2 (DETERMINISTIC PROOF SYSTEMS)
We say that a language L has a *k-round deterministic interactive proof system* if there's a deterministic TM V that on input $x, a_1, \ldots, a_i$ runs in time polynomial in $|x|$, satisfying:

$$
\begin{aligned}
\text{(Completeness)} x \in L &\Rightarrow \exists P : \{0,1\}^* \to \{0,1\}^* \, \mathsf{out}_V\langle V, P \rangle(x) = 1 \\
\text{(Soundness)} x \notin L &\Rightarrow \forall P : \{0,1\}^* \to \{0,1\}^* \, \mathsf{out}_V\langle V, P \rangle(x) = 1
\end{aligned}
$$

The class **dIP** contains all languages with a $k(n)$-round deterministic interactive proof systems with $k(n)$ polynomial in n.

It turns out this actually does not change the class of languages we can prove:

THEOREM 8.3
dIP = NP.

PROOF: Clearly, every **NP** language has a 1-round proof system. Now we prove that if a L has an interactive proof system of this type then $L \in \mathbf{NP}$. The certificate for membership is just the transcript $(a_1, a_2, \ldots, a_k)$ causing the verifier to accept. To verify this transcript, check that indeed $V(x) = a_1$, $V(x, a_1, a_2) = a_3$, ..., and $V(x, a_1, \ldots, a_k) = 1$. If $x \in L$ then there indeed exists such a transcript. If there exists such a transcript $(a_1, \ldots, a_k)$ then we can define a prover function P to satisfy $P(x, a_1) = a_2$, $P(x, a_1, a_2, a_3) = a_4$, etc. We see that $\mathsf{out}_V\langle V, P \rangle(x) = 1$ and hence $x \in L$. ∎

8.2 The class IP

In order to realize the full potential of interaction, we need to let the verifier be *probabilistic*. The idea is that, similar to probabilistic algorithms, the verifier will be allowed to come to a wrong conclusion (e.g., accept a proof for a wrong statement) with some small probability. However, as in the case of probabilistic algorithms, this probability is over the verifier's coins and the verifier will reject proofs for a wrong statement with good probability *regardless* of the strategy the prover uses. It turns out that the combination of interaction and randomization has a huge effect: as we will see, the set of languages which have interactive proof systems now jumps from **NP** to **PSPACE**.

EXAMPLE 8.4
As an example for a probabilistic interactive proof system, consider the following scenario: Marla claims to Arthur that she can distinguish between the taste of Coke (Coca-Cola) and Pepsi. To verify this statement, Marla and Arthur repeat the following experiment 50 times: Marla turns her back to Arthur, as he places Coke in one unmarked cup and Pepsi in another, choosing randomly whether Coke will be in the cup on the left or on the right. Then Marla tastes both cups and states which one contained which drinks. While, regardless of her tasting abilities, Marla can answer correctly with probability $\frac{1}{2}$ by a random guess, if she manages to answer correctly for *all* the 50 repetitions, Arthur can indeed be convinced that she can tell apart Pepsi and Coke.

To formally define this we extend the notion of interaction to *probabilistic* functions (actually, we only need to do so for the verifier). To model an interaction between f and g where f is probabilistic, we add an additional m-bit input r to the function f in (1), that is having $a_1 = f(x, r)$, $a_3 = f(x, r, a_1, a_2)$, etc. The interaction $\langle f, g \rangle(x)$ is now a random variable over $r \in_R \{0, 1\}^m$. Similarly the output $\mathsf{out}_f \langle f, g \rangle(x)$ is also a random variable.

DEFINITION 8.5 (**IP**)
Let $k : \mathbb{N} \to \mathbb{N}$ be some function with $k(n)$ computable in poly(n) time. A language L is in **IP**[k] if there is a Turing machine V such that on inputs $x, r, a_1, \ldots, a_i$, V runs in time polynomial in $|x|$ and such that

$$\text{(Completeness)} \qquad x \in L \Rightarrow \exists P \Pr[\mathsf{out}_V \langle V, P \rangle(x) = 1] \geq 2/3 \qquad (2)$$

$$\text{(Soundness)} \qquad x \notin L \Rightarrow \forall P \Pr[\mathsf{out}_V \langle V, P \rangle(x) = 1] \leq 1/3. \qquad (3)$$

We define **IP** $= \cup_{c \geq 1} \mathbf{IP}[n^c]$.

REMARK 8.6
The following observations on the class **IP** are left as an exercise (Exercise 1).

1. Allowing the prover to be probabilistic (i.e., the answer function a_i depends upon some random string used by the prover) does not change the class **IP**. The reason is that for any language L, if a probabilistic prover P results in making verifier V accept with some probability, then averaging implies there is a deterministic prover which makes V accept with the same probability.

Figure unavailable in pdf file.

Figure 8.1: Two isomorphic graphs.

2. Since the prover can use an arbitrary function, it can in principle use unbounded computational power (or even compute undecidable functions). However, one can show that given any verifier V, we can compute the optimum prover (which, given x, maximizes the verifier's acceptance probability) using $\mathrm{poly}(|x|)$ space (and hence $2^{\mathrm{poly}(|x|)}$ time). Thus **IP** $\subseteq$ **PSPACE**.

3. The probabilities of correctly classifying an input can be made arbitrarily close to 1 by using the same boosting technique we used for **BPP** (see Section **??**): to replace $2/3$ by $1-\exp(-m)$, sequentially repeat the protocol m times and take the majority answer. In fact, using a more complicated proof, it can be shown that we can decrease the probability without increasing the number of rounds using *parallel repetition* (i.e., the prover and verifier will run m executions of the protocol in parallel). We note that the proof is easier for the case of *public coin* proofs, which will be defined below.

4. Replacing the constant $2/3$ in the completeness requirement (2) by 1 does not change the class **IP**. This is a nontrivial fact. It was originally proved in a complicated way but today can be proved using our characterization of **IP** later in Section 8.5.

5. In contrast replacing the constant $2/3$ by 1 in the soundness condition (3) is equivalent to having a deterministic verifier and hence reduces the class **IP** to **NP**.

6. We emphasize that the prover functions do not depend upon the verifier's random strings, but only on the messages/questions the verifier sends. In other words, the verifier's random string is *private*. (Often these are called *private coin* interactive proofs.) Later we will also consider the model where all the verifier's questions are simply obtained by tossing coins and revealing them to the prover (this is known as *public coins* or *Arthur-Merlin* proofs).

8.3 Proving that graphs are *not* isomorphic.

We'll now see an example of a language in **IP** that is not known to be in **NP**. Recall that the usual ways of representing graphs —adjacency lists, adjacency matrices— involve a numbering of the vertices. We say two graphs G_1 and G_2 are *isomorphic* if they are the same up to a renumbering of vertices. In other words, if there is a permutation π of the labels of the nodes of G_1 such that $\pi(G_1) = G_2$. The graphs in figure **??**, for example, are isomorphic with $\pi = (12)(3654)$. (That is, 1 and 2 are mapped to each other, 3 to 6, 6 to 5, 5 to 4 and 4 to 1.) If G_1 and G_2 are isomorphic, we write $G_1 \equiv G_2$. The GI problem is the following: given two graphs G_1, G_2 (say in adjacency matrix representation) decide if they are isomorphic. Note that clearly GI $\in$ **NP**, since a certificate is simply the description of the permutation π.

The graph isomorphism problem is important in a variety of fields and has a rich history (see [**?**]). Along with the factoring problem, it is the most famous **NP**-problem that is not known to be

either in **P** or **NP**-complete. The results of this section show that GI is unlikely to be **NP**-complete, unless the polynomial hierarchy collapses. This will follow from the existence of the following proof system for the complement of GI: the problem GNI of deciding whether two given graphs are *not* isomorphic.

Protocol: Private-coin Graph Non-isomorphism

V: pick $i \in \{1, 2\}$ uniformly randomly. Randomly permute the vertices of G_i to get a new graph H. Send H to P.

P: identify which of G_1, G_2 was used to produce H. Let G_j be that graph. Send j to V.

V: accept if $i = j$; reject otherwise.

To see that Definition 8.5 is satisfied by the above protocol, note that if $G_1 \not\equiv G_2$ then there exists a prover such that $\Pr[V \text{ accepts}] = 1$, because if the graphs are non-isomorphic, an all-powerful prover can certainly tell which one of the two is isomorphic to H. On the other hand, if $G_1 \equiv G_2$ the best any prover can do is to randomly guess, because a random permutation of G_1 looks exactly like a random permutation of G_2. Thus in this case for every prover, $\Pr[V \text{ accepts}] \leq 1/2$. This probability can be reduced to $1/3$ by sequential or parallel repetition.

8.4 Public coins and AM

Allowing the prover full access to the verifier's random string leads to the model of *interactive proofs with public-coins*.

DEFINITION 8.7 (AM, MA)
For every k we denote by $\mathbf{AM}[k]$ the class of languages that can be decided by a k round interactive proof in which each verifier's message consists of sending a random string of polynomial length, and these messages comprise of all the coins tossed by the verifier. A proof of this form is called a *public coin* proof (it is sometimes also known an *Arthur Merlin* proof).[1]

We define by $\mathbf{AM}$ the class $\mathbf{AM}[2]$.[2] That is, $\mathbf{AM}$ is the class of languages with an interactive proof that consist of the verifier sending a random string, the prover responding with a message, and where the decision to accept is obtained by applying a deterministic polynomial-time function to the transcript. The class $\mathbf{MA}$ denotes the class of languages with 2-round public coins interactive proof with the prover sending the first message. That is, $L \in \mathbf{MA}$ if there's a proof system for L that consists of the prover first sending a message, and then the verifier tossing coins and applying a polynomial-time predicate to the input, the prover's message and the coins.

---

[1]Arthur was a famous king of medieval England and Merlin was his court magician. Babai named these classes by drawing an analogy between the prover's infinite power and Merlin's magic. One "justification" for this model is that while Merlin cannot predict the coins that Arthur will toss in the future, Arthur has no way of hiding from Merlin's magic the results of the coins he tossed in the past.

[2]Note that $\mathbf{AM} = \mathbf{AM}[2]$ while $\mathbf{IP} = \mathbf{IP}[\text{poly}]$. While this is indeed somewhat inconsistent, this is the standard notation used in the literature. We note that some sources denote the class $\mathbf{AM}[3]$ by $\mathbf{AMA}$, the class $\mathbf{AM}[4]$ by $\mathbf{AMAM}$ etc.

Note that clearly for every k, $\mathbf{AM}[k] \subseteq \mathbf{IP}[k]$. The interactive proof for GNI seemed to crucially depend upon the fact that P cannot see the random bits of V. If P knew those bits, P would know i and so could trivially always guess correctly. Thus it may seem that allowing the verifier to keep its coins private adds significant power to interactive proofs, and so the following result should be quite surprising:

THEOREM 8.8 ([GS87])
For every $k : \mathbb{N} \to \mathbb{N}$ with $k(n)$ computable in poly(n),

$$\mathbf{IP}[k] \subseteq \mathbf{AM}[k+2]$$

The central idea of the proof of Theorem 8.8 can be gleaned from the proof for the special case of GNI.

THEOREM 8.9
GNI $\in \mathbf{AM}[k]$ *for some constant $k \geq 2$.*

The key idea in the proof of Theorem 8.9 is to look at graph nonisomorphism in a different, more quantitative, way. (Aside: This is a good example of how nontrivial interactive proofs can be designed by recasting the problem.) Consider the set $S = \{H : H \equiv G_1 \text{ or } H \equiv G_2\}$. Note that it is easy to prove that a graph H is a member of S, by providing the permutation mapping either G_1 or G_2 to H. The size of this set depends on whether G_1 is isomorphic to G_2. An n vertex graph G has at most $n!$ equivalent graphs. If G_1 and G_2 have each exactly $n!$ equivalent graphs (this will happen if for $i = 1, 2$ there's no non-identity permutation π such that $\pi(G_i) = G_i$) we'll have that

$$\text{if } G_1 \not\equiv G_2 \text{ then } |S| = 2n! \tag{4}$$
$$\text{if } G_1 \equiv G_2 \text{ then } |S| = n! \tag{5}$$

(To handle the general case that G_1 or G_2 may have less than $n!$ equivalent graphs, we actually change the definition of S to

$$S = \{(H, \pi) : H \equiv G_1 \text{ or } H \equiv G_2 \text{ and } \pi \in aut(H)\}$$

where $\pi \in aut(H)$ if $\pi(H) = H$. It is clearly easy to prove membership in the set S and it can be verified that S satisfies (4) and (5).)

Thus to convince the verifier that $G_1 \not\equiv G_2$, the prover has to convince the verifier that case (4) holds rather than (5). This is done by using a *set lower bound protocol*.

8.4.1   Set Lower Bound Protocol.

In a *set lower bound protocol*, the prover proves to the verifier that a given set S (where membership in S is efficiently verifiable) has cardinality at least K up to accuracy of, say, factor of 2. That is, if $|S| \geq K$ then the prover can cause the verifier to accept with high probability, while if $|S| \leq K/2$ then the verifier will reject with high probability, no matter what the prover does. By the observations above, such a protocol suffices to complete the proof of Theorem 8.9.

Tool: Pairwise independent hash functions.

The main tool we use for the set lower bound protocol is a *pairwise independent hash function collection*. This is a simple but incredibly useful tool that has found numerous applications in complexity theory and computer science at large (see Note 8.13).

DEFINITION 8.10 (PAIRWISE INDEPENDENT HASH FUNCTIONS)
Let $\mathcal{H}_{n,k}$ be a collection of functions from $\{0,1\}^n$ to $\{0,1\}^k$. We say that $\mathcal{H}_{n,k}$ is *pairwise independent* if for every $x, x' \in \{0,1\}^n$ with $x \neq x'$ and for every $y, y' \in \{0,1\}^k$, $\Pr_{h \in_R \mathcal{H}_{n,k}}[h(x) = y \;\wedge\; h(x') = y'] = 2^{-2n}$

Note that an equivalent formulation is that for every two distinct strings $x, x' \in \{0,1\}^n$ the random variable $\langle h(x), h(x') \rangle$ for h chosen at random from $\mathcal{H}_{n,k}$ is distributed according to the uniform distribution on $\{0,1\}^k \times \{0,1\}^k$.

Recall that we can identify the elements of $\{0,1\}^n$ with the *finite field* (see Section A.4 in the appendix), denoted $\mathrm{GF}(2^n)$, containing 2^n elements, whose addition $(+)$ and multiplication $(\cdot)$ operations satisfy the usual commutative and distributive laws, where and every element x has an additive inverse (denoted by $-x$) and, if nonzero, a multiplicative inverse (denoted by x^{-1}). The following theorem provides a construction of an *efficiently computable* pairwise independent hash functions (see also Exercise 4 for a different construction):

THEOREM 8.11 (EFFICIENT PAIRWISE INDEPENDENT HASH FUNCTIONS)
For every n define the collection $\mathcal{H}_{n,n}$ to be $\{h_{a,b}\}_{a,b\in\mathrm{GF}(2^n)}$ where for every $a, b \in \mathrm{GF}(2^n)$, the function $h_{a,b} : \mathrm{GF}(2^n) \to \mathrm{GF}(2^n)$ maps x to $ax + b$. Then, $\mathcal{H}_{n,n}$ is a collection of pairwise independent hash functions.

REMARK 8.12
Theorem 8.11 implies the existence of an efficiently computable pairwise independent hash functions $\mathcal{H}_{n,k}$ for every n, k: if $k > n$ we can use the collection $\mathcal{H}_{k,k}$ and reduce the size of the input to n by padding it with zeros. If $k < n$ then we can use the collection $\mathcal{H}_{n,n}$ and truncate the last $n - k$ bits of the output.

PROOF: For every $x \neq x' \in \mathrm{GF}(2^n)$ and $y, y' \in \mathrm{GF}(2^n)$, $h_{a,b}(x) = y$ and $h_{a,b}(x') = y'$ iff a, b satisfy the equations:

$$a \cdot x + b = y$$
$$a \cdot x' + b = y'$$

These imply $a \cdot (x - x') = y - y'$ or $a = (y - y')(x - x')^{-1}$. Since $b = y - a \cdot x$, the pair $\langle a, b \rangle$ is completely determined by these equations, and so the probability that this happens over the choice of a, b is exactly one over the number of possible pairs, which indeed equals $\frac{1}{2^{2n}}$. ∎

NOTE 8.13 (THE HASHING PARADIGM)

A *hash function collection* is a collection of functions mapping a large universe, say $\{0,1\}^n$, to a smaller universe, say $\{0,1\}^k$ for $k \ll n$. Typically, we require of such a collection that it maps its input in a fairly uniform way to the output range. For example, if S is a subset of $\{0,1\}^n$ then we wish that, if h is chosen at random from the collection, then most elements of $\{0,1\}^k$ have roughly $|S|2^{-k}$ preimages in S (which is the expected number if h was a completely random function). In particular, if S has size roughly 2^k then we expect the mapping to be one-to-one or almost one-to-one, and so there should be a relatively small number of *collisions*: pairs $x \neq x' \in S$ such that $h(x) = h(x')$. Therefore, the image of S under h should look like this:
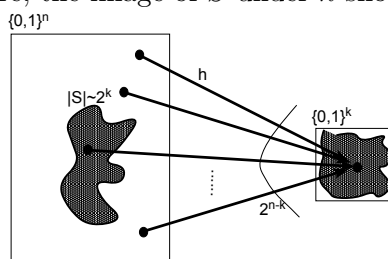
In databases, hash functions are used to maintain very efficient databases (that allow fast membership queries to a subset $S \subseteq \{0,1\}^n$ of size 2^k requiring only 2^k as opposed to 2^n bits of storage). In theoretical computer science, hash functions have a variety of uses. An example is Lemma 9.16 of the next chapter that shows that if the collection is pairwise independent and $S \subseteq \{0,1\}^n$ has size roughly 2^k, then with good probability the value 0^k will have exactly one preimage in S.

In all these cases it is important that the hash function is chosen at random from some collection independently of the choice of set S. It is easy to see that if k is small enough (e.g., $k < n/2$) then for *every* $h : \{0,1\}^n \to \{0,1\}^k$ there is a set $S \subseteq \{0,1\}^n$ of size 2^k that is "very bad" for h in the sense that all the members of S map to the same element under h.

Pairwise independent hash functions are but one example of a hash function collection. Several types of such collections are known in the literature featuring various tradeoffs between efficiency and uniformity of output.

DRAFT

The lower-bound protocol.

The lower-bound protocol is as follows:

Protocol: Goldwasser-Sipser Set Lowerbound

Conditions: $S \subseteq \{0,1\}^m$ is a set such that membership in S can be certified. Both parties know a number K. The prover's goal is to convince the verifier that $|S| \geq K$ and the verifier should reject if $|S| \leq \frac{K}{2}$. Let k be a number such that $2^{k-2} \leq K \leq 2^{k-1}$.

V: Randomly pick a function $h : \{0,1\}^m \rightarrow \{0,1\}^k$ from a pairwise independent hash function collection $\mathcal{H}_{m,k}$. Pick $y \in_R \{0,1\}^k$. Send h, y to prover.

P: Try to find an $x \in S$ such that $h(x) = y$. Send such an x to V, together with a certificate that $x \in S$.

V's output: If certificate validates that $x \in S$ and $h(x) = y$, accept; otherwise reject.

Let $p = \frac{K}{2^k}$. If $|S| \leq \frac{K}{2}$ then clearly $|h(S)| \leq \frac{p2^k}{2}$ and so the verifier will accept with probability at most $\frac{p}{2}$. The main challenge is to show that if $|S| \geq K$ then the verifier will accept with probability noticeably larger than $p/2$ (the gap between the probabilities can then be amplified using repetition). That is, it suffices to prove

CLAIM 8.13.1
Let $S \subseteq \{0,1\}^m$ satisfy $|S| \leq \frac{2^k}{2}$. Then,

$$\Pr_{h \in_R \mathcal{H}_{m,k}, y \in_R \{0,1\}^k}[\exists_{x \in S} h(x) = y] \geq \frac{3}{4}\frac{|S|}{2^k}\,.$$

PROOF: For every $y \in \{0,1\}^m$, we'll prove the claim by showing that

$$\Pr_{h \in_R \mathcal{H}_{m,k}}[\exists_{x \in S} h(x) = y] \geq \frac{3}{4}p\,,$$

(where $p = |S|/2^k$). Indeed, for every $x \in S$ define the event E_x to hold if $h(x) = y$. Then, $\Pr[\exists_{x \in S} h(x) = y] = \Pr[\cup_{x \in S} E_x]$ but by the inclusion-exclusion principle this is at least

$$\sum_{x \in S} \Pr[E_x] - \tfrac{1}{2} \sum_{x \neq x' \in \S} \Pr[E_x \cap E'_x]$$

However, by pairwise independence, if $x \neq x'$, then $\Pr[E_x] = 2^{-k}$ and $\Pr[E_x \cap E'_x] = 2^{-2k}$ and so this probability is at least

$$\frac{|S|}{2^k} - \frac{1}{2}\frac{|S|^2}{2^k} = \frac{|S|}{2^k}\left(1 - \frac{|S|}{2^{k+1}}\right) \geq \frac{3}{4}p$$

■

Figure unavailable in pdf file.

Figure 8.2: **AM**[k] looks like $\prod_k^p$, with the $\forall$ quantifier replaced by probabilitic choice.

Proving Theorem 8.9. The public-coin interactive proof system for GNI consists of the verifier and prover running several iterations of the set lower bound protocol for the set S as defined above, where the verifier accepts iff the fraction of accepting iterations was at least $0.6p$ (note that both parties can compute p). Using the Chernoff bound (Theorem A.18) it can be easily seen that a constant number of iteration will suffices to ensure completeness probability at least $\frac{2}{3}$ and soundness error at most $\frac{1}{3}$. ■

REMARK 8.14
How does this protocol relate to the private coin protocol of Section 8.3? The set S roughly corresponds to the set of possible messages sent by the verifier in the protocol, where the verifier's message is a random element in S. If the two graphs are isomorphic then the verifier's message completely hides its choice of a random $i \in_R \{1, 2\}$, while if they're not then it completely reveals it (at least to a prover that has unbounded computation time). Thus roughly speaking in the former case the mapping from the verifier's coins to the message is 2-to-1 while in the latter case it is 1-to-1, resulting in a set that is twice as large. Indeed we can view the prover in the public coin protocol as convincing the verifier that its probability of convincing the private coin verifier is large. While there are several additional intricacies to handle, this is the idea behind the generalization of this proof to show that **IP**[k] $\subseteq$ **AM**[k + 2].

REMARK 8.15
Note that, unlike the private coins protocol, the public coins protocol of Theorem 8.9 does not enjoy perfect completeness, since the set lowerbound protocol does not satisfy this property. However, we can construct a perfectly complete public-coins set lowerbound protocol (see Exercise 3), thus implying a perfectly complete public coins proof for GNI. Again, this can be generalized to show that any private-coins proof system (even one not satisfying perfect completeness) can be transformed into a perfectly complete public coins system with a similar number of rounds.

8.4.2 Some properties of IP and AM

We state the following properties of **IP** and **AM** without proof:

1. (Exercise 5) **AM**[2] = $BP \cdot$ **NP** where $BP \cdot$ **NP** is the class in Definition **??**. In particular it follows that**AM**[2] $\subseteq \mathbf{\Sigma}_3^p$.

2. (Exercise 4) For constants $k \geq 2$ we have **AM**[k] = **AM**[2]. This "collapse" is somewhat surprising because **AM**[k] at first glance seems similar to **PH** with the $\forall$ quantifiers changed to "probabilistic $\forall$" quantifiers, where *most* of the branches lead to acceptance. See Figure 8.2.

3. It is open whether there is any nice characterization of **AM**[$\sigma(n)$], where $\sigma(n)$ is a suitably slow growing function of n, such as $\log \log n$.

8.4.3 Can GI be NP-complete?

We now prove that if GI is **NP**-complete then the polynomial hierarchy collapses.

THEOREM 8.16 ([**?**])
If GI *is* **NP***-complete then* $\Sigma_2 = \Pi_2$.

PROOF: If GI is **NP**-complete then GNI is **coNP**-complete which implies that there exists a function f such that for every n variable formula φ, $\forall_y \varphi(y)$ holds iff $f(\varphi) \in$ GNI. Let

$$\psi = \exists_{x \in \{0,1\}^n} \forall_{y \in \{0,1\}^n} \varphi(x,y)$$

be a Σ_2SAT formula. We have that ψ is equivalent to

$$\exists_{x \in \{0,1\}^n} g(x) \in \text{GNI}$$

where $g(x) = f(\varphi_{\restriction x})$.

Using Remark 8.15 and the comments of Section 8.4.2, we have that GNI has a two round **AM** proof with perfect completeness and (after appropriate amplification) soundness error less than 2^{-n}. Let V be the verifier algorithm for this proof system, and denote by m the length of the verifier's random tape and by m' the length of the prover's message and . We claim that ψ is equivalent to

$$\psi^* = \forall_{r \in \{0,1\}^{m'}} \exists_{x \in \{0,1\}^n} \exists_{a \in \{0,1\}^m} V(g(x), r, a) = 1$$

Indeed, by perfect completeness if ψ is satisfiable then ψ^* is satisfiable. If ψ is not satisfiable then by the fact that the soundness error is at most 2^{-n}, we have that there exists a single string $r \in \{0,1\}^m$ such that for every x with $g(x) \notin$ GNI, there's no a such that $V(g(x), r, a) = 1$, and so ψ^* is not satisfiable. Since ψ^* can easily be reduced to a Π_2SAT formula, we get that $\Sigma_2 \subseteq \Pi_2$, implying (since $\Sigma_2 = \text{co}\Pi_2$) that $\Sigma_2 = \Pi_2$. ∎

8.5 IP = PSPACE

In this section we show a surprising characterization of the set of languages that have interactive proofs.

THEOREM 8.17 (LFKN, SHAMIR, 1990)
IP = PSPACE.

Note that this is indeed quite surprising: we already saw that interaction alone does not increase the languages we can prove beyond **NP**, and we tend to think of randomization as not adding significant power to computation (e.g., we'll see in Chapter 16 that under reasonable conjectures, **BPP = P**). As noted in Section 8.4.2, we even know that languages with constant round interactive proofs have a two round public coins proof, and are in particular contained in the polynomial hierarchy, which is believed to be a proper subset of **PSPACE**. Nonetheless, it turns out that the combination of sufficient interaction and randomness is quite powerful.

By our earlier Remark 8.6 we need only show the direction **PSPACE** $\subseteq$ **IP**. To do so, we'll show that $\mathsf{TQBF} \in \mathbf{IP}[poly(n)]$. This is sufficient because every $L \in \mathbf{PSPACE}$ is polytime reducible to TQBF. We note that our protocol for TQBF will use public coins and also has the property that if the input is in TQBF then there is a prover which makes the verifier accept with probability 1.

Rather than tackle the job of designing a protocol for TQBF right away, let us first think about how to design one for $\overline{\mathsf{3SAT}}$. How can the prover convince the verifier than a given 3CNF formula has no satisfying assignment? We show how to prove something even more general: the prover can prove to the verifier what the *number* of satisfying assignments is. (In other words, we will design a prover for #SAT.) The idea of *arithmetization* introduced in this proof will also prove useful in our protocol for TQBF.

8.5.1   Arithmetization

The key idea will be to take an algebraic view of boolean formulae by representing them as polynomials. Note that $0, 1$ can be thought of both as truth values and as elements of some finite field $\mathbb{F}$. Thus we have the following correspondence between formulas and polynomials when the variables take 0/1 values:

$$
\begin{aligned}
x \wedge y &\longleftrightarrow X \cdot Y \\
\neg x &\longleftrightarrow 1 - X \\
x \vee y &\longleftrightarrow 1 - (1 - X)(1 - Y) \\
x \vee y \vee \neg z &\longleftrightarrow 1 - (1 - X)(1 - Y)Z
\end{aligned}
$$

Given any 3CNF formula $\varphi(x_1, x_2, \ldots, x_n)$ with m clauses, we can write such a degree 3 polynomial for each clause. Multiplying these polynomials we obtain a degree $3m$ multivariate polynomial $P_\varphi(X_1, X_2, \ldots, X_n)$ that evaluates to 1 for satisfying assignments and evaluates to 0 for unsatisfying assignments. (Note: we represent such a polynomial as a multiplication of all the degree 3 polynomials without "opening up" the parenthesis, and so $P_\varphi(X_1, X_2, \ldots, X_n)$ has a representation of size $O(m)$.) This conversion of φ to P_φ is called *arithmetization*. Once we have written such a polynomial, nothing stops us from going ahead and and evaluating the polynomial when the variables take arbitrary values from the field $\mathbb{F}$ instead of just $0, 1$. As we will see, this gives the verifier unexpected power over the prover.

8.5.2   Interactive protocol for #SAT$_\mathsf{D}$

To design a protocol for $\overline{\mathsf{3SAT}}$ we give a protocol for #SAT$_\mathsf{D}$, which is a decision version of the counting problem #SAT we saw in Chapter **??**:

$$\#\mathsf{SAT}_\mathsf{D} = \{\langle \phi, K \rangle : K \text{ is the number of satisfying assignments of } \phi\} .$$

and ϕ is a 3CNF formula of n variables and m clauses.

THEOREM 8.18
#SAT$_\mathsf{D} \in \mathbf{IP}$.

PROOF: Given input $\langle \phi, K \rangle$, we construct, by arithmetization, P_ϕ. The number of satisfying assignments $\#\phi$ of ϕ is:

$$\#\phi = \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} P_\phi(b_1, \ldots, b_n) \tag{6}$$

To start, the prover sends to the verifier a prime p in the interval $(2^n, 2^{2n}]$. The verifier can check that p is prime using a probabilistic or deterministic primality testing algorithm. All computations described below are done in the field $\mathbb{F} = \mathbb{F}_p$ of numbers modulo p. Note that since the sum in (6) is between 0 and 2^n, this equation is true over the integers iff it is true modulo p. Thus, from now on we consider (6) as an equation in the field $\mathbb{F}_p$. We'll prove the theorem by showing a general protocol, *Sumcheck*, for verifying equations such as (6).

Sumcheck protocol.

Given a degree d polynomial $g(X_1, \ldots, X_n)$, an integer K, and a prime p, we present an interactive proof for the claim

$$K = \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(X_1, \ldots, X_n) \tag{7}$$

(where all computations are modulo p). To execute the protocol V will need to be able to evaluate the polynomial g for any setting of values to the variables. Note that this clearly holds in the case $g = P_\phi$.

For each sequence of values $b_2, b_3, \ldots, b_n$ to $X_2, X_3, \ldots, X_n$, note that $g(X_1, b_2, b_3, \ldots, b_n)$ is a univariate degree d polynomial in the variable X_1. Thus the following is also a univariate degree d polynomial:

$$h(X_1) = \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(X_1, b_2 \ldots, b_n)$$

If Claim (7) is true, then we have $h(0) + h(1) = K$.

Consider the following protocol:

Protocol: Sumcheck protocol to check claim (7)

V: If $n = 1$ check that $g(1) + g(0) = K$. If so accept, otherwise reject. If $n \geq 2$, ask P to send $h(X_1)$ as defined above.

P: Sends some polynomial $s(X_1)$ (if the prover is not "cheating" then we'll have $s(X_1) = h(X_1)$).

V: Reject if $s(0) + s(1) \neq K$; otherwise pick a random a. Recursively use the same protocol to check that

$$s(a) = \sum_{b \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(a, b_2, \ldots, b_n).$$

DRAFT

If Claim (7) is true, the prover that always returns the correct polynomial will always convince V. If (7) is false then we prove that V rejects with high probability:

$$\Pr[V \text{ rejects } \langle K, g \rangle] \geq \left(1 - \frac{d}{p}\right)^n. \tag{8}$$

With our choice of p, the right hand side is about $1 - dn/p$, which is very close to 1 since $d \leq n^3$ and $p \gg n^4$.

Assume that (7) is false. We prove (8) by induction on n. For $n = 1$, V simply evaluates $g(0), g(1)$ and rejects with probability 1 if their sum is not K. Assume the hypothesis is true for degree d polynomials in $n - 1$ variables.

In the first round, the prover P is supposed to return the polynomial h. If it indeed returns h then since $h(0) + h(1) \neq K$ by assumption, V will immediately reject (i.e., with probability 1). So assume that the prover returns some $s(X_1)$ different from $h(X_1)$. Since the degree d nonzero polynomial $s(X_1) - h(X_1)$ has at most d roots, there are at most d values a such that $s(a) = h(a)$. Thus when V picks a random a,

$$\Pr_a[s(a) \neq h(a)] \geq 1 - \frac{d}{p}. \tag{9}$$

If $s(a) \neq h(a)$ then the prover is left with an incorrect claim to prove in the recursive step. By the induction hypothesis, the prover fails to prove this false claim with probability at least $\geq \left(1 - \frac{d}{p}\right)^{n-1}$. Thus we have

$$\Pr[V \text{ rejects}] \geq \left(1 - \frac{d}{p}\right) \cdot \left(1 - \frac{d}{p}\right)^{n-1} = \left(1 - \frac{d}{p}\right)^n \tag{10}$$

This finishes the induction. ∎

8.5.3   Protocol for TQBF: proof of Theorem 8.17

We use a very similar idea to obtain a protocol for TQBF. Given a quantified Boolean formula $\Psi = \exists x_1 \forall x_2 \exists x_3 \cdots \forall x_n \phi(x_1, \ldots, x_n)$, we use arithmetization to construct the polynomial P_ϕ. We have that $\Psi \in$ TQBF if and only if

$$0 < \sum_{b_1 \in \{0,1\}} \prod_{b_2 \in \{0,1\}} \sum_{b_3 \in \{0,1\}} \cdots \prod_{b_n \in \{0,1\}} P_\phi(b_1, \ldots, b_n) \tag{11}$$

A first thought is that we could use the same protocol as in the #SAT$_D$ case, except check that $s(0) \cdot s(1) = K$ when you have a $\prod$. But, alas, multiplication, unlike addition, increases the degree of the polynomial — after k steps, the degree could be 2^k. Such polynomials may have 2^k coefficients and cannot even be transmitted in polynomial time if $k \gg \log n$.

The solution is to look more closely at the polynomials that are are transmitted and their relation to the original formula. We'll change Ψ into a logically equivalent formula whose arithmetization

does not cause the degrees of the polynomials to be so large. The idea is similar to the way circuits are reduced to formulas in the Cook-Levin theorem: we'll add auxiliary variables. Specifically, we'll change ψ to an equivalent formula ψ' that is not in prenex form in the following way: work from right to left and whenever encountering a $\forall$ quantifier on a variable x_i — that is, when considering a postfix of the form $\forall_{x_i}\tau(x_1,\ldots,x_i)$, where τ may contain quantifiers over additional variables $x_{i+1},\ldots,x_n$ — ensure that the variables $x_1,\ldots,x_i$ never appear to the right of another $\forall$ quantifier in τ by changing the postfix to $\forall x_i \exists x_1',\ldots,x_i'(x_1'=x_1)\wedge\cdots\wedge(x_i'=x_i)\wedge\tau(x_1,\ldots,x_n)$. Continuing this way we'll obtain the formula ψ' which will have $O(n^2)$ variables and will be at most $O(n^2)$ larger than ψ. It can be seen that the natural arithmetization for ψ' will lead to the polynomials transmitted in the sumcheck protocol never having degree more than 2.

Note that the prover needs to prove that the arithmetization of Ψ' leads to a number K different than 0, but because of the multiplications this number can be as large as 2^{2^n}. Nevertheless the prover can find a prime p between 0 and 2^n such that $K \mod p \neq 0$ (in fact as we saw in Chapter 7 a random prime will do). This finishes the proof of Theorem 8.17. ∎

REMARK 8.19
An alternative way to obtain the same result (or, more accurately, an alternative way to describe the same protocol) is to notice that for $x \in \{0,1\}$, $x^k = x$ for all $k \geq 1$. Thus, in principle we can convert any polynomial $p(x_1,\ldots,x_n)$ into a *multilinear* polynomial $q(x_1,\ldots,x_n)$ (i.e., the degree of $q(\cdot)$ in any variable x_i is at most one) that agrees with $p(\cdot)$ on all $x_1,\ldots,x_n \in \{0,1\}$. Specifically, for any polynomial $p(\cdot)$ let $L_i(p)$ be the polynomial defined as follows

$$L_i(p)(x_1,\ldots,x_n) = x_i P(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n)+$$
$$(1-x_i)P(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n) \quad (12)$$

then $L_1(L_2(\cdots(L_n(p)\cdots)$ is such a multilinear polynomial agreeing with $p(\cdot)$ on all values in $\{0,1\}$. We can thus use $O(n^2)$ invocations operator to convert (11) into an equivalent form where all the intermediate polynomials sent in the sumcheck protocol are multilinear. We'll use this equivalent form to run the sumcheck protocol, where in addition to having round for a $\sum$ or $\prod$ operator, we'll also have a round for each application of the operator L (in such rounds the prover will send a polynomial of degree at most 2).

8.6 The power of the prover

A curious feature of many known interactive proof systems is that in order to prove membership in language L, the prover needs to do more powerful computation than just deciding membership in L. We give some examples.

1. The public coin system for graph nonisomorphism in Theorem 8.9 requires the prover to produce, for some randomly chosen hash function h and a random element y in the range of h, a graph H such that $h(H)$ is isomorphic to either G_1 or G_2 and $h(x) = y$. This seems harder than just solving graph non-isomorphism.

2. The interactive proof for $\overline{\mathsf{3SAT}}$, a language in **coNP**, requires the prover to do $\#\mathbf{P}$ computations, doing summations of exponentially many terms. (Recall that all of **PH** is in $\mathbf{P}^{\#\mathbf{P}}$.)

In both cases, it is an open problem whether the protocol can be redesigned to use a weaker prover.

Note that the protocol for TQBF is different in that the prover's replies can be computed in **PSPACE** as well. This observation underlies the following result, which is in the same spirit as the Karp-Lipton results described in Chapter **??**, except the conclusion is stronger since **MA** is contained in $\mathbf{\Sigma_2}$ (indeed, a perfectly complete **MA**-proof system for L trivially implies that $L \in \mathbf{\Sigma_2}$).

THEOREM 8.20
If $\mathbf{PSPACE} \subseteq \mathbf{P}/poly$ then $\mathbf{PSPACE} = \mathbf{MA}$.

PROOF: If $\mathbf{PSPACE} \subseteq \mathbf{P}/poly$ then the prover in our TQBF protocol can be replaced by a circuit of polynomial size. Merlin (the prover) can just give this circuit to Arthur (the verifier) in Round 1, who then runs the interactive proof using this "prover." No more interaction is needed. Note that there is no need for Arthur to put blind trust in Merlin's circuit, since the correctness proof of the TQBF protocol shows that if the formula is not true, then *no* prover can make Arthur accept with high probability. ∎

In fact, using the Karp-Lipton theorem one can prove a stronger statement, see Lemma **??** below.

8.7 Program Checking

The discovery of the interactive protocol for the permanent problem was triggered by a field called *program checking*. Blum and Kannan's motivation for introducing this field was the fact that program verification (deciding whether or not a given program solves a certain computational task) is undecidable. They observed that in many cases we can guarantee a weaker guarantee of the program's "correctness" on an instance by instance basis. This is encapsulated in the notion of a *program checker*. A checker C for a program P is itself another program that may run P as a subroutine. Whenever P is run on an input x, C's job is to detect if P's answer is incorrect ("buggy") on that particular instance x. To do this, the checker may also compute P's answer on some other inputs. Program checking is sometimes also called *instance checking*, perhaps a more accurate name, since the fact that the checker did not detect a bug does not mean that P is a correct program in general, but only that P's answer on x is correct.

DEFINITION 8.21
Let P be a claimed program for computational task T. A **checker** for T is a probabilistic polynomial time TM, C, that, given any x, has the following behavior:

1. If P is a correct program for T (i.e., $\forall y \ \ P(y) = T(y)$), then $P[C^P \text{ accepts } P(x)] \geq \frac{2}{3}$

2. If $P(x) \neq T(x)$ then $P[C^P \text{ accepts } P(x)] < \frac{1}{3}$

Note that in the case that P is correct on x (i.e., $P(x) = C(x)$) but the program P is not correct everywhere, there is no guarantee on the output of the checker.

Surprisingly, for many problems, checking seems easier than actually computing the problem. (Blum and Kannan's suggestion was to build checkers into the software whenever this is true; the overhead introduced by the checker would be negligible.)

EXAMPLE 8.22 (CHECKER FOR GRAPH NON-ISOMORPHISM)
The input for the problem of Graph Non-Isomorphism is a pair of labelled graphs $\langle G_1, G_2 \rangle$, and the problem is to decide whether $G_1 \equiv G_2$. As noted, we do not know of an efficient algorithm for this problem. But it has an efficient checker.

There are two types of inputs, depending upon whether or not the program claims $G_1 \equiv G_2$. If it claims that $G_1 \equiv G_2$ then one can change the graph little by little and use the program to actually obtain the permutation π (). We now show how to check the claim that $G_1 \not\equiv G_2$ using our earlier interactive proof of Graph non-isomorphism.

Recall the IP for Graph Non-Isomorphism:

- In case prover admits $G_1 \not\equiv G_2$ repeat k times:

- Choose $i \in_R \{1, 2\}$. Permute G_i randomly into H

- Ask the prover $\langle G_1, H \rangle; \langle G_2, H \rangle$ and check to see if the prover's first answer is consistent.

Given a computer program that supposedly computes graph isomorphism, P, how would we check its correctness? The program checking approach suggests to use an IP while regarding the program as the prover. Let C be a program that performs the above protocol with P as the prover, then:

THEOREM 8.23
If P is a correct program for Graph Non-Isomorphism then C outputs "correct" always. Otherwise, if $P(G_1, G_2)$ is incorrect then $P[C$ outputs "correct" $] \leq 2^{-k}$. Moreover, C runs in polynomial time.

8.7.1   Languages that have checkers

Whenever a language L has an interactive proof system where the prover can be implemented using oracle access to L, this implies that L has a checker. Thus, the following theorem is a direct consequence of the interactive proofs we have seen:

THEOREM 8.24
The problems Graph Isomorphism (GI), Permanent (perm) and True Quantified Boolean Formulae (TQBF) have checkers.

Using the fact that **P**-complete languages are reducible to each other via **NC**-reductions, it suffices to show a checker in **NC** for one **P**-complete language (as was shown by Blum & Kannan) to obtain the following interesting fact:

THEOREM 8.25
For any **P***-complete language there exists a program checker in* **NC**

Since we believe that **P**-complete languages cannot be computed in **NC**, this provides additional evidence that checking is easier than actual computation.

8.8 Multiprover interactive proofs (MIP)

It is also possible to define interactive proofs that involve more than one prover. The important assumption is that the provers do not communicate with each other *during the protocol*. They may communicate *before* the protocol starts, and in particular, agree upon a shared strategy for answering questions. (The analogy often given is that of the police interrogating two suspects in separate rooms. The suspects may be accomplices who have decided upon a common story to tell the police, but since they are interrogated separately they may inadvertently reveal an inconsistency in the story.)

The set of languages with multiprover interactive provers is call **MIP**. The formal definition is analogous to Definition 8.5. We assume there are two provers (though one can also study the case of polynomially many provers; see the exercises), and in each round the verifier sends a query to each of them —the two queries need not be the same. Each prover sends a response in each round.

Clearly, **IP** $\subseteq$ **MIP** since we can always simply ignore one prover. However, it turns out that **MIP** is probably strictly larger than **IP** (unless **PSPACE** = **NEXP**). That is, we have:

THEOREM 8.26 ([BFL91])
NEXP = **MIP**

We will outline a proof of this theorem in Chapter **??**. One thing that we can do using two rounds is to force *non-adaptivity*. That is, consider the interactive proof as an "interrogation" where the verifier asks questions and gets back answers from the prover. If the verifier wants to ensure that the answer of a prover to the question q is a function only of q and does not depend on the previous questions the prover heard, the prover can ask the second prover the question q and accept only if both answers agree with one another. This technique was used to show that multi-prover interactive proofs can be used to implement (and in fact are equivalent to) a model of a "probabilistically checkable proof in the sky". In this model we go back to an **NP**-like notion of a proof as a static string, but this string may be huge and so is best thought of as a huge table, consisting of the prover's answers to all the possible verifier's questions. The verifier checks the proof by looking at only a few entries in this table, that are chosen randomly from some distribution. If we let the class **PCP**$[r, q]$ be the set of languages that can be proven using a table of size 2^r and q queries to this table then Theorem 8.26 can be restated as

THEOREM 8.27 (THEOREM 8.26, RESTATED)
NEXP = **PCP**$[poly, poly]$ = $\cup_c$**PCP**$[n^c, n^c]$

It turns out Theorem 8.26 can be scaled down to to obtain **NP** = **PCP**$[\text{polylog}, \text{polylog}]$. In fact (with a lot of work) the following is known:

THEOREM 8.28 (THE **PCP** THEOREM, [AS98, ALM$^+$98])
NP = **PCP**$[O(\log n), O(1)]$

This theorem, which will be proven in Chapter 18, has had many applications in complexity, and in particular establishing that for many **NP**-complete optimization problems, obtaining an *approximately optimal* solution is as hard as coming up with the optimal solution itself. Thus, it seems that complexity theory has gone a full circle with interactive proofs: by adding interaction, randomization, and multiple provers, and getting to classes as high as **NEXP**, we have gained new and fundamental insights on the class **NP** the represents static deterministic proofs (or equivalently, efficiently verifiable search problems).

WHAT HAVE WE LEARNED?

- An *interactive proof* is a generalization of mathematical proofs in which the prover and polynomial-time probabilistic verifier interact.

- Allowing randomization and interaction seems to add significantly more power to proof system: the class **IP** of languages provable by a polynomial-time interactive proofs is equal to **PSPACE**.

- All languages provable by a *constant round* proof system are in the class **AM**: that is, they have a proof system consisting of the the verifier sending a single random string to the prover, and the prover responding with a single message.

Chapter notes and history

Interactive proofs were defined in 1985 by Goldwasser, Micali, Rackoff [GMR89] for cryptographic applications and (independently, and using the public coin definition) by Babai and Moran [BM88]. The private coins interactive proof for graph non-isomorphism was given by Goldreich, Micali and Wigderson [GMW87]. Simulations of private coins by public coins we given by Goldwasser and Sipser [GS87]. The general feeling at the time was that interactive proofs are only a "slight" extension of **NP** and that not even $\overline{\mathsf{3SAT}}$ has interactive proofs. The result **IP** = **PSPACE** was a big surprise, and the story of its discovery is very interesting.

In the late 1980s, Blum and Kannan [BK95] introduced the notion of program checking. Around the same time, manuscripts of Beaver and Feigenbaum [BF90] and Lipton [Lip91] appeared. Inspired by some of these developments, Nisan proved in December 1989 that #SAT has *multiprover* interactive proofs. He announced his proof in an email to several colleagues and then left on vacation to South America. This email motivated a flurry of activity in research groups around the world. Lund, Fortnow, Karloff showed that #SAT is in **IP** (they added Nisan as a coauthor and the final paper is [LFK92]). Then Shamir showed that **IP** =**PSPACE** [Sha92] and Babai, Fortnow and Lund [BFL91] showed **MIP** = **NEXP**. The entire story —as well as related developments—are described in Babai's entertaining survey [Bab90].

Vadhan [Vad00] explores some questions related to the power of the prover.

The result that approximating the shortest vector is probably not **NP**-hard (as mentioned in the introduction) is due to Goldreich and Goldwasser [GG00].

Exercises

§1 Prove the assertions in Remark 8.6. That is, prove:

(a) Let **IP**$'$ denote the class obtained by allowing the prover to be probabilistic in Definition 8.5. That is, the prover's strategy can be chosen at random from some distribution on functions. Prove that **IP**$'$ = **IP**.

(b) Prove that **IP** $\subseteq$ **PSPACE**.

(c) Let **IP**$'$ denote the class obtained by changing the constant 2/3 in (2) and (3) to $1-2^{-|x|}$. Prove that **IP**$'$ = **IP**.

(d) Let **IP**$'$ denote the class obtained by changing the constant 2/3 in (2) to 1. Prove that **IP**$'$ = **IP**.

(e) Let **IP**$'$ denote the class obtained by changing the constant 2/3 in (3) to 1. Prove that **IP**$'$ = **NP**.

§2 We say integer y is a *quadratic residue modulo m* if there is an integer x such that $y \equiv x^2$ (mod m). Show that the following language is in **IP**[2]:

$$\text{QNR} = \{(y,m): \ y \text{ is not a quadratic residue modulo } m\}.$$

§3 Prove that there exists a perfectly complete **AM**[$O(1)$] protocol for the proving a lowerbound on set size.

Hint: First note that in the current set lowerbound protocol we can have the prover choose the hash function. Consider the easier case of constructing a protocol to distinguish between the case $|S| \geq K$ and $|S| \lesssim \frac{1}{2}K$ where $c > 2$ can be even a function of K. If c is large enough the we can allow the prover to use *several* hash functions $h_1, \ldots, h_i$, and it can be proven that if i is large enough we'll have $\cup_i h_i(S) = \{0,1\}^k$. The gap can be increased by considering instead of S the set S^ℓ, that is the ℓ times Cartesian product of S.

§4 Prove that for every constant $k \geq 2$, **AM**$[k+1] \subseteq$ **AM**$[k]$.

§5 Show that **AM**[2] = $BP \cdot$ **NP**

§6 [BFNW93] Show that if **EXP** $\subseteq$ **P**/*poly* then **EXP** = **MA**.

Hint: The interactive proof for TQBF requires a prover that is a **PSPACE** machine.

§7 Show that the problem GI is downward self reducible. That is, prove that given two graphs G_1, G_2 on n vertices and access to a subroutine P that solves the GI problem on graphs with up to $n - 1$ vertices, we can decide whether or not G_1 and G_2 are isomorphic in polynomial time.

§8 Prove that in the case that G_1 and G_2 are isomorphic we can obtain the permutation π mapping G_1 to G_2 using the procedure of the above exercise. Use this to complete the proof in Example 8.22 and show that graph isomorphism has a checker. Specifically, you have to show that if the program claims that $G_1 \equiv G_2$ then we can do some further investigation (including calling the programs on other inputs) and with high probability conclude that either (a) conclude that the program was right on this input or (b) the program is wrong on *some* input and hence is not a correct program for graph isomorphism.

§9 Define a language L to be *downward self reducible* there's a polynomial-time algorithm R that for any n and $x \in \{0,1\}^n$, $R^{L_{n-1}}(x) = L(x)$ where by L_k we denote an oracle that solves L on inputs of size at most k. Prove that if L is downward self reducible than $L \in$ **PSPACE**.

§10 Show that **MIP** $\subseteq$ **NEXP**.

§11 Show that if we redefine multiprover interactive proofs to allow, instead of two provers, as many as $m(n) = \text{poly}(n)$ provers on inputs of size n, then the class **MIP** is unchanged.

Hint: Show how to simulate poly(n) provers using two. In this simulation, one of the provers plays the role of all $m(n)$ provers, and the other prover is asked to simulate one of the provers, chosen randomly from among the $m(n)$ provers. Then repeat this a few times.

8.A   Interactive proof for the Permanent

The *permanent* is defined as follows:

DEFINITION 8.29
Let $A \in F^{n \times n}$ be a matrix over the field F. The permanent of A is:

$$\mathsf{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$$

The problem of calculating the permanent is clearly in **PSPACE**. In Chapter 9 we will see that if the permanent can be computed in polynomial time then **P** = **NP**, and hence this problem likely does not have a polynomial-time algorithm.

Although the existence of an interactive proof for the Permanent follows from that for #SAT and TQBF, we describe a specialized protocol as well. This is both for historical context (this protocol was discovered before the other two protocols) and also because this protocol may be helpful for further research. (One example will appear in a later chapter.)

We use the following observation:

$$f(x_1, x_2, ..., x_n) := \mathsf{perm} \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & \ddots & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,n} \end{bmatrix}$$

is a degree n polynomial since

$$f(x_1, x_2, \ldots, x_n) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} x_{i,\sigma(i)}.$$

We now show two properties of the permanent problem. The first is *random self reducibility*, earlier encountered in Section **??**:

THEOREM 8.30 (LIPTON '88)
There is a randomized algorithm that, given an oracle that can compute the permanent on $1 - \frac{1}{3n}$ fraction of the inputs in $F^{n \times n}$ (where the finite field F has size $> 3n$), can compute the permanent on all inputs correctly with high probability.

PROOF: Let A be some input matrix. Pick a random matrix $R \in_R F^{n \times n}$ and let $B(x) := A + x \cdot R$ for a variable x. Notice that:

- $f(x) := \mathsf{perm}(B)$ is a degree n univariate polynomial.

- For any fixed $b \neq 0$, $B(b)$ is a random matrix, hence the probability that oracle computes $\mathsf{perm}(B(b))$ correctly is at least $1 - \frac{1}{3n}$.

Now the algorithm for computing the permanent of A is straightforward: query oracle on all matrices $\{B(i) | 1 \leq i \leq n + 1\}$. According to the union bound, with probability of at least $1 - \frac{n+1}{n} \approx \frac{2}{3}$ the oracle will compute the permanent correctly on all matrices.
Recall the fact (see Section **??** in Appendix A) that given $n + 1$ (point, value) pairs $\{(a_i, b_i) | i \in [n+1]\}$, there exists a unique a degree n polynomial p that satisfies $\forall i \; p(a_i) = b_i$. Therefore, given that the values $B(i)$ are correct, the algorithm can interpolate the polynomial $B(x)$ and compute $B(0) = A$. ∎

Note: The above theorem can be strengthened to be based on the assumption that the oracle can compute the permanent on a fraction of $\frac{1}{2} + \varepsilon$ for any constant $\varepsilon > 0$ of the inputs. The observation is that not all values of the polynomial must be correct for unique interpolation. See Chapter **??**

Another property of the permanent problem is downward self reducibility, encountered earlier in context of SAT:

$$\mathsf{perm}(A) = \sum_{i=1}^{n} a_{1i} \mathsf{perm}(A_{1,i}),$$

where $A_{1,i}$ is a $(n-1) \times (n-1)$ sub-matrix of A obtained by removing the 1'st row and i'th column of A (recall the analogous formula for the determinant uses alternating signs).

DEFINITION 8.31
Define a $(n-1) \times (n-1)$ matrix $D_A(x)$, such that each entry contains a degree n polynomial. This polynomial is uniquely defined by the values of the matrices $\{A_{1,i} | i \in [n]\}$. That is:

$$\forall i \in [n] \ . \ D_A(i) = A_{1,i}$$

Where $D_A(i)$ is the matrix $D_A(x)$ with i substituted for x. (notice that these equalities force n points and values on them for each polynomial at a certain entry of $D_A(x)$, and hence according to the previously mentioned fact determine this polynomial uniquely)

Observation: $\mathsf{perm}(D_A(x))$ is a degree $n(n-1)$ polynomial in x.

8.A.1 The protocol

We now show an interactive proof for the permanent (the decision problem is whether $\mathsf{perm}(A) = k$ for some value k):

- Round 1: Prover sends to verifier a polynomial $g(x)$ of degree $n(n-1)$, which is supposedly $\mathsf{perm}(D_A(x))$.

- Round 2: Verifier checks whether:
$$k = \sum_{i=1}^{m} a_{1,i} g(i)$$

  If not, rejects at once. Otherwise, verifier picks a random element of the field $b_1 \in_R F$ and asks the prover to prove that $g(b_1) = perm(D_A(b_1))$. This reduces the matrix dimension to $(n-2) \times (n-2)$.

$$\vdots$$

- Round $2(n-1) - 1$: Prover sends to verifier a polynomial of degree 2, which is supposedly the permanent of a 2×2 matrix.

- Round $2(n-1)$: Verifier is left with a 2×2 matrix and calculates the permanent of this matrix and decides appropriately.

CLAIM 8.32
The above protocol is indeed an interactive proof for perm.

PROOF: If $\mathsf{perm}(A) = k$, then there exists a prover that makes the verifier accept with probability 1, this prover just returns the correct values of the polynomials according to definition.
On the other hand, suppose that $\mathsf{perm}(A) \neq k$. If on the first round, the polynomial $g(x)$ sent is the correct polynomial $D_A(x)$, then:

$$k \neq \sum_{i=1}^{m} a_{1,i} g(i) = \mathsf{perm}(A)$$

And the verifier would reject. Hence $g(x) \neq D_A(x)$. According to the fact on polynomials stated above, these polynomials can agree on at most $n(n-1)$ points. Hence, the probability that they would agree on the randomly chosen point b_1 is at most $\frac{n(n-1)}{|F|}$. The same considerations apply to all subsequent rounds if exist, and the overall probability that the verifier will not accepts is thus (assuming $|F| \geq 10n^3$ and sufficiently large n):

$$
\begin{aligned}
Pr \quad &\geq \quad \left(1 - \frac{n(n-1)}{|F|}\right) \cdot \left(1 - \frac{(n-1)(n-2)}{|F|}\right) \cdot \ldots \cdot \left(1 - \frac{3 \cdot 2}{|F|}\right) \\
&\geq \quad \left(1 - \frac{n(n-1)}{|F|}\right)^{n-1} \geq \frac{1}{2}
\end{aligned}
$$

∎

Chapter 9

Complexity of counting

> *"It is an empirical fact that for many combinatorial problems the detection of the existence of a solution is easy, yet no computationally efficient method is known for counting their number.... for a variety of problems this phenomenon can be explained."*
> L. Valiant 1979

The class **NP** captures the difficulty of finding *certificates*. However, in many contexts, one is interested not just in a single certificate, but actually counting the *number* of certificates. This chapter studies #**P**, (pronounced "sharp p"), a complexity class that captures this notion.

Counting problems arise in diverse fields, often in situations having to do with estimations of probability. Examples include statistical estimation, statistical physics, network design, and more. Counting problems are also studied in a field of mathematics called *enumerative combinatorics*, which tries to obtain closed-form mathematical expressions for counting problems. To give an example, in the 19th century Kirchoff showed how to count the number of *spanning trees* in a graph using a simple determinant computation. Results in this chapter will show that for many natural counting problems, such efficiently computable expressions are unlikely to exist.

Here is an example that suggests how counting problems can arise in estimations of probability.

EXAMPLE 9.1

In the GraphReliability problem we are given a directed graph on n nodes. Suppose we are told that each node can fail with probability $1/2$ and want to compute the probability that node 1 has a path to n.

A moment's thought shows that under this simple edge failure model, the remaining graph is uniformly chosen at random from all subgraphs of the original graph. Thus the correct answer is

$$\frac{1}{2^n}(\text{number of subgraphs in which node 1 has a path to } n.)$$

We can view this as a *counting* version of the PATH problem.

In the rest of the chapter, we study the complexity class #**P**, a class containing the GraphReliability problem and many other interesting counting problems. We will show that it has a natural and important complete problem, namely the problem of computing the *permanent* of a given matrix. We also show a surprising connection between **PH** and #**P**, called *Toda's Theorem*. Along the way we encounter related complexity classes such as **PP** and ⊕**P**.

9.1   The class #**P**

We now define the class #**P**. Note that it contains functions whose output is a natural number, and not just 0/1.

DEFINITION 9.2 (#**P**)
A function $f : \{0,1\}^* \to \mathbb{N}$ is in #**P** if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time TM M such that for every $x \in \{0,1\}^*$:

$$f(x) = \left| \left\{ y \in \{0,1\}^{p(|x|)} : M(x,y) = 1 \right\} \right| .$$

REMARK 9.3
As in the case of **NP**, we can also define #**P** using non-deterministic TMs. That is, #**P** consists of all functions f such that $f(x)$ is equal to the number of paths from the initial configuration to an accepting configuration in the configuration graph $G_{M,x}$ of a polynomial-time NDTM M.

The big open question regarding #**P**, is whether all problems in this class are efficiently solvable. In other words, whether #**P** = **FP**. (Recall that **FP** is the analog of the class **P** for functions with more than one bit of output, that is, **FP** is the set of functions from $\{0,1\}^*$ to $\{0,1\}^*$ computable by a deterministic polynomial-time Turing machine. Thinking of the output as the binary representation of an integer we can identify such functions with functions from $\{0,1\}^*$ to $\mathbb{N}$. Since computing the number of certificates is at least as hard as finding out whether a certificate exists, if #**P** = **FP** then **NP** = **P**. We do not know whether the other direction also holds: whether **NP** = **P** implies that #**P** = **FP**. We do know that if **PSPACE** = **P** then #**P** = **FP**, since counting the number of certificates can be done in polynomial space.

Here are two more examples for problems in #**P**:

- #SAT is the problem of computing, given a Boolean formula ϕ, the number of satisfying assignments for ϕ.

- #CYCLE is the problem of computing, given a directed graph G, the number of simple cycles in G. (A simple cycle is one that does not visit any vertex twice.)

Clearly, if #SAT ∈ **FP** then SAT ∈ **P** and so **P** = **NP**. Thus presumably #SAT ∉ **FP**. How about #CYCLE? The corresponding decision problem —given a directed graph decide if it has a

cycle—can be solved in linear time by breadth-first-search. The next theorem suggests that the counting problem may be much harder.
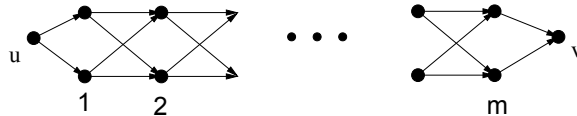
Figure 9.1: Reducing Ham to #CYCLE: by replacing every edge in G with the above gadget to obtain G', every simple cycle of length ℓ in G becomes $(2^m)^\ell$ simple cycles in G'.

THEOREM 9.4
If #CYCLE $\in$ **FP**, *then* **P** $=$ **NP**.

PROOF: We show that if #CYCLE can be computed in polynomial time, then Ham $\in$ **P**, where Ham is the **NP**-complete problem of deciding whether or not a given digraph has a Hamiltonian cycle (i.e., a simple cycle that visits all the vertices in the graph). Given a graph G with n vertices, we construct a graph G' such that G has a Hamiltonian cycle iff G' has at least n^{n^2} cycles.

To obtain G', replace each edge (u, v) in G by the gadget shown in Figure 9.1. The gadget has $m = n \log n + 1$ levels. It is an acyclic digraph, so cycles in G' correspond to cycles in G. Furthermore, there are 2^m directed paths from u to v in the gadget, so a simple cycle of length ℓ in G yields $(2^m)^\ell$ simple cycles in G'.

Notice, if G has a Hamiltonian cycle, then G' has at least $(2^m)^n > n^{n^2}$ cycles. If G has no Hamiltonian cycle, then the longest cycle in G has length at most $n - 1$. The number of cycles is bounded above by n^{n-1}. So G' can have at most $(2^m)^{n-1} \times n^{n-1} < n^{n^2}$ cycles. ∎

9.1.1 The class PP: decision-problem analog for #P.

Similar to the case of search problems, even when studying counting complexity, we can often restrict our attention to *decision problems*. The reason is that there exists a class of decision problems **PP** such that

$$\mathbf{PP} = \mathbf{P} \Leftrightarrow \#\mathbf{P} = \mathbf{FP} \tag{1}$$

Intuitively, **PP** corresponds to computing the most significant bit of functions in #**P**. That is, L is in **PP** if there exists a polynomial-time TM M and a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$,

$$x \in L \Leftrightarrow \left| \left\{ y \in \{0,1\}^{p(|x|)} : M(x, y) = 1 \right\} \right| \geq \frac{1}{2} \cdot 2^{p(|x|)}$$

You are asked to prove the non-trivial direction of (1) in Exercise 1. It is instructive to compare the class **PP**, which we believe contains problem requiring exponential time to solve, with the class **BPP**, which although it has a seemingly similar definition, can in fact be solved efficiently using probabilistic algorithms (and perhaps even also using deterministic algorithms, see Chapter 16). Note that we do not know whether this holds also for the class of decision problems corresponding to the *least* significant bit of #**P**, namely ⊕**P** (see Definition 9.13 below).

9.2   #P completeness.

Now we define #**P**-completeness. Loosely speaking, a function f is #**P**-complete if it is in #**P** and a polynomial-time algorithm for f implies that #**P** = **FP**. To formally define #**P**-completeness, we use the notion of *oracle* TMs, as defined in Section 3.5. Recall that a TM M has *oracle access* to a language $O \subseteq \{0,1\}^*$ if it can make queries of the form "Is $q \in O$?" in one computational step. We generalize this to non-Boolean functions by saying that M has oracle access to a function $f : \{0,1\}^* \rightarrow \{0,1\}^*$, if it is given access to the language $O = \{\langle x, i \rangle : f(x)_i = 1\}$. We use the same notation for functions mapping $\{0,1\}^*$ to $\mathbb{N}$, identifying numbers with their binary representation as strings. For a function $f : \{0,1\}^* \rightarrow \{0,1\}^*$, we define $\mathbf{FP}^f$ to be the set of functions that are computable by polynomial-time TMs that have access to an oracle for f.

DEFINITION 9.5
A function f is #**P**-*complete* if it is in #**P** and every $g \in$ #**P** is in $\mathbf{FP}^f$

If $f \in \mathbf{FP}$ then $\mathbf{FP}^f = \mathbf{FP}$. Thus the following is immediate.

PROPOSITION 9.6
*If f is #**P**-complete and $f \in \mathbf{FP}$ then $\mathbf{FP} =$ #**P**.*

Counting versions of many **NP**-complete languages such as 3SAT, Ham, and CLIQUE naturally lead to #**P**-complete problems. We demonstrate this with #SAT:

THEOREM 9.7
#SAT *is* #**P**-*complete*

PROOF: Consider the Cook-Levin reduction from any L in **NP** to SAT we saw in Section 2.3. This is a polynomial-time computable function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ such that for every $x \in \{0,1\}^*$, $x \in L \Leftrightarrow f(x) \in$ SAT. However, the proof that the reduction works actually gave us more information than that. It provided a *Levin reduction*, by which we mean the proof showed a way to transform a *certificate* that x is in L into a certificate (i.e., satisfying assignment) showing that $f(x) \in$ SAT, and also vice versa (transforming a satisfying assignment for $f(x)$ into a witness that $x \in L$).

In particular, it means that the mapping from the certificates of x to the assignments of $f(x)$ was invertible and hence one-to-one. Thus the number of satisfying assignments for $f(x)$ is equal to the number of certificates for x. ∎

As shown below, there are #**P**-complete problems for which the corresponding decision problems are in fact in **P**.

9.2.1   Permanent and Valiant's Theorem

Now we study another problem. The *permanent* of an $n \times n$ matrix A is defined as

$$\mathsf{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} A_{i,\sigma(i)} \tag{2}$$

where S_n denotes the set of all permutations of n elements. Recall that the expression for the determinant is similar

$$\det(A) = \sum_{\sigma \in S_n} (-1)^{sgn(\sigma)} \prod_{i=1}^{n} A_{i\sigma(i)}$$

except for an additional "sign" term.[1] This similarity does not translate into computational equivalence: the determinant can be computed in polynomial time, whereas computing the permanent seems much harder, as we see below.

The permanent function can also be interpreted combinatorially. First, suppose the matrix A has each entry in $\{0, 1\}$. It may be viewed as the adjacency matrix of a bipartite graph $G(X, Y, E)$, with $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_n\}$ and $\{x_i, y_j\} \in E$ iff $A_{i,j} = 1$. Then the term $\prod_{i=1}^{n} A_{i\sigma(i)}$ is 1 iff σ is a *perfect matching* (which is a set of n edges such that every node is in exactly one edge). Thus if A is a 0.1 matrix then $\mathsf{perm}(A)$ is simply the number of perfect matchings in the corresponding graph G and in particular computing $\mathsf{perm}(A)$ is in #**P**. If A is a $\{-1, 0, 1\}$ matrix, then $\mathsf{perm}(A) = \left| \left\{ \sigma : \prod_{i=1}^{n} A_{i\sigma(i)} = 1 \right\} \right| - \left| \left\{ \sigma : \prod_{i=1}^{n} A_{i\sigma(i)} = -1 \right\} \right|$, so one can make two calls to a #SAT oracle to compute $\mathsf{perm}(A)$. In fact one can show for general integer matrices that computing the permanent is in $\mathbf{FP}^{\#\mathsf{SAT}}$ (see Exercise 2).

The next theorem came as a surprise to researchers in the 1970s, since it implies that if $\mathsf{perm} \in \mathbf{FP}$ then $\mathbf{P} = \mathbf{NP}$. Thus, unless $\mathbf{P} = \mathbf{NP}$, computing the permanent is much more difficult then computing the determinant.

THEOREM 9.8 (VALIANT'S THEOREM)
perm *for* $0, 1$ *matrices is* #**P**-*complete.*

Before proving Theorem 9.8, we introduce yet another way to look at the permanent. Consider matrix A as the the adjacency matrix of a weighted n-node digraph (with possible self loops). Then the expression $\prod_{i=1}^{n} A_{i,\sigma(i)}$ is nonzero iff σ is a cycle-cover of A (a *cycle cover* is a subgraph in which each node has in-degree and out-degree 1; such a subgraph must be composed of cycles). We define the *weight* of the cycle cover to be the product of the weights of the edges in it. Thus $\mathsf{perm}(A)$ is equal to the sum of weights of all possible cycle covers.

EXAMPLE 9.9
Consider the graph in Figure 9.2. Even without knowing what the subgraph G' is, we show that the permanent of the whole graph is 0. For each cycle cover in G' of weight w there are exactly two cycle covers for the three nodes, one with weight $+w$ and one with weight $-w$. Any non-zero weight cycle cover of the whole graph is composed of a cycle cover for G' and one of these two cycle covers. Thus the sum of the weights of all cycle covers of G is 0.

---

[1]It is known that every permutation $\sigma \in S_n$ can be represented as a composition of transpositions, where a transposition is a permutation that only switches between two elements in $[n]$ and leaves the other elements intact (one proof for this statement is the Bubblesort algorithm). If $\tau_1, \ldots, \tau_m$ is a sequence of transpositions such that their composition equals σ, then the *sign* of σ is equal to $+1$ if m is even and -1 if m is odd. It can be shown that the sign is well-defined in the sense that it does not depend on the representation of σ as a composition of transpositions.
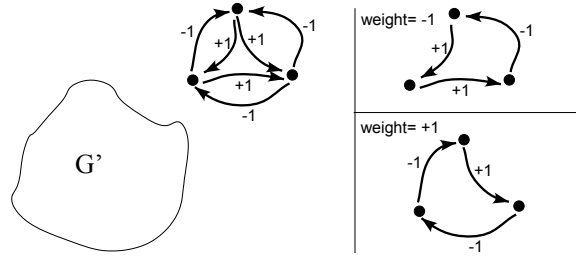
Figure 9.2: The above graph G has cycle cover weight zero regardless of the choice of G', since for every cycle cover of weight w in G', there exist two covers of weight $+w$ and $-w$ in the graph G. (Unmarked edges have $+1$ weight; we follow this convention through out this chapter.)

PROOF OF VALIANT'S THEOREM (THEOREM 9.8): We reduce the #**P**-complete problem #3SAT to perm. Given a boolean formula ϕ with n variables and m clauses, first we shall show how to construct an integer matrix A' with negative entries such that $\mathsf{perm}(A') = 4^m \cdot (\#\phi)$. ($\#\phi$ stands for the number of satisfying assignments of ϕ). Later we shall show how to to get a 0-1 matrix A from A' such that knowing $\mathsf{perm}(A)$ allows us to compute $\mathsf{perm}(A')$.

The main idea is that our construction will result in two kinds of cycle covers in the digraph G' associated with A': those that correspond to satisfying assignments (we will make this precise) and those that don't. We will use negative weights to ensure that the contribution of the cycle covers that do not correspond to satisfying assignments cancels out. (This is similar reasoning to the one used in Example 9.9.) On the other hand, we will show that each satisfying assignment contributes 4^m to $\mathsf{perm}(A')$, and so $\mathsf{perm}(A') = 4^m \cdot (\#\phi)$.

To construct G' from ϕ, we combine the following three kinds of gadgets shown in Figure 9.3:

Variable gadget The variable gadget has two possible cycle covers, corresponding to an assignment of 0 or 1 to that variable. Assigning 1 corresponds to a single cycle taking all the external edges ("true-edges"), and assigning 0 correspond to taking all the self-loops and taking the "false-edge". Each external edge of a variable is associated with a clause in which the variable appears.

Clause gadget The clause gadget is such that the only possible cycle covers exclude at least one external edge. Also for a given (proper) subset of external edges used there is a unique cycle cover (of weight 1). Each external edge is associated with a variable appearing in the clause.

XOR gadget We also use a graph called the XOR gadget whose purpose is to ensure that for some pair of edges $\overrightarrow{u\,u'}$ and $\overrightarrow{v\,v'}$, *exactly one* of these edges is present in any cycle cover that counts towards the final sum.

Suppose that we replace a pair of edges $\overrightarrow{u\,u'}$ and $\overrightarrow{v\,v'}$ in some graph G with the XOR gadget as described in Figure count:fig:valiantgad to obtain some graph G'. Then, via similar reasoning to Example 9.9, every cycle cover of G of weight w that uses exactly one of the edges $\overrightarrow{u\,u'}$ and

Gadget: | Symbolic description:

variable gadget:

clause gadget:

XOR gadget:

The overall construction:
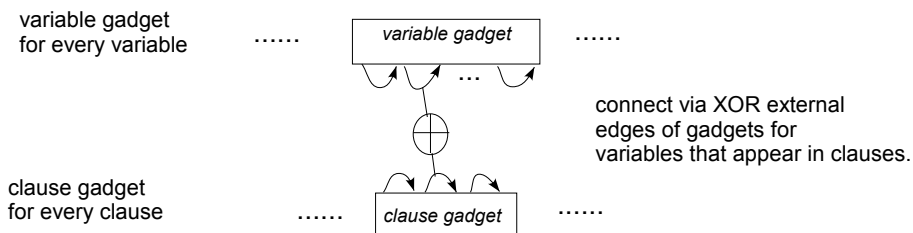
Figure 9.3: The gadgets used in the proof of Valiant's Theorem.

$\overrightarrow{v\,v'}$ is mapped to a set of cycle covers in G' whose total weight is $4w$ (i.e., the set of covers that enter the gadget at u and exit at u' or enter it at v and exit it at v'), while all the other cycle covers of G' have total weight 0 (Exercise 3). For this reason, whenever we replace edges $\overrightarrow{u\,u'}$ and $\overrightarrow{v\,v'}$ with a XOR gadget, we can consider in the analysis only cycle covers that use exactly one of these edges, as the other covers do not contribute anything to the total sum.

The XOR gadgets are used to connect the variable gadgets to the corresponding clause gadgets so that only cycle covers corresponding to a satisfying assignment will be counted towards the total number of cycle covers. Consider a clause, and a variable appearing in it. Each has an external edge corresponding to the other, connected by an XOR gadget. If the external edge in the clause is not taken then by the analysis of the XOR gadget the external edge in the variable must be taken (and hence the variable is true). Since at least one external edge of each clause gadget has to be omitted, each cycle cover that is counted towards the sum corresponds to a satisfying assignment. Conversely, for each satisfying assignment, there is a a set of cycle covers with total weight 4^{3m} (since they passes through the XOR gadget exactly $3m$ times). So $\mathsf{perm}(G') = 4^{3m}\#\phi$.

Reducing to the case $0, 1$ **matrices.** Finally we have to reduce finding $\mathsf{perm}(G')$ to finding $\mathsf{perm}(G)$, where G is an unweighted graph (or equivalently, its adjacency matrix has only $0, 1$ entries). We start by reducing to the case that all edges have weights in $\{\pm 1\}$. First, note that replacing an edge of weight k by k parallel edges of weight 1 does not change the permanent. Parallel edges are not allowed, but we can make edges non-parallel by cutting each edge $\overrightarrow{u\,v}$ in two and inserting a new node w with an edge from u to w, w to v and a self loop at w. To get rid of the negative weights, note that the permanent of an n vertex graph with edge weights in $\{\pm 1\}$ is a number x in $[-n!, +n!]$ and hence this permanent can be computed from $y = x \pmod{2^{m+1}}$ where m is sufficiently large (e.g., $m = n^2$ will do). But to compute y it is enough to compute the permanent of the graph where all weight -1 edges are replaced with edges of weight 2^m. Such edges can be converted to m edges of weight 2 in series, which again can be transformed to parallel edges of weight $+1$ as above. ∎

9.2.2   Approximate solutions to #P problems

Since computing exact solutions to #**P**-complete problems is presumably difficult, a natural question is whether we can *approximate* the number of certificates in the sense of the following definition.

DEFINITION 9.10
Let $f : \{0,1\}^* \to \mathbb{N}$ and $\alpha < 1$. An algorithm A is an α-*approximation* for f if for every x, $\alpha f(x) \le A(x) \le f(x)/\alpha$.

Not all #**P** problems behave identically with respect to this notion. Approximating certain problems within any constant factor $\alpha > 0$ is **NP**-hard (see Exercise 5). For other problems such as 0/1 permanent, there is a *Fully polynomial randomized approximation scheme* (FPRAS), which is an algorithm which, for any ϵ, δ, approximates the function within a factor $1 + \epsilon$ (its answer may be incorrect with probability δ) in time $\mathrm{poly}(n, \log 1/\delta, \log 1/\epsilon)$. Such approximation of counting problems is sufficient for many applications, in particular those where counting is needed to obtain

estimates for the probabilities of certain events (e.g., see our discussion of the graph reliability problem).

The approximation algorithm for the permanent —as well as other similar algorithms for a host of #**P**-complete problems—use the *Monte Carlo Markov Chain* technique. The result that spurred this development is due to Valiant and Vazirani and it shows that under fairly general conditions, approximately counting the number of elements in a set (membership in which is testable in polynomial time) is equivalent —in the sense that the problems are interreducible via polynomial-time randomized reductions— to the problem of generating a *random sample* from the set. We will not discuss this interesting area any further.

Interestingly, if $\mathbf{P} = \mathbf{NP}$ then *every* #**P** problem has an FPRAS (and in fact an FPTAS: i.e., a *deterministic* polynomial-time approximation scheme), see Exercise 6.

9.3 Toda's Theorem: $\mathbf{PH} \subseteq \mathbf{P}^{\#\mathsf{SAT}}$

An important question in the 1980s was the relative power of the polynomial-hierarchy **PH** and the class of counting problems #**P**. Both are natural generalizations of **NP**, but it seemed that their features— alternation and the ability to count certificates, respectively — are not directly comparable to each other. Thus it came as big surprise when in 1989 Toda showed:

THEOREM 9.11 (TODA'S THEOREM [TOD91])
$\mathbf{PH} \subseteq \mathbf{P}^{\#\mathsf{SAT}}$.

That is, we can solve any problem in the polynomial hierarchy given an oracle to a #**P**-complete problem.

REMARK 9.12
Note that we already know, even without Toda's theorem, that if #**P** = **FP** then **NP** = **P** and so **PH** = **P**. However, this does not imply that any problem in **PH** can be computed in polynomial-time using an oracle to #SAT. For example, one implication of Toda's theorem is that a *subexponential* (i.e., $2^{n^{o(1)}}$-time) algorithm for #SAT will imply such an algorithm for any problem in **PH**. Such an implication is not known to hold from a $2^{n^{o(1)}}$-time algorithm for SAT.

9.3.1 The class $\oplus\mathbf{P}$ and hardness of satisfiability with unique solutions.

The following complexity class will be used in the proof:

DEFINITION 9.13
A language L in the class $\oplus\mathbf{P}$ (pronounced "parity P") iff there exists a polynomial time NTM M such that $x \in L$ iff the number of accepting paths of M on input x is odd.

Thus, $\oplus\mathbf{P}$ can be considered as the class of decision problems corresponding to the least significant bit of a #**P**-problem. As in the proof of Theorem 9.7, the fact that the standard **NP**-completeness reduction is parsimonious implies the following problem $\oplus\mathsf{SAT}$ is $\oplus\mathbf{P}$-complete (under many-to-one Karp reductions):

DEFINITION 9.14
Define the quantifier $\bigoplus$ as follows: for every Boolean formula φ on n variables. $\bigoplus_{x \in \{0,1\}^n} \varphi(x)$ is true if the number of x's such that $\varphi(x)$ is true is odd.[2] The language $\oplus\mathsf{SAT}$ consists of all the true quantified Boolean formula of the form $\bigoplus_{x \in \{0,1\}^n} \varphi(x)$ where φ is an unquantified Boolean formula (not necessarily in CNF form).

Unlike the class $\#\mathbf{P}$, it is not known that a polynomial-time algorithm for $\oplus\mathbf{P}$ implies that $\mathbf{NP} = \mathbf{P}$. However, such an algorithm does imply that $\mathbf{NP} = \mathbf{RP}$ since $\mathbf{NP}$ can be probabilistically reduced to $\oplus\mathsf{SAT}$:

THEOREM 9.15 (VALIANT-VAZIRANI THEOREM)
There exists a probabilistic polynomial-time algorithm A such that for every n-variable Boolean formula φ

$$\varphi \in \mathsf{SAT} \Rightarrow \Pr[A(\varphi) \in \oplus\mathsf{SAT}] \geq \tfrac{1}{8n}$$

$$\varphi \notin \mathsf{SAT} \Rightarrow \Pr[A(\varphi) \in \oplus\mathsf{SAT}] = 0$$

To prove Theorem 9.15 we use the following lemma on pairwise independent hash functions:

LEMMA 9.16 (VALIANT-VAZIRANI LEMMA [?])
Let $\mathcal{H}_{n,k}$ be a pairwise independent hash function collection from $\{0,1\}^n$ to $\{0,1\}^k$ and $S \subseteq \{0,1\}^n$ such that $2^{k-2} \leq |S| \leq 2^{k-1}$. Then,

$$\Pr_{h \in_R \mathcal{H}_{n,k}}[\left|\left\{x \in S : h(x) = 0^k\right\}\right| = 1] \geq \tfrac{1}{8}$$

PROOF: For every $x \in S$, let $p = 2^{-k}$ be the probability that $h(x) = 0^k$ when $h \in_R \mathcal{H}_{n,k}$. Note that for every $x \neq x'$, $\Pr[h(x) = 0^k \wedge h(x') = 0^k] = p^2$. Let N be the random variable denoting the number of $x \in S$ satisfying $h(x) = 0^k$. Note that $\mathsf{E}[N] = |S|p \in [\tfrac{1}{4}, \tfrac{1}{2}]$. By the inclusion-exclusion principle

$$\Pr[N \geq 1] \geq \sum_{x \in S} \Pr[h(x) = 0^k] - \sum_{x < x' \in S} \Pr[h(x) = 0^k \wedge h(x') = 0^k] = |S|p - \binom{|S|}{2}p^2$$

and by the union bound we get that $\Pr[N \geq 2] \leq \binom{|S|}{2}p^2$. Thus

$$\Pr[N = 1] = \Pr[N \geq 1] - \Pr[N \geq 2] \geq |S|p - 2\binom{|S|}{2}p^2 \geq |S|p - |S|^2 p^2 \geq \frac{1}{8}$$

where the last inequality is obtained using the fact that $\tfrac{1}{4} \leq |S|p \leq \tfrac{1}{2}$. $\blacksquare$

---

[2]Note that if we identify true with 1 and 0 with false then $\bigoplus_{x \in \{0,1\}^n} \varphi(x) = \sum_{x \in \{0,1\}^n} \varphi(x) \pmod 2$. Also note that $\bigoplus_{x \in \{0,1\}^n} \varphi(x) = \bigoplus_{x_1 \in \{0,1\}} \cdots \bigoplus_{x_n \in \{0,1\}} \varphi(x_1, \ldots, x_n)$.

Proof of Theorem 9.15

We now use Lemma 9.16 to prove Theorem 9.15. Given a formula φ on n variables, our probabilistic algorithm A chooses k at random from $\{2, \ldots, n+1\}$ and a random hash function $h \in_R \mathcal{H}_{n,k}$. It then uses the Cook-Levin reduction to compute a formula τ on variables $x \in \{0,1\}^n$, $y \in \{0,1\}^m$ (for $m = \mathrm{poly}(n)$) such that $h(x) = 0$ if and only if there exists a *unique* y such that $\tau(x,y) = 1$.[3] The output of A if the formula

$$\psi = \bigoplus_{x \in \{0,1\}^n, y \in \{0,1\}^m} \varphi(x) \wedge \tau(x,y) \,,$$

It is equivalent to the statement

$$\bigoplus_{x \in \{0,1\}^n} \varphi(x) \wedge h(x) = 0^k \,,$$

If φ is unsatisfiable then ψ is false, since we'll have no x's satisfying the inner formula and zero is an even number. If φ is satisfiable, we let S be the set of its satisfying assignments. With probability $1/n$, k satisfies $2^{k-2} \leq |S| \leq 2^k$, conditioned on which, with probability $1/8$, there is a unique x such that $\varphi(x) \wedge h(x) = 0^n$. Since one happens to be an odd number, this implies that ψ is true. ∎

REMARK 9.17 (HARDNESS OF UNIQUE SATISFIABILITY)
The proof of Theorem 9.15 implies the following stronger statement: the existence of an algorithm to distinguish between an unsatisfiable Boolean formula and a formula with exactly one satisfying assignment implies the existence of a probabilistic polynomial-time algorithm for all of **NP**. Thus, the guarantee that a particular search problem has either no solutions or a unique solution does not necessarily make the problem easier to solve.

9.3.2 Step 1: Randomized reduction from PH to ⊕P

We now go beyond **NP** (that is to say, the Valiant-Vazirani theorem) and show that we can actually reduce any language in the polynomial hierarchy to ⊕SAT.

LEMMA 9.18
Let $c \in \mathbb{N}$ be some constant. There exists a probabilistic polynomial-time algorithm A such that for every ψ a Quantified Boolean formula with c levels of alternations,

$$\psi \text{ is true} \Rightarrow \Pr[A(\psi) \in \oplus\mathsf{SAT}] \geq \tfrac{2}{3}$$
$$\psi \text{ is false} \Rightarrow \Pr[A(\psi) \in \oplus\mathsf{SAT}] = 0$$

Before proving the Lemma, let us make a few notations and observations: For a Boolean formula φ on n variables, let $\#(\varphi)$ denote the number of satisfying assignments of φ. We consider also formulae φ that are *partially quantified*. That is, in addition to the n variables φ takes as input

---

[3]For some implementations of hash functions, such as the one described in Exercise 4, one can construct directly (without going through the Cook-Levin reduction) such a formula τ that does not use the y variables.

it may also have other variables that are bound by a $\forall, \exists$ or $\bigoplus$ quantifiers (for example φ can be of the form $\varphi(x_1, \ldots, x_n) = \forall y \in \{0,1\}^n \, \tau(x_1, \ldots, x_n, y)$ where τ is, say, a 3CNF Boolean formula).

Given two (possibly partially quantified) formulae φ, ψ on variables $x \in \{0,1\}^n$, $y \in \{0,1\}^m$ we can construct in polynomial-time an $n+m$ variable formula $\varphi \cdot \psi$ and a $(\max\{n,m\}+1)$-variable formula $\varphi + \psi$ such that $\#(\varphi \cdot \psi) = \#(\varphi)\#(\varphi)$ and $\#(\varphi + \psi) = \#(\varphi) + \#(\psi)$. Indeed, take $\varphi \cdot \psi(x,y) = \varphi(x) \wedge \varphi(y)$ and $\varphi + \psi(z) = \big((z_0 = 0) \wedge \varphi(z_1, \ldots, z_n)\big) \vee \big((z_0 = 1) \wedge \psi(z_1, \ldots, z_m)\big)$. For a formula φ, we use the notation $\varphi + 1$ to denote the formula $\varphi + \psi$ where ψ is some canonical formula with a single satisfying assignment. Since the product of numbers is even iff one of the numbers is even, and since adding one to a number flips the parity, for every two formulae φ, ψ as above

$$\left(\bigoplus_x \varphi(x)\right) \wedge \left(\bigoplus_y \psi(y)\right) \Leftrightarrow \bigoplus_{x,y}(\varphi \cdot \psi)(x,y) \tag{3}$$

$$\neg \bigoplus_x \varphi(x) \Leftrightarrow \bigoplus_{x,z}(\varphi + 1)(x,z) \tag{4}$$

$$\left(\bigoplus_x \varphi(x)\right) \vee \left(\bigoplus_y \psi(y)\right) \Leftrightarrow \bigoplus_{x,y,z}((\varphi+1)\cdot(\psi+1)+1)(x,y,z) \tag{5}$$

PROOF OF LEMMA 9.18: Recall that membership in a $\mathbf{PH}$-language can be reduced to deciding the truth of a quantified Boolean formula with a constant number of alternating quantifiers. The idea behind the proof is to replace one-by-one each $\exists/\forall$ quantifiers with a $\bigoplus$ quantifier.

Let ψ be a formula with c levels of alternating $\exists/\forall$ quantifiers, possibly with an initial $\bigoplus$ quantifier. We transform ψ in probabilistic polynomial-time to a formula ψ' such that ψ' has only $c-1$ levels of alternating $\exists/\forall$ quantifiers, an initial $\bigoplus$ quantifier, satisfying (1) if ψ is false then so is ψ', and (2) if ψ is true then with probability at least $1 - \frac{1}{10c}$, ψ' is true as well. The lemma follows by repeating this step c times.

For ease of notation, we demonstrate the proof for the case that ψ has a single $\bigoplus$ quantifier and two additional $\exists/\forall$ quantifiers. We can assume without loss of generality that ψ is of the form

$$\psi = \bigoplus_{z \in \{0,1\}^\ell} \exists_{x \in \{0,1\}^n} \forall_{w \in \{0,1\}^k} \varphi(z,x,w),$$

as otherwise we can use the identities $\forall_x P(x) = \neg \exists_x \neg P(x)$ and (4) to transform ψ into this form.

The proof of Theorem 9.15 provides for every n, a probabilistic algorithm that outputs a formula τ on variables $x \in \{0,1\}^n$ and $y \in \{0,1\}^m$ such that for every nonempty set $S \subseteq \{0,1\}^n$, $\Pr[\oplus_{x \in \{0,1\}^n, y \in \{0,1\}^m} \tau(x,y)] \geq 1/(8n)$. Run this algorithm $t = 100c\ell \log n$ times to obtain the formulae $\tau_1, \ldots, \tau_t$. Then, for every nonempty set $S \subseteq \{0,1\}^n$ the probability that there does not exist $i \in [t]$ such that $\oplus_{x \in \{0,1\}^n, y \in \{0,1\}^m} \tau(x,y)$ is TRUE is less than $2^{-\ell}/(10c)$. We claim that this implies that with probability at least $1 - 1/(10c)$, the following formula is equivalent to ψ:

$$\bigoplus_{z \in \{0,1\}^\ell} \theta(z), \tag{6}$$

where

$$\theta(z) = \vee_{i=1}^t \left(\bigoplus_{x \in \{0,1\}^n, y \in \{0,1\}^m} \forall_{w \in \{0,1\}^k} \tau_i(x,y) \wedge \varphi(x,z,w) \right)$$

Indeed, for every $z \in \{0,1\}^\ell$ define $S_z = \left\{ x \in \{0,1\}^n : \forall_{w \in \{0,1\}^k} \varphi(x,z,w) \right\}$. Then, ψ is equivalent to $\oplus_{z \in \{0,1\}^\ell} |S_z|$ is nonempty. But by the union bound, with probability at least $1 - 1/(10c)$ it holds that for *every* z such that S_z is nonempty, there exists τ_i satisfying $\oplus_{x,y} \tau_i(x,y)$. This means that for every such z, $\theta(z)$ is true. On the other hand, if S_z is empty then certainly $\theta(z)$ is false, implying that indeed ψ is equivalent to (6).

By applying the identity (5), we can transform (6) into an equivalent formula of the desired form

$$\bigoplus_{z,x,y,w} \forall_w \varphi'(x,y,z,w)$$

for some unquantified polynomial-size formula φ'. $\blacksquare$

9.3.3 Step 2: Making the reduction deterministic

To complete the proof of Toda's Theorem (Theorem 9.11), we prove the following lemma:

LEMMA 9.19
There is a (deterministic) polynomial-time transformation T that, for every formula ψ that is an input for $\oplus\mathsf{SAT}$, $T(\psi, 1^m)$ is an unquantified Boolean formula and

$$\psi \in \oplus\mathsf{SAT} \Rightarrow \#(\varphi) = -1 \pmod{2^{m+1}}$$
$$\psi \notin \oplus\mathsf{SAT} \Rightarrow \#(\varphi) = 0 \pmod{2^{m+1}}$$

PROOF OF THEOREM 9.11 USING LEMMAS 9.18 AND 9.19.: Let $L \in PH$. We show that we can decide whether an input $x \in L$ by asking a single question to a #SAT oracle. For every $x \in \{0,1\}^n$, Lemmas 9.18 and 9.19 together imply there exists a polynomial-time TM M such that

$$x \in L \Rightarrow \Pr_{r \in_R \{0,1\}^m}[\#(M(x,r)) = -1 \pmod{2^{m+1}}] \geq \tfrac{2}{3}$$
$$x \notin L \Rightarrow \forall_{r \in_R \{0,1\}^m} \#(M(x,r)) = 0 \pmod{2^{m+1}}$$

where m is the (polynomial in n) number of random bits used by the procedure described in that Lemma. Furthermore, even in the case $x \in L$, we are guaranteed that for every $r \in \{0,1\}^m$, $\#(M(x,r)) \in \{0,-1\} \pmod{2^{m+1}}$.

Consider the function that maps two strings r, u into the evaluation of the formula $M(x,r)$ on the assignment u. Since this function is computable in polynomial-time, the Cook-Levin transformation implies that we can obtain in polynomial-time a CNF formula θ_x on variables r, u, y such that for every r, u, $M(x,r)$ is satisfied by u if and only if there exist a unique y such that $\theta_x(r,u,y)$ is true. Let $f_x(r)$ be the number of u, y such that $\theta_x(r,u,y)$ is true, then

$$\#(\theta_x) = \sum_{r \in \{0,1\}^m} f_x(r),$$

But if $x \notin L$ then $f_x(r) = 0 \pmod{2^{m+1}}$ for every r, and hence $\#(\theta_x) = 0 \pmod{2^{m+1}}$. On the other hand, if $x \in L$ then $f_x(r) = -1 \pmod{2^{m+1}}$ for between $\tfrac{2}{3}2^m$ and 2^m values of r, and is

equal to 0 on the other values, and hence $\#(\theta_x) \neq 0 \pmod{2^{m+1}}$. We see that deciding whether $x \in L$ can be done by computing $\#(\theta_x)$.∎

PROOF OF LEMMA 9.19: For every pair of formulae φ, τ recall that we defined formulas $\varphi + \tau$ and $\varphi \cdot \tau$ satisfying $\#(\varphi + \tau) = \#(\varphi) + \#(\tau)$ and $\#(\varphi \cdot \tau) = \#(\varphi)\#(\tau)$, and note that these formulae are of size at most a constant factor larger than φ, τ. Consider the formula $4\tau^3 + 3\tau^4$ (where τ^3 for example is shorthand for $\tau \cdot (\tau \cdot \tau)$). One can easily check that

$$\#(\tau) = -1 \pmod{2^{2^i}} \Rightarrow \#(4\tau^3 + 3\tau^4) = -1 \pmod{2^{2^{i+1}}} \tag{7}$$

$$\#(\tau) = 0 \pmod{2^{2^i}} \Rightarrow \#(4\tau^3 + 3\tau^4) = 0 \pmod{2}^{2^{i+1}} \tag{8}$$

Let $\psi_0 = \psi$ and $\psi_{i+1} = 4\psi_i^3 + 3\psi_i^4$. Let $\psi^* = \psi_{\lceil \log(m+1) \rceil}$. Repeated use of equations (7), (8) shows that if $\#(\psi)$ is odd, then $\#(\psi^*) = -1 \pmod{2^{m+1}}$ and if $\#(\psi)$ is even, then $\#(\psi^*) = 0 \pmod{2^{m+1}}$. Also, the size of ψ^* is only polynomially larger than size of ψ. ∎

WHAT HAVE WE LEARNED?

- The class $\#\mathbf{P}$ consists of functions that count the number of certificates for a given instance. If $\mathbf{P} \neq \mathbf{NP}$ then it is not solvable in polynomial time.

- Counting analogs of many natural $\mathbf{NP}$-complete problems are $\#\mathbf{P}$-complete, but there are also $\#\mathbf{P}$-complete counting problems for which the corresponding decision problem is in $\mathbf{P}$. One example for this is the problem perm of computing the permanent.

- Surprisingly, counting is more powerful than alternating quantifiers: we can solve every problem in the polynomial hierarchy using an oracle to a $\#\mathbf{P}$-complete problem.

- The classes $\mathbf{PP}$ and $\oplus\mathbf{P}$ contain the decision problems that correspond to the most significant and least significant bits (respectively) of a $\#\mathbf{P}$ function. The class $\mathbf{PP}$ is as powerful as $\#\mathbf{P}$ itself, in the sense that if $\mathbf{PP} = \mathbf{P}$ then $\#\mathbf{P} = \mathbf{FP}$. We do not know if this holds for $\oplus\mathbf{P}$ but do know that every language in $\mathbf{PH}$ randomly reduces to $\oplus\mathbf{P}$.

9.4 Open Problems

- What is the exact power of $\oplus\mathsf{SAT}$ and $\#SAT$?

- What is the average case complexity of $n \times n$ permanent modulo small prime, say 3 or 5 ? Note that for a prime $p > n$, random self reducibility of permanent implies that if permanent is hard to compute on at least one input then it is hard to compute on $1 - O(p/n)$ fraction of inputs, i.e. hard to compute on average (see Theorem **??**).

Chapter notes and history

The definition of $\#\mathbf{P}$ as well as several interesting examples of $\#\mathbf{P}$ problems appeared in Valiant's seminal paper [Val79b]. The $\#\mathbf{P}$-completeness of the permanent is from his other paper [Val79a]. Toda's Theorem is proved in [Tod91]. The proof given here follows the proof of [KVVY93] (although we use formulas where they used circuits.)

For an introduction to FPRAS's for computing approximations to many counting problems, see the relevant chapter in Vazirani [Vaz01] (an excellent resource on approximation algorithms in general).

.

Exercises

§1 Let $f \in \#\mathbf{P}$. Show a polynomial-time algorithm to compute f given access to an oracle for some language $L \in \mathbf{PP}$ (see Remark **??**).

> **Hint:** without loss of generality you can think that $f = \#\mathsf{CKT} - \mathsf{SAT}$, the problem of computing the number of satisfying assignments for a given Boolean circuit C, and that you are given an oracle that tells you if a given n-variable circuit, has at least 2^{n-1} satisfying assignments or not. The main observation you can use is that if C has at least 2^{n-1} satisfying assignments then it is possible to use the oracle to find a string x such that C has exactly 2^{n-1} satisfying assignments that are larger than x in the natural lexicographic ordering of the strings in $\{0,1\}^n$.

§2 Show that computing the permanent for matrices with integer entries is in $\mathbf{FP}^{\#\mathsf{SAT}}$.

§3 Complete the analysis of the XOR gadget in the proof of Theorem 9.8. Let G be any weighted graph containing a pair of edges $\overrightarrow{u\,u'}$ and $\overrightarrow{v\,v'}$, and let G' be the graph obtained by replacing these edges with the XOR gadget. Prove that every cycle cover of G of weight w that uses exactly one of the edges $\overrightarrow{u\,u'}$ is mapped to a set of cycle covers in G' whose total weight is $4w$, and all the other cycle covers of G' have total weight 0.

§4 Let $k \leq n$. Prove that the following family $\mathcal{H}_{n,k}$ is a collection of pairwise independent functions from $\{0,1\}^n$ to $\{0,1\}^k$: Identify $\{0,1\}$ with the field $\mathrm{GF}(2)$. For every $k \times n$ matrix A with entries in $\mathrm{GF}(2)$, and k-length vector $b \in \mathrm{GF}(2)^n$, $\mathcal{H}_{n,k}$ contains the function $h_{A,b} : \mathrm{GF}(2)^n \to \mathrm{GF}(2)^k$ defined as follows: $h_{A,\mathbf{b}}(x) = Ax + b$.

§5 Show that if there is a polynomial-time algorithm that approximates $\#\mathsf{CYCLE}$ within a factor $1/2$, then $\mathbf{P} = \mathbf{NP}$.

§6 Show that if $\mathbf{NP} = \mathbf{P}$ then for every $f \in \#\mathbf{P}$ and there is a polynomial-time algorithm that approximates f within a factor of $1/2$. Can you show the same for a factor of $1 - \epsilon$ for arbitrarily small constant $\epsilon > 0$? Can you make these algorithms *deterministic*?

Note that we do not know whether $\mathbf{P} = \mathbf{NP}$ implies that exact computation of functions in $\#\mathbf{P}$ can be done in polynomial time.

ideas of the proof that $\mathbf{BPP} \subseteq \mathbf{PH}$ (Theorem 7.18).

tocol of Chapter 8. To make the algorithm deterministic use the

back to it after seeing the Goldwasser-Sipser set lowerbound pro-

strings. If you find this question difficult you might want to come

theorem, where we also needed to estimate the size of a set of

Hint: Use hashing and ideas similar to those in the proof of Toda's

§7 Show that every for every language in $\mathbf{AC}^0$ there is a depth 3 circuit of $n^{\text{poly}(\log n)}$ size that decides it on $1 - 1/\text{poly}(n)$ fraction of inputs and looks as follows: it has a single $\oplus$ gate at the top and the other gates are $\vee, \wedge$ of fanin at most $\text{poly}(\log n)$.

Hint: use the proof of Lemma 9.18.

Chapter 10

Cryptography

"From times immemorial, humanity has gotten frequent, often cruel, reminders that many things are easier to do than to reverse."
L. Levin [Lev]

SOMEWHAT ROUGH STILL

The importance of cryptography in today's online world needs no introduction. Here we focus on the complexity issues that underlie this field. The traditional task of cryptography was to allow two parties to *encrypt* their messages so that eavesdroppers gain no information about the message. (See Figure 10.1.) Various encryption techniques have been invented throughout history with one common characteristic: sooner or later they were broken.

Figure unavailable in pdf file.

Figure 10.1: People sending messages over a public channel (e.g., the internet) wish to use encryption so that eavesdroppers learn "nothing."

In the post **NP**-completeness era, a crucial new idea was presented: the code-breaker should be thought of as a resource-bounded computational device. Hence the security of encryption schemes ought to be proved by *reducing* the task of breaking the scheme into the task of solving some computationally intractable problem (say requiring exponential time complexity or circuit size), thus one could hope to design encryption schemes that are efficient enough to be used in practice, but whose breaking will require, say, millions of years of computation time.

Early researchers tried to base the security of encyption methods upon the (presumed) in-tractability of **NP**-complete problems. This effort has not succeeded to date, seemingly because **NP**-completeness concerns the intractability of problems in the *worst-case* whereas cryptography seems to need problems that are intractable on *most instances*. After all, when we encrypt email, we require that decryption should be difficult for an eavesdropper for all (or *almost all*) messages, not just for a few messages. Thus the concept most useful in this chapter will be *average-case complexity*[1]. We will see a class of functions called *one-way functions* that are easy to compute

---

[1]A problem's average-case and worst-case complexities can differ radically. For instance, 3COL is **NP**-complete

but hard to invert for most inputs —they are alluded to in Levin's quote above. Such functions exist under a variety of assumptions, including the famous assumption that factoring integers requires time super-polynomial time in the integer's bit-length to solve in the average case (e.g., for a product of two random primes).

Furthermore, in the past two decades, cryptographers have taken on tasks above and beyond the basic task of encryption—from implementing digital cash to maintaining the privacy of individuals in public databases. (We survey some applications in Section 10.4.) Surprisingly, many of these tasks can be achieved using the same computational assumptions used for encryption. A crucial ingredient in these developments turns out to be an answer to the question: "What is a random string and how can we generate one?" The complexity-theoretic answer to this question leads to the notion of a *pseudorandom generator*, which is a central object; see Section 10.2. This notion is very useful in itself and is also a template for several other key definitions in cryptography, including that of encryption (see Section 10.4).

Private key versus public key: Solutions to the encryption problem today come in two distinct flavors. In *private-key cryptography,* one assumes that the two (or more) parties participating in the protocol share a private "key" —namely, a statistically random string of modest size—that is not known to the eavesdropper[2]. In a *public-key encryption system* (a concept introduced by Diffie and Hellman in 1976 [DH76]) we drop this assumption. Instead, a party P picks a pair of keys: an *encryption* key and *decryption* key, both chosen at random from some (correlated) distribution. The encryption key will be used to encrypt messages to P and is considered public —i.e., published and known to everybody including the eavesdropper. The decryption key is kept secret by P and is used to decrypt messages. A famous public-key encryption scheme is based upon the RSA function of Example 10.4. At the moment we do not know how to base public key encryption on the sole assumption that one-way functions exist and current constructions require the assumption that there exist one-way functions with some special structure (such as RSA, factoring-based, and Lattice-based one way functions). Most topics described in this chapter are traditionally labeled *private key cryptography.*

10.1   Hard-on-average problems and one-way functions

A basic cryptographic primitive is a *one-way function.* Roughly speaking, this is a function f that is easy to compute but hard to invert. Notice that if f is not one-to-one, then the inverse $f^{-1}(x)$ may not be unique. In such cases "inverting" means that given $f(x)$ the algorithm is able to produce some preimage, namely, any element of $f^{-1}(f(x))$. We say that the function is one-way function if inversion is difficult for the "average" (or "many") x. Now we define this formally; a discussion of this definition appears below in Section 10.1.1. A function family (g_n) is a family of functions where g_n takes n-bit inputs. It is *polynomial-time computable* if there is a polynomial-time TM that given an input x computes $g_{|x|}(x)$.

---

on general graphs, but on most n-node graphs is solvable in quadratic time or less. A deeper study of average case complexity appears in Chapter 15.

[2]Practically, this could be ensured with a face-to-face meeting that might occur long before the transmission of messages.

DRAFT

DEFINITION 10.1 (ONE-WAY FUNCTION)
A family of functions $\{f_n : \{0,1\}^n \mapsto \{0,1\}^{m(n)}\}$ is $\epsilon(n)$ *one-way* with *security* $s(n)$ if it is polynomial-time computable and furthermore for every algorithm A that runs in time $s(n)$,

$$\mathbf{Pr}_{x \in \{0,1\}^n}[A \text{ inverts } f_n(x)] \leq \epsilon(n). \tag{1}$$

Now we give a few examples and discuss the evidence that they are hard to invert "on average inputs."

EXAMPLE 10.2
The first example is motivated by the fact that finding the prime factors of a given integer is the famous FACTORING problem, for which the best current algorithm has running time about $2^{O(n^{1/3})}$ (and even that bounds relies on the truth of some unproven conjectures in number theory). The hardest inputs for current algorithms appear to be of the type $x \cdot y$, where x, y are random primes of roughly equal size.

Here is a first attempt to define a one-way function using this observation. Let $\{f_n\}$ be a family of functions where $f_n : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^{2n}$ is defined as $f_n([x]_2, [y]_2) = [x \cdot y]_2$. If x and y are primes —which by the Prime Number Theorem happens with probability $\Theta(1/n^2)$ when x, y are random n-bit integers— then f_n seems hard to invert. It is widely believed that there are $c > 1, f > 0$ such that family f_n is $(1 - 1/n^c)$-one-way with security parameter 2^{n^f}.

An even harder version of the above function is obtained by using the existence of a randomized polynomial-time algorithm A (which we do not describe) that, given 1^n, generates a random n-bit prime number. Suppose A uses m random bits, where $m = \text{poly}(n)$. Then A may be seen as a (deterministic) mapping from m-bit strings to n-bit primes. Now let function $\tilde{f}_m$ map (r_1, r_2) to $[A(r_1) \cdot A(r_2)]_2$, where $A(r_1), A(r_2)$ are the primes output by A using random strings r_1, r_2 respectively. This function seems hard to invert for almost all r_1, r_2. (Note that any inverse (r_1', r_2') for $\tilde{f}_m(r_1, r_2)$ allows us to factor the integer $A(r_1) \cdot A(r_2)$ since unique factorization implies that the prime pair $A(r_1'), A(r_2')$ must be the same as $A(r_1), A(r_2)$.) It is widely conjecture that there are $c > 1, f > 0$ such that $\tilde{f}_n$ is $1/n^c$-one-way with security parameter 2^{n^f}.

The FACTORING problem, a mainstay of modern cryptography, is of course the inverse of multiplication. Who would have thought that the humble multiplication, taught to children in second grade, could be the source of such power? The next two examples also rely on elementary mathematical operations such as exponentiation, albeit with modular arithmetic.

EXAMPLE 10.3
Let $p_1, p_2, \ldots$ be a sequence of primes where p_i has i bits. Let g_i be the generator of the group $Z_{p_i}^*$, the set of numbers that are nonzero mod p_i. Then for every $y \in 1, .., p_i - 1$, there is a unique $x \in \{1, .., p - 1\}$ such that

$$g_i^x \equiv y \pmod{p_i}.$$

Then $x \to g_i^x \pmod{p_i}$ is a permutation on $1, .., p_i - 1$ and is conjectured to be one-way. The inversion problem is called the *DISCRETE LOG* problem. We show below using random self-reducibility that if it is hard on worst-case inputs, then it is hard on average.

We list some more conjectured one-way functions.

EXAMPLE 10.4

RSA function. Let $m = pq$ where p, q are large random primes and e be a random number coprime to $\phi(m) = (p - 1)(q - 1)$. Let Z_m^* be the set of integers in $[1, \ldots, m]$ coprime to m. Then the function is defined to be $f_{p,q,e}(x) = x^e \pmod{m}$. This function is used in the famous RSA public-key cryptosystem.

Rabin function. For a composite number m, define $f_m(x) = x^2 \pmod{m}$. If we can invert this function on a $1/\text{poly}(\log m)$ fraction of inputs then we can factor m in $\text{poly}(\log m)$ time (see exercises).

Both the RSA and Rabin functions are useful in public-key cryptography. They are examples of *trapdoor* one-way functions: if the factors of m (the "trapdoor" information) are given as well then it is easy to invert the above functions. Trapdoor functions are fascinating objects but will not be studied further here.

Random subset sum. Let $m = 10n$. Let the inputs to f be n positive m-bit integers $a_1, a_2, \ldots, a_n$, and a subset S of $\{1, 2, \ldots, n\}$. Its output is $(a_1, a_2, \ldots, a_n, \sum_{i \in S} a_i)$. Note that f maps $n(m+1)$-bit inputs to $nm + m$ bits.

When the inputs are randomly chosen, this function seems hard to invert. It is conjectured that there is $c > 1, d > 0$ such that this function is $1/n^c$-one-way with security 2^{n^d}.

10.1.1   Discussion of the definition of one-way function

We will always assume that the the one-way function under consideration is such that the security parameter $s(n)$ is superpolynomial, i.e., larger than n^k for every $k > 0$. The functions described earlier are actually believed to be one-way with a larger security parameter 2^{n^ϵ} for some fixed $\epsilon > 0$.

Of greater interest is the error parameter $\epsilon(n)$, since it determines the *fraction* of inputs for which inversion is easy. Clearly, a continuum of values is possible, but two important cases to consider are (i) $\epsilon(n) = (1 - 1/n^c)$ for some fixed $c > 0$, in other words, the function is difficult to invert on at least $1/n^c$ fraction of inputs. Such a function is often called a *weak* one-way function. The simple one-way function f_n of Example 10.2 is conjectured to be of this type. (ii) $\epsilon(n) < 1/n^k$ for every $k > 1$. Such a function is called a *strong* one-way function.

Yao showed that if weak one-way functions exist then so do strong one-way functions. We will prove this surprising theorem (actually, something close to it) in Chapter 17. We will not use it in this chapter, except as a justification for our intuition that strong one-way functions exist.

(Another justification is of course the empirical observation that the candidate one-way functions mentioned above do seem appear difficult to invert on most inputs.)

10.1.2 Random self-reducibility

Roughly speaking, a problem is *random-self-reducible* if solving the problem on any input x reduces to solving the problem on a sequence of random inputs $y_1, y_2, \ldots$, where each y_i is uniformly distributed among all inputs. To put it more intuitively, the worst-case can be reduced to the average case. Hence the problem is either easy on *all* inputs, or hard on *most* inputs. (In other words, we can exclude the possibility that problem is easy on almost all the inputs but not all.) If a function is one-way and also randomly self-reducible then it must be a strong one-way function. This is best illustrated with an example.

THEOREM 10.5
Suppose A is an algorithm with running time $t(n)$ that, given a prime p, a generator g for $\mathbb{Z}_p^*$, and an input $g^x(\mod p)$, manages to find x for δ fraction of $x \in \mathbb{Z}_p^*$. Then there is a randomized algorithm A' with running time $O(\frac{1}{\delta \log 1/\epsilon}(t(n) + poly(n)))$ that solves DISCRETE LOG on every input with probability at least $1 - \epsilon$.

PROOF: Suppose we are given $y = g^x(\mod p)$ and we are trying to find x. Repeat the following trial $O(1/(\delta \log 1/\epsilon))$ times: "Randomly pick $r \in \{0, 1, \ldots, p-2\}$ and use A to try to compute the logarithm of $y \cdot g^r (\mod p)$. Suppose A outputs z. Check if $g^{z-r}(\mod p)$ is y, and if so, output $z - r (\mod(p-1))$ as the answer."

The main observation is that if r is randomly chosen, then $y \cdot g^r (\mod p)$ is randomly distributed in $\mathbb{Z}_p^*$ and hence the hypothesis implies that A has a δ chance of finding its discrete log. After $O(1/(\delta \log 1/\epsilon)$ trials, the probability that A failed every time is at most ϵ. ∎

COROLLARY 10.6
If for any infinite sequence of primes $p_1, p_2, \ldots$, DISCRETE LOG mod p_i is hard on worst-case $x \in \mathbb{Z}_{p_i}^*$, then it is hard for almost all x.

Later as part of the proof of Theorem 10.14 we give another example of random self-reducibility: linear functions over $GF(2)$.

10.2 What is a random-enough string?

Cryptography often becomes much easier if we have an abundant supply of random bits. Here is an example.

EXAMPLE 10.7 (ONE-TIME PAD)
Suppose the message sender and receiver share a long string r of random bits that is not available to eavesdroppers. Then secure communication is easy. To encode message $m \in \{0, 1\}^n$, take the first n bits of r, say the string s. Interpret both strings as vectors in $GF(2)^n$ and encrypt m by the vector $m + s$. The receiver decrypts this message by adding s to it (note that $s + s = 0$ in $GF(2)^n$).

If s is statistically random, then so is $m + s$. Hence the eavesdropper provably cannot obtain even a single bit of information about m *regardless of how much computational power he expends.*

Note that reusing s is a strict no-no (hence the name "one-time pad"). If the sender ever reuses s to encrypt another message m' then the eavesdropper can add the two vectors to obtain $(m + s) + (m' + s) = m + m'$, which is some nontrivial information about the two messages.

Of course, the one-time pad is just a modern version of the old idea of using "codebooks" with a new key prescribed for each day.

One-time pads are conceptually simple, but impractical to use, because the users need to agree in advance on a secret pad that is large enough to be used for all their future communications. It is also hard to generate because sources of quality random bits (e.g., those based upon quantum phenomena) are often too slow. Cryptography's suggested solution to such problems is to use a pseudorandom generator. This is a deterministically computable function $g : \{0,1\}^n \to \{0,1\}^{n^c}$ (for some $c > 1$) such that if $x \in \{0,1\}^n$ is randomly chosen, then $g(x)$ "looks" random. Thus so long as users have been provided a common n-bit random string, they can use the generator to produce n^c "random looking" bits, which can be used to encrypt n^{c-1} messages of length n. (In cryptography this is called a *stream cipher.*)

Clearly, at this point we need an answer to the question posed in the Section's title! Philosophers and statisticians have long struggled with this question.

EXAMPLE 10.8

What is a random-enough string? Here is Kolmogorov's definition: *A string of length n is random if no Turing machine whose description length is $< 0.99n$ (say) outputs this string when started on an empty tape.* This definition is the "right" definition in some philosophical and technical sense (which we will not get into here) but is not very useful in the complexity setting because checking if a string is random according to this definition is undecidable.

Statisticians have also attempted definitions which boil down to checking if the string has the "right number" of patterns that one would expect by the laws of statistics, e.g. the number of times 11100 appears as a substring. (See Knuth Volume 3 for a comprehensive discussion.) It turns out that such definitions are too weak in the cryptographic setting: one can find a distribution that passes these statistical tests but still will be completely insecure if used to generate the pad for the one-time pad encryption scheme.

10.2.1   Blum-Micali and Yao definitions

Now we introduce two complexity-theoretic definitions of pseudorandomness due to Blum-Micali and Yao in the early 1980s. For a string $y \in \{0,1\}^n$ and $S \subseteq [n]$, we let $y|_S$ denote the projection of Y to the coordinates of S. In particular, $y|_{[1..i]}$ denotes the first i bits of y.

The Blum-Micali definition is motivated by the observation that one property (in fact, the defining property) of a statistically random sequence of bits y is that given $y|_{[1..i]}$, we cannot predict y_{i+1} with odds better than 50/50 *regardless of the computational power available to us.* Thus one could define a "pseudorandom" string by considering predictors that have limited computational resources, and to show that they cannot achieve odds much better than 50/50 in predicting y_{i+1} from $y|_{[1..i]}$. Of course, this definition has the shortcoming that any single finite string would be predictable for a trivial reason: it could be hardwired into the program of the predictor Turing machine. To get around this difficulty the Blum-Micali definition (and also Yao's definition below) defines pseudorandomness for distributions of strings rather than for individual strings. Furthermore, the definition concerns an infinite sequence of distributions, one for each input size.

DEFINITION 10.9 (BLUM-MICALI)
Let $\{g_n\}$ be a polynomial-time computable family of functions, where $g_n : \{0,1\}^n \to \{0,1\}^m$ and $m = m(n) > n$. We say the family is $(\epsilon(n), t(n))$-*unpredictable* if for every probabilistic polynomial-time algorithm A that runs in time $t(n)$ and every large enough input size n,

$$\Pr[A(g(x)_{[1..i]}) = g(x)_{i+1}] \leq \frac{1}{2} + \epsilon(n),$$

where the probability is over the choice of $x \in \{0,1\}^n, i \in \{1, \ldots, n\}$, and the randomness used by A.

If for every fixed k, the family $\{g_n\}$ is $(1/n^c, n^k)$-unpredictable for every $c > 1$, then we say in short that it is *unpredictable by polynomial-time algorithms.*

REMARK 10.10
Allowing the tester to be an arbitrary polynomial-time machine makes perfect sense in a cryptographic setting where we wish to assume nothing about the adversary except an upperbound on her computational power.

Pseudorandom generators proposed in the pre-complexity era, such as the popular linear or quadtratic congruential generators do not satisfy the Blum-Micali definition because bit-prediction can in fact be done in polynomial time.

Yao gave an alternative definition in which the tester machine is given access to the entire string at once. This definition implicitly sets up a test of randomness analogous to the more famous Turing test for intelligence (see Figure 10.2). The tester machine A is given a string $y \in \{0,1\}^{n^c}$ that is produced in one of two ways: it is either drawn from the uniform distribution on $\{0,1\}^{n^c}$ or generated by taking a random string $x \in \{0,1\}^n$ and stretching it using a deterministic function $g : \{0,1\}^n \to \{0,1\}^{n^c}$. The tester is asked to output "1" if the string looks random to it and 0 otherwise. We say that g is a pseudorandom generator if no polynomial-time tester machine A has a great chance of being able to determine which of the two distributions the string came from.

DEFINITION 10.11 ([YAO82])
Let $\{g_n\}$ be a polynomial-time computable family of functions, where $g_n : \{0,1\}^n \to \{0,1\}^m$ and $m = m(n) > n$. We say it is a $(\delta(n), s(n))$-*pseudorandom generator* if for every probabilistic algorithm A running in time $s(n)$ and for all large enough n

$$|\mathbf{Pr}_{y \in \{0,1\}^{n^c}}[A(y) = 1] - \mathbf{Pr}_{x \in \{0,1\}^n}[A(g_n(x)) = 1]| \leq \delta(n). \tag{2}$$

We call $\delta(n)$ the *distinguishing probability* and $s(n)$ the *security parameter*.

If for every $c', k > 1$, the family is $(1/n^{c'}, n^k)$-pseudorandom then we say in short that it is a *pseudorandom generator*.

<center>Figure unavailable in pdf file.</center>

Figure 10.2: Yao's definition: If $c > 1$ then $g : \{0, 1\}^n \to \{0, 1\}^{n^c}$ is a *pseudorandom generator* if no polynomial-time tester has a good chance of distinguishing between truly random strings of length n^c and strings generated by applying g on random n-bit strings.

10.2.2   Equivalence of the two definitions

Yao showed that the above two definitions are equivalent —up to minor changes in the security parameter, a family is a pseudorandom generator iff it is (bitwise) unpredictable. The *hybrid argument* used in this proof has become a central idea of cryptography and complexity theory.

The nontrivial direction of the equivalence is to show that pseudorandomness of the Blum-Micali type implies pseudorandomness of the Yao type. Not surprisingly, this direction is also more important in a practical sense. Designing pseudorandom generators seems easier for the Blum-Micali definition —as illustrated by the Goldreich-Levin construction below— whereas Yao's definition seems more powerful for applications since it allows the adversary unrestricted access to the pseudorandom string. Thus Yao's theorem provides a bridge between what we can prove and what we need.

THEOREM 10.12 (PREDICTION VS. INDISTINGUISHABILITY [**?**])
Let Let $g_n : \{0, 1\}^n \to \{0, 1\}^{N(n)}$ be a family of functions where $N(n) = n^k$ for some $k > 1$.
If g_n is $(\frac{\epsilon(n)}{N(n)}, 2t(n))$-unpredictable where $t(n) \geq N(n)^2$ then it is $(\epsilon(n), t(n))$-pseudorandom.
Conversely, if g_n is $(\epsilon(n), t(n))$-pseudorandom, then it is $(\epsilon(n), t(n))$-unpredictable.

PROOF: The

---

converse part is trivial since a bit-prediction algorithm can in particular be used to distinguish $g(x)$ from random strings of the same length. It is left to the reader.

Let N be shorthand for $N(n)$. Suppose g is not $(\epsilon(n), t(n))$-pseudorandom, and A is a distinguishing algorithm that runs in $t(n)$ time and satisfies:

$$\left| \Pr_{x \in B^n}[A(g(x)) = 1] - \Pr_{y \in \{0,1\}^N}[A(y) = 1] \right| > \epsilon(n). \qquad (3)$$

By considering either A or the algorithm that is A with the answer flipped, we can assume that the $|\cdot|$ can be removed and in fact

$$\Pr_{x \in B^n}[A(g(x)) = 1] - \Pr_{y \in \{0,1\}^N}[A(y) = 1] > \epsilon(n). \qquad (4)$$

Consider B, the following bit-prediction algorithm. Let its input be $g(x)|_{\leq i}$ where $x \in \{0,1\}^n$ and $i \in \{0, \ldots, N-1\}$ are chosen uniformly at random. B's program is: "*Pick bits $u_{i+1}, u_{i+2}, \ldots, u_N$ randomly and run A on the input $g(x)|_{\leq i}u_{i+1}u_{i+2}\ldots u_N$. If A outputs 1, output u_{i+1} else output $\overline{u_{i+1}}$.*" Clearly, B runs in time less than $t(n) + O(N(n)) < 2t(n)$. To complete the proof we show that B predicts $g(x)_{i+1}$ correctly with probability at least $\frac{1}{2} + \frac{\epsilon(n)}{N}$.

Consider a sequence of $N+1$ distributions $\mathcal{D}_0$ through $\mathcal{D}_N$ defined as follows (in all cases, $x \in \{0,1\}^n$ and $u_1, u_2, \ldots, u_N \in \{0,1\}$ are assumed to be chosen randomly)

$$\mathcal{D}_0 = u_1 u_2 u_3 u_4 \cdots u_N$$
$$\mathcal{D}_1 = g(x)_1 u_2 u_3 \cdots u_N$$
$$\vdots \qquad \vdots$$
$$\mathcal{D}_i = g(x)_{\leq i} u_{i+1} \cdots u_N$$
$$\vdots \qquad \vdots$$
$$\mathcal{D}_N = g(x)_1 g(x)_2 \cdots g(x)_N$$

Furthermore, we denote by $\overline{\mathcal{D}_i}$ the distribution obtained from $\mathcal{D}_i$ by flipping the ith bit (i.e., replacing $g(x)_i$ by $\overline{g(x)_i}$). If $\mathcal{D}$ is any of these $2(N+1)$ distributions then we denote $\Pr_{y \in \mathcal{D}}[A(y) = 1]$ by $q(\mathcal{D})$. With this notation we rewrite (4) as

$$q(\mathcal{D}_N) - q(\mathcal{D}_0) > \epsilon(n). \tag{5}$$

Furthermore, in $\mathcal{D}_i$, the $(i+1)$th bit is equally likely to be $g(x)_{i+1}$ and $\overline{g(x)_{i+1}}$, so

$$q(\mathcal{D}_i) = \tfrac{1}{2}(q(\mathcal{D}_{i+1}) + q(\overline{\mathcal{D}_{i+1}})), \tag{6}$$

Now we analyze the probability that B predicts $g(x)_{i+1}$ correctly. Since i is picked randomly we have

$$\Pr_{i,x}[B \text{ is correct}] = \frac{1}{N} \sum_{i=0}^{n-1} \frac{1}{2} \left(\Pr_{x,\mathbf{u}}[B\text{'s guess for } g(x)_{i+1} \text{ is correct} \mid u_{i+1} = g(x)_{i+1}] \right.$$

$$\left. + \Pr_{x,\mathbf{u}}[B\text{'s guess for } g(x)_{i+1} \text{ is correct} \mid u_{i+1} = \overline{g(x)_{i+1}}] \right).$$

Since B's guess is u_{i+1} iff A outputs 1 this is

$$= \frac{1}{2N} \sum_{i=0}^{N-1} (q(\mathcal{D}_{i+1}) + 1 - q(\overline{\mathcal{D}_{i+1}}))$$

$$= \frac{1}{2} + \frac{1}{2N} \sum_{i=0}^{N-1} (q(\mathcal{D}_{i+1}) - q(\overline{\mathcal{D}_{i+1}}))$$

From (6), $q(\mathcal{D}_{i+1}) - q(\overline{\mathcal{D}_{i+1}}) = 2(q(\mathcal{D}_{i+1}) - q(\mathcal{D}_i))$, so this becomes

$$
\begin{aligned}
&= \frac{1}{2} + \frac{1}{2N} \sum_{i=0}^{N-1} 2(q(\mathcal{D}_{i+1}) - q(\mathcal{D}_i)) \\
&= \frac{1}{2} + \frac{1}{N}(q(\mathcal{D}_N) - q(\mathcal{D}_0)) \\
&> \frac{1}{2} + \frac{\epsilon(n)}{N}.
\end{aligned}
$$

This finishes our proof. ∎

10.3 One-way functions and pseudorandom number generators

Do pseudorandom generators exist? Surprisingly the answer (though we will not prove it in full generality) is that they do if and only if *one-way functions* exist.

THEOREM 10.13
One-way functions exist iff pseudorandom generators do.

Since we had several plausible candidates for one-way functions in Section 10.1, this result helps us design pseudorandom generators using those candidate one-way functions. If the pseudorandom generators are ever proved to be insecure, then the candidate one-way functions were in fact not one-way, and so we would obtain (among other things) efficient algorithms for FACTORING and DISCRETE LOG.

The "if" direction of Theorem 10.13 is trivial: if g is a pseudorandom generator then it must also be a one-way function since otherwise the algorithm that inverts g would be able to distinguish its outputs from random strings. The "only if" direction is more difficult and involves using a one-way function to explicitly construct a pseudorandom generator. We will do this only for the special case of one-way functions that are *permutations*, namely, they map $\{0,1\}^n$ to $\{0,1\}^n$ in a one-to-one and onto fashion. As a first step, we describe the Goldreich-Levin theorem, which gives an easy way to produce one pseudorandom bit, and then describe how to produce n^c pseudorandom bits.

10.3.1 Goldreich-Levin hardcore bit

Let $\{f_n\}$ be a one-way permutation where $f_n : \{0,1\}^n \to \{0,1\}^n$. Clearly, the function $g : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^{2n}$ defined as $g(x,r) = (f(x), r)$ is also a one-way permutation. Goldreich and Levin showed that given $(f(x), r)$, it is difficult for a polynomial-time algorithm to predict $x \odot r$, the scalar product of x and r (mod 2). Thus even though the string $(f(x), r)$ in principle contains all the information required to extract (x, r), it is computationally difficult to extract even the single bit $x \odot r$. This bit is called a *hardcore* bit for the permutation. Prior to the Goldreich-Levin result we knew of hardcore bits for some specific (conjectured) one-way permutations, not all.

THEOREM 10.14 (GOLDREICH-LEVIN THEOREM)
Suppose that $\{f_n\}$ is a family of $\epsilon(n)$)-one-way permutation with security $s(n)$. Let $S(n) = (\min\left\{s(n), \frac{1}{\epsilon(n)}\right\})^{1/8}$ Then for all algorithms A running in time $S(n)$

$$\mathbf{Pr}_{x,r\in\{0,1\}^n}[A(f_n(x),r) = x \odot r] \leq \frac{1}{2} + O(\frac{1}{S(n)}). \tag{7}$$

PROOF: Suppose that some algorithm A can predict $x \odot r$ with probability $1/2 + \delta$ in time $t(n)$. We show how to invert $f_n(x)$ for $O(\delta)$ fraction of the inputs in $O(n^3 t(n)/\delta^4)$ time, from which the theorem follows.

CLAIM 10.15
Suppose that

$$\mathbf{Pr}_{x,r\in\{0,1\}^n}[A(f_n(x),r) = x \odot r] \geq \frac{1}{2} + \delta. \tag{8}$$

Then for at least δ fraction of x's

$$\mathbf{Pr}_{r\in\{0,1\}^n}[A(f_n(x),r) = x \odot r] \geq \frac{1}{2} + \frac{\delta}{2}. \tag{9}$$

PROOF: We use an averaging argument. Suppose that p is the fraction of x's satisfying (9). We have $p \cdot 1 + (1-p)(1/2 + \delta/2) \geq 1/2 + \delta$. Solving this with respect to p, we obtain

$$p \geq \frac{\delta}{2(1/2 - \delta/2)} \geq \delta.$$

∎

We design an inversion algorithm that given $f_n(x)$, where $x \in_R \{0,1\}^n$, will try to recover x. It succeeds with high probability if x is such that (9) holds, in other words, for at least δ fraction of x. Note that the algorithm can always check the correctness of its answer, since it has $f_n(x)$ available to it and it can apply f_n to its answer and see if $f_n(x)$ is obtained.

WARMUP: *Reconstruction when the probability in (9) is $\geq 3/4 + \delta$.*

Let P be any program that computes some unknown linear function over $GF(2)^n$ but errs on some inputs. Specifically, there is an unknown vector $x \in GF(2)^n$ such that

$$\Pr_r[P(r) = x \cdot r] = 3/4 + \delta. \tag{10}$$

Then we show to add a simple "correction" procedure to turn P into a probabilistic program P' such that

$$\forall \, r \quad \Pr[P'(r) = x \cdot r] \geq 1 - \frac{1}{n^2}. \tag{11}$$

(Once we know how to compute $x \cdot r$ for every r with high probability, it is easy to recover x bit-by-bit using the observation that if e_i is the n-bit vector that is 1 in the ith position and zero elsewhere then $x \cdot e_i = a_i$, the ith bit of a.)

"On input r, repeat the following trial $O(\log n/\delta^2)$ times. Pick y randomly from $GF(2)^n$ and compute the bit $P(r + y) + P(y)$. At the end, output the majority value."

The main observation is that when y is randomly picked from $GF(2)^n$ then $r + y$ and y are both randomly distributed in $GF(2)^n$, and hence the probability that $P(r + y) \neq a \cdot (r + y)$ or $P(y) \neq a \cdot y$ is at most $2 \cdot (1/4 - \delta) = 1/2 - 2\delta$. Thus with probability at least $1/2 + 2\delta$, each trial produces the correct bit. Then Chernoff bounds imply that probability is at least $1 - 1/n^2$ that the final majority is correct.

GENERAL CASE:

The idea for the general case is very similar, the only difference being that this time we want to pick $r_1, \ldots, r_m$ so that we already "know" $x \odot r_i$. The preceding statement may appear ridiculous, since knowing the inner product of x with $m \geq n$ random vectors is, with high probability, enough to reconstruct x (see exercises). The explanation is that the r_i's will not be completely random. Instead, they will be pairwise independent. Recall the following construction of a set of pairwise independent vectors: Pick k random vectors $t_1, t_2, \ldots, t_k \in GF(2)^n$ and for each nonempty $S \subseteq \{1, \ldots, k\}$ define $Y_S = \sum_{i \in S} t_i$. This gives $2^k - 1$ vectors and for $S \neq S'$ the random variables $Y_S, Y_{S'}$ are independent of each other.

Now let us describe the observation at the heart of the proof. Suppose $m = 2^k - 1$ and our random strings $r_1, \ldots, r_m$ are $\{Y_S\}$'s from the previous paragraph. Then $x \odot Y_S = x \odot (\sum_{i \in S} t_i) = \sum_{i \in S} x \odot t_i$. Hence if we know $x \odot t_i$ for $i = 1, \ldots, k$, we also know $x \odot Y_S$. Of course, we don't actually know $x \odot t_i$ for $i = 1, \ldots, k$ since x is unknown and the t_i's are random vectors. But we can just try all 2^k possibilities for the vector $(x \odot t_i)_{i=1,\ldots,k}$ and run the rest of the algorithm for each of them. Whenever our "guess" for these innerproducts is correct, the algorithm succeeds in producing x and this answer can be checked by applying f_n on it (as already noted). Thus the guessing multiplies the running time by a factor 2^k, which is only m. This is why we can assume that we know $x \odot Y_S$ for each subset S.

The details of the rest of the algorithm are similar to before. *Pick m pairwise independent vectors Y_S's such that, as described above, we "know" $x \odot Y_S$ for all S. For each $i = 1, 2, \ldots, n$, and each S run A on the input $(f_n(x), Y_S \oplus e_i)$ (where $Y_S \oplus e_i$ is Y_S with its ith entry flipped). Compute the majority value of $A(f_n(x), Y_S \oplus e_i) - x \odot Y_S$ among all S's and use it as your guess for x_i.*

Suppose $x \in GF(2)^n$ satisfies (9). We will show that this algorithm produces all n bits of x with probability at least $1/2$. Fix i. For each i, the guess for x_i is a majority of m bits. The expected number of bits among these that agree with x_i is $m(1/2 + \delta/2)$, so for the majority vote to result in the incorrect answer it must be the case that the number of incorrect values deviates from its expectation by more than $m\delta/2$. Now, we can bound the variance of this random variable and apply Chebyshev's inequality (Lemma A.16 in Appendix A) to conclude that the probability of such a deviation is $\leq \frac{4}{m\delta^2}$.

Here is the calculation using Chebyshev's inequality. Let ξ_S denote the event that A produces the correct answer on $(f_n(x), Y_S \oplus e_i)$. Since x satisfies (9) and $Y_S \oplus e_i$ is randomly distributed over $GF(2)^n$, we have $E(\xi_S) = 1/2 + \delta/2$ and $Var(\xi_S) = E(\xi_S)(1 - E(\xi_S)) < 1$. Let $\xi = \sum_S \xi_S$ denote the number of correct answers on a sample of size m. By linearity of expectation, $E[\xi] = m(1/2 + \delta/2)$. Furthermore, the Y_S's are pairwise independent, which implies that the same is true for the outputs ξ_S's produced by the algorithm A on them. Hence by pairwise independence $Var(\xi) < m$. Now, by

Chebyshev's inequality, the probability that the majority vote is incorrect is at most $\frac{4Var(\xi)}{m^2\delta^2} \leq \frac{4}{m\delta^2}$.

Finally, setting $m > 8/n\delta^2$, the probability of guessing the ith bit incorrectly is at most $1/2n$. By the union bound, the probability of guessing the whole word incorrectly is at most $1/2$. Hence, for every x satisfying (9), we can find the preimage of $f(x)$ with probability at least $1/2$, which makes the overall probability of inversion at least $\delta/2$. The running time is about $m^2 n \times$ (running time of A), which is $\frac{n^3}{\delta^4} \times t(n)$, as we had claimed. $\blacksquare$

10.3.2 Pseudorandom number generation

We saw that if f is a one-way permutation, then $g(x,r) = (f(x), r, x \odot r)$ is a pseudorandom generator that stretches $2n$ bits to $2n+1$ bits. Stretching to even more bits is easy too, as we now show. Let $f^i(x)$ denote the i-th iterate of f on x (i.e., $f(f(f(\cdots(f(x)))))$) where f is applied i times).

THEOREM 10.16
If f is a one-way permutation then $g_N(x,r) = (r, x \odot r, f(x) \odot r, f^2(x) \odot r, \ldots, f^N(x) \odot r)$ is a pseudorandom generator for $N = n^c$ for any constant $c > 0$.

PROOF: Since any distinguishing machine could just reverse the string as a first step, it clearly suffices to show that the string $(r, f^N(x) \odot r, f^{N-1}(x) \odot r, \ldots, f(x) \odot r, x \odot r)$ looks pseudorandom. By Yao's theorem (Theorem 10.12), it suffices to show the difficulty of bit-prediction. For contradiction's sake, assume there is a PPT machine A such that when $x, r \in \{0,1\}^n$ and $i \in \{1, \ldots, N\}$ are randomly chosen,

$$\Pr[A \text{ predicts } f^i(x) \odot r \text{ given } (r, f^N(x) \odot r, f^{N-1}(x) \odot r, \ldots, f^{i+1}(x) \odot r)] \geq \frac{1}{2} + \epsilon.$$

We describe an algorithm B that given $f(z), r$ where $z, r \in \{0,1\}^n$ are randomly chosen, predicts the hardcore bit $z \odot r$ with reasonable probability, which contradicts Theorem 10.14.

Algorithm B picks $i \in \{1, \ldots, N\}$ randomly. Let $x \in \{0,1\}^n$ be such that $f^i(x) = z$. There is of course no efficient way for B to find x, but for any $l \geq 1$, B can efficiently compute $f^{i+l}(x) = f^{l-1}(f(z))$! So it produces the string $r, f^N(x) \odot r, f^{N-1}(x) \odot r, \ldots, f^{i+1}(x) \odot r$ and uses it as input to A. By assumption, A predicts $f^i(x) \odot r = z \odot r$ with good odds. Thus we have derived a contradiction to Theorem 10.14. $\blacksquare$

10.4 Applications

Now we give some applications of the ideas introduced in the chapter.

10.4.1 Pseudorandom functions

Pseudorandom functions are a natural generalization of (and are easily constructed using) pseudorandom generators. This is a function $g: \{0,1\}^m \times \{0,1\}^n \rightarrow \{0,1\}^m$. For each $K \in \{0,1\}^m$ we denote by $g|_K$ the function from $\{0,1\}^n$ to $\{0,1\}^m$ defined by $g|_K(x) = g(K,x)$. Thus the family contains 2^m functions from $\{0,1\}^n$ to $\{0,1\}^m$, one for each K.

We say g is a pseudorandom function generator if it passes a "Turing test" of randomness analogous to that in Yao's definition of a pseudorandom generator (Definition 10.11).

Recall that the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$, denoted $\mathcal{F}_{n,m}$, has cardinality $(2^m)^{2^n}$. The PPT machine is presented with an "oracle" for a function from $\{0,1\}^n$ to $\{0,1\}^n$. The function is one of two types: either a function chosen randomly from $\mathcal{F}_{n,m}$, or a function $f|_K$ where $K \in \{0,1\}^m$ is randomly chosen. The PPT machine is allowed to query the oracle in any points of its choosing. We say $f|_K$ is a pseudorandom function generator if for all $c > 1$ the PPT has probability less than n^{-c} of detecting which of the two cases holds. (A completely formal definition would resemble Definition 10.1 and talk about a family of generators, one for each n. Then m is some function of n.)

Figure unavailable in pdf file.

Figure 10.3: Constructing a pseudorandom function from $\{0,1\}^n$ to $\{0,1\}^m$ using a random key $K \in \{0,1\}^m$ and a length-doubling pseudorandom generator $g \colon \{0,1\}^m \to \{0,1\}^{2m}$.

Now we describe a construction of a pseudorandom function generator g from a length-doubling pseudorandom generator $f \colon \{0,1\}^m \to \{0,1\}^{2m}$. For any $K \in \{0,1\}^m$ let T_K be a complete binary tree of depth n whose each node is labelled with an m-bit string. The root is labelled K. If a node in the tree has label y then its left child is labelled with the first m bits of $f(y)$ and the right child is labelled with the last m bits of $f(y)$. Now we define $g(K, x)$. For any $x \in \{0,1\}^n$ interpret x as a label for a path from root to leaf in T_K in the obvious way and output the label at the leaf. (See Figure 10.3.)

We leave it as an exercise to prove that this construction is correct.

A pseudorandom function generator is a way to turn a random string K into an implicit description of an exponentially larger "random looking" string, namely, the table of all values of the function $g|_K$. This has proved a powerful primitive in cryptography; see the next section. Furthermore, pseudorandom function generators have also figured in a very interesting explanation of why current lowerbound techniques have been unable to separate **P** from **NP**; see Chapter **??**.

10.4.2   Private-key encryption: definition of security

We hinted at a technique for private-key encryption in our discussion of a one-time pad (including the pseudorandom version) at the start of Section 10.2. But that discussion completely omitted what the *design goals* of the encryption scheme were. This is an important point: design of insecure systems often traces to a misunderstanding about the type of security ensured (or not ensured) by an underlying protocol.

The most basic type of security that a private-key encryption should ensure is *semantic security*. Informally speaking, this means that whatever can be computed from the encrypted message is also computable without access to the encrypted message and knowing only the *length* of the message. The formal definition is omitted here but it has to emphasize the facts that we are talking about an ensemble of encryption functions, one for each message size (as in Definition 10.1) and that the encryption and decryption is done by probabilistic algorithms that use a shared private key, and

DRAFT

that *for every message* the guarantee of security holds with high probability with respect to the choice of this private key.

Now we describe an encryption scheme that is semantically secure. Let $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ be a pseudorandom function generator. The two parties share a secret random key $K \in \{0,1\}^n$. When one of them wishes to send a message $x \in \{0,1\}^n$ to the other, she picks a random string $r \in \{0,1\}^n$ and transmits $(r, x \oplus f_K(r))$. To decrypt the other party computes $f_K(r)$ and then XORs this string with the last n bits in the received text.

We leave it as an exercise to show that this scheme is semantically secure.

10.4.3 Derandomization

The existence of pseudorandom generators implies subexponential deterministic algorithms for **BPP**: this is usually referred to as *derandomization* of **BPP**. (In this case, the derandomization is only partial since it results in a subexponential deterministic algorithm. Stronger complexity assumptions imply a full derandomization of **BPP**, as we will see in Chapter 16.)

THEOREM 10.17
If for every $c > 1$ there is a pseudorandom generator that is secure against circuits of size n^c, then **BPP** $\subseteq \cap_{\varepsilon > 0}$**DTIME**$(2^{n^{\varepsilon}})$.

PROOF: Let us fix an $\varepsilon > 0$ and show that **BPP** $\subseteq$ **DTIME**$(2^{n^{\varepsilon}})$.

Suppose that M is a **BPP** machine running in n^k time. We can build another probabilistic machine M' that takes n^{ε} random bits, streches them to n^k bits using the pseudorandom generator and then simulates M using this n^k bits as a random string. Obviously, M' can be simulated by going over all binary strings n^{ε}, running M' on each of them, and taking the majority vote.

It remains to prove that M and M' accept the same language. Suppose otherwise. Then there exists an infinite sequence of inputs $x_1, \ldots, x_n, \ldots$ on which M distinguishes a truly random string from a pseudorandom string with a high probability, because for M and M' to produce different results, the probability of acceptance should drop from 2/3 to below 1/2. Hence we can build a distinguisher similar to the one described in the previous theorem by hardwiring these inputs into a circuit family. ∎

The above theorem shows that the existence of hard problems implies that we can reduce the randomness requirement of algorithms. This "hardness versus randomness" tradeoff is studied more deeply in Chapter 16.

REMARK 10.18
There is an interesting connection to *discrepancy theory*, a field of mathematics. Let $\mathcal{S}$ be a set of subsets of $\{0,1\}^n$. Subset $A \subset \{0,1\}^n$ has *discrepancy ϵ with respect to $\mathcal{S}$* if for every $s \in \mathcal{S}$,

$$\left| \frac{|s \cap A|}{|S|} - \frac{|A|}{2^n} \right| \le \epsilon.$$

Our earlier result that **BPP** $\subseteq$ **P**/*poly* showed the *existence* of polynomial-size sets A that have low discrepancy for all sets defined by polynomial-time Turing machines (we only described discrepancy for the universe $\{0,1\}^n$ but one can define it for all input sizes using lim sup). The goal of derandomization is to explicitly construct such sets; see Chapter 16.

10.4.4 Tossing coins over the phone and bit commitment

How can two parties A and B toss a fair random coin over the phone? (Many cryptographic protocols require this basic primitive.) If only one of them actually tosses a coin, there is nothing to prevent him from lying about the result. The following fix suggests itself: both players toss a coin and they take the XOR as the shared coin. Even if B does not trust A to use a fair coin, he knows that as long as his bit is random, the XOR is also random. Unfortunately, this idea also does not work because the player who reveals his bit first is at a disadvantage: the other player could just "adjust" his answer to get the desired final coin toss.

This problem is addressed by the following scheme, which assumes that A and B are polynomial time turing machines that cannot invert one-way permutations. The protocol itself is called *bit commitment*. First, A chooses two strings x_A and r_A of length n and sends a message $(f_n(x_A), r_A)$, where f_n is a one-way permutation. This way, A commits the string x_A without revealing it. Now B selects a random bit b and conveys it. Then A reveals x_A and they agree to use the XOR of b and $(x_A \odot r_A)$ as their coin toss. Note that B can verify that x_A is the same as in the first message by applying f_n, therefore A cannot change her mind after learning B's bit. On the other hand, by the Goldreich–Levin theorem, B cannot predict $x_A \odot r_A$ from A's first message, so this scheme is secure.

10.4.5 Secure multiparty computations

This concerns a vast generalization of the setting in Section 10.4.4. There are k parties and the ith party holds a string $x_i \in \{0,1\}^n$. They wish to compute $f(x_1, x_2, \ldots, x_k)$ where $f : \{0,1\}^{nk} \to \{0,1\}$ is a polynomial-time computable function known to all of them. (The setting in Section 10.4.4 is a subcase whereby each x_i is a bit —randomly chosen as it happens—and f is XOR.) Clearly, the parties can just exchange their inputs (suitably encrypted if need be so that unauthorized eavesdroppers learn nothing) and then each of them can compute f on his/her own. However, this leads to all of them knowing each other's input, which may not be desirable in many situations. For instance, we may wish to compute statistics (such as the average) on the combination of several medical databases that are held by different hospitals. Strict privacy and nondisclosure laws may forbid hospitals from sharing information about individual patients. (The original example Yao gave in introducing the problem was of k people who wish to compute the average of their salaries without revealing their salaries to each other.)

We say that a multiparty protocol for computing f is *secure* if at the end no party learns anything new apart from the value of $f(x_1, x_2, \ldots, x_k)$. The formal definition is inspired by the definition of a pseudorandom generator, and states that for each i, the bits received by party i during the protocol should be computationally indistinguishable from completely random bits[3].

It is completely nonobvious why such protocols must exist. Yao [Yao86] proved existence for $k = 2$ and Goldreich, Micali, Wigderson [GMW87] proved existence for general k. We will not

---

[3]Returning to our medical database example, we see that the hospitals can indeed compute statistics on their combined databases without revealing any information to each other —at least any information that can be extracted feasibly. Nevertheless, it is unclear if current privacy laws allow hospitals to perform such secure multiparty protocols using patient data— an example of the law lagging behind scientific progress.

describe this protocol in any detail here except to mention that it involves "scrambling" the circuit that computes f.

10.4.6   Lowerbounds for machine learning

In *machine learning* the goal is to learn a succinct function $f : \{0,1\}^n \to \{0,1\}$ from a sequence of type $(x_1, f(x_1)), (x_2, f(x_2)), \ldots$, where the x_i's are randomly-chosen inputs. Clearly, this is impossible in general since a random function has no succinct description. But suppose f has a succinct description, e.g. as a small circuit. Can we learn f in that case?

   The existence of pseudorandom functions implies that even though a function may be polynomial-time computable, there is no way to learn it from examples in polynomial time. In fact it is possible to extend this impossibility result (though we do not attempt it) to more restricted function families such as $\mathbf{NC}^1$ (see Kearns and Valiant [KV94]).

10.5   Recent developments

The earliest cryptosystems were designed using the SUBSET SUM problem. They were all shown to be insecure by the early 1980s. In the last few years, interest in such problems —and also the related problems of computing approximate solutions to the shortest and nearest lattice vector problems— has revived, thanks to a one-way function described in Ajtai [Ajt96], and a public-key cryptosystem described in Ajtai and Dwork [AD97] (and improved on since then by other researchers). These constructions are secure on *most* instances iff they are secure on *worst-case* instances. (The idea used is a variant of random self-reducibility.)

   Also, there has been a lot of exploration of the exact notion of security that one needs for various cryptographic tasks. For instance, the notion of semantic security in Section 10.4.2 may seem quite strong, but researchers subsequently realized that it leaves open the possibility of some other kinds of attacks, including *chosen ciphertext* attacks, or attacks based upon *concurrent* execution of several copies of the protocol. Achieving security against such exotic attacks calls for many ideas, most notably *zero knowledge* (a brief introduction to this concept appears in Section **??**).

Chapter notes and history

In the 1940s, Shannon speculated about topics reminiscent of complexity-based cryptography. The first concrete proposal was made by Diffie and Hellman [DH76], though their cryptosystem was later broken. The invention of the RSA cryptosystem (named after its inventors Ron Rivest, Adi Shamir, and Len Adleman) [RSA78] brought enormous attention to this topic. In 1981 Shamir [Sha83] suggested the idea of replacing a one-time pad by a pseudorandom string. He also exhibited a weak pseudorandom generator assuming the average-case intractability of the RSA function. The more famous papers of Blum and Micali [BM84] and then Yao [Yao82] laid the intellectual foundations of private-key cryptography. (The hybrid argument used by Yao is a stronger version of one in an earlier important manuscript of Goldwasser and Micali [GM84] that proposed probabilistic encryption schemes.) The construction of pseudorandom functions in Section 10.4.1 is due to Goldreich, Goldwasser, and Micali [GGM86]. The question about tossing coins over a telephone

was raised in an influential paper of Blum [Blu82]. Today complexity-based cryptography is a vast field with several dedicated conferences. Goldreich [Gol04]'s two-volume book gives a definitive account.

A scholarly exposition of number theoretic algorithms (including generating random primes and factoring integers) appears in Victor Shoup's recent book [?] and the book of Bach and Shallit [BS96].

Theorem 10.13 and its very technical proof is in Håstad et al. [HILL99] (the relevant conference publications are a decade older).

Our proof of the Goldreich-Levin theorem is usually attributed to Rackoff (unpublished).

Exercises

§1 Show that if $\mathbf{P} = \mathbf{NP}$ then one-way functions and pseudorandom generators do not exist.

§2 (Requires just a little number theory). Prove that if some algorithm inverts the Rabin function $f_m(x) = x^2 \pmod{m}$ on a $1/\text{poly}(\log m)$ fraction of inputs then we can factor m in $\text{poly}(\log m)$ time.

Hint: Suppose $m = pq$ where p, q are prime numbers. Then x^2 has 4 "square roots" modulo m.

§3 Show that if f is a one-way permutation then so is f^k (namely, $f(f(f(\cdots(f(x)))))$ where f is applied k times) where $k = n^c$ for some fixed $c > 0$.

§4 Assuming one-way functions exist, show that the above fails for one-way functions.

Hint: You have to design a one-way function where f^k is not one-way.

§5 Suppose $a \in \mathbf{GF(2)}^m$ is an unknown vector. Let $r_1, r_2, \ldots, r_m \in \mathbf{GF(2)}^m$ be randomly chosen, and $a \odot r_i$ revealed to us for all $i = 1, 2, \ldots, m$. Describe a deterministic algorithm to reconstruct a from this information, and show that the probability (over the choice of the r_i's) is at least $1/4$ that it works.

Hint: You need to show that a certain determinant is nonzero.

This shows that the "trick" in Goldreich-Levin's proof is necessary.

§6 Suppose somebody holds an unknown n-bit vector a. Whenever you present a randomly chosen subset of indices $S \subseteq \{1, \ldots, n\}$, then with probability at least $1/2 + \epsilon$, she tells you the parity of the all the bits in a indexed by S. Describe a guessing strategy that allows you to guess a (an n bit string!) with probability at least $(\frac{\epsilon}{n})^c$ for some constant $c > 0$.

§7 Suppose $g : \{0,1\}^n \to \{0,1\}^{n+1}$ is any pseudorandom generator. Then use g to describe a pseudorandom generator that stretches n bits to n^k for any constant $k > 1$.

§8 Show the correctness of the pseudorandom function generator in Section 10.4.1.

Hint: Use a hybrid argument which replaces the labels on the first k levels of the tree by completely random strings. Note that the random labels do not need to be assigned ahead of time — this would take at least 2^k time — but can be assigned on the fly whenever they are needed by the distinguishing algorithm.

§9 Formalize the definition of semantic security and show that the encryption scheme in Section 10.4.2 is semantically secure.

Hint: First show that for all message pairs x, y their encryptions are indistinguishable by polynomial-time algorithms. Why does this suffice?

DRAFT

Part II

Lowerbounds for Concrete Computational Models

In the next few chapters the topic will be concrete complexity, the study of lowerbounds on models of computation such as decision trees, communication games, circuits, etc. Algorithms or devices considered in this lecture take inputs of a fixed size n, and we study the complexity of these devices as a function of n.

Chapter 11

Decision Trees

A *decision tree* is a model of computation used to study the number of bits of an input that need to be examined in order to compute some function on this input. Consider a function $f : \{0,1\}^n \to \{0,1\}$. A decision tree for f is a tree for which each node is labelled with some x_i, and has two outgoing edges, labelled 0 and 1. Each tree leaf is labelled with an output value 0 or 1. The computation on input $x = x_1 x_2 \dots x_n$ proceeds at each node by inspecting the input bit x_i indicated by the node's label. If $x_i = 1$ the computation continues in the subtree reached by taking the 1-edge. The 0-edge is taken if the bit is 0. Thus input x follows a path through the tree. The output value at the leaf is $f(x)$. An example of a simple decision tree for the majority function is given in Figure 11.1

Figure unavailable in pdf file.

Figure 11.1: A decision tree for computing the majority function $Maj(x_1, x_2, x_3)$ on three bits. Outputs 1 if at least two input bits are 1, else outputs 0.

Recall the use of decision trees in the proof of the lower bound for comparison-based sorting algorithms. That study can be recast in the above framework by thinking of the input —which consisted of n numbers — as consisting of $\binom{n}{2}$ bits, each giving the outcome of a pairwise comparison between two numbers.

We can now define two useful decision tree metrics.

DEFINITION 11.1
The cost of tree t on input x, $cost(t,x)$, is the number of bits of x examined by t.

DEFINITION 11.2
The **decision tree complexity** of function f, $D(f)$, is defined as follows, where T below refers to the set of decision trees that decide f.

$$D(f) = \min_{t \in T} \max_{x \in \{0,1\}^n} cost(t,x) \qquad (1)$$

The decision tree complexity of a function is the number of bits examined by the most efficient decision tree on the worst case input to that tree. We are now ready to consider several examples.

EXAMPLE 11.3

(*Graph connectivity*) Given a graph G as input, in adjacency matrix form, we would like to know how many bits of the adjacency matrix a decision tree algorithm might have to inspect in order to determine whether G is connected. We have the following result.

THEOREM 11.4

Let f be a function that computes the connectivity of input graphs with m vertices. Then $D(f) = \binom{m}{2}$.

The idea of the proof of this theorem is to imagine an adversary that constructs a graph, edge by edge, in response to the queries of a decision tree. For every decision tree that decides connectivity, the strategy implicitly produces an input graph which requires the decision tree to inspect each of the $\binom{m}{2}$ possible edges in a graph of m vertices.

Adversary Strategy:
Whenever the decision tree algorithm asks about edge e_i,
answer "no" unless this would force the graph to be disconnected.

After i queries, let N_i be the set of edges for which the adversary has replied "no", Y_i the set of edges for which the adversary has replied "yes". and E_i the set of edges not yet queried. The adversary's strategy maintains the invariant that Y_i is a disconnected forest for $i < \binom{m}{2}$ and $Y_i \cup E_i$ is connected. This ensures that the decision tree will not know whether the graph is connected until it queries every edge.

EXAMPLE 11.5

(*OR Function*) Let $f(x_1, x_2, \ldots x_n) = \bigvee_{i=1}^{n} x_i$. Here we can use an adversary argument to show that $D(f) = n$. For any decision tree query of an input bit x_i, the adversary responds that x_i equals 0 for the first $n - 1$ queries. Since f is the OR function, the decision tree will be in suspense until the value of the nth bit is revealed. Thus $D(f)$ is n.

EXAMPLE 11.6

Consider the AND-OR function, with $n = 2^k$. We define f_k as follows.

$$f_k(x_1, \ldots, x_n) = \begin{cases} f_{k-1}(x_1, \ldots x_{2^{k-1}-1}) \wedge f_{k-1}(x_{2^{k-1}}, \ldots x_{2^k}) & \text{if } k \text{ is even} \\ f_{k-1}(x_1, \ldots x_{2^{k-1}-1}) \vee f_{k-1}(x_{2^{k-1}}, \ldots x_{2^k}) & \text{if } k > 1 \text{ and is odd} \\ x_i & \text{if } k = 1 \end{cases} \qquad (2)$$

A diagram of a circuit that computes the AND-OR function is shown in Figure 11.2. It is left as an exercise to prove, using induction, that $D(f_k) = 2^k$.

Figure unavailable in pdf file.

Figure 11.2: A circuit showing the computation of the AND-OR function. The circuit has k layers of alternating gates, where $n = 2^k$.

11.1 Certificate Complexity

We now introduce the notion of *certificate complexity*, which, in a manner analogous to decision tree complexity above, tells us the minimum amount of information needed to be convinced of the value of a function f on input x.

DEFINITION 11.7
Consider a function $f : \{0,1\}^n \to \{0,1\}$. If $f(x) = 0$, then a **0-certificate** for x is a sequence of bits in x that proves $f(x) = 0$. If $f(x) = 1$, then a **1-certificate** is a sequence of bits in x that proves $f(x) = 1$.

DEFINITION 11.8
The **certificate complexity** $C(f)$ of f is defined as follows.

$$C(f) = \max_{x:input} \{\text{number of bits in the smallest 0- or 1- certificate for x}\} \qquad (3)$$

EXAMPLE 11.9
If f is a function that decides connectivity of a graph, a 0-certificate for an input must prove that some cut in the graph has no edges, hence it has to contain all the possible edges of a cut of the graph. When these edges do not exist, the graph is disconnected. Similarly, a 1-certificate is the edges of a spanning tree. Thus for those inputs that represent a connected graph, the minimum size of a 1-certificate is the number of edges in a spanning tree, $n - 1$. For those that represent a disconnected graph, a 0 certificate is the set of edges in a cut. The size of a 0-certificate is at most $(n/2)^2 = n^2/4$, and there are graphs (such as the graph consisting of two disjoint cliques of size $n/2$) in which no smaller 0-certificate exists. Thus $C(f) = n^2/4$.

EXAMPLE 11.10
We show that the certificate complexity of the AND-OR function f_k of Example 11.6 is $2^{\lceil k/2 \rceil}$. Recall that f_k is defined using a circuit of k layers. Each layer contains only OR-gates or only AND-gates, and the layers have alternative gate types. The bottom layer receives the bits of input x as input and the single top layer gate outputs the answer $f_k(x)$. If $f(x) = 1$, we can construct a 1-certificate as follows. For every AND-gate in the tree of gates we have to prove that both its children evaluate to 1, whereas for every OR-gate we only need to prove that *some* child evaluates to 1. Thus the 1-certificate is a subtree in which the AND-gates have two children but the OR gates only have one each. Thus the subtree only needs to involve $2^{\lceil k/2 \rceil}$ input bits. If $f(x) = 0$, a similar

argument applies, but the role of OR-gates and AND-gates, and values 1 and 0 are reversed. The result is that the certificate complexity of f_k is $2^{\lceil k/2 \rceil}$, or about $\sqrt{n}$.

The following is a rough way to think about these concepts in analogy to Turing machine complexity as we have studied it.

$$\text{low decision tree complexity} \leftrightarrow \quad \mathbf{P} \tag{4}$$
$$\text{low 1-certificate complexity} \leftrightarrow \quad \mathbf{NP} \tag{5}$$
$$\text{low 0-certificate complexity} \leftrightarrow \quad \mathbf{coNP} \tag{6}$$

The following result shows, however, that the analogy may not be exact since in the decision tree world, $\mathbf{P} = \mathbf{NP} \cap \mathbf{coNP}$. It should be noted that the result is tight, for example for the AND-OR function.

THEOREM 11.11
For function f, $D(f) \leq C(f)^2$.

PROOF: Let S_0, S_1 be the set of minimal 0-certificates and 1-certificates, respectively, for f. Let $k = C(f)$, so each certificate has at most k bits.

REMARK 11.12
Note that every 0-certificate must share a bit position with every 1-certificate, and furthermore, assign this bit differently. If this were not the case, then it would be possible for both a 0-certificate and 1-certificate to be asserted at the same time, which is impossible.

The following decision tree algorithm then determines the value of f in at most k^2 queries.

Algorithm: Repeat until the value of f is determined: Choose a remaining 0-certificate from S_0 and query all the bits in it. If the bits are the values that prove the f to be 0, then stop. Otherwise, we can prune the set of remaining certificates as follows. Since all 1-certificates must intersect the chosen 0-certificate, for any $c_1 \in S_1$, one bit in c_1 must have been queried here. Eliminate c_1 from consideration if the certifying value of c_1 at at location is different from the actual value found. Otherwise, we only need to consider the remaining $k - 1$ bits of c_1.

This algorithm can repeat at most k times. For each iteration, the unfixed lengths of the uneliminated 1-certificates decreases by one. This is because once some values of the input have been fixed due to queries, for any 0-certificate, it remains true that all 1-certificates must intersect it in at least one location that has not been fixed, otherwise it would be possible for both a 0-certificate and a 1-certificate to be asserted. With at most k queries for at most k iterations, a total of k^2 queries is used. ∎

11.2 Randomized Decision Trees

There are two equivalent ways to look at randomized decision trees. We can consider decision trees in which the branch taken at each node is determined by the query value and by a random coin flip. We can also consider probability distributions over deterministic decision trees. The analysis that follows uses the latter model.

We will call $\mathcal{P}$ a probability distribution over a set of decision trees $\mathcal{T}$ that compute a particular function. $\mathcal{P}(t)$ is then the probability that tree t is chosen from the distribution. For a particular input x, then, we define $c(\mathcal{P}, x) = \sum_{t in \mathcal{T}} \mathcal{P}(t) cost(t, x)$. $c(\mathcal{P}, x)$ is thus the expected number of queries a tree chosen from $\mathcal{T}$ will make on input x. We can then characterize how well randomized decision trees can operate on a particular problem.

DEFINITION 11.13
The **randomized decision tree complexity**, $\mathcal{R}(f)$, of f, is defined as follows.

$$\mathcal{R}(f) = \min_{\mathcal{P}} \max_{x} c(\mathcal{P}, x) \tag{7}$$

The randomized decision tree complexity thus expresses how well the best possible probability distribution of trees will do against the worst possible input for a particular probability distribution of trees. We can observe immediately that $\mathcal{R}(f) \geq C(f)$. This is because $C(f)$ is a minimum value of $cost(t, x)$. Since $\mathcal{R}(f)$ is just an expected value for a particular probability distribution of these *cost* values, the minimum such value can be no greater than the expected value.

EXAMPLE 11.14
Consider the majority function, $f = Maj(x_1, x_2, x_3)$. It is straightforward to see that $D(f) = 3$. We show that $\mathcal{R}(f) \leq 8/3$. Let $\mathcal{P}$ be a uniform distribution over the (six) ways of ordering the queries of the three input bits. Now if all three bits are the same, then regardless of the order chosen, the decision tree will produce the correct answer after two queries. For such x, $c(\mathcal{P}, x) = 2$. If two of the bits are the same and the third is different, then there is a 1/3 probability that the chosen decision tree will choose the two similar bits to query first, and thus a 1/3 probability that the cost will be 2. There thus remains a 2/3 probability that all three bits will need to be inspected. For such x, then, $c(\mathcal{P}, x) = 8/3$. Therefore, $\mathcal{R}(f)$ is at most 8/3.

How can we prove *lowerbounds* on randomized complexity? For this we need another concept.

11.3 Lowerbounds on Randomized Complexity

NEEDS CLEANUP NOW

To prove lowerbounds on randomized complexity, it suffices by Yao's Lemma (see Section 11.6) to prove lowerbounds on *distributional complexity*. Where randomized complexity explores distributions over the space of decision trees for a problem, distributional complexity considers probability distributions on inputs. It is under such considerations that we can speak of "average case analysis."

Let $\mathcal{D}$ be a probability distribution over the space of input strings of length n. Then, if A is a deterministic algorithm, such as a decision tree, for a function, then we define the distributional complexity of A on a function f with inputs distributed according to $\mathcal{D}$ as the expected cost for algorithm A to compute f, where the expectation is over the distribution of inputs.

DEFINITION 11.15
The **distributional complexity** $d(A, \mathcal{D})$ of algorithm A given inputs distributed according to $\mathcal{D}$ is defined as:

$$d(A, \mathcal{D}) = \sum_{x:input} \mathcal{D}(x) cost(A, x) = \mathbf{E}_{x \in \mathcal{D}}[cost(A, x)] \qquad (8)$$

From this we can characterize distributional complexity as a function of a single function f itself.

DEFINITION 11.16
The **distributional** decision tree complexity, $\Delta(f)$ of function f is defined as:

$$\Delta(f) = \max_{\mathcal{D}} \min_{A} d(A, \mathcal{D}) \qquad (9)$$

Where A above runs over the set of decision trees that are deciders for f.

So the distributional decision tree complexity measures the expected efficiency of the most efficient decision tree algorithm works given the worst case distribution of inputs.

The following theorem follows from Yao's lemma.

THEOREM 11.17
$\mathcal{R}(f) = \Delta(f)$.

So in order to find a lower bound on some randomized algorithm, it suffices to find a lower bound on $\Delta(f)$. Such a lower bound can be found by postulating an input distribution $\mathcal{D}$ and seeing whether every algorithm has expected cost at least equal to the desired lower bound.

EXAMPLE 11.18
We return to considering the majority function, and we seek to find a lower bound on $\Delta(f)$. Consider a distribution over inputs such that inputs in which all three bits match, namely 000 and 111, occur with probability 0. All other inputs occur with probability $1/6$. For any decision tree, that is, for any order in which the three bits are examined, there is exactly a $1/3$ probability that the first two bits examined will be the same value, and thus there is a $1/3$ probability that the cost is 2. There is then a $2/3$ probability that the cost is 3. Thus the overall expected cost for this distribution is $8/3$. This implies that $\Delta(f) \geq 8/3$ and in turn that $\mathcal{R}(f) \geq 8/3$. So $\Delta(f) = \mathcal{R}(f) = 8/3$.

DRAFT

11.4   Some techniques for decision tree lowerbounds

DEFINITION 11.19 (SENSITIVITY)
If $f : \{0,1\}^n \to \{0,1\}$ is a function and $x \in \{0,1\}^n$ then the *sensitivity of f on x*, denoted $s_x(f)$, is the number of bit positions i such that $f(x) \neq f(x^i)$, where x^i is x with its ith bit flipped. The *sensitivity* of f, denoted $s(f)$, is $\max_x \{s_x(f)\}$.

The *block sensitivity of f on x*, denoted $bs_x(f)$, is the maximum number b such that there are disjoint blocks of bit positions $B_{1,2}, \ldots, B_b$ such that $f(x) \neq f(x^{B_i})$ where x^{B_i} is x with all its bits flipped in block B_i. The *block sensitivity* of f denoted $bs(f)$ is $\max_x \{bs_x(f)\}$.

It is conjectured that there is a constant c (as low as 2) such that $bs(f) = O(s(f)^c)$ for all f but this is wide open. The following easy observation is left as an exercise.

LEMMA 11.20
For any function, $s(f) \leq bs(f) \leq D(f)$.

THEOREM 11.21 (NISAN)
$C(f) \leq s(f)bs(f)$.

PROOF: For any input $x \in \{0,1\}^n$ we describe a certificate for x of size $s(f)bs(f)$. This certificate is obtained by considering the largest number of disjoint blocks of variables $B_1, B_2, \ldots, B_b$ that achieve $b = bs_x(f) \leq bs(f)$. We claim that setting these variables according to x constitutes a certificate for x.

Suppose not, and let x' be an input that is consistent with the above certificate. Let B_{b+1} be a block of variables such that $x' = x^{B_{b+1}}$. Then B_{b+1} must be disjoint from $B_1, B_2, \ldots B_b$, which contradicts $b = bs_x(f)$.

Note that each of $B_1, B_2, \ldots, B_b$ has size at most $s(f)$ by definition of $s(f)$, and hence the size of the certificate we have exhibited is at most $s(f)bs(f)$. ∎

Recent work on decision tree lowerbounds has used *polynomial representations* of boolean functions. Recall that a multilinear polynomial is a polynomial whose degree in each variable is 1.

DEFINITION 11.22
An n-variate polynomial $p(x_1, x_2, \ldots, x_n)$ *represents* $f : \{0,1\}^n \to \{0,1\}$ if $p(x) = f(x)$ for all $x \in \{0,1\}^n$.

The *degree* of f, denoted $deg(f)$, is the degree of the multilinear polynomial that represents f.

(The exercises ask you to show that the multilinear polynomial representation is unique, so $deg(f)$ is well-defined.)

EXAMPLE 11.23
The AND of n variables $x_1, x_2, \ldots, x_n$ is represented by the multilinear polynomial $\prod_{i=1}^n x_i$ and OR is represented by $1 - \prod_{i=1}^n (1 - x_i)$.

The degree of AND and OR is n, and so is their decision tree complexity. There is a similar connection for other problems too, but it is not as tight. The first part of the next theorem is an easy exercise; the second part is nontrivial.

THEOREM 11.24
 1. $deg(f) \leq D(f)$.

 2. (Nisan-Smolensky) $D(f) \leq deg(f)^2 bs(f) \leq O(deg(f)^4)$.

11.5 Comparison trees and sorting lowerbounds

TO BE WRITTEN

11.6 Yao's MinMax Lemma

This section presents Yao's minmax lemma, which is used in a variety of settings to prove lower-bounds on randomized algorithms. Therefore we present it in a very general setting.

Let $\mathcal{X}$ be a finite set of inputs and $\mathcal{A}$ be a finite set of algorithms that solve some computational problem on these inputs. For $x \in \mathcal{X}, a \in \mathcal{A}$, we denote by $\text{cost}(A, x)$ the *cost* incurred by algorithm A on input x. A *randomized algorithm* is a probability distribution $\mathcal{R}$ on $\mathcal{A}$. The *cost* of $\mathcal{R}$ on input x, denoted $\text{cost}(\mathcal{R}, x)$, is $E_{A \in \mathcal{R}}[\text{cost}(A, x)]$. The *randomized complexity* of the problem is

$$\min_{\mathcal{R}} \max_{x \in \mathcal{X}} \text{cost}(\mathcal{R}, x). \tag{10}$$

Let $\mathcal{D}$ be a distribution on inputs. For any deterministic algorithm A, the cost incurred by it on $\mathcal{D}$, denoted $\text{cost}(A, \mathcal{D})$, is $E_{x \in \mathcal{D}}[\text{cost}(A, x)]$. The *distributional complexity* of the problem is

$$\max_{\mathcal{D}} \min_{A \in \mathcal{A}} \text{cost}(A, \mathcal{D}). \tag{11}$$

Yao's Lemma says that these two quantitities are the same. It is easily derived from von Neumann's minmax theorem for zero-sum games, or with a little more work, from linear programming duality.

Yao's lemma is typically used to lowerbound randomized complexity. To do so, one defines (using some insight and some luck) a suitable distribution $\mathcal{D}$ on the inputs. Then one proves that *every* deterministic algorithm incurs high cost, say C, on this distribution. By Yao's Lemma, it follows that the randomized complexity then is at least C.

Exercises

§1 Suppose f is any function that depends on all its bits; in other words, for each bit position i there is an input x such that $f(x) \neq f(x^i)$. Show that $s(f) = \Omega(\log n)$.

§2 Consider an f defined as follows. The n-bit input is partitioned into $\lfloor \sqrt{n} \rfloor$ blocks of size about $\sqrt{n}$. The function is 1 iff there is at least one block in which two consecutive bits are 1 and the remaining bits in the block are 0. Estimate $s(f), bs(f), C(f), D(f)$ for this function.

§3 Show that there is a unique multilinear polynomial that represents $f : \{0,1\}^n \to \{0,1\}$. Use this fact to find the multilinear representation of the PARITY of n variables.

§4 Show that $deg(f) \leq D(f)$.

Chapter notes and history

The result that the decision tree complexity of connectivity and many other problems is $\binom{n}{2}$ has motivated the following conjecture (atributed variously to Anderaa, Karp, Yao):

Every monotone graph property has $D(\cdot) = \binom{n}{2}$.

Here "monotone" means that adding edges to the graph cannot make it go from having the property to not having the property (e.g., connectivity). "Graph property" means that the property does not depend upon the vertex indices (e.g., the property that vertex 1 and vertex 2 have an edge between them). This conjecture is known to be true up to a $O(1)$ factor; the proof uses topology and is excellently described in Du and Ko [DK00]. A more ambitious conjecture is that even the randomized decision tree complexity of monotone graph properties is $\Omega(n^2)$ but here the best lowerbound is close to $n^{4/3}$.

The polynomial method for decision tree lowerbounds is surveyed in Buhrman and de Wolf [BdW02]. The method can be used to lowerbound randomized decision tree complexity (and more recently, *quantum* decision tree complexity) but then one needs to consider polynomials that *approximately* represent the function.

Chapter 12

Communication Complexity

Communication complexity concerns the following scenario. There are two players with unlimited computational power, each of whom holds an n bit input, say x and y. Neither knows the other's input, and they wish to collaboratively compute $f(x, y)$ where function $f : \{0, 1\}^n \times \{0, 1\}^n \to \{0, 1\}$ is known to both. Furthermore, they had foreseen this situation (e.g., one of the parties could be a spacecraft and the other could be the base station on earth), so they had already —before they knew their inputs x, y— agreed upon a protocol for communication[1]. The *cost* of this protocol is the *number of bits communicated* by the players for the *worst-case* choice of x, y.

Researchers have studied many modifications of the above basic scenario, including randomized protocols, nondeterministic protocols, average-case protocols (where x, y are assumed to come from a distribution), multiparty protocols, etc. Truly, this is a self-contained mini-world within complexity theory. Furthermore, lowerbounds on communication complexity have uses in a variety of areas, including lowerbounds for parallel and VLSI computation, circuit lowerbounds, polyhedral theory, data structure lowerbounds, etc. We give a very rudimentary introduction to this area; an excellent and detailed treatment can be found in the book by Kushilevitz and Nisan [KN97].

12.1 Definition

Now we formalize the informal description of communication complexity given above.

A t-round *communication protocol* for f is a sequence of function pairs $(S_1, C_1), (S_2, C_2), \ldots, (S_t, C_t), (f_1, f_2)$. The input of S_i is the communication pattern of the first $i - 1$ rounds and the output is from $\{1, 2\}$, indicating which player will communicate in the ith round. The input of C_i is the input string of this selected player as well as the communication pattern of the first $i - 1$ rounds. The output of C_i is the bit that this player will communicate in the ith round. Finally, f_1, f_2 are 0/1-valued functions that the players apply at the end of the protocol to their inputs as well as the communication pattern in the t rounds in order to compute the output. These two outputs must be $f(x, y)$. The

---

[1] Do not confuse this situation with *information theory*, where an algorithm is given messages that have to be transmitted over a noisy channel, and the goal is to transmit them robustly while minimizing the amount of communication. In communication complexity the channel is not noisy and the players determine what messages to send.

communication complexity of f is

$$C(f) = \min_{\text{protocols } \mathcal{P}} \max_{x,y} \quad \{\text{Number of bits exchanged by } \mathcal{P} \text{ on } x, y.\}$$

Notice, $C(f) \leq n+1$ since the trivial protocol is for one player to communicate his entire input, whereupon the second player computes $f(x, y)$ and communicates that single bit to the first. Can they manage with less communication?

EXAMPLE 12.1 (PARITY)
Suppose the function $f(x, y)$ is the *parity* of all the bits in x, y. We claim that $C(f) = 2$. Clearly, $C(f) \geq 2$ since the function depends nontrivially on each input, so each player must transmit at least one bit. Next, $C(f) \leq 2$ since it suffices for each player to transmit the parity of all the bits in his possession; then both know the parity of all the bits.

REMARK 12.2
Sometimes students ask whether a player can communicate by not saying anything? (After all, they have three options: send a 0, or 1, or not say anything in that round.) We can regard such protocols as communicating with a ternary, not binary, alphabet, and analyze them analogously.

12.2 Lowerbound methods

Now we discuss methods for proving lowerbounds on communication complexity. As a running example in this chapter, we will use the equality function:

$$\text{EQ}(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

We will see that $C(EQ) \geq n$.

12.2.1 Fooling set

We show $C(EQ) \geq n$. For contradiction's sake, suppose a protocol exists whose complexity is at most $n - 1$. Then there are only 2^{n-1} communication patterns possible between the players. Consider the set of all 2^n pairs (x, x). Using the pigeonhole principle we conclude there exist two pairs (x, x) and (x', x') on which the communication pattern is the same. Of course, thus far we have nothing to object to, since the answers $\text{EQ}(x, x)$ and $\text{EQ}(x', x')$ on both pairs are 1. However, now imagine giving one player x and the other player x' as inputs. A moment's thought shows that the communication pattern will be the same as the one on (x, x) and (x', x'). (Formally, this can be shown by induction. If player 1 communicates a bit in the first round, then clearly this bit is the same whether his input is x or x'. If player 2 communicates in the 2nd round, then his bit must also be the same on both inputs since he receives the same bit from player 1. And so on.)

Hence the player's answer on (x, x) must agree with their answer on (x, x'). But then the protocol must be incorrect, since $\mathrm{EQ}(x, x') = 0 \neq \mathrm{EQ}(x, x)$.

The lowerbound argument above is called a *fooling set* argument. It is formalized as follows.

DEFINITION 12.3
A *fooling set* for $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ is a set $S \subseteq \{0,1\}^n \times \{0,1\}^n$ and a value $b \in \{0,1\}$ such that:

1. For every $(x, y) \in S$, $f(x, y) = b$.

2. For every two distinct pairs $(x_1, y_1), (x_2, y_2) \in S$, either $f(x_1, y_2) \neq b$ or $f(x_2, y_1) \neq b$.

LEMMA 12.4
If f has a fooling set with m pairs then $C(f) \geq \log_2 m$.

EXAMPLE 12.5 (DISJOINTNESS)
Let x, y be interpreted as characteristic vectors of subsets of $\{1, 2, \ldots, n\}$. Let $\mathrm{DISJ}(x, y) = 1$ if these two subsets are disjoint, otherwise $\mathrm{DISJ}(x, y) = 0$. Then $C(\mathrm{DISJ}) \geq n$ since the following 2^n pairs constitute a fooling set:

$$S = \left\{ (A, \overline{A}) : A \subseteq \{1, 2, \ldots, n\} \right\}.$$

12.2.2 The tiling lowerbound

The tiling lowerbound takes a more global view of f. Consider the matrix of f, denoted $M(f)$, which is a $2^n \times 2^n$ matrix whose (x, y)'th entry is $f(x, y)$. See Figure 12.1. We visualize the

Figure unavailable in pdf file.

Figure 12.1: Matrix M(f) for the equality function when the inputs to the players have 3 bits. The numbers in the matrix are values of f.

communication protocol in terms of this matrix. A *combinatorial rectangle* (or just rectangle) in the matrix is a submatrix corresponding to $A \times B$ where $A \subseteq \{0,1\}^n$, $B \subseteq \{0,1\}^n$. If the protocol begins with the first player sending a bit, then $M(f)$ partitions into two rectangles of the type $A_0 \times \{0,1\}^n$, $A_1 \times B^n$, where A_b is the subset of strings for which the first player communicates bit b. Notice, $A_0 \cup A_1 = \{0,1\}^n$. If the next bit is sent by the second player, then each of the two rectangles above is further partitioned into two smaller rectangles depending upon what this bit was. If the protocol continues for k steps, the matrix gets partitioned into 2^k rectangles. Note that each rectangle in the partition corresponds to a subset of input pairs for which the communication sequence thus far has been identical. (See Figure 12.2 for an example.)

Figure unavailable in pdf file.

Figure 12.2: Two-way communication matrix after two steps. The large number labels are the concatenation of the bit sent by the first player with the bit sent by the second player.

If the protocol stops, then the value of f is determined within each rectangle, and thus must be the same for all pairs x, y in that rectangle. Thus the set of all communication patterns must lead to a partition of the matrix into *monochromatic* rectangles. (A rectangle $A \times B$ is *monochromatic* if for all x in A and y in B, $f(x, y)$ is the same.)

DEFINITION 12.6
A monochromatic tiling of $M(f)$ is a partition of $M(f)$ into disjoint monochromatic rectangles. We denote by $\chi(f)$ the minimum number of rectangles in any monochromatic tiling of $M(f)$.

The following theorem is immediate from our discussion above.

THEOREM 12.7
If f has communication complexity C then it has a monochromatic tiling with at most 2^C rectangles. Consequently, $C \geq \log_2 \chi(f)$.

The following observation shows that the tiling bound subsumes the fooling set bound.

LEMMA 12.8
If f has a fooling set with m pairs, then $\chi(f) \geq m$.

PROOF: If (x_1, y_1) and (x_2, y_2) are two of the pairs in the fooling set, then they cannot be in a monochromatic rectangle since not all of $(x_1, y_1), (x_2, y_2), (x_1, y_2), (x_2, y_1)$ have the same f value. ∎

12.2.3   Rank lowerbound

Now we introduce an algebraic method to lowerbound $\chi(f)$ (and hence communication complexity). Recall the high school notion of *rank* of a square matrix: it is the size of the largest subset of rows/colums that are independent. The following is another definition.

DEFINITION 12.9
If a matrix has entries from a field F then the *rank* of an $n \times n$ matrix M is the minimum value of l such that M can be expressed as

$$M = \sum_{i=1}^{l} \alpha_i B_i,$$

where $\alpha_i \in F \setminus \{0\}$ and each B_i is an $n \times n$ matrix of rank 1.

Note that $0, 1$ are elements of every field, so we can compute the rank over any field we like. The choice of field can be crucial; see Problem 5 in the exercises.

The following theorem is trivial, since each monochromatic rectangle can be viewed (by filling out entries outside the rectangle with 0's) as a matrix of rank at most 1 .

THEOREM 12.10
For every function f, $\chi(f) \geq \operatorname{rank}(M(f))$.

12.2.4 Discrepancy

The *discrepancy* of a rectangle $A \times B$ in $M(f)$ is

$$\frac{1}{2^{2n}} \left| \text{number of 1's in } A \times B - \text{number of 0's in } A \times B \right|. \tag{1}$$

The *discrepancy* of the matrix $M(f)$, denote $\text{Disc}(f)$, is the largest discrepancy among all rectangles. The following Lemma relates it to $\chi(f)$.

LEMMA 12.11

$$\chi(f) \geq \frac{1}{Disc(f)}.$$

PROOF: For a monochromatic rectangle, the discrepancy is its size divided by 2^{2n}. The total number of entries in the matrix is 2^{2n}. The bound follows. ∎

EXAMPLE 12.12
Lemma 12.11 can be very loose. For the $EQ()$ function, the discrepancy is at least $1 - 2^{-n}$ (namely, the discrepancy of the entire matrix), which would only give a lowerbound of 2 for $\chi(f)$. However, $\chi(f)$ is at least 2^n, as already noted.

Now we describe a method to upperbound the discrepancy using *eigenvalues*.

LEMMA 12.13 (EIGENVALUE BOUND)
For any matrix M, the discrepancy of a rectangle $A \times B$ is at most $\lambda_{max}(M)\sqrt{|A|\,|B|}/2^{2n}$, where $\lambda_{max}(M)$ is the magnitude of the largest eigenvalue of M.

PROOF: Let $1_A, 1_B \in \mathbb{R}^n$ denote the characteristic vectors of A, B. Then $|1_A|_2 = \sqrt{\sum_{i \in A} 1^2} = \sqrt{|A|}$.
   The discrepancy of the rectangle $A \times B$ is

$$\frac{1}{2^{2n}} 1_A^T M 1_B \leq \frac{1}{2^{2n}} \lambda_{max}(M) \left| 1_A^T 1_B \right| \leq \frac{1}{2^{2n}} \lambda_{max}(M) \sqrt{|A|\,|B|}.$$

explain this.

∎

EXAMPLE 12.14
The *mod 2 inner product* function defined as $f(x,y) = (x \cdot y)_2 = \sum_i x_i y_i (\text{mod } 2)$ has been encountered a few times in this book. To bound its discrepancy, we consider the matrix $2M(f) - 1$. This transformation makes the range of the function $\{-1, 1\}$ and will be useful again later. Let this new

matrix be denoted N. It is easily checked that every two distinct rows (columns) of N are orthogonal, every row has ℓ_2 norm $2^{n/2}$, and that $N^T = N$. Thus we conclude that $N^2 = 2^n I$ where I is the unit matrix. Hence every eigenvalue is either $+2^{n/2}$ or $-2^{n/2}$, and thus Lemma 12.13 implies that the discrepancy of a rectangle $A \times B$ is at most $2^{n/2}\sqrt{|A|\,|B|}$ and the overall discrepancy is at most $2^{3n/2}$ (since $|A|,|B| \leq 2^n$).

A technique for upperbounding the discrepancy

Now we describe an upperbound technique for the discrepancy that will later be useful in the multiparty setting (Section 12.3). For ease of notation, in this section we change the range of f to $\{-1,1\}$ by replacing 1's in $M(f)$ with -1's and replacing 0's with 1's. Note that now

$$\text{Disc}(f) = \max_{A,B} \frac{1}{2^{2n}} \left| \sum_{a\in A, b\in B} f(a,b) \right|.$$

DEFINITION 12.15
$\mathcal{E}(f) = \mathbf{E}_{a_1,a_2,b_1,b_2} \left[\prod_{i=1,2} \prod_{j=1,2} f(a_i, b_j) \right].$

Note that $\mathcal{E}(f)$ can be computed, like the rank, in polynomial time given the $M(f)$ as input.

LEMMA 12.16

$$Disc(f) \leq \mathcal{E}(f)^{1/4}.$$

PROOF: The proof follows in two steps.

CLAIM 1: *For every function* $h:\{0,1\}^n \times \{0,1\}^n \to \{1,-1\}$, $\mathcal{E}(h) \geq (\mathbf{E}_{a,b}[f(a,b)])^4$.
    We will use the Cauchy-Schwartz inequality, specifically, the version according to which $\mathbf{E}[z^2] \geq (\mathbf{E}[z])^2$ for every random variable z.

$$\mathcal{E}(h) = \mathbf{E}_{a_1,a_2} \left[\mathbf{E}_{b_1,b_2} \left[\prod_{i=1,2} \prod_{j=1,2} h(a_i, b_j) \right] \right] \tag{2}$$

$$= \mathbf{E}_{a_1,a_2} \left[(\mathbf{E}_b[h(a_1,b)h(a_2,b)])^2 \right] \tag{3}$$

$$\geq (\mathbf{E}_{a_1,a_2} [\mathbf{E}_b[h(a_1,b)h(a_2,b)]])^2 \qquad \text{(Cauchy Schwartz)} \tag{4}$$

$$\geq (\mathbf{E}_{a,b}[h(a,b)])^4. \qquad \text{(repeating prev. two steps)} \tag{5}$$

CLAIM 2: *For every function* f *there is a function* h *such that* $\mathcal{E}(f) = \mathcal{E}(h)$ *and* $\mathbf{E}_{a,b}[h(a,b)] \geq Disc(f)$.

First, we note that for every two functions $g_1, g_2 : \{0,1\}^n \to \{-1,1\}$, if we define $h = f \circ g_1 \circ g_2$ as

$$h(a,b) = f(a,b)g_1(a)g_2(b)$$

then $\mathcal{E}(f) = \mathcal{E}(h)$. The reason is that for all a_1, a_2, b_1, b_2,

$$\prod_{i=1,2} \prod_{j=1,1} h(a_i, b_j) = g_1(a_1)^2 g_1(a_2)^2 g_2(b_1)^2 g_2(b_2)^2 \prod_{i=1,2} \prod_{j=1,2} f(a_i, b_j)$$

and the square of any value of g_1, g_2 is 1.

Now we prove Claim 2 using the probabilistic method. Define two random functions $g_1, g_2 : \{0,1\}^n \to \{-1,1\}$ as follows:

$$g_1(a) = \begin{cases} 1 & \text{if } a \in A \\ r_a & r_a \in \{-1,1\} \text{ is randomly chosen} \end{cases}$$

$$g_2(b) = \begin{cases} 1 & \text{if } b \in B \\ s_b & s_b \in \{-1,1\} \text{ is randomly chosen} \end{cases}$$

Let $h = f \circ g_1 \circ g_2$, and therefore $\mathcal{E}(h) = \mathcal{E}(f)$. Furthermore

$$\mathbf{E}_{g_1,g_2}\left[\mathbf{E}_{a,b}[h(a,b)]\right] = \mathbf{E}_{a,b}\left[\mathbf{E}_{g_1,g_2}[f(a,b)g_1(a)g_2(b)]\right] \tag{6}$$

$$= \frac{1}{2^{2n}} \sum_{a \in A, b \in B} f(a,b) \tag{7}$$

$$= \text{Disc}(f) \tag{8}$$

where the second line follows from the fact that $\mathbf{E}_{g_1}[g_1(a)] = \mathbf{E}_{g_2}[g_2(b)] = 0$ for $a \notin A$ and $b \notin B$.

Thus in particular there *exist* g_1, g_2 such that $|\mathbf{E}_{a,b}[h(a,b)]| \geq \text{Disc}(f)$. ∎

12.2.5   Comparison of the lowerbound methods

As already noted, discrepancy upperbounds imply lowerbounds on $\chi(f)$. Of the other three methods, the tiling argument is the strongest, since it subsumes the other two. The rank method is the weakest, since the rank lowerbound always implies a tiling lowerbound and a fooling set lowerbound (the latter follows from Problem 3 in the exercises).

Also, we can separate the power of these lowerbound arguments. For instance, we know functions for which there is a significant gap between $\log \chi(f)$ and $\log \text{rank}(M(f))$. However, the following conjecture (we only state one form of it) says that all three methods (except discrepancy, which as already noted can be arbitrarily far from $\chi(f)$) give the same bound up to a polynomial factor.

CONJECTURE 12.17 (LOG RANK CONJECTURE)
There is a constant $c > 1$ such that $C(f) = O(\log(rank(M(f)))^c)$ for all f and all input sizes n.

12.3 Multiparty communication complexity

There is more than one way to generalize communication complexity to a multiplayer setting. The most interesting model is the "number on the forehead" model often encountered in math puzzles that involve people in a room, each person having a bit on their head which everybody else can see but they cannot. More formally, there is some function $f : (\{0,1\}^n)^k \to \{0,1\}$, and the input is $(x_1, x_2, \ldots, x_k)$ where each $x_i \in \{0,1\}^n$. The ith player can see all the x_j such that $j \neq i$. As in the 2-player case, the k players have an agreed-upon protocol for communication, and all this communication is posted on a "public blackboard". At the end of the protocol all parties must know $f(x_1, \ldots, x_k)$.

EXAMPLE 12.18
Consider computing the function

$$f(x_1, x_2, x_3) = \bigoplus_{i=1}^{n} \mathrm{maj}(x_{1i}, x_{2i}, x_{3i})$$

in the 3-party model where x_1, x_2, x_3 are n bit strings. The communication complexity of this function is 3: each player counts the number of i's such that she can determine the majority of x_{1i}, x_{2i}, x_{3i} by examining the bits available to her. She writes the parity of this number on the blackboard, and the final answer is the parity of the players' bits. This protocol is correct because the majority for each row is known by either 1 or 3 players, and both are odd numbers.

EXAMPLE 12.19 (GENERALIZED INNER PRODUCT)
The *generalized inner product function* $GIP_{k,n}$ maps nk bits to 1 bit as follows

$$f(x_1, \ldots, x_k) = \bigoplus_{i=1}^{n} \bigwedge_{j=1}^{k} x_{ij}. \tag{9}$$

Notice, for $k = 2$ this reduces to the mod 2 inner product of Example 12.14.

In the 2-party model we introduced the notion of a monochromatic rectangle in order to prove lower bounds. For the k-party case we will use cylinder intersections. A *cylinder in dimension i* is a subset S of the inputs such that if $(x_1, \ldots, x_k) \in S$ then for all x_i' we have that $(x_1, \ldots, x_{i-1}, x_i', x_{i+1}, \ldots, x_k) \in S$ also. A *cylinder intersection* is $\cap_{i=1}^{k} T_i$ where T_i is a cylinder in dimension i.

As noted in the 2-party case, a communication protocol can be viewed as a way of partitioning the matrix $M(f)$. Here $M(f)$ is a k-dimensional cube, and player i's communication does not depend upon x_i. Thus we conclude that if f has a multiparty protocol that communicates c bits, then its matrix has a tiling using at most 2^c monochromatic cylinder intersections.

LEMMA 12.20
If every partition of $M(f)$ into monochromatic cylinder intersections requires at least R cylinder intersections, then the k-party communication complexity isat least $\log_2 R$.

Discrepancy-based lowerbound

In this section, we will assume as in our earlier discussion of discrepancy that the range of the function f is $\{-1, 1\}$. We define the *k-party discrepancy* of f by analogy to the 2-party case

$$\text{Disc}(f) = \frac{1}{2^{nk}} \max_T \left| \sum_{(a_1, a_2, \ldots, a_k) \in T} f(a_1, a_2, \ldots, a_k) \right|,$$

where T ranges over all cylinder intersections.

To upperbound the discrepancy we introduce the k-party analogue of $\mathcal{E}(f)$. Let a *cube* be a set D in $\{0, 1\}^{nk}$ of 2^k points of the form $\{a_{1,1}, a_{2,1}\} \times \{a_{1,2}, a_{2,2}\} \times \cdots \times \{a_{1,k}, a_{2,k}\}$, where each $a_{i,j} \in \{0, 1\}^n$.

$$\mathcal{E}(f) = E_D \left[\prod_{\overline{a} \in D} f(\overline{a}) \right].$$

Notice that the definition of $\mathcal{E}()$ for the 2-party case is recovered when $k = 2$. The next lemma is also an easy generalization.

LEMMA 12.21

$$Disc(f) \leq (\mathcal{E}(f))^{1/2^k}.$$

PROOF: The proof is analogous to Lemma 12.16 and left as an exercise. The only difference is that instead of defining 2 random functions we need to define k random functions $g_1, g_2, g_k \colon \{0, 1\}^{nk} \to \{-1, 1\}$, where g_i depends on every one of the k coordinates except the ith. ∎

Now we can prove a lowerbound for the Generalized Inner Product function. Note that since we changed the range to $\{-1, 1\}$ it is now defined as

$$GIP_{k,n}(x_1, x_2, \ldots, x_k) = (-1)^{\sum_{i \leq n} \prod_{j \leq k} x_{ij}(mod 2)}. \tag{10}$$

THEOREM 12.22
The function $GIP_{k,n}$ has k-party communication complexity $\Omega(n/8^k)$ as n grows larger.

PROOF: We use induction on k. For $k \geq 1$ let β_k be defined using $\beta_1 = 0$ and $\beta_{k+1} = \frac{1+\beta_k}{2}$. We claim that

$$\mathcal{E}(GIP_{k,n}) \leq \beta_k^n.$$

Assuming truth for $k - 1$ we prove for k. A random cube D in $\{0,1\}^{nk}$ is picked by picking $a_{11}, a_{21} \in \{0,1\}^n$ and then picking a random cube D' in $\{0,1\}^{(k-1)n}$.

$$\mathcal{E}(GIP_{k,n}) = \mathbf{E}_{a_{11},a_{21}} \left[\mathbf{E}_{D'} \left[\prod_{\overline{a} \in \{a_{11},a_{21}\} \times D'} GIP_{k,n}(\overline{a}) \right] \right] \tag{11}$$

The proof proceeds by considering the number of coordinates where strings a_{11} and a_{21} are identical. Examining the expression for $GIP_{k,n}$ in (10) we see that these coordinates contribute nothing once we multiply all the terms in the cube, since their contributions get squared and thus become 1. The coordinates that contribute are

TO BE COMPLETED ■

12.4   Probabilistic Communication Complexity

Will define the model, give the protocol for EQ, and describe the discrepancy-based lowerbound.

12.5   Overview of other communication models

We outline some of the alternative settings in which communication complexity has been studied.

Nondeterministic protocols: These are defined by analogy to **NP**. In a nondeterministic protocol, the players are both provided an additional third input z ("nondeterministic guess"). Apart from this guess, the protocol is deterministic. The *cost* incurred on x, y is

$$\min_z \quad \{|z| + \text{number of bits exchanged by protocol when guess is } z\}.$$

The *nondeterministic communication complexity* of f is the minimum k such that there is a nondeterministic protocol whose cost for all input pairs is at most k.

In general, one can consider communication protocols analogous to **NP**, **coNP**, **PH** etc.

Randomized protocols: These are defined by analogy to **RP**, **BPP**. The players are provided with an additional input r that is chosen uniformly at random from m-bit strings for some m. Randomization can significantly reduce the need for communication. For instance we can use fingerprinting with random primes (explored in Chapter 7), to compute the equality function by exchanging $O(\log n)$ bits: the players just pick a random prime p of $O(\log n)$ bits and exchange $x \pmod p$ and $y \pmod p$.

Average case protocols: Just as we can study average-case complexity in the Turing machine model, we can study communication complexity when the inputs are chosen from a distribution $\mathcal{D}$. This is defined as

$$C_{\mathcal{D}}(f) = \min_{\text{protocols } \mathcal{P}} \sum_{x,y} \Pr[(x,y) \in \mathcal{D}] \times \{\text{Number of bits exchanged by } \mathcal{P} \text{ on } x, y.\}$$

DRAFT

Computing a non boolean function: Here the function's output is not just $\{0,1\}$ but an m-bit number for some m. We discuss one example in the exercises.

Asymmetric communication: The "cost" of communication is asymmetric: there is some B such that it costs the first player B times as much to transmit a bit than it does the second player. The goal is to minimize the total cost.

Multiparty settings: The most obvious generalization to multiparty settings is whereby f has k arguments $x_1, x_2, \ldots, x_k$ and player i gets x_i. At the end all players must know $f(x_1, x_2, \ldots, x_k)$. This is not as interesting as the so-called "number of the forehead" where player i can see all of the input except for x_i. We discuss it in Section **??** together with some applications.

Computing a relation: There is a relation $R \subseteq \{0,1\}^n \times \{0,1\}^n \times \{1, 2, \ldots, m\}$ and given $x, y \in B^n$ the players seek to agree on any $b \in \{1, 2, \ldots, m\}$ such that $(x, y, b) \in R$. See section **??**.

These and many other settings are discussed in [KN97].

12.6 Applications of communication complexity

We briefly discussed parallel computation in Chapter 6. Yao [Yao79] invented communication complexity as a way to lowerbound the running time of parallel computers for certain tasks. The idea is that the input is distributed among many processors, and if we partition these processors into two halves, we may lowerbound the computation time by considering the amount of communication that must necessarily happen between the two halves. A similar idea is used to prove time/space lowerbounds for VLSI circuits. For instance, in a VLSI chip that is an $m \times m$ grid, if the communication complexity for a function is greater than c, then the time required to compute it is at least c/m.

Communication complexity is also useful in time-space lowerbounds for Turing machines (see Problem 1 in exercises), and circuit lowerbounds (see Chapter 13).

Data structures such as heaps, sorted arrays, lists etc. are basic objects in algorithm design. Often, algorithm designers wish to determine if the data structure they have designed is the best possible. Communication complexity lowerbounds can be used to establish such results. See [KN97].

Yannakakis [Yan91] has shown how to use communication complexity lowerbounds to prove lowerbounds on the size of polytopes representing **NP**-complete problems. Solving the open problem mentioned in Problem 8 in the exercises would prove a lowerbound for the polytope representing vertex cover.

Exercises

§1 If $S(n) \leq n$, show that a space $S(n)$ TM takes at least $\Omega(n/S(n))$ steps to decide the language $\{x\#x : x \in \{0,1\}^*\}$.

§2 Show that the high school definition of rank (the size of the largest set of independent rows or columns) is equivalent to that in Definition 12.9.

§3 Give a fooling set argument that proves that $C(f) \geq \lceil \log \text{rank}(M(f)) \rceil$.

§4 Show that $C(f)\text{rank}(M(f) + 1$.

§5 Consider x, y as vectors over $GF(2)^n$ and let $f(x, y)$ be their inner product mod 2. Prove that the communication complexity is n.

<div style="text-align:center">the all-1 matrix.

Hint: Lowerbound the rank of the matrix $2M(f) - J$ where J is</div>

What field should you use to compute the rank? Does it matter?

§6 Let $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ be such that all rows of $M(f)$ are distinct. Show that $C(f) \geq \log n$.

<div style="text-align:center">**Hint:** Lowerbound the rank.</div>

§7 (Aho, Ullman, Yannakakis) Show that $C(f) = O(\log^2 \chi(f))$.

<div style="text-align:center">in each phase $O(\log \chi(f))$ bits get communicated.

their input-pair lies in. The protocol has $O(\log \chi(f))$ phases, and

Hint: The players try to determine which of the $|\chi(f)|$ rectangles</div>

§8 For any graph G with n vertices, consider the following communication problem: Player 1 receives a clique C in G, and Player 2 receives an independent set I. They have to communicate in order to determine $|C \cap I|$. (Note that this number is either 0 or 1.) Prove an $O(\log^2 n)$ upperbound on the communication complexity.

Can you improve your upperbound or prove a lower bound better than $\Omega(\log n)$? (Open question)

§9 Prove Lemma 12.21 using the hint given there.

§10 (Karchmer-Wigderson) Consider the following problem about computing a relation. Associate the following communication problem with any function $f : \{0,1\}^n \to \{0,1\}$. Player 1 gets any input x such that $f(x) = 0$ and player 2 gets any input y such that $f(y) = 1$. They have to communicate in order to determine a bit position i such that $x_i \neq y_i$.

Show that the communication complexity of this problem is *exactly* the minixmum depth of any circuit that computes f. (The maximum fanin of each gate is 2.)

§11 Use the previous question to show that computing the parity of n bits requires depth at least $2 \log n$.

§12 Show that the following computational problem is in **EXP**: given the matrix $M(f)$ of a boolean function, and a number K, decide if $C(f) \leq K$.

(Open since Yao [Yao79]) Can you show this problem is complete for some complexity class?

Chapter notes and history

Communication complexity was first defined by Yao [Yao79]. Other early papers that founded the field were Papadimitriou and Sipser [PS84], Mehlhorn and Schmidt [MS82] (who introduced the rank lowerbound) and Aho, Ullman and Yannakakis [AUY83].

The original log rank conjecture was that $C(f) = O(\text{rank}(M(f)))$ but this was disproved by Raz and Spieker [RS95].

The book by Nisan and Kushilevitz [KN97] is highly recommended.

DRAFT

Chapter 13

Circuit lowerbounds

Complexity theory's Waterloo

We believe that **NP** does not have polynomial-sized circuits. We've seen that if true, this implies that $\mathbf{NP} \neq \mathbf{P}$. In the 1970s and 1980s, many researchers came to believe that the route to resolving **P** versus **NP** should go via circuit lowerbounds, since circuits seem easier to reason about than Turing machines. The success in this endeavor was mixed.

Progress on general circuits has been almost nonexistent: a lowerbound of n is trivial for any function that depends on all its input bits. We are unable to prove even a superlinear circuit lowerbound for any **NP** problem— the best we can do after years of effort is $4.5n - o(n)$.

To make life (comparatively) easier, researchers focussed on restricted circuit classes, and were successful in proving some decent lowerbounds. We prove some of the major results of this area and indicate where researchers are currently stuck. In Chapter 22 we'll explain some of the inherent obstacles that need to be overcome to make further progress.

13.1 AC^0 and Håstad's Switching Lemma

As we saw in Chapter 6, **AC^0** is the class of languages computable by circuit families of constant depth, polynomial size, and whose gates have unbounded fanin. (Constant depth circuits with fanin 2 can only compute functions depending on a constant number of input bits.) The burning question in the late 1970s was whether problems like Clique and TSP have **AC^0** circuits. However, in 1981, Furst, Saxe and Sipser and independently, Ajtai, proved a lowerbound for a much simpler function:

THEOREM 13.1 ([**?**, **?**])
Let $\bigoplus$ be the parity function. That is, for every $x \in \{0,1\}^n$, $\bigoplus(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i \pmod{2}$. Then $\bigoplus \notin \mathbf{AC^0}$.

Often courses in digital logic design teach students how to do "circuit minimization" using Karnaugh maps. Note that circuits talked about in those courses are depth 2 circuits, i.e. CNF or DNF. Indeed, it is easy to show (using for example the Karnaugh map technique studied in logic

design) that the parity function requires exponentially many gates if the depth is two. However, those simple ideas do not seem to generalize to even depth 3 circuits.

The main tool in the proof of Theorem 13.1 is the concept of *random restrictions*. Let f be a function computable by a depth d circuit and suppose that we choose at random a vast majority (i.e., $n - n^\epsilon$ for some constant $\epsilon > 0$ depending on d) of the input variables and assign to each such variable either 0 or 1 at random. We'll prove that with positive probability, the function f subject to this restriction is *constant* (i.e., either always zero or always one). Since the parity function cannot be made a constant by fixing values to a subset of the variables, it follows that it cannot be computed by a constant depth circuit.

13.1.1   The switching lemma

Now we prove the main lemma about how a circuit simplifies under a random restriction. A k-DNF (resp. k-CNF) formula is an OR of AND's (resp. AND or OR's) where each AND (resp. OR) involves at most k variables.

LEMMA 13.2 (HÅSTAD'S SWITCHING LEMMA [HAS86])
Suppose f is expressible as a k-DNF, and let ρ denote a random restriction that assigns random values to t randomly selected input bits. Then for every $s \geq 2$.

$$\mathbf{Pr}_\rho[f|_\rho \text{ is not expressible as } s\text{-CNF}\,] \leq \left(\frac{(n-t)k^{10}}{n}\right)^{s/2} \tag{1}$$

where $f|_\rho$ denotes the function f restricted to the partial assignment ρ.

We'll typically use this lemma with k, s constant and $t \approx n - \sqrt{n}$ in which case the guaranteed bound on the probability will be n^{-c} for some constant c. Note that by applying the lemma to the function $\neg f$, we can get the same result with the terms DNF and CNF interchanged.

Proving Theorem 13.1 from Lemma 13.2.   Now we show how Håstad's lemma implies that parity is not in **AC**^{0}. We start with any **AC**^{0} circuit and assume that the circuit has been simplified as follows (the simplifications are straightforward to do and are left as Exercises 1 and 2): **(a)** All fanouts are 1; the circuit is a tree **(b)** All *not* gates to the input level of the circuit; equivalently, the circuit has $2n$ input wires, with the last n of them being the negations of the first n **(c)** $\vee$ and $\wedge$ gates alternate —at worst this assumption doubles the depth of the circuit **(d)** The bottom level has $\wedge$ gates of fanin 1.

We randomly restrict more and more variables, where each step with high probability will reduce the depth of the circuit by 1 and will keep the bottom level at a constant fanin. Specifically, letting n_i stand for the number of unrestricted variables after step i, we restrict $n_i - \sqrt{n_i}$ variables at step $i + 1$. Since $n_0 = n$, we have $n_i = n^{1/2^i}$. Let n^b denote an upper bound on the number of gates in the circuit and let $k_i = 10b2^i$. We'll show that with high probability, after the ith restriction we're left with a depth-$d - i$ circuit with at most k^i fanin in the bottom level. Indeed, suppose that the bottom level contains $\wedge$ gates and the level above it contains $\vee$ gates. The function each such $\vee$ gate computes is a k_i-DNF and hence by Lemma 13.2, with probability $1 - \left(\frac{k_i^{10}}{n^{1/2^{i+1}}}\right)^{k_{i+1}/2}$, which

is at least $1 - 1/(10n^b)$ for large enough n, the function such a gate computes will be expressible as a k_{i+1}-CNF. We can then merge this CNF with the $\wedge$-gate above it, reducing the depth of the circuit by one (see Figures 13.1 and 13.2). The symmetric reasoning applies in the case the bottom level consists of $\vee$ gates— in this case we use the lemma to transform the k_i-CNF of the level above it into a k_{i+1}-DNF. Note that we apply the lemma at most once per each of the at most n^b gates of the original circuit. By the union bound, with probability $9/10$, if we continue this process for $d - 2$ steps, we'll get a depth two circuit with fanin $k = k_{d-2}$ at bottom level (i.e., a k-CNF or k-DNF formula). If we then choose to restrict each variable with probability half (i.e., restrict about half of the variables to a random value), this circuit will be reduced to a constant function with probability at least 2^{-k}. Since the parity function is not constant under any restriction of less than n variables, this proves Theorem 13.1. ∎

Figure unavailable in pdf file.

Figure 13.1: Circuit before Håstad switching transformation.

Figure unavailable in pdf file.

Figure 13.2: Circuit after Håstad switching transformation. Notice that the new layer of $\wedge$ gates can be collapsed with the single $\wedge$ parent gate, to reduce the number of levels by one.

13.1.2   Proof of the switching lemma (Lemma 13.2)

Now we prove the Switching Lemma. The original proof was more complicated; this one is due to Razborov. Let f be expressible as a k-DNF on n variables. Let t be as in the lemma and let $\mathcal{R}_t$ denote the set of all restrictions to t variables (note we can assume $t > n/2$). We have that $|\mathcal{R}_t| = \binom{n}{t}2^t$. Let $K_{t,s}$ denote the set of restrictions ρ such that $f|_\rho$ is not a s-CNF. We need to bound $|K_{t,s}|/|\mathcal{R}_t|$ by the right hand side of (1) to prove the lemma. We'll do that by showing a one-to-one function mapping $K_{t,s}$ into the set $Z \times S$ where Z is the set of restrictions of at least $t+s$ variables (i.e. $Z = \cup_{t' \geq t+s} \mathcal{R}_{t'}$) and S is some set of size 3^{2ks}. This will prove the lemma since at he range $t' \gg n/2$, $\binom{n}{t'} \approx \left(\frac{n}{n-t'}\right)^{n-t'}$ and hence Z will be of size bounded by roughly $n2^s \left(\frac{n-t}{n}\right)^s |\mathcal{R}_t|$. We leave verifying the exact bound as Exercise 3.

Mapping $K_{t,s}$ into $Z \times S$.   Let $\rho \in K_{t,s}$ be a restriction fixing t variables such that $f|_\rho$ is not an s-CNF. We need to map ρ in a one-to-one way into some restriction ρ^* of at least $t + s$ variables, and some additional element in a set S of size at most 3^{2ks}.

Special case: each term has at most one "live" variable.   To get some intuition for the proof, consider first the case that for each term t in the k-DNF formula for f, ρ either fixed t to the value 0 or left a *single* unassigned variable in t, in which case we say that t's value is ? (ρ can't fix a term to the value 1 since we assume $f|_\rho$ is not constant). We denote by $x_1, \ldots, x_s$ denote the

first s such unassigned variables, according to some canonical ordering of the terms for the k-DNF formula of f (there are more than s since otherwise $f|_\rho$ would be expressible as an s-CNF). For each such variable x_i, let $\mathtt{term}_i$ be the ?-valued term in which x_i appears. Let R_i be the operation of setting x_i to the value that ensures $\mathtt{term}_i$ is true. We'll map ρ to $\tau_1 = R_1 R_2 \cdots R_s \rho$. That is, apply R_s to ρ, then apply R_{k-1} to ρ, $\cdots$, then apply R_1 to ρ. The crucial insight is that given τ_1, one can deduce $\mathtt{term}_1$: this is the first term that is true in $f|_{\tau_1}$. One might think that the second term that is true in $f|_{\tau_1}$ is $\mathtt{term}_2$ but that's not necessarily the case, since the variable x_1 may have appeared several times, and so setting it to R_1 may have set other terms to true (it could not have set other terms to false, since this would imply that $f|_\rho$ includes an OR of x_i and $\neg x_i$, and hence is the constant one function). We thus supply as part of the mapping a string $w_1 \in \{0, 1, \star\}^s$ that tells us the assignment of the k variables of $\mathtt{term}_1$ in $\tau_2 = R_2 \cdots R_s \rho$. Given that information we can "undo" R_1 and move from τ_1 to τ_2. Now in τ_2, $\mathtt{term}_2$ is the first satisfied term. Continuing on this way we see that from τ_1 (which is an assignment of at least $t + s$ variables) and strings $w_1, \ldots, w_s$ that are defined as above, we can recover ρ, implying that we have a one-to-one mapping that takes ρ into an assignment of at least $t + s$ variables and a sequence in $\{0, 1, \star\}^{ks}$.

The general case.   We now consider the general case, where some terms might have more than one unassigned variable in them. We let $\mathtt{term}_1$ be the first ?-valued term in $f|_\rho$ and let x_1 be the first unassigned variable in $\mathtt{term}_1$. Once again, we have an operation R_1 that will make $\mathtt{term}_1$ true, although this time we think of R_1 as assigning to all the k variables in $\mathtt{term}_1$ the unique value that makes the term true. We also have an operation L_1 assigning a value to x_1 such that $f|_{L_1 \rho}$ cannot be expressed by an $s - 1$-CNF. Indeed, if for both possible assignments to x_1 we get an $s - 1$-CNF then $f|_\rho$ is an s-CNF. We note that it's not necessarily the case that x_1's value under $L_1 \rho$ is different from its value under $R_1 \rho$, but it is the case that $\mathtt{term}_1$'s value is either ? or FALSE under $L_1 \rho$ (since otherwise $f|_{L_1 \rho}$ would be constant). We let $\mathtt{term}_2$ be the first ?-valued term in $f|_{L_1 \rho}$ (note that $\mathtt{term}_2 \geq \mathtt{term}_1$) and let x_2 be the first unassigned variable in $\mathtt{term}_2$. Once again, we have an operation R_2 such that $\mathtt{term}_2$ is the first true term in $f|_{R_2 L_1 \rho}$ and operation L_2 such that $f|_{L_2 L_1 \rho}$ is not a $s - 2$-CNF. Continuing in this way we come up with operations $L_1, \ldots, L_s, R_1, \ldots, R_s$ such that if we let ρ_i be the assignment $L_i \cdots L_1 \rho$ (with $\rho_0 = \rho$) then for $1 \leq i \leq s$:

- $\mathtt{term}_i$ is the first ?-valued term in $f|_{\rho_{i-1}}$.

- $\mathtt{term}_i$ is the first true-valued term in $f|_{R_i \rho_{i-1}}$.

- L_i agrees with ρ_{i-1} on all variables assigned a value by ρ_{i-1}.

- R_i agrees with ρ_i on all variables assigned a value by ρ_i.

For $1 \leq i \leq s$, define τ_i to be $R_i R_{i+1} \cdots R_s \rho_s$, and define $\tau_{s+1} = \rho_s$. We have that $\mathtt{term}_i$ is the first true term in $f|_{\tau_i}$: indeed, all the operations in τ_i do not change variables assigned values by ρ_{i-1} and there $\mathtt{term}_i$ is the first ?-valued term. Thus τ_i cannot make any earlier term true. However, since the last operation applied is R_i, $\mathtt{term}_i$ is true in $f|_{\tau_i}$.

Let $z_1, \ldots, z_s$ and $w_1, \ldots, w_s$ be $2s$ strings in $\{0, 1, \star\}^s$ defined as follows: z_i describes the values assigned to the k variables appearing in $\mathtt{term}_i$ by ρ_{i-1} and w_i describes the value assigned to $\mathtt{term}_i$'s variables by τ_{i+1}. Clearly, from $\mathtt{term}_i$, z_i and the assignment ρ_i one can compute ρ_{i-1} and

from $\texttt{term}_i$, w_i and the assignment τ_i one can compute τ_{i+1}. We'll map ρ to τ_1 and the sequence $z_1, \ldots, z_s, w_1, \ldots, w_s$. Note that τ_1 does assign values to at least s variables not assigned by ρ, and that from τ_1 we can find $\texttt{term}_1$ (as this is the first true term in $f|_{\tau_1}$) and then using w_1 recover τ_2 and continue in this way until we recover the original assignment ρ. Thus this mapping is a one-to-one map from $T_{t,s}$ to $Z \times \{0, 1, \star\}^{2ks}$. ∎

13.2 Circuits With "Counters":ACC

One way to extend the $\mathbf{AC}^0$ lowerbounds of the previous section was to define a more general class of circuits. What if we allow more general gates? The simplest example is a parity gate. Clearly, an $\mathbf{AC}^0$ circuit provided with parity gates can can compute the parity function. But are there still other functions that it cannot compute? Razborov proved the first such lowerbound using his *Method of Approximations*. Smolensky later extended this work and clarified this method for the circuit class considered here.

Normally we think of a modular computation as working with numbers rather than bit, but it is sufficient to consider modular gates whose output is always $0/1$.

DEFINITION 13.3 (MODULAR GATES)
For any integer m, the MOD_m gate outputs 0 if the sum of its inputs is 0 modulo m, and 1 otherwise.

DEFINITION 13.4 (**ACC**)
For integers $m_1, m_2, \ldots, m_k > 1$ we say a language L is in $\mathbf{ACC}^0[m_1, m_2, \ldots, m_k]$ if there exists a circuit family $\{C_n\}$ with constant depth and polynomial size (and unbounded fan-in) consisting of $\wedge$, $\vee$, $\neg$ and $MOD_{m_1}, \ldots, MOD_{m_k}$ gates accepting L.

The class $\mathbf{ACC}^0$ contains every language that is in $\mathbf{ACC}^0(m_1, m_2, \ldots, m_k)$ for some $k \geq 0$ and $m_1, m_2, \ldots, m_k > 1$.

Good lowerbounds are known only when the circuit has one kind of modular gate.

THEOREM 13.5 (RAZBOROV,SMOLENSKY)
For distinct primes p and q, the function MOD_p is not in $\mathbf{ACC}^0(q)$.

We exhibit the main idea of this result by proving that the parity function cannot be computed by an $\mathbf{ACC}^0(3)$ circuit.
PROOF: The proof proceeds in two steps.

Step 1. In the first step, we show (using induction on h) that for any depth h MOD_3 circuit on n inputs and size S, there is a polynomial of degree $(2l)^h$ which agrees with the circuit on $1 - S/2^l$ fraction of the inputs. If our circuit C has depth d then we set $2l = n^{1/2d}$ to obtain a degree $\sqrt{n}$ polynomial that agrees with C on $1 - S/2^{n^{1/2d}/2}$ fraction of inputs.

Step 2 We show that no polynomial of degree $\sqrt{n}$ agrees with MOD_2 on more than $49/50$ fraction of inputs.

Together, the two steps imply that $S > 2^{n^{1/2d}/2}/50$ for any depth d circuit computing MOD_2, thus proving the theorem. Now we give details.

Step 1. Consider a node g in the circuit at a depth h . (The input is assumed to have depth 0.) If $g(x_1, \cdots, x_n)$ is the function computed at this node, we desire a polynomial $\tilde{g}(x_1, \cdots, x_n)$ over $GF(3)$ with degree $(2l)^h$, such that $g(x_1, \ldots, x_n) = \tilde{g}(x_1, \ldots, x_n)$ for "most" $x_1, \ldots, x_n \in \{0, 1\}$. We will also ensure that on every input in $\{0, 1\}^n \subseteq GF(3)$, polynomial $\tilde{g}$ takes a value in $\{0, 1\}$. This is without loss of generality since we can just square the polynomial. (Recall that the elements of $GF(3)$ are $0, -1, 1$ and $0^2 = 0$, $1^2 = 1$ and $(-1)^2 = 1$.)

We construct the approximator polynomial by induction. When $h = 0$ the "gate" is an input wire x_i, which is exactly represented by the degree 1 polynomial x_i. Suppose we have constructed approximators for all nodes up to height $h - 1$ and g is a gate at height h.

1. If g is a NOT gate, then $g = \neg f_1$ for some other gate f_1 that is at height $h - 1$ or less. The inductive hypothesis gives an approximator $\tilde{f}_1$ for f_1. Then we use $\tilde{g} = 1 - \tilde{f}_1$ as the approximator polynomial for g; this has the same degree as $\tilde{f}_1$. Whenever $\tilde{f}_1 = f_1$ then $\tilde{g} = g$, so we introduced no new error.

2. If g is a MOD_3 gate with inputs $f_1, f_2, \ldots, f_k$, we use the approximation $\tilde{g} = (\sum_{i=0}^{k} \tilde{f}_i)^2$. The degree increases to at most $2 \times (2l)^{h-1} < (2l)^h$. Since $0^2 = 0$ and $(-1)^2 = 1$, we introduced no new error.

3. If g is an AND or an OR gate, we need to be more careful. Suppose $g = \wedge_{i=0}^{k} f_i$. The naive approach would be to replace g with the polynomial $\Pi_{i \in I} \tilde{f}_i$. For an OR gate $g = \vee_{i=0}^{k} f_i$ De Morgan's law gives a similar naive approximator $1 - \prod_{i \in I}(1 - \tilde{f}_i)$. Unfortunately, both of these multiply the degree by k, the fanin of the gate, which could greatly exceed $2l$.

   The correct solution involves introducing some error. We give the solution for OR; De Morgan's law allows AND gates to be handled similarly.

   If $g = \vee_{i=0}^{k} f_i$, then $g = 1$ if and only if at least one of the $f_i = 1$. Furthermore, by the *random subsum principle* (see Section **??** in Appendix A) if any of the $f_i = 1$, then the sum (over $GF(3)$) of a random subset of $\{f_i\}$ is nonzero with probability at least $1/2$.

   Randomly pick l subsets $S_1, \cdots, S_l$ of $\{1, \ldots, k\}$. Compute the l polynomials $(\sum_{j \in S_i} \tilde{f}_j)^2$, each of which has degree at most twice that of the largest input polynomial. Compute the OR of these l terms using the naive approach. We get a polynomial of degree at most $2l \times (2l)^{h-1} = (2l)^h$. For any x, the probability over the choice of subsets that this polynomial differs from $OR(\tilde{f}_1, \ldots, \tilde{f}_k)$ is at most $\frac{1}{2^l}$. So, by the probabilistic method, there *exists* a choice for the l subsets such that the probability over the choice of x that this polynomial differs from $OR(\tilde{f}_1, \cdots, \tilde{f}_k)$ is at most $\frac{1}{2^l}$. We use this choice of the subsets to construct the approximator.

Applying the above procedure for each gate gives an approximator for the output gate of degree $(2l)^d$ where d is depth of the entire circuit. Each operation of replacing the gate by its approximator polynomial introduces error on at most $1/2^l$ fraction of all inputs, so the overall fraction of erroneous inputs for the approximator is at most $S/2^l$. (Note that errors at different gates may affect each other. Error introduced at one gate may be cancelled out by errors at another gate higher up. We

are being pessimistic in applying the union bound to *upperbound* the probability that any of the approximator polynomials anywhere in the circuit miscomputes.)

Step 2. Suppose that a polynomial f agrees with the MOD_2 function for all inputs in a set $G' \subseteq 0, 1^n$. If the degree of f is bounded by $\sqrt{n}$, then we show $|G'| < \left(\frac{49}{50}\right)2^n$.

Consider the change of variables $y_i = 1 + x_i \pmod 3$. (Thus $0 \to 1$ and $1 \to -1$.) Then, G' becomes some subset G of $\{-1, 1\}^n$, and f becomes some other polynomial, say $g(y_1, y_2, \ldots, y_n)$, which still has degree $\sqrt{n}$. Moreover,

$$
MOD_2(x_1, x_2, \ldots, x_n) = \begin{cases} 1 & \Rightarrow \Pi_{i=1}^n y_i = -1 \\ 0 & \Rightarrow \Pi_{i=1}^n y_i = 1 \end{cases} \tag{2}
$$

Thus $g(y_1, y_2, \ldots, y_n)$, a degree $\sqrt{n}$ polynomial, agrees with $\Pi_{i=1}^n y_i$ on G. This is decidedly odd, and we show that any such G must be small. Specifically, let F_G be the set of all functions $S : G \to \{0, 1, -1\}$. Clearly, $|F_G| = 3^{|G|}$, and we will show $|F_G| \leq 3^{\left(\frac{49}{50}\right)2^n}$, whence Step 2 follows.

LEMMA 13.6
For every $S \in F_G$, *there exists a polynomial* g_S *which is a sum of monomials* $a_I \prod_{i \in I} y_i$ *where* $|I| \leq \frac{n}{2} + \sqrt{n}$ *such that* $g_S(x) = S(x)$ *for all* $x \in G$.

PROOF: Let $\hat{S} : GF(3)^n \to GF(3)$ be any function which agrees with S on G. Then $\hat{S}$ can be written as a polynomial in the variables y_i. However, we are only interested in its values on $(y_1, y_2, \ldots, y_n) \in \{-1, 1\}^n$, when $y_i^2 = 1$ and so every monomial $\Pi_{i \in I} y_i^{r_i}$ has, without loss of generality, $r_i \leq 1$. Thus $\hat{S}$ is a polynomial of degree at most n. Now consider any of its monomial terms $\Pi_{i \in I} y_i$ of degree $|I| > n/2$. We can rewrite it as

$$
\Pi_{i \in I} y_i = \Pi_{i=1}^n y_i \Pi_{i \in \bar{I}} y_i, \tag{3}
$$

which takes the same values as $g(y_1, y_2, \ldots, y_n)\Pi_{i \in \bar{I}} y_i$ over $\{-1, 1\}^n$. Thus every monomial in $\hat{S}$ has degree at most $\frac{n}{2} + \sqrt{n}$. ∎

To conclude, we bound the number of polynomials whose every monomial with a degree at most $\frac{n}{2} + \sqrt{n}$. Clearly this number is #polynomials $\leq 3^{\#\text{monomials}}$, and

$$
\#\text{monomials} \leq \left|\left\{N \subseteq \{1 \cdots n\} \,|\, |N| \leq \frac{n}{2} + \sqrt{n}\right\}\right| \tag{4}
$$

$$
\leq \sum_{i \leq \frac{n}{2} + \sqrt{n}} \binom{n}{i} \tag{5}
$$

Using knowledge of the tails of a binomial distribution (or alternatively, direct calculation),

$$
\leq \frac{49}{50}2^n \tag{6}
$$

∎

13.3 Lowerbounds for monotone circuits

A Boolean circuit is *monotone* if it contains only AND and OR gates, and no NOT gates. Such a circuit can only compute monotone functions, defined as follows.

DEFINITION 13.7
For $x, y \in \{0,1\}^n$, we denote $x \preceq y$ if every bit that is 1 in x is also 1 in y. A function $f : \{0,1\}^n \to \{0,1\}$ is *monotone* if $f(x) \le f(y)$ for every $x \preceq y$.

REMARK 13.8
An alternative characterization is that f is *monotone* if for every input x, changing a bit in x from 0 to 1 cannot change the value of the function from 1 to 0.

It is easy to check that every monotone circuit computes a monotone function, and every monotone function can be computed by a (sufficiently large) monotone circuit. CLIQUE is a monotone function since adding an edge to the graph cannot destroy any clique that existed in it. In this section we show that the CLIQUE function can not be computed by polynomial (and in fact even subexponential) sized monotone circuits:

THEOREM 13.9 ([RAZ85B, AB87])
Denote by $\mathsf{CLIQUE}_{k,n} : \{0,1\}^{\binom{n}{2}} \to \{0,1\}$ *be the function that on input an adjacency matrix of an n-vertex graph G outputs 1 iff G contains a k-vertex clique.*

There exists some constant $\epsilon > 0$ such that for every $k \le n^{1/4}$, there's no monotone circuit of size less than $2^{\epsilon\sqrt{k}}$ that computes $\mathsf{CLIQUE}_{k,n}$.

We believe CLIQUE does not have polynomial-size circuits even allowing NOT gates (i.e., that $\mathbf{NP} \not\subseteq \mathbf{P}/\text{poly}$). In fact, a seemingly plausible approach to proving this might be to show that for *every* monotone function f, the monotone circuit complexity of f is polynomially related to the general (non-monotone) circuit complexity. Alas, this conjecture was refuted by Razborov ([Raz85a], see also [Tar88]).

13.3.1 Proving Theorem 13.9

Clique Indicators

To get some intuition why this theorem might be true, lets show that $\mathsf{CLIQUE}_{k,n}$ can't be computed (or even approximated) by subexponential monotone circuits of a very special form. For every $S \subseteq [n]$, let C_S denote the function on $\{0,1\}^{\binom{n}{2}}$ that outputs 1 on a graph G iff the set S is a clique in G. We call C_S the *clique indicator* of S. Note that $\mathsf{CLIQUE}_{k,n} = \bigvee_{S \subseteq [n], |S|=k} \mathsf{C}_S$. We'll now prove that $\mathsf{CLIQUE}_{k,n}$ can't be computed by an OR of less than $n^{\sqrt{k}/20}$ clique indicators.

Let $\mathcal{Y}$ be the following distribution on n-vertex graphs: choose a set $K \subseteq [n]$ with $|K| = k$ at random, and output the graph that has a clique on K and no other edges. Let $\mathcal{N}$ be the following distribution on n-vertex graphs: choose a function $c : [n] \to [k-1]$ at random, and place an edge between u and v iff $c(u) \ne c(v)$. With probability one, $\mathsf{CLIQUE}_{n,k}(\mathcal{Y}) = 1$ and $\mathsf{CLIQUE}_{n,k}(\mathcal{N}) = 0$. The fact that $\mathsf{CLIQUE}_{n,k}$ requires an OR of at least $n^{\sqrt{k}/20}$ clique indicators follows immediately from the following lemma:

LEMMA 13.10
Let n be sufficiently large, $k \leq n^{1/4}$ and $S \subseteq [n]$. Then either $\mathbf{Pr}[\mathsf{C}_S(\mathcal{N}) = 1] \geq 0.99$ or $\mathbf{Pr}[\mathsf{C}_S(\mathcal{Y}) = 1] \leq n^{-\sqrt{k}/20}$

PROOF: Let $\ell = \sqrt{k-1}/10$. If $|S| \leq \ell$ then by the birthday bound, we expect a random $f : S \to [k-1]$ to have less than 0.01 collisions and hence by Markov the probability f is one to one is at least 0.99. This implies that $\mathbf{Pr}[\mathsf{C}_S(\mathcal{N}) = 1] \geq 0.99$.

If $|S| > \ell$ then $\mathbf{Pr}[\mathsf{C}_S(\mathcal{Y}) = 1]$ is equal to the probability that $S \subseteq K$ for a random $K \subseteq [n]$ of size k. This probability is equal to $\binom{n-\ell}{k-\ell}/\binom{n}{k}$ which is at most $\binom{n}{k-\sqrt{k-1}/10}/\binom{n}{k}$ which, by the formula for the binomial coefficients, is less than $\left(\frac{2k}{n}\right)^\ell \leq n^{-0.7\ell} < n^{-\sqrt{k}/20}$ (for sufficiently large n). ∎

Approximation by clique indicators.

Together with Lemma 13.10, the following lemma implies Theorem 13.9:

LEMMA 13.11
Let C be a monotone circuit of size s. Let $\ell = \sqrt{k}/10$. Then, there exist sets $S_1, \ldots, S_m$ with $m \leq n^{\sqrt{k}/20}$ such that

$$\mathbf{Pr}_{G \in_R \mathcal{Y}}[\bigvee_i \mathsf{C}_{S_i}(G) \geq C(G)] > 0.9 \tag{7}$$

$$\mathbf{Pr}_{G \in_R \mathcal{N}}[\bigvee_i \mathsf{C}_{S_i}(G) \leq C(G)] > 0.9 \tag{8}$$

$$\tag{9}$$

PROOF: Set $\ell = \sqrt{k}/10$, $p = 10\sqrt{k}\log n$ and $m = (p-1)^\ell \ell!$. Note that $m \ll n^{\sqrt{k}/20}$. We can think of the circuit C as the sequence of s monotone functions $f_1, \ldots, f_s$ from $\{0,1\}^{\binom{n}{2}}$ to $\{0,1\}$ where each function f_k is either the AND or OR of two functions $f_{k'}, f_{k''}$ for $k', k'' < k$ or is the value of an input variable $x_{u,v}$ for $u, v \in [n]$ (i.e., $f_k = \mathsf{C}_{\{u,v\}}$). The function that C computes is f_s. We'll show a sequence of functions $\tilde{f}_1, \ldots, \tilde{f}_s$ such that each function $\tilde{f}_k$ is **(1)** an OR of at most m clique indicators $\mathsf{C}_{S_1}, \ldots, \mathsf{C}_{S_m}$ with $|S_i| \leq \ell$ and **(2)** $\tilde{f}_k$ approximates f_k in the sense of (7) and (8). We call a function $\tilde{f}_k$ satisfying **(1)** an (ℓ, m)-function. The result will follow by considering the function $\tilde{f}_s$.

We construct the functions $\tilde{f}_1, \ldots, \tilde{f}_s$ by induction. For $1 \leq k \leq s$, if f_k is an input variable then we let $\tilde{f}_k = f_k$. If $f_k = f_{k'} \vee f_{k''}$ then we let $\tilde{f}_{k'} \sqcup \tilde{f}_{k''}$ and if $f_k = f_{k'} \wedge f_{k''}$ then we let $\tilde{f}_{k'} \sqcap \tilde{f}_{k''}$, where the operations $\sqcup, \sqcap$ will be defined below. We'll prove that for every $f, g : \{0,1\}^{\binom{n}{2}} \to \{0,1\}$ **(a)** if f and g are (m, ℓ)-functions then so is $f \sqcup g$ (resp. $f \sqcap g$) and **(b)** $\mathbf{Pr}_{G \in_R \mathcal{Y}}[f \sqcup g\,(G) < f \cup g\,(G)] < 1/(10S)$ (resp. $\mathbf{Pr}_{G \in_R \mathcal{Y}}[f \sqcap g\,(G) < f \cap g\,(G)] < 1/(10S))$ and $\mathbf{Pr}_{G \in_R \mathcal{N}}[f \sqcup g\,(G) > f \cup g\,(G)] < 1/(10S)$ (resp. $\mathbf{Pr}_{G \in_R \mathcal{Y}}[f \sqcap g\,(G) < f \cap g\,(G)] < 1/(10S))$. The lemma will then follow by showing using the union bound that with probability ≥ 0.9 the equations of Condition **(b)** hold for all $\tilde{f}_1, \ldots, \tilde{f}_s$. We'll now describe the two operations $\sqcup, \sqcap$. Condition **(a)** will follow from the definition of the operations, while Condition **(b)** will require a proof.

The operation $f \sqcup g$. Let f, g be two (m, ℓ)-functions: that is $f = \bigvee_{i=1}^{m} C_{S_i}$ and $g = \bigvee_{j=1}^{m} C_{T_j}$ (if f or g is the OR of less than m clique indicators we can add duplicate sets to make the number m). Consider the function $h = C_{Z_1} \cup \cdots \cup C_{Z_{2m}}$ where $Z_i = S_i$ and $Z_{m+j} = T_j$ for $1 \le i, j \le m$. The function h is not an (m, ℓ)-function since it is the OR of $2m$ clique indicators. We make it into an (m, ℓ)-function in the following way: as long as there are more than m distinct sets, find p subsets $Z_{i_1}, \ldots, Z_{i_p}$ that are in a *sunflower* formation. That is, there exists a set $Z \subseteq [n]$ such that for every $1 \le j, j' \le p$, $Z_{i_j} \cap Z_{i,j'} = Z$. Replace the functions $C_{Z_{i_1}}, \ldots, C_{Z_{i_p}}$ in the function h with the function C_Z. Once we obtain an (m, ℓ)-function h' we define $f \sqcup g$ to be h'. We won't get stuck because of the following lemma (whose proof we defer):

LEMMA 13.12 (SUNFLOWER LEMMA [ER60])
Let $\mathcal{Z}$ be a collection of distinct sets each of cardinality at most ℓ. If $|\mathcal{Z}| > (p-1)^{\ell}\ell!$ then there exist p sets $Z_1, \ldots, Z_p \in \mathcal{Z}$ and set Z such that $Z_i \cap Z_j = Z$ for every $1 \le i, j \le p$.

The operation $f \sqcap g$. Let f, g be two (m, ℓ)-functions: that is $f = \bigvee_{i=1}^{m} C_{S_i}$ and $g = \bigvee_{j=1}^{m} C_{T_j}$. Let h be the function $\bigvee_{1 \le i, j \le m} C_{S_i \cup T_j}$. We perform the following steps on h: **(1)** Discard any function C_Z for $|Z| > \ell$. **(2)** Reduce the number of functions to m by applying the sunflower lemma as above.

Proving Condition (b). To complete the proof of the lemma, we prove the following four equations:

- $\mathbf{Pr}_{G \in_R \mathcal{Y}}[f \sqcup g\ (G) < f \cup g\ (G)] < 1/(10S)$.

  If $Z \subseteq Z_1, \ldots, Z_p$ then for every i, $C_{Z_i}(G)$ implies that $C_Z(G)$ and hence the operation $f \sqcup g$ can't introduce any "false negatives".

- $\mathbf{Pr}_{G \in_R \mathcal{N}}[f \sqcup g\ (G) > f \cup g\ (G)] < 1/(10S)$.

  We can introduce a "false positive" on a graph G only if when we replace the clique indicators for a sunflower $Z_1, \ldots, Z_p$ with the clique indicator for the common intersection Z, it is the case that $C_Z(G)$ holds even though $C_{Z_i}(G)$ is false for every i. Recall that we choose $G \in_R \mathcal{N}$ by choosing a random function $c : [n] \to [k-1]$ and adding an edge for every two vertices u, v with $c(u) \ne c(v)$. Thus, we get a false positive if c is one-to-one on Z (we denote this event by B) but *not* one-to-one on Z_i for every $1 \le i \le p$ (we denote these events by $A_1, \ldots, A_p$). We'll show that the intersection of B and $A_1, \ldots, A_p$ happens with probability at most 2^{-p} which (by the choice of p) is less than $1/(10m^2 s)$. Since we apply the reduction step at most m times the equation will follow.

  Since $\ell < \sqrt{k-1}/10$, for every i, $\mathbf{Pr}[A_i | B] < 1/2$ (the probability that there'll be a collision on the at most ℓ elements of $Z_i \setminus Z$ is less than half). Conditioned on B, the events $A_1, \ldots, A_p$ are independent, since they depend on the values of c on disjoint sets, and hence we have that $\mathbf{Pr}[A_1 \wedge \cdots \wedge A_p \wedge B] \le \mathbf{Pr}[A_1 \wedge \cdots \wedge A_p | B] = \prod_{i=1}^{p} \mathbf{Pr}[A_p | B] \le 2^{-p}$.

- $\mathbf{Pr}_{G \in_R \mathcal{Y}}[f \sqcap g\ (G) < f \cap g\ (G)] < 1/(10S)$.

  By the distributive law $f \cap g = \bigvee_{i,j}(C_{S_i} \cap C_{T_j})$. A graph G in the support of $\mathcal{Y}$ consists of a clique over some set K. For such a graph $C_{S_i} \cap C_{T_j}$ holds iff $S_i, T_j \subseteq K$ and thus $C_{S_i} \cap C_{T_j}$ holds

iff $C_{S_i \cup T_j}$ holds. We can introduce a false negative when we discard functions of the form C_Z for $|Z| > \ell$, but by Lemma 13.10, for such sets Z, $\mathbf{Pr}[C_Z(\mathcal{Y}) = 1] < n^{-\sqrt{k}/20} < 1/(10sm^2)$. The equation follows since we discard at most m^2 such sets.

- $\mathbf{Pr}_{G \in_R \mathcal{N}}[f \sqcap g\,(G) > f \cap g\,(G)] < 1/(10S)$.

  Since $C_{S \cup T}$ implies both C_S and C_T, we can't introduce false positives by moving from $f \cap g$ to $\bigvee_{i,j} C_{S_i \cup T_j}$. We can't introduce false positives by discarding functions from the OR. Thus, the only place where we can introduce false positives is where we replace the clique indicators of a sunflower with the clique indicator of the common intersection. We bound this probability in the same way as this was done for the $\sqcup$ operator.

■

Proof of the sunflower lemma (Lemma 13.12). The proof is by induction on ℓ. The case $\ell = 1$ is trivial since distinct sets of size 1 must be disjoint. For $\ell > 1$ let $\mathcal{M}$ be a maximal subcollection of $\mathcal{Z}$ containing only disjoint sets. Because of $\mathcal{M}$'s maximality for every $Z \in \mathcal{Z}$ there exists $x \in \cup\mathcal{M} = \cup_{M \in \mathcal{M}} M$ such that $x \in Z$. If $|\mathcal{M}| \geq p$ we're done, since such a collection is already a sunflower. Otherwise, since $|\cup\mathcal{M}| \leq (p-1)\ell$ by averaging there's an $x \in \cup\mathcal{M}$ that appears in at least a $\frac{1}{\ell(p-1)}$ fraction of the sets in $\mathcal{Z}$. Let $Z_1, \ldots, Z_k$ be the sets containing x, and note that $k > (p-1)^{\ell-1}(\ell-1)!$. Thus, by induction there are p sets among the $\ell-1$-sized sets $Z_1 \setminus \{x\}, \cdots, Z_k \setminus \{x\}$ that form a sunflower, adding back x we get the desired sunflower among the original sets. Note that the statement (and proof) assume nothing about the size of the universe the sets in $\mathcal{Z}$ live in. ■

13.4 Circuit complexity: The frontier

Now we sketch the "frontier" of circuit lowerbounds, namely, the dividing line between what we can prove and what we cannot. Along the way we also define multi-party communication, since it may prove useful for proving some new circuit lowerbounds.

13.4.1 Circuit lowerbounds using diagonalization

We already mentioned that the best lowerbound on circuit size for an **NP** problem is $4.5n - o(n)$. For **PH** better lowerbounds are known: one of the exercises in Chapter 6 asked you to show that some for every $k > 0$, some language in **PH** (in fact in $\mathbf{\Sigma}_2^p$) requires circuits of size $\Omega(n^k)$. The latter lowerbound uses diagonalization, and one imagines that classes "higher up" than **PH** should have even harder languages.

Frontier 1: Does **NEXP** have languages that require super-polynomial size circuits?

If we go a little above **NEXP**, we can actually prove a super-polynomial lowerbound: we know that $\mathbf{MA_{EXP}} \not\subseteq \mathbf{P}/poly$ where $\mathbf{MA_{EXP}}$ is the set of languages accepted by a one round proof with an all powerful prover and an exponential time *probabilistic* verifier. This follows from the fact

Figure unavailable in pdf file.

Figure 13.3: The depth 2 circuit with a symmetric output gate from Theorem 13.13.

that if $\mathbf{MA_{EXP}} \subseteq \mathbf{P}/\text{poly}$ then in particular $\mathbf{PSPACE} \subseteq \mathbf{P}/\text{poly}$. However, by $\mathbf{IP} = \mathbf{PSPACE}$ (Theorem 8.17) we have that in this case $\mathbf{PSPACE} = \mathbf{MA}$ (the prover can send in one round the circuit for computing the prover strategy in the interactive proof). However, by simple padding this implies that $\mathbf{MA_{EXP}}$ equals the class of languages in *exponential space*, which can be directly shown to not contain $\mathbf{P}/\text{poly}$ using diagonalization. Interestingly, this lower bound does not relativize (i.e., there's an oracle under which $\mathbf{MA_{NEXP}} \subseteq \mathbf{P}/\text{poly}$ [BFT98]).

13.4.2 Status of ACC versus P

The result that PARITY is not in $\mathbf{AC}^0$ separates $\mathbf{NC}^1$ from $\mathbf{AC}^0$. The next logical step would be to separate $\mathbf{ACC}^0$ from $\mathbf{NC}^1$. Less ambitiously, we would like to show even a function in $\mathbf{P}$ or $\mathbf{NP}$ that is not in $\mathbf{ACC}^0$.

The Razborov-Smolenksy method seems to fail when we allow the circuit even two types of modular gates, say MOD_2 and MOD_3. In fact if we allow the bounded depth circuit modular gates that do arithmetic mod q, when q is not a prime —a prime power, to be exact— we reach the limits of our knowledge. (The exercises ask you to figure out why the proof of Theorem 13.5 does not seem to apply when the modulus is a composite number.) To give one example, it it is consistent with current knowledge that the majority of n bits can be computed by linear size circuits of constant depth consisting entirely of MOD_6 gates. The problem seems to be that low-degree polynomials modulo m where m is composite are surprisingly expressive [BBR94].

Frontier 2: Show Clique is not in $\mathbf{ACC}^0(6)$.

Or even less ambitiously:

Frontier 2.1: Exhibit a language in $\mathbf{NEXP}$ that is not in $\mathbf{ACC}^0(6)$.

It is worth noting that thus far we are talking about *nonuniform* circuits (to which Theorem 13.5 also applies). Stronger lower bounds are known for uniform circuits: Allender and Gore [AG94] have shown that a decision version of the Permanent (and hence the Permanent itself) requires exponential size "Dlogtime-uniform" $\mathbf{ACC}^0$ circuits. (A circuit family $\{C_n\}$ is *Dlogtime uniform* if there exists a deterministic Turing machine M that given a triple (n, g, h) determines in linear time —i.e., $O(\log n)$ time when $g, h \leq \text{poly}(n)$— what types of gates g and h are and whether g is h's parent in C_n.)

But going back to nonuniform $\mathbf{ACC}^0$, we wish to mention an alternative representation of $\mathbf{ACC}^0$ circuits that may be useful in further lowerbounds. Let a *symmetric* gate be a gate whose output depends only on the number of inputs that are 1. For example, majority and mod gates are symmetric. Yao has shown that $\mathbf{ACC}^0$ circuits can be simplified to give an equivalent depth 2 circuits with a symmetric gate at the output (figure **??**). Beigel and Tarui subsequently improved Yao's result:

THEOREM 13.13 (YAO [YAO90], BEIGEL AND TARUI [BT94])
If $f \in \mathbf{ACC}^0$, then f can be computed by a depth 2 circuit C with a symmetric gate with quasipolynomial (i.e., $2^{\log^k n}$) fan-in at the output level and $\vee$ gates with polylogarithmic fan-in at the input level.

We will revisit this theorem below in Section 13.5.1.

13.4.3 Linear Circuits With Logarithmic Depth

When we restrict circuits to have bounded fanin we necessarily need to allow them to have non-constant (in fact, $\Omega(\log n)$) depth to have any reasonable power. With this in mind, the simplest interesting circuit class seems to be one of circuits wth linear size and logarithmic depth.

Frontier 3: Find an explicit function that cannot be computed by circuits of linear size and logarithmic depth.

(Note that by counting one can easily show that *some* function on n bits requires superpolynomial size circuits and hence bounded fan-in circuits with more than logarithmic depth; see the exercises on the chapter on circuits. Hence we want to show this for an explicit function, e.g. CLIQUE.)

Valiant thought about this problem in the '70s. His initial candidates for lowerbounds boiled down to showing that a certain graph called a *superconcentrator* needed to have superlinear size. He failed to prove thisand instead ended up proving that such superconcentrators do exist!

Another sideproduct of Valiant's investigations was the following important lemma concerning depth-reduction for such circuits.

LEMMA 13.14 (VALIANT)
In any circuit with m edges and depth d, there are $km/\log d$ edges whose removal leaves a circuit with depth at most $d/2^{k-1}$.

This lemma can be applied as follows. Suppose we have a circuit C of depth $c \log n$ with n inputs $\{x_1, \ldots, x_n\}$ and n outputs $\{y_1, \ldots, y_n\}$, and suppose $2^k \sim c/\epsilon$ where $\epsilon > 0$ is arbitrarily small. Removing $O(n/\log \log n)$ edges from C then results in a circuit with depth at most $\epsilon \log n$. But then, since C has *bounded* fan-in, we must have that each output y_i is connected to at most $2^{\epsilon \log n} = n^\epsilon$ inputs. So each output y_i in C is completely determined by n^ϵ inputs and the values of the omitted edges. So we have a "dense" encoding for the function $f_i(x_1, \ldots, x_n) = y_i$. We do not expect this to be the case for any reasonably difficult function.

13.4.4 Branching Programs

Just as circuits are used to investigate time requirements of Turing Machines, *branching programs* are used to investigate space complexity.

A branching program on n input variables $x_1, x_2, \ldots, x_n$ is a directed acyclic graph all of whose nodes of nonzero outdegree are labeled with a variable x_i. It has two nodes of outdegree zero that are labeled with an output value, ACCEPT or REJECT. The edges are labeled by 0 or 1. One of the nodes is designated the start node. A setting of the input variables determines a way to walk

on the directed graph from the start node to an output node. At any step, if the current node has label x_i, then we take an edge going out of the node whose label agrees with the value of x_i. The branching program is *deterministic* if every nonoutput node has exactly one 0 edge and one 1 edge leaving it. Otherwise it is *nondeterministic*. The *size* of the branching program is the number of nodes in it. The branching program complexity of a language is defined analogously with circuit complexity. Sometimes one may also require the branching program to be *leveled*, whereby nodes are arranged into a sequence of levels with edges going only from one level to the next. Then the *width* is the size of the largest level.

THEOREM 13.15
If $S(n) \geq \log n$ and $L \in \mathbf{SPACE}(S(n))$ then L has branching program complexity at most $c^{S(n)}$ for some constant $c > 1$.

PROOF: Essentially mimics our proof of Theorem'**??** that $\mathbf{SPACE}(S(n)) \subseteq \mathbf{DTIME}(2^{O(S(n))})$. The nodes of the branching program correspond to the configurations of the space-bounded TM, and it is labeled with variable x_i if the configuration shows the TM reading the ith bit in the input. ∎

Of course, a similar theorem is true about NDTMs and nondeterministic branching program complexity.

Frontier 4: Describe a problem in **P** (or even **NP**) that requires branching programs of size greater than $n^{1+\epsilon}$ for some constant $\epsilon > 0$.

There is some evidence that branching programs are more powerful than one may imagine. For instance, branching programs of constant width (reminiscent of a TM with $O(1)$ bits of memory) seem inherently weak. Thus the next result is unexpected.

THEOREM 13.16 (BARRINGTON [**?**])
A language has polynomial size, width 5 branching programs iff it is in $\mathbf{NC}^1$.

13.5 Approaches using communication complexity

Here we outline a concrete approach (rather, a setting) in which better lowerbounds may lead to a resolution of some of the questions above. It relates to generalizations of communication complexity introduced earlier. Mostly we will use *multiparty communication complexity*, though Section 13.5.4 will use *communication complexity of a relation*.

13.5.1 Connection to ACC^0 Circuits

Suppose $f(x_1, \ldots, x_k)$ has a depth-2 circuit with a symmetric gate with fan-in N at the output and $\wedge$ gates with fan-in $k-1$ at the input level (figure 2). The claim is that f's k-party communication complexity is at most $k \log N$. (This observation is due to Razborov and Wigderson [RW93]). To see the claim, first partition the $\wedge$ gates amongst the players. Each bit is not known to exactly one player, so the input bits of each $\wedge$ gate are known to at least one player; assign the gate to such a player with the lowest index. Players then broadcast how many of their gates output 1. Since this number has at most $\log N$ bits, the claim follows.

Figure unavailable in pdf file.

Figure 13.4: If f is computed by the above circuit, then f has a k-party protocol of complexity $k \log N$.

Our hope is to employ this connection with communication complexity in conjunction with Theorem 13.13 to obtain lower bounds on $\mathbf{ACC}^0$ circuits. For example, note that the function in Example ?? above cannot have $k < \log n/4$. However, this is not enough to obtain a lower bound on $\mathbf{ACC}^0$ circuits since we need to show that k is not polylogarithmic to employ Theorem 13.13. Thus a strengthening of the Babai Nisan Szegedy lowerbound to $\Omega(n/\mathrm{poly}(k))$ for say the CLIQUE function would close Frontier 2.

13.5.2 Connection to Linear Size Logarithmic Depth Circuits

Suppose that $f : \{0,1\}^n \times \{0,1\}^{\log n} \to \{0,1\}^n$ has bounded fan-in circuits of linear size and logarithmic depth. If $f(x,j,i)$ denotes the ith bit of $f(x,j)$, then Valiant's Lemma implies that $f(x,j,i)$ has a simultaneous 3-party protocol—that is, a protocol where all parties speak only once and write simultaneously on the blackboard (i.e., non-adaptively)—where,

- (x,j) player sends $n/\log\log n$ bits;

- (x,i) player sends n^ϵ bits; and

- (i,j) player sends $O(\log n)$ bits.

So, if we can show that a function does not have such a protocol, then we would have a lower bound for the function on linear size logarithmic depth circuits with bounded fan-in.

Conjecture: The function $f(x,j,i) = x_{j\oplus i}$, where $j \oplus i$ is the bitwise xor, is conjectured to be hard, i.e., f should not have a compact representation.

13.5.3 Connection to branching programs

The notion of multiparty communication complexity (at least the "number on the forehead" model discussed here) was invented by Chandra Furst and Lipton [?] for proving lowerbounds on branching programs, especially constant-width branching programs discussed in Section ??

13.5.4 Karchmer-Wigderson communication games and depth lowerbounds

The result that PARITY is not in $\mathbf{AC}^0$ separates $\mathbf{NC}^1$ from $\mathbf{AC}^0$. The next step would be to separate $\mathbf{NC}^2$ from $\mathbf{NC}^1$. (Of course, ignoring for the moment the issue of separating $\mathbf{ACC}^0$ from $\mathbf{NC}^1$.) Karchmer and Wigderson [KW90] described how communication complexity can be used to prove lowerbounds on the minimum depth required to compute a function. They showed the following result about monotone circuits, which we will not prove this result.

THEOREM 13.17
Detecting whether a graph has a perfect matching is impossible with monotone circuits of depth $O(\log n)$

However, we do describe the basic *Karchmer-Wigderson* game used to prove the above result, since it is relevant for nonmonotone circuits as well. For a function $f : \{0,1\}^n \to \{0,1\}$ this game is defined as follows.

*There are two players, **ZERO** and **ONE**. Player **ZERO** receives an input x such that $f(x) = 0$ and Player **ONE** receives an input y such that $f(y) = 1$. They communicate bits to each other, until they can agree on an $i \in \{1, 2, \ldots, n\}$ such that $x_i \neq y_i$.*

The mechanism of communication is defined similarly as in Chapter 12; there is a protocol that the players agree on in advance before receiving the input. Note that the key difference from the scenario in Chapter 12 is that the final answer is not a single bit, and furthermore, the final answer is not unique (the number of acceptable answers is equal to the number of bits that x, y differ on). Sometimes this is described as *computing a relation*. The relation in this case consists of all triples (x, y, i) such that $f(x) = 0$, $f(y) = 1$ and $x_i \neq y_i$.

We define $C_{KW}(f)$ as the communication complexity of the above game; namely, the maximum over all $x \in f^{-1}(0), y \in f^{-1}(1)$ of the number of bits exchanged in computing an answer for x, y. The next theorem shows that this parameter has a suprising alternative characterization. It assumes that circuits don't have NOT gates and instead the NOT gates are pushed down to the inputs using De Morgan's law. (In other words, the inputs may be viewed as $x_1, x_2, \ldots, x_n, \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}$.) Furthermore, AND and OR gates have fanin 2. (None of these assumptions is crucial and affects the theorem only marginally.)

THEOREM 13.18 ([KW90])
$C_{KW}(f)$ *is exactly the minimum depth among all circuits that compute* f.

PROOF: First, we show that if there is a circuit $\mathcal{C}$ of depth K that computes f then $C_{KW}(f) \leq K$. Each player has a copy of $\mathcal{C}$, and evaluates this circuit on the input given to him. Of course, it ealuates to 0 for Player **ZERO** and to 1 for Player **ONE**. Suppose the top gate is an OR. Then at least one of the two incoming wires to this gate must be 1, and in the first round, Player **ONE** sends one bit communicating which of these wires it was. Note that this wire is 0 for Player **ZERO**. In the next round the players focus on the gate that produced the value on this wire. (If the top gate is an AND on the other hand, then in the first round Player **ZERO** speaks, conveying which of the two incoming wires was 0. This wire will be 1 for Player **ONE**.) This goes on and the players go deeper down the circuit, always maintaining the invariant that the current gate has value 1 for Player **ONE** and 0 for Player **ZERO**. Finally, after at most K steps they arrive at an input bit. According to the invariant being maintained, this bit must be 1 for Player **ONE** and 0 for Player **ZERO**. Thus they both know an index i that is a valid answer.

For the reverse direction, we have to show that if $C_{KW}(f) = K$ then there is a circuit of depth at most K that computes f. We prove a more general result. For any two disjoint nonempty subsets $A \subseteq f^{-1}(0)$ and $B \subseteq f^{-1}(1)$, let $C_{KW}(A, B)$ be the communication complexity of the Karchmer-Wigderson game when x always lies in A and y in B. We show that there is a circuit of depth $C_{KW}(A, B)$ that outputs 0 on every input from A and 1 on every input from B. Such a circuit is called a *distinguisher* for sets A, B. The proof is by induction on $K = C_{KW}(A, B)$. The base case $K = 0$ is trivial since this means the players do not have to communicate at all to agree on an answer, say i. Hence $x_i \neq y_i$ for all $x \in A, y \in B$, which implies that either (a) $x_i = 0$ for

every $x \in A$ and $y_i = 0$ for every $y \in B$ or (b) $x_i = 1$ for every $x \in A$ and $y_i = 1$ for every $y \in B$. In case (a) we can use the depth 0 circuit x_i and in case (b) we can use the circuit $\overline{x_i}$ to distinguish A, B.

For the inductive step, suppose $C_{KW}(A, B) = K$, and at the first round Player **ZERO** speaks. Then A is the disjoint union of two sets A_0, A_1 where A_b is the set of inputs in A for which Player **ZERO** sends bit b. Then $C_{KW}(A_b, B) \leq K - 1$ for each b, and the inductive hypothesis gives a circuit C_b of depth at most $K - 1$ that distinguishes A_b, B. We claim that $C_0 \wedge C_1$ distinguishes A, B (note that it has depth at most K). The reason is that $C_0(y) = C_1(y) = 1$ for every $y \in B$ whereas for every $x \in A$, $C_0(x) \wedge C_1(x) = 0$ since if $x \in A_b$ then $C_b(x) = 0$. $\blacksquare$

Thus we have the following frontier.

Frontier 5: Show that some function f in **P** (or even **NEXP**!) has $C_{KW}(f) = \Omega(\log n \log \log n)$.

Karchmer, Raz, and Wigderson [KRW95] describe a candidate function that may work. It uses the fact a function on k bits has a truth table of size 2^k, and that most functions on k bits are hard (e.g., require circuit size $\Omega(2^k/k)$, circuit depth $\Omega(k)$, etc.). They define the function by assuming that part of the n-bit input encodes a very hard function, and this hard function is applied to the remaining input in a "tree" fashion.

For any function $g : \{0, 1\}^k \to \{0, 1\}$ and $s \geq 1$ define $g^{\circ s} : \{0, 1\}^{k^s} \to \{0, 1\}$ as follows. If $s = 1$ then $g^{\circ s} = g$. Otherwise express the input $x \in \{0, 1\}^{k^s}$ as $x_1 x_2 x_3 \cdots x_k$ where each $x_i \in \{0, 1\}^{k^{s-1}}$ and define

$$g^{\circ s}(x_1 x_2 \cdots x_k) = g(g^{\circ(s-1)}(x_1) g^{\circ(s-1)}(x_2) \cdots g^{\circ(s-1)}(x_k)).$$

Clearly, if g can be computed in depth d then $g^{\circ s}$ can be computed in depth sd. Furthermore, if one fails to see how one could reduce the depth for an arbitrary function.

Now we describe the KRW candidate function $f : \{0, 1\}^n \to \{0, 1\}$. Let $k = \lceil \log \frac{n}{2} \rceil$ and s be the largest integer such that $k^s \leq n/2$ (thus $s = \Theta(\frac{\log n}{\log \log n})$.) For any n-bit input x, let g_x be the function whose truth table is the first 2^k bits of x. Let $x|_2$ be the string of the last k^s bits of x. Then

$$f(x) = g_x^{\circ s}(x|_2).$$

According to our earlier intuition, when the first 2^k bits of x represent a really hard function —as they must for many choices of the input— then $g_x^{\circ s}(x|_2)$ should require depth $\Omega(sk) = \Omega(\frac{\log^2 n}{\log \log n})$. Of course, proving this seems difficult.

This type of complexity questions, whereby we are asking whether s instances of a problem are s times as hard as a single instance, are called *direct sum* questions. Similar questions have been studied in a variety of computational models, and sometimes counterintuitive results have been proven for them. One example is that by a counting argument there exists an $n \times n$ matrix A over $\{0, 1\}$, such that the smallest circuit computing the linear function $v \mapsto Av$ for $v \in \{0, 1\}^n$ is of size $\Omega(n^2)$. However, computing this function on n instances $v_1, \ldots, v_n$ can be done significantly faster than n^3 steps using fast matrix multiplication [Str69] (the current record is roughly $O(n^{2.38})$ [CW90]).

Chapter notes and history

Shannon defined circuit complexity, including monotone circuit complexity, in 1949. The topic was studied in Russia since the 1950s. (See Trakhtenbrot [Tra84] for some references.) Savage [Sav72] was the first to observe the close relationship between time required to decide a language on a TM and its circuit complexity, and to suggest circuit lowerbounds as a way to separate complexity classes. A burst of results in the 1980s, such as the separation of **P** from $\mathbf{AC}^0$ [FSS84, Ajt83] and Razborov's separation of monotone **NP** from monotone **P**/*poly* [Raz85b] raised hopes that a resolution of **P** versus **NP** might be near. These hopes were dashed by Razborov himself [Raz89] when he showed that his method of approximations was unlikely to apply to nonmonotone circuits. Later Razborov and Rudich [RR97] formalized what they called *natural proofs* to show that all lines of attack considered up to that point were unlikely to work. (See Chapter 22.)

Our presentation in Sections 13.2 and 13.3 closely follows that in Boppana and Sipser's excellent survey of circuit complexity [BS90], which is still useful and current 15 years later. (It omits discussion of lowerbounds on algebraic circuits; see [Raz04] for a recent result.)

Håstad's switching lemma [Has86] is a stronger form of results from[FSS84, Ajt83, Yao85]. The Razborov-Smolensky method of using approximator polynomials is from [Raz87], strengthened in[Smo87]. Valiant's observations about superlinear circuit lowerbounds are from a 1975 paper [Val75] and an unpublished manuscript—lack of progress on this basic problem gets more embarrassing by the day!.

The $4.5n - o(n)$ lowerbound on general circuits is from Lachish-Raz [LR01].

Exercises

§1 Suppose that f is computable by an **AC** 0 circuit C of depth d and size S. Prove that f is computable by an **AC** 0 circuit C' of size $10S$ and depth d that does not contain NOT gates but instead has n additional inputs that are negations of the original n inputs.

Hint: each gate in the old circuit gets a twin that computes its negation.

§2 Suppose that f is computable by an **AC** 0 circuit C of depth d and size S. Prove that f is computable by an **AC**0 C' circuit of size $(10S)^d$ and depth d where each gate has fanout 1.

§3 Prove that for $t > n/2$, $\binom{n}{t+k} \le \binom{n}{t} \left(\frac{n}{n-t}\right)^k$. Use this to complete the proof of Lemma 13.2 (Section 13.1.2).

§4 Show that $\mathbf{ACC}^0 \subseteq \mathbf{NC}^1$.

§5 Identify reasons why the Razborov-Smolensky method does not work when the circuit has mod$m$ gates, where m is a composite number.

§6 Show that representing the OR of n variables $x_1, x_2, \ldots, x_n$ exactly with a polynomial over $GF(q)$ where q is prime requires degree exactly n.

§7 The Karchmer-Wigderson game can be used to prove *upperbounds*, and not just lowerbounds. Show using this game that PARITY and MAJORITY are in $\mathbf{NC}^1$.

§8 Show that if a language is computed by a polynomial-size branching program of width 5 then it is in $\mathbf{NC}^1$.

§9 Prove Valiant's Lemma (Lemma 13.14).

Hint: A directed acyclic graph can be be turned into a leveled graph, such that if $u \leftarrow v$ is an edge then u occurs at a lower level than v. Label this edge by looking at the numbers given to the levels of u, v and remove the edges corresponding to the least popular label.

Chapter 14

Algebraic computation models

The Turing machine model captures computations on bits (equivalently, integers), but it does not always capture the spirit of algorithms which operate on, say the real numbers $\mathbb{R}$ or complex numbers $\mathbf{C}$. Such algorithms arise in a variety of applications such as numerical analysis, computational geometry, robotics, and symbolic algebra. A simple example is *Newton's method* for finding roots of a given real-valued function function f. It iteratively produces a sequence of candidate solutions $x_0, x_1, x_2, \ldots, \in \mathbb{R}$ where $x_{i+1} = x_i - f(x_i)/f'(x_i)$. Under appropriate conditions this sequence can be shown to converge to a root of f.

Of course, a perfectly defensible position to take is that even the behavior of such algorithms should be studied using TMs, since they will be run on real-life computers, which represent real numbers using finite precision. In this chapter though, we take a different approach and study models which do allow arithmetic operations on real numbers (or numbers from fields other than $\mathbb{R}$). Such an idealized model may not be implementable, strictly speaking, but it provides a useful approximation to the asymptotic behavior as computers are allowed to use more and more precision in their computations. Furthermore, one may be able to prove nontrivial lowerbounds for these models using techniques from well-developed areas of mathematics such as algebraic geometry and topology. (By contrast, boolean circuit lowerbounds have proven very difficult.)

However, coming up with a meaningful, well-behaved model of algebraic computation is not an easy task, as the following example suggests.

EXAMPLE 14.1 (PITFALLS AWAITING DESIGNERS OF SUCH MODELS)
A real number can encode infinite amount of information. For example, a single real number is enough to encode the answer to every instance of SAT (or any other language, in general). Thus, a model that can store any real number with infinite precision may not be realistic. Shamir has shown how to factor any integer n in poly$(\log n)$ time on a computer that can do real arithmetic with arbitrary precision.

The usual way to avoid this pitfall is to restrict the algorithms' ability to access individual bits (e.g., the machine may require more than polynomial time to extract a particular digit from

a real number). Or, sometimes (as in case of Algebraic Computation Trees) it is OK to consider unrealistically powerful models since the goal is to prove nontrivial lowerbounds —say, superlinear or quadratic— rather than arbitrary polynomial lowerbounds. After all, lowerbounds for unrealistically powerful models will apply to more realistic (and weaker) models as well.

This chapter is a sketchy introduction to algebraic complexity. It introduces three algebraic computation models: algebraic circuits, algebraic computation trees, and algebraic Turing Machines. The algebraic TM is closely related to the standard Turing Machine model and allows us to study similar questions for arbitrary fields — including decidability and complexity–that we earlier studied for strings over $\{0, 1\}$. We introduce an undecidable problem (namely, deciding membership in the Mandelbrot set) and and an **NP**-complete problem (decision version of Hilbert's Nullstellensatz) in this model.

14.1   Algebraic circuits

An *algebraic circuit* over a field F is defined by analogy with a boolean circuit. It consists of a directed acyclic graph. The leaves are called input nodes and labeled $x_1, x_2, \ldots, x_n$, except these take values in a field F rather than boolean variables. There are also special input nodes, labeled with the constants 1 and -1 (which are field elements). Each internal node, called a *gate*, is labeled with one of the arithmetic operations $\{+, \star\}$ rather than with the boolean operations $\vee, \wedge, \neg$ used in boolean circuits. There is only output node. We restrict indegree of each gate to 2. The *size* of the circuit is the number of gates in it. One can also consider algebraic circuits that allow division ($\div$) at the gates. One can also study circuits that have access to "constants" other than 1; though typically one assumes that this set is fixed and independent of the input size n. Finally, as in the boolean case, if each gate has outdegree 1, we call it an *arithmetic formula*.

A gate's operation consists of performing the operation it is labeled with on the numbers present on the incoming wires, and then passing this output to all its outgoing wires. After each gate has performed its operation, an output appears on the circuit's lone output node. Thus the circuit may be viewed as a computing a function $f(x_1, x_2, \ldots, x_n)$ of the input variables, and simple induction shows that this output function is a (multivariate) polynomial in $x_1, x_2, \ldots, x_n$. If we allow gates to also be labeled with the division operation (denoted "$\div$") then the function is a rational function of $x_1, \ldots, x_n$, in other words, functions of the type $f_1(x_1, x_2, \ldots, x_n)/f_2(x_1, \ldots, x_n)$ where f_1, f_2 are polynomials. Of course, if the inputs come from a field such as $\mathbb{R}$, then rational functions can be used to approximate —via Taylor series expansion —all "smooth" real-valued functions.

As usual, we are interested in the asymptotic size (as a function of n) of the smallest family of algebraic circuits that computes a family of polynomials $\{f_n\}$ where f_n is a polynomial in n variables. The exercises ask you to show that circuits over $GF(2)$ (with no $\div$) are equivalent to boolean circuits, and the same is true for circuits over any finite field. So the case when F is infinite is usually of greatest interest.

EXAMPLE 14.2
The *discrete fourier transform* of a vector $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$ where $a_i \in \mathbf{C}$ is vector $M \cdot \mathbf{a}$, where M is a fixed $n \times n$ matrix whose (i, j) entry is ω^{ij} where ω is an nth root of 1 (in other words, a complex number satisfying $\omega^n = 1$).

Interpreting the trivial algorithm for matrix-vector product as an arithmetic circuit, one obtains an algebraic formula of size $O(n^2)$. Using the famous *fast fourier transform* algorithm, one can obtain a smaller circuit (or formula??; CHECK) of size $O(n \log n)$.

STATUS OF LOWERBOUNDS??

EXAMPLE 14.3
The *determinant* of an $n \times n$ matrix $X = (X_{ij})$ is

$$det(X) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} x_{i\sigma(i)}, \tag{1}$$

where S_n is the set of all $n!$ permutations on $\{1, 2, \ldots, n\}$. This can be computed using the familiar Gaussian elimination algorithm. Interpreting the algorithm as a circuit one obtains an arithmetic circuit of size $O(n^3)$. Using the $\mathbf{NC}^2$ algorithm for Gaussian elimination, one obtains an arithmetic formula of size $2^{O(\log^2 n)}$. No matching lowerbounds are known for either upperbound.

The previous example is a good illustration of how the polynomial defining a function may have exponentially many terms —in this case $n!$—but nevertheless be computable with a polynomial-size circuit (as well as a subexponential-size formula).

By contrast, no polynomial-size algebraic circuit is conjectured to exist for the *permanent* function, which at first sight seems is very similar to the determinant but as we saw in Section **??**, is #$\mathbf{P}$-complete.

$$permanent(X) = \sum_{\sigma \in S_n} (-1)^{sgn(\sigma)} \prod_{i=1}^{n} x_{i\sigma(i)}, \tag{2}$$

The determinant and permanent functions also play a vital role in the world of algebraic circuits, since they are *complete* problems for two important classes. To give the definition, we need the notion of *degree* of a multivariate polynomial, namely, the minimum d such that each monomial term $\prod_i x_i^{d_i}$ satisfies $\sum_i d_i \leq d$. A family of polynomials in $x_1, x_2, \ldots, x_n$ is *poly-bounded* if the degree is at most $O(n^c)$ for some constant $c > 0$.

DEFINITION 14.4 (*Alg*$\mathbf{P}$)
The class *Alg*$\mathbf{P}$ is the class of polynomials of polynomial degree that are computable by arithmetic formulae (using no $\div$) of polynomial size.

DEFINITION 14.5 (ALG$\mathbf{NP}$)
Alg$\mathbf{NP}$ is the class of polynomials of polynomial degree that are definable as

$$f(x_1, x_2, \ldots, x_n) = \sum_{e \in \{0,1\}^{m-n}} g_n(x_1, x_2, \ldots, x_n, e_{n+1}, \ldots, e_m),$$

where $g_n \in Alg\mathbf{P}$ and m is polynomial in n.

DEFINITION 14.6 (PROJECTION REDUCTION)
A function $f(x_1, \ldots, x_n)$ is a *projection* of a function $g(y_1, y_2, \ldots, y_m)$ if there is a mapping σ from $\{y_1, y_2, \ldots, y_m\}$ to $\{0, 1, x_1, x_2, \ldots, x_n\}$ such that $f(x_1, x_2, \ldots, x_n) = g(\sigma(y_1), \sigma(y_2), \ldots, \sigma(y_m))$.
   We say that f is *projection-reducible* to g if f is a projection of g.

THEOREM 14.7 (VALIANT)
Every polynomial on n variables that is computable by a circuit of size u is projection reducible to the Determinant function (over the same field) on $u + 2$ variables.

   *Every function in Alg**NP** is projection reducible to the Permanent function (over the same field).*

14.2 Algebraic Computation Trees

An algebraic computation tree is reminiscent of a boolean decision tree (Chapter **??**) but it computes a boolean-valued function $f : \mathbb{R}^n \to \{0, 1\}$. Consider for example the ELEMENT DISTINCTNESS problem of deciding, given n numbers $x_1, x_2, \ldots, x_n$, whether any two of them are the same. To study it in the decision tree model, we might study it by thinking of the input as a matrix of size n^2 where the (i, j) entry indicates whether or or not $x_i > x_j$ or $x_i = x_j$ or $x_i < x_j$. But one can also study it as a problem whose input is a vector of n real numbers. Consider the trivial algorithm in either viewpoint: sort the numbers in $O(n \log n)$ time and then check if any two adjacent numbers in the sorted order are the same. Is this trivial algorithm actually optimal? This question is still open, but one can prove optimality with respect to a more restricted class of algorithms that includes the above trivial algorithm.

   Recall that comparison-based sorting algorithms only ask questions of the type "Is $x_i > x_j$?", which is the same as asking whether $x_i - x_j > 0$. The left hand side term of this last inequality is a linear function. Other algorithms may use more complicated functions. In this section we consider a model called *Algebraic Computation Trees*, where we examine the effect of allowing a) the use of any polynomial function and b) the introduction of new variables together with the ability to ask questions about them.

DEFINITION 14.8 (ALGEBRAIC COMPUTATION TREE)
An *Algebraic Computation Tree* is a way to represent a function $f : \Re^n \to \{0, 1\}$ by showing how to compute $f(x_1, x_2, \ldots, x_n)$ for any input vector $(x_1, x_2, \ldots, x_n)$. It is a complete binary tree that describes where each of the nodes has one of the following types:

- Leaf labeled "Accept" or "Reject".

- Computation node v labeled with y_v, where $y_v = y_u \circ y_w$ and y_u, y_w are either one of $\{x_1, x_2, \ldots, x_n\}$ or the labels of ancestor nodes and the operator $\circ$ is in $\{+, -, \times, \div, \sqrt{\ }\}$.

- Branch node with out-degree 2. The branch that is taken depends on the evaluation of some condition of the type $y_u = 0$ or $y_u \geq 0$ or $y_u \leq 0$ where y_u is either one of $\{x_1, x_2, \ldots, x_n\}$ or the labels of an ancestor node in the tree.

Figure unavailable in pdf file.

Figure 14.1: An Algebraic Computation Tree

Figure unavailable in pdf file.

Figure 14.2: A computation path p of length d defines a set of constraints over the n input variables x_i and d additional variables y_j, which correspond to the nodes on p.

The computation on any input $(x_1, x_2, \ldots, x_n)$ follows a single path the root to a leaf, evaluating functions at internal nodes (including branch nodes) in the obvious way. The complexity of the computation on the path is measured using the following costs (which reflect real-life costs to some degree):

- $+, -$ are free.

- $\times, \div, \sqrt{\ }$ are charged unit cost.

The *depth* of the tree is the maximum cost of any path in it.

A fragment of an algebraic decision tree is shown in figure 14.1. The following examples illustrate some of the languages (over real numbers) whose complexity we want to study.

EXAMPLE 14.9
[Element Distinctness Problem] Given n numbers $x_1, x_2, \ldots, x_n$ we need to determine whether they are all distinct. This is equivalent to the question whether $\prod_{i \neq j}(x_i - x_j) \neq 0$. As indicated earlier, this can be computed by a tree of depth $O(n \log n)$ whose internal nodes only compute functions of the type $x_i - x_j$.

EXAMPLE 14.10
[Real number version of SUBSET SUM] Given a set of n real numbers $X = \{x_1, x_2, \ldots, x_n\}$ we ask whether there is a subset $S \subseteq X$ such that $\sum_{i \in S} x_i = 1$.

Of course, a tree of depth d could have 2^d nodes, so a small depth decision tree does not always guarantee an efficient algorithm. This is why the following theorem (which we do not prove) does not have any implication for **P** versus **NP**.

THEOREM 14.11
The real number version of SUBSET SUM *can be solved using an algebraic computation tree of depth* $O(n^5)$.

This theorem suggests that Algebraic Computation Trees are best used to investigate lower-bounds such as $n \log n$ or n^2. To prove lowerbounds for a function f, we will use the *topology* of the sets $f^{-1}(1)$ and $f^{-1}(0)$, specifically, the number of connected components. In fact, we will think of any function $f : \mathbb{R}^n \to \mathbb{R}$ as being defined by a subset $W \subseteq \mathbb{R}^n$, where $W = f^{-1}(1)$.

DEFINITION 14.12
Let $W \subseteq \mathbb{R}^n$. The *algebraic computation tree complexity* of W is

$$C(W) = \min_{\substack{\text{computation} \\ \text{trees } C \text{ for } W}} \{\text{depth of } C\}$$

DEFINITION 14.13 (CONNECTED COMPONENTS)
A set $S \subseteq \mathbb{R}^n$ is *connected* if for all $x, y \in S$ there is path p that connects x and y and lies entirely in S. For $S \subseteq \mathbb{R}^n$ we define $\#(S)$ to be the number of connected components of S.

THEOREM 14.14
Let $W = \{(x_1, \ldots, x_n) | \prod_{i \neq j} (x_i - x_j) \neq 0\}$. Then,

$$\#(W) \geq n!$$

PROOF: For each permutation σ let

$$W_\sigma = \{(x_1, \ldots, x_n) \mid x_{\sigma(1)} < x_{\sigma(2)} < \ldots < x_{\sigma(n)}\}.$$

That is, let W_σ be the set of n-tuples $(x_1, \ldots, x_n)$ to which σ gives order. It suffices to prove for all $\sigma' \neq \sigma$ that the sets W_σ and $W_{\sigma'}$ are not connected.

For any two distinct permutations σ and σ', there exist two distinct i, j with $1 \leq i, j \leq n$, such that $\sigma^{-1}(i) < \sigma^{-1}(j)$ but $\sigma^{-1}(i) > \sigma^{-1}(j)$. Thus, in W_σ we have $X_j - X_i > 0$ while in $W_{\sigma'}$ we have $X_i - X_j > 0$. Consider any path from W_σ to $W_{\sigma'}$. Since $X_j - X_i$ has different signs at the endpoints, the intermediate value principle says that somewhere along the path this term must become 0. Definition 14.13 then implies that W_σ and $W_{\sigma'}$ cannot be connected. ∎

The connection between the two parameters we have defined thus far is the following theorem, whose proof will use a fundamental theorem of topology. It also implies, using our observation above, that the algebraic computation tree complexity of ELEMENT DISTINCTNESS is $\Omega(\log(n!)) = \Omega(n \log n)$.

THEOREM 14.15 (BEN-OR)

$$C(W) = \Omega\Big(\log\left(\max\left\{\#(W), \#(\mathbb{R}^n - W)\right\}\right) - n \Big)$$

This theorem is proved in two steps. First, we try to identify the property of functions with low decision tree complexity: they can be defined using a "few" systems of equations.

LEMMA 14.16
If $f: \Re^n \to \{0, 1\}$ has a decision tree of depth d then $f^{-1}(1)$ (and also $f^{-1}(0)$) is a union of at most 2^d sets $C_1, C_2, \ldots$, where C_i is the set of solutions to some algebraic system of up to d equations of the type

$$p_i(y_1, \ldots, y_d, x_1, \ldots, x_n) \bowtie 0,$$

where p_i for $i \leq d$ is a degree 2 polynomial, $\bowtie$ is in $\{\leq, \geq, =, \neq\}$, and $y_1, \ldots, y_d$ are new variables.

(Rabinovitch's Trick) Additionally, we may assume without loss of generality (at the cost of doubling the number of y_i's) that there are no $\neq$ constraints in this system of equations.

PROOF: The tree has 2^d leaves, so it suffices to associate a set with each leaf. This is simply the set of $(x_1, x_2, \ldots, x_n)$ that end up at that leaf. Associate variables $y_1, y_2, \ldots, y_d$ with the d tree nodes appearing along the path from root to that leaf. For each tree nodes associate an equation with it in the obvious way (see figure 14.2). For example, if the node computes $y_v = y_u \div y_w$ then it implies the constraint $y_v y_w - y_u = 0$. Thus any $(x_1, x_2, \ldots, x_n)$ that end up at the leaf is a vector with an associated value of $y_1, y_2, \ldots, y_d$ such that the combined vector is a solution to these d equations.

To replace the "$\neq$" constraints with "$=$" constraints we take a constraint like

$$p_i(y_1, \ldots, y_m) \neq 0,$$

introduce a new variable z_i and impose the constraint

$$q_i(y_1, \ldots, y_m, z_i) \equiv 1 - z_i p_i(y_1, \ldots, y_m) = 0.$$

(This transformation holds for all fields.) Notice, the maximum degree of the constraint remains 2, because the trick is used only for the branch $y_u \neq 0$ which is converted to $1 - z_v y_u = 0$. ∎

REMARK 14.17
We find Rabinovitch's trick useful also in Section 14.3.2 where we prove a completeness result for Hilbert's Nullstellensatz.

Another version of the trick is to add the constraint

$$p_i^2(y_1, \ldots, y_m) > 0,$$

which doubles the degree and does not hold for all fields (e.g., the complex numbers).

Thus we need some result about the number of connected components of the set of solutions to an algebraic system. The following is a central result in mathematics.

THEOREM 14.18 (SIMPLE CONSEQUENCE OF MILNOR-THOM)
If $S \subseteq \mathbb{R}^n$ is defined by degree d constraints with m equalities and h inequalities then

$$\#(S) \leq d(2d - 1)^{n+h-1}$$

REMARK 14.19
Note that the above upperbound is independent of m.

Figure unavailable in pdf file.

Figure 14.3: Projection can merge but not add connected components

Now we can prove Ben-Or's Theorem.

PROOF: (Theorem 14.15) Suppose that the depth of a computation tree for W is d, so that there are at most 2^d leaves. We will use the fact that if $S \subseteq \mathbb{R}^n$ and $S|_k$ is the set of points in S with their $n - k$ coordinates removed (projection on the first k coordinates) then $\#(S|_k) \leq \#(S)$ (figure 14.3).

For every leaf there is a set of degree 2 constraints. So, consider a leaf ℓ and the corresponding constraints $\mathcal{C}_\ell$, which are in variables $x_1, \ldots, x_n, y_1, \ldots, y_d$. Let $W_\ell \subseteq \mathbb{R}^n$ be the subset of inputs that reach ℓ and $S_\ell \subseteq \mathbb{R}^{n+d}$ the set of points that satisfy the constraints $\mathcal{C}_\ell$. Note that $W_\ell = \mathcal{C}_\ell|_n$ i.e., W_ℓ is the projection of $\mathcal{C}_\ell$ onto the first n coordinates. So, the number of connected components in W_ℓ is upperbounded by $\#(\mathcal{C}_\ell)$. By Theorem 14.18 $\#(\mathcal{C}_\ell) \leq 2 \cdot 3^{n+d-1} \leq 3^{n+d}$. Therefore the total number of connected components is at most $2^d 3^{n+d}$, so $d \geq \log(\#(W)) - O(n)$. By repeating the same argument for $\mathbb{R}^n - W$ we have that $d \geq \log(\#(\mathbb{R}^n - W)) - O(n)$. ∎

14.3   The Blum-Shub-Smale Model

Blum, Shub and Smale introduced Turing Machines that compute over some arbitrary field K (e.g., $K = \mathbb{R}, \mathbf{C}, \mathbb{Z}_2$). This is a generalization of the standard Turing Machine model which operates over the ring $\mathbb{Z}_2$. Each cell can hold an element of K, Initially, all but a finite number of cells are "blank." In our standard model of the TM, the computation and branch operations can be executed in the same step. Here we perform these operations separately. So we divide the set of states into the following three categories:

- Shift state: move the head to the left or to the right of the current position.

- Branch state: if the content of the current cell is a then goto state q_1 else goto state q_2.

- Computation state: replace the contents of the current cell with a new value. The machine has a hardwired function f and the new contents of the cell become $a \leftarrow f(a)$. In the standard model for rings, f is a polynomial over K, while for fields f is a rational function p/q where p, q are polynomials in $K[x]$ and $q \neq 0$. In either case, f can be represented using a constant number of elements of K.

- The machine has a single "register" onto which it can copy the contents of the cell currently under the head. This register's contents can be used in the computation.

In the next section we define some complexity classes related to the BSS model. As usual, the time and space complexity of these Turing Machines is defined with respect to the input size, which is the number of cells occupied by the input.

REMARK 14.20
The following examples show that some modifications of the BSS model can increase significantly the power of an algebraic Turing Machine.

- If we allow the branch states to check, for arbitrary real number a, whether $a > 0$ (in other words, with arbitrary precision) the model becomes unrealistic because it can decide problems that are undecidable on the normal Turing machine. In particular, such a machine can compute $\mathbf{P}/poly$ in polynomial time; see Exercises. (Recall that we showed that $\mathbf{P}/poly$ contains undecidable languages.) If a language is in $\mathbf{P}/poly$ we can represent its circuit family by a single real number hardwired into the Turing machine (specifically, as the coefficient of of some polynomial $p(x)$ belonging to a state). The individual bits of this coefficient can be accessed by dividing by 2, so the machine can extract the polynomial length encoding of each circuit. Without this ability we can prove that the individual bits cannot be accessed.

- If we allow rounding (computation of $\lfloor x \rfloor$) then it is possible to factor integers in polynomial time, using some ideas of Shamir. (See exercises.)

Even without these modifications, the BSS model seems more powerful than real-world computers: Consider the execution of the operation $x \leftarrow x^2$ for n times. Since we allow each cell to store a real number, the Turing machine can compute and store in one cell (without overflow) the number x^{2^n} in n steps.

14.3.1   Complexity Classes over the Complex Numbers

Now we define the corresponding to $\mathbf{P}$ and $\mathbf{NP}$ complexity classes over $\mathbf{C}$:

DEFINITION 14.21 ($\mathbf{P_C}$,$\mathbf{NP_C}$)
$\mathbf{P_C}$ is the set of languages that can be decided by a Turing Machine over $\mathbf{C}$ in polynomial time. $\mathbf{NP_C}$ is the set of languages L for which there exists a language L_0 in $\mathbf{P_C}$, such that an input x is in L iff there exists a string $(y_1, \ldots, y_{n^c})$ in $\mathbf{C}^{n^c}$ such that (x, y) is in L_0.

The following definition is a restriction on the inputs of a TM over $\mathbf{C}$. These classes are useful because they help us understand the relation between algebraic and binary complexity classes.

DEFINITION 14.22 (0-1-$\mathbf{NP_C}$)

$$0\text{-}1\text{-}\mathbf{NP_C} = \{L \cap \{0,1\}^* \mid L \in \mathbf{NP_C}\}$$

Note that the input for an $\mathbf{NP_C}$ machine is binary but the nondeterministic "witness" may consist of complex numbers. Trivially, 3SAT is in 0-1-$\mathbf{NP}_C$: even though the "witness" consists of a string of complex numbers, the machine first checks if they are all 0 or 1 using equality checks. Having verified that the guess represents a boolean assignment to the variables, the machine continues as a normal Turing Machine to verify that the assignment satisfies the formula.

It is known that 0-1-$\mathbf{NP_C} \subseteq \mathbf{PSPACE}$. In 1997 Koiran proved that if one assumes the Riemann hypothesis, then 0-1-$\mathbf{NP_C} \subseteq \mathbf{AM}[2]$. Recall that $\mathbf{AM}[2]$ is $\mathbf{BP} \cdot \mathbf{NP}$ so Koiran's result suggests that 0-1-$\mathbf{NP_C}$ may not be much bigger than $\mathbf{NP}$.

Figure unavailable in pdf file.

Figure 14.4: Tableau of Turing Machine configurations

14.3.2 Hilbert's Nullstellensatz

The language $\mathbf{HN_C}$ is defined as the decision version of Hilbert's Nullstellensatz over $\mathbf{C}$. The input consists of m polynomials p_i of degree d over $x_1, \ldots, x_n$. The output is "yes" iff the polynomials have a common root $a_1, \ldots, a_n$. Note that this problem is general enough to include SAT. We illustrate that by the following example:

$$x \vee y \vee z \leftrightarrow (1-x)(1-y)(1-z) = 0.$$

Next we use this fact to prove that the language $0\text{-}1\text{-}\mathbf{HN_C}$ (where the polynomials have 0-1 coefficients) is complete for $0\text{-}1\text{-}\mathbf{NP_C}$.

THEOREM 14.23 (BSS)
$0\text{-}1\text{-}\mathbf{HN_C}$ is complete for $0\text{-}1\text{-}\mathbf{NP_C}$.

PROOF: (Sketch) It is straightforward to verify that $0\text{-}1\text{-}\mathbf{HN_C}$ is in $0\text{-}1\text{-}\mathbf{NP_C}$. To prove the hardness part we imitate the proof of the Cook-Levin theorem; we create a computation tableau and show that the verification is in $0\text{-}1\text{-}\mathbf{HN_C}$.

To that end, consider the usual computation tableau of a Turing Machine over $\mathbf{C}$ and as in the case of the standard Turing Machines express the fact that the tableau is valid by verifying all the 2×3 windows, i.e., it is sufficient to perform local checks (Figure 14.4). Reasoning as in the case of algebraic computation trees (see Lemma 14.16) we can express these local checks with polynomial constraints of bounded degree. The computation states $c \leftarrow q(a,b)/r(a,b)$ are easily handled by setting $p(c) \equiv q(a,b) - cr(a,b)$. For the branch states $p(a,b) \neq 0$ we can use Rabinovitch's trick to convert them to equality checks $q(a,b,z) = 0$. Thus the degree of our constraints depends upon the degree of the polynomials hardwired into the machine. Also, the polynomial constraints use real coefficients (involving real numbers hardwired into the machine). Converting these polynomial constraints to use only 0 and 1 as coefficients requires work. The idea is to show that the real numbers hardwired into the machine have no effect since the input is a binary string. We omit this mathematical argument here. ∎

14.3.3 Decidability Questions: Mandelbrot Set

Since the Blum-Shub-Smale model is more powerful than the ordinary Turing Machine, it makes sense to revisit decidability questions. In this section we show that some problems do indeed remain undecidable. We study the decidability of the Mandelbrot set with respect to Turing Machines over $\mathbf{C}$. Roger Penrose had raised this question in his meditation regarding artificial intelligence.

DEFINITION 14.24 (MANDELBROT SET DECISION PROBLEM)
Let $P_C(Z) = Z^2 + C$. Then, the Mandelbrot set is defined as

$$\mathcal{M} = \{C \mid \text{the sequence } P_C(0), P_C(P_C(0)), P_C(P_C(P_C(0)))\ldots \text{ is bounded }\}.$$

Note that the complement of $\mathcal{M}$ is recognizable if we allow inequality constraints. This is because the sequence is unbounded iff some number $P_C^k(0)$ has complex magnitude greater than 2 for some k (exercise!) and this can be detected in finite time. However, detecting that $P_C^k(0)$ is bounded for every k seems harder. Indeed, we have:

THEOREM 14.25
$\mathcal{M}$ is undecidable by a machine over $\mathbf{C}$.

PROOF: (Sketch) The proof uses the topology of the Mandelbrot set. Let $\mathcal{M}$ be any TM over the complex numbers that supposedly decides this set. Consider T steps of the computation of this TM. Reasoning as in Theorem 14.23 and in our theorems about algebraic computation trees, we conclude that the sets of inputs accepted in T steps is a finite union of semialgebraic sets (i.e., sets defined using solutions to a system of polynomial equations). Hence the language accepted by $\mathcal{M}$ is a countable union of semi-algebraic sets, which implies that its Hausdorft dimension is 1. But it is known Mandelbrot set has Hausdorff dimension 2, hence $\mathcal{M}$ cannot decide it. $\blacksquare$

Exercises

§1 Show that if field F is finite then arithmetic circuits have exactly the same power —up to constant factors—as boolean circuits.

§2 Equivalence of circuits of depth d to straight line programs of size $\exp(d)$. (Lecture 19 in Madhu's notes.)

§3 Bauer-Strassen lemma?

§4 If function computed in time T on algebraic TM then it has algebraic computation tree of depth $O(d)$.

§5 Prove that if we give the BSS model (over $\mathbb{R}$) the power to test "$a > 0$?" with arbitrary precision, then all of $\mathbf{P}/poly$ can be decided in polynomial time. (Hint: the machine's "program" can contain a constant number of arbitrary real numbers.)

§6 Shamir's trick?

Chapter notes and history

NEEDS A LOT
$\quad$ General reference on algebraic complexity

P. Brgisser, M. Clausen, and M. A. Shokrollahi, Algebraic complexity theory, Springer-Verlag, 1997.

Best reference on BSS model

Blum Cucker Shub Smale.

Algebraic P and NP from Valiant 81 and Skyum-Valiant'86.

Roger Penrose: emperor's new mind.

Mandelbrot : fractals.

DRAFT

Part III

Advanced topics

Chapter 15

Average Case Complexity: Levin's Theory

[1]

NEEDS MORE WORK

Our study of complexity —- **NP**-completeness, #**P**-completeness etc.— thus far only concerned worst-case complexity. However, algorithms designers have tried to design efficient algorithms for NP-hard problems that work for "many" or "most" instances. This motivates a study of the difficulty of the "average" instance. Let us first examine the issues at an intuitive level, so we may be better prepared for the elements of the theory we will develop.

Many average case algorithms are targeted at graph problems in *random graphs*. One can define random graphs in many ways: the simplest one generates a graph on n vertices randomly by picking each potential edge with probability $1/2$. (This method ends up assigning equal probability to every n-vertex graph.) On such rand om graphs, many **NP**-complete problems are easy. 3-COLOR can be solved in linear time with high probability (exercise). CLIQUE and INDEPENDENT SET can be solved in $n^{2\log n}$ time (exercise) which is only a little more than polynomial and much less than $2^{\epsilon n}$, the running time of the best algorithms on worst-case instances.

However, other NP-complete problems appear to require exponential time even on average. One example is SUBSET SUM: we pick n integers $a_1, a_2, \ldots, a_n$ randomly from $[1, 2^n]$, pick a random subset S of $\{1, \ldots, n\}$, and produce $b = \sum_{i \in S} a_i$. We do not know of any efficient average-case algorithm that, given the a_i's and b, finds S. Surprisingly, efficient algorithms do exist if the a_i's are picked randomly from the slightly larger interval $[1, 2^{n\log^2 n}]$. This illustrates an important point, namely, that average-case complexity is sensitive to the choice of the input distribution.

The above discussion suggests that even though **NP**-complete problems are essentially equivalent with respect to worst case complexity, they may differ vastly in their average case complexity. Can we nevertheless identify some problems that remain "complete" even for the average case; in other words, are at least as hard as every other average-case **NP** problem?

This chapter covers Levin's theory of average-case complexity. We will formalize the notion of "distributional problems," introduce a working definition of "algorithms that are efficient on

---

[1]This chapter written with Luca Trevisan

DRAFT

NOTE 15.1 (IMPAGLIAZZO'S POSSIBLE WORLDS)
At the moment we don't know if the best algorithm for 3SAT runs in time
$O(n)$ or $2^{\Omega(n)}$ but there are also many other *qualitative* open questions about
the hardness of problems in **NP**. Russell Impagliazzo characterized a central
goal of complexity theory as the question of finding out which of the following
possible worlds is the world we live in:

Algorithmica.

Heuristica.

Pessiland.

Minicrypt.

---

average," and define a reduction that preserves efficient average-case solvability. We will also
exhibit an **NP**-complete problem that is complete with respect to such reductions. However, we
cannot yet prove the completeness of natural distributional problems such as SUBSET SUM or one
of the number theoretic problems described in the chapter on cryptography.

15.1   Distributional Problems

In our intuitive discussion of average case problems, we first fixed an input size n and then considered
the average running time of the algorithm when inputs of size n are chosen from a distribution. At
the back of our mind, we knew that complexity has to be measured asymptotically as a function of
n. To formalize this intuitive discussion, we will define distributions on all (infinitely many) inputs.

DEFINITION 15.2 (DISTRIBUTIONAL PROBLEM)
A *distributional problem is a pair $\langle L, \mathcal{D} \rangle$, where L is a decision problem and $\mathcal{D}$ is a distribution
over the set $\{0,1\}^*$ of possible inputs.*

EXAMPLE 15.3
We can define the "uniform distribution" to be one that assigns an input $x \in \{0,1\}^*$ the probability

$$\Pr[x] = \frac{1}{|x|\,(1+|x|)} 2^{-|x|}. \tag{1}$$

We call this "uniform" because it assigns equal probabilities to all strings with the same length.
rect??   It is a valid distribution because the probabilities sum to 1:

$$\sum_{x \in \{0,1\}^*} \frac{1}{|x| \, (1 + |x|)} 2^{-|x|} = \sum_{n \geq 0} 2^n \frac{2^{-n}}{n(n+1)} = 1. \tag{2}$$

Here is another distribution; the probabilities sum to 1 since $\sum_{n \geq 1} \frac{1}{n^2} = \pi^2/6$.

$$\Pr[x] = \frac{6}{\pi^2} \frac{2^{-|x|}}{|x|^2} \qquad \text{if } |x| \geq 1 \tag{3}$$

To pick a string from these distributions, we can first an input length n with the appropriate probability (for the distribution in (2), we pick n with probability $6/\pi^2 n^2$) and then pick x uniformly from inputs of length n. This uniform distribution corresponds to the intuitive approach to average case complexity discussed in the introduction. However, the full generality of Definition 15.2 will be useful later when we study nonuniform input distributions.

15.1.1 Formalizations of "real-life distributions."

Real-life problem instances arise out of the world around us (images that have to be understood, a building that has to be navigated by a robot, etc.), and the world does not spend a lot of time tailoring instances to be hard for our algorithm —arguably, the world is indifferent to our algorithm. One may formalize this indifference in terms of computational effort, by hypothesizing that the instances are produced by an efficient algorithm. We can formalize this in two ways.

Polynomial time computable distributions. Such distributions have an associated deterministic polynomial time machine that, given input x, can compute the *cumulative probability* $\mu_{\mathcal{D}}(x)$, where

$$\mu_{\mathcal{D}}(x) = \sum_{y \leq x} \Pr_{\mathcal{D}}[y] \tag{4}$$

Here $\Pr_{\mathcal{D}}[y]$ denotes the probability assigned to string y and $y \leq x$ means y either precedes x in lexicographic order or is equal to x. Denoting the lexicographic predecessor of x by $x - 1$, we have

$$\Pr_{\mathcal{D}}[x] = \mu_{\mathcal{D}}(x) - \mu_{\mathcal{D}}(x - 1), \tag{5}$$

which shows that if $\mu_{\mathcal{D}}$ is computable in polynomial time, then so is $\Pr_{\mathcal{D}}[x]$. The uniform distributions in (1) and (1) are polynomial time computable, as are many other distributions that are defined using explicit formulae.

Polynomial time samplable distributions. These distributions have an associated probabilistic polynomial time machine that can produce samples from the distribution. In other words, it outputs x with probability $\Pr_{\mathcal{D}}[x]$. The expected running time is polynomial in the length of the output $|x|$.

Many such samplable distributions are now known, and the sampling algorithm often uses Monte Carlo Markov Chain (MCMC) techniques.

If a distribution is polynomial time computable then we can efficiently produce samples from it. (Exercise.) However, if $\mathbf{P} \neq \mathbf{P}^{\#\mathbf{P}}$ there are polynomial time samplable distributions (including some very interesting ones) that are not polynomial time computable. (See exercises.)

In this lecture, we will restrict attention to distributional problems involving a polynomial time computable distribution. This may appear to a serious limitation, but with some work the results of this chapter can be generalized to samplable distributions.

15.2 DistNP and its complete problems

The following complexity class is at the heart of our study of average case complexity.

$$\text{dist } \mathbf{NP} = \{\langle L, \mathcal{D}\rangle : L \in NP, \mathcal{D} \text{ polynomial-time computable}\}. \tag{6}$$

Since the same NP language may have different complexity behavior with respect to two different input distributions (SUBSET SUM was cited earlier as an example), the definition wisely treats the two as distinct computational problems. Note that every problem mentioned in the introduction to the chapter is in dist $\mathbf{NP}$.

Now we need to define the average-case analogue of $\mathbf{P}$.

15.2.1 Polynomial-Time on Average

Now we define what it means for a deterministic algorithm A to solve a distributional problem $\langle L, \mathcal{D}\rangle$ in polynomial time on average. The definition should be *robust* to simple changes in model of computation or representation. If we migrate the algorithm to a slower machine that has a quadratic slowdown (so t steps now take t^2 time), then polynomial-time algorithms should not suddenly turn into exponential-time algorithms. (This migration to a slower machine is not merely hypothetical, but also one way to look at a reduction.) As we will see, some intuitively appealing definitions do not have this robustness property.

Denote by $t(x)$ the running time of A on input x. First, note that $\mathcal{D}$ is a distribution on all possible inputs. The most intuitive choice of saying that A is efficient if

$$\mathbf{E}[t(x)] \text{ is small}$$

is problematic because the expectation could be infinite even if A runs in worst-case polynomial time.

Next, we could try to define A to be polynomial provided that for some constant c and for every sufficiently large n,

$$\mathbf{E}[t(x)|\ |x| = n] \leq n^c$$

This has two problems. First, it ignores the possibility that there could be input lengths on which A takes a long time, but that are generated with very low probability under $\mathcal{D}$. In such cases A may still be regarded as efficient, but the definition ignores this possibility. Second, and

more seriously, the definition is not robust to changes in computational model. To give an example, suppose $\mathcal{D}$ is the uniform distribution and $t(x_0) = 2^n$ for just one input x_0 of size n For every other input of size n, $t(x) = n$. Then $E[t(x) \mid |x| = n] \leq n + 1$. However, changing to a model with a quadratic slowdown will square all running times, and $E[(t(x))^2 \mid |x| = n] > 2^n$.

We could try to define A to be polynomial if there is a $c > 0$ such that

$$\mathbf{E}\left[\frac{t(x)}{|x|^c}\right] = O(1),$$

but this is also not robust. (Verify this!)

We now come to a satisfying definition.

DEFINITION 15.4 (POLYNOMIAL ON AVERAGE AND DIST $\mathbf{P}$)
A problem $\langle L, \mathcal{D} \rangle \in$ dist $\mathbf{NP}$ is said to be in dist $\mathbf{P}$ if there is an algorithm A for L that satisfies for some constants c, c_1

$$\mathbf{E}[\ \frac{t(x)^{1/c}}{|x|}\ ] = c_1, \tag{7}$$

where $t(x)$ is the running time of A on input x.

Notice that $\mathbf{P} \subseteq$ dist $\mathbf{P}$: if a language can be decided deterministically in time $t(x) = O(|x|^c)$, then $t(x)^{1/c} = O(|x|)$ and the expectation in (7) converges regardless of the distribution. Second, the definition is robust to changes in computational models: if the running times get squared, we just multiply c by 2 and the expectation in (7) again converges.

We also point out an additional interesting property of the definition: there is a high probability that the algorithm runs in polynomial time. For, if

$$\mathbf{E}[\frac{t(x)^{1/c}}{|x|}] = c_1, \tag{8}$$

then we have

$$\Pr[t(x) \geq k \cdot |x|^c] \quad = \ \Pr[\frac{t(x)^{1/c}}{|x|} \geq k^{1/c}] \quad \leq \ \frac{c_1}{k^{1/c}} \tag{9}$$

where the last claim follows by Markov's inequality. Thus by increasing k we may reduce this probability as much as required.

15.2.2 Reductions

Now we define reductions. Realize that we think of instances as being generated according to a distribution. Defining a mapping on strings (e.g., a reduction) gives rise to a new distribution on strings. The next definition formalizes this observation.

DEFINITION 15.5
If f is a function mapping strings to strings and $\mathcal{D}$ is a distribution then the distribution $f \circ \mathcal{D}$ is one that assigns to string y the probability $\sum_{x : f(x) = y} \Pr_{\mathcal{D}}[x]$

DEFINITION 15.6 (REDUCTION)
A distributional problem $\langle L_1, \mathcal{D}_1 \rangle$ *reduces* to a distributional problem $\langle L_2, \mathcal{D}_2 \rangle$ (denoted $\langle L_1, \mathcal{D}_1 \rangle \leq \langle L_2, \mathcal{D}_2 \rangle$) if there is a polynomial-time computable function f and an $\epsilon > 0$ such that:

1. $x \in L_1$ iff $f(x) \in L_2$.

2. For every x, $|f(x)| = \Omega(|x|^\epsilon)$.

3. There are constants c, c_1 such that for every string y,

$$\Pr_{f \circ \mathcal{D}_1}(y) \leq c_1 |y|^c \Pr_{\mathcal{D}_2}(y). \qquad \textbf{(Domination)}$$

The first condition is standard for many-to-one reductions, ensuring that a decision algorithm for L_2 easily converts into a decision algorithm for L_1. The second condition is a technical one, needed later. All interesting reductions we know of satisfy this condition. Next, we motivate the third condition, which says that $\mathcal{D}_2$ "dominates" (up to a polynomial factor) the distribution $f \circ \mathcal{D}_1$ obtained by applying f on $\mathcal{D}_1$.

Realize that the goal of the definition is to ensure that "if $(L_1, \mathcal{D}_1)$ is hard, then so is $(L_2, \mathcal{D}_2)$" (or equivalently, the contrapositive "if $(L_2, \mathcal{D}_2)$ is easy, then so is $(L_1, \mathcal{D}_1)$.") Thus if an algorithm A_2 is efficient for problem $(L_2, \mathcal{D}_2)$, then the following algorithm ought to be efficient for problem $(L_1, \mathcal{D}_1)$: on input x obtained from distribution $\mathcal{D}_1$, compute $f(x)$ and then run algorithm A_2 on $f(x)$. *A priori*, one cannot rule out the possibility that that A_2 is very slow on some inputs, which are unlikely to be sampled according to distribution $\mathcal{D}_2$ but which show up with high probability when we sample x according to $\mathcal{D}_1$ and then consider $f(x)$. The domination condition helps rule out this possibility.

In fact we have the following result, whose non-trivial proof we omit.

THEOREM 15.7
If $\langle L_1, \mathcal{D}_1 \rangle \leq \langle L_2, \mathcal{D}_2 \rangle$ and $\langle L_2, \mathcal{D}_2 \rangle$ has an algorithm that is polynomial on average, then $\langle L_1, \mathcal{D}_1 \rangle$ also has an algorithm that is polynomial on average.

Of course, Theorem 15.7 is useful only if we can find reductions between interesting problems. Now we show that this is the case: we exhibit a problem (albeit an artificial one) that is complete for dist **NP**. Let the inputs have the form $\langle M, x, 1^t, 1^l \rangle$, where M is an encoding of a Turing machine and 1^t is a sequence of t ones. Then we define the following "universal" problem U.

- Decide whether there exists a string y such that $|y| \leq l$ and $M(x, y)$ accepts in at most t steps.

Since part of the input is in unary, we need to modify our definition of a "uniform" distribution to the following.

$$\Pr_{\mathcal{D}}\left(\langle M, x, 1^t, 1^l \rangle\right) = \frac{1}{|M|(|M|+1)\, 2^{|M|}} \cdot \frac{1}{|x|(|x|+1)\, 2^{|x|}} \cdot \frac{1}{(t+l)(t+l+1)}. \qquad (10)$$

This distribution is polynomial-time computable (exercise).

THEOREM 15.8 (LEVIN)
$\langle U, \mathcal{D} \rangle$ is complete for dist **NP**, where $\mathcal{D}$ is the uniform ditribution.

The proof requires the following lemma, which shows that for polynomial-time computable distributions, we can apply a simple transformation on the inputs such that the resulting distribution has no "peaks" (i.e., no input has too high a probability).

LEMMA 15.9 (PEAK ELIMINATION)
Suppose $\mathcal{D}$ is a polynomial-time computable distribution over x. Then there is a polynomial-time computable function g such that

1. g is injective: $g(x) = g(z)$ iff $x = z$.

2. $|g(x)| \leq |x| + 1$.

3. For every string y, $\mathrm{Pr}_{g \circ \mathcal{D}}(y) \leq 2^{-|y|+1}$.

PROOF: For any string x such that $\mathrm{Pr}_{\mathcal{D}}(x) > 2^{-|x|}$, define $h(x)$ to be the largest common prefix of binary representations of $\mu_{\mathcal{D}}(x)$, $\mu_{\mathcal{D}}(x-1)$. Then h is polynomial-time computatable since $\mu_{\mathcal{D}}(x) - \mu_{\mathcal{D}}(x-1) = \mathrm{Pr}_{\mathcal{D}}(x) > 2^{-|x|}$, which implies that $\mu_{\mathcal{D}}(x)$ and $\mu_{\mathcal{D}}(x-1)$ must differ in the somewhere in the first $|x|$ bits. Thus $|h(x)| \leq \log 1/\mathrm{Pr}_{\mathcal{D}}(x) \leq |x|$. Furthermore, h is injective because only two binary strings s_1 and s_2 can have the longest common prefix z; a third string s_3 sharing z as a prefix must have a longer prefix with either s_1 or s_2.

Now define

$$g(x) = \begin{cases} 0x & \text{if } \mathrm{Pr}_{\mathcal{D}}(x) \leq 2^{-|x|} \\ 1h(x) & \text{otherwise} \end{cases} \tag{11}$$

Clearly, g is injective and satisfies $|g(x)| \leq |x| + 1$. We now show that $g \circ \mathcal{D}$ does not give probability more than $2^{-|y|+1}$ to any string y. If y is not $g(x)$ for any x, this is trivially true since $\mathrm{Pr}_{g \circ \mathcal{D}}(y) = 0$.

If $y = 0x$, where $\mathrm{Pr}_{\mathcal{D}}(x) \leq 2^{-|x|}$, then $\mathrm{Pr}_{g \circ \mathcal{D}}(y) \leq 2^{-|y|+1}$ and we also have nothing to prove.

Finally, if $y = g(x) = 1h(x)$ where $\mathrm{Pr}_{\mathcal{D}}(x) > 2^{-|x|}$, then as already noted, $|h(x)| \leq \log 1/\mathrm{Pr}_{\mathcal{D}}(x)$ and so $\mathrm{Pr}_{g \circ \mathcal{D}}(y) = \mathrm{Pr}_{\mathcal{D}}(x) \leq 2^{-|y|+1}$.

Thus the Lemma has been proved. ∎

Now we are ready to prove Theorem 15.8.

PROOF: (Theorem 15.8) At first sight the proof may seem trivial since U is just the "universal" decision problem for nondeterministic machines, and every **NP** language trivially reduces to it. However, we also need to worry about the input distributions and enforce the domination condition as required by Definition 15.6.

Let $\langle L, \mathcal{D}_1 \rangle \in$ dist **NP**. Let M be a proof-checker for language L that runs in time n^c; in other words, $x \in L$ iff there is a witness y of length $|y| = |x|^c$ such that $M(x, y) =$ Accept. (For notational ease we drop the big-O notation in this proof.) In order to define a reduction from L to

U, the first idea would be to map input x for L to $\left\langle M, x, 1^{|x|^c}, 1^{|x|^c} \right\rangle$. However, this may violate the domination condition because the uniform distribution assigns a probability $2^{-|x|}/\mathrm{poly}(|x|)$ to $\langle M, x, 1^{|x|^c} \rangle$ whereas x may have much higher probability under $\mathcal{D}_1$. Clearly, this difficulty arises only if the distribution $\mathcal{D}_1$ has a "peak" at x, so we see an opportunity to use Lemma 15.9, which gives us an injective mapping g such that $g \circ \mathcal{D}_1$ has no "peaks" and g is computable say in n^d time for some fixed constant d.

The reduction is as follows: map x to $\langle M', g(x), 1^{|x|^c + |x|}, 1^{|x|^c + |x|^d} \rangle$. Here M' is a modification of M that expects as input a string z and a witness (x, y) of length $|x| + |x|^c$. Given (z, x, y) where $y = |x|^c$, M' checks in $|x|^d$ time if $g(x) = z$. If so, it simulates M on (x, y) and outputs its answer. If $g(x) \neq z$ then M' rejects.

To check the domination condition, note that $y = \langle M', g(x), 1^{|x|^c + |x|}, 1^{|x|^c + |x|^d} \rangle$ has probability

$$
\begin{aligned}
\Pr_{\mathcal{D}}(y) &= \frac{2^{-|M'|}}{|M'|\,(|M'|+1)} \cdot \frac{2^{-|g(x)|}}{|g(x)|\,(|g(x)|+1)} \cdot \frac{1}{(|x| + 2\,|x|^c + |x|^d)(|x| + 2\,|x|^c + |x|^d + 1)} \\
&\leq \frac{2^{-|M'|}}{|M'|\,(|M'|+1)} \frac{1}{|x|^{2(c+d+1)}} \cdot 2^{-g(x)}
\end{aligned}
\tag{12}
$$

under the uniform distribution whereas

$$
\Pr_{\mathcal{D}_1}(x) \leq 2^{-g(x)+1} \leq G\,|x|^{2(c+d+1)} \Pr_{\mathcal{D}}(y) \, ,
$$

if we allow the constant G to absorb the term $2^{|M'|}\,|M'|\,(|M'|+1)$. Thus the domination condition is satisfied.

Notice, we rely crucially on the fact that $2^{|M'|}\,|M'|\,(|M'|+1)$ is a constant once we fix the language L; of course, this constant will usually be quite large for typical **NP** languages, and this would be a consideration in practice. ∎

15.2.3   Proofs using the simpler definitions

In the setting of one-way functions and in the study of the average-case complexity of the permanent and of problems in EXP (with applications to pseudorandomness), we normally interpret "average case hardness" in the following way: that an algorithm of limited running time will fail to solve the problem on a noticeable fraction of the input. Conversely, we would interpret average-case tractability as the existence of an algorithm that solves the problem in polynomial time, except on a negligible fraction of inputs. This leads to the following formal definition.

DEFINITION 15.10 (HEURISTIC POLYNOMIAL TIME)
We say that an algorithm A is a heuristic polynomial time algorithm for a distributional problem $\langle L, \mu \rangle$ if A always runs in polynomial time and for every polynomial p

$$
\sum_{x : A(x) \neq \chi_L(x)} \mu'(x) p(|x|) = O(1)
$$

In other words, a polynomial time algorithm for a distributional problem is a heuristic if the algorithm fails on a negligible fraction of inputs, that is, a subset of inputs whose probability mass is bounded even if multiplied by a polynomial in the input length. It might also make sense to consider a definition in which A is always correct, although it does not necessarily work in polynomial time, and that A is heuristic polynomial time if there is a polynomial q such that for every polynomial p, $\sum_{x \in S_q} \mu'(x)p(|x|) = O(1)$, where S_q is the set of inputs x such that $A(x)$ takes more than $q(|x|)$ time. Our definition is only more general, because from an algorithm A as before one can obtain an algorithm A satisfying Definition 15.10 by adding a clock that stops the computation after $q(|x|)$ steps.

The definition of heuristic polynomial time is *incomparable* with the definition of average polynomial time. For example, an algorithm could take time 2^n on a fraction $1/n^{\log n}$ of the inputs of length n, and time n^2 on the remaining inputs, and thus be a heuristic polynomial time algorithm with respect to the uniform distribution, while not beign average polynomial time with respect to the uniform distribution. On the other hand, consider an algorithm such that for every input length n, and for $1 \le k \le 2^{n/2}$, there is a fraction about $1/k^2$ of the inputs of length n on which the algorithm takes time $\Theta(kn)$. Then this algorithm satisfies the definition of average polynomial time under the uniform distribution, but if we impose a polynomial clock there will be an inverse polynomial fraction of inputs of each length on which the algorithm fails, and so the definition of heuristic polynomial time cannot be met.

It is easy to see that heuristic polynomial time is preserved under reductions.

THEOREM 15.11
If $\langle L_1, \mu_1 \rangle \le \langle L_2, \mu_2 \rangle$ and $\langle L_2, \mu_2 \rangle$ admits a heuristic polynomial time algorithm, then $\langle L_1, \mu_1 \rangle$ also admits a heuristic polynomial time algorithm.

PROOF: Let A_2 be the algorithm for $\langle L_2, \mu_2 \rangle$, let f be the function realizing the reduction, and let p be the polynomial witnessing the domination property of the reduction. Let c and ϵ be such that for every x we have $|x| \le c|f(x)|^{1/\epsilon}$.

Then we define the algorithm A_1 than on input x outputs $A_2(f(x))$. Clearly this is a polynomial time algorithm, and whenever A_2 is correct on $f(x)$, then A_1 is correct on x. We need to show that for every polynomial q

$$\sum_{x: A_2(f(x)) \neq \chi_{L_2}(f(x))} \mu'_1(x)q(|x|) = O(1)$$

and the left-hand side can be rewritten as

$$\sum_{y: A_2(y) \neq \chi_{L_2}(y)} \sum_{x: f(x)=y} \mu'_1(x)q(|x|)$$
$$\le \sum_{y: A_2(y) \neq \chi_{L_2}(y)} \sum_{x: f(x)=y} \mu'_1(x)q(c \cdot |y|^{1/\epsilon}))$$
$$\le \sum_{y: A_2(y) \neq \chi_{L_2}(y)} \mu'_2(y)p(|y|)q'(|y|)$$
$$= O(1)$$

where the last step uses the fact that A_2 is a polynomial heuristic for $\langle L_2, \mu_2 \rangle$ and in the second-to-last step we introduce the polynomial $q'(n)$ defined as $q(c \cdot n^{1/\epsilon})$

∎

15.3 Existence of Complete Problems

We now show that there exists a problem (albeit an artificial one) complete for dist **NP**. Let the inputs have the form $\langle M, x, 1^t, 1^l \rangle$, where M is an encoding of a Turing machine and 1^t is a sequence of t ones. Then we define the following "universal" problem U.

- Decide whether there exists a string y such that $|y| \leq l$ and $M(x, y)$ accepts in at most t steps.

That U is **NP**-complete follows directly from the definition. Recall the definition of **NP**: we say that $L \in \mathbf{NP}$ if there exists a machine M running in $t = \text{poly}(|x|)$ steps such that $x \in L$ iff there exists a y with $y = \text{poly}(|x|)$ such that $M(x, y)$ accepts. Thus, to reduce L to U we need only map x onto $R(x) = \langle M, x, 1^t, 1^l \rangle$ where t and l are sufficiently large bounds.

15.4 Polynomial-Time Samplability

DEFINITION 15.12 (SAMPLABLE DISTRIBUTIONS)
We say that a distribution μ is *polynomial-time samplable if there exists a probabilistic algorithm A, taking no input, that outputs x with probability $\mu\prime(x)$ and runs in* poly $(|x|)$ *time.*

Any polynomial-time computable distribution is also polynomial-time samplable, provided that for all x,

$$\mu\prime(x) \geq 2^{-\text{poly}(|x|)} \text{ or } \mu\prime(x) = 0. \tag{13}$$

For a polynomial-time computable μ satisfying the above property, we can indeed construct a sampler A that first chooses a real number r uniformly at random from $[0, 1]$, to poly $(|x|)$ bits of precision, and then uses binary search to find the first x such that $\mu(x) \geq r$.

On the other hand, under reasonable assumptions, there are efficiently samplable distributios μ that are not efficiently computable.

In addition to dist **NP**, we can look at the class

$$\langle \mathbf{NP}, \mathbf{P}\text{-samplable} \rangle = \{ \langle L, \mu \rangle : L \in \mathbf{NP}, \mu \text{ polynomial-time samplable} \}. \tag{14}$$

A result due to Impagliazzo and Levin states that if $\langle L, \mu \rangle$ is dist **NP**-complete, then $\langle L, \mu \rangle$ is also complete for the class $\langle \mathbf{NP}, \mathbf{P}\text{-samplable} \rangle$.

This means that the completeness result established in the previous section extends to the class of NP problems with samplable distributions.

Exercises

§1 Describe an algorithm that decides 3-colorability on almost all graphs in linear expected time.

Hint: A 3-colorable graph better not contain a complete graph on 4 vertices.

§2 Describe an algorithm that decides CLIQUE on almost all graphs in $n^{2\log n}$ time.

Hint: The chance that a random graph has a clique of size more than k is at most $\binom{n}{k}2^{-k^2/2}$.

§3 Show that if a distribution is polynomial-time computable, then it is polynomial-time sampleable.

Hint: Binary search.

§4 Show that if $\mathbf{P}^{\#\mathbf{P}} \neq \mathbf{P}$ then there is a polynomial time samplable distribution that is not polynomial time computable.

§5 Show that the function g defined in Lemma 15.9 (Peak Elimination) is efficiently invertible in the following sense: if $y = g(x)$, then given y we can reconstruct x in $|x|^{O(1)}$ time.

§6 Show that if one-way functions exist, then dist $\mathbf{NP} \nsubseteq$ dist $\mathbf{P}$.

Chapter notes and history

Suppose $\mathbf{P} \neq \mathbf{NP}$ and yet dist $\mathbf{NP} \subseteq$ dist $\mathbf{P}$. This would mean that generating hard instances of $\mathbf{NP}$ problems requires superpolynomial computations. Cryptography is thus impractical. Also, it seems to imply that everyday instances of $\mathbf{NP}$-complete problems would also be easily solvable. Such instances arise from the world around us —we want to understand an image, or removing the obstacles in the path of a robot— and it is hard to imagine how the inanimate world would do the huge amounts of computation necessary to generate a hard instance.

Chapter 16

Derandomization, Expanders and Extractors

"God does not play dice with the universe"
Albert Einstein

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."
John von Neumann, quoted by Knuth 1981

"How hard could it be to find hay in a haystack?"
Howard Karloff

The concept of a *randomized algorithm*, though widespread, has both a philosophical and a practical difficulty associated with it.

The philosophical difficulty is best represented by Einstein's famous quote above. Do random events (such as the unbiased coin flip assumed in our definition of a randomized turing machine) truly exist in the world, or is the world deterministic? The practical difficulty has to do with actually generating random bits, assuming they exist. A randomized algorithm running on a modern computer could need billions of random bits each second. Even if the world contains some randomness —say, the ups and downs of the stock market — it may not have enough randomness to provide billions of uncorrelated random bits every second in the tiny space inside a microprocessor. Current computing environments rely on shortcuts such as taking a small "fairly random looking" bit sequence—e.g., interval between the programmer's keystrokes measured in microseconds—and applying a deterministic generator to turn them into a longer sequence of "sort of random looking" bits. Some recent devices try to use quantum phenomena. But for all of them it is unclear how random and uncorrelated those bits really are.

Such philosophical and practical difficulties look deterring; the philosophical aspect alone has been on the philosophers' table for centuries. The results in the current chapter may be viewed as complexity theory's contribution to these questions.

The first contribution concerns the place of randomness in our world. We indicated in Chapter 7 that randomization seems to help us design more efficient algorithms. A surprising conclusion in this chapter is this could be a mirage to some extent. If certain plausible complexity-theoretic conjectures are true (e.g., that certain problems can not be solved by subexponential-sized circuits) then *every* probabilistic algorithm can be simulated deterministically with only a polynomial slow-down. In other words, randomized algorithms can be *derandomized* and $\mathbf{BPP} = \mathbf{P}$. Nisan and Wigderson [NW94] named this research area *Hardness versus Randomness* since the existence of *hard* problems is shown to imply derandomization. Section 16.3 shows that the converse is also true to a certain extent: ability to derandomize implies circuit lowerbounds (thus, hardness) for concrete problems. Thus the Hardness $\leftrightarrow$ Randomness connection is very real.

Is such a connection of any use at present, given that we have no idea how to prove circuit lowerbounds? Actually, yes. Just as in cryptography, we can use *conjectured* hard problems in the derandomization instead of *provable* hard problems, and end up with a win-win situation: if the conjectured hard problem is truly hard then the derandomization will be successful; and if the derandomization fails then it will lead us to an algorithm for the conjectured hard problem.

The second contribution of complexity theory concerns another practical question: how can we run randomized algorithms given only an imperfect source of randomness? We show the existence of *randomness extractors*: efficient algorithms to extract (uncorrelated, unbiased) random bits from any *weakly random* device. Their analysis is unconditional and uses no unproven assumptions. Below, we will give a precise definition of the properties that such a weakly random device needs to have. We do not resolve the question of whether such weakly random devices exist; this is presumably a subject for physics (or philosophy).

A central result in both areas is Nisan and Wigderson's beautiful construction of a certain *pseudorandom generator*. This generator is tailor-made for derandomization and has somewhat different properties than the *secure* pseudorandom generators we encountered in Chapter 10.

Another result in the chapter is a (unconditional) derandomization of randomized logspace computations, albeit at the cost of some increase in the space requirement.

Example 16.1 (Polynomial identity testing)
One example for an algorithm that we would like to derandomize is the algorithm described in Section 7.2.2 for testing if a given polynomial (represented in the form of an arithmetic zero) is the identically zero polynomial. If p is an n-variable nonzero polynomial of total degree d over a large enough finite field $\mathbb{F}$ ($|\mathbb{F}| > 10d$ will do) then most of the vectors $\mathbf{u} \in \mathbb{F}^n$ will satisfy $p(\mathbf{u}) \neq 0$ (see Lemma A.25. Therefore, checking whether $p \equiv 0$ can be done by simply choosing a random $\mathbf{u} \in_R \mathbb{F}^n$ and applying p on $\mathbf{u}$. In fact, it is easy to show that there exists a set of m^2-vectors $\mathbf{u}^1, \ldots, \mathbf{u}^{m^2}$ such that for *every* such nonzero polynomial p that can be computed by a size m arithmetic circuit, there exists an $i \in [m^2]$ for which $p(\mathbf{u}^i) \neq 0$.

This suggests a natural approach for a deterministic algorithm: show a deterministic algorithm that for every $m \in \mathbb{N}$, runs in poly(m) time and outputs a set $\mathbf{u}^1, \ldots, \mathbf{u}^{m^2}$ of vectors satisfying the above property. This shouldn't be too difficult— after all the vast majority of the sets of vectors

have this property, so hard can it be to find a single one? (Howard Karloff calls this task "finding a hay in a haystack"). Surprisingly this turns out to be quite hard: without using complexity assumptions, we do not know how to obtain such a set, and in Section 16.3 we will see that in fact such an algorithm will imply some nontrivial circuit lowerbounds.[1]

16.1 Pseudorandom Generators and Derandomization

The main tool in derandomization is a pseudorandom generator. This is a twist on the definition of a *secure* pseudorandom generator we gave in Chapter 10, with the difference that here we consider nonuniform distinguishers –in other words, circuits— and allow the generator to run in exponential time.

DEFINITION 16.2 (PSEUDORANDOM GENERATORS)
Let R be a distribution over $\{0,1\}^m$, $S \in \mathbb{N}$ and $\epsilon > 0$. We say that R is an (S, ϵ)-*pseudorandom distribution* if for every circuit C of size at most S,

$$|\Pr[C(R) = 1] - \Pr[C(U_m) = 1]| < \epsilon$$

where U_m denotes the uniform distribution over $\{0,1\}^m$.

If $S : \mathbb{N} \to \mathbb{N}$ is a polynomial-time computable monotone function (i.e., $S(m) \geq S(n)$ for $m \geq n$)[2] then a function $G : \{0,1\}^* \to \{0,1\}^*$ is called an $(S(\ell))$-*pseudorandom generator* (see Figure 16.1) if:

- For every $z \in \{0,1\}^\ell$, $|G(z)| = S(\ell)$ and $G(z)$ can be computed in time $2^{c\ell}$ for some constant c. We call the input z the *seed* of the pseudorandom generator.

- For every $\ell \in \mathbb{N}$, $G(U_\ell)$ is an $(S(\ell)^3, 1/10)$-pseudorandom distribution.

REMARK 16.3
The choices of the constant 3 and $1/10$ in the definition of an $S(\ell)$-pseudorandom generator are arbitrary and made for convenience.

The relation between pseudorandom generators and simulating probabilistic algorithm is straightforward:

---

[1] Perhaps it should not be so surprising that "finding a hay in a haystack" is so hard. After all, the hardest open problems of complexity— finding explicit functions with high circuit complexity— are of this form, since the vast majority of the functions from $\{0,1\}^n$ to $\{0,1\}$ have exponential circuit complexity.

[2] We place these easily satisfiable requirements on the function S to avoid weird cases such as generators whose output length is not computable or generators whose output shrinks as the input grows.
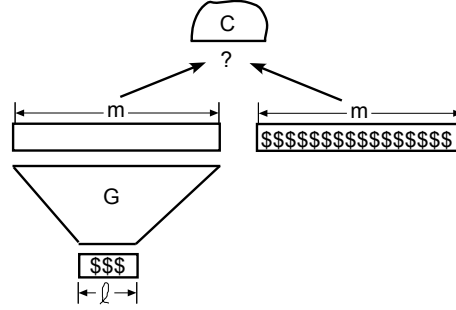
Figure 16.1: A *pseudorandom generator* G maps a short uniformly chosen seed $z \in_R \{0,1\}^{\ell}$ into a longer output $G(z) \in \{0,1\}^m$ that is indistinguishable from the uniform distribution U_m by any small circuit C.

LEMMA 16.4
Suppose that there exists an $S(\ell)$-pseudorandom generator for some polynomial-time computable monotone $S : \mathbb{N} \to \mathbb{N}$. Then for every polynomial-time computable function $\ell : \mathbb{N} \to \mathbb{N}$, $\mathbf{BPTIME}(S(\ell(n))) \subseteq \mathbf{DTIME}(2^{c\ell(n)})$ for some constant c.

PROOF: A language L is in $\mathbf{BPTIME}(S(\ell(n)))$ if there is an algorithm A that on input $x \in \{0,1\}^n$ runs in time $cS(\ell(n))$ for some constant c, and satisfies

$$\Pr_{r \in_R \{0,1\}^m}[A(x,r) = L(x)] \geq \frac{2}{3}$$

where $m \leq S(\ell(n))$ and we define $L(x) = 1$ if $x \in L$ and $L(x) = 0$ otherwise.

The main idea is that if we replace the truly random string r with the string $G(z)$ produced by picking a random $z \in \{0,1\}^{\ell(n)}$, then an algorithm like A that runs in only $S(\ell)$ time cannot detect this switch most of the time, and so the probability $2/3$ in the previous expression does not drop below $2/3 - 0.1$. Thus to derandomize A, we do not need to enumerate over all r; it suffices to enumerates over all $z \in \{0,1\}^{\ell(n)}$ and check how many of them make A accept. This derandomized algorithm runs in $\exp(\ell(n))$ time instead of the trivial 2^m time.

Now we make this formal. Our deterministic algorithm B will on input $x \in \{0,1\}^n$, go over all $z \in \{0,1\}^{\ell(n)}$, compute $A(x, G(z))$ and output the majority answer. Note this takes $2^{O(\ell(n))}$ time. We claim that for n sufficiently large, the fraction of z's such that $A(x, G(z)) = L(x)$ is at least $\frac{2}{3} - 0.1$. (This suffices to prove that $L \in \mathbf{DTIME}(2^{c\ell(n)})$ as we can "hardwire" into the algorithm the correct answer for finitely many inputs.)

Suppose this is false and there exists an infinite sequence of x's for which $\Pr[A(x, G(z)) = L(x)] < 2/3 - 0.1$. Then we would get a distinguisher for the pseudorandom generator —just use the Cook-Levin transformation to construct a circuit that computes the function $z \mapsto A(x, G(z))$, where x is hardwired into the circuit. This circuit has size $O(S(\ell(n)))^2$ which is smaller than $S(\ell(n))^3$ for sufficiently large n. ∎

REMARK 16.5
The proof shows why it is OK to allow the pseudorandom generator in Definition 16.2 to run in time exponential in its seed length. The derandomized algorithm enumerates over all possible seeds

of length ℓ, and thus would take exponential time (in ℓ) even if the generator itself were to run in less than exponential time.

Notice, these generators have to fool distinguishers that run for *less* time than they do. By contrast, the definition of *secure pseudorandom generators* (Definition 10.11 in Chapter 10) required the generator to run in polynomial time, and yet have the ability to fool distinguishers that have super-polynomial running time. This difference in these definitions stems from the intended usage. In the cryptographic setting the generator is used by honest users and the distinguisher is the adversary attacking the system — and it is reasonable to assume the attacker can invest more computational resources than those needed for normal/honest use of the system. In derandomization, generator is used by the derandomized algorithm, the "distinguisher" is the probabilistic algorithm that is being derandomized, and it is reasonable to allow the derandomized algorithm higher running time than the original probabilistic algorithm.

Of course, allowing the generator to run in exponential time as in this chapter potentially makes it easier to prove their existence compared with secure pseudorandom generators, and this indeed appears to be the case. (Note that if we place no upperbounds on the generator's efficiency, we could prove the existence of generators *unconditionally* as shown in Exercise 2, but these do not suffice for derandomization.)

We will construct pseudorandom generators based on complexity assumptions, using quantitatively stronger assumptions to obtain quantitatively stronger pseudorandom generators (i.e., $S(\ell)$-pseudorandom generators for larger functions S). The strongest (though still reasonable) assumption will yield a $2^{\Omega(\ell)}$-pseudorandom generator, thus implying that $\mathbf{BPP} = \mathbf{P}$. These are described in the following easy corollaries of the Lemma that are left as Exercise 1.

COROLLARY 16.6
1. If there exists a $2^{\epsilon \ell}$-pseudorandom generator for some constant $\epsilon > 0$ then $\mathbf{BPP} = \mathbf{P}$.

2. If there exists a $2^{\ell^{\epsilon}}$-pseudorandom generator for some constant $\epsilon > 0$ then $\mathbf{BPP} \subseteq \mathbf{QuasiP} = \mathbf{DTIME}(2^{polylog(n)})$.

3. If there exists an $S(\ell)$-pseudorandom generator for some super-polynomial function S (i.e., $S(\ell) = \ell^{\omega(1)}$) then $\mathbf{BPP} \subseteq \mathbf{SUBEXP} = \cap_{\epsilon > 0} \mathbf{DTIME}(2^{n^{\epsilon}})$.

16.1.1 Hardness and Derandomization

We construct pseudorandom generators under the assumptions that certain explicit functions are hard. In this chapter we use assumptions about *average-case* hardness, while in the next chapter we will be able to construct pseudorandom generators assuming only *worst-case* hardness. Both worst-case and average-case hardness refers to the size of the minimum Boolean circuit computing the function:

DEFINITION 16.7 (HARDNESS)
Let $f : \{0,1\}^* \to \{0,1\}$ be a Boolean function. The *worst-case hardness* of f, denoted $H_{wrs}(f)$, is a function from $\mathbb{N}$ to $\mathbb{N}$ that maps every $n \in \mathbb{N}$ to the largest number S such that every Boolean circuit of size at most S fails to compute f on some input in $\{0,1\}^n$.

The *average-case hardness* of f, denoted $H_{avg}(f)$, is a function from $\mathbb{N}$ to $\mathbb{N}$ that maps every $n \in \mathbb{N}$, to the largest number S such that $\Pr_{x \in_R \{0,1\}^n}[C(x) = f(x)] < \frac{1}{2} + \frac{1}{S}$ for every Boolean circuit C on n inputs with size at most S.

Note that for every function $f : \{0,1\}^* \to \{0,1\}$ and $n \in \mathbb{N}$, $H_{avg}(f)(n) \leq H_{wrs}(f)(n) \leq n2^n$.

REMARK 16.8
This definition of average-case hardness is tailored to the application of derandomization, and in particular only deals with the uniform distribution over the inputs. See Chapter 15 for a more general treatment of average-case complexity. We will also sometimes apply the notions of worst-case and average-case to *finite* functions from $\{0,1\}^n$ to $\{0,1\}$, where $H_{wrs}(f)$ and $H_{avg}(f)$ are defined in the natural way. (E.g., if $f : \{0,1\}^n \to \{0,1\}$ then $H_{wrs}(f)$ is the largest number S for which every Boolean circuit of size at most S fails to compute f on some input in $\{0,1\}^n$.)

EXAMPLE 16.9
Here are some examples of functions and their conjectured or proven hardness:

1. If f is a random function (i.e., for every $x \in \{0,1\}^*$ we choose $f(x)$ using an independent unbiased coin) then with high probability, both the worst-case and average-case hardness of f are exponential (see Exercise 3). In particular, with probability tending to 1 with n, both $H_{wrs}(f)(n)$ and $H_{avg}(f)(n)$ exceed $2^{0.99n}$. We will often use the shorthand $H_{wrs}(f), H_{avg}(f) \geq 2^{0.99n}$ for such expressions.

2. If $f \in \mathbf{BPP}$ then, since $\mathbf{BPP} \subseteq \mathbf{P}/\text{poly}$, both $H_{wrs}(f)$ and $H_{avg}(f)$ are bounded by some polynomial.

3. It seems reasonable to believe that 3SAT has exponential worst-case hardness; that is, $H_{wrs}(3SAT) \geq 2^{\Omega(n)}$. It is even more believable that $\mathbf{NP} \not\subseteq \mathbf{P}/\text{poly}$, which implies that $H_{wrs}(3SAT)$ is superpolynomial. The average case complexity of 3SAT is unclear, and in any case dependent upon the way we choose to represent formulas as strings.

4. If we trust the security of current cryptosystems, then we do believe that $\mathbf{NP}$ contains functions that are hard on the average. If g is a one-way permutation that cannot be inverted with polynomial probability by polynomial-sized circuits, then by Theorem 10.14, the function f that maps the pair $x, r \in \{0,1\}^n$ to $g^{-1}(x) \odot r$ has super-polynomial *average-case* hardness: $H_{avg}(f) \geq n^{\omega(1)}$. (Where $x \odot r = \sum_{i=1}^n x_i r_i \pmod 2$.) More generally there is a polynomial relationship between the size of the minimal circuit that inverts g (on the average) and the average-case hardness of f.

The main theorem of this section uses hard-on-the average functions to construct pseudorandom generators:

THEOREM 16.10 (CONSEQUENCES OF NW GENERATOR)
For every polynomial-time computable monotone $S : \mathbb{N} \to \mathbb{N}$, if there exists a constant c and function $f \in \mathbf{DTIME}(2^{cn})$ such that $\mathsf{H}_{\mathsf{avg}}(f) \geq S(n)$ then there exists a constant $\epsilon > 0$ such that an $S(\epsilon \ell)^{\epsilon}$-pseudorandom generator exists. In particular, the following corollaries hold:

1. *If there exists $f \in \mathbf{E} = \mathbf{DTIME}(2^{O(n)})$ and $\epsilon > 0$ such that $\mathsf{H}_{\mathsf{avg}}(f) \geq 2^{\epsilon n}$ then* $\mathbf{BPP} = \mathbf{P}$.

2. *If there exists $f \in \mathbf{E} = \mathbf{DTIME}(2^{O(n)})$ and $\epsilon > 0$ such that $\mathsf{H}_{\mathsf{avg}}(f) \geq 2^{n^{\epsilon}}$ then* $\mathbf{BPP} \subseteq \mathbf{QuasiP}$.

3. *If there exists $f \in \mathbf{E} = \mathbf{DTIME}(2^{O(n)})$ such that $\mathsf{H}_{\mathsf{avg}}(f) \geq n^{\omega(1)}$ then* $\mathbf{BPP} \subseteq \mathbf{SUBEXP}$.

REMARK 16.11
We can replace $\mathbf{E}$ with $\mathbf{EXP} = \mathbf{DTIME}(2^{\mathrm{poly}(n)})$ in Corollaries 2 and 3 above. Indeed, for every $f \in \mathbf{DTIME}(2^{n^c})$, the function g that on input $x \in \{0,1\}^*$ outputs the f applies to the first $|x|^{1/c}$ bits of x is in $\mathbf{DTIME}(2^n)$ and satisfies $\mathsf{H}_{\mathsf{avg}}(g)(n) \geq \mathsf{H}_{\mathsf{avg}}(f)(n^{1/c})$. Therefore, if there exists $f \in \mathbf{EXP}$ with $\mathsf{H}_{\mathsf{avg}}(f) \geq 2^{n^{\epsilon}}$ then there there exists a constant $\epsilon' > 0$ and a function $g \in \mathbf{E}$ with $\mathsf{H}_{\mathsf{avg}}(g) \geq 2^{n^{\epsilon'}}$, and so we can replace $\mathbf{E}$ with $\mathbf{EXP}$ in Corollary 2. A similar observation holds for Corollary 3. Note that $\mathbf{EXP}$ contains many classes we believe to have hard problems, such as $\mathbf{NP}, \mathbf{PSPACE}, \oplus \mathbf{P}$ and more, which is why we believe it does contain hard-on-the-average functions. In the next chapter we will give even stronger evidence to this conjecture, by showing it is implied by the assumption that $\mathbf{EXP}$ contains hard-in-the-worst-case functions.

REMARK 16.12
The original paper of Nisan and Wigderson [NW94] did not prove Theorem 16.10 as stated above. It was proven in a sequence of works [**?**]. Nisan and Wigderson only proved that under the same assumptions there exists an $S'(\ell)$-pseudorandom generator, where $S'(\ell) = S\left(\epsilon\sqrt{\ell}\log(S(\epsilon\sqrt{\ell}))\right)^{\epsilon}$ for some $\epsilon > 0$. Note that this is still sufficient to derive all three corollaries above. It is this weaker version we prove in this book.

16.2 Proof of Theorem 16.10: Nisan-Wigderson Construction

How can we use a hard function to construct a pseudorandom generator?

16.2.1   Warmup: two toy examples

For starters, we demonstrate this by considering the "toy example" of a pseudorandom generator whose output is only one bit longer than its input. Then we show how to extend by two bits. Of course, neither suffices to prove Theorem 16.10 but they do give insight to the connection between hardness and randomness.

Extending the input by one bit using Yao's Theorem.

The following Lemma uses a hard function to construct such a "toy" generator:

LEMMA 16.13 (ONE-BIT GENERATOR)
Suppose that there exist $f \in \mathbf{E}$ with $\mathsf{H}_{\mathsf{avg}}(f) \geq n^4$. Then, there exists an $S(\ell)$-pseudorandom generator G for $S(\ell) = \ell + 1$.

PROOF: The generator G will be very simple: for every $z \in \{0,1\}^{\ell}$, we set

$$G(z) = z \circ f(z)$$

(where $\circ$ denotes concatenation). G clearly satisfies the output length and efficiency requirements of an $(\ell+1)$-pseudorandom generator. To prove that its output is $1/10$-pseudorandom we use Yao's Theorem from Chapter 10 showing that pseudorandomness is implied by unpredictiability:[3]

THEOREM 16.14 (THEOREM 10.12, RESTATED)
Let Y be a distribution over $\{0,1\}^m$. Suppose that there exist $S > 10n, \epsilon > 0$ such that for every circuit C of size at most $2S$ and $i \in [m]$,

$$\Pr_{r \in_R Y}[C(r_1, \ldots, r_{i-1}) = r_i] \leq \frac{1}{2} + \frac{\epsilon}{m}$$

Then Y is (S, ϵ)-pseudorandom.

Using Theorem 16.14 it is enough to show that there does not exist a circuit C of size $2(\ell+1)^3 < \ell^4$ and a number $i \in [\ell + 1]$ such that

$$\Pr_{r = G(U_\ell)}[C(r_1, \ldots, r_{i-1}) = r_i] > \tfrac{1}{2} + \tfrac{1}{20(\ell+1)}. \tag{1}$$

However, for every $i \leq \ell$, the ith bit of $G(z)$ is completely uniform and independent from the first $i - 1$ bits, and hence cannot be predicted with probability larger than $1/2$ by a circuit of any size. For $i = \ell + 1$, Equation (1) becomes,

$$\Pr_{z \in_R \{0,1\}^\ell}[C(z) = f(z)] > \frac{1}{2} + \frac{1}{20(\ell+1)} > \frac{1}{2} + \frac{1}{\ell^4},$$

which cannot hold under the assumption that $\mathsf{H}_{\mathsf{avg}}(f) \geq n^4$. ∎

---

[3] Although this theorem was stated and proved in Chapter 10 for the case of *uniform* Turing machines, the proof easily extends to the case of circuits.

Extending the input by two bits using the averaging principle.

We now continue to progress in "baby steps" and consider the next natural toy problem: constructing a pseudorandom generator that extends its input by two bits. This is obtained in the following Lemma:

LEMMA 16.15 (TWO-BIT GENERATOR)
Suppose that there exists $f \in \mathbf{E}$ *with* $\mathsf{H}_{\mathsf{avg}}(f) \geq n^4$. *Then, there exists an* $(\ell+2)$-*pseudorandom generator* G.

PROOF: The construction is again very natural: for every $z \in \{0,1\}^{\ell}$, we set

$$G(z) = z_1 \cdots z_{\ell/2} \circ f(z_1, \ldots, z_{\ell/2}) \circ z_{\ell/2+1} \cdots z_{\ell} \circ f(z_{\ell/2+1}, \ldots, z_{\ell}).$$

Again, the efficiency and output length requirements are clearly satisfied.

To show $G(U_\ell)$ is $1/10$-pseudorandom, we again use Theorem 16.14, and so need to prove that there does not exists a circuit C of size $2(\ell+1)^3$ and $i \in [\ell+2]$ such that

$$\Pr_{r=G(U_\ell)}[C(r_1, \ldots, r_{i-1}) = r_i] > \frac{1}{2} + \frac{1}{20(\ell+2)}. \tag{2}$$

Once again, (2) cannot occur for those indices i in which the ith output of $G(z)$ is truly random, and so the only two cases we need to consider are $i = \ell/2 + 1$ and $i = \ell + 2$. Equation (2) cannot hold for $i = \ell/2+1$ for the same reason as in Lemma 16.13. For $i = \ell+2$, Equation (2) becomes:

$$\Pr_{r,r' \in_R \{0,1\}^{\ell/2}}[C(r \circ f(r) \circ r') = f(r')] > \frac{1}{2} + \frac{1}{20(\ell+2)} \tag{3}$$

This may seem somewhat problematic to analyze since the input to C contains the bit $f(r)$, which C could not compute on its own (as f is a hard function). Couldn't it be that the input $f(r)$ helps C in predicting the bit $f(r')$? The answer is NO, and the reason is that r' and r are *independent*. Formally, we use the following principle (see Section A.3.2 in the appendix):

> THE AVERAGING PRINCIPLE: If A is some event depending on two independent random variables X, Y, then there exists some x in the range of X such that
>
> $$\Pr_Y[A(x,Y) \geq \Pr_{X,Y}[A(X,Y)]$$

Applying this principle here, if (3) holds then there exists a string $r \in \{0,1\}^{\ell/2}$ such that

$$\Pr_{r' \in_R \{0,1\}^{\ell/2}}[C(r, f(r), r') = f(r')] > \frac{1}{2} + \frac{1}{20(\ell+2)}.$$

(Note that this probability is now only over the choice of r'.) If this is the case, we can "hardwire" the $\ell/2+1$ bits $r \circ f(r)$ to the circuit C and obtain a circuit D of size at most $(\ell+2)^3 + 2\ell < (\ell/2)^4$ such that

$$\Pr_{r' \in_R \{0,1\}^{\ell/2}}[D(r') = f(r')] > \frac{1}{2} + \frac{1}{20(\ell+2)},$$

contradicting the hardness of f. ∎

Beyond two bits:

A generator that extends the output by two bits is still useless for our goals. We can generalize the proof Lemma 16.15 to obtain a generator G that extends the output by k bits setting

$$G(z_1, \ldots, z_\ell) = z^1 \circ f(z^1) \circ z^2 \circ f(z^2) \cdots z^k \circ f(z^k), \tag{4}$$

where z^i is the ith block of ℓ/k bits in z. However, no matter how big we set k and no matter how hard the function f is, we cannot get a generator that expands its input by a multiplicative factor larger than two. Note that to prove Theorem 16.10 we need a generator that, depending on the hardness we assume, has output that can be exponentially larger than the input! Clearly, we need a new idea.

16.2.2   The NW Construction

The new idea is still inspired by the construction of (4), but instead of taking $z^1, \ldots, z^k$ to be independently chosen strings (or equivalently, disjoint pieces of the input z), we take them to be *partly dependent* by using *combinatorial designs*. Doing this will allow us to take k so large that we can drop the actual inputs from the generator's output and use only $f(z^1) \circ f(z^2) \cdots \circ f(z^k)$. The proof of correctness is similar to the above toy examples and uses Yao's technique, except the fixing of the input bits has to be done more carefully because of dependence among the strings.

First, some notation. For a string $z \in \{0,1\}^\ell$ and subset $I \subseteq [\ell]$, we define $z_{\restriction I}$ to be $|I|$-length string that is the projection of z to the coordinates in I. For example, $z_{\restriction [1..i]}$ is the first i bits of z.

DEFINITION 16.16 (NW GENERATOR)
If $\mathcal{I} = \{I_1, \ldots, I_m\}$ is a family of subsets of $[\ell]$ with each $|I_j| = l$ and $f : \{0,1\}^n \to \{0,1\}$ is any function then the $(\mathcal{I}, f)$-*NW generator* (see Figure 16.2) is the function $\mathsf{NW}_{\mathcal{I}}^f : \{0,1\}^\ell \to \{0,1\}^m$ that maps any $z \in \{0,1\}^\ell$ to

$$\mathsf{NW}_{\mathcal{I}}^f(z) = f(z_{\restriction I_1}) \circ f(z_{\restriction I_2}) \cdots \circ f(z_{\restriction I_m}) \tag{5}$$
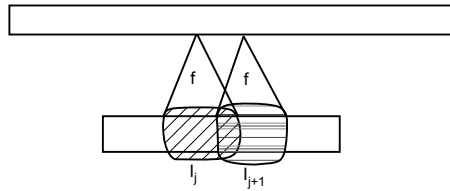
Figure 16.2: The *NW generator*, given a set system $\mathcal{I} = \{I_1, \ldots, I_m\}$ of size n subsets of $[\ell]$ and a function $f : \{0,1\}^n \to \{0,1\}$ maps a string $z \in \{0,1\}^\ell$ to the output $f(z_{\restriction I_1}), \ldots, f(z_{\restriction I_m})$. Note that these sets are not necessarily disjoint (although we will see their intersections need to be small).

Conditions on the set systems and function.

We will see that in order for the generator to produce pseudorandom outputs, function f must display some *hardness*, and the family of subsets must come from an efficiently constructible combinatorial design.

DEFINITION 16.17 (COMBINATORIAL DESIGNS)
If $d, n, \ell \in \mathbb{N}$ are numbers with $\ell > n > d$ then a family $\mathcal{I} = \{I_1, \ldots, I_m\}$ of subsets of $[\ell]$ is an (ℓ, n, d)-*design* if $|I_j| = n$ for every j and $|I_j \cap I_k| \leq d$ for every $j \neq k$.

The next lemma yields efficient constructions of these designs and is proved later.

LEMMA 16.18 (CONSTRUCTION OF DESIGNS)
There is an algorithm A such that on input $\ell, d, n \in \mathbb{N}$ where $n > d$ and $\ell > 10n^2/d$, runs for $2^{O(\ell)}$ steps and outputs an (ℓ, n, d)-design $\mathcal{I}$ containing $2^{d/10}$ subsets of $[\ell]$.

The next lemma shows that if f is a hard function and $\mathcal{I}$ is a design with sufficiently good parameters, than $\mathsf{NW}^f_{\mathcal{I}}(U_\ell)$ is indeed a pseudorandom distribution:

LEMMA 16.19 (PSEUDORANDOMNESS USING THE NW GENERATOR)
If $\mathcal{I}$ is an (ℓ, n, d)-design with $|\mathcal{I}| = 2^{d/10}$ and $f : \{0,1\}^n \to \{0,1\}$ a function satisfying $2^d < \sqrt{\mathsf{H}_{\mathsf{avg}}(f)(n)}$, then the distribution $\mathsf{NW}^f_{\mathcal{I}}(U_\ell)$ is a $(\mathsf{H}_{\mathsf{avg}}(f)(n)/10, 1/10)$-pseudorandom distribution.

PROOF: Let S denote $\mathsf{H}_{\mathsf{avg}}(f)(n)$. By Yao's Theorem, we need to prove that for every $i \in [2^{d/10}]$ there does *not* exist an $S/2$-sized circuit C such that

$$\Pr_{\substack{Z \sim U_\ell \\ R = \mathsf{NW}^f_{\mathcal{I}}(Z)}} [C(R_1, \ldots, R_{i-1}) = R_i] \geq \frac{1}{2} + \frac{1}{10 \cdot 2^{d/10}} \,. \tag{6}$$

For contradiction's sake, assume that (6) holds for some circuit C and some i. Plugging in the definition of $\mathsf{NW}^f_{\mathcal{I}}$, Equation (6) becomes:

$$\Pr_{Z \sim U_\ell} [C(f(Z_{\restriction I_1}), \cdots, f(Z_{\restriction I_{i-1}})) = f(Z_{\restriction I_i})] \geq \frac{1}{2} + \frac{1}{10 \cdot 2^{d/10}} \,. \tag{7}$$

Letting Z_1 and Z_2 denote the two independent variables corresponding to the coordinates of Z in I_i and $[\ell] \setminus I_i$ respectively, Equation (7) becomes:

$$\Pr_{\substack{Z_1 \sim U_n \\ Z_2 \sim U_{\ell-n}}} [C(f_1(Z_1, Z_2), \ldots, f_{i-1}(Z_1, Z_2)) = f(Z_1)] \geq \frac{1}{2} + \frac{1}{10 \cdot 2^{d/10}} \,, \tag{8}$$

where for every $j \in [2^{d/10}]$, f_j applies f to the coordinates of Z_1 corresponding to $I_j \cap I_i$ and the coordinates of Z_2 corresponding to $I_j \setminus I_i$. By the averaging principle, if (8) holds then there exists a string $z_2 \in \{0,1\}^{\ell-n}$ such that

$$\Pr_{Z_1 \sim U_n} [C(f_1(Z_1, z_2), \ldots, f_{i-1}(Z_1, z_2)) = f(Z_1)] \geq \frac{1}{2} + \frac{1}{10 \cdot 2^{d/10}} \,. \tag{9}$$

We may now appear to be in some trouble, since all of $f_j(Z_1, z_2)$ for $j \leq i-1$ do depend upon Z_1, and the fear is that if they together contain enough information about Z_1 then a circuit *could potentially* predict $f_i(Z_1)$ after looking at all of them. To prove that this fear is baseless we use the fact that the circuit C is small and f is a very hard function.

Since $|I_j \cap I_i| \leq d$ for $j \neq i$, the function $Z_1 \mapsto f_j(Z_1, z_2)$ depends at most d coordinates of z_1 and hence can be computed by a $d2^d$-sized circuit. (Recall that z_2 is fixed.) Thus if if (8) holds then there exists a circuit B of size $2^{d/10} \cdot d2^d + S/2 < S$ such that

$$\Pr_{Z_1 \sim U_n}[B(Z_1) = f(Z_1)] \geq \frac{1}{2} + \frac{1}{10 \cdot 2^{d/10}} > \frac{1}{2} + \frac{1}{S}. \tag{10}$$

But this contradicts the fact that $\mathsf{H}_{\mathrm{avg}}(f)(n) = S$. ∎

REMARK 16.20 (BLACK-BOX PROOF)
Lemma 16.19 shows that if $\mathsf{NW}^f_{\mathcal{I}}(U_\ell)$ is distinguishable from the uniform distribution $U_{2^{d/10}}$ by some circuit D, then there exists a circuit B (of size polynomial in the size of D and in 2^d) that computes the function f with probability noticeably larger than $1/2$. The construction of this circuit B actually uses the circuit D as a *black-box*, invoking it on some chosen inputs. This property of the NW generator (and other constructions of pseudorandom generators) turned out to be useful in several settings. In particular, Exercise 5 uses it to show that under plausible complexity assumptions, the complexity class **AM** (containing all languages with a constant round interactive proof, see Chapter 8) is equal to **NP**. We will also use this property in the construction of *randomness extractors* based on pseudorandom generators.

Putting it all together: Proof of Theorem 16.10 from Lemmas 16.18 and 16.19

As noted in Remark 16.12, we do not prove here Theorem 16.10 as stated but only the weaker statement, that given $f \in \mathbf{E}$ and $S : \mathbb{N} \to \mathbb{N}$ with $\mathsf{H}_{\mathrm{avg}}(f) \geq S$, we can construct an $S'(\ell)$-pseudorandom generator, where $S'(\ell) = S\left(\epsilon \sqrt{\ell} \log(S(\epsilon \sqrt{\ell}))\right)^\epsilon$ for some $\epsilon > 0$.

For such a function f, we denote our pseudorandom generator by NW^f. Given input $z \in \{0,1\}^\ell$, the generator NW^f operates as follows:

- Set n to be the largest number such that $\ell > 100n^2 / \log S(n)$. Set $d = \log S(n)/10$. Since $S(n) < 2^n$, we can assume that $\ell \leq 300n^2 / \log S(n)$.

- Run the algorithm of Lemma 16.18 to obtain an (ℓ, n, d)-design $\mathcal{I} = \{I_1, \ldots, I_{2^{d/5}}\}$.

- Output the first $S(n)^{1/40}$ bits of $\mathsf{NW}^f_{\mathcal{I}}(z)$.

Clearly, $\mathsf{NW}^f(z)$ runs in $2^{O(\ell)}$ time. Moreover, since $2^d \leq S(n)^{1/10}$, Lemma 16.19 implies that the distribution $\mathsf{NW}^f(U_\ell)$ is $(S(n)/10, 1/10)$-pseudorandom. Since $n \geq \sqrt{\ell} \log S(n)/300 \geq \sqrt{\ell} \log S(\frac{\sqrt{\ell}}{300})/300$ (with the last inequality following from the fact that S is monotone), this concludes the proof of Theorem 16.10. ∎

Construction of combinatorial designs.

All that is left to complete the proof is to show the construction of combinatorial designs with the required parameters:

PROOF OF LEMMA 16.18 (CONSTRUCTION OF COMBINATORIAL DESIGNS): On inputs ℓ, d, n with $\ell > 10n^2/d$, our Algorithm A will construct an (ℓ, n, d)-design $\mathcal{I}$ with $2^{d/10}$ sets using the simple greedy strategy:

> Start with $\mathcal{I} = \emptyset$ and after constructing $\mathcal{I} = \{I_1, \ldots, I_m\}$ for $m < 2^{d/10}$, search all subsets of $[\ell]$ and add to $\mathcal{I}$ the first n-sized set I satisfying $|I \cap I_j| \leq d$ for every $j \in [m]$. We denote this latter condition by **(*)**.

Clearly, A runs in $\text{poly}(m)2^\ell = 2^{O(\ell)}$ time and so we only need to prove it never gets stuck. In other words, it suffices to show that if $\ell = 10n^2/d$ and $\{I_1, \ldots, I_m\}$ is a collection of n-sized subsets of $[\ell]$ for $m < 2^{d/10}$, then there exists an n-sized subset $I \subseteq [\ell]$ satisfying **(*)**. We do so by showing that if we pick I at random by choosing independently every element $x \in [\ell]$ to be in I with probability $2n/\ell$ then:

$$\Pr[|I| \geq n] \geq 0.9 \tag{11}$$

$$\Pr[|I \cap I_j| \geq d] \leq 0.5 \cdot 2^{-d/10} \qquad (\forall j \in [m]) \tag{12}$$

Because the expected size of I is $2n$, while the expected size of the intersection $I \cap I_j$ is $2n^2/\ell < d/5$, both (12) and (11) follow from the Chernoff bound. Yet together these two conditions imply that with probability at least 0.4, the set I will simultaneously satisfy **(*)** and have size at least n. Since we can always remove elements from I without damaging **(*)**, this completes the proof. ■

16.3   Derandomization requires circuit lowerbounds

We saw in Section 16.2 that if we can prove certain strong circuit lowerbounds, then we can partially (or fully) derandomize **BPP**. Now we prove a result in the reverse direction: derandomizing **BPP** requires proving circuit lowerbounds. Depending upon whether you are an optimist or a pessimist, you can view this either as evidence that derandomizing **BPP** is difficult, or, as a reason to double our efforts to derandomize **BPP**.

We say that a function is in **AlgP**/*poly* if it can be computed by a polynomial size arithmetic circuit whose gates are labeled by $+, -, \times$ and $\div$, which are operations over some underlying field or ring. We let perm denote the problem of computing the permanent of matrices over the integers. (The proof can be extended to permanent computations over finite fields of characteristic > 2.) We prove the following result.

THEOREM 16.21 ([**?**])
P = **BPP** $\Rightarrow$ **NEXP** $\not\subseteq$ **P**/*poly* or perm $\notin$ **AlgP**/*poly*.

REMARK 16.22
It is possible to replace the "poly" in the conclusion perm $\notin$ **AlgP**/*poly* with a subexponential function by appropriately modifying Lemma 16.25. It is open whether the conclusion **NEXP** $\not\subseteq$ **P**/*poly* can be similarly strengthened.

In fact, we will prove the following stronger theorem. Recall the *Polynomial Identity Testing* (ZEROP) problem in which the input consists of a polynomial represented by an arithmetic circuit computing it (see Section 7.2.2 and Example 16.1), and we have to decide if it is the identically zero polynomial. This problem is in **coRP** $\subseteq$ **BPP** and we will show that if it is in **P** then the conclusions of Theorem 16.21 hold:

THEOREM 16.23 (DERANDOMIZATION IMPLIES LOWER BOUNDS)
If ZEROP $\in$ **P** *then either* **NEXP** $\not\subseteq$ **P**/*poly or* perm $\notin$ **AlgP**/*poly.*

The proof relies upon many results described earlier in the book.[4] Recall that **MA** is the class of languages that can be proven by a one round interactive proof between two players Arthur and Merlin (see Definition 8.7). Merlin is an all-powerful prover and Arthur is a polynomial-time verifier that can flip random coins. That is, given an input x, Merlin first sends Arthur a "proof" y. Then Arthur with y in hand flips some coins and decides whether or not to accept x. For this to be an **MA** protocol, Merlin must convince Arthur to accept strings in L with probability one while at the same time Arthur must not be fooled into accepting strings not in L except with probability smaller than $1/2$. We will use the following result regarding **MA**:

LEMMA 16.24 ([BFL91],[BFNW93])
EXP $\subseteq$ **P**/*poly* $\Rightarrow$ **EXP** = **MA**.

PROOF: Suppose **EXP** $\subseteq$ **P**/*poly*. By the Karp-Lipton theorem (Theorem 6.14), in this case **EXP** collapses to the second level Σ_2^p of the polynomial hierarchy. Hence Σ_2^p = **PH** = **PSPACE** = **IP** = **EXP** $\subseteq$ **P**/*poly*. Thus every $L \in$ **EXP** has an interactive proof, and furtheremore, since **EXP** = **PSPACE**, we can just the use the interactive proof for TQBF, for which the prover is a **PSPACE** machine. Hence the prover can be replaced by a polynomial size circuit family C_n. Now we see that the interactive proof can actually be carried out in 2 rounds, with Merlin going first. Given an input x of length n, Merlin gives Arthur a polynomial size circuit C, which is supposed to be the C_n for L. Then Arthur runs the interactive proof for L, using C as the prover. Note that if the input is not in the language, then *no* prover has a decent chance of convincing the verifier, so this is true also for prover described by C. Thus we have described an **MA** protocol for L implying that **EXP** $\subseteq$ **MA** and hence that **EXP** = **MA**. ∎

Our next ingredient for the proof of Theorem 16.23 is the following lemma:

LEMMA 16.25
If ZEROP $\in$ **P**, *and* perm $\in$ **AlgP**/*poly. Then* **P**$^{\text{perm}}$ $\subseteq$ **NP**.

---

[4]This is a good example of "third generation" complexity results that use a clever combination of both "classical" results from the 60's and 70's and newer results from the 1990's.

PROOF: Suppose perm has algebraic circuits of size n^c, and that ZEROP has a polynomial-time algorithm. Let L be a language that is decided by an n^d-time TM M using queries to a perm-oracle. We construct an **NP** machine N for L.

Suppose x is an input of size n. Clearly, M's computation on x makes queries to perm of size at most $m = n^d$. So N will use nondeterminism as follows: it guesses a sequence of m algebraic circuits $C_1, C_2, \ldots, C_m$ where C_i has size i^c. The hope is that C_i solves perm on $i \times i$ matrices, and N will verify this in poly(m) time. The verification starts by verifying C_1, which is trivial. Inductively, having verified the correctness of $C_1, \ldots, C_{t-1}$, one can verify that C_t is correct using downward self-reducibility, namely, that for a $t \times t$ matrix A,

$$\mathsf{perm}(A) = \sum_{i=1}^{t} a_{1i}\mathsf{perm}(A_{1,i}),$$

where $A_{1,i}$ is the $(t-1) \times (t-1)$ sub-matrix of A obtained by removing the 1st row and ith column of A. Thus if circuit C_{t-1} is known to be correct, then the correctness of C_t can be checked by substituting $C_t(A)$ for $\mathsf{perm}(A)$ and $C_{t-1}(A_{1,i})$ for $\mathsf{perm}(A_{1,i})$: this yields an identity involving algebraic circuits with t^2 inputs which can be verified deterministically in poly(t) time using the algorithm for ZEROP. Proceeding this way N verifies the correctness of $C_1, \ldots, C_m$ and then simulates M^{perm} on input x using these circuits. ∎

The heart of the proof is the following lemma, which is interesting in its own right:

LEMMA 16.26 ([**?**])
NEXP $\subseteq$ **P**/*poly* $\Rightarrow$ **NEXP** = **EXP**.

PROOF: We prove the contrapositive. Suppose that **NEXP** $\neq$ **EXP** and let $L \in$ **NEXP** $\setminus$ **EXP**. Since $L \in$ **NEXP** there exists a constant $c > 0$ and a relation R such that

$$x \in L \Leftrightarrow \exists y \in \{0,1\}^{2^{|x|^c}} \text{ s.t. } R(x,y) \text{ holds},$$

where we can test whether $R(x,y)$ holds in time $2^{|x|^{c'}}$ for some constant c'.

For every constant $d > 0$, let M_d be the following machine: on input $x \in \{0,1\}^n$ enumerate over all possible Boolean circuits C of size n^{100d} that take n^c inputs and have a single output. For every such circuit let $\mathsf{tt}(C)$ be the 2^{n^c}-long string that corresponds to the truth table of the function computed by C. If $R(x, \mathsf{tt}(C))$ holds then halt and output 1. If this does not hold for any of the circuits then output 0.

Since M_d runs in time $2^{n^{101d}+n^c}$, under our assumption that $L \notin$ **EXP**, for every d there exists an infinite sequence of inputs $\mathcal{X}_d = \{x_i\}_{i \in \mathbb{N}}$ on which $M_d(x_i)$ outputs 0 even though $x_i \in L$ (note that if $M_d(x) = 1$ then $x \in L$). This means that for every string x in the sequence $\mathcal{X}_d$ and every y such that $R(x,y)$ holds, the string y represents the truth table of a function on n^c bits that cannot be computed by circuits of size n^{100d}, where $n = |x|$. Using the pseudorandom generator based on worst-case assumptions (Theorem **??**), we can use such a string y to obtain an ℓ^d-pseudorandom generator.

Now, if **NEXP** $\subseteq$ **P**/poly then as noted above **NEXP** $\subseteq$ **MA** and hence every language in **NEXP** has a proof system where Merlin proves that an n-bit string is in the language by sending

a proof which Arthur then verifies using a probabilistic algorithm of at most n^d steps. Yet, if n is the input length of some string in the sequence $\mathcal{X}_d$ and we are given $x \in \mathcal{X}_d$ with $|x| = n$, then we can replace Arthur by non-deterministic poly$(n^d)2^{n^c}$ time algorithm that does not toss any coins: Arthur will guess a string y such that $R(x, y)$ holds and then use y as a function for a pseudorandom generator to verify Merlin's proof.

This means that there is a constant $c > 0$ such that *every* language in **NEXP** can be decided on infinitely many inputs by a non-deterministic algorithm that runs in poly(2^{n^c})-time and uses n bits of advice (consisting of the string $x \in \mathcal{X}_d$). Under the assumption that **NEXP** $\subseteq$ **P**/poly we can replace the poly(2^{n^c}) running time with a circuit of size $n^{c'}$ where c' is a constant depending only on c, and so get that there is a constant c' such that every language in **NEXP** can be decided on infinitely many inputs by a circuit family of size $n + n^{c'}$. Yet this can be ruled out using elementary diagonalization. ∎

REMARK 16.27
It might seem that Lemma 16.26 should have an easier proof that goes along the proof that **EXP** $\subseteq$ **P**/poly $\Rightarrow$ **EXP** = **MA**, but instead of using the interactive proof for TQBF uses the *multi-prover* interactive proof system for **NEXP**. However, we do not know how to implement the provers' strategies for this latter system in **NEXP**. (Intuitively, the problem arises from the fact that a **NEXP** statement may have several certificates, and it is not clear how we can ensure all provers use the same one.)

We now have all the ingredients for the proof of Theorem 16.23.

PROOF OF THEOREM 16.23: For contradiction's sake, assume that the following are all true:

$$ZEROP \in \mathbf{P} \tag{13}$$

$$\mathbf{NEXP} \subseteq \mathbf{P}/poly, \tag{14}$$

$$\mathsf{perm} \in \mathbf{AlgP}/poly. \tag{15}$$

Statement (14) together with Lemmas 16.24 and 16.26 imply that **NEXP** = **EXP** = **MA**. Now recall that **MA** $\subseteq$ **PH**, and that by Toda's Theorem (Theorem 9.11) **PH** $\subseteq$ **P**$^{\#\mathbf{P}}$. Recall also that by Valiant's Theorem (Theorem 9.8) perm is #**P**-complete. Thus, under our assumptions

$$\mathbf{NEXP} \subseteq \mathbf{P}^{\mathsf{perm}}. \tag{16}$$

Since we assume that ZEROP $\in$ **P**, Lemma 16.25 together with statements (15) and (16) implies that **NEXP** $\subseteq$ **NP**, contradicting the Nondeterministic Time Hierarchy Theorem (Theorem 3.3). Thus the three statements at the beginning of the proof cannot be simultaneously true. ∎

16.4 Explicit construction of expander graphs

Recall that an *expander graph family* is a family of graphs $\{G_n\}_{n \in I}$ such that for some constants λ and d, for every $n \in I$, the graph G_n has n-vertices, degree d and its second eigenvalue is at most λ (see Section 7.B). A *strongly explicit* expander graph family is such a family where there

is an algorithm that given n and the index of a vertex v in G_n, outputs the list of v's neighbors in $\text{poly}(\log(n))$ time. In this section we show a construction for such a family. Such construction have found several applications in complexity theory and other areas of computer science (one such application is the randomness efficient error reduction procedure we saw in Chapter 7).

The main tools in our construction will be several types of graph products. A *graph product* is an operation that takes two graphs G, G' and outputs a graph H. Typically we're interested in the relation between properties of the graphs G, G' to the properties of the resulting graph H. In this section we will mainly be interested in three parameters: the number of vertices (denoted n), the degree (denoted d), and the 2nd largest eigenvalue of the normalized adjacency matrix (denoted λ), and study how different products affect these parameters. We then use these products to obtain a construction of a strongly explicit expander graph family. In the next section we will use the same products to show a *deterministic* logspace algorithm for undirected connectivity.

16.4.1   Rotation maps.

In addition to the adjacency matrix representation, we can also represent an n-vertex degree-d graph G as a function $\hat{G}$ from $[n] \times [d]$ to $[n]$ that given a pair $\langle v, i \rangle$ outputs u where the ith neighbor of v in G. In fact, it will be convenient for us to have $\hat{G}$ output an additional value $j \in [d]$ where j is the index of v as a neighbor of u. Given this definition of $\hat{G}$ it is clear that we can invert it by applying it again, and so it is a permutation on $[n] \times [d]$. We call $\hat{G}$ the *rotation map* of G. For starters, one may think of the case that $\hat{G}(u, i) = (v, i)$ (i.e., v is the ith neighbor of u iff u is the ith neighbor of v). In this case we can think of $\hat{G}$ as operating only on the vertex. However, we will need the more general notion of a rotation map later on.

We can describe a graph product in the language of graphs, adjacency matrices, or rotation maps. Whenever you see the description of a product in one of this forms (e.g., as a way to map two graphs into one), it is a useful exercise to work out the equivalent descriptions in the other forms (e.g., in terms of adjacency matrices and rotation maps).

16.4.2   The matrix/path product

G: (n,d,λ)-graph   G': (n,d',λ')-graph   G'G: (n,dd',λλ')-graph
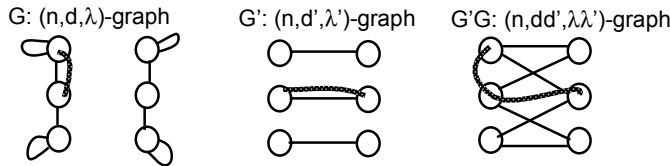
For every two n vertex graphs G, G' with degrees d, d' and adjacency matrices A, A', the graph $G'G$ is the graph described by the adjacency matrix $A'A$. That is, $G'G$ has an edge (u, v) for every length 2-path from u to v where the first step in the path is taken on en edge of G and the second is on an edge of G'. Note that G has n vertices and degree dd'. Typically, we are interested in the case $G = G'$, where it is called *graph squaring*. More generally, we denote by G^k the graph $G \cdot G \cdots G$ (k times). We already encountered this case before in Lemma 7.27, and similar analysis yields the following lemma (whose proof we leave as exercise):

LEMMA 16.28 (MATRIX PRODUCT IMPROVES EXPANSION)
$\lambda(G'G) \le \lambda(G')\lambda(G')$

It is also not hard to compute the rotation map of $G'G$ from the rotation maps of G and G'. Again, we leave verifying this to the reader.

16.4.3   The tensor product

G: (n,d,λ)-graph   G': (n',d',λ')-graph   G⊗G': (nn',dd',max{λ,λ'})-graph
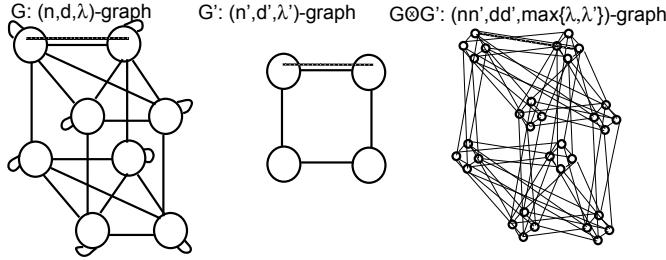
Let G and G' be two graphs with n (resp n') vertices and d (resp. d') degree, and let $\hat{G}$: $[n] \times [d] \to [n] \times [d]$ and $\hat{G}' : [n'] \times [d'] \to [n'] \times [d']$ denote their respective *rotation maps*. The *tensor product* of G and G', denoted $G \otimes G'$, is the graph over nn' vertices and degree dd' whose rotation map $G \hat{\otimes} G'$ is the permutation over $([n] \times [n']) \times ([d] \times [d'])$ defined as follows

$$G \hat{\otimes} G'(\langle u, v \rangle, \langle i, j \rangle) = \langle u', v' \rangle, \langle i', j' \rangle,$$

where $(u', i') = \hat{G}(u, i)$ and $(v', j') = \hat{G}'(v, j)$. That is, the vertex set of $G \otimes G'$ is pairs of vertices, one from G and the other from G', and taking a the step $\langle i, j \rangle$ on $G \otimes G'$ from the vertex $\langle u, v \rangle$ is akin to taking two independent steps: move to the pair $\langle u', v' \rangle$ where u' is the ith neighbor of u in G and v' is the ith neighbor of v in G'.

In terms of adjacency matrices, the tensor product is also quite easy to describe. If $A = (a_{i,j})$ is the $n \times n$ adjacency matrix of G and $A' = (a'_{i',j'})$ is the $n' \times n'$ adjacency matrix of G', then the adjacency matrix of $G \otimes G'$, denoted as $A \otimes A'$, will be an $nn' \times nn'$ matrix that in the $\langle i, i' \rangle^{th}$ row and the $\langle j, j' \rangle$ column has the value $a_{i,j} \cdot a'_{i',j'}$. That is, $A \otimes A'$ consists of n^2 copies of A', with the $(i, j)^{th}$ copy scaled by $a_{i,j}$:

$$A \otimes A' = \begin{pmatrix} a_{1,1}A' & a_{1,2}A' & \ldots & a_{1,n}A' \\ a_{2,1}A' & a_{2,2}A' & \ldots & a_{2,n}A' \\ \vdots & & & \vdots \\ a_{n,1}A' & a_{n,2}A' & \ldots & a_{n,n}A' \end{pmatrix}$$

The tensor product can also be described in the language of graphs as having a cluster of n' vertices in $G \otimes G'$ for every vertex of G. Now if, u and v are two neighboring vertices in G, we will put a bipartite version of G' between the cluster corresponding to u and the cluster corresponding to v in G. That is, if (i, j) is an edge in G' then there is an edge between the ith vertex in the cluster corresponding to u and the jth vertex in the cluster corresponding to v.

LEMMA 16.29 (TENSOR PRODUCT PRESERVES EXPANSION)
Let $\lambda = \lambda(G)$ and $\lambda' = \lambda(G')$ then $\lambda(G \otimes G') \leq \max\{\lambda, \lambda'\}$.

One intuition for this bound is the following: taking a T step random walk on the graph $G \otimes G'$ is akin to taking two independent random walks on the graphs G and G'. Hence, if both walks converge to the uniform distribution within T steps, then so will the walk on $G \otimes G'$.

PROOF: Given some basic facts about tensor products and eigenvalues this is immediate since if $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of A (where A is the adjacency matrix of G) and $\lambda'_1, \ldots, \lambda'_{n'}$ are the eigenvalues of A (where A' is the adjacency matrix of G'), then the eigenvalues of $A \otimes A'$ are all numbers of the form $\lambda_i \cdot \lambda'_j$, and hence the largest ones apart from 1 are of the form $1 \cdot \lambda(G')$ or $\lambda(G) \cdot 1$ (see also Exercise 14). ∎

We note that one can show that $\lambda(G \otimes G') \leq \lambda(G) + \lambda(G')$ without relying on any knowledge of eigenvalues (see Exercise 15). This weaker bound suffices for our applications.

16.4.4 The replacement product

G: (n,D,1-ε)-graph   G': (D,d,1-ε')-graph   G⊙G': (nD,2d,1-εε'/4)-graph
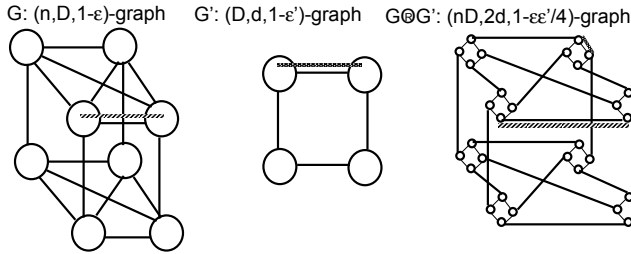
In both the matric and tensor products, the degree of the resulting graph is larger than the degree of the input graphs. The following product will enable us to reduce the degree of one of the graphs. Let G, G' be two graphs such that G has n vertices and degree D, and G' has D vertices and degree d. The *balanced replacement product* (below we use simply *replacement product* for short) of G and G' is denoted by $G \circledR G'$ is the nn'-vertex $2d$-degree graph obtained as follows:

1. For every vertex u of G, the graph $G \circledR G'$ has a copy of G' (including both edges and vertices).

2. If u, v are two neighboring vertices in G then we place d parallel edges between the ith vertex in the copy of G' corresponding to u and the jth vertex in the copy of G' corresponding to v, where i is the index of v as a neighbor of u and j is the index of u as a neighbor of v in G. (That is, taking the ith edge out of u leads to v and taking the jth edge out of v leads to u.)

Note that we essentially already encountered this product in the proof of Claim **??** (see also Figure **??**), where we reduced the degree of an arbitrary graph by taking its replacement product with a cycle (although there we did not use parallel edges).[5] The replacement product also has

---

[5]The addition of parallel edges ensures that a random step from a vertex v in $G \circledR G'$ will move to a neighbor within the same cluster and a neighbor outside the cluster with the same probability. For this reason, we call this product the *balanced* replacement product.

a simple description in terms of rotation maps: since $G \,\textcircled{R}\, G'$ has nD vertices and $2d$ degree, its rotation map $G \,\hat{\textcircled{R}}\, G'$ is a permutation over $([n] \times [D]) \times ([d] \times \{0,1\})$ and so can be thought of as taking four inputs u, v, i, b where $u \in [n]$, $v \in [D]$, $i \in [d]$ and $b \in \{0,1\}$. If $b = 0$ then it outputs $u, \hat{G}'(v, i), b$ and if $b = 1$ then it outputs $\hat{G}(u, v), i, b$. That is, depending on whether b is equal to 0 or 1, the rotation map either treats v as a vertex of G' or as an edge label of G.

In the language of adjacency matrices the replacement product can be easily seen to be described as follows: $A \,\textcircled{R}\, A' = 1/2(A \otimes I_D) + 1/2(I_n \otimes A')$, where A, A' are the adjacency matrices of the graphs G and G' respectively, and I_k is the $k \times k$ identity matrix.

If $D \gg d$ then the replacement product's degree will be significantly smaller than G's degree. The following Lemma shows that this dramatic degree reduction does not cause too much of a deterioration in the graph's expansion:

LEMMA 16.30 (EXPANSION OF REPLACEMENT PRODUCT)
If $\lambda(G) \le 1 - \epsilon$ and $\lambda(G') \le 1 - \epsilon'$ then $\lambda(G \,\textcircled{R}\, G') \le 1 - \epsilon\epsilon'/4$.

The intuition behind Lemma 16.30 is the following: Think of the input graph G as a good expander whose only drawback is that it has a too high degree D. This means that a k step random walk on G' requires $O(k \log D)$ random bits. However, as we saw in Section 7.B.3, sometimes we can use fewer random bits if we use an expander. So a natural idea is to generate the edge labels for the walk by taking a walk using a smaller expander G' that has D vertices and degree $d \ll D$. The definition of $G \,\textcircled{R}\, G'$ is motivated by this intuition: a random walk on $G \,\textcircled{R}\, G'$ is roughly equivalent to using an expander walk on G' to generate labels for a walk on G. In particular, each step a walk over $G \,\textcircled{R}\, G'$ can be thought of as tossing a coin and then, based on its outcome, either taking a a random step on G', or using the current vertex of G' as an edge label to take a step on G. Another way to gain intuition on the replacement product is to solve Exercise 16, that analyzes the *combinatorial* (edge) expansion of the resulting graph as a function of the edge expansion of the input graphs.

PROOF OF LEMMA 16.30: Let A (resp. A') denote the $n \times n$ (resp. $D \times D$) adjacency matrix of G (resp. G') and let $\lambda(A) = 1 - \epsilon$ and $\lambda(A') = 1 - \epsilon'$. Then by Lemma 7.40, $A = (1 - \epsilon)C + J_n$ and $A' = (1 - \epsilon')C' + J_D$, where J_k is the $k \times k$ matrix with all entries equal to $1/k$.

The adjacency matrix of $G \,\textcircled{R}\, G'$ is equal to

$$\tfrac{1}{2}(A \otimes I_D) + \tfrac{1}{2}(I_n \otimes A') = \tfrac{1-\epsilon}{2}C \otimes I_D + \tfrac{\epsilon}{2}J_n \otimes I_D + \tfrac{1-\epsilon'}{2}I_n \otimes C' + \tfrac{\epsilon'}{2}I_n \otimes J_D\,,$$

where I_k is the $k \times k$ identity matrix.

Thus, the adjacency matrix of $(G \,\textcircled{R}\, G')^2$ is equal to

$$\left(\tfrac{1-\epsilon}{2}C \otimes I_D + \tfrac{\epsilon}{2}J_n \otimes I_D + \tfrac{1-\epsilon'}{2}I_n \otimes C' + \tfrac{\epsilon'}{2}I_n \otimes J_D\right)^2 =$$
$$\tfrac{\epsilon\epsilon'}{4}(J_n \otimes I_D)(I_n \otimes J_D) + \tfrac{\epsilon'\epsilon}{4}(I_n \otimes J_D)(J_n \otimes I_D) + (1 - \tfrac{\epsilon\epsilon'}{2})F\,,$$

where F is some $nD \times nD$ matrix of norm at most 1 (obtained by collecting together all the other terms in the expression). But

$$(J_n \otimes I_D)(I_n \otimes J_D) = (I_n \otimes J_D)(J_n \otimes I_D) = J_n \otimes J_D = J_{nD}\,.$$

(This can be verified by either direct calculation or by going through the graphical representation or the rotation map representation of the tensor and matrix products.)

Since every vector $\mathbf{v} \in \mathbb{R}^{nD}$ that is orthogonal to $\mathbf{1}$ satisfies $J_{nD}\mathbf{v} = \mathbf{0}$, we get that

$$\left(\lambda(G \circledR G')\right)^2 = \lambda\left((G \circledR G')^2\right) = \lambda\left((1 - \tfrac{\epsilon\epsilon'}{2})F + \tfrac{\epsilon\epsilon'}{2}J_{nD}\right) \leq 1 - \tfrac{\epsilon\epsilon'}{2},$$

and hence

$$\lambda(G \circledR G') \leq 1 - \tfrac{\epsilon\epsilon'}{4}.$$

∎

16.4.5 The actual construction.

We now use the three graph products of described above to show a strongly explicit construction of an expander graph family. Recall This is an infinite family $\{G_k\}$ of graphs (with efficient way to compute neighbors) that has a constant degree and an expansion parameter λ. The construction is recursive: we start by a finite size graph G_1 (which we can find using brute force search), and construct the graph G_k from the graph G_{k-1}. On a high level the construction is as follows: each of the three product will serve a different purpose in the construction. The *Tensor product* allows us to take G_{k-1} and increase its number of vertices, at the expense of increasing the degree and possibly some deterioration in the expansion. The *replacement product* allows us to dramatically reduce the degree at the expense of additional deterioration in the expansion. Finally, we use the *Matrix/Path product* to regain the loss in the expansion at the expense of a mild increase in the degree.

THEOREM 16.31 (EXPLICIT CONSTRUCTION OF EXPANDERS)
There exists a strongly-explicit λ, d-expander family for some constants d and $\lambda < 1$.

PROOF: Our expander family will be the following family $\{G_k\}_{k \in \mathbb{N}}$ of graphs:

- Let H be a $(D = d^{40}, d, 0.01)$-graph, which we can find using brute force search. (We choose d to be a large enough constant that such a graph exists)

- Let G_1 be a $(D, d^{20}, 1/2)$-graph, which we can find using brute force search.

- For $k > 1$, let $G_k = ((G_{k-1} \otimes G_{k-1}) \circledZ H)^{20}$.

The proof follows by noting the following points:

1. For every k, G_k has at least 2^{2^k} vertices.

    Indeed, if n_k denotes the number of vertices of G_k, then $n_k = (n_{k-1})^2 D$. If $n_{k-1} \geq 2^{2^{k-1}}$ then $n_k \geq \left(2^{2^{k-1}}\right)^2 = 2^{2^k}$.

2. For every k, the degree of G_k is d^{20}.

   Indeed, taking a replacement produce with H reduces the degree to d, which is then increased to d^{20} by taking the 20th power of the graph (using the matrix/path product).

3. There is a $2^{O(k)}$-time algorithm that given a label of a vertex u in G_k and an index $i \in [d^{20}]$, outputs the ith neighbor of u in G_k. (Note that this is polylogarithmic in the number of vertices.)

   Indeed, such a recursive algorithm can be directly obtained from the definition of G_k. To compute G_k's neighborhood function, the algorithm will make 40 recursive calls to G_{k-1}'s neighborhood function, resulting in $2^{O(k)}$ running time.

4. For every k, $\lambda(G_k) \leq 1/3$.

   Indeed, by Lemmas 16.28, 16.29, and 16.30 If $\lambda(G_{k-1}) \leq 1/3$ then $\lambda(G_{k-1} \otimes G_{k-1}) \leq 2/3$ and hence $\lambda((G_{k-1} \otimes G_{k-1})\circledR H) \leq 1 - \frac{0.99}{12} \leq 1 - 1/13$. Thus, $\lambda(G_k) \leq (1 - 1/13)^{20} \sim e^{-20/13} \leq 1/3$.

∎

Using graph powering we can obtain such a construction for *every* constant $\lambda \in (0,1)$, at the expense of a larger degree. There is a variant of the above construction supplying a *denser* family of graphs that contains an n-vertex graph for every n that is a power of c, for some constant c. Since one can transform an (n, d, λ)-graph to an (n', cd', λ)-graph for any $n/c \leq n' \leq n$ by making a single "mega-vertex" out of a set of at most c vertices, the following theorem is also known:

THEOREM 16.32
There exist constants $d \in \mathbb{N}$, $\lambda < 1$ and a strongly-explicit graph family $\{G_n\}_{n \in \mathbb{N}}$ such that G_n is an (n, d, λ)-graph for every $n \in \mathbb{N}$.

REMARK 16.33
As mentioned above, there are known constructions of expanders (typically based on number theory) that are more efficient in terms of computation time and relation between degree and the parameter λ than the product-based construction above. However, the proofs for these constructions are more complicated and require deeper mathematical tools. Also, the replacement product (and its close cousin, the zig-zag product) have found applications beyond the constructions of expander graphs. One such application is the deterministic logspace algorithm for undirected connectivity described in the next section. Another application is a construction of combinatorial expanders with greater expansion that what is implied by the parameter λ. (Note that even for for the impossible to achieve value of $\lambda = 0$, Theorem ?? implies combinatorial expansion only $1/2$ while it can be shown that a random graph has combinatorial expansion close to 1.)

16.5 Deterministic logspace algorithm for undirected connectivity.

This section describes a recent result of Reingold, showing that at least the most famous randomized logspace algorithm, the random walk algorithm for s-t-connectivity in undirected graphs (

Chapter 7) can be completely "derandomized." Thus the *s-t*-connectivity problem in undirected graphs is in **L**.

THEOREM 16.34 (REINGOLD'S THEOREM [**?**])
UPATH $\in$ **L**.

Reingold describes a set of poly(n) walks starting from s such that if s is connected to t then one of the walks is guaranteed to hit t. Of course, the *existence* of such a small set of walks is trivial; this arose in our discussion of universal traversal sequences of Definition **??**. The point is that Reingold's enumeration of walks can be carried out deterministically in logspace.

In this section, all graphs will be *multigraphs*, of the form $G = (V, E)$ where E is a *multiset* (i.e., some edges may appear multiple times, and each appearance is counted separately). We say the graph is *d-regular* if for each vertex i, the number of edges incident to it is exactly d. We will assume that the input graph for the *s-t* connectivity problem is d-regular for say $d = 4$. This is without loss of generality: if a vertex has degree $d'' < 3$ we add a self-loop of multiplicity to bring the degree up to d, and if the vertex has degree $d' \geq 3$ we can replace it by a cycle of d' vertices, and each of the d' edges that were incident to the old vertex then attach to one of the cycle nodes. Of course, the logspace machine does not have space to store the modified graph, but it can pretend that these modifications have taken place, since it can perform them on the fly whenever it accesses the graph. (Formally speaking, the transformation is implicitly computable in logspace; see Claim **??**.) In fact, the proof below will perform a series of other local modifications on the graph, each with the property that the logspace algorithm can perform them on the fly.

Recall that checking connectivity in *expander* graphs is easy. Specifically, if every connected component in G is an expander, then there is a number $\ell = O(\log n)$ such that if s and t are connected then they are connected with a path of length at most ℓ.

THEOREM 16.35
If an n-vertex graph G is d-regular graph and $\lambda(G) < 1/4$ then the maximum distance between every pair of nodes is at most $O(d \log n)$.

PROOF: The exercises ask you to prove that for each subset S of size at most $|V|/2$, the number of edges between S and $\overline{S}$ is at least $(1 - \lambda)|S|/2 \geq 3|S|/8$. Thus at least $3|S|/8d$ vertices in $\overline{S}$ must be neighbors of vertices in S. Iterating this argument l times we conclude the following about the number of vertices whose distance to S is at most l: it is either more than $|V|/2$ (when the abovementioned fact stops applying) or at least $(1 + \frac{3}{8d})^l$. Let s, t be any two vertices. Using $S = \{s\}$, we see that at least $|V|/2 + 1$ vertices must be within distance $l = 10d \log n$ of s. The same is true for vertex t. Every two subsets of vertices of size at least $|V|/2 + 1$ necessarily intersect, so there must be some vertex within distance l of both s and t. Hence the distance from s to t is at most $2l$. ∎

We can enumerate over all ℓ-step random walks of a d-degree graph in $O(d\ell)$ space by enumerating over all sequences of indices $i_1, \ldots, i_\ell \in [d]$. Thus, in a constant-degree graph where all connected components are expanders we can check connectivity in logarithmic space.

The idea behind Reingold's algorithm is to transform the graph G (in an implicitly computable in logspace way) to a graph G' such that every connected component in G becomes an expander in G', but two vertices that were not connected will stay unconnected.

By adding more self-loops we may assume that the graph is of degree d^{20} for some constant d that is sufficiently large so that there exists a $(d^{20}, d, 0.01)$-graph H. (See Fact **??** in the Appendix.) Since the size of H is some constant, we assume the algorithm has access to it (either H could be "hardwired" into the algorithm or the algorithm could perform brute force search to discover it). Consider the following sequence of transformations.

- Let $G_0 = G$.

- For $k \geq 1$, we define $G_k = (G_{k-1} \textcircled{R} H)^{20}$.

Here $\textcircled{R}$ is the replacement product of the graph, defined in Chapter **??**. If G_{k-1} is a graph with degree d^{20}, then $G_{k-1} \textcircled{R} H$ is a graph with degree d and thus $G_k = (G_{k-1} \textcircled{R} H)^{20}$ is again a graph with degree d^{20} (and size $(2d^{20} |G_{k-1}|)^{20}$). Note also that if two vertices were connected (resp., disconnected) in G_{k-1}, then they are still connected (resp., disconnected) in G_k. Thus to solve the UPATH in G it suffices to solve a UPATH problem in any of the G_k's.

Now we show that for $k = O(\log n)$, the graph G_k is an expander, and therefore an easy instance of UPATH. By Lemmas 16.28 and 16.30, for every $\epsilon < 1/20$ and D-degree graph F, if $\lambda(F) \leq 1-\epsilon$ then $\lambda(F \textcircled{R} H) \leq 1 - \epsilon/5$ and hence $\lambda\big((F \textcircled{R} H)^{20}\big) \leq 1 - 2\epsilon$. By Lemma 7.28, every connected component of G has expansion parameter at most $1 - 1/(8Dn^3)$, where n denotes the number of G's vertices which is at least as large as the number of vertices in the connect component. It follows that for $k = 10 \log D \log N$, in the graph G_k every connected component has expansion parameter at most $\max\{1 - 1/20, 2^k/(8Dn^3)\} = 1 - 1/20$.

To finish, we show how to solve the UPATH problem for G_k in logarithmic space for this value of k. The catch is of course that the graph we are given is G, not G_k. Given G, we wish to enumerate length ℓ starting from a given vertex in G_k since the graph is an expander. A walk describes, for each step, which of the d^{20} outgoing edges to take from the current vertex. Thus it suffices to show how we can compute in $O(k + \log n)$ space, the ith outgoing edge of a given vertex u in G_k. This map's input length is $O(k + \log n)$ and hence we can assume it is placed on a read/write tape, and will compute the rotation map "in-place" changing the input to the output. Let s_k be the additional space (beyond the input) required to compute the rotation map of G_k. Note that $s_0 = O(\log n)$. We show a recursive algorithm to compute G_k satisfying the equation $s_k = s_{k-1} + O(1)$. In fact, the algorithm will be a pretty straightforward implementation of the definitions of the replacement and matrix products.

The input to $\hat{G}_k$ is a vertex in $(G_{k-1} \textcircled{R} H)$ and 20 labels of edges in this graph. If we can compute the rotation map of $G_{k-1} \textcircled{R} H$ in $s_{k-1} + O(1)$ space then we can do so for $\hat{G}_k$, since we can simply make 20 consecutive calls to this procedure, each time reusing the space.[6] Now, to compute the rotation map of $(G_{k-1} \textcircled{R} H)$ we simply follow the definition of the replacement product. Given

---

[6]One has to be slightly careful while making recursive calls, since we don't want to lose even the $O(\log \log n)$ bits of writing down k and keeping an index to the location in the input we're working on. However, this can be done by keeping k in global read/write storage and since storing the identity of the current step among the 50 calls we're making only requires $O(1)$ space.

an input of the form u, v, i, b (which we think of as read/write variables), if $b = 0$ then we apply the rotation map of H to (v, i) (can be done in constant space), while if $b = 1$ then we apply the rotation map of G_{k-1} to (u, v) using a recursive call at the cost of s_{k-1} space (note that u, v are conveniently located consecutively at the beginning of the input tape).

16.6 Weak Random Sources and Extractors

Suppose, that despite the philosophical difficulties, we are happy with probabilistic algorithms, and see no need to "derandomize" them, especially at the expense of some unproven assumptions. We still need to tackle the fact that real world sources of randomness and unpredictability rarely, if ever, behave as a sequence of perfectly uncorrelated and unbiased coin tosses. Can we still execute probabilistic algorithms using real-world "weakly random" sources?

16.6.1 Min Entropy

For starters, we need to define what we mean by a weakly random source.

DEFINITION 16.36
Let X be a random variable. The *min entropy* of X, denoted by $H_\infty(X)$, is the largest real number k such that $\Pr[X = x] \leq 2^{-k}$ for every x in the range of X.
    If X is a distribution over $\{0, 1\}^n$ with $H_\infty(X) \geq k$ then it is called an (n, k)-*source*.

It is not hard to see that if X is a random variable over $\{0, 1\}^n$ then $H_\infty(X) \leq n$ with $H_\infty(X) = n$ if and only if X is distributed according to the uniform distribution U_n. Our goal in this section is to be able to execute probabilistic algorithms given access to a distribution X with $H_\infty(X)$ as small as possible. It can be shown that min entropy is a *minimal requirement* in the sense that in general, to execute a probabilistic algorithm that uses k random bits we need access to a distribution X with $H_\infty(X) \geq k$ (see Exercise **??**).

EXAMPLE 16.37
Here are some examples for distributions X over $\{0, 1\}^n$ and their min-entropy:

- (Bit fixing and generalized bit fixing sources) If there is subset $S \subseteq [n]$ with $|S| = k$ such that X's projection to the coordinates in S is uniform over $\{0, 1\}^k$, and X's projection to $[n] \setminus S$ is a fixed string (say the all-zeros string) then $H_\infty(X) = k$. The same holds if X's projection to $[n] \setminus S$ is a fixed deterministic function of its projection to S. For example, if the bits in the odd positions of X are independent and uniform and for every even position $2i$, $X_{2i} = X_{2i-1}$ then $H_\infty(X) = \lceil \frac{n}{2} \rceil$. This may model a scenario where we measure some real world data at too high a rate (think of measuring every second a physical event that changes only every minute).

- (Linear subspaces) If X is the uniform distribution over a linear subspace of $\mathrm{GF}(2)^n$ of dimension k, then $H_\infty(X) = k$. (In this case X is actually a generalized bit-fixing source— can you see why?)

- (Biased coins) If X is composed of n independent coins, each outputting 1 with probability $\delta < 1/2$ and 0 with probability $1 - \delta$, then as n grows, $H_\infty(X)$ tends to $H(\delta)n$ where H is the Shannon entropy function. That is, $H(\delta) = \delta \log \frac{1}{\delta} + (1 - \delta) \log \frac{1}{1-\delta}$.

- (Santha-Vazirani sources) If X has the property that for every $i \in [n]$, and every string $x \in \{0,1\}^{i-1}$, conditioned on $X_1 = x_1, \ldots, X_{i-1} = x_{i-1}$ it holds that both $\Pr[X_i = 0]$ and $\Pr[X_i = 1]$ are between δ and $1 - \delta$ then $H_\infty(X) \geq H(\delta)n$. This can model sources such as stock market fluctuations, where current measurements do have some limited dependence on the previous history.

- (Uniform over subset) If X is the uniform distribution over a set $S \subseteq \{0,1\}^n$ with $|S| = 2^k$ then $H_\infty(X) = k$. As we will see, this is a very general case that "essentially captures" all distributions X with $H_\infty(X) = k$.

We see that min entropy is a pretty general notion, and distributions with significant min entropy can model many real-world sources of randomness.

16.6.2 Statistical distance and Extractors

Now we try to formalize what it means to *extract* random —more precisely, almost random— bits from an (n, k) source. To do so we will need the following way of quantifying when two distributions are close.

DEFINITION 16.38 (STATISTICAL DISTANCE)
For two random variables X and Y with range $\{0,1\}^m$, their *statistical distance* (also known as *variation distance*) is defined as $\delta(X, Y) = \max_{S \subseteq \{0,1\}^m} \{|\mathbf{Pr}[X \in S] - \mathbf{Pr}[Y \in S]|\}$. We say that X, Y are ϵ-*close*, denoted $X \approx_\epsilon Y$, if $\delta(X, Y) \leq \epsilon$.

Statistical distance lies in $[0, 1]$ and satisfies triangle inequality, as suggested by its name. The next lemma gives some other useful properties; the proof is left as an exercise.

LEMMA 16.39
Let X, Y be any two distributions taking values in $\{0,1\}^n$.

1. $\delta(X, Y) = \frac{1}{2} \sum_{x \in \{0,1\}^n} |\mathbf{Pr}[X = x] - \mathbf{Pr}[Y = x]|$.

2. (Restatement of Definition 16.38) $\delta(X, Y) \geq \epsilon$ iff there is a boolean function $D : \{0,1\}^m \to \{0,1\}$ such that $|\Pr_{x \in X}[D(x) = 1] - \Pr_{y \in Y}[D(y) = 1]| \geq \epsilon$.

3. If $f : \{0,1\}^n \to \{0,1\}^s$ is any function, then $\delta(f(X), f(Y)) \leq \delta(X, Y)$. (Here $f(X)$ is a distribution on $\{0,1\}^s$ obtained by taking a sample of X and applying f.)

Now we define an extractor. This is a (deterministic) function that transforms an (n, k) source into an almost uniform distribution. It uses a small number of additional truly random bits, denoted by t in the definition below.

DEFINITION 16.40
A function $\mathsf{Ext} : \{0,1\}^n \times \{0,1\}^t \to \{0,1\}^m$ is a (k, ϵ) extractor if for any (n, k)-source X, the distribution $\mathsf{Ext}(X, U_t)$ is ϵ-close to U_m. (For every ℓ, U_ℓ denotes the uniform distribution over $\{0,1\}^\ell$.)

Equivalently, if $\mathsf{Ext} : \{0,1\}^n \times \{0,1\}^t \to \{0,1\}^m$ is a (k, ϵ) extractor, then for every distribution X ranging over $\{0,1\}^n$ of min-entropy k, and for every $S \subseteq \{0,1\}^m$, we have

$$|\mathbf{Pr}_{a \in X, z \in \{0,1\}^t}[\mathsf{Ext}(a, z) \in S] - \mathbf{Pr}_{r \in \{0,1\}^m}[r \in S]| \le \epsilon$$

We use this fact to show in Section 16.7.2 how to use extractors and (n, k)-sources to to simulate any probabilistic computation.

Why an additional input? Our stated motivation for extractors is to execute probabilistic algorithms without access to perfect unbiased coins. Yet, it seems that an extractor is not sufficient for this task, as we only guarantee that its output is close to uniform if it is given an additional input that is uniformly distributed. First, we note that the requirement of an additional input is necessary: for every function $\mathsf{Ext} : \{0,1\}^n \to \{0,1\}^m$ and every $k \le n - 1$ there exists an (n, k)-source X such that the first bit of $\mathsf{Ext}(X)$ is constant (i.e, is equal to some value $b \in \{0,1\}$ with probability 1), and so is at least of statistical distance 1/2 from the uniform distribution (Exercise 7). Second, if the length t of the second input is sufficiently short (e.g., $t = O(\log n)$) then, for the purposes of simulating probabilistic algorithms, we can do without any access to true random coins, by enumerating over all the 2^t possible inputs (see Section 16.7.2). Clearly, t has to be somewhat short for the extractor to be non-trivial: for $t \ge m$, we can have a trivial extractor that ignores its first input and outputs the second input. This second input is called the *seed* of the extractor.

16.6.3 Extractors based upon hash functions

One can use pairwise independent (and even weaker notions of) hash functions to obtain extractors. In this section, $\mathcal{H}$ denotes a family of hash functions $h : \{0,1\}^n \to \{0,1\}^k$. We say it has *collision error* δ if for any $x_1 \ne x_2 \in \{0,1\}^n$, $\Pr_{h \in \mathcal{H}}[h(x_1) = h(x_2)] \le (1 + \delta)/2^k$. We assume that one can choose a random function $h \in \mathcal{H}$ by picking a string at random from $\{0,1\}^t$. We define the extractor $\mathsf{Ext} : \times \{0,1\}^t \to \{0,1\}^{k+t}$ as follows:

$$\mathsf{Ext}(x, h) = h(x) \circ h, \tag{17}$$

where $\circ$ denotes concatenation of strings.

To prove that this is an extractor, we relate the min-entropy to the collision probability of a distribution, which is defined as $\sum_a p_a^2$, where p_a is the probability assigned to string a.

LEMMA 16.41
If a distribution X has min-entropy at least k then its collision probability is at most $1/2^k$.

PROOF: For every a in X's range, let p_a be the probability that $X = a$. Then, $\sum_a p_a^2 \le \max_a \{p_a\} (\sum_a p_a) \le \frac{1}{2^k} \cdot 1 = \frac{1}{2^k}$. ∎

LEMMA 16.42 (LEFTOVER HASH LEMMA)
If x is chosen from a distribution on $\{0,1\}^n$ with min-entropy at least k/δ and $\mathcal{H}$ has collision error δ, then $h(X) \circ h$ has distance at most $\sqrt{2\delta}$ to the uniform distribution.

PROOF: Left as exercise. (Hint: use the relation between the L_2 and L_1 norms ∎

16.6.4   Extractors based upon random walks on expanders

This section assumes knowledge of random walks on expanders, as described in Chapter **??**.

LEMMA 16.43
Let $\epsilon > 0$. For every n and $k \leq n$ there exists a (k, ϵ)-extractor $\mathsf{Ext} : \{0,1\}^n \times \{0,1\}^t \to \{0,1\}^n$ where $t = O(n - k + \log 1/\epsilon)$.

PROOF: Suppose X is an (n, k)-source and we are given a sample a from it. Let G be a $(2^n, d, 1/2)$-graph for some constant d (see Definition 7.31 and Theorem 16.32).

Let z be a truly random seed of length $t = 10 \log d(n - k + \log 1/\epsilon) = O(n - k + \log 1/\epsilon)$. We interpret z as a random walk in G of length $10(n - k + \log 1/\epsilon)$ starting from the node whose label is a. (That is, we think of z as $10(n - k + \log 1/\epsilon)$ labels in $[d]$ specifying the steps taken in the walk.) The output $\mathsf{Ext}(a, z)$ of the extractor is the label of the final node on the walk.

We have $\|X - \mathbf{1}\|_2^2 \leq \|X\|_2^2 = \sum_a \Pr[X = a]^2$, which is at most 2^{-k} by Lemma 16.41 since X is an (n, k)-source. Therefore, after a random walk of length t the distance to the uniform distribution is (by the upperbound in (**??**)):

$$\|M^t X - \frac{1}{2^N}\mathbf{1}\|_1 \leq \lambda_2^t \|X - \frac{1}{2^N}\mathbf{1}\|_2 \sqrt{2^N} \leq \lambda_2^t 2^{(N-k)/2}.$$

When t is a sufficiently large multiple of $N - k + \log 1/\varepsilon$, this distance is smaller than ε. ∎

16.6.5   An extractor based upon Nisan-Wigderson

THIS SECTION IS STILL QUITE ROUGH

Now we describe an elegant construction of extractors due to Trevisan.

Suppose we are given a string x obtained from an (N, k)-source. How can we extract k random bits from it, given $O(\log N)$ truly random bits? Let us check that the trivial idea fails. Using $2 \log N$ random bits we can compute a set of k (where $k < N - 1$) indices that are uniformly distributed and pairwise independent. Maybe we should just output the corresponding bits of x? Unfortunately, this does not work: the source is allowed to set $N - k$ bits (deterministically) to 0 so long as the remaining k bits are completely random. In that case the expected number of random bits in our sample is at most k^2/N, which is less than even 1 if $k < \sqrt{N}$.

This suggests an important idea: we should first apply some transformation on x to "smear out" the randomness, so it is not localized in a few bit positions. For this, we will use error-correcting codes. Recall that such codes are used to introduce error-tolerance when transmitting messages over noisy channels. Thus intuitively, the code must have the property that it "smears" every bit of the message all over the transmitted message.

Having applied such an encoding to the weakly random string, the construction selects bits from it using a better sampling method than pairwise independent sampling, namely, the Nisan-Wigderson combinatorial design.

Nisan-Wigderson as a sampling method:

In (**??**) we defined a function $NW_{f,\mathcal{S}}(z)$ using any function $f : \{0,1\}^l \rightarrow \{0,1\}$ and a combinatorial design $\mathcal{S}$. Note that the definition works for every function, not just hard-to-compute functions. Now we observe that $NW_{f,\mathcal{S}}(z)$ is actually a way to sample entries from the truth table of f.

Think of f as a bitstring of length 2^l, namely, its truth table. (Likewise, we can think of any circuit with l-bit inputs and with $0/1$ outputs as computing a string of length 2^l.) Given any z ("the seed"), $NW_{f,\mathcal{S}}(z)$ is just a method to use z to sample a sequence of m bits from f. This is completely analogous to pairwise independent sampling considered above; see Figure **??**.

Figure unavailable in pdf file.

Figure 16.3: Nisan-Wigderson as a sampling method: An (l, α)-design $(S_1, S_2, \ldots, S_m)$ where each $S_i \subseteq [t], |S_i| = l$ can be viewed as a way to use $z \in \{0,1\}^t$ to sample m bits from any string of length 2^l, which is viewed as the truth table of a function $f : \{0,1\}^l \rightarrow \{0,1\}$.

List-decodable codes

The construction will use the following kind of codes.

DEFINITION 16.44
If $\delta > 0$, a mapping $\sigma : \{0,1\}^N \rightarrow \{0,1\}^{\bar{N}}$ is called *an error-correcting code that is list-decodable up to error $1/2 - \delta$* if for every $w \in \{0,1\}^{\bar{N}}$, the number of $y \in B^N$ such that $w, \sigma(y)$ disagree in at most $1/2 - \delta$ fraction of bits is at most $1/\delta^2$.

The set $\left\{ \sigma(x) : x \in \{0,1\}^N \right\}$ is called the set of *codewords*.

The name "list-decodable" owes to the fact that if we transmit x over a noisy channel after first encoding with σ then even if the channel flips $1/2 - \delta$ fraction of bits, there is a small "list" of y that the received message could be decoded to. (Unique decoding may not be possible, but this will be of no consequence in the construction below.) The exercises ask you to prove that list-decodable codes exist with $\bar{N} = \text{poly}(N, 1/\delta)$, where σ is computable in polynomial time.

Trevisan's extractor:

Suppose we are given an (N, k)-source. We fix $\sigma : \{0,1\}^N \rightarrow \{0,1\}^{\bar{N}}$, a polynomial-time computable code that is list-decodable upto to error $1/2 - \epsilon/m$. We assume that $\bar{N}$ is a power of 2 and let $l = \log_2 \bar{N}$. Now every string $x \in \{0,1\}^{\bar{N}}$ may be viewed as a boolean function $< x >: \{0,1\}^{\log \bar{N}} \rightarrow \{0,1\}$ whose truth table is x. Let $\mathcal{S} = (S_1, \ldots, S_m)$ be a $(l, \log m)$ design over $[t]$.

The extractor $ExtNW : \{0,1\}^N \times \{0,1\}^t \rightarrow \{0,1\}^m$ is defined as

$$ExtNW_{\sigma,\mathcal{S}}(x,z) = NW_{<\sigma(x)>,\mathcal{S}}(z) \ .$$

That is, $ExtNW$ encodes its first ("weakly random") input x using an error-correcting code, then uses Nisan-Wigderson sampling on the resulting string using the second ("truly random") input z as a seed.

LEMMA 16.45
For sufficiently large m and for $\epsilon > 2^{-m^2}$, $ExtNW_{\sigma,\mathcal{S}}$ is a $(m^3, 2\epsilon)$-extractor.

PROOF: Let X be an (N,k) source where the min-entropy k is m^3. To prove that the distribution $ExtNW(a,z)$ where $a \in X, z \in \{0,1\}^t$ is close to uniform, it suffices (see our remarks after Definition 16.38) to show for each function $D : \{0,1\}^m \to \{0,1\}$ that

$$\left| \mathbf{Pr}_r[D(r)=1] - \mathbf{Pr}_{a\in X, z\in\{0,1\}^t}[D(ExtNW(a,z))=1] \right| \leq 2\epsilon. \tag{18}$$

For the rest of this proof, we fix an arbitrary D and prove that (18) holds for it.

The role played by this test D is somewhat reminiscent of that played by the distinguisher algorithm in the definition of a pseudorandom generator, except, of course, D is allowed to be arbitrarily inefficient. This is why we will use the black-box version of the Nisan-Wigderson analysis (Corollary ??), which does not care about the complexity of the distinguisher.

Let B be the set of bad a's for this D, where string $a \in X$ is *bad for D* if

$$\left| \mathbf{Pr}[D(r)=1] - \mathbf{Pr}_{z\in\{0,1\}^t}[D(ExtNW(a,z))=1] \right| > \epsilon.$$

We show that B is small using a counting argument: we exhibit a 1-1 mapping from the set of bad a's to another set G, and prove G is small. Actually, here is G:

$$G = \left\{ \text{circuits of size } O(m^2) \right\} \times \{0,1\}^{2\log(m/\epsilon)} \times \{0,1\}^2 \ .$$

The number of circuits of size $O(m^2)$ is $2^{O(m^2 \log m)}$, so $|G| \leq 2^{O(m^2 \log m)} \times 2(m/\epsilon)^2 = 2^{O(m^2 \log m)}$.

Let us exhibit a 1-1 mapping from B to G. When a is bad, Corollary ?? implies that there is a circuit C of size $O(m^2)$ such that either the circuit $D(C())$ or its negation –XORed with some fixed bit b—agrees with $\sigma(a)$ on a fraction $1/2 + \epsilon/m$ of its entries. (The reason we have to allow either $D(C())$ or its complement is the $|\cdot|$ sign in the statement of Corollary ??.) Let $w \in \{0,1\}^{\bar{N}}$ be the string computed by this circuit. Then $\sigma(a)$ disagrees with w in at most $1/2 - \epsilon/m$ fraction of bits. By the assumed property of the code σ, at most $(m/\epsilon)^2$ other codewords have this property. Hence a is completely specified by the following information: (a) circuit C; this is specified by $O(m^2 \log m)$ bits (b) whether to use $D(C())$ or its complement to compute w, and also the value of the unknown bit b; this is specified by 2 bits (c) which of the $(m/\epsilon)^2$ codewords around w to pick as $\sigma(a)$; this is specified by $\lceil 2\log(m/\epsilon) \rceil$ bits assuming the codewords around w are ordered in some canonical way. Thus we have described the mapping from B to G.

We conclude that for any fixed D, there are at most $2^{O(m^2 \log m)}$ bad strings. The probability that an element a taken from X is bad for D is (by Lemma ??) at most $2^{-m^3} \cdot 2^{O(m^2 \log m)} < \epsilon$ for

sufficiently large m. We then have

$$
\Big| \mathbf{Pr}_r[D(r) = 1] - \mathbf{Pr}_{a \in X, z \in \{0,1\}^t}[D(ExtNW(a,z)) = 1] \Big|
$$
$$
\leq \quad \sum_a \mathbf{Pr}[X = a] \Big| \mathbf{Pr}[D(r) = 1] - \mathbf{Pr}_{z \in \{0,1\}^t}[D(ExtNW(a,z)) = 1] \Big|
$$
$$
\leq \quad \mathbf{Pr}[X \in B] + \epsilon \leq 2\epsilon,
$$

where the last line used the fact that if $a \notin B$, then by definition of B,
$\Big| \mathbf{Pr}[D(r) = 1] - \mathbf{Pr}_{z \in \{0,1\}^t}[D(ExtNW(a,z)) = 1] \Big| \leq \epsilon.$ ∎

The following theorem is an immediate consequence of the above lemma.

THEOREM 16.46
Fix a constant ϵ; for every N and $k = N^{\Omega(1)}$ there is a polynomial-time computable (k, ϵ)-extractor
$\mathsf{Ext} : \{0,1\}^N \times \{0,1\}^t \to \{0,1\}^m$ *where $m = k^{1/3}$ and $t = O(\log N)$.*

16.7 Applications of Extractors

Extractors are deterministic objects with strong pseudorandom properties. We describe a few important uses for them; many more will undoubtedly be found in future.

16.7.1 Graph constructions

An extractor is essentially a graph-theoretic object; see Figure **??**. (In fact, extractors have been used to construct expander graphs.) Think of a (k, ϵ) extractor $\mathsf{Ext} : \{0,1\}^N \times \{0,1\}^t \to \{0,1\}^m$ as a bipartite graph whose left side contains one node for each string in $\{0,1\}^N$ and the right side contains a node for each string in $\{0,1\}^m$. Each node a on the left is incident to 2^t edges, labelled with strings in $\{0,1\}^t$, with the right endpoint of the edge labeled with z being $\mathsf{Ext}(a, z)$.

An (N, k)-source corresponds to any distribution on the left side with min-entropy at least k. The extractor's definition implies that picking a node according to this distribution and a random outgoing edge gives a node on the right that is essentially uniformly distributed.

Figure unavailable in pdf file.

Figure 16.4: An extractor $\mathsf{Ext} : \{0,1\}^N \times \{0,1\}^T \to \{0,1\}^m$ defines a bipartite graph where every node on the left has degree 2^T.

This implies in particular that for every set X on the left side of size exactly 2^k —notice, this is a special case of an (N, k)-source— its neighbor set $\Gamma(X)$ on the right satisfies $|\Gamma(X)| \geq (1 - \epsilon)2^m$.

One can in fact show a converse, that high expansion implies that the graph is an extractor; see Chapter notes.

16.7.2 Running randomized algorithms using weak random sources

We now describe how to use extractors to simulate probabilistic algorithms using weak random sources. Suppose that $A(\cdot, \cdot)$ is a probabilistic algorithm that on an input of length n uses $m = m(n)$ random bits, and suppose that for every x we have $\mathbf{Pr}_r[A(x, r) = \text{ right answer }] \geq 3/4$. If A's answers are $0/1$, then such algorithms can be viewed as defining a **BPP** language, but here we allow a more general scenario. Suppose $\mathsf{Ext} : \{0,1\}^N \times \{0,1\}^t \to \{0,1\}^m$ is a $(k, 1/4)$-extractor.

Consider the following algorithm A': on input $x \in \{0,1\}^n$ and given a string $a \in \{0,1\}^N$ from the weakly random source, the algorithm enumerates all choices for the seed z and computes $A(x, \mathsf{Ext}(a, z))$. Let

$$A'(x, a) = \text{majority value of } \left\{ A(x, \mathsf{Ext}(a, z)) : z \in \{0,1\}^t \right\} \tag{19}$$

The running time of A' is approximately 2^t times that of A. We show that if a comes from an $(n, k+2)$ source, then A' outputs the correct answer with probability at least $3/4$.

Fix the input x. Let $R = \{ r \in \{0,1\}^m : A(x, r) = \text{ right answer }\}$, and thus $|R| \geq \frac{3}{4} 2^m$. Let B be the set of strings $a \in \{0,1\}^N$ for which the majority answer computed by algorithm A' is incorrect, namely,

$$B = \left\{ a : \mathbf{Pr}_{z \in \{0,1\}^t}[A(x, \mathsf{Ext}(a, z)) = \text{ right answer}] < 1/2 \right\}$$
$$= \left\{ a : \mathbf{Pr}_{z \in \{0,1\}^t}[\mathsf{Ext}(a, z) \in R] < 1/2 \right\}$$

CLAIM: $|B| \leq 2^k$.

Let random variable Y correspond to picking an element uniformly at random from B. Thus Y has min-entropy $\log B$, and may be viewed as a $(N, \log B)$-source. By definition of B,

$$\mathbf{Pr}_{a \in Y, z \in \{0,1\}^t}[\mathsf{Ext}(a, z) \in R] < 1/2.$$

But $|R| = \frac{3}{4} 2^m$, so we have

$$\left| \mathbf{Pr}_{a \in Y, z \in \{0,1\}^t}[\mathsf{Ext}(a, z) \in R] - \mathbf{Pr}_{r \in \{0,1\}^m}[r \in R] \right| > 1/4,$$

which implies that the statistical distance between the uniform distribution and $\mathsf{Ext}(Y, z)$ is at least $1/4$. Since Ext is a $(k, 1/4)$-extractor, Y must have min-entropy less than k. Hence $|B| \leq 2^k$ and the Claim is proved.

The correctness of the simulation now follows since

$$\mathbf{Pr}_{a \in X}[A'(x, a) = \text{ right answer }] = 1 - \mathbf{Pr}_{a \in X}[a \in B]$$
$$\geq 1 - 2^{-(k+2)} \cdot |B| \geq 3/4, \quad \text{(by Lemma ??)}.$$

Thus we have shown the following.

THEOREM 16.47
Suppose A is a probabilistic algorithm running in time $T_A(n)$ and using $m(n)$ random bits on inputs of length n. Suppose we have for every $m(n)$ a construction of a $(k(n), 1/4)$-extractor $\mathsf{Ext}_n : \{0,1\}^N \times \{0,1\}^{t(n)} \to \{0,1\}^{m(n)}$ running in $T_E(n)$ time. Then A can be simulated in time $2^t(T_A + T_E)$ using one sample from a $(N, k+2)$ source.

16.7.3 Recycling random bits

We addressed the issue of recycling random bits in Section **??**. An extractor can also be used to recycle random bits. (Thus it should not be surprising that random walks on expanders, which were used to recycle random bits in Section **??**, were also used to construct extractors above.)

Suppose A be a randomized algorithm that uses m random bits. Let $\mathsf{Ext} : \{0,1\}^N \times \{0,1\}^{t)} \to \{0,1\}^m$ be any (k,ϵ)-extractor. Consider the following algorithm. Randomly pick a string $a \in \{0,1\}^N$, and obtain 2^t strings in $\{0,1\}^m$ obtained by computing $\mathsf{Ext}(a,z)$ for all $z \in \{0,1\}^t$. Run A for all these random strings. Note that this manages to run A as many as 2^t times while using only N random bits. (For known extractor constructions, $N \ll 2^t m$, so this is a big saving.)

Now we analyse how well the error goes down. Suppose $D \subseteq \{0,1\}^m$ be the subset of strings for which A gives the correct answer. Let $p = |D|/2^m$; for a **BPP** algorithm $p \geq 2/3$. Call an $a \in \{0,1\}^N$ *bad* if the above algorithm sees the correct answer for less than $p - \epsilon$ fraction of z's. If the set of all bad a's were to have size more than 2^k, the (N,k)-source X corresponding to drawing uniformly at random from the bad a's would satisfy

$$\Pr[\mathsf{Ext}(X, U_t) \in D] - \Pr[U_m \in D] > \epsilon,$$

which would contradict the assumption that Ext is a (k,ϵ)-extractor. We conclude that the probability that the above algorithm gets an incorrect answer from A in $p - \epsilon$ fraction of the repeated runs is at most $2^k/2^N$.

16.7.4 Pseudorandom generators for spacebounded computation

Now we describe Nisan's pseudo-random generators for space-bounded randomized computation, which allows randomized logspace computations to be run with $O(\log^2 n)$ random bits.

Throughout this section we represent logspace machines by their *configuration graph*, which has size $\text{poly}(n)$.

THEOREM 16.48 (NISAN)
For every d there is a $c > 0$ and a polynomial-time computable function $g : \{0,1\}^{c \log^2 n} \to \{0,1\}^{n^d}$ such that for every space-bounded machine M that has a configuration graph of size $\leq n^d$ on inputs of size n:

$$\left| \Pr_{r \in \{0,1\}^{n^d}}[M(x,r) = 1] - \Pr_{z \in \{0,1\}^{c \log^2 n}}[M(x,g(z)) = 1] \right| < \frac{1}{10}. \tag{20}$$

We give a proof due to Impagliazzo, Nisan, and Wigderson [INW94] (with further improvements by Raz and Reingold [RR99]) that uses extractors. Nisan's original paper did not explicitly use extractors —the definition of extractors came later and was influenced by results such as Nisan's.

In fact, Nisan's construction proves a result stronger than Theorem 16.48: there is a polynomial-time simulation of every algorithm in **BPL** using $O(\log^2 n)$ space. (See Exercises.) Note that Savitch's theorem (Theorem **??**) also implies that **BPL** $\subseteq$ **SPACE**$(\log^2 n)$, but the algorithm in Savitch's proof takes $n^{\log n}$ time. Saks and Zhou [SZ99a] improved Nisan's ideas to show that **BPL** $\subseteq$ **SPACE**$(\log^{1.5} n)$, which leads many experts to conjecture that **BPL** = **L** (i.e., randomness does not help logspace computations at all). (For partial progress, see Section **??** later.)

The main intuition behind Nisan's construction —and also the conjecture $\mathbf{BPL} = \mathbf{L}$— is that the logspace machine has one-way access to the random string and only $O(\log n)$ bits of memory. So it can only "remember" $O(\log n)$ of the random bits it has seen. To exploit this we will use the following simple lemma, which shows how to recycle a random string about which only a little information is known. (Throughout this section, $\circ$ denotes concatenation of strings.)

LEMMA 16.49 (RECYCLING LEMMA)
Let $f : \{0,1\}^n \to \{0,1\}^s$ be any function and $\mathsf{Ext} : \{0,1\}^n \times \{0,1\}^t \to \{0,1\}^m$ be a $(k, \epsilon/2)$-extractor, where $k = n - (s+1) - \log \frac{1}{\epsilon}$. When $X \in_R \{0,1\}^n$, $W \in_R \{0,1\}^m$, $z \in_R \{0,1\}^t$, then

$$f(X) \circ W \approx_\epsilon f(X) \circ \mathsf{Ext}(X, z).$$

REMARK 16.50
When the lemma is used, $s \ll n$ and $n = m$. Thus $f(X)$, which has length s, contains only a small amount of information about X. The Lemma says that using an appropriate extractor (whose random seed can have length as small as $t = O(s + \log(1/\epsilon))$ if we use Lemma 16.43) we can get a new string $\mathsf{Ext}(X, z)$ that looks essentially random, even to somebody who knows $f(X)$.

PROOF: For $v \in \{0,1\}^s$ we denote by X_v the random variable that is uniformly distributed over the set $f^{-1}(v)$. Then we can express $\| (f(X) \circ W - f(X) \circ \mathsf{Ext}(X, z) \|$ as

$$= \frac{1}{2} \sum_{v,w} \left| \Pr[f(X) = v \wedge W = w] - \Pr_z[f(X) = v \wedge \mathsf{Ext}(X, z) = w] \right|$$

$$= \sum_v \Pr[f(X) = v] \cdot \| W - \mathsf{Ext}(X_v, z) \| \tag{21}$$

Let $V = \left\{ v : \Pr[f(X) = v] \geq \epsilon/2^{s+1} \right\}$. If $v \in V$, then we can view X_v as a (n, k)-source, where $k \geq n - (s+1) - \log \frac{1}{\epsilon}$. Thus by definition of an extractor, $\mathsf{Ext}(X_v, r) \approx_{\epsilon/2} W$ and hence the contributions from $v \in V$ sum to at most $\epsilon/2$. The contributions from $v \notin V$ are upperbounded by $\sum_{v \notin V} \Pr[f(X) = v] \leq 2^s \times \frac{\epsilon}{2^{s+1}} = \epsilon/2$. The lemma follows. ∎

Now we describe how the Recycling Lemma is useful in Nisan's construction. Let M be a logspace machine. Fix an input of size n and view the graph of all configurations of M on this input as a *leveled branching program*. For some $d \geq 1$, M has $\leq n^d$ configurations and runs in time $L \leq n^d$. Assume without loss of generality —since unneeded random bits can always be ignored— that it uses 1 random bit at each step. Without loss of generality (by giving M a separate worktape that maintains a time counter), we can assume that the configuration graph is leveled: it has L levels, with level i containing configurations obtainable at time i. The first level contains only the start node and the last level contains two nodes, "accept" and "reject;" every other level has $W = n^d$ nodes. Each level i node has two outgoing edges to level $i + 1$ nodes and the machine's computation at this node involves using the next bit in the random string to pick one of these two outgoing edges. We sometimes call L the *length* of the configuration graph and W the *width*.

For simplicity we first describe how to reduce the number of random bits by a factor 2. Think of the L steps of the computation as divided in two halves, each consuming $L/2$ random bits. Suppose we use some random string X of length $L/2$ to run the first half, and the machine is now

Figure unavailable in pdf file.

Figure 16.5: Configuration graph for machine M

at node v in the middle level. The only information known about X at this point is the index of v, which is a string of length $d \log n$. We may thus view the first half of the branching program as a (deterministic) function that maps $\{0,1\}^{L/2}$ bits to $\{0,1\}^{d \log n}$ bits. The Recycling Lemma allows us to use a random seed of length $O(\log n)$ to recycle X to get an almost-random string $\mathsf{Ext}(X, z)$ of length $L/2$, which can be used in the second half of the computation. Thus we can run L steps of computation using $L/2 + O(\log n)$ bits, a saving of almost a factor 2. Using a similar idea recursively, Nisan's generator runs L steps using $O(\log n \log L)$ random bits.

Now we formally define Nisan's generator.

DEFINITION 16.51 (NISAN'S GENERATOR)
For some $r > 0$ let $\mathsf{Ext}_k : \{0,1\}^{kr} \times \{0,1\}^r \to \{0,1\}^{kr}$ be an extractor function for each $k \geq 0$. For every integer $k \geq 0$ the associated Nisan generator $G_k : \{0,1\}^{kr} \to \{0,1\}^{2^k}$ is defined recursively as (where $|a| = (k-1)r, |z| = r$)

$$G_k(a \circ z) = \begin{cases} z_1 & \text{(i.e., first bit of } z) & k = 1 \\ \\ G_{k-1}(a) \circ G_{k-1}(\mathsf{Ext}_{k-1}(a, z)) & k > 1 \end{cases}$$

Now we use this generator to prove Theorem 16.48. We only need to show that the probability that the machine goes from the start node to the "accept" node is similar for truly random strings and pseudorandom strings. However, we will prove a stronger statement involving intermediate steps as well.

If nodes u is a node in the configuration graph, and s is a string of length 2^k, then we denote by $f_{u,2^k}(s)$ the node that the machine reaches when started in u and its random string is s. Thus if s comes from some distribution $\mathcal{D}$, we can define a distribution $f_{u,2^k}(\mathcal{D})$ on nodes that are 2^k levels further from u.

THEOREM 16.52
Let $r = O(\log n)$ be such that for each $k \leq d \log n$, $\mathsf{Ext}_k : \{0,1\}^{kr} \times \{0,1\}^r \to \{0,1\}^{kr}$ is a $(kr - 2d \log n, \epsilon)$-extractor. For every machine of the type described in the previous paragraphs, and every node u in its configuration graph:

$$\| f_{u,2^k}(U_{2^k}) - f_{u,2^k}(G_k(U_{kr})) \| \leq 3^k \epsilon, \tag{22}$$

where U_l denotes the uniform distribution on $\{0,1\}^l$.

REMARK 16.53
To prove Theorem 16.48 let $u = u_0$, the start configuration, and $2^k = L$, the length of the entire computation. Choose $3^k \epsilon < 1/10$ (say), which means $\log 1/\epsilon = O(\log L) = O(\log n)$. Using the extractor of Section 16.6.4 as Ext_k, we can let $r = O(\log n)$ and so the seed length $kr = O(r \log L) = O(\log^2 n)$.

PROOF: (Theorem 16.52) Let ϵ_k denote the maximum value of the left hand side of (22) over all machines. The lemma is proved if we can show inductively that $\epsilon_k \leq 2\epsilon_{k-1} + 2\epsilon$. The case $k = 1$ is trivial. At the inductive step, we need to upperbound the distance between two distributions $f_{u,2^k}(\mathcal{D}_1)$, $f_{u,2^k}(\mathcal{D}_4)$, for which we introduce two distributions $\mathcal{D}_2, \mathcal{D}_3$ and use triangle inequality:

$$\| f_{u,2^k}(\mathcal{D}_1) - f_{u,2^k}(\mathcal{D}_4) \| \leq \sum_{i=1}^{3} \| f_{u,2^k}(\mathcal{D}_i) - f_{u,2^k}(\mathcal{D}_{i+1}) \| . \tag{23}$$

The distributions will be:

$$\mathcal{D}_1 = U_{2^k}$$
$$\mathcal{D}_4 = G_k(U_{kr})$$
$$\mathcal{D}_2 = U_{2^{k-1}} \circ G_{k-1}(U_{(k-1)r})$$
$$\mathcal{D}_3 = G_{k-1}(U_{(k-1)r}) \circ G_{k-1}(U'_{(k-1)r}) \qquad (U, U' \text{ are identical but independent}).$$

We bound the summands in (23) one by one.

Claim 1: $\| f_{u,2^k}(\mathcal{D}_1) - f_{u,2^k}(\mathcal{D}_2) \| \leq \epsilon_{k-1}$.
Denote $\Pr[f_{u,2^{k-1}}(U_{2^{k-1}}) = w]$ by $p_{u,w}$ and $\Pr[f_{u,2^{k-1}}(G_{k-1}(U_{(k-1)r})) = w]$ by $q_{u,w}$. According to the inductive assumption,

$$\frac{1}{2} \sum_w |p_{u,w} - q_{u,w}| = \| f_{u,2^{k-1}}(U_{2^{k-1}}) - f_{u,2^{k-1}}(G_{k-1}(U_{(k-1)r})) \| \leq \epsilon_{k-1}.$$

Since $\mathcal{D}_1 = U_{2^k}$ may be viewed as two independent copies of $U_{2^{k-1}}$ we have

$$\| f_{u,2^k}(\mathcal{D}_1) - f_{u,2^k}(\mathcal{D}_2) \| = \sum_v \frac{1}{2} \left| \sum_w p_{uw}p_{wv} - \sum_w p_{uw}q_{wv} \right|$$

where w, v denote nodes 2^{k-1} and 2^k levels respectively from u

$$= \sum_w p_{uw} \frac{1}{2} \sum_v |p_{wv} - q_{wv}|$$

$$\leq \epsilon_{k-1} \quad \text{(using inductive hypothesis and } \sum_w p_{uw} = 1)$$

Claim 2: $\| f_{u,2^k}(\mathcal{D}_2) - f_{u,2^k}(\mathcal{D}_3) \| \leq \epsilon_{k-1}$.

The proof is similar to the previous case.

Claim 3: $\| f_{u,2^k}(\mathcal{D}_3) - f_{u,2^k}(\mathcal{D}_4) \| \leq 2\epsilon$.
We use the Recycling Lemma. Let $g_u : \{0,1\}^{(k-1)r} \to [1, W]$ be defined as $g_u(a) = f_{u,2^{k-1}}(G_{k-1}(a))$. (To put it in words, apply the Nisan generator to the seed a and use the result as a random string for the machine, using u as the start node. Output the node you reach after 2^{k-1} steps.) Let $X, Y \in U_{(k-1)r}$ and $z \in U_r$. According to the Recycling Lemma,

$$g_u(X) \circ Y \approx_\epsilon g_u(X) \circ \mathsf{Ext}_{k-1}(X, z),$$

and then part 3 of Lemma 16.39 implies that the equivalence continues to hold if we apply a (deterministic) function to the second string on both sides. Thus

$$g_u(X) \circ g_w(Y) \approx_\epsilon g_u(X) \circ g_w(\mathsf{Ext}_{k-1}(X, z))$$

for all nodes w that are 2^{k-1} levels after u. The left distribution corresponds to $f_{u,2^k}(\mathcal{D}_3)$ (by which we mean that $\Pr[f_{u,2^k}(\mathcal{D}_3) = v] = \sum_w \Pr[g_u(X) = w \wedge g_w(Y) = v]$) and the right one to $f_{u,2^k}(\mathcal{D}_4)$ and the proof is completed. ∎

Chapter notes and history

The results of this section have not been presented in chronological order and some important intermediate results have been omitted. Yao [Yao82] first pointed out that cryptographic pseudorandom generators can be used to derandomize **BPP**. A short paper of Sipser [Sip88] initiated the study of "hardness versus randomness," and pointed out the usefulness of a certain family of highly expanding graphs that are now called *dispersers* (they are reminiscent of extractors). This research area received its name as well as a thorough and brilliant development in a paper of Nisan and Wigderson [NW94]. MISSING DISCUSSION OF FOLLOWUP WORKS TO NW94

Weak random sources were first considered in the 1950s by von Neumann [von61]. The second volume of Knuth's seminal work studies real-life pseudorandom generators and their limitations. The study of weak random sources as defined here started with Blum [Blu84]. Progressively weaker models were then defined, culminating in the "correct" definition of an (N, k) source in Zuckerman [Zuc90]. Zuckerman also observed that this definition generalizes all models that had been studied to date. (See [SZ99b] for an account of various models considered by previous researchers.) He also gave the first simulation of probabilistic algorithms with such sources assuming $k = \Omega(N)$. A succession of papers has improved this result; for some references, see the paper of Lu, Reingold, Vadhan, and Wigderson [LRVW03], the current champion in this area (though very likely dethroned by the time this book appears).

The earliest work on extractors —in the guise of *leftover hash lemma* of Impagliazzo, Levin, and Luby [ILL89] mentioned in Section 16.6.3— took place in context of cryptography, specifically, cryptographically secure pseudorandom generators. Nisan [Nis92] then showed that hashing could be used to define provably good pseudorandom generators for logspace.

The notion of an extractor was first formalized by Nisan and Zuckerman [NZ96]. Trevisan [Tre01] pointed out that any "black-box" construction of a pseudorandom generator gives an extractor, and in particular used the Nisan-Wigderson generator to construct extractors as described in the chapter. His methodology has been sharpened in many other papers (e.g.,see [LRVW03]).

Our discussion of derandomization has omitted many important papers that successively improved Nisan-Wigderson and culminated in the result of Impagliazzo and Wigderson [IW01]that either **NEXP = BPP** (randomness is truly powerful!) or **BPP** has an a subexponential "simulation." [7] Such results raised hopes that we were getting close to at least a partial derandomization of **BPP**, but these hopes were dashed by the Impagliazzo-Kabanets [KI03] result of Section 16.3.

---

[7]The "simulation" is in quotes because it could fail on some instances, but finding such instances itself requires exponential computational power, which nature presumably does not have.

Trevisan's insight about using pseudorandom generators to construct extractors has been greatly extended. It is now understood that three combinatorial objects studied in three different fields are very similar: pseudorandom generators (cryptography and derandomization), extractors (weak random sources) and list-decodable error-correcting codes (coding theory and information theory). Constructions of any one of these objects often gives constructions of the other two. For a survey, see Vadhan's lecture notes [?].

STILL A LOT MISSING

Expanders were well-studied for a variety of reasons in the 1970s but their application to pseudorandomness was first described by Ajtai, Komlos, and Szemeredi [AKS87]. Then Cohen-Wigderson [CW89] and Impagliazzo-Zuckerman (1989) showed how to use them to "recycle" random bits as described in Section 7.B.3. The upcoming book by Hoory, Linial and Wigderson (draft available from their web pages) provides an excellent introduction to expander graphs and their applications.

The explicit construction of expanders is due to Reingold, Vadhan and Wigderson [RVW00], although we chose to present it using the replacement product as opposed to the closely related zig-zag product used there. The deterministic logspace algorithm for undirected connectivity is due to Reingold [?].

Exercises

§1 Verify Corollary 16.6.

§2 Show that there exists a number $\epsilon > 0$ and a function $G : \{0,1\}^* \to \{0,1\}^*$ that satisfies all of the conditions of a $2^{\epsilon n}$-pseudorandom generator per Definition ??, save for the computational efficiency condition.

> **Hint:** show that if for every n, a random function mapping n bits to $2^{n/10}$ bits will have desired properties with high probabilities.

§3 Show by a counting argument (i.e., probabilistic method) that for every large enough n there is a function $f : \{0,1\}^n \to \{0,1\}$, such that $\mathsf{H}_{\mathsf{avg}}(f) \geq 2^{n/10}$.

§4 Prove that if there exists $f \in \mathbf{E}$ and $\epsilon > 0$ such that $\mathsf{H}_{\mathsf{avg}}(f)(n) \geq 2^{\epsilon n}$ for every $n \in \mathbb{N}$, then $\mathbf{MA} = \mathbf{NP}$.

§5 We define an *oracle Boolean circuit* to be a Boolean circuit that have special gates with unbounded fanin that are marked ORACLE. For a Boolean circuit C and language $O \subseteq \{0,1\}^*$, we define by $C^O(x)$ the output of C on x, where the operation of the oracle gates when fed input q is to output 1 iff $q \in O$.

  (a) Prove that if every $f \in \mathbf{E}$ can be computed by a polynomial-size circuits with oracle to SAT, then the polynomial hierarchy collapses.

  (b) For a function $f : \{0,1\}^* \to \{0,1\}$ and $O \subseteq \{0,1\}^*$, define $\mathsf{H}_{\mathsf{avg}}{}^O(f)$ to be the function that maps every $n \in \mathbb{N}$ to the largest S such that $\Pr_{x \in_R \{0,1\}^n}[C^O(x) = f(x)] \leq 1/2 + 1/S$.

§6 Prove Lemma 16.39.

§7 Prove that for every function $\mathsf{Ext} : \{0,1\}^n \to \{0,1\}^m$ and there exists an $(n, n-1)$-source X and a bit $b \in \{0,1\}$ such that $\Pr[\mathsf{Ext}(X)_1 = b] = 1$ (where $\mathsf{Ext}(X)_1$ denotes the first bit of $\mathsf{Ext}(X)$). Prove that this implies that $\delta(\mathsf{Ext}(X), U_m) \geq 1/2$.

§8 Show that there is a constant $c > 0$ such that if an algorithm runs in time T and requires m random bits, and $m > k + c \log T$, then it is not possible in general to simulate it in a blackbox fashion using an (N, k) source and $O(\log n)$ truly random bits.

Hint: For each source show that there is a randomized algorithm —it need not be efficient, since it is being used as a "black box"— for which the simulation fails.

§9 A *flat* (N, k) source is a (N, k) source where for every $x \in \{0,1\}^N$ p_x is either 0 or exactly 2^{-k}.

Show that a source X is an (N, k)-source iff it is a distribution on flat sources. In other words, there is a set of flat (N, k)-sources $X_1, X_2, \ldots$ and a distribution $\mathcal{D}$ on them such that drawing a sample of X corresponds to picking one of the X_i's according to $\mathcal{D}$, and then drawing a sample from X_i.

Hint: You need to view a distribution as a point in a 2^N-dimensional space, and show that X is in the convex hull of the points that represent all possible flat sources.

§10 Use Nisan's generator to give an algorithm that produces universal traversal sequences for n-node graphs (see Definition **??**) in $n^{O(\log n)}$-time and $O(\log^2 n)$ space.

§11 Suppose boolean function f is (S, ϵ)-hard and let D be the distribution on m-bit strings defined by picking inputs $x_1, x_2, \ldots, x_m$ uniformly at random and outputting $f(x_1)f(x_2)\cdots f(x_m)$. Show that the statistical distance between D and the uniform distribution is at most ϵm.

§12 Prove Lemma 16.42.

§13 (Klivans and van Melkebeek 1999) Suppose the conclusion of Lemma **??** is true. Then show that $\mathbf{MA} \subseteq \mathbf{i.o.} - [\mathbf{NTIME}(2^n)/n]$.

(Slightly harder) Show that if $\mathbf{NEXP} \neq \mathbf{EXP}$ then $\mathbf{AM} \subseteq \mathbf{i.o.} - [\mathbf{NTIME}(2^n)/n]$.

§14 Let A be an $n \times n$ matrix with eigenvectors $\mathbf{u}^1, \ldots, \mathbf{u}^n$ and corresponding values $\lambda_1, \ldots, \lambda_n$. Let B be an $m \times m$ matrix with eigenvectors $\mathbf{v}^1, \ldots, \mathbf{v}^m$ and corresponding values $\alpha_1, \ldots, \alpha_m$. Prove that the matrix $A \otimes B$ has eigenvectors $\mathbf{u}^i \otimes \mathbf{v}^j$ and corresponding values $\lambda_i \cdot \alpha_j$.

§15 Prove that for every two graphs G, G', $\lambda(G \otimes G') \leq \lambda(G) + \lambda(G')$ without using the fact that every symmetric matrix is diagonalizable.

Hint: Use Lemma 7.40.

§16 Let G be an n-vertex D-degree graph with ρ combinatorial edge expansion for some $\rho > 0$. (That is, for every a subset S of G's vertices of size at most $n/2$, the number of edges between S and its complement is at least $\rho d|S|$.) Let G' be a D-vertex d-degree graph with ρ' combinatorial edge expansion for some $\rho' > 0$. Prove that $G \circledR G'$ has at least $\rho^2 \rho'/1000$ edge expansion.

the latter use the expansion of G'.

up less than that. For the former use the expansion of G, while for

more than $1 - \rho/10$ portion of their clusters and those that take take

the individual clusters. Treat differently the subsets that take up

Hint: Every subset of $G \circledR G'$ can be thought of as n subsets of

Acknowledgements

Chapter 17

Hardness Amplification and Error Correcting Codes

We pointed out in earlier chapters (e.g., Chapter **??** the distinction between worst-case hardness and average-case hardness. For example, the problem of finding the smallest factor of every given integer seems difficult on worst-case instances, and yet is trivial for at least half the integers – namely, the even ones. We also saw that functions that are average-case hard have many uses, notably in cryptography and derandomization.

In this chapter we study techniques for *amplifying* hardness. First, we see Yao's XOR Lemma, which transforms a "mildly hard" function (i.e., one that is hard to compute on a small fraction of the instances) to a function that is *extremely hard,* for which the best algorithm is as bad as the algorithm that just randomly guesses the answer. We mentioned Yao's result in the chapter on cryptography as a means to transform weak one-way functions into strong one-way functions. The second result in this chapter is a technique to use *error-correcting codes* to transform *worst-case hard* functions into average-case hard functions. This transformation unfortunately makes the running time exponential, and is thus useful only in derandomization, and not in cryptography.

In addition to their applications in complexity theory, the ideas covered here have had other uses, including new constructions of error-correcting codes and new algorithms in machine learning.

17.1 Hardness and Hardness Amplification.

We now define a slightly more refined notion of hardness, that generalizes both the notions of worst-case and average-case hardness given in Definition 16.7:

DEFINITION 17.1 (HARDNESS)
Let $f : \{0,1\}^* \to \{0,1\}$ and $\rho : \mathbb{N} \to [0,1]$. We define $\mathsf{H}^{\rho}_{\mathrm{avg}}(f)$ to be the function from $\mathbb{N}$ to $\mathbb{N}$ that maps every number n to the largest number S such that $\Pr_{x \in_R \{0,1\}^n}[C(x) = f(x)] < \rho(n)$ for every Boolean circuit C on n inputs with size at most S.

Note that, in the notation of Definition 16.7, $\mathsf{H}_{\mathsf{wrs}}(f) = \mathsf{H}^1_{\mathsf{avg}}(f)$ and $\mathsf{H}_{\mathsf{avg}}(f)(n) = \max\left\{S : \mathsf{H}^{1/2+1/S}_{\mathsf{avg}}(f)(n) \geq S\right\}$. In this chapter we show the following results for every two functions $S, S' : \mathbb{N} \to \mathbb{N}$:

Worst-case to mild hardness. If there is a function $f \in \mathbf{E} = \mathbf{DTIME}(2^{O(n)})$ such that $\mathsf{H}_{\mathsf{wrs}}(f)(n) = \mathsf{H}^1_{\mathsf{avg}}(f)(n) \geq S(n)$ then there is a function $f' \in \mathbf{E}$ such that $\mathsf{H}^{0.99}_{\mathsf{avg}}(f)(n) \geq S(\epsilon n)^\epsilon$ for some constant $\epsilon > 0$ and every sufficiently large n.

Mild to strong hardness. If $f' \in \mathbf{E}$ satisfies $\mathsf{H}^{0.99}_{\mathsf{avg}}(f')(n) \geq S'(n)$ then there is $f'' \in \mathbf{E}$ and $\epsilon > 0$ such that $\mathsf{H}_{\mathsf{avg}}(f'')(n) \geq S'(n^\epsilon)^\epsilon$.

Combining these two results with Theorem 16.10, this implies that if there exists a function $f \in \mathbf{E}$ with $\mathsf{H}_{\mathsf{wrs}}(f)(n) \geq S(n)$ then there exists an $S(\ell^\epsilon)^\epsilon$-pseudorandom generator for some $\epsilon > 0$, and hence:

Corollary 1 If there exists $f \in \mathbf{E}$ and $\epsilon > 0$ such that $\mathsf{H}_{\mathsf{wrs}}(f) \geq 2^{n^\epsilon}$ then $\mathbf{BPP} \subseteq \mathbf{QuasiP} = \cup_c \mathbf{DTIME}(2^{\log n^c})$.

Corollary 2 If there exists $f \in \mathbf{E}$ such that $\mathsf{H}_{\mathsf{wrs}}(f) \geq n^{\omega(1)}$ then $\mathbf{BPP} \subseteq \mathbf{SUBEXP} = \cap_\epsilon \mathbf{DTIME}(2^{n^\epsilon})$.

To get to $\mathbf{BPP} = \mathbf{P}$, we need a stronger transformation. We do this by showing how to transform in one fell swoop, a function $f \in \mathbf{E}$ with $\mathsf{H}_{\mathsf{wrs}}(f) \geq S(n)$ into a function $f' \in \mathbf{E}$ with $\mathsf{H}_{\mathsf{avg}}(f) \geq S(\epsilon n)^\epsilon$ for some $\epsilon > 0$. Combined with Theorem 16.10, this implies that $\mathbf{BPP} = \mathbf{P}$ if there exists $f \in \mathbf{E}$ with $\mathsf{H}_{\mathsf{wrs}}(f) \geq 2^{\Omega(n)}$.

17.2   Mild to strong hardness: Yao's XOR Lemma.

We start with the second result described above: transforming a function that has "mild" average-case hardness to a function that has strong average-case hardness. The transformation is actually quite simple and natural, but its analysis is somewhat involved (yet, in our opinion, beautiful).

THEOREM 17.2 (YAO'S XOR LEMMA)
For every $f : \{0,1\}^n \to \{0,1\}$ and $k \in \mathbb{N}$, define $f^{\oplus k} : \{0,1\}^{nk} \to \{0,1\}$ as follows:
$f^{\oplus k}(x_1, \ldots, x_k) = \sum_{i=1}^k f(x_i) \pmod 2$.
For every $\delta > 0, S$ and $\epsilon > 2(1 - \delta/2)^k$, if $\mathsf{H}^{1-\delta}_{\mathsf{avg}}(f) \geq S$ then

$$\mathsf{H}^{1/2+\epsilon}_{\mathsf{avg}}(f^{\oplus k}) \geq \frac{\epsilon^2}{100 \log(1/\delta\epsilon)} S$$

The intuition behind Theorem 17.2 derives from the following fact. Suppose we have a biased coin that, whenever it is tossed, comes up heads with probability $1 - \delta$ and tails with probability δ. If δ is small, each coin toss is fairly predictable. But suppose we now toss it k times and define a composite coin toss that is "heads" iff the coin came up heads an odd number of times. Then the probability of "heads" in this composite coin toss is at most $1/2 + (1 - 2\delta)^k$ (see Exercise 1), which tends to $1/2$ as k increases. Thus the parity of coin tosses becomes quite unpredictable. The

analogy to our case is that intuitively, for each i, a circuit of size S has chance at most $1 - \delta$ of "knowing" $f(x_i)$ if x_i is random. Thus from its perspective, whether or not it will be able to know $f(x_i)$ is like a biased coin toss. Hence its chance of guessing the parity of the k bits should be roughly like $1/2 + (1 - 2\delta)^k$.

We transform this intuition into a proof via an elegant result of Impagliazzo, that provides some fascinating insight on mildly hard functions.

DEFINITION 17.3 (δ-DENSITY DISTRIBUTION)
For $\delta < 1$ a δ-*density* distribution H over $\{0,1\}^n$ is one such that for every $x \in \{0,1\}^n$, $\Pr[H = x] \leq \frac{2^{-n}}{\delta}$.

REMARK 17.4
Note that in Chapter 16 we would have called it a distribution with min entropy $n - \log 1/\delta$.

The motivating example for this definition is the distribution that is uniform over some subset of size $\delta 2^n$ and has 0 probability outside this set.

A priori, one can think that a function f that is hard to compute by small circuits with probability $1 - \delta$ could have two possible forms: **(a)** the hardness is sort of "spread" all over the inputs, and it is roughly $1 - \delta$-hard on every significant set of inputs or **(b)** there is a subset H of roughly a δ fraction of the inputs such that on H the function is *extremely hard* (cannot be computed better than $\frac{1}{2} + \epsilon$ for some tiny ϵ) and on the rest of the inputs the function may be even very easy. Such a set may be thought of as lying at the core of the hardness of f and is sometimes called the *hardcore set*. Impagliazzo's Lemma shows that actually *every* hard function has the form **(b)**. (While the Lemma talks about distributions and not sets, one can easily transform it into a result on sets.)

LEMMA 17.5 (IMPAGLIAZZO'S HARDCORE LEMMA)
For every $\delta > 0$, $f : \{0,1\}^n \to \{0,1\}^n$, and $\epsilon > 0$, if $\mathsf{H}_{\mathrm{avg}}^{1-\delta}(f) \geq S$ then there exists a distribution H over $\{0,1\}^n$ of density at least $\delta/2$ such that for every circuit C of size at most $\frac{\epsilon^2 S}{100 \log(1/\delta\epsilon)}$,

$$\Pr_{x \in_R H}[C(x) = f(x)] \leq \frac{1}{2} + \epsilon \,,$$

Proof of Yao's XOR Lemma using Impagliazzo's Hardcore Lemma.

We now use Lemma 17.5 to transform the biased-coins intuition discussed above into a proof of the XOR Lemma. Let $f : \{0,1\}^n \to \{0,1\}$ be a function such that $\mathsf{H}_{\mathrm{avg}}^{1-\delta}(f) \geq S$, let $k \in \mathbb{N}$ and suppose, for the sake of contradiction, that there is a circuit C of size $\frac{\epsilon^2}{100 \log(1/\delta\epsilon)} S$ such that

$$\Pr_{(x_1,\ldots,x_k) \in_R U_n^k}\left[C(x_1,\ldots,x_k) = \sum_{i=1}^{k} f(x_i) \pmod 2\right] \geq \frac{1}{2} + \epsilon \,, \tag{1}$$

where $\epsilon > 2(1 - \delta/2)^k$.

Let H be the hardcore distribution of density at least $\delta' = \delta/2$ that is obtained from Lemma 17.5, on which every circuit C' fails to compute f with probability better than $1/2 + \epsilon/2$. Define a distribution G over $\{0,1\}^n$ as follows: for every $x \in \{0,1\}^n$, $\Pr[G = x] = (1 - \delta' \Pr[H = x])/(1 - \delta')$.

Note that G is indeed a well-defined distribution, as H has density at least δ'. Also note that if H was the uniform distribution over some subset of $\{0,1\}^n$ of size $\delta' 2^n$, then G will be the uniform distribution over the complement of this subset.

We can think of the process of picking a uniform element in $\{0,1\}^n$ as follows: first toss a δ'-biased coin that comes up "heads" with probability δ. Then, if it came up "heads" choose a random element out of H, and with probability $1 - \delta'$, and otherwise choose a random element out of G. We shorthand this and write

$$U_n = (1 - \delta')G + \delta'H \,. \tag{2}$$

If we consider the distribution $(U_n)^2$ of picking *two* random strings, then by (2) it can be written as $(1 - \delta')^2 G^2 + (1 - \delta')\delta' GH + \delta'(1 - \delta')HG + \delta'^2 H^2$. Similarly, for every k

$$(U_n)^k = (1 - \delta')^k G^k + (1 - \delta')^{k-1}\delta' G^{k-1}H + \cdots + \delta'^k H^k \,. \tag{3}$$

For every distribution $\mathcal{D}$ over $\{0,1\}^{nk}$ let $P_\mathcal{D}$ be the probability of the event of the left-hand side of (1) that $C(x_1, \ldots, x_k) = \sum_{i=1}^k f(x_i) \pmod 2$ where $x_1, \ldots, x_k$ are chosen from $\mathcal{D}$. Then, combining (1) and (3),

$$\tfrac{1}{2} + \epsilon \leq P_{(U_n)^k} = (1 - \delta')^k P_{G^k} + (1 - \delta')^{k-1}\delta' P_{G^{k-1}H} + \cdots + \delta'^k P_{H^k} \,.$$

But since $\delta' = \delta/2$ and $\epsilon > 2(1 - \delta/2)^k$ and $P_{G^k} \leq 1$ we get

$$\tfrac{1}{2} + \epsilon/2 \leq \tfrac{1}{2} + \epsilon - (1 - \delta')^k \leq (1 - \delta')^{k-1}\delta' P_{G^{k-1}H} + \cdots + \delta'^k P_{H^k} \,.$$

Notice, the coefficients of all distributions on the right hand side sum up to less than one, so there must exist a distribution $\mathcal{D}$ that has at least one H component such that $P_\mathcal{D} \geq \tfrac{1}{2} + \epsilon/2$. Suppose that $\mathcal{D} = G^{k-1}H$ (all other cases are handled in a similar way). Then, we get that

$$\Pr_{X_1,\ldots,X_{k-1}\in_R G, X_k\in_R H}[C(X_1, \ldots, X_{k-1}, X_k) = \sum_{i=1}^k f(X_i) \pmod 2] \geq \tfrac{1}{2} + \epsilon/2 \,. \tag{4}$$

By the averaging principle, (4) implies that there *exist* $k - 1$ strings $x_1, \ldots, x_{k-1}$ such that if $b = \sum_{i=1}^{k-1} f(x_i) \pmod 2$, then,

$$\Pr_{X_k\in_R H}[C(x_1, \ldots, x_{k-1}, X_k) = b + f(X_k) \pmod 2] \geq \tfrac{1}{2} + \epsilon/2 \,. \tag{5}$$

But by "hardwiring" the values $x_1, \ldots, x_k$ and b into the circuit C, (5) shows a direct contradiction to the fact that H is a hardcore distribution for the function f. ∎

17.3 Proof of Impagliazzo's Lemma

Let f be a function with $\mathsf{H}^{1-\delta}_{\text{avg}}(f) \geq S$. To Prove Lemma 17.5 we need to show a distribution H over $\{0,1\}^n$ (with no element of weight more than $2 \cdot 2^{-n}/\delta$) on which *every* circuit C of size S' cannot compute f with probability better than $\tfrac{1}{2} + \epsilon$ (where S', ϵ are as in the Lemma's statement).

Let's think of this task as a game between two players named *Russell* and *Noam*. Russell first sends to Noam some distribution H over $\{0,1\}^n$ with density at least δ. Then Noam sends to Russell some circuit C of size at most S'. Russell then pays to Noam $\mathsf{E}_{x \in_R H}[\mathsf{Right}_C(x)]$ dollars, where $\mathsf{Right}_C(x)$ is equal to 1 if $C(x) = f(x)$ and equal to 0 otherwise. What we need to prove is that there is distribution that Russell can choose, such that no matter what circuit Noam sends, Russell will not have to pay him more than $1/2 + \epsilon$ dollars.

An initial observation is that Russell could have easily ensured this if he was allowed to play second instead of first. Indeed, under our assumptions, for *every* circuit C of size S (and so, in particular also for circuits of size S' which is smaller than S), there exists a set S_C of at least $\delta 2^n \geq (\delta/2)2^n$ inputs such that $C(x) \neq f(x)$ for every $x \in S_C$. Thus, if Noam had to send his circuit C, then Russell could have chosen H to be the uniform distribution over S_C. Thus H would have density at least $\delta/2$ and $\mathsf{E}_{x \in_R H}[\mathsf{Right}_C(x)] = 0$, meaning that Russell wouldn't have to pay Noam a single cent.

Now this game is a *zero sum game*, since whatever Noam gains Russell loses and vice versa, tempting us to invoke von-Neumann's famous *Min-Max Theorem* (see Note 17.7) that says that in a zero-sum game it does not matter who plays first *as long as we allow randomized strategies.*[1] What does it mean to allow randomized strategies in our context? It means that Noam can send a *distribution* $\mathcal{C}$ over circuits instead of a single circuit, and the amount Russell will pay is $\mathsf{E}_{C \in_R \mathcal{C}}\mathsf{E}_{x \in_R H}[\mathsf{Right}_C(x)]$. (It also means that Russell is allowed to send a distribution over $\delta/2$-density distributions, but this is equivalent to sending a single $\delta/2$-density distribution.)

Thus, we only need to show that, when playing second, Russell can still ensure a payment of at most $1/2 + \epsilon$ dollars even when Noam sends a distribution $\mathcal{C}$ of S'-sized circuits. For every distribution $\mathcal{C}$, we say that an input $x \in \{0,1\}^n$ is *good for Noam* (*good* for short) with respect to $\mathcal{C}$ if $\mathsf{E}_{C \in_R \mathcal{C}}[\mathsf{Right}_C(x)] \geq 1/2 + \epsilon$. It suffices to show that for every distribution $\mathcal{C}$ over circuits of size at most S', the number of good x's with respect to $\mathcal{C}$ is at most $1 - \delta/2$. (Indeed, this means that for every $\mathcal{C}$, Russell could choose as its distribution H the uniform distribution over the bad inputs with respect to $\mathcal{C}$.)

Suppose otherwise, that there is at least a $1 - \delta/2$ fraction of inputs that are good for $\mathcal{C}$. We will use this to come up with an S-sized circuit C that computes f on at least a $1 - \delta$ fraction of the inputs in $\{0,1\}^n$, contradicting the assumption that $\mathsf{H}_{\mathrm{avg}}^{1-\delta}(f) \geq S$. Let $t = 10\log(1/\delta\epsilon)/\epsilon^2$, choose $C_1, \ldots, C_t$ at random from $\mathcal{C}$ and let $C = \mathrm{maj}\{C_1, \ldots, C_t\}$ be the circuit of size $tS' < S$ circuit that on input x outputs the majority value of $\{C_1(x), \ldots, C_t(x)\}$. If x is good for $\mathcal{C}$, then by the Chernoff bound we have that $C(x) = f(x)$ with probability at least $1 - \delta/2$ over the choice of $C_1, \ldots, C_t$. Since we assume at least $1 - \delta/2$ of the inputs are good for $\mathcal{C}$, we get that

$$\mathsf{E}_{x \in_R \{0,1\}^n}\mathsf{E}_{C_1 \in_R \mathcal{C}, \ldots, C_t \in_R \mathcal{C}}[\mathsf{Right}_{\mathrm{maj}\{C_1,\ldots,C_t\}}(x)] \geq (1-\tfrac{\delta}{2})(1-\tfrac{\delta}{2}) \geq 1-\delta. \qquad (6)$$

But by linearity of expectation, we can switch the order of expectations in (6) obtaining that

$$\mathsf{E}_{C_1 \in_R \mathcal{C}, \ldots, C_t \in_R \mathcal{C}}\mathsf{E}_{x \in_R \{0,1\}^n}[\mathsf{Right}_{\mathrm{maj}\{C_1,\ldots,C_t\}}(x)] \geq 1-\delta,$$

---

[1] The careful reader might note that another requirement is that the set of possible moves by each player is *finite*, which does not seem to hold in our case as Russell can send any one of the infinitely many $\delta/2$-density distributions. However, by either requiring that the probabilities of the distribution are multiples of $\frac{\epsilon}{100 \cdot 2^n}$ (which won't make any significant difference in the game's outcome), or using the fact that each such distribution is a convex sum of uniform distributions over sets of size at least $(\delta/2)2^n$ (see Exercise 9 of Chapter 16), we can make this game finite.

which in particular implies that *there exists* a circuit C of size at most S such that $\mathsf{E}_{x \in_R U_n}[\mathsf{Right}_C(x)] \geq 1 - \delta$, or in other words, C computes f on at least a $1 - \delta$ fraction of the inputs. $\blacksquare$

REMARK 17.6
Taken in the contrapositive, Lemma 17.5 implies that if for every significant chunk of the inputs there is some circuit that computes f with on this chunk with some advantage over $1/2$, then there is a single circuit that computes f with good probability over all inputs. In machine learning such a result (transforming a way to weakly predict some function into a way to strongly predict it) is called *Boosting* of learning methods. Although the proof we presented here is non-constructive, Impagliazzo's original proof was constructive, and was used to obtain a boosting algorithm yielding some new results in machine learning, see [**?**].

DRAFT

NOTE 17.7 (THE MIN-MAX THEOREM)

A *zero sum game* is, as the name implies, a game between two parties in which whatever one party loses is won by the other party. It is modeled by an $m \times n$ matrix $A = (a_{i,j})$ of real numbers. The game consists of only a single move. One party, called the *minimizer* or *column player*, chooses an index $j \in [n]$ while the other party, called the *maximizer* or *row player*, chooses an index $i \in [m]$. The *outcome* is that the column player has to pay $a_{i,j}$ units of money to the row player (if $a_{i,j}$ is negative then actually the row player has to pay). Clearly, the *order* in which players make their moves is important. Surprisingly, if we allow the players randomized strategies, then the order of play becomes unimportant.

The game with *randomized* (also known as *mixed*) strategies is as follows. The column player chooses a *distribution* over the columns; that is, a vector $\mathbf{p} \in [0,1]^n$ with $\sum_{i=1}^{n} p_i = 1$. Similarly, the row player chooses a distribution $\mathbf{q}$ over the rows. The amount paid is the expectation of $a_{i,j}$ for j chosen from $\mathbf{p}$ and i chosen from $\mathbf{q}$. If we think of $\mathbf{p}$ as a column vector and $\mathbf{q}$ as a row vector then this is equal to $\mathbf{q}A\mathbf{p}$. The min-max theorem says:

$$\min_{\substack{\mathbf{p}\in[0,1]^n \\ \Sigma_i p_i=1}} \max_{\substack{\mathbf{q}\in[0,1]^m \\ \Sigma_i q_i=1}} \mathbf{q}A\mathbf{p} = \max_{\substack{\mathbf{q}\in[0,1]^m \\ \Sigma_i q_i=1}} \min_{\substack{\mathbf{p}\in[0,1]^n \\ \Sigma_i p_i=1}} \mathbf{q}A\mathbf{p} \qquad (7)$$

The min-max theorem can be proven using the following result, known as Farkas' Lemma:[2] if C and D are disjoint convex subsets of $\mathbb{R}^m$, then there is an $m-1$ dimensional hyperplane that separates them. That is, there is a vector $\mathbf{z}$ and a number a such that for every $\mathbf{x} \in C$, $\langle \mathbf{x}, \mathbf{z} \rangle = \sum_i x_i z_i \leq a$ and for every $\mathbf{y} \in D$, $\langle \mathbf{y}, \mathbf{z} \rangle \geq a$. (A subset $C \subseteq \mathbb{R}^m$ is *convex* if whenever it contains a pair of points $\mathbf{x}, \mathbf{y}$, it contains the line segment $\{\alpha\mathbf{x} + (1-\alpha)\mathbf{y} : 0 \leq \alpha \leq 1\}$ that lies between them.) We ask you to prove Farkas' Lemma in Exercise 2 but here is a "proof by picture" for the two dimensional case:
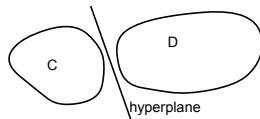
Farkas' Lemma implies the min-max theorem by noting that $\max_{\mathbf{q}} \min_{\mathbf{p}} \mathbf{q}A\mathbf{p} \geq c$ if and only if the convex set $D = \{A\mathbf{p} : \mathbf{p} \in [0,1]^n \sum_i p_i = 1\}$ does not intersect with the convex set $C = \{\mathbf{x} \in \mathbb{R}^m : \forall_{i \in [m]} x_i < c\}$ and using the Lemma to show that this implies the existence of a probability vector $\mathbf{q}$ such that $\langle \mathbf{q}, \mathbf{y} \rangle \geq c$ for every $\mathbf{y} \in D$ (see Exercise 3). The Min-Max Theorem is equivalent to another well-known result called *linear programming duality*, that can also be proved using Farkas' Lemma (see Exercise 4).

17.4 Error correcting codes: the intuitive connection to hardness amplification

Now we construct average-case hard functions using functions that are only worst-case hard. To do so, we desire a way to transform any function f to another function g such that if there is a small circuit that computes g approximately (i.e., correctly outputs $g(x)$ for many x) then there is a small circuit that computes f at all points. Taking the contrapositive, we can conclude that if there is no small circuit that computes f then there is no small circuit that computes g approximately.

Let us reason abstractly about how to go about the above task.

View a function $f : \{0,1\}^n \to \{0,1\}$ as its truth table, namely, as a string of length 2^n, and view any circuit C for computing this function as a device that, given any index $x \in [2^n]$, gives the x'th bit in this string. If the circuit only computes g on "average" then this device may be thought of as only partially correct; it gives the right bit only for many indices x's, but not all. Thus we need to show how to turn a partially correct string for g into a completely correct string for f. This is of course reminiscent of *error correcting codes* (ECC), but with a distinct twist involving computational efficiency of decoding, which we will call *local decoding*.

The classical theory of ECC's (invented by Shannon in 1949) concerns the following problem. We want to record some data $x \in \{0,1\}^n$ on a compact disk to retrieve at a later date, but that compact disk might scratched and say 10% of its contents might be corrupted. The idea behind error correcting codes is to encode x using some *redundancy* so that such corruptions do not prevent us from recovering x.

The naive idea of redundancy is to introduce repetitions but that does not work. For example suppose we repeat each bit three times, in other words encode x as the string $y = x_1x_1x_1x_2x_2x_2\ldots x_nx_nx_n$. But now if the first three coordinates of y are corrupted then we cannot recover x_1, even if all other coordinates of y are intact. (Note that the first three coordinates take only a $1/n \ll 10\%$ fraction of the entire string y.) Clearly, we need a smarter way.

DEFINITION 17.8 (ERROR CORRECTING CODES)
For $x, y \in \{0,1\}^m$, the *fractional Hamming distance* of x and y, denoted $\Delta(x, y)$, is equal to $\frac{1}{m} |\{i : x_i \neq y_i\}|$.
For every $\delta \in [0,1]$, a function $E : \{0,1\}^n \to \{0,1\}^m$ is an *error correcting code (ECC) with distance* δ, if for every $x \neq y \in \{0,1\}^n$, $\Delta(E(x), E(y)) \geq \delta$. We call the set $Im(E) = \{E(x) : x \in \{0,1\}^n\}$ the set of *codewords* of E.

Suppose $E : \{0,1\}^n \to \{0,1\}^m$ is an ECC of distance $\delta > 0.2$. Then the encoding $x \to E(x)$ suffices for the CD storage problem (momentarily ignoring issues of computational efficiency). Indeed, if y is obtained by corrupting $0.1m$ coordinates of $E(x)$, then $\Delta(y, E(x)) < \delta/2$ and by the triangle inequality $\Delta(y, E(x')) > \delta/2$ for every $x' \neq x$. Thus, x is the unique string that satisfies

---

[2]Many texts use the name Farkas' Lemma only to denote a special case of the result stated in Note 17.7. Namely the result that there is a separating hyperplane between any disjoint sets C, D such that C is a single point and D is a set of the form $\{A\mathbf{x} : \forall_i x_i > 0\}$ for some matrix A.
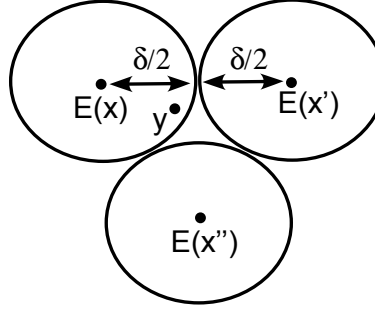
Figure 17.1: In a δ-distance error correcting code, $\Delta(E(x), E(x')) \geq \delta$ for every $x \neq x'$. We can recover x from every string y satisfying $\Delta(y, E(x)) < \delta/2$ since the $\delta/2$-radius ball around every codeword $z = E(x)$ does not contain any other codeword.

$\Delta(y, E(x)) < \delta/2$. (See Figure 17.1.)

Of course, we still need to show that error correcting codes with minimum distance 0.2 actually exist. The following lemma shows this. It introduces $H(\delta)$, the so-called *entropy function*, which lies strictly between 0 and 1 when $\delta \in (0, 1)$.

LEMMA 17.9
For every $\delta < 1/2$ and sufficiently large n, there exists a function $E : \{0, 1\}^n \rightarrow \{0, 1\}^{2n/(1-H(\delta))}$ that is an error correcting code with distance δ, where $H(\delta) = \delta \log(1/\delta) + (1 - \delta) \log(1/(1 - \delta))$.

PROOF: We simply choose the function $E : \{0, 1\}^n \rightarrow \{0, 1\}^m$ at random for $m = 2n/(1 - H(\delta)n$. That is, we choose 2^n random strings $y_1, y_2, \ldots, y_{2^n}$ and E will map the input $x \in \{0, 1\}^n$ (which we can identify with a number in $[2^n]$) to the string y_x.

It suffices to show that the probability that for some $i < j$ with $i, j \in [2^n]$, $\Delta(y_i, y_j) < \delta$ is less than 1. But for every string y_i, the number of strings that are of distance at most δ to it is $\binom{m}{\lceil \delta m \rceil}$ which at most $0.99 \cdot 2^{H(\delta)m}$ for m sufficiently large (see Appendix A) and so for every $j > i$, the probability that y_j falls in this ball is bounded by $0.99 \cdot 2^{H(\delta)m}/2^m$. Since there are at most 2^{2n} such pairs i, j, we only need to show that

$$0.99 \cdot 2^{2n} \frac{2^{H(\delta)m}}{2^m} < 1 .$$

which is indeed the case for our choice of m. ∎

REMARK 17.10
By a slightly more clever argument, we can get rid of the constant 2 above, and show that there exists such a code $E : \{0, 1\}^n \rightarrow \{0, 1\}^{n/(1-H(\delta))}$ (see Exercise 6). We do not know whether this is the smallest value of m possible.

Why half? Lemma 17.9 only provides codes of distance δ for $\delta < 1/2$ and you might wonder whether this is inherent or can we have codes of even greater distance. It turns out we can have codes of distance $1/2$ but only if we allow m to be exponentially larger than n (i.e., $m \geq 2^{n/2}$). For

every $\delta > 1/2$, if n is sufficiently large then there is no ECC $E : \{0,1\}^n \to \{0,1\}^m$ that has distance δ, no matter how large m is. Both these bounds are explored in Exercise 7.

The mere existence of an error correcting code is not sufficient for most applications: we need to actually be able to compute them. For this we need to show an *explicit function* $E : \{0,1\}^n \to \{0,1\}^m$ that is an ECC satisfying the following properties:

Efficient encoding There is a polynomial time algorithm to compute $E(x)$ from x.

Efficient decoding There is a polynomial time algorithm to compute x from every y such that $\Delta(y, E(x)) < \rho$ for some ρ. (For this to be possible, the number ρ must be less than $\delta/2$, where δ is the distance of E.)

There is a very rich and still ongoing body of work dedicated to this task, of which Section 17.5 describes a few examples.

17.4.1 Local decoding

For use in hardness amplification, we need ECCs with more than just efficient encoding and decoding algorithms: we need *local decoders*, in other words, decoding algorithms whose running time is polylogarithmic. Let us see why.

Recall that we are viewing a function from $\{0,1\}^n$ to $\{0,1\}$ as a string of length 2^n. To amplify its hardness, we take an *ECC* and map function f to its encoding $E(f)$. To *prove* that this works, it suffices to show how to turn any circuit that correctly computes many bits of $E(f)$ into a circuit that correctly computes all bits of f. This is formalized using a *local decoder*, which is a decoding algorithm that can compute any desired bit in the string for f using a small number of random queries in any string y that has high agreement with (in other words, low hamming distance to) $E(f)$. Since we are interested in the circuits of size poly(n)— in other words, *polylogarithmic* in 2^n —this must also be the running time of the local decoder.

DEFINITION 17.12 (LOCAL DECODER)
Let $E : \{0,1\}^n \to \{0,1\}^m$ be an ECC and let ρ and q be some numbers. A *local decoder for E handling ρ errors* is an algorithm L that, given random access to a string y such that $\Delta(y, E(x)) < \rho$ for some (unknown) $x \in \{0,1\}^n$, and an index $j \in \mathbb{N}$, runs for polylog(m) time and outputs x_j with probability at least $2/3$.

REMARK 17.13
The constant $2/3$ is arbitrary and can be replaced with any constant larger than $1/2$, since the probability of getting a correct answer can be amplified by repetition.

Notice, local decoding may be useful in applications of ECC's that have nothing to do with hardness amplification. Even in context of CD storage, it seems nice if we do not to have to read the entire CD just to recover one bit of x.

Using a local decoder, we can turn our intuition above of hardness amplification into a proof.

NOTE 17.11 (HIGH DIMENSIONAL GEOMETRY)
While we are normally used to geometry in two or three dimensions, we can
get some intuition on error correcting codes by considering the geometry of
high dimensional spaces. Perhaps the strongest effect of high dimension is the
following: compare the cube with all sides 1 and the ball of radius $1/4$. In one
dimension, the ratio between their areas is $1/(1/2) = 2$, in two dimensions
it is $1/(\pi 1/4^2) = 16/\pi$, while in three dimensions it is $1/(4/3\pi 1/4^3) = 48/\pi$.
Note that as the number of dimension grows, this ratio grows exponentially
in the number of dimensions. (Similarly for any two radii $r_1 > r_2$ the volume
of the m-dimension ball of radius r_1 is exponentially larger than the volume
of the r_2-radius ball.)

0 1/4   3/4 1   0 1/4   3/4 1   0 1/4   3/4 1
Ball volume=1/2
            B.V. = $\pi(1/4)^2$~3.14/16   B.V. =$4/3\pi(1/4)^3$ ~ 3.14/48
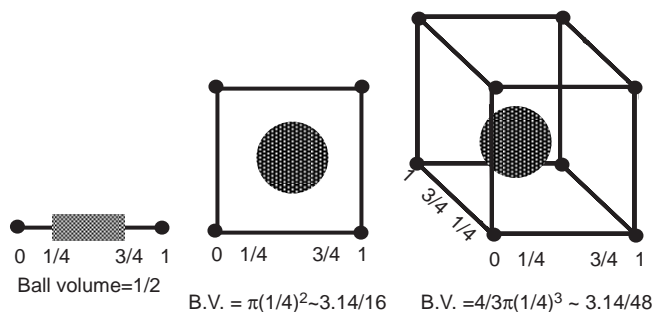
This intuition lies behind the existence of an error correcting code with
distance $1/4$ mapping n bit strings into $m = 5n$ bit strings. We can have $2^{m/5}$
codewords that are all of distance at least $1/4$ from one another because, also
in the Hamming distance, the volume of the radius $1/4$ ball is exponentially
smaller than the volume of the cube $\{0,1\}^n$. Therefore, we can "pack" $2^{m/5}$
such balls within the cube.

DRAFT
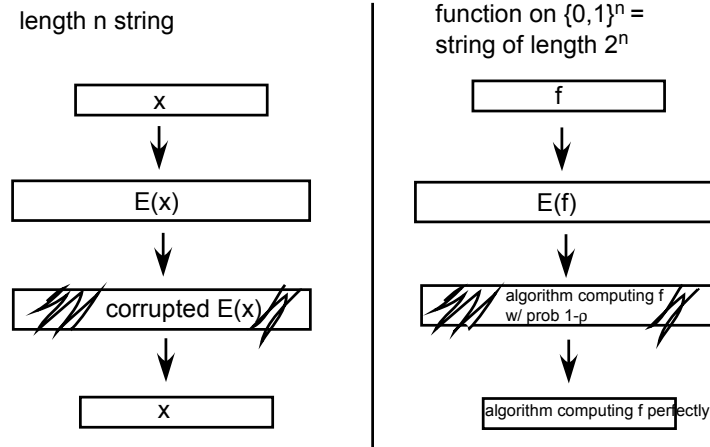
Figure 17.2: An ECC allows to map a string x to $E(x)$ such as x can be reconstructed from a corrupted version of $E(x)$. The idea is to treat a function $f : \{0,1\}^n \to \{0,1\}$ as a string in $\{0,1\}^{2^n}$, encode it using an ECC to a function $\hat{f}$. Intuitively, $\hat{f}$ should be hard on the average case if f was hard on the worst case, since an algorithm to solve $\hat{f}$ with probability $1 - \rho$ could be transformed (using the ECC's decoding algorithm) to an algorithm computing f on every input.

THEOREM 17.14
Suppose that there is an ECC with polynomial-time encoding algorithm and a local decoding algorithm handling ρ errors (where ρ is a constant independent of the input length). Suppose also that there is $f \in \mathbf{E}$ with $\mathsf{H}_{\mathsf{wrs}}(f)(n) \geq S(n)$ for some function $S : \mathbb{N} \to \mathbb{N}$ satisfying $S(n) \geq n$. Then, there exists $\epsilon > 0$ and $g \in \mathbf{E}$ with $\mathsf{H}_{\mathsf{wrs}}(g)(n) \geq S(\epsilon n)^\epsilon$

The proof of Theorem 17.14 follows essentially from the definition, and we will prove it for the case of a particular code later on in Theorem 17.24.

17.5   Constructions of Error Correcting Codes

We now describe some explicit functions that are error correcting codes, building up to the construction of an explicit ECC of constant distance with polynomial-time encoding and decoding. Section 17.6 describes *local decoding* algorithms for some of these codes.

17.5.1   Walsh-Hadamard Code.

For two strings $x, y \in \{0,1\}^n$, define $x \odot y$ to be the number $\sum_{i=1}^n x_i y_i \pmod 2$. The *Walsh-Hadamard code* is the function $\mathsf{WH} : \{0,1\}^n \to \{0,1\}^{2^n}$ that maps a string $x \in \{0,1\}^n$ into the string $z \in \{0,1\}^{2^n}$ where for every $y \in \{0,1\}^n$, the yth coordinate of z is equal to $x \odot y$ (we identify $\{0,1\}^n$ with $[2^n]$ in the obvious way).

CLAIM 17.15
The function WH *is an error correcting code of distance* $1/2$.
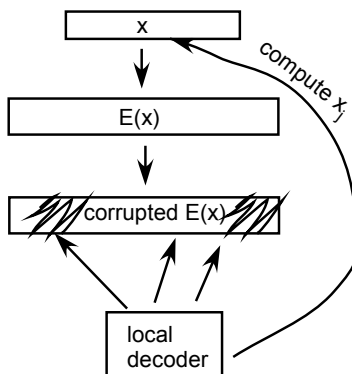
Figure 17.3: A local decoder gets access to a corrupted version of $E(x)$ and an index i and computes from it x_i (with high probability).

PROOF: First, note that WH is a linear function. By this we mean that if we take $x + y$ to be the componentwise addition of x and y modulo 2, then $\mathsf{WH}(x + y) = \mathsf{WH}(x) + \mathsf{WH}(y)$. Now, for every $x \neq y \in \{0,1\}^n$ we have that the number of 1's in the string $\mathsf{WH}(x) + \mathsf{WH}(y) = \mathsf{WH}(x + y)$ is equal to the number of coordinates on which $\mathsf{WH}(x)$ and $\mathsf{WH}(y)$ differ. Thus, it suffices to show that for every $z \neq 0^n$, at least half of the coordinates in $\mathsf{WH}(z)$ are 1. Yet this follows from the random subsum principle (Claim A.5) that says that the probability for $y \in_R \{0,1\}^n$ that $z \odot y = 1$ is exactly $1/2$. ∎

17.5.2 Reed-Solomon Code

The Walsh-Hadamard code has a serious drawback: its output size is exponential in the input size. By Lemma 17.9 we know that we can do much better (at least if we're willing to tolerate a distance slightly smaller than $1/2$). To get towards explicit codes with better output, we need to make a detour to codes with *non-binary alphabet*.

DEFINITION 17.16
For every set Σ and $x, y \in \Sigma^m$, we define $\Delta(x, y) = \frac{1}{m} |\{i : x_i \neq y_i\}|$. We say that $E : \Sigma^n \to \Sigma^m$ is an *error correcting code with distance δ over alphabet Σ* if for every $x \neq y \in \Sigma^n$, $\Delta(E(x), E(y)) \geq \delta$.

Allowing a larger alphabet makes the problem of constructing codes easier. For example, every ECC with distance δ over the binary ($\{0,1\}$) alphabet automatically implies an ECC with the same distance over the alphabet $\{0,1,2,3\}$: just encode strings over $\{0,1,2,3\}$ as strings over $\{0,1\}$ in the obvious way. However, the other direction does not work: if we take an ECC over $\{0,1,2,3\}$ and transform it into a code over $\{0,1\}$ in the natural way, the distance might grow from δ to 2δ (Exercise 8).

The Reed-Solomon code is a construction of an error correcting code that can use as its alphabet any field $\mathbb{F}$:
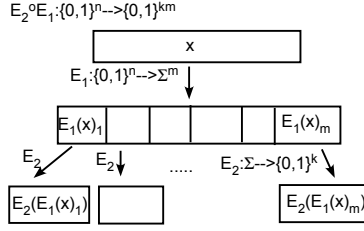
Figure 17.4: If E_1, E_2 are ECC's such that $E_1 : \{0,1\}^n \to \Sigma^m$ and $E_2 : \sigma \to \{0,1\}^k$, then the concatenated code $E : \{0,1\}^n \to \{0,1\}^{nk}$ maps x into the sequence of blocks $E_2(E_1(x)_1), \ldots, E_2(E_1(x)_m)$.

DEFINITION 17.17
Let $\mathbb{F}$ be a field and n, m numbers satisfying $n \le m \le |\mathbb{F}|$. The *Reed-Solomon code* from $\mathbb{F}^n$ to $\mathbb{F}^m$ is the function $\mathsf{RS} : \mathbb{F}^n \to \mathbb{F}^m$ that on input $a_0, \ldots, a_{n-1} \in \mathbb{F}^n$ outputs the string $z_0, \ldots, z_{m-1}$ where

$$z_j = \sum_{i=0}^{n-1} a_i f_j^i$$

and f_j denotes the jth element of $\mathbb{F}$ under some ordering.

LEMMA 17.18
The Reed-Solomon code $\mathsf{RS} : \mathbb{F}^n \to \mathbb{F}^m$ *has distance* $1 - \frac{n}{m}$.

PROOF: As in the case of Walsh-Hadamard code, the function RS is also linear in the sense that $\mathsf{RS}(a + b) = \mathsf{RS}(a) + \mathsf{RS}(b)$ (where addition is taken to be componentwise addition in $\mathbb{F}$). Thus, as before we only need to show that for every $a \ne 0^n$, $\mathsf{RS}(a)$ has at most n coordinates that are zero. But this immediate from the fact that a nonzero $n - 1$ degree polynomial has at most n roots (see Appendix A). ∎

17.5.3   Concatenated codes

The Walsh-Hadamard code has the drawback of exponential-sized output and the Reed-Solomon code has the drawback of a non-binary alphabet. We now show we can combine them both to obtain a code without neither of these drawbacks:

DEFINITION 17.19
If RS is the Reed-Solomon code mapping $\mathbb{F}^n$ to $\mathbb{F}^m$ (for some $n, m, \mathbb{F}$) and WH is the Walsh-Hadamard code mapping $\{0,1\}^{\log |\mathbb{F}|}$ to $\{0,1\}^{2^{\log |\mathbb{F}|}} = \{0,1\}^{|\mathbb{F}|}$, then the code $\mathsf{WH} \circ \mathsf{RS}$ maps $\{0,1\}^{n \log |\mathbb{F}|}$ to $\{0,1\}^{m|\mathbb{F}|}$ in the following way:

1. View RS as a code from $\{0,1\}^{n \log |\mathbb{F}|}$ to $\mathbb{F}^m$ and WH as a code from $\mathbb{F}$ to $\{0,1\}^{|\mathbb{F}|}$ using the canonical representation of elements in $\mathbb{F}$ as strings in $\{0,1\}^{\log |\mathbb{F}|}$.

2. For every input $x \in \{0,1\}^{n \log |\mathbb{F}|}$, $\mathsf{WH} \circ \mathsf{RS}(x)$ is equal to $\mathsf{WH}(\mathsf{RS}(x)_1), \ldots, \mathsf{WH}(\mathsf{RS}(x)_m)$ where $\mathsf{RS}(x)_i$ denotes the ith symbol of $\mathsf{RS}(x)$.

Note that the code $\mathsf{WH} \circ \mathsf{RS}$ can be computed in time polynomial in n, m and $|\mathbb{F}|$. We now analyze its distance:

CLAIM 17.20
Let $\delta_1 = 1 - n/m$ be the distance of RS and $\delta_2 = 1/2$ be the distance of WH. Then $\mathsf{WH} \circ \mathsf{RS}$ is an ECC of distance $\delta_1\delta_2$.

PROOF: Let x, y be two distinct strings in $\{0,1\}^{\log |\mathbb{F}|n}$. If we set $x' = \mathsf{RS}(x')$ and $y' = \mathsf{RS}(y')$ then $\Delta(x', y') \geq \delta_1$. If we let x'' (resp. y'') to be the binary string obtained by applying WH to each of these blocks, then whenever two blocks are distinct, the corresponding encoding will have distance δ_2, and so $\delta(x'', y'') \geq \delta_1\delta_2$. $\blacksquare$

REMARK 17.21
Because for every $k \in \mathbb{N}$, there exists a finite field $|\mathbb{F}|$ of size in $[k, 2k]$ (e.g., take a prime in $[k, 2k]$ or a power of two) we can use this construction to obtain, for every n, a polynomial-time computable ECC $E : \{0,1\}^n \to \{0,1\}^{20n^2}$ of distance 0.4.

Both Definition 17.19 and Lemma 17.20 easily generalize for codes other than Reed-Solomon and Hadamard. Thus, for every two ECC's $E_1 : \{0,1\}^n \to \Sigma^m$ and $E_2 : \Sigma \to \{0,1\}^k$ their concatenation $E_2 \circ E_1$ is a code from $\{0,1\}^n$ to $\{0,1\}^{mk}$ that has distance at least $\delta_1\delta_2$ where δ_1 (resp. δ_2) is the distance of E_1 (resp. E_2), see Figure 17.6. In particular, using a different binary code than WH, it is known how to use concatenation to obtain a polynomial-time computable ECC $E : \{0,1\}^n \to \{0,1\}^m$ of constant distance $\delta > 0$ such that $m = O(n)$.

17.5.4   Reed-Muller Codes.

Both the Walsh-Hadamard and and the Reed-Solomon code are special cases of the following family of codes known as Reed-Muller codes:

DEFINITION 17.22 (REED-MULLER CODES)
Let $\mathbb{F}$ be a finite field, and let ℓ, d be numbers with $d < |\mathbb{F}|$. The *Reed Muller* code with parameters $\mathbb{F}, \ell, d$ is the function $\mathsf{RM} : \mathbb{F}^{\binom{\ell+d}{d}} \to \mathbb{F}^{|\mathbb{F}|^\ell}$ that maps every ℓ-variable polynomial P over $\mathbb{F}$ of total degree d to the values of P on all the inputs in $\mathbb{F}^\ell$.

That is, the input is a polynomial of the form

$$g(x_1, \ldots, x_\ell) = \sum_{i_1+i_2+\ldots+i_\ell \leq \ell} c_{i_1,\ldots,i_\ell} x_1^{i_1} x_2^{i_2} \cdots x_\ell^{i_\ell}$$

specified by the vector of $\binom{\ell+d}{d}$ coefficients $\{c_{i_1,\ldots,i_\ell}\}$ and the output is the sequence $\{g(x_1,\ldots,x_\ell)\}$ for every $x_1, \ldots, x_\ell \in \mathbb{F}$.

Setting $\ell = 1$ one obtains the Reed-Solomon code (for $m = |\mathbb{F}|$), while setting $d = 1$ and $\mathbb{F} = GF(2)$ one obtains a slight variant of the Walsh-Hadamard code. (I.e., the code that maps every $x \in \{0,1\}^n$ into the $2 \cdot 2^n$ long string z such that for every $y \in \{0,1\}^n, a \in \{0,1\}$, $z_{y,a} = x \odot y + a$ (mod 2).)

The Schwartz-Zippel Lemma (Lemma A.25 in Appendix A) shows that the Reed-Muller code is an ECC with distance $1 - d/|\mathbb{F}|$. Note that this implies the previously stated bounds for the Walsh-Hadamard and Reed-Solomon codes.

17.5.5   Decoding Reed-Solomon.

To actually use an error correcting code to store and retrieve information, we need a way to efficiently *decode* a data x from its encoding $E(x)$ even if $E(x)$ has been corrupted in a fraction ρ of its coordinates. We now show this for the Reed-Solomon code, that treats x as a polynomial g, and outputs the values of this polynomial on m inputs.

We know (see Theorem A.24 in Appendix A) that a univariate degree d polynomial can be interpolated from any $d+1$ values. Here we consider a *robust* version of this procedure, whereby we wish to recover the polynomial from m values of which ρm are "faulty" or "noisy".

Let $(a_1, b_1), (a_2, b_2), \ldots, (a_m, b_m)$ be a sequence of (point, value) pairs. We say that a degree d polynomial $g(x)$ *describes* this (a_i, b_i) if $g(a_i) = b_i$.

We are interested in determining if there is a degree d polynomial g that describes $(1 - \rho)m$ of the pairs. If $2\rho m > d$ then this polynomial is unique (exercise). We desire to recover it, in other words, find a degree d polynomial g such that

$$g(a_i) = b_i \qquad \text{for at } (1 - \rho)m \text{ least values of } i. \tag{8}$$

The apparent difficulty is in identifying the noisy points; once those points are identified, we can recover the polynomial.

Randomized interpolation: the case of $\rho < 1/(d+1)$

If ρ is very small, say, $\rho < 1/(2d)$ then we can actually use the standard interpolation technique: just select $d+1$ points at random from the set $\{(a_i, b_i)\}$ and use them to interpolate. By the union bound, with probability at least $1 - \rho(d+1) > 0.4$ all these points will be non-corrupted and so we will recover the correct polynomial. (Because the correct polynomial is unique, we can verify that we have obtained it, and if unsuccessful, try again.)

Berlekamp-Welch Procedure: the case of $\rho < (m-d)/(2m)$

The Berlekamp-Welch procedure works when the error rate ρ is bounded away from $1/2$; specifically, $\rho < (m-d)/(2m)$. For concreteness, assume $m = 4d$ and $\rho = 1/4$.

1. We claim that if the polynomial g exists then there is a degree $2d$ polynomial $c(x)$ and a degree d nonzero polynomial $e(x)$ such that

$$c(a_i) = b_i e(a_i) \qquad \text{for all } i. \tag{9}$$

   The reason is that the desired $e(x)$ can be any nonzero degree d polynomial whose roots are precisely the a_i's for which $g(a_i) \neq b_i$, and then just let $c(x) = g(x)e(x)$. (Note that this is just an *existence* argument; we do not know g yet.))

2. Let $c(x) = \sum_{i \le 2d} c_i x^i$ and $e(x) = \sum_{i \le d} e_i x^i$. The e_i's and c_i's are our unknowns, and these satisfy $4d$ linear equations given in (??), one for each a_i. The number of unknowns is $3d + 2$, and our existence argument in part 1 shows that the system is feasible. Solve it using Gaussian elimination to obtain a candidate c, e.

3. Let c, e are *any* polynomials obtained in part 2. Since they satisfy (9) and $b_i = g(a_i)$ for at least $3d$ values of i, we conclude that

$$c(a_i) = g(a_i)e(a_i) \qquad \text{for at least } 3d \text{ values of } i.$$

Hence $c(x) - g(x)e(x)$ is a degree $2d$ polynomial that has at least $3d$ roots, and hence is identically zero. Hence e divides c and that in fact $c(x) = g(x)e(x)$.

4. Divide c by e to recover g.

17.5.6 Decoding concatenated codes.

Decoding concatenated codes can be achieved through the natural algorithm. Recall that if $E_1 : \{0,1\}^n \to \Sigma^m$ and $E_2 : \Sigma \to \{0,1\}^k$ are two ECC's then $E_2 \circ E_1$ maps every string $x \in \{0,1\}^n$ to the string $E_2(E_1(x)_1) \cdots E_2(E_1(x)_n)$. Suppose that we have a decoder for E_1 (resp. E_2) that can handle ρ_1 (resp. ρ_2) errors. Then, we have a decoder for $E_2 \circ E_1$ that can handle $\rho_2\rho_1$ errors. The decoder, given a string $y \in \{0,1\}^{mk}$ composed of m blocks $y_1, \ldots, y_m \in \{0,1\}^k$, first decodes each block y_i to a symbol z_i in Σ, and then uses the decoder of E_1 to decode $z_1, \ldots, z_m$. The decoder can indeed handle $\rho_1\rho_2$ errors since if $\Delta(y, E_2 \circ E_1(x)) \le \rho_1\rho_2$ then at most ρ_1 of the blocks of y are of distance at least ρ_2 from the corresponding block of $E_2 \circ E_1(x)$.

17.6 Local Decoding of explicit codes.

We now show *local decoder* algorithm (c.f. Definition 17.12) for several explicit codes.

17.6.1 Local decoder for Walsh-Hadamard.

The following is a two-query local decoder for the Walsh-Hadamard code that handles ρ errors for every $\rho < 1/4$. This fraction of errors we handle is best possible, as it can be easily shown that there cannot exist a local (or non-local) decoder for a binary code handling ρ errors for every $\rho \ge 1/4$.

WALSH-HADAMARD LOCAL DECODER for $\rho < 1/4$:

Input: $j \in [n]$, random access to a function $f : \{0,1\}^n \to \{0,1\}$ such that $\Pr_y[g(y) \ne x \odot y] \le \rho$ for some $\rho < 1/4$ and $x \in \{0,1\}^n$.

Output: A bit $b \in \{0,1\}$. *(Our goal: $x_j = b$.)*

Operation: Let e^j be the vector in $\{0,1\}^n$ that is equal to 0 in all the coordinates except for the jth and equal to 1 on the jth coordinate. The algorithm chooses $y \in_R \{0,1\}^n$ and outputs $f(y) + f(y + e^j) \pmod 2$ (where $y + e^j$ denotes componentwise addition modulo 2, or equivalently, flipping the jth coordinate of y).

Analysis: Since both y and $y + e^j$ are uniformly distributed (even though they are dependent), the union bound implies that with probability $1 - 2\rho$, $f(y) = x \odot y$ and $f(y + e^j) = x \odot (y + e^j)$. But by the bilinearity of the operation $\odot$, this implies that $f(y) + f(y + e^j) = x \odot y + x \odot (y + e^j) =$
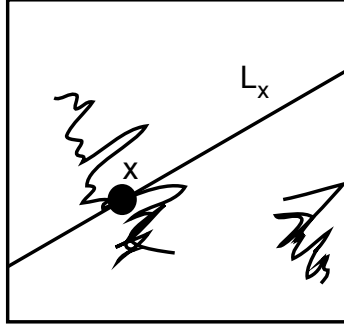
Figure 17.5: Given access to a corrupted version of a polynomial $P : \mathbb{F}^\ell \to \mathbb{F}$, to compute $P(x)$ we pass a random line L_x through x, and use Reed-Solomon decoding to recover the restriction of P to the line L_x.

$2(x \odot y) + x \odot e^j = x \odot e^j \pmod 2$. Yet, $x \odot e^j = x_j$ and so with probability $1 - 2\rho$, the algorithm outputs the right value.

REMARK 17.23
This algorithm can be modified to locally compute not just $x_i = x \odot e^j$ but in fact the value $x \odot z$ for every $z \in \{0,1\}^n$. Thus, we can use it to compute not just every bit of the original message x but also every bit of the uncorrupted codeword $\mathsf{WH}(x)$. This property is sometimes called the *self correction property* of the Walsh-Hadamard code.

17.6.2   Local decoder for Reed-Muller

We now show a local decoder for the Reed-Muller code. (Note that Definition 17.12 can be easily extended to the case of codes, such as Reed-Muller, that use *non-binary* alphabet.) It runs in time polynomial in ℓ and d, which, for an appropriate setting of the parameters, is polylogarithmic in the output length of the code. **Convention:** Recall that the input to a Reed-Muller code is an ℓ-variable d-degree polynomial P over some field $\mathbb{F}$. When we discussed the code before, we assumed that this polynomial is represented as the list of its coefficients. However, below it will be more convenient for us to assume that the polynomial is represented by a list of its values on its first $\binom{d+\ell}{\ell}$ inputs according to some canonical ordering. Using standard interpolation, we still have a polynomial-time encoding algorithm even given this representation. Thus, it suffices to show an algorithm that, given access to a corrupted version of P, computes $P(x)$ for every $x \in \mathbb{F}^\ell$

REED-MULLER LOCAL DECODER for $\rho < (1 - d/|\mathbb{F}|)/4 - 1/|\mathbb{F}|$.

Input: A string $x \in \mathbb{F}^\ell$, random access to a function f such that $\Pr_{x \in \mathbb{F}^\ell}[P(x) \neq f(x)] < \rho$, where $P : \mathbb{F}^\ell \to \mathbb{F}$ is an ℓ-variable degree-d polynomial.

Output: $y \in \mathbb{F}$ *(Goal: $y = P(x)$.)*

Operation: 1. Let L_x, be a random line passing through x. That is $L_x = \{x + ty : t \in \mathbb{F}\}$ for a random $y \in \mathbb{F}^\ell$.
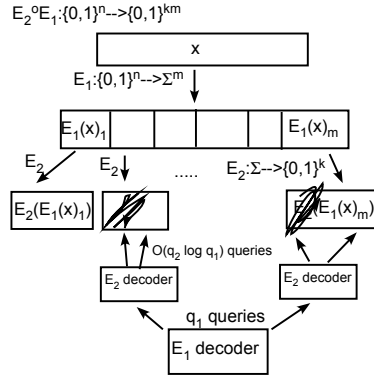
Figure 17.6: To locally decode a concatenated code $E_2 \circ E_1$ we run the decoder for E_1 using the decoder for E_2. The crucial observation is that if y is within $\rho_1 \rho_2$ distance to $E_2 \circ E_1(x)$ then at most a ρ_1 fraction of the blocks in y are of distance more than ρ_2 the corresponding block in $E_2 \circ E_1(x)$.

2. Query f on all the $|\mathbb{F}|$ points of L_x to obtain a set of points $\{(t, f(x + ty))\}$ for every $t \in \mathbb{F}$.

3. Run the Reed-Solomon decoding algorithm to obtain the univariate polynomial $Q : \mathbb{F} \to \mathbb{F}$ such that $Q(t) = f(x + ty)$ for the largest number of t's (see Figure 17.5).[3]

4. Output $Q(0)$.

Analysis: For every d-degree ℓ-variable polynomial P, the univariate polynomial $Q(t) = P(x + ty)$ has degree at most d. Thus, to show that the Reed-Solomon decoding works, it suffices to show that with probability at least $1/2$, the number of points on $z \in L_x$ for which $f(z) \neq P(z)$ is less than $(1 - d/|\mathbb{F}|)/2$. Yet, for every $t \neq 0$, the point $x + ty$ is uniformly distributed (independently of x), and so the expected number of points on L_x for which f and P differ is at most $\rho|\mathbb{F}| + 1$. By Markov inequality, the probability that there will be more than $2\rho|\mathbb{F}| + 2 < (1 - d/|\mathbb{F}|)|\mathbb{F}|/2$ such points is at most $1/2$ and hence Reed-Solomon decoding will be successful with probability $1/2$. In this case, we obtain the correct polynomial q that is the restriction of Q to the line L_x and hence $q(0) = P(x)$.

17.6.3 Local decoding of concatenated codes.

Given two locally decodable ECC's E_1 and E_2, we can locally decode their concatenation $E_1 \circ E_2$ by the natural algorithm. Namely, we run the decoder for E_1, but answer its queries using the decoder for E_2 (see Figure 17.6).

LOCAL DECODER FOR CONCATENATED CODE: $\rho < \rho_1 \rho_2$

The code: If $E_1 : \{0,1\}^n \to \Sigma^m$ and $E_2 : \Sigma \to \{0,1\}^k$ are codes with decoders of q_1 (resp. q_2) queries with respect to ρ_1 (resp. ρ_2) errors, let $E = E_2 \circ E_1$ be the concatenated code mapping $\{0,1\}^n$ to $\{0,1\}^{mk}$.

---

[3]If ρ is sufficiently small, (e.g., $\rho < 1/(10d)$), then we can use the simpler randomized Reed-Solomon decoding procedure described in Section 17.5.5.

Input: An index $i \in [n]$, random access to a string $y \in \{0,1\}^{km}$ such that $\Delta(y, E_1 \circ E_2(x)) < \rho_1 \rho_2$ for some $x \in \{0,1\}^n$.

Output: $b \in \{0,1\}^n$ *(Goal: $b = x_i$)*

Operation: Simulate the actions of the decoder for E_1, whenever the decoder needs access to the jth symbol of $E_1(x)$, use the decoder of E_2 with $O(q_2 \log q_1 \log |\Sigma|)$ queries applied to the jth block of y to recover all the bits of this symbol with probability at least $1 - 1/(2q_1)$.

Analysis: The crucial observation is that at most a ρ_1 fraction of the length k blocks in y can be of distance more than ρ_2 from the corresponding blocks in $E_2 \circ E_1(x)$. Therefore, with probability at least 0.9, all our q_1 answers to the decoder of E_1 are consistent with the answer it would receive when accessing a string that is of distance at most ρ_1 from a codeword of E_1.

17.6.4   Putting it all together.

We now have the ingredients to prove our second main theorem of this chapter: transformation of a hard-on-the-worst-case function into a function that is "mildly" hard on the average case.

> THEOREM 17.24 (WORST-CASE HARDNESS TO MILD HARDNESS)
> Let $S : \mathbb{N} \to \mathbb{N}$ and $f \in \mathbf{E}$ such that $\mathsf{H}_{\mathsf{wrs}}(f)(n) \geq S(n)$ for every n. Then there exists a function $g \in \mathbf{E}$ and a constant $c > 0$ such that $\mathsf{H}_{\mathsf{avg}}^{0.99}(g)(n) \geq S(n/c)/n^c$ for every sufficiently large n.

PROOF: For every n, we treat the restriction of f to $\{0,1\}^n$ as a string $f' \in \{0,1\}^N$ where $N = 2^n$. We then encode this string f' using a suitable error correcting code $E : \{0,1\}^N \to \{0,1\}^{N^C}$ for some constant $C > 1$. We will define the function g on every input $x \in \{0,1\}^{Cn}$ to output the xth coordinate of $E(f')$.[4] For the function g to satisfy the conclusion of the theorem, all we need is for the code E to satisfy the following properties:

1. For every $x \in \{0,1\}^N$, $E(x)$ can be computed in $\text{poly}(N)$ time.

2. There is a local decoding algorithm for E that uses $\text{polylog}(N)$ running time and queries and can handle a 0.01 fraction of errors.

But this can be achieved using a concatenation of a Walsh-Hadamard code with a Reed-Muller code of appropriate parameters:

1. Let RM denote the Reed-Muller code with the following parameters:

   - The field $\mathbb{F}$ is of size $\log^5 N$.
   - The number of variables ℓ is equal to $\log N / \log \log N$.

---

[4]By padding with zeros as necessary, we can assume that all the inputs to g are of length that is a multiple of C.

- The degree is equal to $\log^2 N$.

RM takes an input of length at least $(\frac{d}{\ell})^{\ell} > N$ (and so using padding we can assume its input is $\{0,1\}^n$). Its output is of size $|\mathbb{F}|^{\ell} \le \text{poly}(n)$. Its distance is at least $1 - 1/\log N$.

2. Let WH denote the Walsh-Hadamard code from $\{0,1\}^{\log F} = \{0,1\}^{5 \log \log N}$ to $\{0,1\}^{|\mathbb{F}|} = \{0,1\}^{\log^5 N}$.

Our code will be $\mathsf{WH} \circ \mathsf{RM}$. Combining the local decoders for Walsh-Hadamard and Reed-Muller we get the desired result. ∎

Combining Theorem 17.24 with Yao's XOR Lemma (Theorem 17.2), we get the following corollary:

COROLLARY 17.25
Let $S : \mathbb{N} \to \mathbb{N}$ and $f \in \mathbf{E}$ with $\mathsf{H}_{\mathsf{wrs}}(f)(n) \ge S(n)$ for every n. Then, there exists an $S(\sqrt{\ell})^{\epsilon}$-pseudorandom generator for some constant $\epsilon > 0$.

PROOF: By Theorem 17.24, under this assumption there exists a function $g \in \mathbf{E}$ with $\mathsf{H}_{\mathsf{avg}}^{0.99}(g)(n) \ge S'(n) = S(n)/\text{poly}(n)$, where we can assume $S'(n) \ge \sqrt{S(n)}$ for sufficiently large n (otherwise S is polynomial and the theorem is trivial). Consider the function $g^{\oplus k}$ where $k = c \log S'(n)$ for a sufficiently small constant c. By Yao's XOR Lemma, on inputs of length kn, it cannot be computed with probability better than $1/2 + 2^{-cS'(n)/1000}$ by circuits of size $S'(n)$. Since $S(n) \le 2^n$, $kn < \sqrt{n}$, and hence we get that $\mathsf{H}_{\mathsf{avg}}(g^{\oplus k}) \ge S^{c/2000}$. ∎

As already mentioned, this implies the following corollaries:

1. If there exists $f \in \mathbf{E}$ such that $\mathsf{H}_{\mathsf{wrs}}(f) \ge 2^{n^{\Omega(1)}}$ then $\mathbf{BPP} \subseteq \mathbf{QuasiP}$.

2. If there exists $f \in \mathbf{E}$ such that $\mathsf{H}_{\mathsf{wrs}}(f) \ge n^{\omega(1)}$ then $\mathbf{BPP} \subseteq \mathbf{SUBEXP}$.

However, Corollary 17.25 is still not sufficient to show that $\mathbf{BPP} = \mathbf{P}$ under any assumption on the worst-case hardness of some function in $\mathbf{E}$. It only yields an $S(\sqrt{\ell})^{\Omega(1)}$-pseudorandom generator, while what we need is an $S(\Omega(\ell))^{\Omega(1)}$-pseudorandom generator.

17.7 List decoding

Our approach to obtain stronger worst-case to average-case reduction will be to bypass the XOR Lemma, and use error correcting codes to get directly from worst-case hardness to a function that is hard to compute with probability slightly better than $1/2$. However, this idea seems to run into a fundamental difficulty: if f is worst-case hard, then it seems hard to argue that the encoding of f, *under any error correcting code* is hard to compute with probability 0.6. The reason is that any error-correcting code has to have distant at most $1/2$, which implies that there is no decoding algorithm that can recover x from $E(x)$ if the latter was corrupted in more than a $1/4$ of its locations. Indeed, in this case there is not necessarily a unique codeword closest to the corrupted word. For example, if $E(x)$ and $E(x')$ are two codewords of distance $1/2$, let y be the string that is equal to

$E(x)$ on the first half of the coordinates and equal to $E(x')$ on the second half. Given y, how can a decoding algorithm know whether to return x or x'?

This seems like a real obstacle, and indeed was considered as such in many contexts where ECC's were used, until the realization of the importance of the following insight: "If y is obtained by corrupting $E(x)$ in, say, a 0.4 fraction of the coordinates (where E is some ECC with good enough distance) then, while there may be more than one codeword within distance 0.4 to y, *there can not be too many such codewords*."

THEOREM 17.26 (JOHNSON BOUND)
If $E : \{0,1\}^n \to \{0,1\}^m$ is an ECC with distance at least $1/2 - \epsilon$, then for every $x \in \{0,1\}^m$, and $\delta \geq \sqrt{\epsilon}$, there exist at most $1/(2\delta^2)$ vectors $y_1, \ldots, y_\ell$ such that $\Delta(x, y_i) \leq 1/2 - \delta$ for every $i \in [\ell]$.

PROOF: Suppose that $x, y_1, \ldots, y_\ell$ satisfy this condition, and define ℓ vectors $z_1, \ldots, z_\ell$ in $\mathbb{R}^m$ as follows: for every $i \in [\ell]$ and $k \in [m]$, set $z_{i,k}$ to equal $+1$ if $y_k = x_k$ and set it to equal -1 otherwise. Under our assumptions, for every $i \in [\ell]$,

$$\sum_{k=1}^{m} z_{i,k} \geq 2\delta m, \tag{10}$$

since z_i agrees with x on an $1/2 + \delta$ fraction of its coordinates. Also, for every $i \neq j \in [\ell]$,

$$\langle z_i, z_j \rangle = \sum_{k=1}^{m} z_{i,k} z_{j,k} \leq 2\epsilon m \leq 2\delta^2 m \tag{11}$$

since E is a code of distance at least $1/2 - \epsilon$. We will show that (10) and (11) together imply that $\ell \leq 1/(2\delta^2)$.

Indeed, set $w = \sum_{i=1}^{\ell} z_i$. On one hand, by (11)

$$\langle w, w \rangle = \sum_{i=1}^{\ell} \langle z_i, z_i \rangle + \sum_{i \neq j} \langle z_i, z_j \rangle \leq \ell m + \ell^2 2\delta^2 m \,.$$

On the other hand, by (10), $\sum_k w_k = \sum_{i,j} z_{i,j} \geq 2\delta m \ell$ and hence

$$\langle w, w \rangle \geq |\sum_k w_k|^2/m \geq 4\delta^2 m \ell^2 \,,$$

since for every c, the vector $w \in \mathbb{R}^m$ with minimal two-norm satisfying $\sum_k w_k = c$ is the uniform vector $(c/m, c/m, \ldots, c/m)$. Thus $4\delta^2 m \ell^2 \leq \ell m + 2\ell^2 \delta^2 m$, implying that $\ell \leq 1/(2\delta^2)$. ∎

17.7.1 List decoding the Reed-Solomon code

In many contexts, obtaining a list of candidate messages from a corrupted codeword can be just as good as unique decoding. For example, we may have some outside information on which messages are likely to appear, allowing us to know which of the messages in the list is the correct one.

However, to take advantage of this we need an efficient algorithm that computes this list. Such an algorithm was discovered in 1996 by Sudan for the popular and important Reed-Solomon code. It can recover a polynomial size list of candidate codewords given a Reed-Solomon codeword that was corrupted in up to a $1 - 2\sqrt{d/|\mathbb{F}|}$ fraction of the coordinates. Note that this tends to 1 as $|\mathbb{F}|/d$ grows, whereas the Berlekamp-Welch unique decoding algorithm of Section 17.5.5 gets "stuck" when the fraction of errors surpasses $1/2$.

On input a set of data points $\{(a_i, b_i)\}_{i=1}^m$ in $\mathbb{F}^2$, Sudan's algorithm returns *all* degree d polynomials g such that the number of i's for which $g(a_i) = b_i$ is at least $2\sqrt{d/|\mathbb{F}|}m$. It relies on the following observation:

LEMMA 17.27
For every set of m data pairs $(a_1, b_1), \ldots, (a_m, b_m)$, there is a bivariate polynomial $Q(z, x)$ of degree at most $\lceil\sqrt{m}\rceil + 1$ in z and x such that $Q(b_i, a_i) = 0$ for each $i = 1, \ldots, m$. Furthermore, there is a polynomial-time algorithm to construct such a Q.

PROOF: Let $k = \lceil\sqrt{m}\rceil + 1$. Then the unknown bivariate polynomial $Q = \sum_{i=0}^{k}\sum_{j=0}^{k} Q_{ij}z^i x^j$ has $(k+1)^2$ coefficients and these coefficients are required to satisfy m linear equations of the form:

$$\sum_{i=0}^{k}\sum_{j=0}^{k} Q_{ij}(b_t)^i (a_t)^j \qquad \text{for } t = 1, 2, \ldots, m.$$

Note that the a_t's, b_t's are known and so we can write down these equations.

Since the system is homogeneous and the number of unknowns exceeds the number of constraints, it has a nonzero solution. Furthermore this solution can be found in polynomial time. ∎

LEMMA 17.28
Let d be any integer and $k > (d+1)(\lceil\sqrt{m}\rceil + 1)$. If $p(x)$ is a degree d polynomial that describes k of the data pairs, then $z - p(x)$ divides the bivariate polynomial $Q(z, x)$ described in Lemma 17.27.

PROOF: By construction, $Q(b_t, a_t) = 0$ for every data pair (a_t, b_t). If $p(x)$ describes this data pair, then $Q(p(a_t), a_t) = 0$. We conclude that the univariate polynomial $Q(p(x), x)$ has at least k roots, whereas its degree is $d(\lceil\sqrt{n}\rceil + 1) < k$. Hence $Q(p(x), x) = 0$. By the division algorithm for polynomials, $Q(p(x), x)$ is exactly the remainder when $Q(z, x)$ is divided by $(z - p(x))$. We conclude that $z - p(x)$ divides $Q(z, x)$. ∎

Now it is straightforward to describe Sudan's list decoding algorithm. First, find $Q(z, x)$ by the algorithm of Lemma 17.27. Then, factor it using a standard algorithm for bivariate factoring (see [VG99]). For every factor of the form $(z - p(x))$, check by direct substitution whether or not $p(x)$ describes $2\sqrt{d/|\mathbb{F}|}m$ data pairs. Output all such polynomials.

17.8   Local list decoding: getting to BPP = P.

Analogously to Section 17.4.1, to actually use list decoding for hardness amplification, we need to provide *local* list decoding algorithms for the codes we use. Fortunately, such algorithms are known for the Walsh-Hadamard code, the Reed-Muller code, and their concatenation.

DEFINITION 17.29 (LOCAL LIST DECODER)
Let $E : \{0,1\}^n \to \{0,1\}^m$ be an ECC and let $\rho > 0$ and q be some numbers. An algorithm L is called a *local list decoder for E handling ρ errors*, if for every $x \in \{0,1\}^n$ and $y \in \{0,1\}^m$ satisfying $\Delta(E(x), y) \leq \rho$, *there exists a number* $i_0 \in [poly(n/\epsilon)]$ such that for every $j \in [m]$, on inputs i_0, j and with random access to y, L runs for $\mathrm{poly}(\log(m)/\epsilon)$ time and outputs x_j with probability at least $2/3$.

REMARK 17.30
One can think of the number i_0 as the index of x in the list of $\mathrm{poly}(n/\epsilon)$ candidate messages output by L. Definition 17.29 can be easily generalized to codes with non-binary alphabet.

17.8.1   Local list decoding of the Walsh-Hadamard code.

It turns out we already encountered a local list decoder for the Walsh-Hadamard code: the proof of the Goldreich-Levin Theorem (Theorem 10.14) provided an an algorithm that given access to a "black box" that computes the function $y \mapsto x \odot y$ (for $x, y \in \{0,1\}^n$) with probability $1/2 + \epsilon$, computes a list of values $x_1, \ldots, x_{\mathrm{poly}(n/\epsilon)}$ such that $x_{i_0} = x$ for some i_0. In the context of that theorem, we could find the right value of x from that list by checking it against the value $f(x)$ (where f is a one-way permutation). This is a good example for how once we have a list decoding algorithm, we can use outside information to narrow the list down.

17.8.2   Local list decoding of the Reed-Muller code

We now present an algorithm for local list decoding of the Reed-Muller code. Recall that the codeword of this code is the list of evaluations of a d-degree ℓ-variable polynomial $P : \mathbb{F}^\ell \to \mathbb{F}$. The local decoder for Reed-Muller gets random access to a corrupted version of P and two inputs: an index i and $x \in \mathbb{F}^\ell$. Below we describe such a decoder that runs in $\mathrm{poly}(d, \ell, |\mathbb{F}|)$ and outputs $P(x)$ with probability at least 0.9 assuming that i is equal to the "right" index i_0. **Note:** To be a valid local list decoder, given the index i_0, the algorithm should output $P(x)$ with high probability for *every* $x \in \mathbb{F}^\ell$. The algorithm described below is only guaranteed to output the right value for *most* (i.e., a 0.9 fraction) of the x's in $\mathbb{F}^\ell$. We transform this algorithm to a valid local list decoder by combining it with the Reed-Muller local decoder described in Section 17.6.2.

REED-MULLER LOCAL LIST DECODER for $\rho < 1 - 10\sqrt{d/|\mathbb{F}|}$

Inputs: • Random access to a function f such that $\Pr_{x \in \mathbb{F}^\ell}[P(x) = f(x)] > 10\sqrt{d/|\mathbb{F}|}$ where $P : \mathbb{F}^\ell \to \mathbb{F}$ is an ℓ-variable d-degree polynomial. We assume $|\mathbb{F}| > d^4$ and that both $d > 1000$. (This can always be ensured in our applications.)

   • An index $i_0 \in [|\mathbb{F}|^{\ell+1}]$ which we interpret as a pair (x_0, y_0) with $x_0 \in \mathbb{F}^\ell$, $y_0 \in \mathbb{F}$,

   • A string $x \in \mathbb{F}^\ell$.

Output: $y \in \mathbb{F}$ *(For some pair (x_0, y_0), it should hold that $P(x) = y$ with probability at least 0.9 over the algorithm's coins and x chosen at random from $\mathbb{F}^\ell$.)*

Operation: 1. Let L_{x,x_0} be a random degree 3 curve passing through x, x_0. That is, we find a random degree 3 univariate polynomial $q : \mathbb{F} \to \mathbb{F}^\ell$ such that $q(0) = x$ and $q(r) = x_0$ for some random $r \in \mathbb{F}$. (See Figure 17.7.)

2. Query f on all the $|\mathbb{F}|$ points of L_{x,x_0} to obtain the set $\mathcal{S}$ of the $|\mathbb{F}|$ pairs $\{(t, f(q(t)) : t \in \mathbb{F})\}$.

3. Run Sudan's Reed-Solomon list decoding algorithm to obtain a list $g_1, \ldots, g_k$ of all degree $3d$ polynomials that have at least $8\sqrt{d|\mathbb{F}|}$ agreement with the pairs in $\mathcal{S}$.

4. If there is a unique i such that $g_i(r) = y_0$ then output $g_i(0)$. Otherwise, halt without outputting anything.
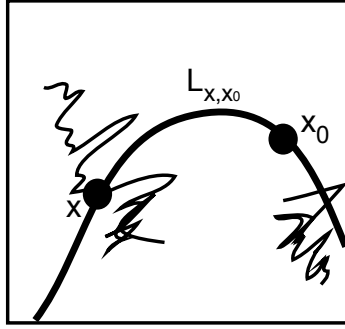
Figure 17.7: Given access to a corrupted version of a polynomial $P : \mathbb{F}^\ell \to \mathbb{F}$ and some index (x_0, y_0), to compute $P(x)$ we pass a random degree-3 curve L_{x,x_0} through x and x_0, and use Reed-Solomon list decoding to recover a list of candidates for the restriction of P to the curve L_{x,x_0}. If only one candidate satisfies that its value on x_0 is y_0, then we use this candidate to compute $P(x)$.

We will show that for every $f : \mathbb{F}^\ell \to \mathbb{F}$ that agrees with an ℓ-variable degree d polynomial on a $10\sqrt{d/|\mathbb{F}|}$ fraction of its input, and every $x \in \mathbb{F}^\ell$, if x_0 is chosen at random from $\mathbb{F}^\ell$ and $y_0 = P(x_0)$, then with probability at least 0.9 (over the choice of x_0 and the algorithm's coins) the above decoder will output $P(x)$. By a standard averaging argument, this implies that there exist a pair (x_0, y_0) such that given this pair, the algorithm outputs $P(x)$ for a 0.9 fraction of the x's in $\mathbb{F}^\ell$.

Let $x \in \mathbb{F}^\ell$, if x_0 is chosen randomly in $\mathbb{F}^\ell$ and $y_0 = P(x_0)$ then the following

For every $x \in \mathbb{F}^\ell$, the following fictitious algorithm can be easily seen to have an identical output to the output of our decoder on the inputs x, a random $x_0 \in_R \mathbb{F}^\ell$ and $y_0 = P(x_0)$:

1. Choose a random degree 3 curve L that passes through x. That is, $L = \{q(t) : t \in \mathbb{F}\}$ where $q : \mathbb{F} \to \mathbb{F}^\ell$ is a random degree 3 polynomial satisfying $q(0) = x$.

2. Obtain the list $g_1, \ldots, g_m$ of all univariate polynomials over $\mathbb{F}$ such that for every i, there are at least $6\sqrt{d|\mathbb{F}|}$ values of t such that $g_i(t) = f(q(t))$.

3. Choose a random $r \in \mathbb{F}$. Assume that you are given the value $y_0 = P(q(r))$.

4. If there exists a unique i such that $g_i(r) = y_0$ then output $g_i(0)$. Otherwise, halt without an input.

Yet, this fictitious algorithm will output $P(x)$ with probability at least 0.9. Indeed, since all the points other than x on a random degree 3 curve passing through x are pairwise independent, Chebyshev's inequality implies that with probability at least 0.99, the function f will agree with the polynomial P on at least $8\sqrt{d|\mathbb{F}|}$ points on this curve (this uses the fact that $\sqrt{d/|\mathbb{F}|}$ is smaller than 10^{-6}). Thus the list $g_1, \ldots, g_m$ we obtain in Step 2 contains the polynomial $g : \mathbb{F} \to \mathbb{F}$ defined as $g(t) = P(q(t))$. We leave it as Exercise 9 to show that there can not be more than $\sqrt{|F|}/4d$ polynomials in this list. Since two $3d$-degree polynomials can agree on at most $3d + 1$ points, with probability at least $\frac{(3d+1)\sqrt{|F|}/4d}{|\mathbb{F}|} < 0.01$, if we choose a random $r \in \mathbb{F}$, then $g(r) \neq g_i(r)$ for every $g_i \neq g$ in this list. Thus, with this probability, we will identify the polynomial g and output the value $g(0) = P(x)$. ∎

17.8.3 Local list decoding of concatenated codes.

If $E_1 : \{0,1\}^n \to \Sigma^m$ and $E_2 : \Sigma \to \{0,1\}^k$ are two codes that are locally list decodable then so is the concatenated code $E_2 \circ E_1 : \{0,1\}^n \to \{0,1\}^{mk}$. As in Section 17.6.3, the idea is to simply run the local decoder for E_1 while answering its queries using the decoder of E_2. More concretely, assume that the decoder for E_1 takes an index in the set I_1, uses q_1 queries, and can handle $1 - \epsilon_1$ errors, and that I_2, q_2 and ϵ_2 are defined analogously. Our decoder for $E_2 \circ E_1$ will take a pair of indices $i_1 \in I_1$ and $i_2 \in I_2$ and run the decoder for E_1 with the index i_1, and whenever this decoder makes a query answer it using the decoder E_2 with the index i_2. (See Section 17.6.3.) We claim that this decoder can handle $1/2 - \epsilon_1\epsilon_2|I_2|$ number of errors. Indeed, if y agrees with some codeword $E_2 \circ E_1(x)$ on an $\epsilon_1\epsilon_2|I_2|$ fraction of the coordinates then there are $\epsilon_1|I_2|$ blocks on which it has at least $1/2 + \epsilon_2$ agreement with the blocks this codeword. Thus, by an averaging argument, there exists an index i_2 such that given i_2, the output of the E_2 decoder agrees with $E_1(x)$ on ϵ_1 symbols, implying that there exists an index i_1 such that given (i_1, i_2) and every coordinate j, the combined decoder will output x_j with high probability.

17.8.4 Putting it all together.

As promised, we can use local list decoding to transform a function that is merely worst-case hard into a function that cannot be computed with probability significantly better than $1/2$:

THEOREM 17.31 (WORST-CASE HARDNESS TO STRONG HARDNESS)
Let $S : \mathbb{N} \to \mathbb{N}$ and $f \in \mathbf{E}$ such that $\mathsf{H}_{\mathsf{wrs}}(f)(n) \geq S(n)$ for every n. Then there exists a function $g \in \mathbf{E}$ and a constant $c > 0$ such that $\mathsf{H}_{\mathsf{avg}}(g)(n) \geq S(n/c)^{1/c}$ for every sufficiently large n.

PROOF SKETCH: As in Section 17.6.4, for every n, we treat the restriction of f to $\{0,1\}^n$ as a

string $f' \in \{0,1\}^N$ where $N = 2^n$ and encode it using the concatenation of a Reed-Muller code with the Walsh-Hadamard code. For the Reed-Muller code we use the following parameters:

- The field $\mathbb{F}$ is of size $S(n)^{1/100}$. [5]

- The degree d is of size $\log^2 N$.

- The number of variables ℓ is $2 \log N / \log S(n)$.

The function g is obtained by applying this encoding to f. Given a circuit of size $S(n)^{1/100}$ that computes g with probability better than $1/2 + 1/S(n)^{1/50}$, we will be able to transform it, in $S(n)^{O(1)}$ time, to a circuit computing f perfectly. We hardwire the index i_0 to this circuit as part of its description. ∎

WHAT HAVE WE LEARNED?

- Yao's XOR Lemma allows to amplify hardness by transforming a Boolean function with only mild hardness (cannot be computed with say 0.99 success) into a Boolean function with strong hardness (cannot be computed with 0.51 success).

- An *error correcting code* is a function that maps every two strings into a pair of strings that differ on many of their coordinates. An error correcting code with a *local decoding* algorithm can be used to transform a function hard in the worst-case into a function that is mildly hard on the average case.

- A code over the binary alphabet can have distance at most $1/2$. A code with distance δ can be uniquely decoded up to $\delta/2$ errors. *List decoding* allows to a decoder to handle almost a δ fraction of errors, at the expense of returning not a single message but a short list of candidate messages.

- We can transform a function that is merely hard in the worst case to a function that is strongly hard in the average case using the notion of *local list decoding* of error correcting codes.

Chapter notes and history

MANY ATTRIBUTIONS STILL MISSING.

Impagliazzo and Wigderson [IW01] were the first to prove that **BPP** = **P** if there exists $f \in \mathbf{E}$ such that $\mathsf{H}_{\mathsf{wrs}}(f) \geq 2^{\Omega(n)}$ using a *derandomized* version of Yao's XOR Lemma. However,

---

[5] We assume here that $S(n) > \log N^{1000}$ and that it can be computed in $2^{O(n)}$ time. These assumptions can be removed by slightly complicating the construction (namely, executing it while guessing that $S(n) = 2^k$, and concatenating all the results.)

the presentation here follows Sudan, Trevisan, and Vadhan [STV], who were the first to point the connection between local list decoding and hardness amplification, and gave (a variant of) the Reed-Muller local list decoding algorithm described in Section 17.8. They also showed a different approach to achieve the same result, by first showing that the NW generator and a mildly hard function can be used to obtain from a short random seed a distribution that has high *pseudoentropy*, which is then converted to a pseudorandom distribution via a randomness extractor (see Chapter 16).

The question raised in Problem 5 is treated in O'Donnell [O'D04], where a hardness amplification lemma is given for $\mathbf{NP}$. For a sharper result, see Healy, Vadhan, and Viola [HVV04].

Exercises

§1 Let $X_1, \ldots, X_n$ be independent random variables such that X_i is equal to 1 with probability $1 - \delta$ and equal to 0 with probability δ. Let $X = \sum_{i=1}^{k} X_i \pmod 2$. Prove that $\Pr[X = 1] = 1/2 + (1 - 2\delta)^k$.

<div style="transform: rotate(180deg)">

Hint: Define $Y_i = (-1)^{X_i}$ and $Y = \prod_{i=1}^{k} Y_i$. Then, use the fact that the expectation of a product of independent random variables is the product of their expectations.

</div>

§2 Prove Farkas' Lemma: if $C, D \subseteq \mathbb{R}^m$ are two convex sets then there exists a vector $\mathbf{z} \in \mathbb{R}^m$ and a number $a \in \mathbb{R}$ such that

$$\mathbf{x} \in C \Rightarrow \langle \mathbf{x}, \mathbf{z} \rangle \geq a$$
$$\mathbf{y} \in D \Rightarrow \langle \mathbf{y}, \mathbf{z} \rangle \leq a$$

<div style="transform: rotate(180deg)">

Hint: Start by proving this in the case that C and D are e-separated, which means that for some $\epsilon > 0$, $\|\mathbf{x} - \mathbf{y}\|_2 \geq \epsilon$ for every $\mathbf{x} \in C$ and $\mathbf{y} \in D$. In this case you can take $\mathbf{z}$ to be the shortest vector of the form $\mathbf{x} - \mathbf{y}$ for $\mathbf{x} \in C$ and $\mathbf{y} \in D$.

</div>

§3 Prove the Min-Max Theorem (see Note 17.7) using Farkas' Lemma.

§4 Prove the duality theorem for linear programming using Farkas' Lemma. That is, prove that for every $m \times n$ matrix A, and vectors $\mathbf{c} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^n$,

$$\max_{\substack{\mathbf{x} \in \mathbb{R}^n \text{ s.t.} \\ A\mathbf{x} \leq \mathbf{b} \\ \mathbf{x} \geq \mathbf{0}}} \langle \mathbf{x}, \mathbf{c} \rangle = \min_{\substack{\mathbf{y} \in \mathbb{R}^m \text{ s.t.} \\ A^\dagger \mathbf{y} \geq \mathbf{c} \\ \mathbf{y} \geq \mathbf{0}}} \langle \mathbf{y}, \mathbf{b} \rangle$$

where $A^\dagger$ denotes the transpose of A and for two vectors $\mathbf{u}, \mathbf{v}$ we say that $\mathbf{u} \geq \mathbf{v}$ if $u_i \geq v_i$ for every i.

§5 Suppose we know that $\mathbf{NP}$ contains a function that is weakly hard for all polynomial-size circuits. Can we use the XOR Lemma to infer the existence of a strongly hard function in $\mathbf{NP}$? Why or why not?

§6 For every $\delta < 1/2$ and sufficiently large n, prove that there exists a function $E : \{0, 1\}^n \to \{0, 1\}^{n/(1-H(\delta))}$ that is an error correcting code with distance δ, where $H(\delta) = \delta \log(1/\delta) + (1 - \delta) \log(1/(1 - \delta))$.

Hint: Use a greedy strategy, to select the codewords of E one by one, never adding a codeword that is within distance δ to previous ones. When will you get stuck?

§7 Show that for every $E : \{0, 1\}^n \to \{0, 1\}^m$ that is an error correcting code of distance $1/2$, $2^n < 10\sqrt{n}$. Show if E is an error correcting code of distance $\delta > 1/2$, then $2^n < 10/(\delta - 1/2)$.

§8 Let $E : \{0, 1\}^n \to \{0, 1\}^m$ be a δ-distance ECC. Transform E to a code $E' : \{0, 1, 2, 3\}^{n/2} \to \{0, 1, 2, 3\}^{m/2}$ in the obvious way. Show that E' has distance δ. Show that the opposite direction is not true: show an example of a δ-distance ECC $E' : \{0, 1, 2, 3\}^{n/2} \to \{0, 1, 2, 3\}^{m/2}$ such that the corresponding binary code has distance 2δ.

§9 Let $f : \mathbb{F} \to \mathbb{F}$ be any function. Suppose integer $d \geq 0$ and number ϵ satisfy $\epsilon > 2\sqrt{\frac{d}{|\mathbb{F}|}}$. Prove that there are at most $2/\epsilon$ degree d polynomials that agree with f on at least an ϵ fraction of its coordinates.

Hint: The first polynomial describes f in an ϵ fraction of points say S_1, the second polynomial describes f in $\epsilon - d/|\mathbb{F}|$ fraction of points S_2 where $S_1 \cup S_2 = \emptyset$, etc.

§10 *(Linear codes)* We say that an ECC $E : \{0, 1\}^n \to \{0, 1\}^m$ is *linear* if for every $x, x' \in \{0, 1\}^n$, $E(x + x') = E(x) + E(x')$ where $+$ denotes componentwise addition modulo 2. A linear ECC E can be described by an $m \times n$ matrix A such that (thinking of x as a column vector) $E(x) = Ax$ for every $x \in \{0, 1\}^n$.

   (a) Prove that the distance of a linear ECC E is equal to the minimum over all nonzero $x \in \{0, 1\}^n$ of the fraction of 1's in $E(x)$.

   (b) Prove that for every $\delta > 0$, there exists a linear ECC $E : \{0, 1\}^n \to \{0, 1\}^{1.1n/(1-H(\delta))}$ with distance δ, where $H(\delta) = \delta \log(1/\delta) + (1 - \delta) \log(1/(1 - \delta))$¿

   Hint: Use the probabilistic method - show this holds for a random matrix.

   (c) Prove that for some $\delta > 0$ there is an ECC $E : \{0, 1\}^n \to \{0, 1\}^{\mathrm{poly}(n)}$ of distance δ with polynomial-time encoding and decoding mechanisms. (You need to know about the field $GF(2^k)$ to solve this, see Appendix A.)

   Hint: Use the concatenation of Reed-Solomon over $GF(2^k)$ with the Walsh-Hadamard code.

   (d) We say that a linear code $E : \{0, 1\}^n \to \{0, 1\}^m$ is ϵ-*biased* if for every non-zero $x \in \{0, 1\}^n$, the fraction of 1's in $E(x)$ is between $1/2 - \epsilon$ and $1/2 + \epsilon$. Prove that for every $\epsilon > 0$, there exists an ϵ-biased linear code $E : \{0, 1\}^n \to \{0, 1\}^{\mathrm{poly}(n/\epsilon)}$ with a polynomial-time encoding algorithm.

Chapter 18

PCP and Hardness of Approximation

"...most problem reductions do not create or preserve such gaps...To create such a gap in the generic reduction (cf. Cook)...also seems doubtful. The intuitive reason is that computation is an inherently unstable, non-robust mathematical object, in the the sense that it can be turned from non-accepting to accepting by changes that would be insignificant in any reasonable metric."
Papadimitriou and Yannakakis, 1991 [PY91]

The **PCP** Theorem provides an interesting new characterization for **NP**, as the set of languages that have a "locally testable" membership proof. It is reminiscent of —and was motivated by— results such as **IP** =**PSPACE**. Its essence is the following:

Suppose somebody wants to convince you that a Boolean formula is satisfiable. He could present the usual certificate, namely, a satisfying assignment, which you could then check by substituting back into the formula. However, doing this requires reading the entire certificate. The **PCP** Theorem shows an interesting alternative: this person can easily rewrite his certificate so you can verify it by probabilistically selecting a constant number of locations—as low as 3 bits— to examine in it. Furthermore, this probabilistic verification has the following properties: **(1)** A correct certificate will never fail to convince you (that is, no choice of your random coins will make you reject it) and **(2)** If the formula is unsatisfiable, then you are guaranteed to reject *every claimed certificate* with high probability.

Of course, since Boolean satisfiability is **NP**-complete, every other **NP** language can be deterministically and efficiently reduced to it. Thus the **PCP** Theorem applies to every **NP** language. We mention one counterintuitive consequence. Let $\mathcal{A}$ be any one of the usual axiomatic systems of mathematics for which proofs can be verified by a deterministic TM in time that is polynomial in the length of the proof. Recall the following language is in **NP**:

$$L = \{\langle \varphi, 1^n \rangle : \varphi \text{ has a proof in } \mathcal{A} \text{ of length } \leq n\}.$$

The **PCP** Theorem asserts that L has probabilistically checkable certificates. Such certificate can be viewed as an alternative notion of "proof" for mathematical statements that is just as valid as the usual notion. However, unlike standard mathematical proofs, where every line of the proof

has to be checked to verify its validity, this new notion guarantees that proofs are probabilistically checkable by examining only a constant number of bits in them[1].

This new, "robust" notion of certificate/proof has an important consequence: it implies that many optimization problems are **NP**-hard not only to solve exactly but even to *approximate*. As mentioned in Chapter 2, the **P** versus **NP** question is practically important —as opposed to "just" philosophically important— because thousands of real-life combinatorial optimization problems are **NP**-hard. By showing that even computing approximate solutions to many of these problems is **NP**-hard, the **PCP** Theorem extends the practical importance of the theory of **NP**-completeness, as well as its philosophical significance.

This seemingly mysterious connection between the **PCP** Theorem —which concerns probabilistic checking of certificates— and the **NP**-hardness of computing approximate solutions is actually quite straightforward. All **NP**-hardness results ultimately derive from the Cook-Levin theorem (Section 2.3), which expresses accepting computations of a nondeterministic Turing Machine with satisfying assignments to a Boolean formula. Unfortunately, the standard representations of computation are quite nonrobust, meaning that they can be incorrect if even one bit is incorrect (see the quote at the start of this chapter). The **PCP** Theorem, by giving a *robust* representation of the certificate for **NP** languages, allow new types of reductions; see Section 18.2.3.

Below, we use the term "**PCP** Theorems" for the body of other results of a similar nature to the **PCP** Theorem that found numerous applications in complexity theory. Some important ones appear in the next Chapter, including one that improves the **PCP** Theorem so that verification is possible by reading only 3 bits in the proof!

18.1   PCP and Locally Testable Proofs

According to our usual definition, language L is in **NP** if there is a poly-time Turing machine V ("verifier") that, given input x, checks certificates (or membership proofs) to the effect that $x \in L$. This means,

$$x \in L \Rightarrow \exists \pi \text{ s.t. } V^\pi(x) = 1$$
$$x \notin L \Rightarrow \forall \pi \quad V^\pi(x) = 0,$$

where V^π denotes "a verifier with access to certificate π".

The class **PCP** (short for "Probabilistically Checkable Proofs") is a generalization of this notion, with the following changes. First, the verifier is probabilistic. Second, the verifier has *random access* to the proof string Π. This means that each bit of the proof string can be independently *queried* by the verifier via a special *address tape*: if the verifier desires say the ith bit in the proof string, it writes i on the address tape and then receives the bit $\pi[i]$.[2] (This is reminiscent of oracle TMs introduced in Chapter 3.) The definition of **PCP** treats *queries* to the proof as a precious resource, to be used sparingly. Note also that since the address size is *logarithmic* in the proof size, this model in principle allows a polynomial-time verifier to check membership proofs of exponential size.

---

[1]One newspaper article about the discovery of the **PCP** Theorem carried the headline "New shortcut found for long math proofs!"

[2]Though widely used, the term "random access" is misleading since it doesn't involve any notion of randomness per se. "Indexed access" would be more accurate.

Verifiers can be *adaptive* or *nonadaptive*. A nonadaptive verifier selects its queries based only on its input and random tape, whereas an adaptive verifier can in addition rely upon bits it has already queried in π to select its next queries. We restrict verifiers to be nonadaptive, since most **PCP** Theorems can be proved using nonadaptive verifiers. (But Exercise 3 explores the power of adaptive queries.)
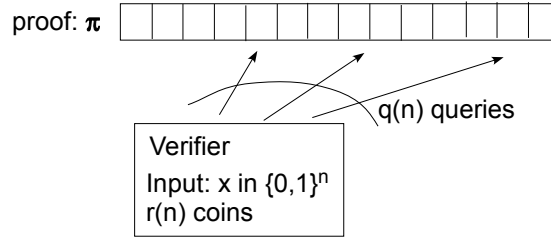
Figure 18.1: A **PCP** verifier for a language L gets an input x and random access to a string π. If $x \in L$ then there exists a string π that makes the verifier accepts, while if $x \notin L$ then the verifier rejects *every* proof π with probability at least $1/2$.

DEFINITION 18.1 ((r,q)-VERIFIER)
Let L be a language and $q, r : \mathbb{N} \to \mathbb{N}$. We say that L has an $(r(n), q(n))$-*verifier* if there's a polynomial-time probabilistic algorithm V satisfying:

Efficiency: On input a string $x \in \{0,1\}^n$ and given random access to a string $\pi \in \{0,1\}^*$ (which we call the *proof*), V uses at most $r(n)$ random coins and makes at most $q(n)$ non-adaptive queries to locations of π (see Figure 18.1). Then it outputs "1"(for "accept") or "0" (for "reject"). We use the notation $V^\pi(x)$ to denote the random variable representing V's output on input x and with random access to π.

Completeness: If $x \in L$ then there exists a proof $\pi \in \{0,1\}^*$ such that $\Pr[V^\pi(x) = 1] = 1$. We call π the *correct proof* for x.

Soundness: If $x \notin L$ then for every proof $\pi \in \{0,1\}^*$, $\Pr[V^\pi(x) = 1] \leq 1/2$.

We say that a language L is in **PCP**$(r(n), q(n))$ if L has a $(c \cdot r(n), d \cdot q(n))$-verifier for some constants c, d.

Sometimes we consider verifiers for which the probability "$1/2$" is replaced by some other number, called the *soundness parameter*.

THEOREM 18.2 (**PCP** THEOREM [AS98, ALM$^+$98])
NP = **PCP**$(\log n, 1)$.

Notes:

1. Without loss of generality, proofs checkable by an (r, q)-verifier contain at most $q2^r$ bits. The verifier looks at only q places of the proof for any particular choice of its random coins, and there are only 2^r such choices. Any bit in the proof that is read with 0 probability (i.e., for no choice of the random coins) can just be deleted.

2. The previous remark implies $\mathbf{PCP}(r(n), q(n)) \subseteq \mathbf{NTIME}(2^{O(r(n))}q(n))$. The proofs check-able by an $(r(n), q(n))$-verifier have size at most $2^{O(r(n))}q(n)$. A nondeterministic machine could guess the proof in $2^{O(r(n))}q(n)$ time, and verify it deterministically by running the verifier for all $2^{O(r(n))}$ possible choices of its random coin tosses. If the verifier accepts for all these possible coin tosses then the nondeterministic machine accepts.

   As a special case, $\mathbf{PCP}(\log n, 1) \subseteq \mathbf{NTIME}(2^{O(\log n)}) = \mathbf{NP}$: this is the trivial direction of the **PCP** Theorem.

3. The constant $1/2$ in the soundness requirement of Definition 18.1 is arbitrary, in the sense that changing it to any other positive constant smaller than 1 will not change the class of languages defined. Indeed, a **PCP** verifier with soundness $1/2$ that uses r coins and makes q queries can be converted into a **PCP** verifier using cr coins and cq queries with soundness 2^{-c} by just repeating its execution c times (see Exercise 1).

EXAMPLE 18.3

To get a better sense for what a **PCP** proof system looks like, we sketch two nontrivial **PCP** systems:

1. The language GNI of pairs of non-isomorphic graphs is in $\mathbf{PCP}(poly(n), 1)$. Say the input for GNI is $\langle G_0, G_1 \rangle$, where G_0, G_1 have both n nodes. The verifier expects π to contain, for each labeled graph H with n nodes, a bit $\pi[H] \in \{0, 1\}$ corresponding to whether $H \equiv G_0$ or $H \equiv G_1$ ($\pi[H]$ can be arbitrary if neither case holds). In other words, π is an (exponentially long) array of bits indexed by the (adjacency matrix representations of) all possible n-vertex graphs.

   The verifier picks $b \in \{0, 1\}$ at random and a random permutation. She applies the permutation to the vertices of G_b to obtain an isomorphic graph, H. She queries the corresponding bit of π and accepts iff the bit is b.

   If $G_0 \not\equiv G_1$, then clearly a proof π can be constructed which makes the verifier accept with probability 1. If $G_1 \equiv G_2$, then the probability that any π makes the verifier accept is at most $1/2$.

2. The protocols in Chapter 8 can be used (see Exercise 5) to show that the *permanent* has **PCP** proof system with polynomial randomness and queries. Once again, the length of the proof will be exponential.

In fact, both of these results are a special case of the following theorem a "scaled-up" version of the **PCP** Theorem which we will not prove.

THEOREM 18.4 (SCALED-UP **PCP**, [**?**, ALM$^+$98, AS98])
$\mathbf{PCP}(poly, 1) = \mathbf{NEXP}$

18.2 PCP and Hardness of Approximation

The **PCP** Theorem implies that for many **NP** optimization problems, computing near-optimal solutions is no easier than computing exact solutions.

We illustrate the notion of approximation algorithms with an example. MAX 3SAT is the problem of finding, given a 3CNF Boolean formula φ as input, an assignment that maximizes the number of satisfied clauses. This problem is of course **NP**-hard, because the corresponding decision problem, 3SAT, is **NP**-complete.

DEFINITION 18.5
For every 3CNF formula φ, define $\mathsf{val}(\varphi)$ to be the maximum fraction of clauses that can be satisfied by any assignment to φ's variables. In particular, if φ is satisfiable then $\mathsf{val}(\varphi) = 1$.

Let $\rho \leq 1$. An algorithm A is a ρ-*approximation* algorithm for MAX 3SAT if for every 3CNF formula φ with m clauses, $A(\varphi)$ outputs an assignment satisfying at least $\rho \cdot \mathsf{val}(\varphi)m$ of φ's clauses.

In many practical settings, obtaining an approximate solution to a problem may be almost as good as solving it exactly. Moreover, for some computational problems, approximation is much easier than an exact solution.

EXAMPLE 18.6 (1/2-APPROXIMATION FOR MAX 3SAT)
We describe a polynomial-time algorithm that computes a 1/2-approximation for MAX 3SAT. The algorithm assigns values to the variables one by one in a greedy fashion, whereby the ith variable is assigned the value that results in satisfying at least 1/2 the clauses in which it appears. Any clause that gets satisfied is removed and not considered in assigning values to the remaining variables. Clearly, the final assignment will satisfy at least 1/2 of all clauses, which is certainly at least half of the maximum that the optimum assignment could satisfy.

Using semidefinite programming one can also design a polynomial-time $(7/8 - \epsilon)$-approximation algorithm for every $\epsilon > 0$ (see references). (Obtaining such a ratio is trivial if we restrict ourselves to $3CNF$ formulae with three distinct variables in each clause. Then a random assignment has probability $7/8$ to satisfy it, and by linearity of expectation, is expected to satisfy a $7/8$ fraction of the clauses. This observation can be turned into a simple probabilistic or even deterministic $7/8$-approximation algorithm.)

For a few problems, one can even design $(1 - \epsilon)$-approximation algorithms for *every* $\epsilon > 0$. Exercise 10 asks you to show this for the **NP**-complete *knapsack* problem.

Researchers are extremely interested in finding the best possible approximation algorithms for **NP**-hard optimization problems. Yet, until the early 1990's most such questions were wide open. In particular, we did not know whether MAX 3SAT has a polynomial-time ρ-approximation algorithm for *every* $\rho < 1$. The **PCP** Theorem has the following Corollary.

COROLLARY 18.7
There exists some constant $\rho < 1$ such that if there is a polynomial-time ρ-approximation algorithm for MAX 3SAT *then* **P** = **NP**.

Later, in Chapter 19, we show a stronger **PCP** Theorem by Håstad which implies that for every $\epsilon > 0$, if there is a polynomial-time $(7/8+\epsilon)$-approximation algorithm for MAX 3SAT then $\mathbf{P} = \mathbf{NP}$. Hence the approximation algorithm for this problem mentioned in Example 18.6 is very likely *optimal*. The **PCP** Theorem (and the other **PCP** theorems that followed it) imply a host of such *hardness of approximation* results for many important problems, often showing that known approximation algorithms are optimal.

18.2.1 Gap-producing reductions

To prove Corollary 18.7 for some fixed $\rho < 1$, it suffices to give a polynomial-time reduction f that maps $3CNF$ formulae to $3CNF$ formulae such that:

$$\varphi \in \mathsf{3SAT} \Rightarrow \mathsf{val}(f(\varphi)) = 1 \tag{1}$$

$$\varphi \notin L \Rightarrow \mathsf{val}(f(\varphi)) < \rho \tag{2}$$

After all, if a ρ-approximation algorithm were to exist for MAX 3SAT, then we could use it to decide membership of any given formula φ in 3SAT by applying reduction f on φ and then running the approximation algorithm on the resultant $3CNF$ formula $f(\varphi)$. If $\mathsf{val}(f(\varphi) = 1$, then the approximation algorithm would return an assignment that satisfies at least ρ fraction of the clauses, which by property (2) tells us that $\varphi \in \mathsf{3SAT}$.

Later (in Section 18.2) we show that the **PCP** Theorem is equivalent to the following Theorem:

THEOREM 18.8
There exists some $\rho < 1$ and a polynomial-time reduction f satisfying (1) and (2).

By the discussion above, Theorem 18.8 implies Corollary 18.7 and so rules out a polynomial-time ρ-approximation algorithm for MAX 3SAT (unless $\mathbf{P} = \mathbf{NP}$).

Why doesn't the Cook-Levin reduction suffice to prove Theorem 18.8?   The first thing one would try is the reduction from any **NP** language to 3SAT in the Cook-Levin Theorem (Theorem 2.10). Unfortunately, it doesn't give such an f because it does not satisfy property (2): we can always satisfy almost all of the clauses in the formulae produced by the reduction (see Exercise 9 and also the "non-robustness" quote at the start of this chapter).

18.2.2 Gap problems

The above discussion motivates the definition of *gap problems*, a notion implicit in (1) and (2). It is also an important concept in the proof of the **PCP** Theorem itself.

DEFINITION 18.9 (GAP 3SAT)
Let $\rho \in (0, 1)$. The ρ-GAP 3SAT problem is to determine, given a 3CNF formula φ whether:

- φ is satisfiable, in which case we say φ is a YES instance of ρ-GAP 3SAT.

- $\mathsf{val}(\varphi) \leq \rho$, in which case we say φ is a NO instance of ρ-GAP 3SAT.

An algorithm A is said to solve ρ-GAP 3SAT if $A(\varphi) = 1$ if φ is a YES instance of ρ-GAP 3SAT and $A(\varphi) = 0$ if φ is a NO instance. Note that we do not make any requirement on $A(\varphi)$ if φ is neither a YES nor a NO instance of ρ-GAP qCSP.

Our earlier discussion of the desired reduction f can be formalized as follows.

DEFINITION 18.10
Let $\rho \in (0, 1)$. We say that ρ-GAP 3SAT is **NP**-*hard* if for every language L there is a polynomial-time computable function f such that

$$x \in L \Rightarrow f(x) \text{ is a YES instance of } \rho\text{-GAP 3SAT}$$
$$x \notin L \Rightarrow f(x) \text{ is a NO instance of } \rho\text{-GAP 3SAT}$$

18.2.3   Constraint Satisfaction Problems

Now we generalize the definition of 3SAT to *constraint satisfaction problems* (CSP), which allow clauses of arbitrary form (instead of just OR of literals) including those depending upon more than 3 variables. Sometimes the variables are allowed to be non-Boolean. CSPs arise in a variety of application domains and play an important role in the proof of the **PCP** Theorem.

DEFINITION 18.11
Let q, W be natural numbers. A qCSP$_W$ instance φ is a collection of functions $\varphi_1, \ldots, \varphi_m$ (called *constraints*) from $\{0..W{-}1\}^n$ to $\{0, 1\}$ such that each function φ_i depends on at most q of its input locations. That is, for every $i \in [m]$ there exist $j_1, \ldots, j_q \in [n]$ and $f : \{0..W{-}1\}^q \to \{0, 1\}$ such that $\varphi_i(\mathbf{u}) = f(u_{j_1}, \ldots, u_{j_q})$ for every $\mathbf{u} \in \{0..W{-}1\}^n$.

We say that an *assignment* $\mathbf{u} \in \{0..W{-}1\}^n$ *satisfies* constraint φ_i if $\varphi_i(\mathbf{u}) = 1$. The fraction of constraints satisfied by $\mathbf{u}$ is $\frac{\sum_{i=1}^m \varphi_i(\mathbf{u})}{m}$, and we let $\mathsf{val}(\varphi)$ denote the maximum of this value over all $\mathbf{u} \in \{0..W{-}1\}^n$. We say that φ is *satisfiable* if $\mathsf{val}(\varphi) = 1$.

We call q the *arity* of φ and W the *alphabet size*. If $W = 2$ we say that φ uses a *binary* alphabet and call φ a qCSP-instance (dropping the subscript 2).

EXAMPLE 18.12
3SAT is the subcase of qCSP$_W$ where $q = 3$, $W = 2$, and the constraints are OR's of the involved literals.

Similarly, the **NP**-complete problem 3COL can be viewed as a subcase of 2CSP$_3$ instances where for each edge (i, j), there is a constraint on the variables u_i, u_j that is satisfied iff $u_i \neq u_j$. The graph is 3-colorable iff there is a way to assign a number in $\{0, 1, 2\}$ to each variable such that all constraints are satisfied.

Notes:

1. We define the *size* of a $q\mathsf{CSP}_W$-instance φ to be the number of constraints m it has. Because variables not used by any constraints are redundant, we always assume $n \leq qm$. Note that a $q\mathsf{CSP}_W$ instance over n variables with m constraints can be described using $O(mn^q W^q)$ bits. Usually q, W will be constants (independent of n, m).

2. As in the case of 3SAT, we can define maximization and gap problems for CSP instances. In particular, for any $\rho \in (0,1)$, we define ρ-GAP $q\mathsf{CSP}_W$ as the problem of distinguishing between a $q\mathsf{CSP}_W$-instance φ that is satisfiable (called a YES instance) and an instance φ with $\mathsf{val}(\varphi) \leq \rho$ (called a NO instance). As before, we will drop the subscript W in the case of a binary alphabet.

3. The simple greedy approximation algorithm for 3SAT can be generalized for the MAX $q\mathsf{CSP}$ problem of maximizing the number of satisfied constraints in a given $q\mathsf{CSP}$ instance. That is, for any $q\mathsf{CSP}_W$ instance φ with m constraints, the algorithm will output an assignment satisfying $\frac{\mathsf{val}(\varphi)}{W^q} m$ constraints. Thus, unless $\mathbf{NP} \subseteq \mathbf{P}$, the problem 2^{-q}-GAP $q\mathsf{CSP}$ is not $\mathbf{NP}$ hard.

18.2.4   An Alternative Formulation of the PCP Theorem

We now show how the **PCP** Theorem is equivalent to the **NP**-hardness of a certain gap version of $q\mathsf{CSP}$. Later, we will refer to this equivalence as the "hardness of approximation viewpoint" of the **PCP** Theorem.

THEOREM 18.13 (**PCP** THEOREM, ALTERNATIVE FORMULATION)
There exist constants $q \in \mathbb{N}$, $\rho \in (0,1)$ such that ρ-GAP $q\mathsf{CSP}$ is $\mathbf{NP}$-hard.

We now show Theorem 18.13 is indeed equivalent to the **PCP** Theorem:

Theorem 18.2 implies Theorem 18.13.   Assume that $\mathbf{NP} \subseteq \mathbf{PCP}(\log n, 1)$. We will show that $^1\!/_2$-GAP $q\mathsf{CSP}$ is $\mathbf{NP}$-hard for some constant q. It is enough to reduce a single $\mathbf{NP}$-complete language such as 3SAT to $^1\!/_2$-GAP $q\mathsf{CSP}$ for some constant q. Under our assumption, 3SAT has a **PCP** system in which the verifier V makes a constant number of queries, which we denote by q, and uses $c \log n$ random coins for some constant c. Given every input x and $r \in \{0,1\}^{c \log n}$, define $V_{x,r}$ to be the function that on input a proof π outputs 1 if the verifier will accept the proof π on input x and coins r. Note that $V_{x,r}$ depends on at most q locations. Thus for every $x \in \{0,1\}^n$, the collection $\varphi = \{V_{x,r}\}_{r \in \{0,1\}^{c \log n}}$ is a polynomial-sized $q\mathsf{CSP}$ instance. Furthermore, since V runs in polynomial-time, the transformation of x to φ can also be carried out in polynomial-time. By the completeness and soundness of the **PCP** system, if $x \in$ 3SAT then φ will satisfy $\mathsf{val}(\varphi) = 1$, while if $x \notin$ 3SAT then φ will satisfy $\mathsf{val}(\varphi) \leq {}^1\!/_2$. ∎

Theorem 18.13 implies Theorem 18.2.   Suppose that ρ-GAP $q\mathsf{CSP}$ is $\mathbf{NP}$-hard for some constants $q, \rho < 1$. Then this easily translates into a **PCP** system with q queries, ρ soundness and logarithmic randomness for any language L: given an input x, the verifier will run the reduction $f(x)$ to obtain a $q\mathsf{CSP}$ instance $\varphi = \{\varphi_i\}_{i=1}^m$. It will expect the proof π to be an assignment to the

variables of φ, which it will verify by choosing a random $i \in [m]$ and checking that φ_i is satisfied (by making q queries). Clearly, if $x \in L$ then the verifier will accept with probability 1, while if $x \notin L$ it will accept with probability at most ρ. The soundness can be boosted to $1/2$ at the expense of a constant factor in the randomness and number of queries (see Exercise 1). $\blacksquare$

REMARK 18.14
Since $3CNF$ formulas are a special case of 3CSP instances, Theorem 18.8 (ρ-GAP 3SAT is **NP**-hard) implies Theorem 18.13 (ρ-GAP qCSP is **NP**-hard). Below we show Theorem 18.8 is also implied by Theorem 18.13, concluding that it is also equivalent to the **PCP** Theorem.

It is worth while to review this very useful equivalence between the "proof view" and the "hardness of approximation view" of the **PCP** Theorem:

PCP verifier (V)	$\longleftrightarrow$	CSP instance (φ)
PCP proof (π)	$\longleftrightarrow$	Assignment to variables ($\mathbf{u}$)
Length of proof	$\longleftrightarrow$	Number of variables (n)
Number of queries (q)	$\longleftrightarrow$	Arity of constraints (q)
Number of random bits (r)	$\longleftrightarrow$	Logarithm of number of constraints ($\log m$)
Soundness parameter	$\longleftrightarrow$	Maximum of val(φ) for a NO instance
Theorem 18.2 (**NP** $\subseteq$ **PCP**$(\log n, 1)$)	$\longleftrightarrow$	Theorem 18.13 (ρ-GAP qCSP is **NP**-hard)

18.2.5 Hardness of Approximation for 3SAT and INDSET.

The CSP problem allows arbitrary functions to serve as constraints, which may seem somewhat artificial. We now show how Theorem 18.13 implies hardness of approximation results for the more natural problems of MAX 3SAT (determining the maximum number of clauses satisfiable in a 3SAT formula) and MAX INDSET (determining the size of the largest independent set in a given graph).

The following two lemmas use the **PCP** Theorem to show that unless $\mathbf{P} = \mathbf{NP}$, both MAX 3SAT and MAX INDSET are hard to approximate within a factor that is a constantless than 1. (Section 18.3 proves an even stronger hardness of approximation result for INDSET.)

LEMMA 18.15 (THEOREM 18.8, RESTATED)
*There exists a constant $0 < \rho < 1$ such that ρ-GAP 3SAT is **NP**-hard.*

LEMMA 18.16
There exist a polynomial-time computable transformation f from 3CNF formulae to graphs such that for every 3CNF formula φ, $f(\varphi)$ is an n-vertex graph whose largest independent set has size val$(\varphi)\frac{n}{7}$.

PROOF OF LEMMA 18.15: Let $\epsilon > 0$ and $q \in \mathbb{N}$ be such that by Theorem 18.13, $(1-\epsilon)$-GAP qCSP is **NP**-hard. We show a reduction from $(1-\epsilon)$-GAP qCSP to $(1-\epsilon')$-GAP 3SAT where $\epsilon' > 0$ is some constant depending on ϵ and q. That is, we will show a polynomial-time function mapping YES instances of $(1-\epsilon)$-GAP qCSP to YES instances of $(1-\epsilon')$-GAP 3SAT and NO instances of $(1-\epsilon)$-GAP qCSP to NO instances of $(1-\epsilon')$-GAP 3SAT.

Let φ be a qCSP instance over n variables with m constraints. Each constraint φ_i of φ can be expressed as an AND of at most 2^q clauses, where each clause is the OR of at most q variables

or their negations. Let φ' denote the collection of at most $m2^q$ clauses corresponding to all the constraints of φ. If φ is a YES instance of $(1-\epsilon)$-GAP qCSP (i.e., it is satisfiable) then there exists an assignment satisfying all the clauses of φ'. if φ is a NO instance of $(1-\epsilon)$-GAP qCSP then every assignment violates at least an ϵ fraction of the constraints of φ and hence violates at least an $\frac{\epsilon}{2^q}$ fraction of the constraints of φ. We can use the Cook-Levin technique of Chapter 2 (Theorem 2.10), to transform any clause C on q variables on $u_1, \ldots, u_q$ to a set $C_1, \ldots, C_q$ of clauses over the variables $u_1, \ldots, u_q$ and additional auxiliary variables $y_1, \ldots, y_q$ such that **(1)** each clause C_i is the OR of at most three variables or their negations, **(2)** if $u_1, \ldots, u_q$ satisfy C then there is an assignment to $y_1, \ldots, y_q$ such that $u_1, \ldots, u_q, y_1, \ldots, y_q$ simultaneously satisfy $C_1, \ldots, C_q$ and **(3)** if $u_1, \ldots, u_q$ does not satisfy C then for every assignment to $y_1, \ldots, y_q$, there is some clause C_i that is not satisfies by $u_1, \ldots, u_q, y_1, \ldots, y_q$.

Let φ'' denote the collection of at most $qm2^q$ clauses over the $n + qm$ variables obtained in this way from φ'. Note that φ'' is a 3SAT formula. Our reduction will map φ to φ''. Completeness holds since if φ was satisfiable then so will be φ' and hence φ''. Soundness holds since if every assignment violates at least an ϵ fraction of the constraints of φ, then every assignment violates at least an $\frac{\epsilon}{2^q}$ fraction of the constraints of φ', and so every assignment violates at least an $\frac{\epsilon}{q2^q}$ fraction of the constraints of φ''. $\blacksquare$

PROOF OF LEMMA 18.16: Let φ be a 3CNF formula on n variables with m clauses. We define a graph G of $7m$ vertices as follows: we associate a cluster of 7 vertices in G with each clause of φ. The vertices in cluster associated with a clause C correspond to the 7 possible assignments to the three variables C depends on (we call these *partial assignments*, since they only give values for some of the variables). For example, if C is $\overline{u_2} \vee \overline{u_5} \vee \overline{u_7}$ then the 7 vertices in the cluster associated with C correspond to all partial assignments of the form $u_1 = a, u_2 = b, u_3 = c$ for a binary vector $\langle a, b, c \rangle \neq \langle 1, 1, 1 \rangle$. (If C depends on less than three variables we treat one of them as repeated and then some of the 7 vertices will correspond to the same assignment.) We put an edge between two vertices of G if they correspond to *inconsistent* partial assignments. Two partial assignments are consistent if they give the same value to all the variables they share. For example, the assignment $u_1 = 1, u_2 = 0, u_3 = 0$ is inconsistent with the assignment $u_3 = 1, u_5 = 0, u_7 = 1$ because they share a variable (u_3) to which they give a different value. In addition, we put edges between every two vertices that are in the same cluster.

Clearly transforming φ into G can be done in polynomial time. Denote by $\alpha(G)$ to be the size of the largest independent set in G. We claim that $\alpha(G) = \mathsf{val}(\varphi)m$. For starters, note that $\alpha(G) \geq \mathsf{val}(\varphi)m$. Indeed, let $\mathbf{u}$ be the assignment that satisfies $\mathsf{val}(\varphi)m$ clauses. Define a set S as follows: for each clause C satisfied by $\mathbf{u}$, put in S the vertex in the cluster associated with C that corresponds to the restriction of $\mathbf{u}$ to the variables C depends on. Because we only choose vertices that correspond to restrictions of the assignment $\mathbf{u}$, no two vertices of S correspond to inconsistent assignments and hence S is an independent set of size $\mathsf{val}(\varphi)m$.

Suppose that G has an independent set S of size k. We will use S to construct an assignment $\mathbf{u}$ satisfying k clauses of φ, thus showing that $\mathsf{val}(\varphi)m \geq \alpha(G)$. We define $\mathbf{u}$ as follows: for every $i \in [n]$, if there is a vertex in S whose partial assignment gives a value a to u_i, then set $u_i = a$; otherwise set $u_i = 0$. This is well defined because S is an independent set, and each variable u_i can get at most a single value by assignments corresponding to vertices in S. On the other hand, because we put all the edges within each cluster, S can contain at most a single vertex in each

cluster, and hence there are k distinct cluster with members in S. By our definition of **u** it satisfies all the clauses associated with these clusters. ■

REMARK 18.17
In Chapter 2, we defined L' to be **NP**-hard if every $L \in \mathbf{NP}$ reduces to L'. The reduction was a polynomial-time function f such that $x \in L \Leftrightarrow f(x) \in L'$. In all cases, we proved that $x \in L \Rightarrow f(x) \in L'$ by showing a way to map a *certificate* to the fact that $x \in L$ to a certificate to the fact that $x' \in L'$. Although the definition of a Karp reduction does not require that this mapping is efficient, it often turned out that the proof did provide a way to compute this mapping in polynomial time. The way we proved that $f(x) \in L' \Rightarrow x \in L$ was by showing a way to map a certificate to the fact that $x' \in L'$ to a certificate to the fact that $x \in L$. Once again, the proofs typically yield an efficient way to compute this mapping.

A similar thing happens in the gap preserving reductions used in the proofs of Lemmas 18.15 and 18.16 and elsewhere in this chapter. When reducing from, say, ρ-GAP qCSP to ρ'-GAP 3SAT we show a function f that maps a CSP instance φ to a 3SAT instance ψ satisfying the following two properties:

Completeness We can map a satisfying assignment of φ to a satisfying assignment to ψ

Soundness Given any assignment that satisfies more than a ρ' fraction of ψ's clauses, we can map it back into an assignment satisfying more than a ρ fraction of φ's constraints.

18.3 $n^{-\delta}$-approximation of independent set is NP-hard.

We now show a much stronger hardness of approximation result for the independent set (INDSET) problem than Lemma 18.16. Namely, we show that there exists a constant $\delta \in (0,1)$ such that unless $\mathbf{P} = \mathbf{NP}$, there is no polynomial-time n^δ-approximation algorithm for INDSET. That is, we show that if there is a polynomial-time algorithm A that given an n-vertex graph G outputs an independent set of size at least $\frac{\mathsf{opt}}{n^\delta}$ (where opt is the size of the largest independent set in G) then $\mathbf{P} = \mathbf{NP}$. We note that an even stronger result is known: the constant δ can be made arbitrarily close to 1 [**?**, **?**]. This factor is almost optimal since the independent set problem has a trivial n-approximation algorithm: output a single vertex.

Our main tool will be the notion of *expander graphs* (see Note 18.18 and Chapter **??**). Expander graphs will also be used in the proof of **PCP** Theorem itself. We use here the following property of expanders:

LEMMA 18.19
Let $G = (V, E)$ be a λ-expander graph for some $\lambda \in (0,1)$. Let S be a subset of V with $|S| = \beta|V|$ for some $\beta \in (0,1)$. Let $(X_1, \ldots, X_\ell)$ be a tuple of random variables denoting the vertices of a uniformly chosen $(\ell-1)$-step path in G. Then,

$$(\beta - 2\lambda)^k \leq \Pr[\forall_{i \in [\ell]} X_i \in S] \leq (\beta + 2\lambda)^k$$

The upper bound of Lemma 18.19 is implied by Theorem **??**; we omit the proof of the lower bound.

The hardness result for independent set follows by combining the following lemma with Lemma 18.16:

NOTE 18.18 (EXPANDER GRAPHS)
Expander graphs are described in Chapter **??**. We define there a parameter
$\lambda(G) \in [0, 1]$, for every regular graph G (see Definition 7.25). The main
property we need in this chapter is that for every regular graph $G = (V, E)$
and every $S \subseteq V$ with $|S| \leq |V|/2$,

$$\Pr_{(u,v)\in E}[u \in S, v \in S] \leq \frac{|S|}{|V|}\left(\frac{1}{2} + \frac{\lambda(G)}{2}\right) \qquad (3)$$

Another property we use is that $\lambda(G^\ell) = \lambda(G)^\ell$ for every $\ell \in \mathbb{N}$, where G^ℓ is
obtained by taking the adjacency matrix of G to the ℓth power (i.e., an edge
in G^ℓ corresponds to an $(\ell-1)$-step path in G).
For every $c \in (0, 1)$, we call a regular graph G satisfying $\lambda(G) \leq c$ a c-
expander graph. If $c < 0.9$, we drop the prefix c and simply call G an
expander graph. (The choice of the constant 0.9 is arbitrary.) As shown
in Chapter **??**, for every constant $c \in (0, 1)$ there is a constant d and an
algorithm that given input $n \in N$, runs in poly(n) time and outputs the
adjacency matrix of an n-vertex d-regular c-expander (see Theorem 16.32).

LEMMA 18.20
*For every $\lambda > 0$ there is a polynomial-time computable reduction f that maps every n-vertex graph
G into an m-vertex graph H such that*

$$(\tilde{\alpha}(G) - 2\lambda)^{\log n} \leq \tilde{\alpha}(H) \leq (\tilde{\alpha}(G) + 2\lambda)^{\log n}$$

where $\tilde{\alpha}(G)$ is equal to the fractional size of the largest independent set in G.

Recall that Lemma 18.16 shows that there are some constants $\beta, \epsilon \in (0, 1)$ such that it is **NP**-
hard to tell whether a given graph G satisfies **(1)** $\tilde{\alpha}(G) \geq \beta$ or **(2)** $\tilde{\alpha}(G) \leq (1 - \epsilon)\beta$. By applying
to G the reduction of Lemma 18.20 with parameter $\lambda = \beta\epsilon/8$ we get that in case **(1)**, $\tilde{\alpha}(H) \geq$
$(\beta - \beta\epsilon/4)^{\log n} = (\beta(1 - \epsilon/4))^{\log n}$, and in case **(2)**, $\tilde{\alpha}(H) \leq ((1-\epsilon)\beta + \beta\epsilon/4)^{\log n} = (\beta(1 - 0.75\epsilon))^{\log n}$.
We get that the gap between the two cases is equal to $c^{\log n}$ for some $c > 1$ which is equal to m^δ
for some $\delta > 0$ (where $m = \text{poly}(n)$ is the number of vertices in H).

PROOF OF LEMMA 18.20:   Let G, λ be as in the lemma's statement. We let K be an n-vertex
λ-expander of degree d (we can obtain such a graph in polynomial-time, see Note 18.18). We will
map G into a graph H of $nd^{\log n-1}$ vertices in the following way:

- The vertices of H correspond to all the $(\log n-1)$-step paths in the λ-expander K.

- We put an edge between two vertices u, v of H corresponding to the paths $\langle u_1, \ldots, u_{\log n}\rangle$ and
  $\langle v_1, \ldots, v_{\log n}\rangle$ if there exists an edge in G between two vertices in the set $\{u_1, \ldots, u_{\log n}, v_1, \ldots, v_{\log n}\}$.

A subset T of H's vertices corresponds to a subset of $\log n$-tuples of numbers in $[n]$, which we can identify as tuples of vertices in G. We let $V(T)$ denote the set of all the vertices appearing in one of the tuples of T. Note that in this notation, T is an independent set in H if and only if $V(T)$ is an independent set of G. Thus for every independent set T in H, we have that $|V(T)| \leq \tilde{\alpha}(G)n$ and hence by the upper bound of Lemma 18.19, T takes up less than an $(\tilde{\alpha}(H) + 2\lambda)^{\log n}$ fraction of H's vertices. On the other hand, if we let S be the independent set of G of size $\tilde{\alpha}(G)n$ then by the lower bound of Lemma 18.19, an $(\tilde{\alpha} - 2\lambda)^{\log n}$ fraction of H's vertices correspond to paths fully contained in S, implying that $\tilde{\alpha}(H) \geq (\tilde{\alpha}(G) - 2\lambda)^{\log n}$. ∎

18.4 NP ⊆ PCP(poly(n), 1): PCP based upon Walsh-Hadamard code

We now prove a weaker version of the **PCP** theorem, showing that every **NP** statement has an exponentially-long proof that can be locally tested by only looking at a constant number of bits. In addition to giving a taste of how one proves **PCP** Theorems, this section builds up to a stronger Corollary 18.26, which will be used in the proof of the **PCP** theorem.

THEOREM 18.21
NP ⊆ **PCP**($poly(n)$, 1)

We prove this theorem by designing an appropriate verifier for an **NP**-complete language. The verifier expects the proof to contain an encoded version of the usual certificate. The verifier checks such an encoded certificate by simple probabilistic tests.

18.4.1 Tool: Linearity Testing and the Walsh-Hadamard Code

We use the *Walsh-Hadamard code* (see Section 17.5, though the treatment here is self-contained). It is a way to encode bit strings of length n by *linear functions* in n variables over GF(2); namely, the function $\mathsf{WH} : \{0,1\}^* \to \{0,1\}^*$ mapping a string $\mathbf{u} \in \{0,1\}^n$ to the truth table of the function $\mathbf{x} \mapsto \mathbf{u} \odot \mathbf{x}$, where for $\mathbf{x}, \mathbf{y} \in \{0,1\}^n$ we define $\mathbf{x} \odot \mathbf{y} = \sum_{i=1}^n x_i y_i \pmod 2$. Note that this is a very inefficient encoding method: an n-bit string $\mathbf{u} \in \{0,1\}^n$ is encoded using $|\mathsf{WH}(\mathbf{u})| = 2^n$ bits. If $f \in \{0,1\}^{2^n}$ is equal to $\mathsf{WH}(\mathbf{u})$ for some $\mathbf{u}$ then we say that f is a *Walsh-Hadamard codeword*. Such a string $f \in \{0,1\}^{2^n}$ can also be viewed as a function from $\{0,1\}^n$ to $\{0,1\}$.

The Walsh-Hadamard code is an error correcting code with minimum distance $1/2$, by which we mean that for every $\mathbf{u} \neq \mathbf{u}' \in \{0,1\}^n$, the encodings $\mathsf{WH}(\mathbf{u})$ and $\mathsf{WH}(\mathbf{u})$ differ in half the bits. This follows from the familiar random subsum principle (Claim A.5) since exactly half of the strings $\mathbf{x} \in \{0,1\}^n$ satisfy $\mathbf{u} \odot \mathbf{x} \neq \mathbf{u}' \odot \mathbf{x}$. Now we talk about local tests for the Walsh-Hadamard code (i.e., tests making only $O(1)$ queries).

Local testing of Walsh-Hadamard code. Suppose we are given access to a function $f : \{0,1\}^n \to \{0,1\}$ and want to *test* whether or not f is actually a codeword of Walsh-Hadamard. Since the Walsh-Hadamard codewords are precisely the set of all *linear* functions from $\{0,1\}^n$ to

$\{0,1\}$, we can test f by checking that

$$f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y}) \tag{4}$$

for all the 2^{2n} pairs $\mathbf{x}, \mathbf{y} \in \{0,1\}^n$ (where "+" on the left side of (pcp:eq:lintest) denotes vector addition over $\mathrm{GF}(2)^n$ and on the right side denotes addition over $\mathrm{GF}(2)$).

But can we test f by querying it in only a *constant* number of places? Clearly, if f is not linear but very close to being a linear function (e.g., if f is obtained by modifying a linear function on a very small fraction of its inputs) then such a *local* test will not be able to distinguish f from a linear function. Thus we set our goal on a test that on one hand accepts every linear function, and on the other hand rejects with high probability every function that is *far from linear*. It turns out that the natural test of choosing $\mathbf{x}, \mathbf{y}$ at random and verifying (4) achieves this goal:

DEFINITION 18.22
Let $\rho \in [0,1]$. We say that $f, g : \{0,1\}^n \to \{0,1\}$ are *ρ-close* if $\Pr_{\mathbf{x} \in_R \{0,1\}^n}[f(\mathbf{x}) = g(\mathbf{x})] \geq \rho$. We say that f is *ρ-close to a linear function* if there exists a linear function g such that f and g are ρ-close.

THEOREM 18.23 (LINEARITY TESTING [**?**])
Let $f : \{0,1\}^n \to \{0,1\}$ be such that

$$\Pr_{\mathbf{x}, \mathbf{y} \in_R \{0,1\}^n}[f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y})] \geq \rho$$

for some $\rho > 1/2$. Then f is ρ-close to a linear function.

We defer the proof of Theorem 18.23 to Section 19.3 of the next chapter. For every $\delta \in (0, 1/2)$, we can obtain a linearity test that rejects with probability at least $1/2$ every function that is not $(1-\delta)$-close to a linear function, by testing Condition (4) repeatedly $O(1/\delta)$ times with independent randomness. We call such a test a *$(1-\delta)$-linearity test*.

Local decoding of Walsh-Hadamard code. Suppose that for $\delta < \frac{1}{4}$ the function $f : \{0,1\}^n \to \{0,1\}$ is $(1-\delta)$-close to some linear function $\tilde{f}$. Because every two linear functions differ on half of their inputs, the function $\tilde{f}$ is uniquely determined by f. Suppose we are given $\mathbf{x} \in \{0,1\}^n$ and random access to f. Can we obtain the value $\tilde{f}(\mathbf{x})$ using only a constant number of queries? The naive answer is that since most $\mathbf{x}$'s satisfy $f(\mathbf{x}) = \tilde{f}(\mathbf{x})$, we should be able to learn $\tilde{f}(\mathbf{x})$ with good probability by making only the single query $\mathbf{x}$ to f. The problem is that $\mathbf{x}$ could very well be one of the places where f and $\tilde{f}$ differ. Fortunately, there is still a simple way to learn $\tilde{f}(\mathbf{x})$ while making only two queries to f:

1. Choose $\mathbf{x}' \in_R \{0,1\}^n$.

2. Set $\mathbf{x}'' = \mathbf{x} + \mathbf{x}'$.

3. Let $\mathbf{y}' = f(\mathbf{x}')$ and $\mathbf{y}'' = f(\mathbf{x}'')$.

4. Output $\mathbf{y}' + \mathbf{y}''$.

Since both $\mathbf{x}'$ and $\mathbf{x}''$ are individually uniformly distributed (even though they are dependent), by the union bound with probability at least $1 - 2\delta$ we have $\mathbf{y}' = \tilde{f}(\mathbf{x}')$ and $\mathbf{y}'' = \tilde{f}(\mathbf{x}'')$. Yet by the linearity of $\tilde{f}$, $\tilde{f}(\mathbf{x}) = \tilde{f}(\mathbf{x}' + \mathbf{x}'') = \tilde{f}(\mathbf{x}') + \tilde{f}(\mathbf{x}'')$, and hence with at least $1 - 2\delta$ probability $\tilde{f}(\mathbf{x}) = \mathbf{y}' + \mathbf{y}''$.[3] This technique is called *local decoding* of the Walsh-Hadamard code since it allows to recover any bit of the correct codeword (the linear function $\tilde{f}$) from a corrupted version (the function f) while making only a constant number of queries. It is also known as *self correction* of the Walsh-Hadamard code.

18.4.2 Proof of Theorem 18.21

We will show a (poly(n), 1)-verifier proof system for a particular **NP**-complete language L. The result that **NP** $\subseteq$ **PCP**(poly(n), 1) follows since every **NP** language is reducible to L. The **NP**-complete language L we use is QUADEQ, the language of systems of quadratic equations over $GF(2) = \{0, 1\}$ that are satisfiable.

EXAMPLE 18.24
The following is an instance of QUADEQ over the variables $u_1, \ldots, u_5$:

$$u_1 u_2 + u_3 u_4 + u_1 u_5 = 1$$
$$u_2 u_3 + u_1 u_4 = 0$$
$$u_1 u_4 + u_3 u_5 + u_3 u_4 = 1$$

This instance is satisfiable since the all-1 assignment satisfies all the equations.

More generally, an instance of QUADEQ over the variables $u_1, \ldots, u_n$ is of the form $AU = b$, where U is the n^2-dimensional vector whose $\langle i, j \rangle^{th}$ entry is $u_i u_j$, A is an $m \times n^2$ matrix and $\mathbf{b} \in \{0, 1\}^m$. In other words, U is the *tensor product* $\mathbf{u} \otimes \mathbf{u}$, where $\mathbf{x} \otimes \mathbf{y}$ for a pair of vectors $\mathbf{x}, \mathbf{y} \in \{0, 1\}^n$ denotes the n^2-dimensional vector (or $n \times n$ matrix) whose (i, j) entry is $x_i y_j$. For every $i, j \in [n]$ with $i \le j$, the entry $A_{k, \langle i, j \rangle}$ is the coefficient of $u_i u_j$ in the kth equation (we identify $[n^2]$ with $[n] \times [n]$ in some canonical way). The vector $\mathbf{b}$ consists of the right hand side of the m equations. Since $u_i = (u_i)^2$ in $GF(2)$, we can assume the equations do not contain terms of the form u_i^2.

Thus a satisfying assignment consists of $u_1, u_2, \ldots, u_n \in GF(2)$ such that its tensor product $U = u \otimes u$ satisfies $AU = b$. We leave it as Exercise 12 to show that QUADEQ, the language of all satisfiable instances, is indeed **NP**-complete.

We now describe the **PCP** system for QUADEQ. Let $A, \mathbf{b}$ be an instance of QUADEQ and suppose that $A, \mathbf{b}$ is satisfiable by an assignment $\mathbf{u} \in \{0, 1\}^n$. The correct **PCP** proof π for A, b will consist of the Walsh-Hadamard encoding for $\mathbf{u}$ and the Walsh-Hadamard encoding for $\mathbf{u} \otimes \mathbf{u}$, by which we mean that we will design the **PCP** verifier in a way ensuring that it accepts proofs

---

[3]We use here the fact that over $GF(2)$, $a + b = a - b$.
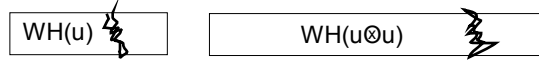
Figure 18.2: The **PCP** proof that a set of quadratic equations is satisfiable consists of WH(u) and WH($u \otimes u$) for some vector u. The verifier first checks that the proof is close to having this form, and then uses the local decoder of the Walsh-Hadamard code to ensure that u is a solution for the quadratic equation instance.

of this form with probability one, satisfying the completeness condition. (Note that π is of length $2^n + 2^{n^2}$.)

Below, we repeatedly use the following fact:

RANDOM SUBSUM PRINCIPLE: *If $\mathbf{u} \neq \mathbf{v}$ then for at least $1/2$ the choices of $\mathbf{x}$, $\mathbf{u} \odot \mathbf{x} \neq \mathbf{v} \odot \mathbf{x}$. Realize that $\mathbf{x}$ can be viewed as a random subset of indices in $[1, \ldots, n]$ and the principle says that with probability $1/2$ the sum of the u_i's over this index set is different from the corresponding sum of v_i's.*

The verifier.   The verifier V gets access to a proof $\pi \in \{0, 1\}^{2^n + 2^{n^2}}$, which we interpret as a pair of functions $f : \{0, 1\}^n \to \{0, 1\}$ and $g : \{0, 1\}^{n^2} \to \{0, 1\}$.

Step 1: Check that f, g are linear functions.

As already noted, this isn't something that the verifier can check per se using local tests. Instead, the verifier performs a 0.99-linearity test on both f, g, and rejects the proof at once if either test fails.

Thus, if either of f, g is not 0.99-close to a linear function, then V rejects with high probability. Therefore for the rest of the procedure we can assume that there exist two linear functions $\tilde{f} : \{0, 1\}^n \to \{0, 1\}$ and $\tilde{g} : \{0, 1\}^{n^2} \to \{0, 1\}$ such that $\tilde{f}$ is 0.99-close to f, and $\tilde{g}$ is 0.99-close to g. (Note: in a correct proof, the tests succeed with probability 1 and $\tilde{f} = f$ and $\tilde{g} = g$.)

In fact, we will assume that for Steps 2 and 3, the verifier can query $\tilde{f}, \tilde{g}$ at any desired point. The reason is that local decoding allows the verifier to recover any desired value of $\tilde{f}, \tilde{g}$ with good probability, and Steps 2 and 3 will only use a small (less than 15) number of queries to $\tilde{f}, \tilde{g}$. Thus with high probability (say > 0.9) local decoding will succeed on all these queries.

NOTATION: To simplify notation in the rest of the procedure we use f, g for $\tilde{f}, \tilde{g}$ respectively. Furthermore, we assume both f and g are linear, and thus they must encode some strings $\mathbf{u} \in \{0, 1\}^n$ and $\mathbf{w} \in \{0, 1\}^{n^2}$. In other words, f, g are the functions given by $f(\mathbf{r}) = \mathbf{u} \odot \mathbf{r}$ and $g(\mathbf{z}) = \mathbf{w} \odot \mathbf{z}$.

Step 2: Verify that g encodes $\mathbf{u} \otimes \mathbf{u}$, where $\mathbf{u} \in \{0, 1\}^n$ is the string encoded by f.

Verifier V does the following test 3 times: "Choose $\mathbf{r}, \mathbf{r}'$ independently at random from $\{0, 1\}^n$, and if $f(\mathbf{r})f(\mathbf{r}') \neq g(\mathbf{r} \otimes \mathbf{r}')$ then halt and reject."

In a correct proof, $w = \mathbf{u} \otimes \mathbf{u}$, so

$$f(\mathbf{r})f(\mathbf{r}') = \left(\sum_{i \in [n]} u_i r_i \right) \left(\sum_{j \in [n]} u_j r_j' \right) =$$

$$\sum_{i, j \in [n]} u_i u_j r_i r_j' = (\mathbf{u} \otimes \mathbf{u}) \odot (\mathbf{r} \otimes \mathbf{r}'),$$

which in the correct proof is equal to $g(\mathbf{r} \otimes \mathbf{r}')$. Thus Step 2 never rejects a correct proof.

Suppose now that, unlike the case of the correct proof, $\mathbf{w} \neq \mathbf{u} \otimes \mathbf{u}$. We claim that in each of the three trials V will halt and reject with probability at least $\frac{1}{4}$. (Thus the probability of rejecting in at least one trial is at least $1 - (3/4)^3 = 37/64$.) Indeed, let W be an $n \times n$ matrix with the same entries as $\mathbf{w}$, let U be the $n \times n$ matrix such that $U_{i,j} = u_i u_j$ and think of $\mathbf{r}$ as a row vector and $\mathbf{r}'$ as a column vector. In this notation,

$$g(\mathbf{r} \otimes \mathbf{r}') = \mathbf{w} \odot (\mathbf{r} \otimes \mathbf{r}') = \sum_{i,j \in [n]} w_{i,j} r_i r_j' = \mathbf{r} W \mathbf{r}'$$

$$f(\mathbf{r})f(\mathbf{r}') = (\mathbf{u} \odot \mathbf{r})(\mathbf{u} \odot \mathbf{r}') = (\sum_{i=1}^n u_i r_i)(\sum_{j=1}^n u_j r_j') = \sum_{i,j \in [n]} u_i u_j r_i r_j = \mathbf{r} U \mathbf{r}'$$

And V rejects if $\mathbf{r} W \mathbf{r}' \neq \mathbf{r} U \mathbf{r}'$. The random subsum principle implies that if $W \neq U$ then at least $1/2$ of all $\mathbf{r}$ satisfy $\mathbf{r}W \neq \mathbf{r}U$. Applying the random subsum principle for each such $\mathbf{r}$, we conclude that at least $1/2$ the $\mathbf{r}'$ satisfy $\mathbf{r}W\mathbf{r}' \neq \mathbf{r}U\mathbf{r}'$. We conclude that the test rejects for at least $1/4$ of all pairs $\mathbf{r}, \mathbf{r}'$.

Step 3: Verify that f encodes a satisfying assignment.

Using all that has been verified about f, g in the previous two steps, it is easy to check that any particular equation, say the kth equation of the input, is satisfied by $\mathbf{u}$, namely,

$$\sum_{i,j} A_{k,(i,j)} u_i u_j = b_k. \tag{5}$$

Denoting by z the n^2 dimensional vector $(A_{k,(i,j)})$ (where i, j vary over $[1..n]$), we see that the left hand side is nothing but $g(\mathbf{z})$. Since the verifier knows $A_{k,(i,j)}$ and b_k, it simply queries g at $\mathbf{z}$ and checks that $g(\mathbf{z}) = b_k$.

The drawback of the above idea is that in order to check that $\mathbf{u}$ satisfies the entire system, the verifier needs to make a query to g for each $k = 1, 2, \ldots, m$, whereas the number of queries is required to be independent of m. Luckily, we can use the random subsum principle again! The verifier takes a random subset of the equations and computes their sum mod 2. (In other words, for $k = 1, 2, \ldots, m$ multiply the equation in (5) by a random bit and take the sum.) This sum is a new quadratic equation, and the random subsum principle implies that if $\mathbf{u}$ does not satisfy even one equation in the original system, then with probability at least $1/2$ it will not satisfy this new equation. The verifier checks that $\mathbf{u}$ satisfies this new equation.

(Actually, the above test has to be repeated twice to ensure that if $\mathbf{u}$ does not satisfy the system, then Step 3 rejects with probability at least $3/4$.)

18.4.3 PCP's of proximity

Theorem 18.21 says that (exponential-sized) certificates for **NP** languages can be checked by examining only $O(1)$ bits in them. The proof actually yields a somewhat stronger result, which will be used in the proof of the **PCP** Theorem. This concerns the following scenario: we hold a circuit C in our hands that has n input wires. Somebody holds a satisfying assignment u. He writes down $\mathsf{WH}(u)$ as well as another string π for us. We do a probabilistic test on this by examining $O(1)$ bits in these strings, and at the end we are convinced of this fact.

Concatenation test.   First we need to point out a property of Walsh-Hadamard codes and a related *concatenation test*. In this setting, we are given two linear functions f, g that encode strings of lengths n and $n + m$ respectively. We have to check by examining only $O(1)$ bits in f, g that if $\mathbf{u}$ and $\mathbf{v}$ are the strings encoded by f, g (that is, $f = \mathsf{WH}(\mathbf{u})$ and $h = \mathsf{WH}(\mathbf{v})$) then $\mathbf{u}$ is the same as the first n bits of $\mathbf{v}$. By the random subsum principle, the following simple test rejects with probability $1/2$ if this is not the case. Pick a random $\mathbf{x} \in \{0,1\}^n$, and denote by $\mathbf{X} \in \mathrm{GF}(2)^{m+n}$ the string whose first n bits are $\mathbf{x}$ and the remaining bits are all-0. Verify that $f(X) = g(\mathbf{x})$.

With this test in hand, we can prove the following corollary.

COROLLARY 18.25 (EXPONENTIAL-SIZED **PCP** OF PROXIMITY.)
There exists a verifier V that given any circuit C of size m and with n inputs has the following property:

1. *If $\mathbf{u} \in \{0,1\}^n$ is a satisfying assignment for circuit C, then there is a string π_2 of size $2^{\mathrm{poly}(m)}$ such that V accepts $\mathsf{WH}(\mathbf{u}) \circ \pi_2$ with probability 1. (Here $\circ$ denotes concatenation.)*

2. *For every strings $\pi_1, \pi_2 \in \{0,1\}^*$, where π_1 has 2^n bits, if V accepts $\pi_1 \circ \pi_2$ with probability at least $1/2$, then π_1 is 0.99-close to $\mathsf{WH}(\mathbf{u})$ for some $\mathbf{u}$ that satisfies C.*

3. *V uses $\mathrm{poly}(m)$ random bits and examines only $O(1)$ bits in the provided strings.*

PROOF: One looks at the proof of **NP**-completeness of QUADEQ to realize that given circuit C with n input wires and size m, it yields an instance of QUADEQ of size $O(m)$ such that $\mathbf{u} \in \{0,1\}^n$ satisfies the circuit iff there is a string $\mathbf{v}$ of size $M = O(m)$ such that $\mathbf{u} \circ \mathbf{v}$ satisfies the instance of QUADEQ. (Note that we are thinking of $\mathbf{u}$ both as a string of bits that is an input to C and as a string over $\mathrm{GF}(2)^n$ that is a partial assignment to the variables in the instance of QUADEQ.)

The verifier expects π_2 to contain whatever our verifier of Theorem 18.21 expects in the proof for this instance of QUADEQ, namely, a linear function f that is $\mathsf{WH}(w)$, and another linear function g that is $\mathsf{WH}(w \otimes w)$ where w satisfies the QUADEQ instance. The verifier checks these functions as described in the proof of Theorem 18.21.

However, in the current setting our verifer is also given a string $\pi_1 \in \{0,1\}^{2^n}$. Think of this as a function $h \colon \mathrm{GF}(2)^n \to \mathrm{GF}(2)$. The verifier checks that h is 0.99-close to a linear function, say $\tilde{h}$. Then to check that $\tilde{f}$ encodes a string whose first n bits are the same as the string encoded by $\tilde{h}$, the verifier does a concatenation test.

Clearly, the verifier only reads $O(1)$ bits overall. ∎

The following Corollary is also similarly proven and is the one that will actually be used later. It concerns a similar situation as above, except the inputs to the circuit C are thought of as the concatenation of two strings of lengths n_1, n_2 respectively where $n = n_1 + n_2$.

COROLLARY 18.26 (**PCP** OF PROXIMITY WHEN ASSIGNMENT IS IN TWO PIECES)
There exists a verifier V that given any circuit C with n input wires and size m and also two numbers n_1, n_2 such that $n_1 + n_2 = n$ has the following property:

1. *If $\mathbf{u_1} \in \{0,1\}^{n_1}, \mathbf{u_2} \in \{0,1\}^{n_2}$ is such that $\mathbf{u_1} \circ \mathbf{u_2}$ is a satisfying assignment for circuit C, then there is a string π_3 of size $2^{\mathrm{poly}(m)}$ such that V accepts $\mathsf{WH}(\mathbf{u_1}) \circ \mathsf{WH}(\mathbf{u_2}) \circ \pi_3$ with probability 1.*

2. *For every strings $\pi_1, \pi_2, \pi_3 \in \{0,1\}^*$, where π_1 and π_2 have 2^{n_1} and 2^{n_2} bits respectively, if V accepts $\pi_1 \circ \pi_2 \circ \pi_3$ with probability at least $1/2$, then π_1, π_2 are 0.99-close to $\mathsf{WH}(\mathbf{u_1})$, $\mathsf{WH}(\mathbf{u_2})$ respectively for some $\mathbf{u_1}, \mathbf{u_2}$ such that $\mathbf{u_1} \circ \mathbf{u_2}$ is a satisfying assignment for circuit C.*

3. *V uses poly(m) random bits and examines only $O(1)$ bits in the provided strings.*

18.5 Proof of the PCP Theorem.

As we have seen, the **PCP** Theorem is equivalent to Theorem 18.13, stating that ρ-GAP qCSP is **NP**-hard for some constants q and $\rho < 1$. Consider the case that $\rho = 1 - \epsilon$ where ϵ is not necessarily a constant but can be a function of m (the number of constraints). Since the number of satisfied constraints is always a whole number, if φ is unsatisfiable then $\mathsf{val}(\varphi) \le 1 - 1/m$. Hence, the gap problem $(1-1/m)$-GAP 3CSP is a generalization of 3SAT and is **NP** hard. The idea behind the proof is to start with this observation, and iteratively show that $(1-\epsilon)$-GAP qCSP is **NP**-hard for larger and larger values of ϵ, until ϵ is as large as some absolute constant independent of m. This is formalized in the following lemma.

DEFINITION 18.27
Let f be a function mapping CSP instances to CSP instances. We say that f is a *CL-reduction* (short for *complete linear-blowup reduction*) if it is polynomial-time computable and for every CSP instance φ with m constraints, satisfies:

Completeness: If φ is satisfiable then so is $f(\varphi)$.

Linear blowup: The new qCSP instance $f(\varphi)$ has at most Cm constraints and alphabet W, where C and W can depend on the arity and the alphabet size of φ (but not on the number of constraints or variables).

LEMMA 18.28 (**PCP** MAIN LEMMA)
There exist constants $q_0 \ge 3$, $\epsilon_0 > 0$, and a CL-reduction f such that for every q_0CSP-instance φ with binary alphabet, and every $\epsilon < \epsilon_0$, the instance $\psi = f(\varphi)$ is a q_0CSP (over binary alphabet) satisfying

$$\mathsf{val}(\varphi) \le 1 - \epsilon \Rightarrow \mathsf{val}(\psi) \le 1 - 2\epsilon$$

Lemma 18.28 can be succinctly described as follows:

	Arity	Alphabet	Constraints	Value
Original	q_0	binary	m	$1 - \epsilon$
	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
Lemma 18.28	q_0	binary	Cm	$1 - 2\epsilon$

This Lemma allows us to easily prove the **PCP** Theorem.

Proving Theorem 18.2 from Lemma 18.28.   Let $q_0 \geq 3$ be as stated in Lemma 18.28. As already observed, the decision problem q_0CSP is **NP**-hard. To prove the **PCP** Theorem we give a reduction from this problem to GAP q_0CSP. Let φ be a q_0CSP instance. Let m be the number of constraints in φ. If φ is satisfiable then $\mathsf{val}(\varphi) = 1$ and otherwise $\mathsf{val}(\varphi) \leq 1 - 1/m$. We use Lemma 18.28 to amplify this gap. Specifically, apply the function f obtained by Lemma 18.28 to φ a total of $\log m$ times. We get an instance ψ such that if φ is satisfiable then so is ψ, but if φ is not satisfiable (and so $\mathsf{val}(\varphi) \leq 1 - 1/m$) then $\mathsf{val}(\psi) \leq 1 - \min\{2\epsilon_0, 1 - 2^{\log m}/m\} = 1 - 2\epsilon_0$. Note that the size of ψ is at most $C^{\log m}m$, which is polynomial in m. Thus we have obtained a gap-preserving reduction from L to the $(1-2\epsilon_0)$-GAP q_0CSP problem, and the **PCP** theorem is proved. ∎

   The rest of this section proves Lemma 18.28 by combining two transformations: the first transformation amplifies the gap (i.e., fraction of violated constraints) of a given CSP instance, at the expense of increasing the alphabet size. The second transformation reduces back the alphabet to binary, at the expense of a modest reduction in the gap. The transformations are described in the next two lemmas.

LEMMA 18.29 (GAP AMPLIFICATION [**?**])
For every $\ell \in \mathbb{N}$, there exists a CL-reduction g_ℓ such that for every CSP instance φ with binary alphabet, the instance $\psi = g_\ell(\varphi)$ has has arity only 2 (but over a non-binary alphabet) and satisfies:

$$\mathsf{val}(\varphi) \leq 1 - \epsilon \Rightarrow \mathsf{val}(\psi) \leq 1 - \ell\epsilon$$

for every $\epsilon < \epsilon_0$ where $\epsilon_0 > 0$ is a number depending only on ℓ and the arity q of the original instance φ.

LEMMA 18.30 (ALPHABET REDUCTION)
There exists a constant q_0 and a CL- reduction h such that for every CSP instance φ, if φ had arity two over a (possibly non-binary) alphabet $\{0..W{-}1\}$ then $\psi = h(\varphi)$ has arity q_0 over a binary alphabet and satisfies:

$$\mathsf{val}(\varphi) \leq 1 - \epsilon \Rightarrow \mathsf{val}(h(\varphi)) \leq 1 - \epsilon/3$$

   Lemmas 18.29 and 18.30 together imply Lemma 18.28 by setting $f(\varphi) = h(g_6(\varphi))$. Indeed, if φ was satisfiable then so will $f(\varphi)$. If $\mathsf{val}(\varphi) \leq 1 - \epsilon$, for $\epsilon < \epsilon_0$ (where ϵ_0 the value obtained in Lemma 18.29 for $\ell = 6$, $q = q_0$) then $\mathsf{val}(g_6(\varphi)) \leq 1 - 6\epsilon$ and hence $\mathsf{val}(h(g_6(\varphi))) \leq 1 - 2\epsilon$. This composition is described in the following table:

	Arity	Alphabet	Constraints	Value
Original	q_0	binary	m	$1 - \epsilon$
	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
Lemma 18.29	2	W	Cm	$1 - 6\epsilon$
	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
Lemma 18.30	q_0	binary	$C'Cm$	$1 - 2\epsilon$

18.5.1 Gap Amplification: Proof of Lemma 18.29

To prove Lemma 18.29, we need to exhibit a function g that maps a $q\mathsf{CSP}$ instance to a $2\mathsf{CSP}_W$ instance over a larger alphabet $\{0..W–1\}$ in a way that increases the fraction of violated constraints.

We will show that we may assume without loss of generality that the instance of $q\mathsf{CSP}$ has a specific form. To describe this we need a definition.

We will assume that the instance satisfies the following properties, since we can give a simple CL-reduction from $q\mathsf{CSP}$ to this special type of $q\mathsf{CSP}$. (See the "Technical Notes" section at the end of the chapter.) We will call such instances "nice."

Property 1: The arity q is 2 (though the alphabet may be nonbinary).

Property 2: Let the *constraint graph* of ψ be the graph G with vertex set $[n]$ where for every constraint of φ depending on the variables u_i, u_j, the graph G has the edge (i, j). We allow G to have parallel edges and self-loops. Then G is d-regular for some constant d (independent of the alphabet size) and at every node, half the edges incident to it are self-loops.

Property 3: The constraint graph is an expander.

The rest of the proof consists of a "powering" operation for nice $2\mathsf{CSP}$ instances. This is described in the following Lemma.

LEMMA 18.31 (POWERING)
Let ψ be a $2\mathsf{CSP}_W$ instance satisfying Properties 1 through 3. For every number t, there is an instance of $2\mathsf{CSP}$ ψ^t such that:

1. ψ^t is a $2\mathsf{CSP}_{W'}$-instance with alphabet size $W' < W^{d^{5t}}$, where d denote the degree of ψ's constraint graph. The instance ψ^t has $d^{t+\sqrt{t}}n$ constraints, where n is the number of variables in ψ.

2. If ψ is satisfiable then so is ψ^t.

3. For every $\epsilon < \frac{1}{d\sqrt{t}}$, if $\mathsf{val}(\psi) \leq 1 - \epsilon$ then $\mathsf{val}(\psi^t) \leq 1 - \epsilon'$ for $\epsilon' = \frac{\sqrt{t}}{10^5 dW^4}\epsilon$.

4. The formula ψ^t is computable from ψ in time polynomial in m and W^{d^t}.

PROOF: Let ψ be a $2\mathsf{CSP}_W$-instance with n variables and $m = nd$ constraints, and as before let G denote the *constraint graph* of ψ.

The formula ψ^t will have the same number of variables as ψ. The new variables $\mathbf{y} = y_1, \ldots, y_n$ take values over an alphabet of size $W' = W^{d^{5t}}$, and thus a value of a new variable y_i is a d^{5t}-tuple of values in $\{0..W–1\}$. We will think of this tuple as giving a value in $\{0..W–1\}$ to every old variable u_j where j can be reached from u_i using a path of at most $t + \sqrt{t}$ steps in G (see Figure 18.3). In other words, the tuple contains an assignment for every u_j such that j is in the ball of radius $t + \sqrt{t}$ and center i in G. For this reason, we will often think of an assignment to y_i as "claiming" a certain value for u_j. (Of course, another variable y_k could claim a different value for u_j.) Note that since G has degree d, the size of each such ball is no more than $d^{t+\sqrt{t}+1}$ and hence this information can indeed be encoded using an alphabet of size W'.
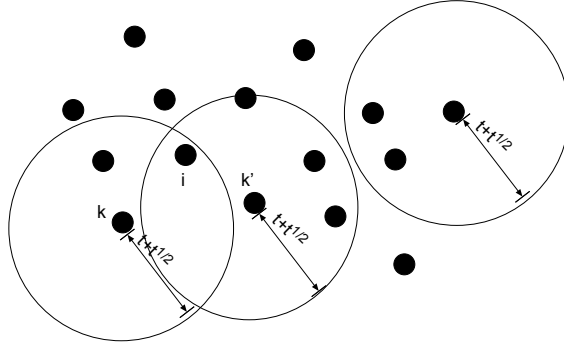
Figure 18.3: An assignment to the formula ψ^t consists of n variables over an alphabet of size less than $W^{d^{5t}}$, where each variable encodes the restriction of an assignment of ψ to the variables that are in some ball of radius $t + \sqrt{t}$ in ψ's constraint graph. Note that an assignment $y_1, \ldots, y_n$ to ψ^t may be *inconsistent* in the sense that if i falls in the intersection of two such balls centered at k and k', then y_k may claim a different value for u_i than the value claimed by $y_{k'}$.

For every $(2t+1)$-step path $p = \langle i_1, \ldots, i_{2t+2} \rangle$ in G, we have one corresponding constraint C_p in ψ^t (see Figure 18.4). The constraint C_p depends on the variables y_{i_1} and $y_{i_{2t+1}}$ and outputs FALSE if (and only if) there is some $j \in [2t+1]$ such that:

1. i_j is in the $t + \sqrt{t}$-radius ball around i_1.

2. i_{j+1} is in the $t + \sqrt{t}$-radius ball around i_{2t+2}

3. If w denotes the value y_{i_1} claims for u_{i_j} and w' denotes the value $y_{i_{2t+2}}$ claims for $u_{i_{j+1}}$, then the pair (w, w') violates the constraint in φ that depends on u_{i_j} and $u_{i_{j+1}}$.
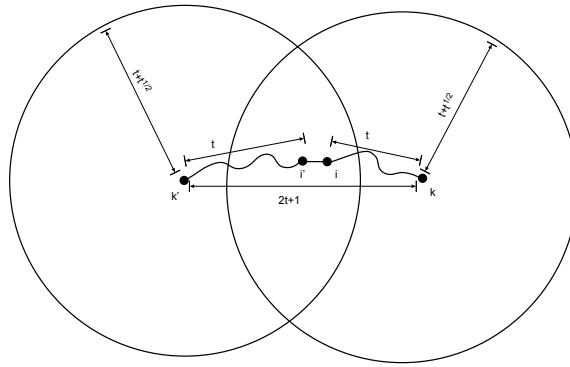
Figure 18.4: ψ^t has one constraint for every path of length $2t + 1$ in ψ's constraint graph, checking that the views of the balls centered on the path's two endpoints are consistent with one another and the constraints of ψ.

A few observations are in order. First, the time to produce such an assignment is polynomial in m and W^{d^t}, so part 4 of Lemma 18.29 is trivial.

Second, for every assignment to $u_1, u_2, \ldots, u_n$ we can "lift" it to a *canonical* assignment to $y_1, \ldots, y_n$ by simply assigning to each y_i the vector of values assumed by u_j's that lie in a ball of radius $t + \sqrt{t}$ and center i in G. If the assignment to the u_j's was a satisfying assignment for ψ, then this canonical assignment satisfies ψ^t, since it will satisfy all constraints encountered in walks of length $2t + 1$ in G. Thus part 2 of Lemma 18.29 is also trivial.

This leaves part 3 of the Lemma, the most difficult part. We have to show that if $\mathsf{val}(\psi) \leq 1 - \epsilon$ then every assignment to the y_i's satisfies at most $1 - \epsilon'$ fraction of constraints in ψ^t, where $\epsilon < \frac{1}{d\sqrt{t}}$ and $\epsilon' = \frac{\sqrt{t}}{10^5 dW^4}\epsilon$. This is tricky since an assignment to the y_i's does not correspond to any obvious assignment for the u_i's: for each u_j, different values could be claimed for it by different y_i's. The intuition will be to show that these *inconsistencies* among the y_i's can't happen too often (at least if the assignment to the y_i's satisfies $1 - \epsilon'$ constraints in ψ^t).

From now on, let us fix some arbitrary assignment $\mathbf{y} = y_1, \ldots, y_n$ to ψ^t's variables. The following notion is key.

The plurality assignment: For every variable u_i of ψ, we define the random variable Z_i over $\{0, \ldots, W - 1\}$ to be the result of the following process: starting from the vertex i, take a t step random walk in G to reach a vertex k, and output the value that y_k claims for u_i. We let z_i denote the *plurality* (i.e., most likely) value of Z_i. If more than one value is most likely, we break ties arbitrarily. This assignment is called a *plurality assignment* (see Figure 18.5). Note that $Z_i = z_i$ with probability at least $1/W$.
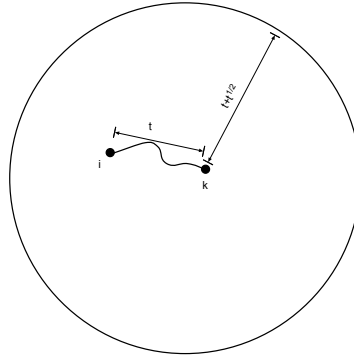
Figure 18.5: An assignment y for ψ^t induces a plurality assignment u for ψ in the following way: u_i gets the most likely value that is claimed for it by y_k, where k is obtained by taking a t-step random walk from i in the constraint graph of ψ.

Since $\mathsf{val}(\psi) \leq 1 - \epsilon$, *every* assignment for ψ fails to satisfy $1 - \epsilon$ fraction of the constraints, and this is therefore also true for the plurality assignment. Hence there exists a set F of $\epsilon m = \epsilon n d$ constraints in ψ that are violated by the assignment $\mathbf{z} = z_1, \ldots, z_n$. We will use this set F to show that at least an ϵ' fraction of ψ^t's constraints are violated by the assignment $\mathbf{y}$.

Why did we define the plurality assignment $\mathbf{z}$ in this way? The reason is illustrated by the following claim, showing that for every edge $f = (i, i')$ of G, among all paths that contain the edge f somewhere in their "midsection", most paths are such that the endpoints of the path claim the plurality values for u_i and $u_{i'}$.

CLAIM 18.32

For every edge $f = (i, i')$ in G define the event $B_{j,f}$ over the set of $(2t+1)$-step paths in G to contain all paths $\langle i_1, \ldots, i_{2t+2} \rangle$ satisfying:

- *f is the jth edge in the path. That is, $f = (i_j, i_{j+1})$.*

- *y_{i_1} claims the plurality value for u_i.*

- *$y_{i_{2t+2}}$ claims the plurality value for $u_{i'}$.*

Let $\delta = \frac{1}{100W^2}$. Then for every $j \in \{t, \ldots, t + \delta\sqrt{t}\}$, $\Pr[B_{j,f}] \geq \frac{1}{nd2W^2}$.

PROOF: First, note that because G is regular, the jth edge of a random path is a random edge, and hence the probability that f is the jth edge on the path is equal to $\frac{1}{nd}$. Thus, we need to prove that,

$$\Pr[\text{endpoints claim plurality values for } u_i, u_{i'} \text{ (resp.)}|f \text{ is } j^{th} \text{ edge}] \geq \frac{1}{2W^2} \qquad (6)$$

We start with the case $j = t + 1$. In this case (6) holds essentially by definition: the left-hand side of (6) is equal to the probability that the event that the endpoints claim the plurality for these variables happens for a path obtained by joining a random t-step path from i to a random t-step path from i'. Let k be the endpoint of the first path and k' be the endpoint of the second path. Let W_i be the distribution of the value that y_k claims for u_i, where k is chosen as above, and similarly define $W_{i'}$ to be the distribution of the value that $y_{k'}$ claims for $u_{i'}$. Note that since k and k' are chosen independently, the random variables W_i and $W_{i'}$ are independent. Yet by definition the distribution of W_i identical to the distribution Z_i, while the distribution of $W_{i'}$ is identical to $Z_{i'}$. Thus,

$$\Pr[\text{endpoints claim plurality values for } u_i, u_{i'} \text{ (resp.)}|f \text{ is } j^{th} \text{ edge}] =$$
$$\Pr_{k,k'}[W_i = z_i \wedge W_{i'} = z_{i'}] = \Pr_k[W_i = z_i] \Pr_{k'}[W_{i'} = z_{i'}] \geq \frac{1}{W^2}$$

In the case that $j \neq 2t+1$ we need to consider the probability of the event that endpoints claim the plurality values happening for a path obtained by joining a random $t-1+j$-step path from i to a random $t+1-j$-step path from i' (see Figure 18.6). Again we denote by k the endpoint of the first path, and by k' the endpoint of the second path, by W_i the value y_k claims for u_i and by $W_{i'}$ the value $y_{k'}$ claims for $u_{i'}$. As before, W_i and $W_{i'}$ are independent. However, this time W_i and Z_i may not be identically distributed. Fortunately, we can show that they are almost identically distributed, in other words, the distributions are *statistically close*. Specifically, because half of the constraints involving each variable are self loops, we can think of a t-step random walk from a vertex i as follows: **(1)** throw t coins and let S_t denote the number of the coins that came up "heads" **(2)** take S_t "real" (non self-loop) steps on the graph. Note that the endpoint of a t-step random walk and a t'-step random walk will be identically distributed if in Step **(1)** the variables S_t and $S_{t'}$ turn out to be the same number. Thus, the statistical distance of the endpoint of a t-step random walk and a t'-step random walk is bounded by the statistical distance of S_t and $S_{t'}$ where S_ℓ denotes the binomial distribution of the sum of ℓ balanced independent coins.

However, the distributions S_t and $S_{t+\delta\sqrt{t}}$ are within statistical distance at most 10δ for every δ, t (see Exercise 15) and hence in our case W_i and $W_{i'}$ are $\frac{1}{10W}$-close to Z_i and $Z_{i'}$ respectively. Thus $|\Pr_k[W_i = z_i] - \Pr[Z_i = z_i]| < \frac{1}{10W}$, $|\Pr_k[W_{i'} = z_{i'}] - \Pr[Z_{i'} = z_{i'}]| < \frac{1}{10W}$ which proves (6) also for the case $j \neq 2t + 1$. ∎
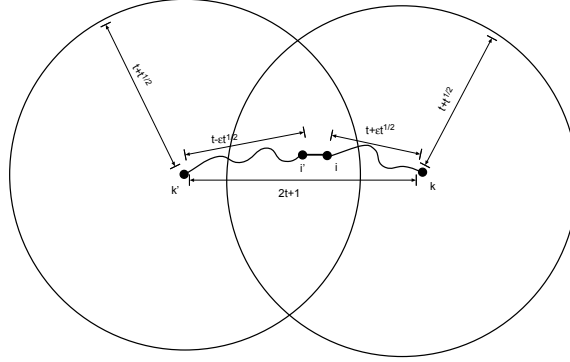
Figure 18.6: By definition, if we take two t-step random walks from two neighbors i and i', then the respective endpoints will claim the plurality assignments for u_i and u_j with probability more than $1/(2W^2)$. Because half the edges of every vertex in G have self loops, this happens even if the walks are not of length t but of length in $[t - \epsilon\sqrt{t}, t + \sqrt{t}]$ for sufficiently small ϵ.

Recall that F is the set of constraints of ψ (=edges in G) violated by the plurality assignment **z**. Therefore, if $f \in F$ and $j \in \{t, \ldots, t + \delta\sqrt{t}\}$ then all the paths in $B_{j,f}$ correspond to constraints of ψ^t that are violated by the assignment **y**. Therefore, we might hope that the fraction of violated constraints in ψ^t is at least the sum of $\Pr[B_{j,f}]$ for every $f \in F$ and $j \in \{t, \ldots, t + \delta\sqrt{t}\}$. If this were the case we'd be done since Claim 18.32 implies that this sum is at least $\frac{\delta\sqrt{t}\epsilon nd}{2nW^2} = \frac{\delta\sqrt{t}\epsilon}{2W^2} > \epsilon'$. However, this is inaccurate since we are overcounting paths that contain more than one such violation (i.e., paths which are in the intersection of $B_{j,f}$ and $B_{j',f'}$ for $(j, f) \neq (j', f')$). To bound the effect of this overcounting we prove the following claim:

CLAIM 18.33
For every $k \in \mathbb{N}$ and set F of edges with $|F| = \epsilon nd$ for $\epsilon < \frac{1}{kd}$,

$$\sum_{\substack{j,j' \in \{t..t+k\} \\ f,f' \in F \\ (j,f) \neq (j',f')}} \Pr[B_{j,f} \cap B_{j,f'}] \leq 30kd\epsilon \tag{7}$$

PROOF: Only one edge can be the jth edge of a path, and so for every $f \neq f'$, $\Pr[B_{j,f} \cap B_{j,f'}] = 0$. Thus the left-hand side of (7) simplifies to

$$\sum_{j \neq j' \in \{t..t+k\}} \sum_{f \neq f'} \Pr[B_{j,f} \cap B_{j',f'}] \tag{8}$$

Let A_j be the event that the jth edge is in the set F. We get that (8) is equal to

$$\sum_{j \neq j' \in \{t..t+k\}} \Pr[A_j \cap A_{j'}] = 2 \sum_{j < j' \in \{t..t+k\}} \Pr[A_j \cap A_{j'}] \tag{9}$$

Let S be the set of at most $d\epsilon n$ vertices that are adjacent to an edge in F. For $j' < j$, $\Pr[A_j \cap A_{j'}]$ is bounded by the probability that a random $(j'-j)$-step path in G has both endpoints in S, or in other words that a random edge in the graph $G^{j'-j}$ has both endpoints in S. Using the fact that $\lambda(G^{j'-j}) = \lambda(G)^{j'-j} \leq 0.9^{j'-j}$, this probability is bounded by $d\epsilon(d\epsilon + 0.9^{|j-j'|})$ (see Note 18.18). Plugging this into (9) and using the formula for summation of arithmetic series, we get that:

$$2 \sum_{j < j' \in \{t,...,t+k\}} \Pr[A_j \cap A_{j'}] \leq$$

$$2 \sum_{j \in \{t,...,t+k\}} \sum_{i=1}^{t+k-j} d\epsilon(d\epsilon + 0.9^i) \leq$$

$$2k^2 d^2 \epsilon^2 + 2kd\epsilon \sum_{i=1}^{\infty} 0.9^i \leq 2k^2 d^2 \epsilon^2 + 20kd\epsilon \leq 30kd\epsilon$$

where the last inequality follows from $\epsilon < \frac{1}{kd}$. $\blacksquare$

Wrapping up.   Claims 18.32 and 18.33 together imply that

$$\sum_{\substack{j \in \{t..t+\delta\sqrt{t}\} \\ f \in F}} \Pr[B_{j,f}] \geq \delta\sqrt{t}\epsilon \frac{1}{2W^2} \tag{10}$$

$$\sum_{\substack{j,j' \in \{t..t+\delta\sqrt{t}\} \\ f,f' \in F \\ (j,f) \neq (j',f')}} \Pr[B_{j,f} \cap B_{j',f'}] \leq 30\delta\sqrt{t}d\epsilon \tag{11}$$

But (10) and (11) together imply that if p is a random constraint of ψ^t then

$$\Pr[p \text{ violated by } \mathbf{y}] \geq \Pr[\bigcup_{\substack{j \in \{t..t+\delta\sqrt{t}\} \\ f \in F}} B_{j,f}] \geq \frac{\delta\sqrt{t}\epsilon}{240dW^2}$$

where the last inequality is implied by the following technical claim:

CLAIM 18.34
Let $A_1, \ldots, A_n$ be n subsets of some set U satisfying $\sum_{i<j} |A_i \cap A_j| \leq C \sum_{i=1}^n |A_i|$ for some number $C \in \mathbb{N}$. Then,

$$\left| \bigcup_{i=1}^n A_i \right| \geq \frac{\sum_{i=1}^n |A_i|}{4C}$$

PROOF: We make $2C$ copies of every element $u \in U$ to obtain a set $\tilde{U}$ with $|\tilde{U}| = 2C|U|$. Now for every subset $A_i \subseteq U$, we obtain $\tilde{A}_i \subseteq \tilde{U}$ as follows: for every $u \in A_i$, we choose at random one of the $2C$ copies to put in $\tilde{A}_i$. Note that $|\tilde{A}_i| = |A_i|$. For every $i, j \in [n]$, $u \in A_i \cap A_j$, we denote by $I_{i,j,u}$ the indicator random variable that is equal to 1 if we made the same choice for the copy of u in $\tilde{A}_i$ and $\tilde{A}_j$, and equal to 0 otherwise. Since $\mathsf{E}[I_{i,j,u}] = \frac{1}{2C}$,

$$\mathsf{E}\left[|\tilde{A}_i \cap \tilde{A}_j|\right] = \sum_{u \in A_i \cap A_j} \mathsf{E}[I_{i,j,u}] = \frac{|A_i \cap A_j|}{2C}$$

and

$$\mathsf{E}\left[\sum_{i<j} |\tilde{A}_i \cap \tilde{A}_j|\right] = \frac{\sum_{i<j} |A_i \cap A_j|}{2C}$$

This means that there exists some choice of $\tilde{A}_1, \ldots, \tilde{A}_j$ such that

$$\sum_{i=1}^{n} |\tilde{A}_i| = \sum_{i=1}^{n} |A_i| \geq 2 \sum_{i<j} |\tilde{A}_i \cap \tilde{A}_j|$$

which by the inclusion-exclusion principle (see Section **??**) means that $|\cup_i \tilde{A}_i| \geq 1/2 \sum_i |\tilde{A}_i|$. But because there is a natural $2C$-to-one mapping from $\cup_i \tilde{A}_i$ to $\cup_i A_i$ we get that

$$|\cup_{i=1}^{n} A_i| \geq \frac{|\cup_{i=1}^{n} \tilde{A}_i|}{2C} \geq \frac{\sum_{i=1}^{n} |\tilde{A}_i|}{4C} = \frac{\sum_{i=1}^{n} |A_i|}{4C}$$

∎

Since $\epsilon' < \frac{\delta \sqrt{t} \epsilon}{240 d W^2}$, this proves the lemma. ∎

18.5.2   Alphabet Reduction: Proof of Lemma 18.30

Lemma 18.30 is actually a simple consequence of Corollary 18.26, once we restate it using our "qCSP view" of **PCP** systems.

COROLLARY 18.35 (qCSP VIEW OF **PCP** OF PROXIMITY.)
There exists positive integer q_0 and an exponential-time transformation that given any circuit C of size m and and n inputs and two numbers n_1, n_2 such that $n_1 + n_2 = n$ produces an instance ψ_C of q_0CSP of size $2^{poly(m)}$ over a binary alphabet such that:

1. *The variables can be thought of as being partitioned into three sets π_1, π_2, π_3 where π_1 has 2^{n_1} variables and π_2 has 2^{n_2} variables.*

2. *If $\mathbf{u_1} \in \{0,1\}^{n_1}, \mathbf{u_2} \in \{0,1\}^{n_2}$ is such that $\mathbf{u_1} \circ \mathbf{u_2}$ is a satisfying assignment for circuit C, then there is a string π_3 of size $2^{poly(m)}$ such that $\mathsf{WH}(\mathbf{u_1}) \circ \mathsf{WH}(\mathbf{u_2}) \circ \pi_3$ satisfies ψ_C.*

3. *For every strings $\pi_1, \pi_2, \pi_3 \in \{0,1\}^*$, where π_1 and π_2 have 2^{n_1} and 2^{n_2} bits respectively, if $\pi_1 \circ \pi_2 \circ \pi_3$ satisfy at least $1/2$ the constraints of ψ_C, then π_1, π_2 are 0.99-close to $\mathsf{WH}(\mathbf{u_1})$, $\mathsf{WH}(\mathbf{u_2})$ respectively for some $\mathbf{u_1}, \mathbf{u_2}$ such that $\mathbf{u_1} \circ \mathbf{u_2}$ is a satisfying assignment for circuit C.*

Now we are ready to prove Lemma 18.30.

PROOF OF LEMMA 18.30:   Suppose the given arity 2 formula φ has n variables $u_1, u_2, \ldots, u_n$, alphabet $\{0..W{-}1\}$ and N constraints $C_1, C_2, \ldots, C_N$. Think of each variable as taking values that are bit strings in $\{0,1\}^k$, where $k = \lceil \log W \rceil$. Then if constraint C_ℓ involves variables say u_i, u_j we may think of it as a circuit applied to the bit strings representing u_i, u_j where the constraint is said to be satisfied iff this circuit outputs 1. Say m is an upperbound on the size of this circuit over all constraints. Note that m is at most $2^{2k} < W^4$. We will assume without loss of generality that all circuits have the same size.

If we apply the transformation of Corollary 18.35 to this circuit we obtain an instance of $q_0\mathsf{CSP}$, say ψ_{C_l}. The strings u_i, u_j get replaced by strings of variables U_i, U_j of size $2^{2^k} < 2^{W^2}$ that take values over a binary alphabet. We also get a new set of variables that play the role analogous to π_3 in the statement of Corollary 18.35. We call these new variables Π_l.

Our reduction consists of applying the above transformation to each constraint, and taking the union of the $q_0\mathsf{CSP}$ instances thus obtained. However, it is important that these new $q_0\mathsf{CSP}$ instances *share* variables, in the following way: *for each old variable u_i, there is a string of new variables U_i of size 2^{2^k} and for each constraint C_l that contains u_i, the new $q_0\mathsf{CSP}$ instance ψ_{C_l} uses this string U_i.* (Note though that the Π_l variables are used only in ψ_{C_l} and never reused.) This completes the description of the new $q_0\mathsf{CSP}$ instance ψ (see Figure 18.7). Let us see that it works.
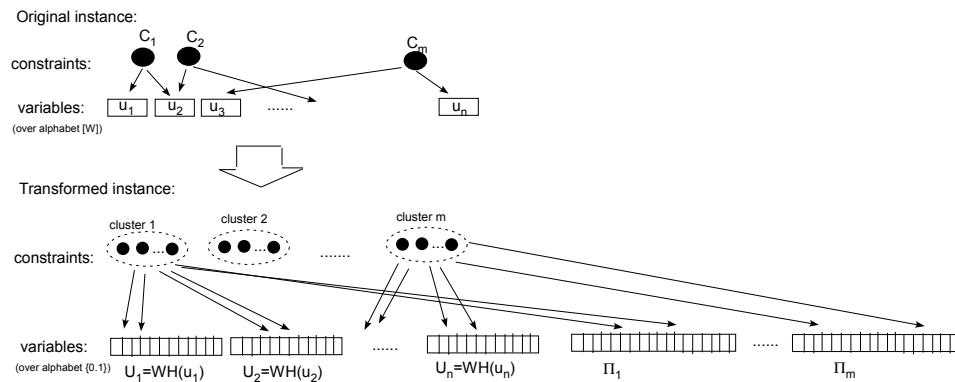
Figure 18.7:   The alphabet reduction transformation maps a 2CSP instance φ over alphabet $\{0..W{-}1\}$ into a $q\mathsf{CSP}$ instance ψ over the binary alphabet. Each variable of φ is mapped to a block of binary variables that in the correct assignment will contain the Walsh-Hadamard encoding of this variable. Each constraint C_ℓ of φ depending on variables u_i, u_j is mapped to a cluster of constraints corresponding to all the **PCP** of proximity constraints for C_ℓ. These constraint depend on the encoding of u_i and u_j, and on additional auxiliary variables that in the correct assignment contain the **PCP** of proximity proof that these are indeed encoding of values that make the constraint C_ℓ true.

Suppose the original instance φ was satisfiable by an assignment $\mathbf{u_1}, \ldots, \mathbf{u_n}$. Then we can produce a satisfying assignment for ψ by using part 2 of Corollary 18.35, so that for each constraint C_l involving u_i, u_j, the encodings $\mathsf{WH}(\mathbf{u_i}), \mathsf{WH}(\mathbf{u_i})$ act as π_1, π_2 and then we extend these via a suitable string π_3 into a satisfying assignment for ψ_{C_l}.

On the other hand if $\mathsf{val}(\varphi) < 1 - \epsilon$ then we show that $\mathsf{val}(\psi) < 1 - \epsilon/2$. Consider any assignment $\mathbf{U_1}, \mathbf{U_2}, \ldots, \mathbf{U_n}, \mathbf{\Pi_1}, \ldots, \mathbf{\Pi_N}$ to the variables of ψ. We "decode" it to an assignment for φ as follows. For each $i = 1, 2, \ldots, n$, if the assignment to U_i is 0.99-close to a linear function, let u_i be the string encoded by this linear function, and otherwise let u_i be some arbitrary string. Since $\mathsf{val}(\varphi) < 1 - \epsilon$, this new assignment fails to satisfy at least ϵ fraction of constraints in φ. For each constraint C_l of φ that is not satisfied by this assignment, we show that at least $1/2$ of the constraints in ψ_{C_l} are not satisfied by the original assignment, which leads to the conclusion that $\mathsf{val}(\psi) < 1 - \epsilon/2$. Indeed, suppose C_l involves u_i, u_j. Then $u_i \circ u_j$ is not a satisfying assignment to circuit C_l, so part 3 of Corollary 18.35 implies that regardless of the value of variables in Π_l, the assignment $\mathbf{U_i} \circ \mathbf{u_j} \circ \Pi_l$ must have failed to satisfy at least $1/2$ the constraints of ψ_{C_l}. ■

18.6 The original proof of the PCP Theorem.

The original proof of the **PCP** Theorem, which resisted simplification for over a decade, used algebraic encodings and ideas that are complicated versions of our proof of Theorem 18.21. (Indeed, Theorem 18.21 is the only part of the original proof that still survives in our writeup.) Instead of the linear functions used in Welsh-Hadamard code, they use low degree multivariate polynomials. These allow procedures analogous to the linearity test and local decoding, though the proofs of correctness are a fair bit harder. The alphabet reduction is also somewhat more complicated. The crucial part of Dinur's simpler proof, the one given here, is the gap amplification lemma (Lemma 18.29) that allows to iteratively improve the soundness parameter of the **PCP** from very close to 1 to being bounded away from 1 by some positive constant. This general strategy is somewhat reminiscent of the zig-zag construction of expander graphs and Reingold's deterministic logspace algorithm for undirect connectivity described in Chapter **??**.

Chapter notes

Problems

§1 Prove that for every two functions $r, q : \mathbb{N} \to \mathbb{N}$ and constant $s < 1$, changing the constant in the soundness condition in Definition 18.1 from $1/2$ to s will not change the class $\mathbf{PCP}(r, q)$.

§2 Prove that for every two functions $r, q : \mathbb{N} \to \mathbb{N}$ and constant $c > 1/2$, changing the constant in the completeness condition in Definition 18.1 from 1 to c will not change the class $\mathbf{PCP}(r, q)$.

§3 Prove that any language L that has a **PCP**-verifier using r coins and q *adaptive* queries also has a standard (i.e., non-adaptive) verifier using r coins and 2^q queries.

§4 Prove that $\mathbf{PCP}(0, \log n) = \mathbf{P}$. Prove that $\mathbf{PCP}(0, \mathrm{poly}(n)) = \mathbf{NP}$.

§5 Let L be the language of matrices A over GF(2) satisfying $\mathsf{perm}(A) = 1$ (see Chapters **??** and 8). Prove that L is in **PCP**(poly(n), poly(n)).

§6 Show that if $\mathsf{SAT} \in \mathbf{PCP}(r(n), 1)$ for $r(n) = o(\log n)$ then $\mathbf{P} = \mathbf{NP}$. (Thus the PCP Theorem is probably optimal up to constant factors.)

§7 (A simple PCP Theorem using logspace verifiers) Using the fact that a correct tableau can be verified in logspace, we saw the following exact characterization of **NP**:

$$\mathbf{NP} = \{L : \text{ there is a logspace machine } M \text{ s.t } x \in L \text{ iff } \exists y \ : \ M \text{ accepts } (x, y).\}.$$

Note that M has two-way access to y.

Let L-PCP($r(n)$) be the class of languages whose membership proofs can be probabilistically checked by a logspace machine that uses $O(r(n))$ random bits but makes only one pass over the proof. (To use the terminology from above, it has 2-way access to x but 1-way access to y.) As in the PCP setting, "probabilistic checking of membership proofs" means that for $x \in L$ there is a proof y that the machine accepts with probability 1 and if not, the machine rejects with probability at least $1/2$. Show that $\mathbf{NP} = \text{L-PCP}(\log n)$. Don't assume the PCP Theorem!

Hint: Design a verifier for 3SAT. The trivial idea would be that the proof contains a satisfying assignment and the verifier randomly picks a clause and reads the corresponding three bits in the proof to check if the clause is satisfied. This doesn't work. Why? The better idea is to require the „proof" to contain many copies of the satisfying assignment. The verifiers uses pairwise independence to run the previous test on these copies — which may or may not be the same string.

(This simple PCP Theorem is implicit in Lipton [Lip90]. The suggested proof is due to van Melkebeek.)

§8 Suppose we define $J - PCP(r(n))$ similarly to $L - PCP(r(n))$, except the verifier is only allowed to read $O(r(n))$ successive bits in the membership proof. (It can decide which bits to read.) Then show that $J - PCP(\log n) \subseteq \mathbf{L}$.

§9 Prove that there is an **NP**-language L and $x \notin L$ such that $f(x)$ is a 3SAT formula with m constraints having an assignment satisfying more than $m - m^{0.9}$ of them, where f is the reduction from f to 3SAT obtained by the proof of the Cook-Levin theorem (Section 2.3).

Hint: show that for an appropriate language L, a slight change in the input for the Cook-Levin reduction will also cause only a slight change in the output, even though this change might cause a YES instance of the language to become a NO instance.

§10 Show a poly$(n, 1/\epsilon)$-time $1 + \epsilon$-approximation algorithm for the knapsack problem. That is, show an algorithm that given $n + 1$ numbers $a_1, \ldots, a_n \in \mathbb{N}$ (each represented by at most n bits) and $k \in [n]$, finds a set $S \subseteq [n]$ with $|S| \leq k$ such that $\sum_{i \in S} a_i \geq \frac{\mathsf{opt}}{1+\epsilon}$ where

$$\mathsf{opt} = \max_{S \subseteq [n], |S| \leq k} \sum_{i \in S} a_i$$

Hint: first show that the problem can be solved exactly using dynamic programming in time poly(n, m) if all the numbers involved are in the set $[m]$. Then, show one can obtain an approximation algorithm by keeping only the $O(\log(1/\epsilon) + \log n)$ most significant bits of every number.

§11 Show a polynomial-time algorithm that given a satisfiable 2CSP-instance φ (over binary alphabet) finds a satisfying assignment for φ.

§12 Prove that QUADEQ is **NP**-complete.

Hint: show you can express satisfiability for SAT formulas using quadratic equations.

§13 Prove that if Z, U are two $n \times n$ matrices over GF(2) such that $Z \neq U$ then

$$\Pr_{\mathbf{r}, \mathbf{r}' \in_R \{0,1\}^n} [\mathbf{r} Z \mathbf{r}' \neq \mathbf{r} U \mathbf{r}'] \geq \frac{1}{4}$$

Hint: using linearity reduce this to the case that U is the all zero matrix, and then prove this using two applications of the random subsum principle.

§14 Show a *deterministic* poly$(n, 2^q)$-time algorithm that given a qCSP-instance φ (over binary alphabet) with m clauses outputs an assignment satisfying $m/2^q$ of these assignment.

Hint: one way to solve this is to use q-wise independent functions
??.

§15 Let S_t be the binomial distribution over t balanced coins. That is, $\Pr[S_t = k] = \binom{t}{k} 2^{-t}$. Prove that for every $\delta < 1$, the statistical distance of S_t and $S_{t+\delta\sqrt{t}}$ is at most 10ϵ.

Hint: approximate the binomial coefficient using Stirling's formula for approximating factorials.

§16 The *long-code* for a set $\{0, \ldots, W - 1\}$ is the function $\mathsf{LC} : \{0, \ldots, W - 1\} \rightarrow \{0, 1\}^{2^W}$ such that for every $i \in \{0..W{-}1\}$ and a function $f : \{0..W{-}1\} \rightarrow \{0, 1\}$, (where we identify f with an index in $[2^w]$) the fth position of $\mathsf{LC}(i)$, denoted by $\mathsf{LC}(i)_f$, is $f(i)$. We say that a function $L : \{0, 1\}^{2^W} \rightarrow \{0, 1\}$ is a *long-code codeword* if $L = \mathsf{LC}(i)$ for some $i \in \{0..W{-}1\}$.

(a) Prove that LC is an error-correcting code with distance half. That is, for every $i \neq j \in \{0..W{-}1\}$, the fractional Hamming distance of $\mathsf{LC}(i)$ and $\mathsf{LC}(j)$ is half.

(b) Prove that LC is *locally-decodable*. That is, show an algorithm that given random access to a function $L : 2^{\{0,1\}^W} \to \{0,1\}$ that is $(1{-}\epsilon)$-close to $\mathsf{LC}(i)$ and $f : \{0..W{-}1\} \to \{0,1\}$ outputs $\mathsf{LC}(i)_f$ with probability at least 0.9 while making at most 2 queries to L.

(c) Let $L = \mathsf{LC}(i)$ for some $i \in \{0..W{-}1\}$. Prove the for every $f : \{0..W{-}1\} \to \{0,1\}$, $L(f) = 1 - L(\overline{f})$, where $\overline{f}$ is the negation of f (i.e. , $\overline{f}(i) = 1 - f(i)$ for every $i \in \{0..W{-}1\}$).

(d) Let T be an algorithm that given random access to a function $L : 2^{\{0,1\}^W} \to \{0,1\}$, does the following:

    i. Choose f to be a random function from $\{0..W{-}1\} \to \{0,1\}$.

   ii. If $L(f) = 1$ then output TRUE.

  iii. Otherwise, choose $g : \{0..W{-}1\} \to \{0,1\}$ as follows: for every $i \in \{0..W{-}1\}$, if $f(i) = 0$ then set $g(i) = 0$ and otherwise set $g(i)$ to be a random value in $\{0,1\}$.

   iv. If $L(g) = 0$ then output TRUE; otherwise output FALSE.

Prove that if L is a long-code codeword (i.e., $L = \mathsf{LC}(i)$ for some i) then T outputs TRUE with probability one.

Prove that if L is a *linear function* that is non-zero and not a longcode codeword then T outputs TRUE with probability at most 0.9.

(e) Prove that LC is *locally testable*. That is, show an algorithm that given random access to a function $L : \{0,1\}^W \to \{0,1\}$ outputs TRUE with probability one if L is a long-code codeword and outputs FALSE with probability at least $1/2$ if L is not 0.9-close to a long-code codeword, while making at most a constant number of queries to L.

> **Hint:** use the test T above combined with linearity testing, self correction, and a simple test to rule out the constant zero function.

(f) Using the test above, give an alternative proof for the Alphabet Reduction Lemma (Lemma 18.30).

> **Hint:** To transform a $2\mathsf{CSP}_W$ formula φ over n variables into a $q\mathsf{CSP}$ ψ over binary alphabet, use 2^W variables $u_1^j, \ldots, u_{2^W}^j$ for each variable u_j of φ. In the correct proof each variable u_j of φ. In the correct proof these variables will contain the longcode encoding of u_j. Then, add a set of 2^{W^2} vari- ables $y_1^i, \ldots, y_{2^{W^2}}^i$ for each constraint φ_i of φ. In the correct proof these variables will contain the longcode encoding of the assign- ment for the constraint φ_i. For every constraint of φ, ψ will contain constraints for testing the longcode of both the x and y variables involved in the constraint, testing consistency between the x vari- ables and the y variables, and testing that the y variables actually encode a satisfying assignment.

Omitted proofs

The preprocessing step transforms a qCSP-instance φ into a "nice" 2CSP-instance ψ through the following three claims:

CLAIM 18.36
There is a CL- reduction mapping any qCSP instance φ into a $2CSP_{2^q}$ instance ψ such that

$$\mathsf{val}(\varphi) \le 1 - \epsilon \Rightarrow \mathsf{val}(\psi) \le 1 - \epsilon/q$$

PROOF: Given a qCSP-instance φ over n variables $u_1, \ldots, u_n$ with m constraints, we construct the following $2CSP_{2^q}$ formula ψ over the variables $u_1, \ldots, u_n, y_1, \ldots, y_m$. Intuitively, the y_i variables will hold the restriction of the assignment to the q variables used by the ith constraint, and we will add constraints to check consistency: that is to make sure that if the ith constraint depends on the variable u_j then u_j is indeed given a value consistent with y_i. Specifically, for every φ_i of φ that depends on the variables $u_1, \ldots, u_q$, we add q constraints $\{\psi_{i,j}\}_{j \in [q]}$ where $\psi_{i,j}(y_i, u_j)$ is true iff y_i encodes an assignment to $u_1, \ldots, u_q$ satisfying φ_i and u_j is in $\{0, 1\}$ and agrees with the assignment y_i. Note that the number of constraints in ψ is qm.

Clearly, if φ is satisfiable then so is ψ. Suppose that $\mathsf{val}(\varphi) \le 1 - \epsilon$ and let $u_1, \ldots, u_k, y_1, \ldots, y_m$ be any assignment to the variables of ψ. There exists a set $S \subseteq [m]$ of size at least ϵm such that the constraint φ_i is violated by the assignment $u_1, \ldots, u_k$. For any $i \in S$ there must be at least one $j \in [q]$ such that the constraint $\psi_{i,j}$ is violated. ∎

CLAIM 18.37
There is an absolute constant d and a CL- reduction mapping any $2CSP_W$ instance φ into a $2CSP_W$ instance ψ such that
$$\mathsf{val}(\varphi) \le 1 - \epsilon \Rightarrow \mathsf{val}(\psi) \le 1 - \epsilon/(100Wd).$$

and the constraint graph of ψ is d-regular. That is, every variable in ψ appears in exactly d constraints.

PROOF: Let φ be a $2CSP_W$ instance, and let $\{G_n\}_{n \in \mathbb{N}}$ be an explicit family of d-regular expanders. Our goal is to ensure that each variable appears in φ at most $d + 1$ times (if a variable appears less than that, we can always add artificial constraints that touch only this variable). Suppose that u_i is a variable that appears in k constraints for some $n > 1$. We will change u_i into k variables $y_i^1, \ldots, y_i^k$, and use a different variable of the form y_i^j in the place of u_i in each constraint u_i originally appeared in. We will also add a constraint requiring that y_i^j is equal to $y_i^{j'}$ for every edge (j, j') in the graph G_k. We do this process for every variable in the original instance, until each variable appears in at most d equality constraints and one original constraint. We call the resulting 2CSP-instance ψ. Note that if φ has m constraints then ψ will have at most $m + dm$ constraints.

Clearly, if φ is satisfiable then so is ψ. Suppose that $\mathsf{val}(\varphi) \le 1 - \epsilon$ and let $\mathbf{y}$ be any assignment to the variables of ψ. We need to show that $\mathbf{y}$ violates at least $\frac{\epsilon m}{100W}$ of the constraints of ψ. Recall that for each variable u_i that appears k times in φ, the assignment $\mathbf{y}$ has k variables $y_i^1, \ldots, y_i^k$. We compute an assignment $\mathbf{u}$ to φ's variables as follows: u_i is assigned the plurality value of

$y_i^1, \ldots, y_i^k$. We define t_i to be the number of y_i^j's that *disagree* with this plurality value. Note that $0 \leq t_i \leq k(1 - 1/W)$ (where W is the alphabet size). If $\sum_{i=1}^n t_i \geq \frac{\epsilon}{4}m$ then we are done. Indeed, by (3) (see Note 18.18), in this case we will have at least $\sum_{i=1}^n \frac{t_i}{10W} \geq \frac{\epsilon}{40W}m$ equality constraints that are violated.

Suppose now that $\sum_{i=1}^n t_i < \frac{\epsilon}{4}m$. Since $\mathsf{val}(\varphi) \leq 1 - \epsilon$, there is a set S of at least ϵm constraints violated in φ by the plurality assignment $\mathbf{u}$. All of these constraints are also present in ψ and since we assume $\sum_{i=1}^n t_i < \frac{\epsilon}{4}m$, at most half of them are given a different value by the assignment $\mathbf{y}$ than the value given by $\mathbf{u}$. Thus the assignment $\mathbf{y}$ violates at least $\frac{\epsilon}{2}m$ constraints in ψ. ∎

CLAIM 18.38
There is an absolute constant d and a CL-reduction mapping any $2\mathsf{CSP}_W$ instance φ with d'-regular constraint graph for $d \geq d'$ into a $2\mathsf{CSP}_W$ instance ψ such that

$$\mathsf{val}(\varphi) \leq 1 - \epsilon \Rightarrow \mathsf{val}(\psi) \leq 1 - \epsilon/(10d)$$

and the constraint graph of ψ is a $4d$-regular expander, with half the edges coming out of each vertex being self-loops.

PROOF: There is a constant d and an explicit family $\{G_n\}_{n \in \mathbb{N}}$ of graphs such that for every n, G_n is a d-regular n-vertex 0.1-expander graph (See Note 18.18).

Let φ be a $2\mathsf{CSP}$-instance as in the claim's statement. By adding self loops, we can assume that the constraint graph has degree d (this can at worst decrease the gap by factor of d). We now add "null" constraints (constraints that always accept) for every edge in the graph G_n. In addition, we add $2d$ null constraints forming self-loops for each vertex. We denote by ψ the resulting instance. Adding these null constraints reduces the fraction of violated constraints by a factor at most four. Moreover, because any regular graph H satisfies $\lambda(H) \leq 1$ and because of λ's subadditivity (see Exercise 11, Chapter **??**), $\lambda(\psi) \leq \frac{3}{4} + \frac{1}{4}\lambda(G_n) \leq 0.9$ where by $\lambda(\psi)$ we denote the parameter λ of ψ's constraint graph. ∎

Chapter 19

More PCP Theorems and the Fourier Transform Technique

The **PCP** Theorem has several direct applications in complexity theory, in particular showing that unless $\mathbf{P} = \mathbf{NP}$, many $\mathbf{NP}$ optimization problems can not be approximated in polynomial-time to within arbitrary precision. However, for some applications, the standard **PCP** Theorem does not suffice, and we need stronger (or simply different) "**PCP** Theorems". In this chapter we survey some of these results and their proofs. The *Fourier transform technique* turned out to be especially useful in advanced **PCP** constructions, and in other areas in theoretical computer science. We describe the technique and show two of its applications. First, we use Fourier transforms to prove the correctness of the linearity testing algorithm of Section 18.4, completing the proof of the **PCP** Theorem. We then use it to prove a stronger **PCP** Theorem due to Håstad, showing *tight* inapproximability results for many important problems, including MAX 3SAT.

19.1 Parallel Repetition of PCP's

Recall that the *soundness parameter* of a **PCP** system is the probability that the verifier may accept a false statement. Definition 18.1 specified the soundness parameter to be $1/2$, but as we noted, it can be reduced to an arbitrary small constant by increasing the number of queries. Yet for some applications we need a system with, say, three queries, but an arbitrarily small constant soundness parameter. Raz has shown that this can be achieved if we consider systems with *non binary alphabet*. (For a finite set S, we say that a **PCP** verifier *uses alphabet S* if it takes as input a proof string π in S^*.) The idea is simple and natural: use *parallel repetition*. That is, we take a **PCP** verifier V and run ℓ independent copies of it, to obtain a new verifier V^ℓ such that a query of V^ℓ is the concatenation of the ℓ queries of V, and an answer is a concatenation of the ℓ answers. (So, if the original verifier V used proofs over, say, the binary alphabet, then the verifier V^ℓ will use the alphabet $\{0, 1\}^\ell$.) The verifier V^ℓ accepts the proof only of all the ℓ executions of V accept. Formally, we define parallel repetition as follows:

DEFINITION 19.1 (PARALLEL REPETITION)
Let S be a finite set. Let V be a **PCP** verifier using alphabet S and let $\ell \in \mathbb{N}$. The ℓ-*times parallel*

	Original V	Parallel repeated V^ℓ	Sequential repeated $V^{\mathsf{seq}\ell}$
Alphabet size	W	W^ℓ	W
Proof size	m	m^ℓ	m
Random coins used	r	ℓr	ℓr
Number of queries	q	q	ℓq
Completeness probability	1	1	1
soundness parameter	$1 - \delta$	$(1 - \delta^a)^{b\ell}$	$(1 - \delta)^\ell$

Table 19.1: Parameters of ℓ-times parallel repeated verifier V^ℓ vs. parameters for sequential repetition.

repeated V is the verifier V^ℓ that operates as follows:

1. V^ℓ uses the alphabet $\hat{S} = S^\ell$. We denote the input proof string to V^ℓ by $\hat{\pi}$.

2. Let q denote the number of queries V makes. On any input x, V^ℓ chooses ℓ independent random tapes $r^1, \ldots, r^\ell$ for V, and runs V on the input and these tapes to obtain ℓ sets of q queries

$$
\begin{array}{cccc}
i_1^1, & i_2^1, & \cdots & , i_q^1 \\
i_1^2, & i_2^2, & \cdots & , i_q^2 \\
       &   & \cdots &   \\
i_1^\ell, & i_2^\ell, & \cdots & , i_q^\ell
\end{array}
$$

3. V^ℓ makes q queries $i_1, \ldots, i_q$ to $\hat{\pi}$ where i_j is $\langle i_j^1, \ldots, i_j^\ell \rangle$ (under a suitable encoding of $\mathbb{N}^\ell$ into $\mathbb{N}$).

4. For $j \in [q]$, denote $\langle a_j^1, \ldots, a_j^\ell \rangle = \hat{\pi}(i_j)$. The verifier V^ℓ accepts if and only for every $k \in [\ell]$, the verifier V on random tape r_k accepts when given the responses $a_1^k, \ldots, a_q^k$.

REMARK 19.2
For every input x, if there is a proof π such that on input x, the verifier V accepts π with probability one, then there is a proof $\hat{\pi}$ such that on input x, the verifier V^ℓ accepts $\hat{\pi}$ with probability one. Namely, for every ℓ-tuple of positions $i^1, \ldots, i^\ell$, the proof $\hat{\pi}$ contains the tuple $\langle \pi[i^1], \ldots, \pi[i^\ell] \rangle$. Note that $|\hat{\pi}| = |\pi|^\ell$.

Why is it called "parallel repetition"? We call the verifier V^ℓ the *parallel* repeated version of V to contrast with *sequential repetition*. If V is a **PCP** verifier and $\ell \in \mathbb{N}$, we say that ℓ-*times sequentially repeated* V, denoted $V^{\mathsf{seq}\ell}$, is the verifier that chooses ℓ random tapes for V, then makes the $q\ell$ queries corresponding to these tapes one after the other, and accepts only if all the instances accept. Note that $V^{\mathsf{seq}\ell}$ uses the same alphabet as V, and uses proofs of the same size. The relation between the parameters of V, V^ℓ and $V^{\mathsf{seq}\ell}$ is described in Table 19.1.

It is a simple exercise to show that if V's soundness parameter was $1 - \delta$ then $V^{\mathsf{seq}\ell}$ soundness parameter will be equal to $(1-\delta)^\ell$. One may expect the soundness parameter of the parallel repeated verifier V^ℓ to also be $(1-\delta)^\ell$. It turns out this is not the case (there is a known counterexample [**?**]), however the soundness parameter does decay exponentially with the number of repetitions:

THEOREM 19.3 (PARALLEL REPETITION LEMMA, [RAZ98])
There exist constants a, b (independent of ℓ but depending on the alphabet size used and number of queries) such that the soundness parameter of V^ℓ is at most $(1 - \delta^a)^{b\ell}$

We omit the proof of Theorem 19.3 for lack of space. Roughly speaking, the reason analyzing soundness of V^ℓ is so hard is the following: for every tuple $\langle i_1, \ldots, i_\ell \rangle$, the corresponding position in the proof for V^ℓ is "supposed" to consist of the values $\pi[i_1] \circ \cdots \pi[i_\ell]$ where π is some proof for V. However, *a priori*, we do not know if the proof satisfies this property. It may be that the proof is *inconsistent* and that two tuples containing the ith position "claim" a different assignment for $\pi[i]$.

REMARK 19.4
The Gap Amplification Lemma (Lemma 18.29) of the previous chapter has a similar flavor, in the sense that it also reduced the soundness parameter at the expense of an increase in the alphabet size. However, that lemma assumed that the soundness parameter is very close to 1, and its proof does not seem to generalize for soundness parameters smaller than $1/2$. We note that a weaker version of Theorem 19.3, with a somewhat simpler proof, was obtained by Feige and Kilian [**?**]. This weaker version is sufficient for many applications, including for Håstad's 3-query **PCP** theorem (see Section 19.2 below).

19.2 Håstad's 3-bit PCP Theorem

In most cases, the **PCP** Theorem does not immediately answer the question of exactly how well can we approximate a given optimization problem (even assuming $\mathbf{P} \neq \mathbf{NP}$). For example, the **PCP** Theorem implies that if $\mathbf{P} \neq \mathbf{NP}$ then MAX 3SAT cannot be c-approximated in polynomial-time for some constant $\rho < 1$. But if one follows closely the proof of Theorem 18.13, this constant ρ turns out to be very close to one, and in particular it is larger than 0.999. On the other hand, as we saw in Example 18.6, there is a known 7/8-approximation algorithm for MAX 3SAT. What is the true "approximation complexity" of this problem? In particular, is there a polynomial-time 0.9-approximation algorithm for it? Similar questions are the motivation behind many stronger **PCP** theorems. In particular, the following theorem by Håstad implies that for *every* $\epsilon > 0$ there is no polynomial-time $(7/8+\epsilon)$-approximation for MAX 3SAT unless $\mathbf{P} = \mathbf{NP}$:

THEOREM 19.5 (HÅSTAD'S 3-BIT **PCP** [**?**])
*For every $\epsilon > 0$ and every language $L \in \mathbf{NP}$ there is a **PCP**-verifier V for L making three (binary) queries having completeness probability at least $1 - \epsilon$ and soundness parameter at most $1/2 + \epsilon$.*

Moreover, the test used by V are linear. *That is, given a proof $\pi \in \{0,1\}^m$, V chooses a triple $(i_1, i_2, i_3) \in [m]^3$ and $b \in \{0, 1\}$ according to some distribution and accepts iff $\pi_{i_1} + \pi_{i_2} + \pi i_3 = b$ (mod 2).*

Theorem 19.5 immediately implies that the problem MAX E3LIN is **NP**-hard to $1/2+\epsilon$-approximate for every $\epsilon > 0$, where MAX E3LIN is the problem of finding a solution maximizing the number of satisfied equations among a given system of linear equations over GF(2), with each equation containing at most 3 variables. Note that this hardness of approximation result is tight since a random assignment is expected to satisfy half of the equations. Also note that finding out whether there exists a solution satisfying *all* of the equations can be done in polynomial-time using Gaussian elimination (and hence the imperfect completeness in Theorem 19.5 is inherent).

The result for MAX 3SAT is obtained by the following corollary:

COROLLARY 19.6
For every $\epsilon > 0$, computing $(7/8+\epsilon)$-approximation to MAX 3SAT *is* **NP**-*hard.*

PROOF: We reduce MAX E3LIN to MAX 3SAT. Take any instance of MAX E3LIN where we are interested in determining whether $(1-\epsilon)$ fraction of the equations can be satisfied or at most $1/2+\epsilon$ are. Represent each linear constraint by four $3CNF$ clauses in the obvious way. For example, the linear constraint $a+b+c = 0 \pmod 2$ is equivalent to the clauses $(\overline{a} \vee b \vee c), (a \vee \overline{b} \vee c), (a \vee b \vee \overline{c}), (\overline{a} \vee \overline{b} \vee \overline{c})$. If a, b, c satisfy the linear constraint, they satisfy all 4 clauses and otherwise they satisfy at most 3 clauses. We conclude that in one case at least $(1 - \epsilon)$ fraction of clauses are simultaneously satisfiable, and in the other case at most $1 - (\frac{1}{2} - \epsilon) \times \frac{1}{4} = \frac{7}{8} - \frac{\epsilon}{4}$ fraction are. The ratio between the two cases tends to $7/8$ as ϵ decreases. Since Theorem 19.5 implies that distinguishing between the two cases is **NP**-hard for every constant ϵ, the result follows. ∎

19.3   Tool: the Fourier transform technique

The continuous Fourier transform is extremely useful in mathematics and engineering. Likewise, the discrete Fourier transform has found many uses in algorithms and complexity, in particular for constructing and analyzing **PCP**'s. The Fourier transform technique for **PCP**'s involves calculating the maximum acceptance probability of the verifier using Fourier analysis of the functions presented in the proof string. It is delicate enough to give "tight" inapproximability results for MAX INDSET, MAX 3SAT, and many other problems.

To introduce the technique we start with a simple example: analysis of the linearity test over GF(2) (i.e., proof of Theorem 18.23). We then introduce the *Long Code* and show how to test for membership in it. These ideas are then used to prove Håstad's 3-bit PCP Theorem.

19.3.1   Fourier transform over $\mathrm{GF}(2)^n$

The Fourier transform over $\mathrm{GF}(2)^n$ is a tool to study functions on the Boolean hypercube. In this chapter, it will be useful to use the set $\{+1, -1\} = \{\pm 1\}$ instead of $\{0, 1\}$. To transform $\{0, 1\}$ to $\{\pm 1\}$, we use the mapping $b \mapsto (-1)^b$ (i.e., $0 \mapsto +1$, $1 \mapsto -1$). Thus we write the hypercube as $\{\pm 1\}^n$ instead of the more usual $\{0, 1\}^n$. Note this maps the XOR operation (i.e., addition in GF(2)) into the multiplication operation.

The set of functions from $\{\pm 1\}^n$ to $\mathbb{R}$ defines a 2^n-*dimensional Hilbert space* (see Section **??**) as follows. Addition and multiplication by a scalar are defined in the natural way: $(f + g)(\mathbf{x}) =$

$f(\mathbf{x}) + g(\mathbf{x})$ and $(\alpha f)(\mathbf{x}) = \alpha f(\mathbf{x})$ for every $f, g : \{\pm 1\}^n \to \mathbb{R}$, $\alpha \in \mathbb{R}$. We define the inner product of two functions f, g, denoted $\langle f, g \rangle$, to be $\mathsf{E}_{\mathbf{x} \in \{\pm 1\}^n}[f(\mathbf{x})g(\mathbf{x})]$.

The *standard basis* for this space is the set $\{\mathbf{e_x}\}_{\mathbf{x} \in \{\pm 1\}^n}$, where $\mathbf{e_x}(\mathbf{y})$ is equal to 1 if $\mathbf{y} = \mathbf{x}$, and equal to 0 otherwise. This is an orthonormal basis, and every function $f : \{\pm 1\}^n \to \mathbb{R}$ can be represented in this basis as $f = \sum_{\mathbf{x}} a_{\mathbf{x}} \mathbf{e_x}$. For every $\mathbf{x} \in \{\pm 1\}^n$, the coefficient $a_{\mathbf{x}}$ is equal to $f(\mathbf{x})$. The *Fourier basis* for this space is the set $\{\chi_\alpha\}_{\alpha \subseteq [n]}$ where $\chi_\alpha(\mathbf{x}) = \prod_{i \in \alpha} x_i$ ($\chi_\emptyset$ is the constant 1 function). These correspond to the *linear* functions over GF(2). To see this, note that every linear function of the form $\mathbf{b} \mapsto \mathbf{a} \odot \mathbf{b}$ (with $\mathbf{a}, \mathbf{b} \in \{0, 1\}^n$) is mapped by our transformation to the function taking $\mathbf{x} \in \{\pm 1\}^n$ to $\prod_{i \text{ s.t. } a_i = 1} x_i$.

The Fourier basis is indeed an orthonormal basis for the Hilbert space. Indeed, the random subsum principle implies that for every $\alpha, \beta \subseteq [n]$, $\langle \chi_\alpha, \chi_\beta \rangle = \delta_{\alpha,\beta}$ where $\delta_{\alpha,\beta}$ is equal to 1 iff $\alpha = \beta$ and equal to 0 otherwise. This means that every function $f : \{\pm 1\}^n \to \mathbb{R}$ can be represented as $f = \sum_{\alpha \subseteq [n]} \hat{f}_\alpha \chi_\alpha$. We call $\hat{f}_\alpha$ the αth *Fourier coefficient* of f.

We will often use the following simple lemma:

LEMMA 19.7
Every two functions $f, g : \{\pm 1\}^n \to \mathbb{R}$ satisfy

1. $\langle f, g \rangle = \sum_\alpha \hat{f}_\alpha \hat{g}_\alpha$.

2. *(Parseval's Identity)* $\langle f, f \rangle = \sum_\alpha \hat{f}_\alpha^2$

PROOF: The second property follows from the first. To prove the first we expand

$$\langle f, g \rangle = \langle \sum_\alpha \hat{f}_\alpha \chi_\alpha, \sum_\beta \hat{g}_\beta \chi_\beta \rangle =$$

$$\sum_{\alpha,\beta} \hat{f}_\alpha \hat{g}_\beta \langle \chi_\alpha, \chi_\beta \rangle = \sum_{\alpha,\beta} \hat{f}_\alpha \hat{g}_\beta \delta_{\alpha,\beta} = \sum_\alpha \hat{f}_\alpha \hat{g}_\alpha$$

∎

EXAMPLE 19.8
Some examples for the Fourier transform of particular functions:

1. If $f(u_1, u_2, \ldots, u_n) = u_i$ (i.e., f is a *coordinate function*, a concept we will see again soon) then $f = \chi_{\{i\}}$ and so $\hat{f}_{\{i\}} = 1$ and $\hat{f}_\alpha = 0$ for $\alpha \neq \{i\}$.

2. If f is a random boolean function on n bits, then each $\hat{f}_\alpha$ is a random variable that is a sum of 2^n binomial variables (equally likely to be $1, -1$) and hence looks like a normally distributed variable with standard deviation $2^{n/2}$ and mean 0. Thus with high probability, all 2^n Fourier coefficients have values in $[-\frac{\text{poly}(n)}{2^{n/2}}, \frac{\text{poly}(n)}{2^{n/2}}]$.

The connection to PCPs: High level view

In the PCP context we are interested in *Boolean-valued* functions, i.e., those from $GF(2)^n$ to $GF(2)$. Under our transformation these are mapped to functions from $\{\pm 1\}^n$ to $\{\pm 1\}$. Thus, we say that : $f \{\pm 1\}^n \to \mathbb{R}$ is *Boolean* if $f(\mathbf{x}) \in \{\pm 1\}$ for every $\mathbf{x} \in \{\pm 1\}^n$. Note that if f is Boolean then $\langle f, f \rangle = \mathsf{E}_\mathbf{x}[f(\mathbf{x})^2] = 1$.

On a high level, we use the Fourier transform in the soundness proofs for PCP's to show that if the verifier accepts a proof π with high probability then π is "close to" being "well-formed" (where the precise meaning of "close-to" and "well-formed" is context dependent). Technically, we will often be able to relate the acceptance probability of the verifier to an expectation of the form $\langle f, g \rangle = \mathsf{E}_\mathbf{x}[f(\mathbf{x})g(\mathbf{x})]$, where f and g are Boolean functions arising from the proof. We then use techniques similar to those used to prove Lemma 19.7 to relate this acceptance probability to the Fourier coefficients of f, g, allowing us to argue that if the verifier's test accepts with high probability, then f and g have few relatively large Fourier coefficients. This will provide us with some nontrivial useful information about f and g, since in a "generic" or random function, all the Fourier coefficient are small and roughly equal.

19.3.2   Analysis of the linearity test over $GF(2)$

We will now prove Theorem 18.23, thus completing the proof of the **PCP** Theorem. Recall that the linearity test is provided a function $f : \mathrm{GF}(2)^n \to \mathrm{GF}(2)$ and has to determine whether f has significant agreement with a linear function. To do this it picks $\mathbf{x}, \mathbf{y} \in \mathrm{GF}(2)^n$ randomly and accepts iff $f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y})$.

Now we rephrase this test using $\{\pm 1\}$ instead of $\mathrm{GF}(2)$, so linear functions turn into Fourier basis functions. For every two vectors $\mathbf{x}, \mathbf{y} \in \{\pm 1\}^n$, we denote by $\mathbf{xy}$ their componentwise multiplication. That is, $\mathbf{xy} = (x_1 y_1, \ldots, x_n y_n)$. Note that for every basis function $\chi_\alpha(\mathbf{xy}) = \chi_\alpha(\mathbf{x})\chi_\alpha(\mathbf{y})$.

For two Boolean functions f, g, $\langle f, g \rangle$ is equal to the fraction of inputs on which they *agree* minus the fraction of inputs on which they *disagree*. It follows that for every $\epsilon \in [0, 1]$ and functions $f, g : \{\pm 1\}^n \to \{\pm 1\}$, f has agreement $\frac{1}{2} + \frac{\epsilon}{2}$ with g iff $\langle f, g \rangle = \epsilon$. Thus, if f has a large Fourier coefficient then it has significant agreement with some Fourier basis function, or in the $\mathrm{GF}(2)$ worldview, f is close to some linear function. This means that Theorem 18.23 can be rephrased as follows:

THEOREM 19.9
Suppose that $f : \{\pm 1\}^n \to \{\pm 1\}$ satisfies $\Pr_{x,y}[f(\mathbf{xy}) = f(\mathbf{x})f(\mathbf{y})] \geq \frac{1}{2} + \epsilon$. Then, there is some $\alpha \subseteq [n]$ such $\hat{f}_\alpha \geq 2\epsilon$.

PROOF: We can rephrase the hypothesis as $E_{\mathbf{x},\mathbf{y}}[f(\mathbf{xy})f(\mathbf{x})f(\mathbf{y})] \geq (\frac{1}{2} + \epsilon) - (\frac{1}{2} - \epsilon) = 2\epsilon$. We note that from now on we do not need f to be Boolean, but merely to satisfy $\langle f, f \rangle = 1$.

Expressing f by its Fourier expansion,

$$2\epsilon \leq \mathsf{E}_{\mathbf{x},\mathbf{y}}[f(\mathbf{xy})f(\mathbf{x})f(\mathbf{y})] = \mathsf{E}_{\mathbf{x},\mathbf{y}}[(\sum_\alpha \hat{f}_\alpha \chi_\alpha(\mathbf{xy}))(\sum_\beta \hat{f}_\beta \chi_\beta(\mathbf{x}))(\sum_\gamma \hat{f}_\gamma \chi_\gamma(\mathbf{y}))].$$

Since $\chi_\alpha(\mathbf{x}\mathbf{y}) = \chi_\alpha(\mathbf{x})\chi_\alpha(\mathbf{y})$ this becomes

$$= \mathsf{E}_{\mathbf{x},\mathbf{y}}[\sum_{\alpha,\beta,\gamma} \hat{f}_\alpha \hat{f}_\beta \hat{f}_\gamma \chi_\alpha(\mathbf{x})\chi_\alpha(\mathbf{y})\chi_\beta(\mathbf{x})\chi_\gamma(\mathbf{y})].$$

Using linearity of expectation:

$$= \sum_{\alpha,\beta,\gamma} \hat{f}_\alpha \hat{f}_\beta \hat{f}_\gamma \mathsf{E}_{\mathbf{x},\mathbf{y}}[\chi_\alpha(\mathbf{x})\chi_\alpha(\mathbf{y})\chi_\beta(\mathbf{x})\chi_\gamma(\mathbf{y})]$$

$$= \sum_{\alpha,\beta,\gamma} \hat{f}_\alpha \hat{f}_\beta \hat{f}_\gamma \mathsf{E}_{\mathbf{x}}[\chi_\alpha(\mathbf{x})\chi_\beta(\mathbf{x})]\, \mathsf{E}_{\mathbf{y}}[\chi_\alpha(\mathbf{y})\chi_\gamma(\mathbf{y})]$$

$$\text{(because } \mathbf{x}, \mathbf{y} \text{ are independent).}$$

By orthonormality $\mathsf{E}_{\mathbf{x}}[\chi_\alpha(\mathbf{x})\chi_\beta(\mathbf{x})] = \delta_{\alpha,\beta}$, so we simplify to

$$= \sum_\alpha \hat{f}_\alpha^3$$

$$\leq (\max_\alpha \hat{f}_\alpha) \times (\sum_\alpha \hat{f}_\alpha^2)$$

Since $\sum_\alpha \hat{f}_\alpha^2 = \langle f, f \rangle = 1$, this expression is at most $\max_\alpha \left\{ \hat{f}_\alpha \right\}$. Hence $\max_\alpha \hat{f}_\alpha \geq 2\epsilon$ and the theorem is proved. ∎

19.3.3 Coordinate functions, Long code and its testing

Let $W \in \mathbb{N}$. We say that $f : \{\pm 1\}^W \to \{\pm 1\}$ is a *coordinate function* if there is some $w \in [W]$, such that $f(x_1, x_2, \ldots, x_W) = x_w$; in other words, $f = \chi_{\{w\}}$.

DEFINITION 19.10 (LONG CODE)
The *long code* for $[W]$ encodes each $w \in [W]$ by the table of all values of the function $\chi_{\{w\}} : \{\pm 1\}^{[W]} \to \{\pm 1\}$.

REMARK 19.11
Note that w, normally written using $\log W$ bits, is being represented using a table of 2^W bits, a doubly exponential blowup! This inefficiency is the reason for calling the code "long."

Similar to the test for the Walsh-Hadamard code, when testing the long code, we are given a function $f : \{\pm 1\}^W \to \{\pm 1\}$, and want to find out if f has good agreement with $\chi_{\{w\}}$ for some w, namely, $\hat{f}_{\{w\}}$ is significant. Such a test is described in Exercise 16 of the previous chapter, but it is not sufficient for the proof of Håstad's Theorem, which requires a test using only *three* queries. Below we show such a three query test albeit at the expense of achieving the following weaker guarantee: if the test passes with high probability then f has a good agreement with a function

χ_α with $|\alpha|$ small (but not necessarily equal to 1). This weaker conclusion will be sufficient in the proof of Theorem 19.5.

Let $\rho > 0$ be some arbitrarily small constant. The test picks two uniformly random vectors $\mathbf{x}, \mathbf{y} \in \{\pm 1\}^W$ and then a vector $\mathbf{z} \in \{\pm 1\}^{[W]}$ according to the following distribution: for every coordinate $i \in [W]$, with probability $1 - \rho$ we choose $z_i = +1$ and with probability ρ we choose $z_i = -1$. Thus with high probability, about ρ fraction of coordinates in $\mathbf{z}$ are -1 and the other $1 - \rho$ fraction are $+1$. We think of $\mathbf{z}$ as a "noise" vector. The test accepts iff $f(\mathbf{x})f(\mathbf{y}) = f(\mathbf{xyz})$. Note that the test is similar to the linearity test except for the use of the noise vector $\mathbf{z}$.

Suppose $f = \chi_{\{w\}}$. Then

$$f(\mathbf{x})f(\mathbf{y})f(\mathbf{xyz}) = x_w y_w (x_w y_w z_w) = 1 \cdot z_w$$

Hence the test accepts iff $z_w = 1$ which happens with probability $1 - \rho$. We now prove a certain converse:

LEMMA 19.12
If the test accepts with probability $1/2 + \epsilon$ then $\sum_\alpha \hat{f}_\alpha^3 (1 - 2\rho)^{|\alpha|} \geq 2\epsilon$.

PROOF: If the test accepts with probability $1/2 + \epsilon$ then $\mathsf{E}[f(\mathbf{x})f(\mathbf{y})f(\mathbf{xyz})] = 2\epsilon$. Replacing f by its Fourier expansion, we have

$$2\epsilon \leq \mathsf{E}_{\mathbf{x},\mathbf{y},\mathbf{z}} \left[(\sum_\alpha \hat{f}_\alpha \chi_\alpha(\mathbf{x})) \cdot (\sum_\beta \hat{f}_\beta \chi_\beta(\mathbf{y})) \cdot (\sum_\gamma \hat{f}_\gamma \chi_\gamma(\mathbf{xyz})) \right]$$

$$= \mathsf{E}_{\mathbf{x},\mathbf{y},\mathbf{z}} \left[\sum_{\alpha,\beta,\gamma} \hat{f}_\alpha \hat{f}_\beta \hat{f}_\gamma \chi_\alpha(\mathbf{x})\chi_\beta(\mathbf{y})\chi_\gamma(\mathbf{x})\chi_\gamma(\mathbf{y})\chi_\gamma(\mathbf{z}) \right]$$

$$= \sum_{\alpha,\beta,\gamma} \hat{f}_\alpha \hat{f}_\beta \hat{f}_\gamma \mathsf{E}_{\mathbf{x},\mathbf{y},\mathbf{z}} \left[\chi_\alpha(\mathbf{x})\chi_\beta(\mathbf{y})\chi_\gamma(\mathbf{x})\chi_\gamma(\mathbf{y})\chi_\gamma(\mathbf{z}) \right].$$

Orthonormality implies the expectation is 0 unless $\alpha = \beta = \gamma$, so this is

$$= \sum_\alpha \hat{f}_\alpha^3 \mathsf{E}_{\mathbf{z}}[\chi_\alpha(z)]$$

Now $\mathsf{E}_{\mathbf{z}}[\chi_\alpha(\mathbf{z})] = \mathsf{E}_{\mathbf{z}} \left[\prod_{w \in \alpha} z_w \right]$ which is equal to $\prod_{w \in \alpha} \mathsf{E}[z_w] = (1 - 2\rho)^{|\alpha|}$ because each coordinate of $\mathbf{z}$ is chosen independently. Hence we get that

$$2\epsilon \leq \sum_\alpha \hat{f}_\alpha^3 (1 - 2\rho)^{|\alpha|}$$

∎

The conclusion of Lemma 19.12 is reminiscent of the calculation in the proof of Theorem 19.9, except for the extra factor $(1 - 2\rho)^{|\alpha|}$. This factor depresses the contribution of $\hat{f}_\alpha$ for large α, allowing us to conclude that the small α's must contribute a lot. This formalized in the following corollary (left as Exercise 2).

COROLLARY 19.13
If f passes the long code test with probability $1/2 + \delta$ then

$$\sum_{\alpha:|\alpha|\leq k} \hat{f}_\alpha^3 \geq 2\delta - \epsilon,$$

where $k = \frac{1}{2\rho} \log \frac{1}{\epsilon}$.

19.4 Proof of Theorem 19.5

Recall that our proof of the **PCP** Theorem implies that there are constants $\gamma > 0, s \in \mathbb{N}$ such that $(1–\gamma)$-GAP 2CSP$_s$ is **NP**-hard (see Claim 18.36). This means that for every **NP**-language L we have a **PCP**-verifier for L making two queries over alphabet $\{0, \ldots, s-1\}$ with perfect completeness and soundness parameter $1-\gamma$. Furthermore this **PCP** system has the property that the verifier accepts the answer pair z_1, z_2 iff $z_2 = h_r(z_1)$ where h_r is a function (depending on the verifier's randomness r) mapping $\{0, \ldots, s-1\}$ to itself (see Exercise 3). We call this the *projection property*. Using the Raz's parallel repetition lemma (Theorem 19.3), we can reduce the soundness parameter to an arbitrary small constant at the expense of increasing the alphabet. Note that parallel repetition preserves the projection property.

Let L be an **NP**-language and $\epsilon > 0$ an arbitrarily small constant. By the above there exists a constant W and **PCP**-verifier V_{Raz} (having the projection property) that makes two queries to a polynomial-sized PCP proof π with alphabet $\{1, \ldots, W\}$ such that for every x, if $x \in L$ then there exists π such that $\Pr[V_{Raz}^\pi(x) = 1] = 1$ and if $x \notin L$ then $\Pr[V_{Raz}^\pi(x) = 1] < \epsilon$ for every π.

Now we describe Håstad's verifier V_H. It essentially follows V_{Raz}, but it expects each entry in the **PCP** proof π to be encoded using the long code. It expects these encodings to be *bifolded*, a technical property we now define and is motivated by the observation that coordinate functions satisfy $\chi_{\{w\}}(-\mathbf{u}) = -\chi_{\{w\}}(\mathbf{u})$, where $-\mathbf{u}$ is the vector $(-u_1, \ldots, -u_W)$.

DEFINITION 19.14
A function $f : \{\pm 1\}^W \to \{\pm 1\}$ is *bifolded* if for all $\mathbf{u} \in \{\pm 1\}^W$, $f(-\mathbf{u}) = -f(\mathbf{u})$.

Whenever the **PCP** proof is supposed to contain a longcode codeword then we may assume without loss of generality that the function is bifolded. The reason is that the verifier can identify, for each pair of inputs $\mathbf{u}, -\mathbf{u}$, one designated representative —say the one whose first coordinate is $+1$— and just define $f(-\mathbf{u})$ to be $-f(\mathbf{u})$. One benefit —though of no consequence in the proof— of this convention is that bifolded functions require only half as many bits to represent. We will use the following fact:

LEMMA 19.15
If $f : \{\pm 1\}^W \to \{\pm 1\}$ is bifolded and $\hat{f}_\alpha \neq 0$ then $|\alpha|$ must be an odd number (and in particular, nonzero).

PROOF: By definition,

$$\hat{f}_\alpha = \langle f, \chi_\alpha \rangle = \tfrac{1}{2^n} \sum_{\mathbf{u}} f(\mathbf{u}) \prod_{i \in \alpha} u_i.$$

If $|\alpha|$ is even then $\prod_{i \in \alpha} u_i = \prod_{i \in \alpha} (-u_i)$. So if f is bifolded, the terms corresponding to $\mathbf{u}$ and $-\mathbf{u}$ have opposite signs and the entire sum is 0. $\blacksquare$

Håstad's verifier. Recall that V_{Raz} uses its randomness to select a function two entries i, j in the table π and a function $h : [W] \rightarrow [W]$, and accepts iff $\pi(j) = h(\pi(i))$. Håstad's verifier, denoted V_H, expects the proof $\tilde{\pi}$ to consist of (bifolded) longcode encodings of each entry of π. The verifier V_H emulates V_{Raz} to pick two locations i, j in the table and a function $h : [W] \rightarrow [W]$ such that V_{Raz}'s test is to accept iff $\pi[j] = h(\pi[i])$. The proof $\tilde{\pi}$ contains in the locations i and j two functions f and g respectively (which may or may not be the longcode encoding of $\pi(i)$ and $\pi(j)$). Instead of reading the long codes f, g in their entirety, the verifier V_H performs a simple test that is reminiscent of the long code test. For a string $\mathbf{y} \in \{\pm 1\}^W$ we denote by $h^{-1}(\mathbf{y})$ the string such that for every $w \in [W]$, $h^{-1}(\mathbf{y})_w = y_{h(w)}$. In other words, for each $u \in [W]$, the bit y_u appears in all coordinates of $h^{-1}(\mathbf{y})$ that are indexed by integers in the subset $h^{-1}(u)$. This is well defined because $\{h^{-1}(u) : u \in [W]\}$ is a partition of $[W]$. V_H chooses uniformly at random $\mathbf{u}, \mathbf{y} \in \{\pm 1\}^W$ and chooses $\mathbf{z} \in \{\pm 1\}^W$ by letting $z_i = +1$ with probability $1 - \rho$ and $z_i = -1$ with probability ρ. It then accepts Iff

$$f(\mathbf{u})g(\mathbf{y}) = f(h^{-1}(\mathbf{y})\mathbf{u}\mathbf{z}) \tag{1}$$

Translating back from $\{\pm 1\}$ to $\{0, 1\}$, note that V_H's test is indeed linear, as it accepts iff $\tilde{\pi}[i_1] + \tilde{\pi}[i_2] + \tilde{\pi}[i_3] = b$ for some $i_1, i_2, i_3 \in [m2^W]$ and $b \in \{0, 1\}$. (The bit b can indeed equal 1 because of the way V_H ensures the bifolding property.)

Completeness of V_H. Suppose f, g are long codes of two integers w, u satisfying $h(w) = u$ (in other words, V_{raz} would have accepted the assignments represented by these integers). Then

$$f(\mathbf{u})g(\mathbf{y})f(h^{-1}(\mathbf{y})\mathbf{u}\mathbf{z}) = u_w y_u (h^{-1}(\mathbf{y})\mathbf{u}\mathbf{z}_w$$
$$= u_w y_u (y_{h(w)} u_w z_w) \qquad = z_w.$$

Hence V_H accepts iff $z_w = 1$, which happens with probability $1 - \rho$.

Soundness of V_H. We now show that if V_H accepts f, g with probability significantly more than $1/2$, then the Fourier transforms of f, g must be correlated. To formalize this we define for $\alpha \subseteq [W]$,

$$h_2(\alpha) = \left\{ u \in [W] : \ \left| h^{-1}(u) \cap \alpha \right| \text{ is odd} \right\}$$

Notice in particular that for *every* $u \in h_2(\alpha)$ there is at least one $w \in \alpha$ such that $h(w) = u$.

In the next Lemma δ is allowed to be negative.

LEMMA 19.16
Let $f, g : \{\pm 1\}^W \rightarrow \{\pm 1\}$, $h : [W] \rightarrow [W]$ *be bifolded functions passing V_H's test* (1) *with probability at least* $1/2 + \delta$. *Then*

$$\sum_{\alpha \subseteq [W], \alpha \neq \emptyset} \hat{f}_\alpha^2 \hat{g}_{h_2(\alpha)} (1 - 2\rho)^{|\alpha|} \geq 2\delta$$

PROOF: By hypothesis, f, g are such that $E[f(\mathbf{u})f(\mathbf{u}h^{-1}(\mathbf{y})\mathbf{z})g(\mathbf{y})] \geq 2\delta$. Replace f, g by their Fourier expansions. We get that

$$2\delta \leq = \mathsf{E}_{\mathbf{u},\mathbf{y},\mathbf{z}} \left[(\sum_{\alpha} \hat{f}_{\alpha}\chi_{\alpha}(x))(\sum_{\beta} \hat{g}_{\beta}\chi_{\beta}(\mathbf{y}))(\sum_{\gamma} \hat{f}_{\gamma}\chi_{\gamma}(\mathbf{u}h^{-1}(\mathbf{y})\mathbf{z})) \right]$$

$$= \sum_{\alpha,\beta,\gamma} \hat{f}_{\alpha}\hat{g}_{\beta}\hat{f}_{\gamma}\mathsf{E}_{\mathbf{u},\mathbf{y},\mathbf{z}} \left[\chi_{\alpha}(\mathbf{u})\chi_{\beta}(\mathbf{y})\chi_{\gamma}(\mathbf{u})\chi_{\gamma}(h^{-1}(\mathbf{y}))\chi_{\gamma}(\mathbf{z}) \right]$$

By orthonormality this simplifies to

$$= \sum_{\alpha,\beta} \hat{f}_{\alpha}^2\hat{g}_{\beta}\mathsf{E}_{\mathbf{y},\mathbf{z}} \left[\chi_{\beta}(\mathbf{y})\chi_{\alpha}(h^{-1}(\mathbf{y}))\chi_{\alpha}(\mathbf{z}) \right]$$

$$= \sum_{\alpha,\beta} \hat{f}_{\alpha}^2\hat{g}_{\beta}(1-2\rho)^{|\alpha|}\mathsf{E}_{\mathbf{y}} \left[\chi_{\alpha}(h^{-1}(\mathbf{y})\chi_{\beta}(\mathbf{y}) \right] \tag{2}$$

since $\chi_{\alpha}(\mathbf{z}) = (1-2\rho)^{|\alpha|}$, as noted in our analysis of the long code test. Now we have

$$\mathsf{E}_{\mathbf{y}}[\chi_{\alpha}(h^{-1}(\mathbf{y}))\chi_{\beta}(\mathbf{y})] = \mathsf{E}_{\mathbf{y}}[\prod_{w\in\alpha} h^{-1}(\mathbf{y})_w \prod_{u\in\beta} y_u]$$

$$= \mathsf{E}_{\mathbf{y}}[\prod_{w\in\alpha} y_{h(w)} \prod_{u\in\beta} y_u],$$

which is 1 if $h_2(\alpha) = \beta$ and 0 otherwise. Hence (2) simplifies to

$$\sum_{\alpha} \hat{f}_{\alpha}^2\hat{g}_{h_2(\alpha)}(1-2\rho)^{|\alpha|}.$$

Finally we note that since the functions are assumed to be bifolded, the Fourier coefficients $\hat{f}_{\emptyset}$ and $\hat{g}_{\emptyset}$ are zero. Thus those terms can be dropped from the summation and the Lemma is proved. ∎

The following corollary of Lemma 19.16 completes the proof of Håstad's 3-bit PCP Theorem.

COROLLARY 19.17
Let ϵ be the soundness parameter of V_{Raz}. If ρ, δ satisfy $\rho\delta^2 > \epsilon$ then the soundness parameter of V_H is at most $1/2 + \delta$.

PROOF: Suppose V_H accepts a proof $\tilde{\pi}$ with probability at least $1/2 + \delta$. We give a probabilistic construction of a proof π causing V_{Raz} to accept the same statement with probability at least $\rho\delta^2$.

Suppose that V_{Raz} uses proofs π with m entries in $[W]$. We can think of $\tilde{\pi}$ as providing, for every $i \in [m]$, a function $f_i : \{\pm 1\}^W \{\pm 1\}$. We will use $\tilde{\pi}$ to construct a proof π for V_{Raz} as follows: we first use f_i to come up with a distribution D_i over $[W]$. We then let $\pi[i]$ be a random element from D_i.

The distribution D_i**.** Let $f = f_i$. The distribution D_i is defined by first selecting $\alpha \subseteq [W]$ with probability $\hat{f}_\alpha^2$ and then selecting w at random from α. This is well defined because $\sum_\alpha \hat{f}_\alpha^2 = 1$ and (due to bifolding) $f_\emptyset = 0$.

Recall that V_{Raz} picks using its random tape a pair i, j of locations and a function $h : [W] \to [W]$ and then verifies that $\pi[j] = h(\pi[i])$. Let r be some possible random tape of V_{Raz} and let i, j, h be the pair of entries in π and function that are determined by r. We define the indicator random variable I_r to be 1 if for $w \in_R D_i$ and $u \in_R D_j$ it holds that $w = h(u)$ and to be 0 otherwise. Thus, our goal is to show that

$$\mathsf{E}_{\pi = D_1, \ldots, D_m}[\mathsf{E}_r[I_r]] \geq \rho\delta^2 \tag{3}$$

since that would imply that there *exists* a table π causing V_{Raz} to accept with probability at least $\rho\delta^2$, proving the corollary.

To prove (3) we first notice that linearity of expectation allows us to exchange the order of the two expectations and so it is enough to bound $\mathsf{E}_r[\mathsf{E}_{D_i, D_j}[I_r]]$ where i, j are the entries determined by the random tape r. For every r denote by δ_r the probability that V_H accepts $\tilde{\pi}$ when it uses r as the random tape for V_{Raz}. The acceptance probability of V_H is $\mathsf{E}_r[\frac{1}{2} + \delta_r]$ and hence $\mathsf{E}_r[\delta_r] = \delta$.

Let i, j, h be the pair and function determined by r and denote by $f = f_i$ and $g = f_j$ where f_i (resp. f_j) is the function at the ith (resp. j^{th}) entry of the table $\tilde{\pi}$. What is the chance that a pair of assignments $w \in_R D_i$ and $v \in_R D_j$ will satisfy the constraint? (i.e., will satisfy $v = h(w)$?). Recall that we pick w and u by choosing α with probability $\hat{f}_\alpha^2$, β with probability $\hat{g}_\beta^2$ and choosing $w \in_R \alpha, v \in_R \beta$. Now if $\beta = h_2(\alpha)$ then for every $v \in \beta$ there exists $w \in \alpha$ with $h(w) = v$ and hence the probability the constraint is satisfied is at least $1/|\alpha|$. Thus, we have that

$$\sum_\alpha \frac{1}{|\alpha|} \hat{f}_\alpha^2 \hat{g}_{h_2(\alpha)}^2 \leq \mathsf{E}_{D_i, D_j}[I_r] \tag{4}$$

This is similar to (but not quite the same as) the expression in Lemma 19.16, according to which

$$2\delta_r \leq \sum_\alpha \hat{f}_\alpha^2 \hat{g}_{h_2(\alpha)}(1 - 2\rho)^{|\alpha|}.$$

However, since one can easily see that $(1 - 2\rho)^{|\alpha|} \leq \dfrac{2}{\sqrt{\rho|\alpha|}}$ we have

$$2\delta_r \leq \sum_\alpha \hat{f}_\alpha^2 \left|\hat{g}_{h_2(\alpha)}\right| \frac{2}{\sqrt{\rho|\alpha|}}$$

Or

$$\delta_r\sqrt{\rho} \leq \sum_\alpha \hat{f}_\alpha^2 \left|\hat{g}_{h_2(\alpha)}\right| \frac{1}{\sqrt{|\alpha|}}$$

Applying the Cauchy-Schwartz inequality, $\sum_i a_i b_i \leq (\sum_i a_i^2)^{1/2}(\sum_i b_i^2)^{1/2}$, with $\hat{f}_\alpha \left|\hat{g}_{\pi_2(\alpha)}\right| \frac{1}{\sqrt{|\alpha|}}$ playing the role of the a_i's and $\hat{f}_\alpha$ playing that of the b_i's, we obtain

$$\delta_r\sqrt{\rho} \leq \sum_\alpha \hat{f}_\alpha^2 \left|\hat{g}_{h_2(\alpha)}\right| \frac{1}{\sqrt{|\alpha|}} \leq \left(\sum_\alpha \hat{f}_\alpha^2\right)^{1/2} \left(\sum_\alpha \hat{f}_\alpha^2 \hat{g}_{h_2(\alpha)}^2 \frac{1}{|\alpha|}\right)^{1/2} \tag{5}$$

Since $\sum_\alpha \hat{f}_\alpha^2 = 1$, by squaring (5) and combining it with (4) we get that for every r,

$$\delta_r^2 \rho \le \mathsf{E}_{D_i,D_j}[I_r]$$

taking expectation over r and using $\mathsf{E}[X]^2 \le \mathsf{E}[X^2]$ we get that

$$\delta^2 \rho = \mathsf{E}_r[\delta_r]^2 \rho \le \mathsf{E}_r[\delta_r^2]\rho \le \mathsf{E}_r[E_{D_i,D_j}[I_r]]$$

proving (3). ∎

19.5 Learning Fourier Coefficients

Suppose that you are given random access to a Boolean function $f : \{\pm1\}^n \to \{\pm1\}$ and want to find the high Fourier coefficients of f. Of course, we can compute all of the coefficients in time polynomial in 2^n, but is there a faster algorithm? By the Parseval equality (Lemma 19.7) we know that there can be at most $1/\epsilon^2$ coefficients with absolute value larger than ϵ, and so we can hope to learn these coefficients in time polynomial in n, and $1/\epsilon$. It turns out we can (almost) achieve this goal:

THEOREM 19.18 ([**?**])
There is an algorithm A that given input $n \in \mathbb{N}, \epsilon \in (0,1)$ and random access to a function $f : \{\pm1\}^n \to \{\pm1\}$, runs in $poly(n,1/\epsilon)$ time and with probability at least 0.9 outputs a set L of size at most $O(1/\epsilon^2)$ such that for every $\alpha \subseteq [n]$, if $|\hat{f}_\alpha| > \epsilon$ then $\alpha \in L$.

PROOF: We identify subsets of $[n]$ with strings in $\{0,1\}^m$ in the obvious way. For $k \le n$ and $\alpha \in \{0,1\}^k$ denote

$$\tilde{f}_{\alpha\star} = \sum_{\beta\in\{0,1\}^{n-k}} \hat{f}_{\alpha\circ\beta}^2,$$

where $\circ$ denotes concatenation. By Parseval (Lemma 19.7) $\tilde{f}_\star = 1$. Note also that for every $k < n$ and $\alpha \in \{0,1\}^k$, $\tilde{f}_{\alpha\star} = \tilde{f}_{\alpha0\star} + \tilde{f}_{\alpha1\star}$. Therefore, if we think of the full depth-n binary labeled by binary strings of length $\le n$ (with the root being the empty word and the two children of α are $\alpha0$ and $\alpha1$), then at any level of this tree there can be at most $1/\epsilon^2$ strings α such that $\tilde{f}_{\alpha\star} > \epsilon^2$ (the kth level of the tree corresponds to all strings of length k). Note that if a string α satisfies $\tilde{f}_{\alpha\star} < \epsilon^2$ then the same holds for every string of the form $\alpha \circ \beta$. Our goal will be to find all these strings at all levels, and then output all the strings that label leaves in the tree (i.e., all n-bit strings).

The heart of the algorithm is a procedure Estimate that given α and oracle access to $f(\cdot)$, outputs an estimate of f_α within $\epsilon/4$ accuracy with probability $1 - \frac{\epsilon^2}{100n}$. Using this procedure we work our way from the root down, and whenever Estimate(α) gives a value smaller than $\epsilon/2$ we "kill" this node and will not deal with it and its subnodes. Note that unless the output of Estimate is more than $\epsilon/4$-far from the real value (which we will ensure by the union bound happens with probability less than 0.1 over all the levels) at most $4/\epsilon$ nodes will survive at any level. The algorithm will output the $4/\epsilon$ leaves that survive.

The procedure Estimate uses the following claim:

CLAIM 19.19
For every α,

$$\tilde{f}_{\alpha\star} = \mathsf{E}_{\mathbf{x},\mathbf{x}'\in_R\{0,1\}^k,\mathbf{y}\in_R\{0,1\}^{n-k}}[f(\mathbf{x}\circ\mathbf{y})f(\mathbf{x}'\circ\mathbf{y})\chi_\alpha(\mathbf{x})\chi_\alpha(\mathbf{x}')]$$

PROOF: We start with the case that $\alpha = 0^k$. To get some intuition, suppose that $\tilde{f}_{0^k\star} = 1$. This means that f can be expressed as a sum of functions of the form $\chi_{0^k\circ\beta}$ and hence it does not depend on its first k variables. Thus $f(\mathbf{x}\circ\mathbf{y}) = f(\mathbf{x}'\circ\mathbf{y})$ and we'll get that $\mathsf{E}[f(\mathbf{x}\circ\mathbf{y})f(\mathbf{x}'\circ\mathbf{y})] = \mathsf{E}[f(\mathbf{z})^2] = 1$. More generally, if $\tilde{f}_{0^k\star}$ is large then that means that in the Fourier representation, the weight of functions not depending on the first k variables is large and hence we expect large correlation between $f(\mathbf{x}'\circ\mathbf{y})$ and $f(\mathbf{x}\circ\mathbf{y})$. This is verified by the following calculations:

$$2^{-n-k}\sum_{\mathbf{x},\mathbf{x}',\mathbf{y}} f(\mathbf{x}\circ\mathbf{y})f(\mathbf{x}'\circ\mathbf{y}) \underset{\text{basis change}}{=}$$

$$2^{-n-k}\sum_{\mathbf{x},\mathbf{x}',\mathbf{y}}\left(\sum_{\gamma\circ\beta}\hat{f}(\gamma\circ\beta)\chi_{\gamma\circ\beta}(\mathbf{x}\circ\mathbf{y})\right)\left(\sum_{\gamma'\circ\beta'}\hat{f}(\gamma'\circ\beta')\chi_{\gamma'\circ\beta'}(\mathbf{x}'\circ\mathbf{y})\right)\underset{\chi_{\gamma\circ\beta}(\mathbf{x}\circ\mathbf{y})=\chi_\gamma(\mathbf{x})\chi_\beta(\mathbf{y})}{=}$$

$$2^{-n-k}\sum_{\mathbf{x},\mathbf{x}',\mathbf{y}}\left(\sum_{\gamma\circ\beta}\hat{f}(\gamma\circ\beta)\chi_\gamma(\mathbf{x})\chi_\beta(\mathbf{y})\right)\left(\sum_{\gamma'\circ\beta'}\hat{f}(\gamma'\circ\beta')\chi_{\gamma'}(\mathbf{x}')\chi_{\beta'}(\mathbf{y})\right)\underset{\text{reordering terms}}{=}$$

$$\sum_{\gamma,\beta,\gamma',\beta'}\hat{f}(\gamma\beta)\hat{f}(\gamma'\beta')2^{-k}\left(\sum_{\mathbf{x}}\chi_{\gamma'}(\mathbf{x})\right)2^{-k}\left(\sum_{\mathbf{x}'}\chi_\gamma(\mathbf{x}')\right)2^{-(n-k)}\left(\sum_{\mathbf{y}}\chi_\beta(\mathbf{y})\chi_{\beta'}(\mathbf{y})\right)\underset{\Sigma\chi_\gamma(\mathbf{x})=0\text{ for }\gamma\neq0^k}{=}$$

$$\sum_{\beta,\beta'}\hat{f}(0^k\circ\beta)\hat{f}(0^k\circ\beta')\delta_{\beta,\beta'} = \sum_\beta\hat{f}(0^k\circ\beta)^2 = \tilde{f}_{0^k\star}$$

For the case $\alpha \neq 0^k$, we essentially add these factors to translate it to the case $\alpha = 0^k$. Indeed one can verify that if we define $g(\mathbf{x}\circ\mathbf{y}) = f(\mathbf{x}\circ\mathbf{y})\chi_\alpha(\mathbf{x})$ then for every $\beta \in \{0,1\}^{n-k}$. $g_{0^k\circ\beta} = f_{\alpha\circ\beta}$.
■

By the Chernoff bound, we can estimate the expectation of Claim 19.19 (and hence $\tilde{f}_{\alpha\star}$) using repeated sampling, thus obtaining the procedure `Estimate` and completing the proof. ■

19.6   Other PCP Theorems: A Survey

The following variants of the **PCP** Theorem have been obtained and used for various applications.

19.6.1   PCP's with sub-constant soundness parameter.

Because ℓ-times parallel repetition transforms a proof of size m to a proof of size m^ℓ, we cannot use it with ℓ larger than a constant and still have a polynomial-sized proof. Fortunately, there have been direct constructions of **PCP**'s achieving low soundness using larger alphabet size, but without increasing the proof's size. Raz and Safra [**?**] show that there is an absolute constant q such that

for every $W \leq \sqrt{\log n}$, every **NP** language has a q-query verifier over alphabet $\{0, \ldots, W-1\}$ that uses $O(\log n)$ random bits, and has soundness $2^{-\Omega(\log W)}$.

19.6.2   Amortized query complexity.

Some applications require binary-alphabet **PCP** systems enjoying a tight relation between the number of queries (that can be an arbitrarily large constant) and the soundness parameter. The relevant parameter here turns out to be the *free bit complexity* [**?**, **?**]. This parameter is defined as follows. Suppose the number of queries is q. After the verifier has picked its random string, and picked a sequence of q addresses, there are 2^q possible sequences of bits that could be contained in those addresses. If the verifier accepts for only t of those sequences, then we say that the free bit parameter is $\log t$ (note that this number need not be an integer). In fact, for most applications it suffices to consider the *amortized free bit complexity* [**?**]. This parameter is defined as $\lim_{s \to 0} f_s / \log(1/s)$, where f_s is the number of free bits needed by the verifier to ensure the soundness parameter is at most s. Håstad constructed systems with amortized free bit complexity tending to zero [**?**]. That is, for every $\epsilon > 0$, he gave a **PCP**-verifier for **NP** that uses $O(\log n)$ random bits and ϵ amortized free bits. He then used this **PCP** system to show (using tools from [**?**, **?**, **?**]) that MAX INDSET (and so, equivalently, MAX CLIQUE) is **NP**-hard to approximate within a factor of $n^{1-\epsilon}$ for arbitrarily small $\epsilon > 0$.

19.6.3   Unique games.

Exercises

§1 Prove that there is a polynomial-time algorithm that given a satisfiable $2\mathsf{CSP}_W$ instance φ over $\{0..W{-}1\}$ where all the constraints are permutations (i.e, φ_i checks that $u_{j'} = h(u_j)$ for some $j, j' \in [n]$ and permutation $h : \{0..W{-}1\} \to \{0..W{-}1\}$) finds a satisfying assignment $\mathbf{u}$ for φ.

§2 Prove Corollary 19.13.

§3 Prove that the **PCP** system resulting from the proof of Claim 18.36 (Chapter 18) satisfies the projection property.

§4 Let $f : \{\pm 1\}^n \to \{\pm 1\}$ and let $I \subseteq [n]$. Let M_I be the following distribution: we choose $z \in_R M_I$ by for $i \in I$, choose z_i to be $+1$ with probability $1/2$ and -1 with probability $1/2$ (independently of other choices), for $i \notin I$ choose $z_i = +1$. We define the *variation of f on I* to be $\mathrm{Pr}_{\mathbf{x} \in_R \{\pm 1\}^n, \mathbf{z} \in_R M_I}[f(\mathbf{x}) \neq f(\mathbf{xz})]$.

Suppose that the variation of f on I is less than ϵ. Prove that there exists a function $g : \{\pm 1\}^n \to \mathbb{R}$ such that **(1)** g does not depend on the coordinates in I and **(2)** g is 10ϵ-close to f (i.e., $\mathrm{Pr}_{\mathbf{x} \in_R \{\pm 1\}^n}[f(\mathbf{x}) \neq g(\mathbf{x})] < 10\epsilon$). Can you come up with such a g that outputs values in $\{\pm 1\}$ only?

§5 For $f : \{\pm 1\}^n \to \{\pm 1\}$ and $\mathbf{x} \in \{\pm 1\}^n$ we define $N_f(\mathbf{x})$ to be the number of coordinates i such that if we let y to be $\mathbf{x}$ flipped at the ith coordinate (i.e., $y = x\mathbf{e}^i$ where $\mathbf{e}^i$ has -1 in the

i^{th} coordinate and $+1$ everywhere else) then $f(\mathbf{x}) \neq f(\mathbf{y})$. We define the *average sensitivity* of f, denoted by $as(f)$ to be the expectation of $N_f(\mathbf{x})$ for $\mathbf{x} \in_R \{\pm 1\}^n$.

(a) Prove that for every balanced function $f : \{\pm 1\}^n \to \{\pm 1\}$ (i.e., $\Pr[f(\mathbf{x}) = +1] = 1/2$), $as(f) \geq 1$.

(b) Let f be balanced function from $\{\pm 1\}^n$ to $\{\pm 1\}$ with $as(f) = 1$. Prove that f is a coordinate function or its negation (i.e., $f(\mathbf{x}) = x_i$ or $f(\mathbf{x}) = -x_i$ for some $i \in [n]$ and for every $\mathbf{x} \in \{\pm 1\}^n$).

Chapter 20

Quantum Computation

> *"Turning to quantum mechanics.... secret, secret, close the doors! we always have had a great deal of difficulty in understanding the world view that quantum mechanics represents ... It has not yet become obvious to me that there's no real problem. I cannot define the real problem, therefore I suspect there's no real problem, but I'm not sure there's no real problem. So that's why I like to investigate things."*
> Richard Feynman 1964

> *"The only difference between a probabilistic classical world and the equations of the quantum world is that somehow or other it appears as if the probabilities would have to go negative.."*
> Richard Feynman, in "Simulating physics with computers", 1982

Quantum computers are a new computational model that may be physically realizable and may have an exponential advantage over 'classical' computational models such as probabilistic and deterministic Turing machines. In this chapter we survey the basic principles of quantum computation and some of the important algorithms in this model.

As complexity theorists, the main reason to study quantum computers is that they pose a serious challenge to the *strong Church-Turing thesis* that stipulates that any physically reasonable computation device can be simulated by a Turing machine with polynomial overhead. Quantum computers seem to violate no fundamental laws of physics and yet currently we do not know any such simulation. In fact, there is some evidence to the contrary: as we will see in Section 20.7, there is a polynomial-time algorithm for quantum computers to factor integers, where despite much effort no such algorithm is known for deterministic or probabilistic Turing machines. In fact, the conjectured hardness of this problem underlies of several cryptographic schemes (such as the RSA cryptosystem) that are currently widely used for electronic commerce and other applications. Physicists are also interested in quantum computers as studying them may shed light on quantum mechanics, a theory which, despite its great success in predicting experiments, is still not fully understood.

This chapter utilizes some basic facts of linear algebra, and the space $\mathbb{C}^n$. These are reviewed in Appendix A. See also Note 20.8 for a quick reminder of our notations.

20.1   Quantum weirdness

It is beyond this book (and its authors) to fully survey quantum mechanics. Fortunately, only very little physics is needed to understand the main results of quantum computing. However, these results do use some of the more counterintuitive notions of quantum mechanics such as the following:

> Any object in the universe, whether it is a particle or a cat, does not have definite properties (such as location, state, etc..) but rather has a kind of *probability wave* over its potential properties. The object only achieves a definite property when it is *observed*, at which point we say that the probability wave *collapses* to a single value.

At first this may seem like philosophical pontification analogous to questions such as "if a tree falls and no one hears, does it make a sound?" but these probability waves are in fact very real, in the sense that they can interact and interfere with one another, creating experimentally measurable effects. Below we describe two of the experiments that led physicists to accept this counterintuitive theory.

20.1.1   The 2-slit experiment

In the 2-slit experiment a source that fires electrons one by one (say, at the rate of one electron per second) is placed in front of a wall containing two tiny slits (see Figure **??**). Beyond the wall we place an array of detectors that light up whenever an electron hits them. We measure the number of times each detector lights up during an hour.

When we cover one of the slits, we would expect that the detectors that are directly behind the open slit will receive the largest number of hits, and as Figure **??** shows, this is indeed the case. When both slits are uncovered we expect that the number of times each detector is hit is the sum of the number of times it is hit when the first slit is open and the number of times it is hit when the second slit is open. In particular, uncovering both slits should only *increase* the number of times each location is hit.

Surprisingly, this is not what happens. The pattern of hits exhibits the "interference" phenomena depicted in Figure **??**. In particular, at several detectors the total electron flux is *lower* when both slits are open as compared to when a single slit is open. This defies explanation if electrons behave as particles or "little balls".

According to Quantum mechanics, the explanation is that it is wrong to think of an electron has a "little ball" that can either go through the first slit or the second (i.e., has a definite property). Rather, somehow the electron instantaneously explores all possible paths to the detectors (and so "finds out" how many slits are open) and then decides on a distribution among the possible paths that it will take.

You might be curious to see this "path exploration" in action, and so place a detector at each slit that light up whenever an electron passes through that slit. When this is done, one can see that every electron passes through only one of the slits like a nice little ball. But furthermore, the interference phenomenon disappears and the graph of electron hits becomes, as originally expected, the sum of the hits when each slit is open. The explanation is that, as stated above, *observing* an

---

NOTE 20.1 (PHYSICALLY IMPLEMENTING QUANTUM COMPUTERS.)

---

object "collapses" their distribution of possibilities, and so changes the result of the experiment. (One moral to draw from this is that quantum computers, if they are ever built, will have to be carefully isolated from external influences and noise, since noise may be viewed as a "measurement" performed by the environment on the system. Of course, we can never completely isolate the system, which means we have to make quantum computation tolerant of a little noise. This seems to be possible under some noise models, see the chapter notes.)

20.1.2 Quantum entanglement and the Bell inequalities.

Even after seeing the results of the 2-slit experiment, you might still be quite skeptical of the explanation that quantum mechanics offers. If you do, you are in excellent company. Albert Einstein didn't buy it either. While he agreed that the 2-slit experiment means that electrons are not exactly "little balls", he didn't think that it is sufficient reason to give up such basic notions of physics such as the existence of an independent reality, with objects having definite properties that do not depend on whether one is observing them. To show the dangerous outcomes of giving up such notions, in a 1951 paper with Podosky and Rosen (EPR for short) he described a thought experiment showing that accepting Quantum mechanics leads to the seemingly completely ridiculous conclusion that systems in two far corners of the universe can instantaneously coordinate their actions.

In 1964 John Bell showed how the principles behind EPR thought experiment can be turned into an *actual* experiment. In the years since, Bell's experiment has been repeated again and again with the same results: quantum mechanics' predictions were verified and, contrary to Einstein's expectations, the experiments refuted his intuitions about how the universe operates. In an interesting twist, in recent years the ideas behind EPR's and Bell's experiments were used for a practical goal: encryption schemes whose security depends only on the principles of quantum mechanics, rather than any unproven conjectures such as $\mathbf{P} \neq \mathbf{NP}$.

For complexity theorists, probably the best way to understand Bell's experiment is as a *two prover game*. Recall that in the two prover setting, two provers are allowed to decide on a strategy and then interact separately with a polynomial-time verifier which then decides whether to accept or reject the interaction (see Chapters 8 and 18). The provers' strategy can involve arbitrary computation and even be randomized, with the only constraint being that the provers are not allowed to communicate during their interaction with the verifier.

Bell's game.   In Bell's setting, we have an extremely simple interaction between the verifier and two provers (that we'll name Alice and Bob): there is no statement that is being proven, and all the communication involves the verifier sending and receiving one bit from each prover. The protocol is as follows:

1. The verifier chooses two random bits $x, y \in_R \{0, 1\}$.

2. It sends x to Alice and y to Bob.

3. Let a denote Alice's answer and b Bob's answer.

4. The verifier accepts if and only if $a \oplus b = x \wedge y$.

It is easy for Alice and Bob to ensure the verifier accepts with probability $3/4$ (e.g., by always sending $a = b = 0$). It turns out this is the best they can do:

THEOREM 20.2 ([BEL64])
Regardless of the strategy the provers use, the verifier will accept with probability at most $3/4$.

PROOF: Assume for the sake of contradiction that there is a (possibly probabilistic) strategy that causes the verifier to accept with probability more than $3/4$. By a standard averaging argument there is a fixed set of provers' coins (and hence a deterministic strategy) that causes the verifier to accept with at least the same probability, and hence we may assume without loss of generality that the provers' strategy is deterministic.

A deterministic strategy for the two provers is a pair of functions $f, g : \{0, 1\} \to \{0, 1\}$ such as the provers' answers a, b are computed as $a = f(x)$ and $b = g(y)$. The function f can be one of only four possible functions: it can be either the constant function zero or one, the function $f(x) = x$ or the function $f(y) = 1 - y$. We analyze the case that $f(x) = x$; the other case are similar.

If $f(x) = x$ then the verifier accepts iff $b = (x \wedge y) \oplus x$. On input y, Bob needs to find b that makes the verifier accept. If $y = 1$ then $x \wedge y = x$ and hence $b = 0$ will ensure the verifier accepts with probability 1. However, if $y = 0$ then $(x \wedge y) \oplus x = x$ and since Bob does not know x, the probability that his output b is equal to x is at most $1/2$. Thus the total acceptance probability is at most $3/4$. ∎

What does this game have to do with quantum mechanics? The main point is that according to "classical" pre-quantum physics, it is possible to ensure that Alice and Bob are isolated from one another. Suppose that you are given a pair of boxes that implement some arbitrary strategy for Bell's game. How can you ensure that these boxes don't have some secret communication mechanism that allows them to coordinate their answers? We might try to enclose the devices in lead boxes, but even this does not ensure complete isolation. However, Einstein's theory of relativity allows us a foolproof way to ensure complete isolation: place the two devices very far apart (say at a distance of a 1000 miles from one another), and perform the interaction with each prover at a breakneck speed: toss each of the coins x, y and demand the answer within less than one millisecond. Since according to the theory of relativity, nothing travels faster than light (that only covers about 200 miles in a millisecond), there is no way for the provers to communicate and coordinate their answers, no matter what is inside the box.

The upshot is that if someone provides you with such devices that consistently succeed in this experiment with more than $3/4 = 0.75$ probability, then she has refuted Einstein's theory. As we will see later in Section 20.3.2, quantum mechanics, with its instantaneous effects of measurements, can be used to actually build devices that succeed in this game with probability at least 0.8 (there are other games with more dramatic differences of probabilities) and this has been experimentally demonstrated.

20.2 A new view of probabilistic computation.

To understand quantum computation, it is helpful to first take a different viewpoint of a process we are already familiar with: probabilistic computation.

Suppose that we have an m-bit register. Normally, we think of such a register as having some definite value $x \in \{0,1\}^m$. However, in the context of probabilistic computation, we can also think of the register's state as actually being a *probability distribution* over its possible values. That is, we think of the register's state as being described by a 2^m-dimensional vector $\mathbf{v} = \langle \mathbf{v}_{0^m}, \mathbf{v}_{0^{m-1}1}, \ldots, \mathbf{v}_{1^m} \rangle$, where for every $x \in \{0,1\}^m$, $\mathbf{v_x} \in [0,1]$ and $\sum_x \mathbf{v}_x = 1$. When we read, or *measure*, the register, we will obtain the value x with probability $\mathbf{v}_x$.

For every $x \in \{0,1\}^n$, we denote by $|x\rangle$ the vector that corresponds to the degenerate distribution that takes the value x with probability 1. That is, $|x\rangle$ is the 2^m-dimensional vector that has zeroes in all the coordinates except a single 1 in the xth coordinate. Note that $\mathbf{v} = \sum_{x \in \{0,1\}^m} \mathbf{v}_x |x\rangle$. (We think of all these vectors as column vectors in the space $\mathbb{R}^m$.)

EXAMPLE 20.3
If a 1-bit register's state is the distribution that takes the value 0 with probability p and 1 with probability $1 - p$, then we describe the state as the vector $p|0\rangle + (1-p)|1\rangle$.

The uniform distribution over the possible values of a 2-bit register is represented by $1/4\,(|00\rangle + |01\rangle + |10\rangle + |11)$ The distribution that is uniform on every individual bit, but always satisfies that the two bits are equal is represented by $1/2\,(|00\rangle + |11\rangle)$.

An *probabilistic operation* on the register involves reading its value, and, based on the value read, modifying it in some deterministic or probabilistic way. If F is some probabilistic operation, then we can think of F as a function from $\mathbb{R}^{2^m}$ to $\mathbb{R}^{2^m}$ that maps the previous state of the register to its new state after the operation is performed. There are certain properties that *every* such operation must satisfy:

- Since F depends only on the contents of the register, and not on the overall distribution, for every $\mathbf{v}$, $F(\mathbf{v}) = \sum_{x \in \{0,1\}^m} \mathbf{v}_x F(|v\rangle)$. That is, F is a *linear* function. (Note that this means that F can be described by a $2^n \times 2^n$ matrix.)

- If $\mathbf{v}$ is a distribution vector (i.e., a vector of non-negative entries that sum up to one), then so is $F(\mathbf{v})$. That is, F is a *stochastic* function. (Note that this means that viewed as a matrix, F has non-negative entries with each column summing up to 1.)

EXAMPLE 20.4
The operation that, regardless of the register's value, writes into it a uniformly chosen random string, is described by the function F such that $F(|x\rangle) = 2^{-m} \sum_{x \in \{0,1\}^m} |x\rangle$ for every $x \in \{0,1\}^m$. (Because the set $\{|x\rangle\}_{x \in \{0,1\}^n}$ is a *basis* for $\mathbb{R}^{2^m}$, a linear function is completely determined by its output on these vectors.)

The operation that flips the first bit of the register is described by the function F such that $F(|x_1 \ldots x_m\rangle) = |(1-x_1)x_2 \ldots x_m\rangle$ for every $x_1, \ldots, x_m \in \{0, 1\}$.

Of course there are many probabilistic operations that cannot be efficiently computed, but there are very simple operations that can certainly be computed. An operation F is *elementary* if it only reads and modifies at most three bits of the register, leaving the rest of the register untouched. That is, there is some operation $G : \mathbb{R}^{2^3} \to \mathbb{R}^{2^3}$ and three indices $j, k, \ell \in [m]$ such that for every $x_1, \ldots, x_m \in \{0, 1\}$, $F(|x_1 \ldots x_m\rangle) = |y_1 \ldots y_m\rangle$ where $|y_j y_k y_\ell\rangle = G(|x_j x_k x_\ell\rangle)$ and $y_i = x_i$ for every $i \notin \{j, k, \ell\}$. Note that such an operation can be represented by a $2^3 \times 2^3$ matrix and three indices in $[m]$.

EXAMPLE 20.5

Here are some examples for operations depending on at most three bits:

For example, if we apply the coin tossing operation to the second bit of the register, then this means that for every $z = z_1 \ldots z_m \in \{0, 1\}^m$, the vector $|z\rangle$ is mapped to $\tfrac{1}{2}|z_1 0 z_3 \ldots z_m\rangle + \tfrac{1}{2}|z_1 1 z_3 \ldots z_m\rangle$.

We define a probabilistic computation to be a sequence of such elementary operations applied one after the other (see Definition 20.6 below). We will later see this corresponds exactly to our previous definition of probabilistic computation as in the class **BPP** defined in Chapter 7.

DEFINITION 20.6 (PROBABILISTIC COMPUTATION)
Let $f : \{0, 1\}^* \to \{0, 1\}^*$ and $T : \mathbb{N} \to \mathbb{N}$ be some functions. We say that f is *computable in probabilistic $T(n)$-time* if for every $n \in \mathbb{N}$ and $x \in \{0, 1\}^n$, $f(x)$ can be computed by the following process:

1. Initialize an m bit register to the state $|x0^{n-m}\rangle$ (i.e., x padded with zeroes), where $m \le T(n)$.

2. Apply one after the other $T(n)$ elementary operations $F_1, \ldots, F_T$ to the register (where we require that there is a polynomial-time TM that on input $1^n, 1^{T(n)}$ outputs the descriptions of $F_1, \ldots, F_T$).

3. Measure the register and let Y denote the obtained value. (That is, if $\mathbf{v}$ is the final state of the register, then Y is a random variable that takes the value y with probability $\mathbf{v}_y$ for every $y \in \{0, 1\}^n$.)

Denoting $\ell = |f(x)|$, we require that the first ℓ bits of Y are equal to $f(x)$ with probability at least $2/3$.

PROBABILISTIC COMPUTATION: SUMMARY OF NOTATIONS.

The *state* of an m-bit register is represented by a vector $\mathbf{v} \in \mathbb{R}^{2^m}$ such that the register takes the value x with probability $\mathbf{v}_x$.

An *operation* on the register is a function $F : \mathbb{R}^{2^m} \to \mathbb{R}^{2^m}$ that is linear and stochastic.

An *elementary operation* only reads and modifies at most three bits of the register.

A *computation* of a function f on input $x \in \{0,1\}^n$ involves initializing the register to the state $|x0^{m-n}\rangle$, applying a sequence of elementary operations to it, and then measuring its value.

Now, as promised, we show that our new notion of probabilistic computation is equivalent to the one encountered in Chapter 7.

THEOREM 20.7
*A Boolean function $f : \{0,1\}^* \to \{0,1\}$ is in **BPP** iff it is computable in probabilistic $p(n)$-time for some polynomial $p : \mathbb{N} \to \mathbb{N}$.*

PROOF: ($\Rightarrow$) Suppose that $f \in$ **BPP**. As we saw before (e.g., in the proof of Theorem 6.7) this means that f can be computed by a polynomial-sized Boolean circuit C (that can be found by a deterministic poly-time TM) if we allow the circuit C access to random coins. Thus we can compute f as follows: we will use a register of $n + r + s$ bits, where r is the number of random coins C uses, and s is the number of gates C uses. That is, we have a location in the register for every coin and every gate of C. The elementary coin tossing operation (see Example 20.5) can transform a location initialized to 0 into a random coin. In addition, we have an elementary operation that transforms three bits x, y and z into $x, y, x \wedge y$ and can similarly define elementary operations for the OR and NOT functions. Thus, we can use these operations to ensure that for every gate of C, the corresponding location in the register contains the result of applying this gate when the circuit is evaluated on input x.

($\Leftarrow$) We will show a probabilistic polynomial-time algorithm to execute an elementary operation on a register. To simulate a $p(n)$-time probabilistic computation we can execute this algorithm $p(n)$ times one after the other. For concreteness, suppose that we need to execute an operation on the first three bits of the register, that is specified by an 8×8 matrix A. The algorithm will read the three bits to obtain the value $z \in \{0,1\}^3$, and then write to them a value chosen according to the distribution specified in the zth column of A.

The only issue remaining is how to pick a value from an arbitrary distribution $(p_1, \ldots, p_8)$ over $\{0,1\}^3$ (which we identify with the set $[8]$). One case is simple: suppose that for every $i \in [8]$, $p_i = K_i/2^\ell$ where ℓ is polynomial in ℓ and $K_1, \ldots, K_8 \in [2^\ell]$. In this case, the algorithm will choose using ℓ random coins a number X between 1 and 2^ℓ and output the largest i such that $X \leq \sum_{j=1}^i K_i$.

However, this essentially captures general case as well: every number $p \in [0,1]$ can be approximated by a number of the form $K/2^\ell$ within $2^{-\ell}$ accuracy. This means that every general

---

NOTE 20.8 (A FEW NOTIONS FROM LINEAR ALGEBRA)
We use in this chapter several elementary facts and notations involving the space $\mathbb{C}^M$. These are reviewed in Appendix A, but here is a quick reminder:

- If $z = a + ib$ is a complex number (where $i = \sqrt{-1}$), then $\overline{z} = a - ib$ denotes the *complex conjugate* of z. Note that $z\overline{z} = a^2 + b^2 = |z|^2$.

- The *inner product* of two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{C}^m$, denoted by $\langle \mathbf{u}, \mathbf{v} \rangle$, is equal to $\sum_{x \in [M]} \mathbf{u}_x \overline{\mathbf{v}}_x$.

- The *norm* of a vector $\mathbf{u}$, denoted by $\|\mathbf{u}\|_2$, is equal to $\sqrt{\langle \mathbf{u}, \mathbf{u} \rangle} = \sqrt{\sum_{x \in [M]} |\mathbf{u}_x|^2}$.

- If $\langle \mathbf{u}, \mathbf{v} \rangle = 0$ we say that $\mathbf{u}$ and $\mathbf{v}$ are *orthogonal*. More generally, $\odot \mathbf{u}, \mathbf{v} = \cos\theta \|\mathbf{u}\|_2 \|\mathbf{v}\|_2$, where θ is the angle between the two vectors $\mathbf{u}$ and $\mathbf{v}$.

- If A is an $M \times M$ matrix, then $A^\dagger$ denotes the *conjugate transpose* of A. That is, $A^\dagger_{x,y} = \overline{A}_{y,x}$ for every $x, y \in [M]$.

- An $M \times M$ matrix A is *unitary* if $AA^\dagger = I$, where I is the $M \times M$ identity matrix.

Note that if z is a *real* number (i.e., z has no imaginary component) then $\overline{z} = z$. Hence, if all vectors and matrices involved are real then the inner product is equal to the standard inner product of $\mathbb{R}^n$ and the conjugate transpose operation is equal to the standard transpose operation. Also a real matrix is unitary if and only if it is symmetric.

---

distribution can be well approximated by a distribution over the form above, and so by choosing a good enough approximation, we can simulate the probabilistic computation by a **BPP** algorithm. ∎

20.3 Quantum superposition and the class BQP

A *quantum register* is also composed of m bits, but in quantum parlance they are called "qubits". In principle such a register can be implemented by any collection of m physical systems that can have an ON and OFF states, although in practice there are significant challenges for such an implementation (see Note 20.1). According to quantum mechanics, the state of such a register can be described by a 2^m-dimensional vector that, unlike the probabilistic case, can actually contain negative and even complex numbers. That is, the state of the register is described by a vector $\mathbf{v} \in \mathbb{C}^{2^m}$. Once again, we denote $\mathbf{v} = \sum_{x \in \{0,1\}^n} \mathbf{v}_x |x\rangle$ (where again $|x\rangle$ is the column vector that has all zeroes and a single one in the xth coordinate). However, according to quantum mechanics,

when the register is *measured*, the probability that we see the value x is given not by $\mathbf{v}_x$ but by $|\mathbf{v}_x|^2$. This means that $\mathbf{v}$ has to be a *unit vector*, satisfying $\sum_x |\mathbf{v}_x|^2 = 1$ (see Note 20.8).

EXAMPLE 20.9
The following are two legitimate state vectors for a two-qubit quantum register: $\frac{1}{\sqrt{2}} |0\rangle + \frac{1}{\sqrt{2}} |1\rangle$ and $\frac{1}{\sqrt{2}} |0\rangle - \frac{1}{\sqrt{2}} |1\rangle$. Even though in both cases, if the register is measured it will contain either 0 or 1 with probability $1/2$, these are considered distinct states and we will see that it is possible to differentiate between them using quantum operations.

Because states are always unit vectors, we often drop the normalization factor and so, say, use $|0\rangle - |1\rangle$ to denote the state $\frac{1}{\sqrt{2}} |0\rangle - \frac{1}{\sqrt{2}} |1\rangle$.

We call the state where all coefficients are equal the *uniform* state. For example, the uniform state for a 4-qubit register is

$$|00\rangle + |01\rangle + |10\rangle + |11\rangle,$$

(where we dropped the normalization factor of $\frac{1}{2}$.) We will also use the notation $|x\rangle |y\rangle$ to denote the standard basis vector $|xy\rangle$. It is easily verified that this operation respects the distributive law, and so we can also write the uniform state of a 4-qubit register as

$$(|0\rangle + |1\rangle)(|0\rangle + |1\rangle)$$

Once again, we can view an *operation* applied to the register as a function F that maps its previous state to the new state. That is, F is a function from $\mathbb{C}^{2^m}$ to $\mathbb{C}^{2^m}$. According to quantum mechanics, such an operation must satisfy the following conditions:

1. F is a linear function. That is, for every $\mathbf{v} \in \mathbb{C}^{2^n}$, $F(\mathbf{v}) = \sum_x \mathbf{v}_x F(|x\rangle)$.

2. F maps unit vectors to unit vectors. That is, for every $\mathbf{v}$ with $\|\mathbf{v}\|_2 = 1$, $\|F(\mathbf{v})\|_2 = 1$.

Together, these two conditions imply that F can be described by a $2^m \times 2^m$ *unitary* matrix. That is, a matrix A satisfying $AA^\dagger = I$ (see Note 20.8). We recall the following simple facts about unitary matrices (left as Exercise 1):

CLAIM 20.11
For every $M \times M$ complex matrix A, the following conditions are equivalent:

1. *A is unitary (i.e., $AA^\dagger = I$).*

2. *For every vector $\mathbf{v} \in \mathbb{C}^M$, $\|A\mathbf{v}\|_2 = \|\mathbf{v}\|_2$.*

3. *For every orthonormal basis $\{\mathbf{v}^i\}_{i \in [M]}$ of $\mathbb{C}^M$ (see below), the set $\{A\mathbf{v}^i\}_{i \in [M]}$ is an orthonormal basis of $\mathbb{C}^M$.*

4. *The columns of A form an orthonormal basis of $\mathbb{C}^M$.*

NOTE 20.10 (THE GEOMETRY OF QUANTUM STATES)
It is often helpful to think of quantum states geometrically as vectors in space. For example, consider a single qubit register, in which case the state is a unit vector in the two-dimensional plane spanned by the orthogonal vectors $|0\rangle$ and $|1\rangle$. For example, the state $\mathbf{v} = \cos\theta\,|0\rangle + \sin\theta\,|1\rangle$ corresponds to a vector making an angle θ with the $|0\rangle$ vector and an angle $\pi/2 - \theta$ with the $|1\rangle$ vector. When $\mathbf{v}$ is measured it will yield 0 with probability $\cos^2\theta$ and 1 with probability $\sin^2\theta$.
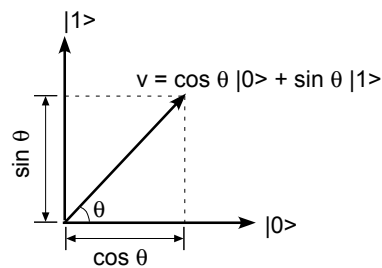
Although it's harder to visualize states with complex coefficients or more than one qubit, geometric intuition can still be useful when reasoning about such states.

5. *The rows of A form an orthonormal basis of $\mathbb{C}^M$.*

(Recall that a set $\left\{\mathbf{v}^i\right\}_{i\in[M]}$ of vectors in $\mathbb{C}^M$ is an *orthonormal basis* of $\mathbb{C}^M$ if for every $i, j \in [M]$, $\langle\mathbf{v}^i, \mathbf{v}^j\rangle$ is equal to 1 if $i = j$ and equal to 0 if $i \neq j$, where $\langle\mathbf{v}, \mathbf{u}\rangle$ is the standard inner product over $\mathbb{C}^M$. That is, $\langle\mathbf{v}, \mathbf{u}\rangle = \sum_{j=1}^{M}\mathbf{v}_i\overline{\mathbf{u}}_i$.)

As before, we define an *elementary quantum operation* to be an operation that only depends and modifies at most three qubits of the register.

EXAMPLE 20.12

Here are some examples for quantum operations depending on at most three qubits. (Because all the quantum operations are linear, it suffices to describe their behavior on any linear basis for the space $\mathbb{C}^{2^m}$ and so we often specify quantum operations by the way they map the standard basis.)

- The standard NOT operation on a single bit can be thought of as the unitary operation that maps $|0\rangle$ to $|1\rangle$ and vice versa.

- The *Hadamard* operation is the single qubit operation that (up to normalization) maps $|0\rangle$ to $|0\rangle + |1\rangle$ and $|1\rangle$ to $|0\rangle - |1\rangle$. (More succinctly, the state $|b\rangle$ is mapped to $|0\rangle + (-1)^b|1\rangle$.)

  It turns out to be a very useful operation in many algorithms for quantum computers. Note that if we apply an Hadamard operation to every qubit of an n-qubit register, then for every $x \in \{0, 1\}^n$, the state $|x\rangle$ is mapped to

  $$(|0\rangle + (-1)^{x_1}|1\rangle)(|0\rangle + (-1)^{k_2}|1\rangle)\cdots(|0\rangle + (-1)^{x_n}|1\rangle) =$$
  $$\sum_{y\in\{0,1\}^n}\left(\pi_{i\ :\ y_i=1}(-1)_i^x\right)|y\rangle = \sum_{y\in\{0,1\}^n}-1^{x\odot y}|y\rangle,$$

  where $x \odot y$ denotes the inner product modulo 2 of x and y. That is, $x \odot y = \sum_{i=1}^{n}x_iy_i$ (mod 2).[1]

- Since we can think of the state of a single qubit register as a vector in two dimensional space, a natural operation is for any angle θ, to rotate the single qubit by θ. That is, map $|0\rangle$ to $\cos\theta|0\rangle + \sin\theta|1\rangle$, and map $|1\rangle$ to $-\sin\theta|0\rangle + \cos\theta|1\rangle$. Note that rotation by an angle of π (i.e., 180°) is equal to flipping the sign of the vector (i.e., the map $\mathbf{v} \mapsto -\mathbf{v}$).

- One simple two qubit operation is exchanging the two bits with one another. That is, mapping $|01\rangle \mapsto |10\rangle$ and $|10\rangle \mapsto |01\rangle$, with $|00\rangle$ and $|11\rangle$ being mapped to them selves. Note that by combining these operations we can reorder the qubits of an n-bit register in any way we see fit.

- Another two qubit operation is the controlled-NOT operation: it performs a NOT on the first bit iff the second bit is equal to 1. That is, it maps $|01\rangle \mapsto |11\rangle$ and $|11\rangle \mapsto |01\rangle$, with $|10\rangle$ and $|11\rangle$ being mapped to themselves.

---

[1]Note the similarity to the definition of the Walsh-Hadamard code described in Chapter 17, Section 17.5.

- The *Tofolli operation* is the three qubit operation that can be called a "controlled-controlled-NOT": it performs a NOT on the first bit iff both the second and third bits are equal to 1. That is, it maps $|011\rangle$ to $|111\rangle$ and vice versa, and maps all other basis states to themselves.

Exercise 2 asks you to write down explicitly the matrices for these operations.

We define quantum computation as consisting of a sequence of elementary operations in an analogous way to our previous definition of probabilistic computation (Definition 20.6):

DEFINITION 20.13 (QUANTUM COMPUTATION)
Let $f : \{0,1\}^* \rightarrow \{0,1\}^*$ and $T : \mathbb{N} \rightarrow \mathbb{N}$ be some functions. We say that f is *computable in quantum $T(n)$-time* if for every $n \in \mathbb{N}$ and $x \in \{0,1\}^n$, $f(x)$ can be computed by the following process:

1. Initialize an m qubit quantum register to the state $|x0^{n-m}\rangle$ (i.e., x padded with zeroes), where $m \leq T(n)$.

2. Apply one after the other $T(n)$ elementary quantum operations $F_1, \ldots, F_T$ to the register (where we require that there is a polynomial-time TM that on input $1^n, 1^{T(n)}$ outputs the descriptions of $F_1, \ldots, F_T$).

3. Measure the register and let Y denote the obtained value. (That is, if $\mathbf{v}$ is the final state of the register, then Y is a random variable that takes the value y with probability $|\mathbf{v}_y|^2$ for every $y \in \{0,1\}^n$.)

Denoting $\ell = |f(x)|$, we require that the first ℓ bits of Y are equal to $f(x)$ with probability at least $2/3$.

The following class aims to capture the decision problems with efficient algorithms on quantum computers:

DEFINITION 20.14 (THE CLASS **BQP**)
A Boolean function $f : \{0,1\}^* \rightarrow \{0,1\}$ is in **BQP** if there is some polynomial $p : \mathbb{N} \rightarrow \mathbb{N}$ such that f is computable in quantum $p(n)$-time.

> QUANTUM COMPUTATION: SUMMARY OF NOTATIONS.
> The *state* of an m-qubit register is represented by a unit vector $\mathbf{v} \in \mathbb{C}^{2^m}$ such that the register takes the value x with probability $|\mathbf{v}_x|^2$.
> An *operation* on the register is a function $F : \mathbb{C}^{2^m} \to \mathbb{C}^{2^m}$ that is unitary. (i.e., linear and norm preserving).
> An *elementary operation* only depends and modifies at most three bits of the register.
> A *computation* of a function f on input $x \in \{0,1\}^n$ involves initializing the register to the state $|x0^{m-n}\rangle$, applying a sequence of elementary operations to it, and then measuring its value.

REMARK 20.15
Readers familiar with quantum mechanics or quantum computing may notice that we did not allow in our definition of quantum computation several features that are allowed by quantum mechanics. These include *mixed* states, that involve both quantum superposition and probability, measuring in different basis than the standard basis, and performing partial measurements during the computation. However, none of these features adds to the computing power of quantum computers.

20.3.1 Universal quantum operations

Can we actually implement quantum computation? This is an excellent question, and no one really knows. However, one hurdle can be overcome: even though there is an infinite set of possible elementary operations, all of them can be generated (or at least sufficiently well approximated) by the Hadamard and Tofolli operations described in Example 20.12. In fact, every operation that depends on k qubits can be approximated by composing $2^{O(k)}$ of these four operations (times an additional small factor depending on the approximation's quality). Using a counting/dimension argument, it can be shown that some unitary transformations do indeed require an exponential number of elementary operations to compute (or even approximate).

One useful consequence of universality is the following: when designing quantum algorithms we can assume that we have at our disposal the all operations that depend on k qubits as elementary operations, for every constant k (even if $k > 3$). This is since these can be implemented by 3 qubit elementary operations incurring only a constant (depending on k) overhead.

20.3.2 Spooky coordination and Bell's state

To get our first glimpse of how things behave differently in the quantum world, we will now show how quantum registers and operations help us win the game described in Section 20.1.2 with higher probability than can be achieved according to pre-quantum "classical" physics.
Recall that the game was the following:

1. The verifier chooses random $x, y \in_R \{0,1\}$ and sends x to Alice and y to Bob, collecting their respective answers a and b.

2. It accepts iff $x \wedge y = a \oplus b$. In other words, it accepts if either $\langle x, y \rangle \neq \langle 1, 1 \rangle$ and $a = b$ or $x = y = 1$ and $a \neq b$.

It was shown in Section 20.1.2 that if Alice and Bob can not coordinate their actions then the verifier will accept with probability at most 3/4, but quantum effects can allow them to bypass this bound as follows:

1. Alice and Bob prepare a 2-qubit quantum register containing the EPR state $|00\rangle + |11\rangle$. (They can start with a register initialized to $|00\rangle$ and then apply an elementary operation that maps the initial state to the EPR state.)

2. Alice and Bob split the register - Alice takes the first qubit and Bob takes the second qubit. (The components containing each qubit of a quantum register do not necessarily need to be adjacent to one another.)

3. Alice receives the qubit x from the verifier, and if $x = 1$ then she applies a rotation by $\pi/8$ (22.5°) operation to her qubit. (Since the operation involves only her qubit, she can perform it even after the register was split.)

4. Bob receives the qubit y from the verifier, and if $y = 1$ he applies a rotation by by $-\pi/8$ (−22.5°) operation to his qubit.

5. Both of them measure their respective qubits and output the values obtained as their answers a and b.

Note that the order in which Alice and Bob perform their rotations and measurements does not matter - it can be shown that all orders yield exactly the same distribution (e.g., see Exercise 3). While splitting a quantum register and applying unitary transformations to the different parts may sound far fetched, this experiment had been performed several times in practice, verifying the following predictions of quantum mechanics:

THEOREM 20.16
Given the above strategy for Alice and Bob, the verifier will accept with probability at least 0.8.

PROOF: Recall that Alice and Bob win the game if they output the same answer when $\langle x, y \rangle \neq \langle 1, 1 \rangle$ and a different answer otherwise. The intuition behind the proof is that in the case that $\langle x, y \rangle \neq \langle 1, 1 \rangle$ then the states of the two qubits will be "close" to one another (the angle between them is less than $\pi/8$ or 22.5°) and in the other case the states will be "far" (having angle $\pi/4$ or 45°). Specifically we will show that:

(1) If $x = y = 0$ then $a = b$ with probability 1.

(2) If $x \neq y$ then $a = b$ with probability $\cos^2(\pi/8) \geq 0.85$

(3) If $x = y = 1$ then $a = b$ with probability 1/2.

Implying that the overall acceptance probability is at least $\frac{1}{4} + \frac{1}{2}0.85 + \frac{1}{8} = 0.8$.

In the case **(1)** both Alice and Bob perform no operation on their register, and so when measured it will be either in the state $|00\rangle$ or $|11\rangle$, both resulting in Alice and Bob's outputs being equal. To analyze case **(2)**, it suffices to consider the case that $x = 0, y = 1$ (the other case is symmetrical).

In this case Alice applies no transformation to her qubit, and Bob rotates his qubit in a $-\pi/8$ angle. Imagine that Bob first makes the rotation, then Alice measures her qubit and then Bob measures his (this is OK as the order of measurements does not change the outcome). With probability $1/2$, Alice will get the value 0 and Bob's qubit will collapse to the state $|0\rangle$ rotated by a $-\pi/8$ angle, meaning that when measuring Bob will obtain the value 0 with probability $\cos^2(\pi/8)$. Similarly, if Alice gets the value 1 then Bob will also output 1 with $\cos^2(\pi/8)$ probability.

To analyze case **(3)**, we just use direct computation. In this case, after both rotations are performed, the register's state is

$$
\begin{aligned}
(\cos(\pi/8)\,|0\rangle + \sin(\pi/8)\,|1\rangle)\,(\cos(\pi/8)\,|0\rangle - \sin(\pi/8)\,|1\rangle) + \\
(-\sin(\pi/8)\,|0\rangle + \cos(\pi/8)\,|1\rangle)\,(\sin(\pi/8)\,|0\rangle + \cos(\pi/8)\,|1\rangle) = \\
\left(\cos^2(\pi/8) - \sin^2(\pi/8)\right)|00\rangle - 2\sin(\pi/8)\cos(\pi/8)\,|01\rangle + \\
2\sin(\pi/8)\cos(\pi/8)\,|10\rangle + \left(\cos^2(\pi/8) - \sin^2(\pi/8)\right)|11\rangle\,.
\end{aligned}
$$

But since

$$
\cos^2(\pi/8) - \sin^2(\pi/8) = \cos(\pi/4) = \tfrac{1}{\sqrt{2}} = \sin(\pi/4) = 2\sin(\pi/8)\cos(\pi/8)\,,
$$

all coefficients in this state have the same absolute value and hence when measured the register will yield either one of the four values $00, 01, 10$ and 11 with equal probability $1/4$. ∎

20.4   Quantum programmer's toolkit

Quantum algorithms have some peculiar features that classical algorithm designers are not used to. The following observations can serve as a helpful "bag of tricks" for designing quantum algorithms:

- If we can compute an n-qubit unitary transformation U in T steps then we can compute the transformation Controlled-U in $O(T)$ steps, where Controlled-U maps a vector $|x_1 \ldots x_n x_{n+1}\rangle$ to $|U(x_1 \ldots x_n)x_{n+1}\rangle$ if $x_{n+1} = 1$ and to itself otherwise.

  The reason is that we can transform every elementary operation F in the computation of U to the analogous "Controlled-F" operation. Since the "Controlled-F" operation depends on at most 4 qubits, it can be considered also as elementary.

  For every two n-qubit transformations U, U', we can use this observation twice to compute the transformation that invokes U on $x_1 \ldots x_n$ if $x_{n+1} = 1$ and invokes U' otherwise.

- Every *permutation* of the standard basis is unitary. That is, any operation that maps a vector $|x\rangle$ into $|\pi(x)\rangle$ where π is a permutation of $\{0,1\}^n$ is unitary. Of course, this does not mean that all such permutations are efficiently computable in quantum polynomial time.

- For every function $f : \{0,1\}^n \to \{0,1\}^\ell$, the function $x, y \mapsto x, (y \oplus f(x))$ is a permutation on $\{0,1\}^{n+\ell}$ (in fact, this function is its own inverse). In particular, this means that we can use as elementary operations the following "permutation variants" of AND, OR and

DRAFT

copying: **(1)** $|x_1x_2x_3\rangle \mapsto |x_1x_2(x_3 \oplus (x_1 \wedge x_2)))\rangle$, **(2)** $|x_1x_2x_3\rangle \mapsto |x_1x_2(x_3 \oplus (x_1 \vee x_2))\rangle$, and **(3)** $|x_1x_2\rangle \mapsto |x_1(x_1 \oplus x_2)\rangle$. Note that in all these cases we compute the "right" function if the last qubit is initially equal to 0.

- If $f : \{0,1\}^n \to \{0,1\}^\ell$ is a function computable by a size-T Boolean circuit, then the following transformation can be computed by a sequence of $O(T)$ elementary operations: for every $x \in \{0,1\}^m, y \in \{0,1\}^\ell, z \in \{0,1\}^T$

$$|x,y,z\rangle \mapsto |x,(y \oplus f(x)),0^T\rangle \text{ if } z = 0^T.$$

(We don't care on what the mapping does for $z \neq 0^T$.)

The reason is that by transforming every AND, OR or NOT gate into the corresponding elementary permutation we can ensure that the ith qubit of z contains the result of the ith gate of the circuit when executed on input x. We can then XOR the result of the circuit into y using ℓ elementary operations and run the entire computation backward to return the state of z to 0^T. .

- We can assume that we are allowed to make a *partial measurement* in the course of the algorithm, and then proceed differently according to its outcome. That is, we can measure a some of the qubits of the register. Note that if the register is at state $\mathbf{v}$ and we measure its ith qubit then with probability $\sum_{z:z_i=1} |\mathbf{v}_z|^2$ we will get the answer "1" and the register's state will change to (the normalized version of) the vector $\sum_{z:z_i=1} \mathbf{v}_z |z\rangle$. Symmetrically, with probability $\sum_{z:z_i=0} |\mathbf{v}_z|^2$ we will get the answer "0" and the new state will be $\sum_{z:z_i=0} \mathbf{v}_z |z\rangle$.

  This is allowed since an algorithm using partial measurement can be replaced with an algorithm not using it with at most a constant overhead (see Exercise 4).

- Since the 1-qubit Hadamard operation maps $|0\rangle$ to the uniform state $|0\rangle + |1\rangle$, it can be used to simulate tossing a coin: we simply take a qubit in our workspace that is initialized to 0, apply Hadamard to it, and measure the result.

Together, the last three observations imply that quantum computation is at least as powerful as "classical" non-quantum computation:

THEOREM 20.17
BPP $\subseteq$ BQP.

20.5 Grover's search algorithm.

Consider the **NP**-complete problem SAT of finding, given an n-variable Boolean formula φ, whether there exists an assignment $a \in \{0,1\}^n$ such that $\varphi(a) = 1$. Using "classical" deterministic or probabilistic TM's, we do not know how to solve this problem better than the trivial $\text{poly}(n)2^n$-time algorithm.[2] We now show a beautiful algorithm due to Grover that solves SAT in $\text{poly}(n)2^{n/2}$-time on a quantum computer. This is a significant improvement over the classical case, even if it falls

---

[2]There are slightly better algorithms for special cases such as 3SAT.

way short of showing that $\mathbf{NP} \subseteq \mathbf{BQP}$. In fact, Grover's algorithm solves an even more general problem: for *every* polynomial-time computable function $f : \{0,1\}^n \to \{0,1\}$ (even if f is not expressed as a small Boolean formula[3]), it finds in $\mathrm{poly}(n)2^{n/2}$ time a string a such that $f(a) = 1$ (if such a string exists).

Grover's algorithm is best described geometrically. We assume that the function f has a *single* satisfying assignment a. (The techniques described in Chapter 9, Section 9.3.1 allow us to reduce the general problem to this case.) Consider an n-qubit register, and let $\mathbf{u}$ denote the *uniform state vector* of this register. That is, $\mathbf{u} = \frac{1}{2^{n/2}} \sum_{x \in \{0,1\}^n} |x\rangle$. The angle between $\mathbf{u}$ and $|a\rangle$ is equal to the inverse cosine of their inner product $\langle \mathbf{u}, |a\rangle \rangle = \frac{1}{2^{n/2}}$. Since this is a positive number, this angle is smaller than $\pi/2$ (90°), and hence we denote it by $\pi/2 - \theta$, where $\sin\theta = \frac{1}{2^{n/2}}$ and hence, assuming n is sufficiently large, $\theta \geq \frac{1}{2\cdot 2^{n/2}}$ (since for small θ, $\sin\theta \sim \theta$).
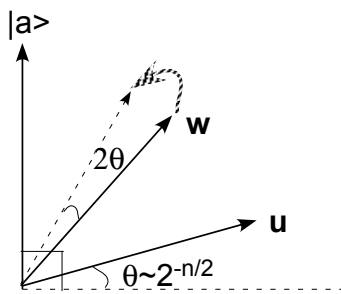
Figure 20.1: Grover's algorithm finds the string a such that $f(a) = 1$ as follows. It starts with the uniform vector $\mathbf{u}$ whose angle with $|a\rangle$ is $\pi/2 - \theta$ for $\theta \sim 2^{-n/2}$ and at each step transforms the state of the register into a vector that is 2θ radians closer to $|a\rangle$. After $O(1/\theta)$ steps, the state is close enough so that measuring the register yields $|a\rangle$ with good probability.

The algorithm starts with the state $\mathbf{u}$, and at each step it gets nearer the state $|a\rangle$ by transforming its current state to a state whose angle with $|a\rangle$ is smaller by 2θ (see Figure 20.1). Thus, in $O(1/\theta) = O(2^{n/2})$ steps it will get to a state $\mathbf{v}$ whose inner product with $|a\rangle$ is larger than, say, $1/2$, implying that a measurement of the register will yield a with probability at least $1/4$.

The main idea is that to rotate a vector $\mathbf{w}$ towards the unknown vector $|a\rangle$ by an angle of θ, it suffices to take two *reflections* around the vector $\mathbf{u}$ and the vector $\mathbf{e} = \sum_{x \neq a} |a\rangle$ (the latter is the vector orthogonal to $|a\rangle$ on the plane spanned by $\mathbf{u}$ and $|a\rangle$). See Figure 20.2 for a "proof by picture".

To complete the algorithm's description, we need to show how we can perform the reflections around the vectors $\mathbf{u}$ and $\mathbf{e}$. That is, we need to show how we can in polynomial time transform a state $\mathbf{w}$ of the register into the state that is $\mathbf{w}$'s reflection around $\mathbf{u}$ (respectively, $\mathbf{e}$). In fact, we will not work with an n-qubit register but with an m-qubit register for m that is polynomial in n. However, the extra qubits will only serve as "scratch workspace" and will always contain zero except during intermediate computations, and hence can be safely ignored.

---

[3]We may assume that f is given to the algorithm in the form of a polynomial-sized circuit.
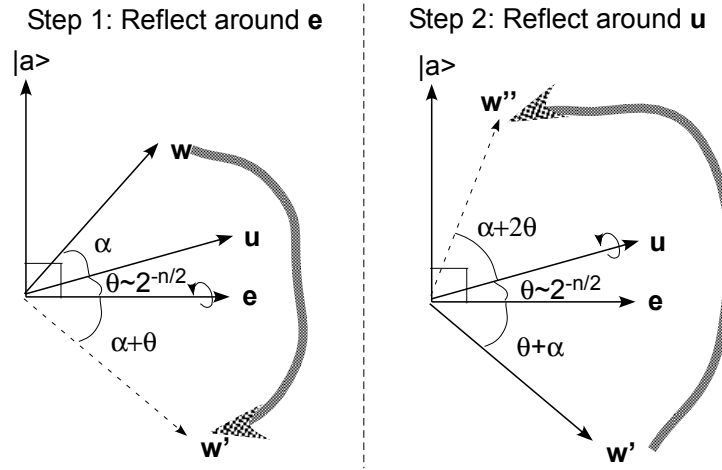
Figure 20.2: We transform a vector $\mathbf{w}$ in the plane spanned by $|a\rangle$ and $\mathbf{u}$ into a vector $\mathbf{w}''$ that is 2θ radians close to $|a\rangle$ by performing two reflections. First, we reflect around $\mathbf{e} = \sum_{x \neq a} |x\rangle$ (the vector orthogonal to $|a\rangle$ on this plane), and then we reflect around $\mathbf{u}$. If the original angle between $\mathbf{w}$ and $|a\rangle$ was $\pi/2 - \theta - \alpha$ then the new angle will be $\pi/2 - \theta - \alpha - 2\theta$.

Reflecting around e. Recall that to reflect a vector $\mathbf{w}$ around a vector $\mathbf{v}$, we express $\mathbf{w}$ as $\alpha\mathbf{v} + \mathbf{v}^{\perp}$ (where $\mathbf{v}^{\perp}$ is orthogonal to $\mathbf{v}$) and output $\alpha\mathbf{v} - \mathbf{v}^{\perp}$. Thus the reflection of $\mathbf{w}$ around $\mathbf{e}$ is equal to $\sum_{x \neq a} \mathbf{w}_x |x\rangle - \mathbf{w}_a |a\rangle$. Yet, it is easy to perform this transformation:

1. Since f is computable in polynomial time, we can compute the transformation $|x\sigma\rangle \mapsto |x(\sigma \oplus f(x))\rangle$ in polynomial (this notation ignores the extra workspace that may be needed, but this won't make any difference). This transformation maps $|x0\rangle$ to $|x0\rangle$ for $x \neq a$ and $|a0\rangle$ to $|a1\rangle$.

2. Then, we apply the elementary transformation that multiplies the vector by -1 if $\sigma = 1$, and does nothing otherwise. This maps $|x0\rangle$ to $|x0\rangle$ for $x \neq a$ and maps $|a1\rangle$ to $-|a1\rangle$.

3. Then, we apply the transformation $|x\sigma\rangle \mapsto |x(\sigma \oplus f(x))\rangle$ again, mapping $|x0\rangle$ to $|x0\rangle$ for $x \neq a$ and maps $|a1\rangle$ to $|a0\rangle$.

The final result is that the vector $|x0\rangle$ is mapped to itself for $x \neq a$, but $|a0\rangle$ is mapped to $-|a0\rangle$. Ignoring the last qubit, this is exactly a reflection around $|a\rangle$.

Reflecting around u. To reflect around $\mathbf{u}$, we first apply the Hadamard operation to each qubit, mapping $\mathbf{u}$ to $|0\rangle$. Then, we reflect around $|0\rangle$ (this can be done in the same way as reflecting around $|a\rangle$, just using the function $g : \{0,1\}^n \to \{0,1\}$ that outputs 1 iff its input is all zeroes instead of f). Then, we apply the Hadamard operation again, mapping $|0\rangle$ back to $\mathbf{u}$.

Together these operations allow us to take a vector in the plane spanned by $|a\rangle$ and $\mathbf{u}$ and rotate it 2θ radians closer to $|a\rangle$. Thus if we start with the vector $\mathbf{u}$, we will only need to repeat them

$O(1/\theta) = O(2^{n/2})$ to obtain a vector that, when measured, yields $|a\rangle$ with constant probability. For the sake of completeness, Figure 20.3 contains the full description of Grover's algorithm. ∎

Grover's Search Algorithm.

Goal: Given a polynomial-time computable $f : \{0,1\}^n \to \{0,1\}$ with a unique $a \in \{0,1\}^n$ such that $f(a) = 1$, find a.

Quantum register: We use an $n+1+m$-qubit register, where m is large enough so we can compute the transformation $|x\sigma 0^m\rangle \mapsto |x(\sigma \oplus f(x))0^m\rangle$.

Operation	State (neglecting normalizing factors)
Apply Hadamard operation to first n qubits.	Initial state: $\left|0^{n+m+1}\right\rangle$ $\mathbf{u}\left|0^{m+1}\right\rangle$ (where $\mathbf{u}$ denotes $\sum_{x\in\{0,1\}^n}\left|x\right\rangle$)
For $i = 1, \ldots, 2^{n/2}$ do:	$\mathbf{v}^i\left|0^{m+1}\right\rangle$ We let $\mathbf{v}^1 = \mathbf{u}$ and maintain the invariant that $\langle\mathbf{v}^i,\left|a\right\rangle\rangle = \sin(i\theta)$, where $\theta \sim 2^{-n/2}$ is such that $\langle\mathbf{u},\left|a\right\rangle\rangle = \sin(\theta)$
Step 1: *Reflect around* $\mathbf{e} = \sum_{x\neq a}\left|x\right\rangle$: 1.1 Compute $\left|x\sigma 0^{m+1}\right\rangle \mapsto \left|x(\sigma \oplus f(x))0^{m+1}\right\rangle$ 1.2 If $\sigma = 1$ then multiply vector by -1, otherwise do not do anything. 1.3 Compute $\left|x\sigma 0^{m+1}\right\rangle \mapsto \left|x(\sigma \oplus f(x))0^{m+1}\right\rangle$.	$\sum_{x\neq a}\mathbf{v}_x^i\left|x\right\rangle\left|0^{m+1}\right\rangle + \mathbf{v}_a^i\left|a\right\rangle\left|10^{m+1}\right\rangle$ $\sum_{x\neq a}\mathbf{v}_x^i\left|x\right\rangle\left|0^{m+1}\right\rangle - \mathbf{v}_a^i\left|a\right\rangle\left|10^{m+1}\right\rangle$ $\mathbf{w}^i\left|0^{m+1}\right\rangle \quad = \quad \sum_{x\neq a}\mathbf{v}_x^i\left|x\right\rangle\left|0^{m+1}\right\rangle - \mathbf{v}_a^i\left|a\right\rangle\left|00^m\right\rangle$. ($\mathbf{w}^i$ is $\mathbf{v}^i$ reflected around $\sum_{x\neq a}\left|x\right\rangle$.)
Step 2: *Reflect around* $\mathbf{u}$: 2.1 Apply Hadamard operation to first n qubits.	$\langle\mathbf{w}^i,\mathbf{u}\rangle\left|0^n\right\rangle\left|0^{m+1}\right\rangle + \sum_{x\neq 0^n}\alpha_x\left|x\right\rangle\left|0^{m+1}\right\rangle$, for some coefficients α_x's (given by $\alpha_x = \sum_z(-1)^{x\odot z}\mathbf{w}_z^i\left|z\right\rangle$).
2.2 Reflect around $\left|0\right\rangle$: 2.2.1 If first n-qubits are all zero then flip $n+1^{st}$ qubit. 2.2.2 If $n+1^{st}$ qubit is 1 then multiply by -1 2.2.3 If first n-qubits are all zero then flip $n+1^{st}$ qubit.	$\langle\mathbf{w}^i,\mathbf{u}\rangle\left|0^n\right\rangle\left|10^m\right\rangle + \sum_{x\neq 0^n}\alpha_x\left|x\right\rangle\left|0^{m+1}\right\rangle$ $-\langle\mathbf{w}^i,\mathbf{u}\rangle\left|0^n\right\rangle\left|10^m\right\rangle + \sum_{x\neq 0^n}\alpha_x\left|x\right\rangle\left|0^{m+1}\right\rangle$ $-\langle\mathbf{w}^i,\mathbf{u}\rangle\left|0^n\right\rangle\left|0^{m+1}\right\rangle + \sum_{x\neq 0^n}\alpha_x\left|x\right\rangle\left|0^{m+1}\right\rangle$
2.3 Apply Hadamard operation to first n qubits.	$\mathbf{v}^{i+1}\left|0^{m+1}\right\rangle$ (where $\mathbf{v}^{i+1}$ is $\mathbf{w}^i$ reflected around $\mathbf{u}$)
Measure register and let a' be the obtained value in the first n qubits. If $f(a') = 1$ then output a'. Otherwise, repeat.	

Figure 20.3: Grover's Search Algorithm

20.6   Simon's Algorithm

Although beautiful, Grover's algorithm still has a significant drawback: it is merely quadratically faster than the best known classical algorithm for the same problem. In contrast, in this section we show *Simon's algorithm* that is a polynomial-time quantum algorithm solving a problem for which the best known classical algorithm takes *exponential* time.

Simon's algorithm solves the following problem: given a polynomial-time computable function $f : \{0,1\}^n \to \{0,1\}^n$ such that there exists $a \in \{0,1\}^*$ satisfying $f(x) = f(y)$ iff $x = y \oplus a$ for every $x, y \in \{0,1\}^n$, find this string a.

Two natural questions are (1) why is this problem interesting? and (2) why do we believe it is hard to solve for classical computers? The best answer to (1) is that, as we will see in Section 20.7, a generalization of Simon's problem turns out to be crucial in the quantum polynomial-time algorithm for famous *integer factorization* problem. Regarding (2), of course we do not know for certain that this problem does not have a classical polynomial-time algorithm (in particular, if $\mathbf{P} = \mathbf{NP}$ then there obviously exists such an algorithm). However, some intuition why it may be hard can be gleaned from the following *black box* model: suppose that you are given access to a black box (or oracle) that on input $x \in \{0,1\}^n$, returns the value $f(x)$. Would you be able to learn a by making at most a subexponential number of queries to the black box? It is not hard to see that if a is chosen at random from $\{0,1\}^n$ and f is chosen at random subject to the condition that $f(x) = f(y)$ iff $x = y \oplus a$ then no algorithm can successfully recover a with reasonable probability using significantly less than $2^{n/2}$ queries to the black box. Indeed, an algorithm using fewer queries is very likely to never get the same answer to two distinct queries, in which case it gets no information about the value of a.

20.6.1   The algorithm

Simon's algorithm is actually quite simple. It uses a register of $2n + m$ qubits, where m is the number of workspace bits needed to compute f. (Below we will ignore the last m qubits of the register, since they will be always set to all zeroes except in intermediate steps of f's computation.) The algorithm first uses n Hadamard operations to set the first n qubits to the uniform state and then apply the operation $|xz\rangle \mapsto |x(z \oplus f(x)\rangle$ to the register, resulting (up to normalization) in the state

$$\sum_{x \in \{0,1\}^n} |x\rangle |f(x)\rangle = \sum_{x \in \{0,1\}^n} (|x\rangle + |x \oplus a\rangle) |f(x)\rangle. \qquad (1)$$

We then *measure* the second n bits of the register, collapsing its state to

$$|xf(x)\rangle + |(x \oplus a)f(x)\rangle \qquad (2)$$

for some string x (that is chosen uniformly from $\{0,1\}^n$). You might think that we're done as the state (2) clearly encodes a, however we cannot directly learn a from this state: if we measure the first n bits we will get with probability $1/2$ the value x and with probability $1/2$ the value $x \oplus a$. Even though a can be deduced from these two values combined, each one of them on its own yields no information about a. (This point is well worth some contemplation, as it underlies the subtleties

involved in quantum computation and demonstrates why a quantum algorithm is *not* generally equivalent to performing exponentially many classical computation in parallel.)

However, consider now what happens if we perform another n Hadamard operations on the first n bits. Since this maps x to the vector $\sum_y (-1)^{x \odot y} |y\rangle$, the new state of the first n bits will be

$$\sum_y \left((-1)^{x \odot y} + (-1)^{(x \oplus a) \odot y} \right) |y\rangle = \sum_y \left((-1)^{x \odot y} + (-1)^{x \odot y} (-1)^{a \odot y} \right) |y\rangle. \tag{3}$$

For every $y \in \{0,1\}^m$, the yth coefficient in the state (3) is nonzero if and only if if and only if $a \odot y = 0$, and in fact if measured, the state (3) yields a uniform $y \in \{0,1\}^n$ satisfying $a \odot y = 0$.

Repeating the entire process k times, we get k uniform strings $y_1, \ldots, y_k$ satisfying $y \odot a = 0$ or in other words, k *linear equations* (over the field GF(2)) on the variables $a_1, \ldots, a_n$. It can be easily shown that if, say, $k \geq 2n$ then with high probability there will be $n-1$ linearly independent equations among these (see Exercise 5), and hence we will be able to retrieve a from these equations using Gaussian elimination. For completeness, a full description of Simon's algorithm can be found in Figure 20.4.

Simon's Algorithm.

Goal: Given a polynomial-time computable $f : \{0,1\}^n \to \{0,1\}^n$ such that there is some $a \in \{0,1\}^n$ satisfying $f(x) = f(y)$ iff $y = x \oplus a$ for every $x, y \in \{0,1\}^n$, find a.

Quantum register: We use an $2n + m$-qubit register, where m is large enough so we can compute the transformation $|xz0^m\rangle \mapsto |x(z \oplus f(x))0^m\rangle$. (Below we ignore the last m qubits of the register as they will always contain 0^m except in intermediate computations of f.)

Operation	**State** (neglecting normalizing factors)
Apply Hadamard operation to first n qubits. Compute $|xz\rangle \mapsto |x(y \oplus f(x))\rangle$ Measure second n bits of register. Apply Hadamard to first n bits.	Initial state: $\left|0^{2n}\right\rangle$ $\sum_x |x0^n\rangle$ $\sum_x |xf(x)\rangle = \sum_x (|x\rangle + |x \oplus a\rangle) |f(x)\rangle$ $(|x\rangle + |x \oplus a\rangle) |f(x)\rangle$ $\left(\sum_y (-1)^{a \odot y} (1 + (-1)^{a \odot y}) |y\rangle \right) |f(x)\rangle \quad = 2 \sum_{y:a \odot y=0} (-1)^{x \odot y} |y\rangle |f(x)\rangle$
Measure first n qubits of register to obtain a value y such that $y \odot a = 0$. Repeat until we get a sufficient number of linearly independent equations on a.	

Figure 20.4: Simon's Algorithm

20.7 Shor's algorithm: integer factorization using quantum computers.

The *integer factorization* problem is to find, given an integer N, the set of all *prime factors* of N (i.e., prime numbers that divide N). By a polynomial-time algorithm for this problem we mean an

algorithm that runs in time polynomial in the description of N, i.e., $\text{poly}(\log(N))$ time. Although people have been thinking about the factorization problem in one form or another for at least 2000 years, we still do not know of a polynomial-time algorithm for it: the best classical algorithm takes roughly $2^{(\log N)^{1/3}}$ steps to factor N [?]. In fact, the presumed difficulty of this problem underlies many popular encryption schemes (such as RSA). Therefore, it was quite a surprise when in 1994 Peter Shor showed a quantum polynomial-time algorithm for this problem. To this day it remains the most famous algorithm for quantum computers, and the strongest evidence that **BQP** may contain problems outside of **BPP**.

The order-finding problem. Rather than showing an algorithm to factor a given number N, we will show an algorithm for a related problem: given a number A with $gcd(A, N) = 1$, find the *order* of A modulo N, defined to be the smallest positive integer r such that $A^r = 1 \pmod{N}$. Using elementary number theory, it is fairly straightforward to reduce the task of factoring N to solving this problem, and we defer the description of this reduction to Section 20.7.3.

REMARK 20.18
It is easy to see that for every positive integer k, if $A^k = 1 \pmod{N}$ then r divides k. (Indeed, otherwise if $k = cr + d$ for $c \in \mathbb{Z}$ and $d \in \{1, .., r-1\}$ then $A^d = 1 \pmod{N}$, contradicting the minimality of r.) Similarly, for every x, y it holds that $A^x = A^y \pmod{N}$ iff $x - y$ is a multiple of r. Therefore, the order finding problem can be defined as follows: given the function $f : \mathbb{N} \to \mathbb{N}$ that maps x to $A^x \pmod{N}$ and satisfies that $f(x) = f(y)$ iff $r|x - y$, find r. In this notation, the similarity to Simon's problem becomes more apparent.

20.7.1 Quantum Fourier Transform over $\mathbb{Z}_M$.

The main tool used to solve the order-finding problem is the *quantum Fourier transform*. We have already encountered the Fourier transform in Chapter 19, but will now use a different variant, which we call the *Fourier transform over* $\mathbb{Z}_M$ where $M = 2^m$ for some integer M. Recall that $\mathbb{Z}_M$ is the group of all number in $\{0, \ldots, M - 1\}$ with the group operation being addition modulo M. The Fourier transform over this group, defined below, is a linear and in fact unitary operation from $\mathbb{C}^{2^m}$ to $\mathbb{C}^{2^m}$. The quantum Fourier transform is a way to perform this operation by composing $O(m^2)$ elementary quantum operations (operations that depend on at most three qubits). This means that we can transform a quantum system whose register is in state f to a system whose register is in the state corresponding to the Fourier transform $\hat{f}$ of f. This does *not* mean that we can compute in $O(m^2)$ the Fourier transform over $\mathbb{Z}_M$ - indeed this is not sufficient time to even write the output! Nonetheless, this transformation still turns out to be very useful, and is crucial to Shor's factoring algorithm in the same way that the Hadamard transformation (which is a Fourier transform over the group $\{0, 1\}^n$ with the operation $\oplus$) was crucial to Simon's algorithm.

Definition of the Fourier transform over $\mathbb{Z}_M$.

Let $M = 2^m$ and let $\omega = e^{2\pi i/M}$. Note that $\omega^M = 1$ and $\omega^K \neq 1$ for every positive integer $K < N$ (we call such a number ω a primitive M^{th} root of unity). A function $\chi : \mathbb{Z}_M \to \mathbb{C}$ is called a *character* of $\mathbb{Z}_M$ if $\chi(y+z) = \chi(y)\chi(z)$ for every $y, z \in \mathbb{Z}_M$. $\mathbb{Z}_M$ has M characters $\{\chi_x\}_{x \in \mathbb{Z}_M}$ where

$\chi_x(y) = \omega^{xy}$. Let $\tilde{\chi}_x = \chi_x/\sqrt{M}$ (this factor is added for normalization), then the set $\{\tilde{\chi}_x\}_{x \in \mathbb{Z}_M}$ is an *orthonormal basis* of the space $\mathbb{C}^M$ since

$$\langle \tilde{\chi}_x, \tilde{\chi}_y \rangle = \tfrac{1}{M} \sum_{z=0}^{M-1} \omega^{xz}\overline{\omega^{yz}} = \tfrac{1}{M} \sum_{z=0}^{M-1} \omega^{(x-y)z}$$

which is equal to 1 if $x = y$ and to $\tfrac{1}{M}\frac{1-\omega^{(x-y)M}}{1-\omega^{x-y}} = 0$ if $x \neq y$ (the latter equality follows by the formula for the sum of a geometric series and the fact that $\omega^{\ell M} = 1$ for every ℓ).

DEFINITION 20.19
For f a vector in $\mathbb{C}^M$, the *Fourier transform of f* is the representation of f in the basis $\{\tilde{\chi}_x\}$. We let $\hat{f}(x)$ denote the coefficient of $\tilde{\chi}_{-x}$ in this representation. Thus $f = \sum_{x=0}^{M-1} \hat{f}(x)\tilde{\chi}_{-x}$ and so $\hat{f}(x) = \langle f, \chi_{-x} \rangle = \frac{1}{\sqrt{M}} \sum_{y=0}^{M-1} \omega^{xy} f(x)$. We let $FT_M(f)$ denote the vector $(\hat{f}(0), \ldots, \hat{f}(M-1))$. The function FT_M is a unitary operation from $\mathbb{C}^M$ to $\mathbb{C}^M$ and is called the *Fourier transform over $\mathbb{Z}^M$*.

Fast Fourier Transform

Note that

$$\hat{f}(x) = \tfrac{1}{\sqrt{M}} \sum_{y \in \mathbb{Z}_M} f(y)\omega^{xy} = \tfrac{1}{\sqrt{M}} \sum_{y \in \mathbb{Z}_M, y \text{ even}} f(y)\omega^{-2x(y/2)} + \omega^x \tfrac{1}{\sqrt{M}} \sum_{y \in \mathbb{Z}_M, y \text{ odd}} f(y)\omega^{2x(y-1)/2} .$$

Now since ω^2 is an $M/2$th root of unity and $\omega^{M/2} = -1$, letting W be the $M/2$ diagonal matrix with diagonal $\omega^0, \ldots, \omega^{M/2-1}$, we get that

$$FT_M(f)_{low} = FT_{M/2}(f_{even}) + WFT_{M/2}(f_{odd}) \tag{4}$$
$$FT_M(f)_{high} = FT_{M/2}(f_{even}) - WFT_{M/2}(f_{odd}) \tag{5}$$

where for an M-dimensional vector $\mathbf{v}$, we denote by $\mathbf{v}_{even}$ (resp. $\mathbf{v}_{odd}$) the $M/2$-dimensional vector obtained by restricting $\mathbf{v}$ to the coordinates whose indices have least significant bit equal to 0 (resp. 1) and by $\mathbf{v}_{low}$ (resp. $\mathbf{v}_{high}$) the restriction of $\mathbf{v}$ to coordinates with most significant bit 0 (resp. 1).

Equations (4) and (5) are the crux of the well known *Fast Fourier Transform* (FFT) algorithm that computes the Fourier transform in $O(M \log M)$ (as opposed to the naive $O(M^2)$) time. We will use them for the *quantum* Fourier transform algorithm, obtaining the following lemma:

LEMMA 20.20
There is an $O(m^2)$-step quantum algorithm that transforms a state $f = \sum_{x \in \mathbb{Z}_m} f(x) |x\rangle$ into the state $\hat{f} = \sum_{x \in \mathbb{Z}_m} \hat{f}(x) |x\rangle$, where $\hat{f}(x) = \frac{1}{\sqrt{M}} \sum_{y \in \mathbb{Z}_m} \omega^{xy} f(x)$.

Quantum Fourier transform: proof of Lemma 20.20

To prove Lemma 20.20, we use the following algorithm:

Quantum Fourier Transform FT_M	
Initial state: $f = \sum_{x \in \mathbb{Z}_M} f(x) \lvert x \rangle$	
Final state: $\hat{f} = \sum_{x \in \mathbb{Z}_M} \hat{f}(x) \lvert x \rangle.$	
Operation	**State** (neglecting normalizing factors)
	$f = \sum_{x \in \mathbb{Z}_M} f(x) \lvert x \rangle$
Recursively run $FT_{M/2}$ on $m-1$ most significant qubits	$(FT_{M/2} f_{even}) \lvert 0 \rangle + (FT_{M/2} f_{odd}) \lvert 1 \rangle$
If LSB is 1 then compute W on $m-1$ most significant qubits (see below).	$(FT_{M/2} f_{even}) \lvert 0 \rangle + (WFT_{M/2} f_{odd}) \lvert 1 \rangle$
Apply Hadmard gate H to least significant qubit.	$(FT_{M/2} f_{even})(\lvert 0 \rangle + \lvert 1 \rangle) + (WWFT_{M/2} f_{odd})(\lvert 0 \rangle - \lvert 1 \rangle) = (FT_{M/2} f_{even} + FT_{M/2} f_{odd}) \lvert 0 \rangle + (FT_{M/2} f_{even} - WFT_{M/2} f_{odd}) \lvert 1 \rangle$
Move LSB to the most significant position	$\lvert 0 \rangle (FT_{M/2} f_{even} + FT_{M/2} f_{odd}) + \lvert 1 \rangle (FT_{M/2} f_{even} - WFT_{M/2} f_{odd}) = \hat{f}$

The transformation W on $m-1$ qubits can be defined by $\lvert x \rangle \mapsto \omega^x = \omega^{\sum_{i=0}^{m-2} 2^i x_i}$ (where x_i is the ith qubit of x). It can be easily seen to be the result of applying for every $i \in \{0, \ldots, m-2\}$ the following elementary operation on the ith qubit of the register: $\lvert 0 \rangle \mapsto \lvert 0 \rangle$ and $\lvert 1 \rangle \mapsto \omega^{2^i} \lvert 1 \rangle$.

The final state is equal to $\hat{f}$ by (4) and (5). (We leave verifying this and the running time to Exercise 9.) ∎

20.7.2 The Order-Finding Algorithm.

We now present a quantum algorithm that on input a number $A < N$, finds the order of A modulo N (i.e., the smallest r such that $A^r = 1 \pmod{N}$). We let $m = 3 \log N$ and $M = 2^m$. Our register will consist of $m + \log(N)$ qubits. Note that the function $x \mapsto A^x \pmod{N}$ can be computed in polylog(N) time (see Exercise 6) and so we will assume that we can compute the map $\lvert x \rangle \lvert y \rangle \mapsto \lvert x \rangle \lvert y \oplus \llcorner A^x \pmod{N} \lrcorner \rangle$ (where $\llcorner X \lrcorner$ denotes the representation of the number $X \in \{0, \ldots, N-1\}$ as a binary string of length $\log N$).[4] The order-finding algorithm is as follows:

---

[4] To compute this map we may need to extend the register by some additional qubits, but we can ignore them as they will always be equal to zero except in intermediate computations.

Order finding algorithm.

Goal: Given numbers N and $A < N$ such that $gcd(A, N) = 1$, find the smallest r such that $A^r = 1 \pmod{N}$.

Quantum register: We use an $m + n$-qubit register, where $m = 3 \log N$ (and hence in particular $M \geq N^3$). Below we treat the first m bits of the register as encoding a number in $\mathbb{Z}_M$.

Operation	State (*including* normalizing factors)
Apply Fourier transform to the first m bits.	$\frac{1}{\sqrt{M}} \sum_{x \in \mathbb{Z}_M} |x\rangle) |0^n\rangle$
Compute the transformation $|x\rangle |y\rangle \mapsto |x\rangle |y \oplus (A^x \pmod{N}))\rangle$.	$\sum_{x \in \mathbb{Z}_M} |x\rangle |A^x \pmod{N}\rangle$
Measure the second register to get a value y_0.	$\frac{1}{\sqrt{\lceil M/r \rceil}} \sum_{\ell=0}^{\lceil M/r \rceil - 1} |x_0 + \ell r\rangle |y_0\rangle$ where x_0 is the smallest number such that $A^{x_0} = y_0$ $\pmod{N}$.
Apply the Fourier transform to the first register.	$\frac{1}{\sqrt{M}\sqrt{\lceil M/r \rceil}} \left(\sum_{x \in \mathbb{Z}_n} \sum_{\ell=0}^{\lceil M/r \rceil - 1} \omega^{(x_0 + \ell r)x} |x\rangle \right) |y_0\rangle$

Measure the first register to obtain a number $x \in \mathbb{Z}_M$. Find the best rational approximation a/b (with a, b coprime) for the fraction $\frac{x}{M}$ with denominator b at most $40M$ (see Section 20.A). If $A^b = A \pmod{M}$ then output b.

In the analysis, it will suffice to show that this algorithm outputs the order r with probability at least $1/\text{poly}(\log(N))$ (we can always amplify the algorithm's success by running it several times and taking the smallest output).

Analysis: the case that $r|M$

We start by analyzing the algorithm in the (rather unrealistic) case that $M = rc$ for some integer c. In this case we claim that the value x measured will be equal to $c'c$ for random $c' \in 0, \ldots, r$. In this case, $x/M = c'/r$. However, with probability at least $\Omega(1/\log(r))$, the number c' will be prime (and in particular coprime to r). In this case, the denominator of the rational approximation for x/M is indeed equal to r.

Indeed, for every $x \in \mathbb{Z}_M$, the absolute value of $|x\rangle$'s coefficient before the measurement is equal (up to some normalization factor) to

$$\left| \sum_{\ell=0}^{c-1} \omega^{(x_0 + \ell r)x} \right| = \left| \omega^{x_0 c'c} \right| \left| \sum_{\ell=0}^{c-1} \omega^{r\ell x} \right| = 1 \cdot \left| \sum_{\ell=0}^{c-1} \omega^{r\ell x} \right| . \tag{6}$$

But if $x = cc'$ then $\omega^{r\ell cc'} = \omega^{Mc'} = 1$, and hence the coefficients of all such x's are equal to the same positive number. On the other hand, if c does not divide x then then since ω^r is a cth root of unity, $\sum_{\ell=0}^{c-1} w^{r\ell x} = 0$ by the formula for sums of geometric progressions. Thus, such a number x would be measured with zero probability.

The case that $r \nmid M$

In the general case, we will not be able to show that the value x measured satisfies $M|xr$. However, we will show that with $\Omega(1/\log r)$ probability, **(1)** xr will be "almost divisible" by M in the sense
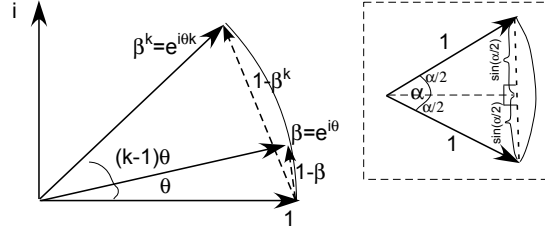
Figure 20.5: A complex number $z = a + ib$ can be thought of as the two-dimensional vector (a, b) of length $|z| = \sqrt{a^2 + b^2}$. The number $\beta = e^{i\theta}$ corresponds to a unit vector of angle θ from the x axis. For any such β, if k is not too large (say $k < 1/\theta$) then by elementary geometric considerations $\frac{|1-\beta^k|}{|1-\beta|} = \frac{2\sin(\theta/2)}{2\sin(k\theta/2)}$. We use here the fact (proved in the boxed figure) that in a unit cycle, the chord corresponding to an angle α is of length $2\sin(\alpha/2)$.

that $0 \leq xr \pmod{M} < r/10$ and **(2)** $\lceil xr/M \rceil$ is coprime to r. Condition **(1)** implies that $|xr - cM| < r/10$ for $c = \lceil xr/M \rceil$. Dividing by rM gives $\left| \frac{x}{M} - \frac{c}{r} \right| < \frac{1}{10M}$. Therefore, $\frac{c}{r}$ is a rational number with denominator at most N that approximates $\frac{x}{M}$ to within $1/(10M) < 1/(2N^2)$. It is not hard to see that such an approximation is unique (Exercise 7) and hence in this case the algorithm will come up with c/r and output the denominator r.

Thus all that is left is to prove the following two lemmas:

LEMMA 20.21
There exist $\Omega(r/\log r)$ values $x \in \mathbb{Z}_M$ such that:

1. *$0 \leq xr \pmod{M} < r/10$*

2. *$\lceil xr/M \rceil$ and r are coprime*

LEMMA 20.22
If x satisfies $0 \leq xr \pmod{M} < r/10$ then, before the measurement in the final step of the order-finding algorithm, the coefficient of $|x\rangle$ is at least $\Omega(\frac{1}{\sqrt{r}})$.

PROOF OF LEMMA 20.21: We prove the lemma for the case that r is coprime to M, leaving the general case as Exercise 10. In this case, the map $x \mapsto rx \pmod{M}$ is a permutation of $\mathbb{Z}_M^*$ and we have a set of at least $r/(20 \log r)$ x's such that $xr \pmod{M}$ is a prime number p between 0 and $r/10$. For every such x, $xr + \lceil r/M \rceil M = p$ which means that $\lceil r/M \rceil$ can not have a nontrivial shared factor with r, as otherwise this factor would be shared with p as well. $\blacksquare$

PROOF OF LEMMA 20.22: Let x be such that $0 \leq xr \pmod{M} < r/10$. The absolute value of $|x\rangle$'s coefficient in the state before the measurement is

$$\frac{1}{\sqrt{\lceil M/r \rceil}\sqrt{M}} \left| \sum_{\ell=0}^{\lceil M/r \rceil - 1} \omega^{\ell r x} \right|. \tag{7}$$

Setting $\beta = \omega^{rx}$ (note that since $M \nmid rx$, $\beta \neq 1$) and using the formula for the sum of a geometric series, this is at least

$$\frac{\sqrt{r}}{2M} \left| \frac{1-\beta^{\lceil M/r \rceil}}{1-\beta} \right| = \frac{\sqrt{r}}{2M} \frac{\sin(\theta \lceil M/r \rceil / 2)}{\sin(\theta/2)}, \tag{8}$$

where $\theta = \frac{rx \ (\mathrm{mod} \ M)}{M}$ is the angle such that $\beta = e^{i\theta}$ (see Figure 20.5 for a proof by picture of the last equality). Under our assumptions $\lceil M/r \rceil \theta < 1/10$ and hence (using the fact that $\sin \alpha \sim \alpha$ for small angles α), the coefficient of x is at least $\frac{\sqrt{r}}{4M} \lceil M/r \rceil \geq \frac{1}{8\sqrt{r}}$ ∎

20.7.3   Reducing factoring to order finding.

The reduction of the factoring problem to the order-finding problem follows immediately from the following two Lemmas:

LEMMA 20.23
For every nonprime N, the probability that a random X in the set $\mathbb{Z}_N^* = \{X \in [N-1] : gcd(X, N) = 1\}$ has an even order r and furthermore, $X^{r/2} \neq +1 \ (\mathrm{mod} \ N)$ and $X \neq -1 \ (\mathrm{mod} \ N)$ is at least $1/4$.

LEMMA 20.24
For every N and Y, if $Y^2 = 1 \ (\mathrm{mod} \ N)$ but $Y \neq +1 \ (\mathrm{mod} \ N)$ and $Y \neq -1 \ (\mathrm{mod} \ N)$, then $gcd(Y-1, N) > 1$.

   Together, Lemmas 20.23 and 20.24 show that the following algorithm will output a prime factor P of N with high probability: (once we have a single prime factor P, we can run the algorithm again on N/P)

   1. Choose X at random from $[N-1]$.

   2. If $gcd(X, N) > 1$ then let $K = gcd(X, N)$, otherwise compute the order r of X, and if r is even let $K = gcd(X^{r/2} - 1, N)$.

   3. If $K \in \{1, N\}$ then go back to Step 1. If K is a prime then output K and halt. Otherwise, use recursion to output a factor of K.

   Note that if $T(N)$ is the running time of the algorithm then it satisfies the equation $T(N) \leq T(N/2) + \mathrm{polylog}(N)$ leading to $\mathrm{polylog}(N)$ running time.
PROOF OF LEMMA 20.24: Under our assumptions, N divides $Y^2 - 1 = (Y-1)(X+1)$ but does not divide neither $Y-1$ or $Y+1$. But this means that $gcd(Y-1, N) > 1Z$ since if $Y-1$ and N were coprime, then since N divides $(Y-1)(Y+1)$, it would have to divide $X+1$ (Exercise 8). ∎

PROOF OF LEMMA 20.23: We prove this for the case that $N = PQ$ for two primes P, Q (the proof for the general case is similar and is left as Exercise **??**). In this case, by the Chinese Reminder Theorem, if we map every number $X \in \mathbb{Z}_N^*$ to the pair $\langle X \ (\mathrm{mod} \ P), X \ (\mathrm{mod} \ Q) \rangle$ then this map is one-to-one. Also, the groups $\mathbb{Z}_P^*$ and $\mathbb{Z}_Q^*$ are known to by *cyclic* which means that there is a number $g \in [P-1]$ such that the map $j \mapsto g^j \ (\mathrm{mod} \ P)$ is a permutation of $[P-1]$ and similarly there is a number $h \in [Q-1]$ such that the map $k \mapsto h^k \ (\mathrm{mod} \ P)$ is a permutation of $[Q-1]$.
   This means that instead of choosing X at random, we can think of choosing two numbers j, k at random from $[P-1]$ and $[Q-1]$ respectively and consider the pair $\langle g^j \ (\mathrm{mod} \ P), h^k \ (\mathrm{mod} \ Q) \rangle$ which is in one-to-one correspondence with the set of X's in $\mathbb{Z}_N^*$. The order of this pair (or equivalently, of X) is the smallest positive integer r such that $g^{jr} = 1 \ (\mathrm{mod} \ P)$ and $h^{kr} = 1 \ (\mathrm{mod} \ Q)$, which

means that $P - 1 | jr$ and $Q - 1 | kr$. Now suppose that j is odd and k is even (this happens with probability $1/4$). In this case r is of the form $2r'$ where r' is the smallest number such that $P - 1 | 2jr'$ and $Q - 1 | kr'$ (the latter holds since we can divide the two even numbers k and $Q - 1$ by two). But this means that $g^{j(r/2)} \neq 1 \pmod{Q}$ and $h^{k(r/2)} = 1 \pmod{Q}$. In other words, if we let X be the number corresponding to $\langle g^j \pmod{P}, h^k \pmod{Q} \rangle$ then $X^{r/2}$ corresponds to a pair of the form $\langle a, 1 \rangle$ where $a \neq 1$. However, since $+1 \pmod{N}$ corresponds to the pair $\langle +1, +1 \rangle$ and $-1 \pmod{N}$ corresponds to the pair $\langle -1 \pmod{P}, -1 \pmod{Q} \rangle$ it follows that $X^{r/2} \neq \pm 1 \pmod{N}$. $\blacksquare$

20.8 BQP and classical complexity classes

What is the relation between **BQP** and the classes we already encountered such as **P**, **BPP** and **NP**? This is very much an open questions. It not hard to show that quantum computers are at least not infinitely powerful compared to classical algorithms:

THEOREM 20.25
BQP $\subseteq$ PSPACE

PROOF SKETCH: To simulate a T-step quantum computation on an m bit register, we need to come up with a procedure `Coeff` that for every $i \in [T]$ and $x \in \{0, 1\}^m$, the xth coefficient (up to some accuracy) of the register's state in the ith execution. We can compute `Coeff` on inputs x, i using at most 8 recursive calls to `Coeff` on inputs $x', i - 1$ (for the at most 8 strings that agree with x on the three bits that the F_i's operation reads and modifies). Since we can reuse the space used by the recursive operations, if we let $S(i)$ denote the space needed to compute `Coeff`(x, i) then $S(i) \leq S(i - 1) + O(\ell)$ (where ℓ is the number of bits used to store each coefficient).

To compute, say, the probability that if measured after the final step the first bit of the register is equal to 1, just compute the sum of `Coeff`(x, T) for every $x \in \{0, 1\}^n$. Again, by reusing the space of each computation this can be done using polynomial space. $\blacksquare$

Theorem 20.25 can be improved to show that **BQP $\subseteq$ P$^{\#P}$** (where **#P** is the counting version of **NP** described in Chapter 9), but this is currently the best upper bound we know on **BQP**.

Does **BQP = BPP**? The main reason to believe this is false is the polynomial-time algorithm for integer factorization. Although this is not as strong as the evidence for, say **NP $\not\subseteq$ BPP** (after all **NP** contains thousands of well-studied problems that have resisted efficient algorithms), the factorization problem is one of the oldest and most well-studied computational problems, and the fact that we still know no efficient algorithm for it makes the conjecture that none exists appealing. Also note that unlike other famous problems that eventually found an algorithm (e.g., linear programming [?] and primality testing [?]), we do not even have a heuristic algorithm that is conjectured to work (even without proof) or experimentally works on, say, numbers that are product of two random large primes.

What is the relation between **BQP** and **NP**? It seems that quantum computers only offer a quadratic speedup (using Grover's search) on **NP**-complete problems, and so most researchers believe that **NP $\not\subseteq$ BPP**. On the other hand, there is a problem in **BQP** (the Recursive Fourier Sampling or RFS problem [BV97]) that is not known to be in the polynomial-hierarchy , and so at the moment we do not know that **BQP = BPP** even if we were given a polynomial-time algorithm for SAT.

Chapter notes and history

We did not include treatment of many fascinating aspects of quantum information and computation. Many of these are covered in the book by Nielsen and Chuang [NC00]. See also Umesh Vazirani's excellent lecture notes on the topic (available from his home page).

One such area is *quantum error correction*, that tackles the following important issue: how can we run a quantum algorithm when at every possible step there is a probability of noise interfering with the computation? It turns out that under reasonable noise models, one can prove the following *threshold theorem*: as long as the probability of noise at a single step is lower than some constant threshold, one can perform arbitrarily long computations and get the correct answer with high probability [**?**].

Quantum computing has a complicated but interesting relation to cryptography. Although Shor's algorithm and its variants break many of the well known public key cryptosystems (those based on the hardness of integer factorization and discrete log), the features of quantum mechanics can actually be used for cryptographic purposes, a research area called *quantum cryptography* (see [**?**]). Shor's algorithm also spurred research on basing public key encryption scheme on other computational problems (as far as we know, quantum computers do not make the task of breaking most known *private key* cryptosystems significantly easier). Perhaps the most promising direction is basing such schemes on certain problems on integer lattices (see the book [**?**] and [**?**]).

While quantum mechanics has had fantastic success in predicting experiments, some would require more from a physical theory. Namely, to tell us what is the "actual reality" of our world. Many physicists are understandably uncomfortable with the description of nature as maintaining a huge array of possible states, and changing its behavior when it is observed. The popular science book [Bru04] contains a good (even if a bit biased) review of physicists' and philosophers' attempts at providing more palatable descriptions that still manage to predict experiments.

On a more technical level, while no one doubts that quantum effects exist at microscopic scales, scientists questioned why they do not manifest themselves at the macrosopic level (or at least not to human consciousness). A *Scientific American* article by Yam [Yam97] describes various explanations that have been advanced over the years. The leading theory is *decoherence*, which tries to use quantum theory to explain the absence of macroscopic quantum effects. Researchers are not completely comfortable with this explanation. The issue is undoubtedly important to quantum computing, which requires hundreds of thousands of particles to stay in quantum superposition for large-ish periods of time. Thus far it is an open question whether this is practically achievable. One theoretical idea is to treat decoherence as a form of noise, and to build noise-tolerance into the computation —a nontrivial process. For details of this and many other topics, see the books by Kitaev, Shen, and Vyalyi [AA02].

The original motivation for quantum computing was to construct computers that are able to simulate quantum mechanical systems, and this still might be their most important application if they are ever built. Feynman [Fey82] was the first to suggest the possibility that quantum mechanics might allow Turing Machines more computational power than classical TMs. In 1985 Deutsch [Deu85] defined a quantum Turing machine, though in retrospect his definition is unsatisfactory. Better definitions then appeared in Deutsch-Josza [DJ92], Bernstein-Vazirani [BV97] and Yao [Yao93], at which point quantum computation was firmly established as a field.

DRAFT

Exercises

§1 Prove Claim 20.11.

> **Hint:** First prove that Condition 3 holds iff Condition 1 holds iff
> Condition 4 holds. This follows almost directly from the definition
> of the inner product and the fact that for every matrices A, B
> it holds that $(AB)^* = B^* A^*$ and $(A^*)^* = A$. Then prove that
> Condition 3 implies Condition 2, which follows from the fact that
> the norm is invariant under a change of basis. Finally, prove that
> Condition 2 implies Condition 3 by showing that if two orthogonal
> unit vectors $\mathbf{v}, \mathbf{u}$ are mapped to non-orthogonal unit vectors $\mathbf{v}', \mathbf{u}'$,
> then the norm of the vector $\mathbf{u} + \mathbf{v}$ is not preserved.

§2 For each one of the following operations: Hadamard, NOT, controlled-NOT, rotation by $\pi/4$, and Tofolli, write down the 8×8 matrix that describes the mapping induced by applying this operation on the first qubits of a 3-qubit register.

§3 Suppose that a two-bit quantum register is in an arbitrary state $\mathbf{v}$. Show that the following three experiments will yield the same probability of output:

   (a) Measure the register and output the result.
   (b) First measure the first bit and output it, then measure the second bit and output it.
   (c) First measure the second bit and output it, then measure the first bit and output it.

§4 Suppose that f is computed in T time by a quantum algorithm that uses a partial measurements in the middle of the computation, and then proceeds differently according to the result of that measurement. Show that f is computable by $O(T)$ elementary operations.

§5 Prove that if for some $a \in \{0,1\}^n$, the strings $y_1, \ldots, y_{n-1}$ are chosen uniformly at random from $\{0,1\}^n$ subject to $y_i \odot a = 0$ for every $i \in [n-1]$, then with probability at least $1/10$, there exists no nonzero string $a' \neq a$ such that $y_i \odot a' = 0$ for every $i \in [n-1]$. (In other words, the vectors $y_1, \ldots, y_{n-1}$ are linearly independent.)

§6 Prove that given $A, x \in \{0, \ldots, M-1\}$, we can compute $A^x \pmod{M}$ in time polynomial in $\log M$.

> **Hint:** Start by solving the case that $x = 2^k$ for some k. Then,
> show an algorithm for general x by using x's binary expansion.

§7 Prove that for every $\alpha < 1$, there is at most a single rational number a/b such that $b < N$ and $|\alpha - a/b| < 1/(2N^2)$.

§8 Prove that if A, B are numbers such that N and A are coprime but N divides AB, then N divides B.

Hint: Use the fact that if N and A are co-prime then there are whole numbers α, β such that $\alpha N + \beta A = 1$ and multiply this equation by B.

§9 Prove Lemma 20.20.

§10 Complete the proof of Lemma **??** for the case that r and M are not coprime. That is, prove that also in this case there exist at least $\Omega(r/\log r)$ values x's such that $0 \leq rx \pmod{M} \leq r/2$ and $\lceil M/x \rceil$ and r are coprime.

Hint: let $d = gcd(r, M)$, $r' = r/d$ and $M' = M/d$. Now use the same argument as in the case that M and r are coprime to argue that there exist $\Omega(\frac{d \log r}{r})$ values $x \in \mathbb{Z}_{M'}$ satisfying this condition, and that if x satisfies it then so does $x + cM'$ for every c.

§11 (Uses knowledge of continued fractions) Suppose $j, r \leq N$ are mutually coprime and unknown to us. Show that if we know the first $2 \log N$ bits of j/r then we can recover j, r in polynomial time.

20.A Rational approximation of real numbers

A *continued fraction* is a number of the following form:

$$\alpha_1 + \cfrac{1}{\alpha_2 + \cfrac{1}{\alpha_2 + \cfrac{1}{\alpha_3 + \dots}}}$$

Given a real number $\alpha > 0$, we can find its representation as an infinite fraction as follows: split α into the integer part $\lfloor \alpha \rfloor$ and fractional part $\alpha - \lfloor \alpha \rfloor$, find recursively the representation R of $1/(\alpha - \lfloor \alpha \rfloor)$, and then write

$$\alpha = \lfloor \alpha \rfloor + \frac{1}{R}.$$

If we stop after ℓ steps, we get a rational number that can be represented as a/b with a, b coprime. It can be verified that $b \in [2^{\ell/2}, 2^{2\ell}]$. The following theorem is also known:

THEOREM 20.26
[**?**] *If a/b is a rational number obtained by running the continued fraction algorithm on α for a finite number of steps then $|\alpha - a/b| > |\alpha - c/d|$ for every rational number c/d with denominator $d \leq b$.*

This means that given any number α and bound N, we can use the continued fraction algorithm to find in polylog(N) steps a rational number a/b such that $b \leq 16N$ and a/b approximates α better than any other rational number with denominator at most b.

Chapter 21

Logic in complexity theory

VERY SKETCHY

As mentioned in the book's introduction, complexity theory (indeed, all of computer science) arose from developments in mathematical logic in the first half of the century. Mathematical logic continues to exert an influence today, suggesting terminology and choice of problems (e.g., "boolean satisfiability") as well as approaches for attacking complexity's central open questions. This chapter is an introduction to the basic concepts.

Mathematical logic has also influenced many other areas of computer science, such as programming languages, program verification, and model checking. We will not touch upon them, except to note that they supply interesting examples of hard computational problems —ranging from NP-complete to EXPSPACE-complete to undecidable.

The rest of the chapter assumes only a nodding familiarity with logic terminology, which we now recount informally; for details see a logic text.

A *logic* usually refers to a set of rules about constructing *valid sentences*. Here are a few logics we will encounter. *Propositional logic* concerns sentences such as $(p \vee \neg q) \wedge (\neg p \vee r)$ where p, q, r are boolean variables. Recall that the SAT problem consists of determining the satisfiability of such sentences. In *first order logic*, we allow *relation* and *function* symbols as well as *quantification* symbols $\exists$ and $\forall$. For instance, the statement $\forall x S(x) \neq x$ is a first order sentence in which x is quantified universally, $S()$ is a unary relation symbol and $\neq$ is a binary relation. Such logics are used in well-known axiomatizations of mathematics, such as Euclidean geometry, Peano Arithmetic or Zermelo Frankel set theory. Finally, *second order logic* allows sentences in which one is allowed quantification over *structures*, i.e., functions and relations. An example of a second order sentence is $\exists S \forall x S(x) \neq x$, where S is a unary relation symbol.

A sentence (or collection of sentences) in a logic has no intrinsic "meaning." The meaning —including truth or falsehood—can be discussed only with reference to a *structure*, which gives a way of interpreting all symbols in the sentence. To give an example, Peano arithmetic consists of five sentences ("axioms") in a logic that consists of symbols like $S(x)$, =, + etc. The standard structure of these sentences is the set of positive integers, with $S()$ given the intepretation of "successor function," + given the interpretation of addition, and so on. A structure is said to be a *model* for a sentence or a group of sentences if those sentences are true in that structure.

Finally, a *proof system* consists of a set of sentences Σ called *axioms* and one or more *derivation*

rules for deriving new sentences from the axioms. We say that sentence σ can be *proved from* Σ, denoted $\Sigma \vdash \sigma$, if it can be derived from Σ using a finite number of applications of the derivation rules. A proveable sentence is called a *theorem*.

Note that a theorem is a result of a mechanical (essentially, algorithmic) process of applying derivation rules to the axioms. There is a related notion of whether or not σ is *logically implied by* Σ, denoted $\Sigma \models \sigma$, which means that every model of Σ is also a model of σ. In other words, there is no "counterexample model" in which the axioms Σ are true but σ is not. The two notions are in general different but Gödel in his *completeness* theorem for first order theories exhibited a natural set of derivation rules such that logically implied sentences are exactly the set of theorems. (This result was a stepping stone to his even more famous *incompleteness* theorem.)

Later in this chapter we give a complexity-theoretic definition of a proof system, and introduce the area of *proof complexity* that studies the *size* of the smallest proof of a mathematical statement in a given proof system.

21.1 Logical definitions of complexity classes

Just as Church and others defined computation using logic without referring to any kind of computing machine, it is possible to give "machineless" characterizations of many complexity classes using logic. We describe a few examples below.

21.1.1 Fagin's definition of NP

In 1974, just as the theory of **NP**-completeness was coming into its own, Fagin showed how to define **NP** using second-order logic. We describe his idea using an example.

EXAMPLE 21.1
(Representing 3-COLOR) We show how to represent the set of 3-colorable graphs using second order logic.

Let E be a symbol for a binary relation, and C_0, C_1, C_2 be symbols for unary relations, and $\phi(E, C_0, C_1, C_2)$ be a first order formula that is a conjunction of the following formulae where $i+1, i+2$ are meant to be understood modulo 3:

$$\forall u,v \quad E(u,v) = E(v,u) \tag{1}$$

$$\forall u \quad \wedge_{i=1,2,3} (C_i(u) \Rightarrow \neg(C_{i+1}(u) \vee C_{i+2}(u))) \tag{2}$$

$$\forall u C_i(u) \vee C_{i+1}(u) \vee C_{i+2}(u) \tag{3}$$

$$\forall u,v \quad E(u,v) \Rightarrow \wedge_{i=1,2,3}(C_i(u) \Rightarrow \neg C_i(v)) \tag{4}$$

What set of E's defined on a finite set satisfy $\exists C_0 \exists C_1 \exists C_2 \phi(E, C_0, C_1, C_2)$? If E is defined on a universe of size n (i.e., u, v take values in this universe) then (1) says that E is symmetric, i.e., it may be viewed as the edge set of an undirected graph on n vertices. Conditions (2) and (3) say that C_0, C_1, C_2 partition the vertices into three classes. Finally, condition (4) says that the partition is a valid coloring.

Now we can sketch the general result. To represent a general **NP** problem, there is a unary relation symbol that represents the input (in the above case, E). The witness is a tableau (see Chapter 2) of an accepting computation. If the tableau has size n^k, the witness can be represented by a k-ary relation (in the above case the witness is a 3-coloring, which has representation size $3n$ and hence was represented using 3 unary relations). The first order formula uses the Cook-Levin observation that the tableau is correct iff it is correct in all 2×3 "windows".

The formal statement of Fagin's theorem is as follows; the proof is left as an exercise.

THEOREM 21.2 (FAGIN)
To be written.

21.1.2   MAX-SNP

21.2   Proof complexity as an approach to NP versus coNP

Proof complexity tries to study the size of the smallest proof of a statement in a given proof system. First, we need a formal definition of what a proof system is. The following definition due to Cook and Reckow focuses attention on the intuitive property that a mathematical proof is "easy to check."

DEFINITION 21.3
A *proof system* consists of a polynomial-time Turing machine M. A statement T is said to be a *theorem* of this proof system iff there is a string $\pi \in \{0,1\}^*$ such that M accepts (T, π).

If T is a theorem of proof system M, then the *proof complexity of T with respect to M* is the minimun k such that there is some $\pi \in \{0,1\}^k$ for which M accepts (T, π).

Note that the definition of theoremhood ignores the issue of the length of the proof, and insists only that the M's running time is polynomial in the input length $|T| + |\pi|$. The following is an easy consequence of the definition and the motivation for much of the field of proof complexity.

THEOREM 21.4
*A proof system M in which $\overline{\mathsf{SAT}}$ has polynomial proof complexity exists iff **NP** = **coNP**.*

Many branches of mathematics, including logic, algebra, geometry, etc. give rise to proof systems. Algorithms for SAT and *automated theorem provers* (popular in some areas of computer science) also may be viewed as proof systems.

21.2.1   Resolution

This concerns

21.2.2   Frege Systems

21.2.3   Polynomial calculus

21.3   Is P $\neq$ NP unproveable?

DRAFT

Chapter 22

Why are circuit lowerbounds so difficult?

Why have we not been able to prove strong lower bounds for circuits? In 1994 Razborov and Rudich formalized the notion of a "natural mathematical proof," for a circuit lowerbound. They pointed out that current lowerbound arguments involve "natural" mathematical proofs, and show that obtaining strong lowerbound with such techniques would violate a widely believed cryptographic assumption (namely, that factoring integers requires time 2^{n^ϵ} for some fixed $\epsilon > 0$). Thus presumably we need to develop mathematical arguments that are not natural. This result may be viewed as a modern analogue of the Baker, Gill, Solovay result from the 1970s (see Chapter **??**) that showed that diagonalization alone cannot resolve **P** versus **NP** and other questions.

Basically, a natural technique is one that proves a lowerbound for a random function and is "constructive." We formalize "constructive" later but first consider why lowerbound proofs may need to work for random functions.

22.1 Formal Complexity Measures

Let us imagine at a high level how one might approach the project of proving circuit lower bounds. For concreteness, focus on formulas, which are boolean circuits where gates have indegree 2 and outdegree 1. It is tempting to use some kind of induction. Suppose we have a function like the one in Figure 22.1 that we believe to be "complicated." Since the function computed at the output is "complicated", intuition says that at least one of the functions on the incoming edges to the output gate should also be "pretty complicated" (after all those two functions can be combined with a single gate to produce a "complicated" function). Now we try to formalize this intuition, and point out why one ends up proving a lowerbound on the formula complexity of random functions.

The most obvious way to formalize a "complicatedness" is as a function μ that maps every boolean function on $\{0,1\}^n$ to a nonnegative integer. (The input to μ is the truth table of the function.) We say that μ is a *formal complexity measure* if it satisfies the following properties: First, the measure is low for trivial functions: $\mu(x_i) \leq 1$ and $\mu(\bar{x}_i) \leq 1$ for all i. Second, we require that

Figure unavailable in pdf file.

Figure 22.1: A formula for a hard function.

- $\mu(f \wedge g) \leq \mu(f) + \mu(g)$ for all f, g; and

- $\mu(f \vee g) \leq \mu(f) + \mu(g)$ for all f, g.

For instance, the following function ρ is trivially a formal complexity measure

$$\rho(f) = 1 + \text{the smallest formula size for } f. \tag{1}$$

In fact, it is easy to prove the following by induction.

THEOREM 22.1
If μ is any formal complexity measure, then $\mu(f)$ is a lowerbound on the formula complexity of f.

Thus to formalize the inductive approach outlined earlier, it suffices to define a measure μ such that $\mu(\text{CLIQUE})$ is high (say superpolynomial). For example, one could try "fraction of inputs for which the function agrees with the CLIQUE function" or some suitably modified version of this. In general, one imagines that defining a measure that lets us prove a good lowerbound for CLIQUE would involve some deep observation about the CLIQUE function. The next lemma seems to show, however, that even though all we care about is the CLIQUE function, our lowerbound necessarily must reason about random functions.

LEMMA 22.2
Suppose μ is a formal complexity measure and there exists a function $f : \{0,1\}^n \rightarrow \{0,1\}$ such that $\mu(f) \geq c$ for some large number c. Then for at least $1/4$ of all functions $g : \{0,1\}^n \rightarrow \{0,1\}$ we must have $\mu(g) \geq c/4$.

PROOF: Let $g : \{0,1\}^n \rightarrow \{0,1\}$ be any function. Write f as $f = h \oplus g$ where $h = f \oplus g$. So $f = (\bar{h} \wedge g) \vee (h \wedge \bar{g})$ and $\mu(f) \leq \mu(g) + \mu(\bar{g}) + \mu(h) + \mu(\bar{h})$.

Now suppose for contradiction's sake that $\{g : \mu(g) < c/4\}$ contains more than $3/4$ of all boolean functions on n-bit inputs. If we pick the above function g randomly, then $\bar{g}, h, \bar{h}$ are also random (though not independent). Using the trivial union bound we have $\Pr[\text{All of } h, \bar{h}, g, \bar{g} \text{ have } \mu < c/4] > 0$. Hence $\mu(f) < c$, which contradicts the assumption. Thus the lemma is proved. ∎

In fact, the following stronger theorem holds:

THEOREM 22.3
If $\mu(f) > c$ then for all $\epsilon > 0$ and for at least $1 - \epsilon$ of all functions g we have that,

$$\mu(g) \geq \Omega\left(\frac{c}{(n + \log(1/\epsilon))^2}\right).$$

The idea behind the proof of the theorem is to write f as the boolean combination of a small number of functions and then proceed similarly as in the proof of the lemma.

22.2   Natural Properties

Moving the above discussion forward, we think of a lowerbound proof as identifying some property of "hard" functions that is not shared by "easy" functions.

DEFINITION 22.4
A *property* Φ is a map from boolean functions to $\{0, 1\}$. A **P**-*natural property useful against* **P**/*poly* is a property Φ such that:

1. $\Phi(f) = 1$ for at least a $1/2^n$ fraction of all boolean functions on n bits (recall that there are 2^{2^n} functions on n bits);

2. $\Phi(f) = 1$ implies that $f \notin \mathbf{P}/poly$ (or more concretely, that f has circuit complexity at least $n^{\log n}$, say); and

3. Φ is computable on n-bit functions in $2^{O(n)}$ time (i.e., polynomial in the length of the function's truth table).

The term **P**-natural refers to requirement (3). The property is useful against **P**/*poly* because of requirement (2). (Note that this requirement also ensures that Φ is not trivial, since it must be 0 for functions in **P**/*poly*.) Requirement (1) corresponds to our above intuition that circuit lowerbounds should prove the hardness of a random function.

By suitably modifying (2) and (3) we can analogously define, for any complexity class $\mathcal{C}_1$ and circuit class $\mathcal{C}_2$, a $\mathcal{C}_1$-natural property that is useful against circuit class $\mathcal{C}_2$. We emphasize that when the property is computed, the input is the truth table of a function, whose size is 2^n. Thus a **P**-natural property is computed in time 2^{cn} for some constant $c > 1$ and a **PSPACE**-natural property is computed in space 2^{cn}.

EXAMPLE 22.5
The result that PARITY is not computable in $\mathbf{AC}^0$ (Section **??**) involved the following steps. (a) Show that every $\mathbf{AC}^0$ circuit can be simplified by restricting at most $n - n^\epsilon$ input bits so that it then becomes a constant function. (b) Show that the PARITY function does not have this property.

Thus the natural property lurking in this proof is the following: $\Phi(f) = 1$ iff for every way of assigning values to at most $n - n^\epsilon$ input bits the function does not become a constant function. Clearly, if $\Phi(f) = 1$ then $f \notin \mathbf{AC}^0$, so f is useful against $\mathbf{AC}^0$. Furthermore, Φ can be computed in $2^{O(n)}$ time — just enumerate all possible choices for the subsets of variables and all ways of setting them to 0/1. This running time is polynomial in the length of the truth-table, so Φ is **P**-natural. Finally, requirement (1) is also met since almost all boolean functions satisfy $\Phi(f) = 1$ (easy to check using a simple probability calculation; left as exercise).

Thinking further, we see that Φ is a $\mathbf{AC}^0$-natural property that is useful against $\mathbf{AC}^0$.

EXAMPLE 22.6

The lowerbound for $\mathbf{ACC}^0$ circuits described in Section **??** is not natural *per se*. Razborov and Rudich show how to *naturalize* the proof, in other words change it —while retaining its essence—so that it does use a natural property. Recall that every boolean function on n bits can be represented by a multilinear polynomial over $GF(3)$. The space of all n-variate multilinear polynomials forms a vector space, whose dimension is $N = 2^n$. Then all multilinear polynomials in n variables of total degree less than $n/2$ form a subspace of dimension $N/2$ (this assumes n is even), and we denote this space by L. For a boolean function f let $\hat{f}$ be a multilinear polynomial over $GF(3)$ that represents f. Then define $\Phi(F) = 1$ iff the dimension of the space

$$\left\{ \hat{f}l_1 + l_2 : l_1, l_2 \in L \right\}$$

is at least $3N/4$. It can be checked that Φ is 1 for the parity function, as well as for most random functions. Furthermore, rank computations can be done in $\mathbf{NC}^2$ so it is $\mathbf{NC}^2$-natural. The technique of Section **??** can be used to show that if $\Phi(f) = 1$ then $f \notin \mathbf{ACC}^0[3]$; thus Φ is useful against $\mathbf{ACC}^0[3]$.

EXAMPLE 22.7

The lowerbound for monotone circuits in Section **??** does use constructive methods, but it is challenging to show that it applies to a random function since a random function is not monotone. Nobody has formulated a good definition of a random monotone function.

In the definition of natural proofs, requirement (3) is the most controversial in that there is no inherent reason why mathematical proofs should go hand in hand with efficient algorithms.

REMARK 22.8

"Constructive mathematics" was a movement within mathematics that rejected any proofs of existence that did not yield an algorithm for constructing the object. Today this viewpoint is considered quaint; nonconstructive proofs are integral to mathematics.

In our context, "constructive" has a stricter meaning, namely the proof has to yield a polynomial-time algorithm. Many proofs that would be "constructive" for a mathematician would be nonconstructive under our definition. Surprisingly, even with this stricter definition, proofs in combinatorial mathematics are usually constructive, and —as Razborov and Rudich are pointing out —the same is true of current circuit lowerbounds as well.

In a few cases, combinatorial results initially proved "nonconstructively" later turned out to have constructive proofs: a famous example is the Lovàsz Local Lemma (discovered in 1974; algorithmic version is in Beck [Bec91]). The same is true for several circuit lowerbounds—cf. the "naturalized" version of the Razborov-Smolensky lowerbound for $\mathbf{ACC}^0[q]$ mentioned earlier, and Raz's proof [Raz00] of the Babai-Nisan-Szegedy [BNS] lowerbound on multiparty communication complexity.

22.3   Limitations of Natural Proofs

The following theorem by Razborov and Rudich explains why we have not been able to use the same techniques to obtain an upper bound on **P**/*poly*: constructing a **P**-natural property useful against **P**/*poly* violates widely believed cryptographic assumptions.

THEOREM 22.9 (RAZBOROV, RUDICH [RR97])
*Suppose a **P**-natural property Φ exists that is useful against **P**/poly. Then there are no strong pseudorandom function generators. In particular, FACTORING and DISCRETE LOG can be solved in less than 2^{n^ϵ} time for all $\epsilon > 0$.*

Pseudorandom function generators were defined in Section **??**. The definition used a distinguisher polynomial-time machine that is given oracle access to either a truly random function or a function from the pseudorandom family. The family is termed *pseudorandom* if the distinguisher cannot distinguish between the two oracles. Now we tailor that more general definition for our narrow purposes in this section. We allow the distinguisher $2^{O(n)}$ time and even allow it to examine the truth table of the function! This is without loss of generality since in $2^{O(n)}$ time the distinguisher could construct the truth table using 2^n queries to the oracle.

DEFINITION 22.10
A *pseudorandom function generator* is a function $f(k, x)$ computable in polynomial time where the input x has n bits and the "key" k has n^c bits, where $c > 2$ is a fixed constant. Denoting by F_n the function obtained by uniformly selecting $k \in \{0,1\}^{n^c}$ and setting F_n to $f(k, \cdot)$, we have the property that the function ensemble $F = \{F_n\}_{n=1}^\infty$ is "pseudorandom," namely, for each Turing machine M running in time $2^{O(n)}$, and for all sufficiently large n,

$$|\Pr[M(F_n) = 1] - \Pr[M(H_n) = 1]| < \frac{1}{2^{n^2}},$$

where H_n is a random function on $\{0,1\}^n$.
   We will denote $f(k, \cdot)$ by f_k.

Intuitively, the above definition says that if f is a pseudorandom function generator, then for a random k, the probability is high that f_k "looks like a random function" to all Turing machines running in time $2^{O(n)}$. Note that f_k cannot look random to machines that run in $2^{O(n^c)}$ time since they can just guess the key k. Thus restricting the running time to $2^{O(n)}$ (or to some other fixed exponential function such as $2^{O(n^2)}$) is crucial.
   Recall that Section **??** described the Goldreich-Goldwasser-Micali construction of pseudorandom function generators $f(k, x)$ using a pseudorandom generator g that stretches n^c random bits to $2n^c$ pseudorandom (also see Figure 22.2): Let $g_0(k)$ and $g_1(k)$ denote, respectively, the first and last n^c bits of $g(k)$. Then the following function is a pseudorandom function generator, where $\text{MSB}(x)$ refers to the first bit of a string x:

$$f(k, x) = \text{MSB}(g_{x_n} \circ g_{x_{n-1}} \circ \cdots \circ g_{x_2} \circ g_{x_1}(k)).$$

The exercises in Chapter 10 explored the security of this construction as a function of the security parameter of g; basically, the two are essentially the same. By the Goldreich-Levin theorem

Figure unavailable in pdf file.

Figure 22.2: Constructing a pseudorandom function generator from a pseudorandom generator.

of Section **??**, a pseudorandom generator with such a high security parameter exists if a oneway permutation exists and some $\epsilon > 0$, such that every 2^{n^ϵ} time algorithm has inversion probability less than 2^{-n^ϵ}. The DISCRETE LOG function —a permutation— is conjectured to satisfy this property. As mentioned in Chapter 10, researchers believe that there is a small $\epsilon > 0$ such that the worst-case complexity of DISCRETE LOG is 2^{n^ϵ}, which by random self-reducibility also implies the hardness of the average case. (One can also obtain pseudorandom generators using FACTORING, versions of which are also believed to be just as hard as DISCRETE LOG.) If this belief is correct, then pseudorandom function generators exist as outlined above. (Exercise.)

Now we can prove the above theorem.

THEOREM 22.9: Suppose the property Φ exists, and f is a pseudorandom function generator. We show that a Turing machine can use Φ to distinguish f_k from a random function. First note that $f_k \in \mathbf{P}/poly$ for every k (just hardwire k into the circuit for f_k) so the contrapositive of property (2) implies that $\Phi(f_k) = 0$. In addition, property (1) implies that $\Pr_{H_n}[\Phi(H_n) = 1] \geq 1/2^n$. Hence,

$$\Pr_{H_n}[\Phi(H_n)] - \Pr_{k \in \{0,1\}^{n^c}}[\Phi(f_k)] \geq 1/2^n,$$

and thus Φ is a distinguisher against f. ∎

22.4 My personal view

Discouraged by the Razborov-Rudich result, researchers (myself included) hardly ever work on circuit lowerbounds. Lately, I have begun to think this reaction was extreme. I still agree that a circuit lowerbound for say CLIQUE, if and when we prove it, will very likely apply to random functions as well. Thus the way to get around the Razborov-Rudich observation is to define properties that are not **P**-natural; in other words, are *nonconstructive*. I feel that this need not be such an insurmountable barrier since a host of mathematical results are nonconstructive.

Concretely, consider the question of separating **NEXP** from $\mathbf{ACC}^0$, one of the (admittedly not very ambitious) frontiers of circuit complexity outlined in Chapter 13. As observed there, $\mathbf{NEXP} \neq \mathbf{ACC}^0$ will follow if we can improve the Babai-Nisan-Szegedy lowerbound of $\Omega(n/2^k)$ for k-party communication complexity to $\Omega(n/\text{poly}(k))$ for some function in **NEXP**. One line of attack is to lowerbound the *discrepancy* of all large cylinder intersections in the truth table, as we saw in Raz's proof of the BNS lowerbound[1]. (In other words, the "unnatural" property we are defining is Φ where $\Phi(f) = 1$ iff f has high discrepancy and thus high multiparty communication complexity.) For a long time, I found this question intimidating because the problem of computing the discrepancy given the truth table of the function is **coNP**-hard (even for $k = 2$). This seemed

---

[1]Interestingly, Raz discovered this naturalization of the BNS proof after being briefly hopeful that the original BNS proof—which is not natural— may allow a way around the Razborov-Rudich result.

to suggest that a proof that the discrepancy is high for an explicit function (which presumably will also show that it is high for random functions) must have a nonconstructive nature, and hence will be very difficult. Lately, I have begun to suspect this intuition.

A relevant example is Lovàsz's lowerbound of the chromatic number of the Kneser graph [Lov78]. Lowerbounding the chromatic number is **coNP**-complete in general. Lovàsz gives a topological proof (using the famous Borsuk-Ulam fixed point theorem) that determines the chromatic number of the Kneser graph exactly. From his proof one can indeed obtain an algorithm for solving chromatic number on all graphs([MZ02]) —but it runs in **PSPACE** for general graphs! So if this were a circuit lowerbound we could call it **PSPACE**-natural, and thus "nonconstructive." Nevertheless, Lovàsz's reasoning for the particular case of the Kneser graph is not overly complicated because the graph is highly symmetrical. This suggests we should not blindly trust the intuition that "nonconstructive ≡ difficult."

I fervently hope that the next generation of researchers will view the Razborov-Rudich theorem as a guide rather than as a big obstacle!

Exercises

§1 Prove Theorem 22.3.

§2 Prove that a random function satisfies $\Phi(f) = 1$ with high probability, where Φ is the property defined in Example 22.5.

§3 Show that if the hardness assumption for discrete log is true, then pseudorandom function generators as defined in this chapter exist.

§4 Prove Wigderson's observation: **P**-natural properties cannot prove that DISCRETE LOG requires circuits of 2^{n^ϵ} size.

Hint: if DISCRETE LOG is hard on worst-case inputs then it is hard on most inputs, and then it can be used to construct pseudo-random functions.

§5 (Razborov [Raz92]) A *submodular* complexity measure is a complexity measure that satisfies $\mu(f \vee g) + \mu(f \wedge g) \leq \mu(f) + \mu(g)$ for all functions f, g. Show that for every n-bit function f_n, such a measure satisfies $\mu(f_n) = O(n)$.

Hint: It suffices to prove this when f_n is a random function. Use induction on the number of variables, and the fact that both f_n and $\underline{f_n}$ are random functions.

Chapter notes and history

The observation that circuit lowerbounds may unwittingly end up reasoning about random functions first appears in Razborov [Raz89]'s result about the limitations of the method of approximation.

We did not cover the full spectrum of ideas in the Razborov-Rudich paper [RR97], where it is observed that candidate pseudorandom function generators exist even in the class TC^0, which lies between $\mathbf{ACC}^0$ and $\mathbf{NC}^1$. Thus natural proofs will probably not allow us to separate even TC^0 from $\mathbf{P}$.

Razborov's observation about submodular measures in Problem 5 is important because many existing approaches for formula complexity use submodular measures; thus they will fail to even prove superlinear lowerbounds.

In contrast with my limited optimism, Razborov himself expresses (in the introduction to [Raz03]) a view that the obstacle posed by the natural proofs observation is very serious. He observes that existing lowerbound approaches use weak theories of arithmetic such as Bounded Arithmetic. He conjectures that any circuit lowerbound attempt in such a logical system must be natural (and hence unlikely to work). But as I mentioned, several theorems even in discrete mathematics use reasoning (e.g., fixed point theorems like Borsuk-Ulam) that does not seem to be formalizable in Bounded Arithmetic. Thus is my reason for optimism.

However, somen other researchers are far more pessimistic: they fear that $\mathbf{P}$ versus $\mathbf{NP}$ may be independent of mathematics (say, of Zermelo-Fraenkel set theory). Razborov says that he has no intuition about this.

DRAFT

Appendices

Appendix A

Mathematical Background.

This appendix reviews the mathematical notions used in this book. However, most of these are only used in few places, and so the reader might want to only quickly review Sections A.1, A.2 and A.3, and come back to the other sections as needed. In particular, apart from probability, the first part of the book essentially requires only comfort with mathematical proofs and some very basic notions of discrete math.

The topics described in this appendix are covered in greater depth in many texts and online sources. Almost all of the mathematical background needed is covered in a good undergraduate "discrete math for computer science" course as currently taught at many computer science departments. Some good sources for this material are the lecture notes by Papadimitriou and Vazirani [PV06], Lehman and Leighton [LL06] and the book of Rosen [Ros06].

Although knowledge of algorithms is not strictly necessary for this book, it would be quite useful. It would be helpful to review either one of the two excellent recent books by Dasgupta et al [DPV06] and Kleinberg and Tardos [KT06] or the earlier text by Cormen et al [CLRS01]. This book does not require prior knowledge of computability and automata theory, but some basic familiarity with that theory could be useful: see Sipser's book [SIP96] for an excellent introduction. Mitzenmacher and Upfal [MU05] and Prabhakar and Raghavan [**?**] cover both algorithmic reasoning and probability. For more insight on discrete probability, see the book by Alon and Spencer [AS00].

A.1 Mathematical Proofs

Perhaps *the* mathematical prerequisite needed for this book is a certain level of comfort with mathematical proofs. While in everyday life we might use "proof" to describe a fairly convincing argument, in mathematics a proof is an argument that is convincing *beyond any shadow of a doubt*.[1] For example, consider the following mathematical statement:

> *Every even number greater than* 2 *is equal to the sum of two primes.*

---

[1] In a famous joke, as a mathematician and an engineer drive in Scotland they see a white sheep on their left side. The engineer says "you see: all the sheep in Scotland are white". The mathematician replies "All I see is that there exists a sheep in Scotland whose right side is white".

This statement, known as "Goldbach's Conjecture", was conjectured to be true by Christian Goldbach in 1742. In the more than 250 years that have passed since, no one has ever found a counterexample to this statement. In fact, it has been verified to be true for all even numbers from 4 till $100,000,000,000,000,000$. Yet still it is not considered proven, since we have not ruled out the possibility that there is some (very large) even number that cannot be expressed as the sum of two primes.

The fact that a mathematical proof has to be absolutely convincing does not mean that it has to be overly formal and tedious. It just has to be clearly written, and contain no logical gaps. When you write proofs try to be clear and concise, rather than using too much formal notation. When you read proofs, try to ask yourself at every statement "am I really convinced that this statement is true?".

Of course, to be absolutely convinced that some statement is true, we need to be certain of what that statement means. This why there is a special emphasis in mathematics on very precise *definitions*. Whenever you read a definition, try to make sure you completely understand it, perhaps by working through some simple examples. Oftentimes, understanding the meaning of a mathematical statement is more than half the work to prove that it is true.

EXAMPLE A.1
Here is an example for a classical mathematical proof, written by Euclid around 300 B.C. Recall that a *prime number* is an integer $p > 1$ whose only divisors are p and 1, and that every number n is a product of prime numbers. Euclid's Theorem is the following:

THEOREM A.2
There exist infinitely many primes.

Before proving it, let's see that we understand what this statement means. It simply means that for every natural number k, there are more than k primes, and hence the number of primes is not finite.

At first, one might think it's obvious that there are infinitely many primes because there are infinitely many natural numbers, and each natural number is a product of primes. However, this is faulty reasoning: for example, the set of numbers of the form 3^n is infinite, even though their only factor is the single prime 3.

To prove Theorem A.2, we use the technique of *proof by contradiction*. That is, we assume it is false and try to derive a contradiction from that assumption. Indeed, assume that all the primes can be enumerated as $p_1, p_2, \ldots, p_k$ for some number k. Define the number $n = p_1 p_2 \cdots p_k + 1$. Since we assume that the numbers $p_1, \ldots, p_k$ are *all* the primes, all of n's prime factors must come from this set, and in particular there is some i between 1 and k such that p_i divides n. That is, $n = p_i m$ for some number m. Thus,

$$p_i m = p_1 p_2 \cdots p_k + 1$$

or equivalently,

$$p_i m - p_1 p_2 \cdots p_k = 1 \,.$$

But dividing both sides of this equation by p_i, we will get a whole number on the left hand side (as p_i is a factor of $p_1 p_2 \cdots p_k$) and the fraction $1/p_i$ on the right hand side, deriving a contradiction. This allows us to rightfully place the QED symbol ∎ and consider Theorem A.2 as proven.

A.2 Sets, Functions, Pairs, Strings, Graphs, Logic.

A *set* contains a finite or infinite number of elements, without repetition or respect to order, for example $\{2, 17, 5\}$, $\mathbb{N} = \{1, 2, 3, \ldots\}$ (the set of natural numbers), $[n] = \{1, 2, \ldots, n\}$ (the set of natural numbers from 1 ro n), $\mathbb{R}$ (the set of real numbers). For a finite set A, we denote by $|A|$ the number of elements in A. Some operations on sets are: **(1)** *union*: $A \cup B = \{x : x \in A \text{ or } x \in B\}$, **(2)** *intersection* : $A \cap B = \{x : x \in A \text{ and } x \in B\}$, and **(3)** *substraction*: $A \setminus B = \{x : x \in A \text{ and } x \notin B\}$.

We say that f is a *function* from a set A to B, denoted by $f : A \to B$, if it maps any element of A into an element of B. If B and A are finite, then the number of possible functions from A to B is $|B|^{|A|}$. We say that f is *one to one* if for every $x, w \in A$ with $x \neq w$, $f(x) \neq f(w)$. If A, B are finite, the existence of such a function implies that $|A| \leq |B|$. We say that f is *onto* if for every $y \in B$ there exists $x \in A$ such that $f(x) = y$. If A, B are finite, the existence of such a function implies that $|A| \geq |B|$. We say that f is a *permutation* if it is both one-to-one and onto. For finite A, B, the existence of a permutation from A to B implies that $|A| = |B|$.

If A, B are sets, then the $A \times B$ denotes the set of all ordered pairs $\langle a, b \rangle$ with $a \in A, b \in B$. Note that if A, B are finite then $|A \times B| = |A| \cdot |B|$. We can define similarly $A \times B \times C$ to be the set of ordered triples $\langle a, b, c \rangle$ with $a \in A, b \in B, c \in C$. For $n \in \mathbb{N}$, we denote by A^n the set $A \times A \times \cdots \times A$ (n times). We will often use the set $\{0, 1\}^n$, consisting of all length-n sequences of bits (i.e., length n strings), and the set $\{0, 1\}^* = \cup_{n \geq 0} \{0, 1\}^n$ ($\{0, 1\}^0$ has a single element: a binary string of length zero, which we call the empty word and denote by ε).

A *graph* G consists of a set V of *vertices* (which we often assume is equal to the set $[n] = \{1, \ldots, n\}$ for some $n \in N$) and a set E of *edges*, which consists of unordered pairs (i.e., size two subsets) of elements in V. We denote the edge $\{u, v\}$ of the graph by $\overline{uv}$. For $v \in V$, the *neighbors* of v are all the vertices $u \in V$ such that $\overline{uv} \in E$. In a *directed graph*, the edges consist of *ordered pairs* of vertices, to stress this we sometimes denote the edge $\langle u, v \rangle$ in a directed graph by $\overrightarrow{uv}$. One can represent an n-vertex graph G by its *adjacency matrix* which is an $n \times n$ matrix A such that $A_{i,j}$ is equal to 1 if the edge $\overrightarrow{ij}$ is present in G i^{th} and is equal to 0 otherwise. One can think of an undirected graph as a directed graph G that satisfies that for every u, v, G contains the edge $\overrightarrow{uv}$ if and only if it contains the edge $\overrightarrow{vu}$. Hence, one can represent an undirected graph by an adjacency matrix that is *symmetric* ($A_{i,j} = A_{j,i}$ for every $i, j \in [n]$).

A *Boolean variable* is a variable that can be either TRUE or FALSE (we sometimes identify TRUE with 1 and FALSE with 0). We can combine variables via the logical operations AND ($\wedge$), OR ($\vee$) and NOT ($\neg$, sometimes also denoted by an overline), to obtain *Boolean formulae*. For example, the following is a Boolean formulae on the variables u_1, u_2, u_3: $(u_1 \wedge \overline{u}_2) \vee \neg(u_3 \wedge \overline{u}_1)$. The definitions of the operations are the usual: $a \wedge b = $ TRUE if $a = $ TRUE and $b = $ TRUE and is equal to FALSE

otherwise; $\bar{a} = \neg a = \text{TRUE}$ if $a = \text{FALSE}$ and is equal to FALSE otherwise; $a \vee b = \neg(\bar{a} \vee \bar{b})$. If φ is a formulae in n variables $u_1, \ldots, u_n$, then for any *assignment* of values $u \in \{\text{FALSE, TRUE}\}^n$ (or equivalently, $\{0,1\}^n$), we denote by $\varphi(u)$ the value of φ when its variables are assigned the values in u. We say that φ is *satisfiable* if there exists a u such that $\varphi(u) = \text{TRUE}$.

We will often use the *quantifiers* $\forall$ (for all) and $\exists$ (exists). That is, if φ is a condition that can be TRUE or FALSE depending on the value of a variable x, then we write $\forall_x \varphi(x)$ to denote the statement that φ is TRUE for *every* possible value that can be assigned to x. If A is a set then we write $\forall_{x \in A} \varphi(x)$ to denote the statement that φ is TRUE for every assignment for x from the set A. The quantifier $\exists$ is defined similarly. Formally, we say that $\exists_x \varphi(x)$ holds if and only if $\neg(\forall_x \neg \varphi(x))$ holds.

A.3   Probability theory

A *finite probability space* is a finite set $\Omega = \{\omega_1, \ldots, \omega_N\}$ along with a set of numbers $p_1, \ldots, p_N \in [0,1]$ such that $\sum_{i=1}^{N} p_i = 1$. A random element is selected from this space by choosing ω_i with probability p_i. If x is chosen from the sample space Ω then we denote this by $x \in_R \Omega$. If no distribution is specified then we use the uniform distribution over the elements of Ω (i.e., $p_i = \frac{1}{N}$ for every i).

An *event* over the space Ω is a subset $A \subseteq \Omega$ and the *probability* that A occurs, denoted by $\Pr[A]$, is equal to $\sum_{i:\omega_i \in A} p_i$. To give an example, the probability space could be that of all 2^n possible outcomes of n tosses of a fair coin (i.e., $\Omega = \{0,1\}^n$ and $p_i = 2^{-n}$ for every $i \in [2^n]$) and the event A can be that the number of coins that come up "heads" (or, equivalently, 1) is even. In this case, $\Pr[A] = 1/2$ (exercise). The following simple bound —called the *union bound*—is often used in the book. For every set of events $A_1, A_2, \ldots, A_n$,

$$\Pr[\cup_{i=1}^n A_i] \leq \sum_{i=1}^n \Pr[A_i]. \tag{1}$$

Inclusion exclusion principle.   The union bound is a special case of a more general principle. Indeed, note that if the sets $A_1, \ldots, A_n$ are not *disjoint* then the probability of $\cup_i A_i$ could be smaller than $\sum_i \Pr[A_i]$ since we are overcounting elements that appear in more than one set. We can correct this by substracting $\sum_{i<j} \Pr[A_i \cap A_j]$ but then we might be undercounting, since we subtracted elements that appear in at least 3 sets too many times. Continuing this process we get

CLAIM A.3 (INCLUSION-EXCLUSION PRINCIPLE)
For every $A_1, \ldots, A_n$,

$$\Pr[\cup_{i=1}^n A_i] = \sum_{i=1}^n \Pr[A_i] - \sum_{1 \leq i < j \leq n} \Pr[A_i \cap A_j] + \cdots + (-1)^{n-1} \Pr[A_1 \cap \cdots \cap A_n].$$

Moreover, this is an alternating sum *which means that if we take only the first k summands of the right hand side, then this upperbounds the left-hand side if k is odd, and lowerbounds it if k is even.*

We sometimes use the following corollary of this claim:

CLAIM A.4
For every events $A_1, \ldots, A_n$,

$$\Pr[\cup_{i=1}^n A_i] \geq \sum_{i=1}^n \Pr[A_i] - \sum_{1 \leq i < j \leq n} \Pr[A_i \cap A_j]$$

Random subsum principle. The following fact is used often in the book:

CLAIM A.5 (THE RANDOM SUBSUM PRINCIPLE)
For $x, y \in \{0,1\}^n$, denote $x \odot y = \sum_{i=1}^n x_i y_i \pmod 2$ (that is, $x \odot y$ is equal to 1 if the number of i's such that $x_i = y_i = 1$ is odd and equal to 0 otherwise). Then for every $y \neq 0^n$,

$$\Pr_{x \in_R \{0,1\}^n}[x \odot y = 1] = \tfrac{1}{2}$$

PROOF: Suppose that y_j is nonzero. We can think of choosing x as follows: first choose all the coordinates of x other than the jth and only choose the jth coordinate last. After we choose all the coordinates of x other than the jth, the value $\sum_{i:i \neq j} x_i y_i \pmod 2$ is fixed to be some $c \in \{0,1\}$. Regardless of what c is, with probability $1/2$ we choose $x_j = 0$, in which case $x \odot y = c$ and with probability $1/2$ we choose $x_j = 1$, in which case $x \odot y = 1 - c$. We see that in any case $x \prod y$ will be equal to 1 with probability $1/2$. ∎

A.3.1 Random variables and expectations.

A *random variable* is a mapping from a probability space to $\mathbb{R}$. For example, if Ω is as above, the set of all possible outcomes of n tosses of a fair coin, then we can denote by X the number of coins that came up heads.

The *expectation* of a random variable X, denoted by $\mathsf{E}[X]$, is its weighted average. That is, $\mathsf{E}[X] = \sum_{i=1}^N p_i X(\omega_i)$. The following simple claim follows from the definition:

CLAIM A.6 (LINEARITY OF EXPECTATION)
For X, Y random variables over a space Ω, denote by $X + Y$ the random variable that maps ω to $X(\omega) + Y(\omega)$. Then,
$$\mathsf{E}[X + Y] = \mathsf{E}[X] + \mathsf{E}[Y]$$

This claims implies that the random variable X from the example above has expectation $n/2$. Indeed $X = \sum_{i=1}^n X_i$ where X_i is equal to 1 if the ith coins came up heads and is equal to 0 otherwise. But clearly, $\mathsf{E}[X_i] = 1/2$ for every i.

For a real number α and a random variable X, we define αX to be the random variable mapping ω to $\alpha \cdot X(\omega)$. Note that $\mathsf{E}[\alpha X] = \alpha \mathsf{E}[X]$.

A.3.2   The averaging argument

We list various versions of the "averaging argument." Sometimes we give two versions of the same result, one as a fact about numbers and one as a fact about probability spaces.

LEMMA A.7
If $a_1, a_2, \ldots, a_n$ are some numbers whose average is c then some $a_i \geq c$.

LEMMA A.8 ("THE PROBABILISTIC METHOD")
If X is a random variable which takes values from a finite set and $\mathsf{E}[X] = \mu$ then the event "$X \geq \mu$" has nonzero probability.

LEMMA A.9
If $a_1, a_2, \ldots, a_n \geq 0$ are numbers whose average is c then the fraction of a_i's that are greater than (resp., at least) kc is less than (resp, at most) $1/k$.

LEMMA A.10 ("MARKOV'S INEQUALITY")
Any non-negative *random variable X satisfies*

$$\mathbf{Pr}\left(X \geq k\mathsf{E}[X]\right) \leq \frac{1}{k}.$$

COROLLARY A.11
If $a_1, a_2, \ldots, a_n \in [0, 1]$ are numbers whose average is $1 - \gamma$ then at least $1 - \sqrt{\gamma}$ fraction of them are at least $1 - \sqrt{\gamma}$.

Can we give any meaningful upperbound on $\mathbf{Pr}[X < c \cdot \mathsf{E}[X]]$ where $c < 1$? Yes, if X is bounded.

LEMMA A.12
If $a_1, a_2, \ldots, a_n$ are numbers in the interval $[0, 1]$ whose average is ρ then at least $\rho/2$ of the a_i's are at least as large as $\rho/2$.

PROOF: Let γ be the fraction of i's such that $a_i \geq \rho/2$. Then $\gamma + (1 - \gamma)\rho/2$ must be at least $\rho/2$, so $\gamma \geq \rho/2$. ∎ More generally, we have

LEMMA A.13
If $X \in [0, 1]$ and $\mathsf{E}[X] = \mu$ then for any $c < 1$ we have

$$\mathbf{Pr}[X \leq c\mu] \leq \frac{1 - \mu}{1 - c\mu}.$$

EXAMPLE A.14
Suppose you took a lot of exams, each scored from 1 to 100. If your average score was 90 then in at least half the exams you scored at least 80.

A.3.3   Conditional probability and independence

If we already know that an event B happened, this reduces the space from Ω to $\Omega \cap B$, where we need to scale the probabilities by $1/\Pr[B]$ so they will sum up to one. Thus, the probability of an event A *conditioned* on an event B, denoted $\Pr[A|B]$, is equal to $\Pr[A \cap B]/\Pr[B]$ (where we always assume that B has positive probability).

We say that two events A, B are *independent* if $\Pr[A \cap B] = \Pr[A]\Pr[B]$. Note that this implies that $\Pr[A|B] = \Pr[A]$ and $\Pr[B|A] = \Pr[B]$. We say that a set of events $A_1, \ldots, A_n$ are *mutually independent* if for every subset $S \subset [n]$,

$$\Pr[\cap_{i \in S} A_i] = \prod_{i \in S} \Pr[A_i]. \tag{2}$$

We say that $A_1, \ldots, A_n$ are *k-wise independent* if (2) holds for every $S \subseteq [n]$ with $|S| \leq k$.

We say that two random variables X, Y are *independent* if for every $x, y \in \mathbb{R}$, the events $\{X = x\}$ and $\{Y = y\}$ are independent. We generalize similarly the definition of mutual independence and k-wise independence to sets of random variables $X_1, \ldots, X_n$. We have the following claim:

CLAIM A.15
If $X_1, \ldots, X_n$ are mutually independent then

$$\mathsf{E}[X_1 \cdots X_n] = \prod_{i=1}^{n} \mathsf{E}[X_i]$$

PROOF:

$$\mathsf{E}[X_1 \cdots X_n] = \sum_{x} x \Pr[X_1 \cdots X_n = x] =$$

$$\sum_{x_1, \ldots, x_n} x_1 \cdots x_n \Pr[X_1 = x_1 \text{ and } X_2 = x_2 \cdots \text{ and } X_n = x_n] = \text{(by independence)}$$

$$\sum_{x_1, \ldots, x_n} x_1 \cdots x_n \Pr[X_1 = x_1] \cdots \Pr[X_n = x_n] =$$

$$(\sum_{x_1} x_1 \Pr[X_1 = x_1])(\sum_{x_2} x_2 \Pr[X_2 = x_2]) \cdots (\sum_{x_n} x_n \Pr[X_n = x_n]) = \prod_{i=1}^{n} \mathsf{E}[X_i]$$

where the sums above are over all the possible real numbers that can be obtained by applying the random variables or their products to the finite set Ω. ∎

A.3.4   Deviation upperbounds

Under various conditions, one can give upperbounds on the probability of a random variable "straying too far" from its expectation. These upperbounds are usually derived by clever use of Markov's inequality.

The *variance* of a random variable X is defined to be $\mathsf{Var}[X] = \mathsf{E}[(X - \mathsf{E}(X))^2]$. Note that since it is the expectation of a non-negative random variable, $\mathsf{Var}[X]$ is always non-negative. Also, using

linearity of expectation, we can derive that $\mathsf{Var}[X] = \mathsf{E}[X^2] - (\mathsf{E}[X])^2$. The *standard deviation* of a variable X is defined to be $\sqrt{\mathsf{Var}[X]}$.

The first bound is Chebyshev's inequality, useful when only the variance is known.

LEMMA A.16 (CHEBYSHEV INEQUALITY)
If X is a random variable with standard deviation σ, then for every $k > 0$,

$$\Pr[|X - \mathsf{E}[X]| > k\sigma] \leq 1/k^2$$

PROOF: Apply Markov's inequality to the random variable $(X - \mathsf{E}[X])^2$, noting that by definition of variance, $E[(X - \mathsf{E}[X])^2] = \sigma^2$. ∎

Chebyshev's inequality is often useful in the case that X is equal to $\sum_{i=1}^{n} X_i$ for pairwise independent random variables $X_1, \ldots, X_n$. This is because of the following claim, that is left as an exercise:

CLAIM A.17
If $X_1, \ldots, X_n$ are pairwise independent then

$$\mathsf{Var}(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} \mathsf{Var}(X_i)$$

The next inequality has many names, and is widely known in theoretical computer science as the *Chernoff bound*. It considers scenarios of the following type. Suppose we toss a fair coin n times. The expected number of heads is $n/2$. How tightly is this number concentrated? Should we be very surprised if after 1000 tosses we have 625 heads? The bound we present is slightly more general, since it concerns n different coin tosses of possibly different expectations (the expectation of a coin is the probability of obtaining "heads"; for a fair coin this is $1/2$). These are sometimes known as Poisson trials.

THEOREM A.18 ("CHERNOFF" BOUNDS)
Let $X_1, X_2, \ldots, X_n$ be mutually independent random variables over $\{0, 1\}$ (i.e., X_i can be either 0 or 1) and let $\mu = \sum_{i=1}^{n} \mathsf{E}[X_i]$. Then for every $\delta > 0$,

$$\mathbf{Pr}[\sum_{i=1}^{n} X_i \geq (1 + \delta)\mu] \leq \left[\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right]^\mu. \tag{3}$$

$$\mathbf{Pr}[\sum_{i=1}^{n} \leq (1 - \delta)\mu] \leq \left[\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right]^\mu. \tag{4}$$

Often, what we use need is only the corollary that under the above conditions, for every $c > 0$

$$\Pr\left[\left|\sum_{i=1}^{n} X_i - \mu\right| \geq c\mu\right] \leq 2^{-c^2 n/2}$$

PROOF: Surprisingly, the Chernoff bound is also proved using the Markov inequality. We only prove the first inequality; a similar proof exists for the second. We introduce a positive dummy variable t, and observe that

$$\mathsf{E}[\exp(tX)] = \mathsf{E}[\exp(t\sum_i X_i)] = \mathsf{E}[\prod_i \exp(tX_i)] = \prod_i \mathsf{E}[\exp(tX_i)], \tag{5}$$

where $\exp(z)$ denotes e^z and the last equality holds because the X_i r.v.s are independent. Now,

$$\mathsf{E}[\exp(tX_i)] = (1 - p_i) + p_i e^t,$$

therefore,

$$\prod_i \mathsf{E}[\exp(tX_i)] = \prod_i [1 + p_i(e^t - 1)] \le \prod_i \exp(p_i(e^t - 1))$$
$$= \exp(\sum_i p_i(e^t - 1)) = \exp(\mu(e^t - 1)), \tag{6}$$

as $1 + x \le e^x$. Finally, apply Markov's inequality to the random variable $\exp(tX)$, viz.

$$\mathbf{Pr}[X \ge (1 + \delta)\mu] = \mathbf{Pr}[\exp(tX) \ge \exp(t(1 + \delta)\mu)] \le \frac{\mathsf{E}[\exp(tX)]}{\exp(t(1 + \delta)\mu)} = \frac{\exp((e^t - 1)\mu)}{\exp(t(1 + \delta)\mu)},$$

using lines (5) and (6) and the fact that t is positive. Since t is a dummy variable, we can choose any positive value we like for it. Simple calculus shows that the right hand side is minimized for $t = \ln(1 + \delta)$ and this leads to the theorem statement. ∎

By the way, if all n coin tosses are fair (Heads has probability $1/2$) then the the probability of seeing N heads where $|N - n/2| > a\sqrt{n}$ is at most $e^{-a^2/2}$. The chance of seeing at least 625 heads in 1000 tosses of an unbiased coin is less than 5.3×10^{-7}.

A.3.5   Some other inequalities.

Jensen's inequality.

The following inequality, generalizing the inequality $\mathsf{E}[X^2] \ge \mathsf{E}[X]^2$, is also often useful:

CLAIM A.19
We say that $f : \mathbb{R} \to \mathbb{R}$ is convex if for every $p \in [0,1]$ and $x, y \in \mathbb{R}$, $f(px + (1 - p)y) \le p \cdot f(x) + (1 - p) \cdot f(y)$. Then, for every random variable X and convex function f, $f(\mathsf{E}[X]) \le \mathsf{E}[f(X)]$.

Approximating the binomial coefficient

Of special interest is the *Binomial* random variable B_n denoting the number of coins that come up "heads" when tossing n fair coins. For every k, $\Pr[B_n = k] = 2^{-n}\binom{n}{k}$ where $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ denotes the number of size-k subsets of $[n]$. Clearly, $\binom{n}{k} \le n^k$, but sometimes we will need a better estimate for $\binom{n}{k}$ and use the following approximation:

CLAIM A.20
For every n, $k < n$,

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{ne}{k}\right)^k$$

The best approximation can be obtained via Stirling's formula:

LEMMA A.21 (STIRLING'S FORMULA)
For every n,

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

It can be proven by taking natural logarithms and approximating $\ln n! = \ln(1 \cdot 2 \cdots n) = \sum_{i=1}^{n} \ln i$ by the integral $\int_1^n \ln x\,dx = n \ln n - n + 1$. It implies the following corollary:

COROLLARY A.22
For every $n \in \mathbb{N}$ and $\alpha \in [0,1]$,

$$\binom{n}{\alpha n} = (1 \pm O(n^{-1})) \frac{1}{\sqrt{2\pi n \alpha(1-\alpha)}} 2^{H(\alpha)n}$$

where $H(\alpha) = \alpha \log(1/\alpha) + (1-\alpha)\log(1/(1-\alpha))$ and the constants hidden in the O notation are independent of both n and α.

More useful estimates.

The following inequalities can be obtained via elementary calculus:

- For every $x \geq 1$, $\left(1 - \frac{1}{x}\right)^x \leq \frac{1}{e} \leq \left(1 - \frac{1}{x+1}\right)^x$

- For every k, $\sum_{i=1}^{n} i^k = \Theta\left(\frac{n^{k+1}}{k+1}\right)$

- For every $k > 1$, $\sum_{i=1}^{\infty} n^{-k} < O(1)$.

- For every $c, \epsilon > 0$, $\sum_{i=1}^{\infty} \frac{n^c}{(1+\epsilon)^n} < O(1)$.

- For every n, $\sum_{i=1}^{n} = \ln n \pm O(1)$

A.4 Finite fields and groups

A *field* is a set $\mathbb{F}$ that has an addition $(+)$ and multiplication $(\cdot)$ operations that behave in the expected way: satisfy associative, commutative and distributive laws, have both additive and multiplicative inverses, and neutral elements 0 and 1 for addition and multiplication respectively. Familiar fields are the real numbers $(\mathbb{R})$, the rational numbers $(\mathbb{Q})$ and the complex numbers $(\mathbb{C})$, but there are also *finite* fields.

If q is a prime, then we denote by $\mathrm{GF}(q)$ the field consisting of the elements $\{0, \ldots, q-1\}$ with addition and multiplication performed modulo q. For example, the numbers $\{0, \ldots, 6\}$ yield a field

if addition and multiplication are performed modulo 7. We leave it to the reader to verify $GF(q)$ is indeed a field for every prime q. The simplest example for such a field is the field $GF(2)$ consisting of $\{0, 1\}$ where multiplication is the AND ($\wedge$) operation and addition is the XOR operation.

Every finite field $\mathbb{F}$ has a number ℓ such that for every $x \in F$, $x + x + \cdots + x$ (ℓ times) is equal to the zero element of $\mathbb{F}$ (exercise). This number ℓ is called the *characteristic* of $\mathbb{F}$. For every prime q, the characteristic of $GF(q)$ is equal to q.

A.4.1   Non-prime fields.

One can see that if n is not prime, then the set $\{0, \ldots, n-1\}$ with addition and multiplication modulo n is not a field, as there exist two non-zero elements x, y in this set such that $x \cdot y = n = 0$ (mod n). Nevertheless, there are finite fields of size n for non-prime n. Specifically, for every prime q, and $k \geq 1$, there exists a field of q^k elements, which we denote by $GF(q^k)$. We will very rarely need to use such fields in this book, but still provide an outline of their construction below.

For every prime q and k there exists an *irreducible* degree k polynomial P over the field $GF(q)$ (P is irreducible if it cannot be expressed as the product of two polynomials P', P'' of lower degree). We then let $GF(q^k)$ be the set of all $k-1$-degree polynomials over $GF(q)$. Each such polynomial can be represented as a vector of its k coefficients. We perform both addition and multiplication modulo the polynomial P. Note that addition corresponds to standard vector addition of k-dimensional vectors over $GF(q)$, and both addition and multiplication can be easily done in $\text{poly}(n, \log q)$ time (we can reduce a polynomial S modulo a polynomial P using a similar algorithm to long division of numbers). It turns out that no matter how we choose the irreducible polynomial P, we will get the same field, up to renaming of the elements. There is a deterministic $\text{poly}(q, k)$-time algorithm to obtain an irreducible polynomial of degree k over $GF(q)$. There are also probabilistic algorithms (and deterministic algorithms whose analysis relies on unproven assumptions) that obtain such a polynomial in $\text{poly}(\log q, k)$ time.

For us, the most important example of a finite field is $GF(2^k)$, which consists of the set $\{0, 1\}^k$, with addition being component-wise XOR, and multiplication being polynomial multiplication via some irreducible polynomial which we can fine in $\text{poly}(k)$ time. In fact, we will mostly not even be interested in the multiplicative structure of $GF(2^k)$ and only use the addition operation (i.e., use it as the vector space $GF(2)^k$, see below).

A.4.2   Groups.

A *group* is a set that only has a single operation, say $\star$, that is associative and has an inverse. That is, $(G, \star)$ is a group if

1. For every $a, b, c \in G$, $(a \star b) \star c = a \star (b \star c)$

2. There exists a special element $id \in G$ such that $a \star id = a$ for every $a \in G$, and for every $a \in G$ there exists $b \in G$ such that $a \star b = b \star a = id$.

If G is a finite group, it is known that for every $a \in G$, $a \star a \star \cdots \star a$ ($|G|$ times) is equal to the element id. A group is called *commutative* or Abelian if its operation satisfies $a \star b = b \star a$ for every $a, n \in G$. For every number $n \geq 2$, the set $\{0, \ldots, n-1\}$ with the operation being addition

modulo n is an Abelian group. Also, the set $\{k : k \in [n-1], gcd(k,n) = 1\}$ with the operation being multiplication modulo n is an Abelian group.

   If $\mathbb{F}$ is a field and $k \geq 1$, then the set of k-dimensional vectors of $\mathbb{F}$ (i.e., $\mathbb{F}^k$) together with the operation of componentwise addition, yields an Abelian group. As mentioned above, the most interesting special case for us is the group $\mathrm{GF}(2)^k$ for some k. Note that in this group the identity element is the vector 0^k and for every $x \in \mathrm{GF}(2)^k$, $x + x = 0^k$. This group is often referred to as the *Boolean cube*.

A.5   Vector spaces and Hilbert spaces

A.6   Polynomials

We list some basic facts about univariate polynomials.

THEOREM A.23
A nonzero polynomial of degree d has at most d distinct roots.

PROOF: Suppose $p(x) = \sum_{i=0}^{d} c_i x^i$ has $d+1$ distinct roots $\alpha_1, \ldots, \alpha_{d+1}$ in some field $\mathbb{F}$. Then

$$\sum_{i=0}^{d} \alpha_j^i \cdot c_i = p(\alpha_j) = 0,$$

for $j = 1, \ldots, d+1$. This means that the system $\mathbf{Ay} = \mathbf{0}$ with

$$\mathbf{A} = \begin{pmatrix} 1 & \alpha_1 & \alpha_1^2 & \ldots & \alpha_1^d \\ 1 & \alpha_2 & \alpha_2^2 & \ldots & \alpha_2^d \\ \cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot \\ 1 & \alpha_{d+1} & \alpha_{d+1}^2 & \ldots & \alpha_{d+1}^d \end{pmatrix}$$

has a solution $\mathbf{y} = \mathbf{c}$. The matrix $\mathbf{A}$ is a *Vandermonde* matrix, and it can be shown that

$$\det \mathbf{A} = \prod_{i>j}(\alpha_i - \alpha_j),$$

which is nonzero for distinct α_i. Hence rank$\mathbf{A} = d+1$. The system $\mathbf{Ay} = \mathbf{0}$ has therefore only a trivial solution — a contradiction to $\mathbf{c} \neq \mathbf{0}$. $\blacksquare$

THEOREM A.24
For any set of pairs $(a_1, b_1), \ldots, (a_{d+1}, b_{d+1})$ there exists a unique polynomial $g(x)$ of degree at most d such that $g(a_i) = b_i$ for all $i = 1, 2, \ldots, d+1$.

PROOF: The requirements are satisfied by *Lagrange Interpolating Polynomial*:

$$\sum_{i=1}^{d+1} b_i \cdot \frac{\prod_{j\neq i}(x - a_j)}{\prod_{j\neq i}(a_i - a_j)}.$$

If two polynomials $g_1(x), g_2(x)$ satisfy the requirements then their difference $p(x) = g_1(x) - g_2(x)$ is of degree at most d, and is zero for $x = a_1, \ldots, a_{d+1}$. Thus, from the previous theorem, polynomial $p(x)$ must be zero and polynomials $g_1(x), g_2(x)$ identical. ∎

The following elementary result is usually attributed to Schwartz and Zippel in the computer science community, though it was certainly known earlier (see e.g. DeMillo and Lipton [**?**]).

LEMMA A.25
If a polynomial $p(x_1, x_2, \ldots, x_m)$ over $F = GF(q)$ is nonzero and has total degree at most d, then

$$\Pr[p(a_1..a_m) \neq 0] \geq 1 - \frac{d}{q},$$

where the probability is over all choices of $a_1..a_m \in F$.

PROOF: We use induction on m. If $m = 1$ the statement follows from Theorem A.23. Suppose the statement is true when the number of variables is at most $m - 1$. Then p can be written as

$$p(x_1, x_2, \ldots, x_m) = \sum_{i=0}^{d} x_1^i p_i(x_2, \ldots, x_m),$$

where p_i has total degree at most $d - i$. Since p is nonzero, at least one of p_i is nonzero. Let k be the largest i such that p_i is nonzero. Then by the inductive hypothesis,

$$\Pr_{a_2, a_3, \ldots, a_m} [p_i(a_2, a_3, \ldots, a_m) \neq 0] \geq 1 - \frac{d - k}{q}.$$

Whenever $p_i(a_2, a_3, \ldots, a_m) \neq 0$, $p(x_1, a_2, a_3, \ldots, a_m)$ is a nonzero univariate polynomial of degree k, and hence becomes 0 only for at most k values of x_1. Hence

$$\Pr[p(a_1..a_m) \neq 0] \geq (1 - \frac{k}{q})(1 - \frac{d - k}{q}) \geq 1 - \frac{d}{q},$$

and the induction is completed. ∎

Bibliography

[AA02]   M.Vyalyi A.Kitaev and A.Shen. *Classical and Quantum computation.* AMS Press, 2002.

[Aar05]   Aaronson. NP-complete problems and physical reality. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 36, 2005.

[AB87]   Noga Alon and Ravi B. Boppana. The monotone circuit complexity of boolean functions. *Combinatorica*, 7(1):1–22, 1987.

[AD97]   M. Ajtai and C. Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing (STOC '97)*, pages 284–293, New York, 1997. Association for Computing Machinery.

[Adl78]   Leonard Adleman. Two theorems on random polynomial time. In *19th Annual Symposium on Foundations of Computer Science*, pages 75–83, Ann Arbor, Michigan, 1978. IEEE.

[AG94]   Eric Allender and Vivek Gore. A uniform circuit lower bound for the permanent. *SIAM Journal on Computing*, 23(5):1026–1049, October 1994.

[Ajt83]   M. Ajtai. Σ_1^1-formulae on finite structures. *Annals of Pure and Applied Logic*, 24:1–48, 1983.

[Ajt96]   M. Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 99–108, Philadelphia, Pennsylvania, 1996.

[AKL+79]   Romas Aleliunas, Richard M. Karp, Lipton Lipton, Laszlo Lovász, and Charles Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *20th Annual Symposium on Foundations of Computer Science*, pages 218–223, San Juan, Puerto Rico, 29–31 October 1979. IEEE.

[AKS87]   M. Ajtai, J. Komlos, and E. Szemeredi. Deterministic simulation in LOGSPACE. In ACM, editor, *Proceedings of the nineteenth annual ACM Symposium on Theory of Computing, New York City, May 25–27, 1987*, pages 132–140, New York, NY, USA, 1987. ACM Press.

[ALM+98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, May 1998. Prelim version IEEE FOCS 1992.

[Aro98]   Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *JACM: Journal of the ACM*, 45, 1998.

[AS98]   Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of NP. *Journal of the ACM*, 45(1):70–122, January 1998. Prelim version IEEE FOCS 1982.

[AS00]   Noga Alon and Joel Spencer. *The Probabilistic Method*. John Wiley, 2000.

[AUY83]   A.V. Aho, J.D. Ullman, and M. Yannakakis. On notions of information transfer in VLSI circuits. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 133–139, 25–27 April 1983.

[Bab90]   László Babai. E-mail and the unexpected power of interaction. In *Proceedings, Fifth Annual Structure in Complexity Theory Conference*, pages 30–44, Barcelona, Spain, 8–11 July 1990. IEEE Computer Society Press.

[Bar02]   Boaz Barak. A probabilistic-time hierarchy theorem for "Slightly Non-uniform" algorithms. *Lecture Notes in Computer Science*, 2483:194–??, 2002.

[BBR94]   David A. Mix Barrington, Richard Beigel, and Rudich Rudich. Representing Boolean functions as polynomials modulo composite numbers. *Computational Complexity*, 4(4):367–382, 1994. Prelim. version in ACM STOC 1992.

[BdW02]   H. Buhrman and R. de Wolf. Complexity measures and decision tree complexity: A survey. *Theoretical Computer Science*, 288:21–43, 2002.

[Bec91]   J. Beck. An algorithmic approach to the lovàsz local lemma. *Random Structures and Algorithms*, 2(4):367–378, 1991.

[Bel64]   John S. Bell. On the Einstein-Podolsky-Rosen paradox. *Physics*, 1(3):195–290, 1964.

[BF90]   Donald Beaver and Joan Feigenbaum. Hiding instances in multioracle queries. In *7th Annual Symposium on Theoretical Aspects of Computer Science*, volume 415 of *lncs*, pages 37–48, Rouen, France, 22–24 February 1990. Springer.

[BFL91]   László Babai, Lance Fortnow, and Lund Lund. Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1:3–40, 1991.

[BFNW93] László Babai, Lance Fortnow, Noam Nisan, and Avi Wigderson. BPP has subexponential time simulations unless EXPTIME has publishable proofs. *Computational Complexity*, 3(4):307–318, 1993.

[BFT98]   H. Buhrman, L. Fortnow, and T. Thierauf. Nonrelativizing separations. In *Proceedings of the 13th Annual IEEE Conference on Computational Complexity (CCC-98)*, pages 8–12, Los Alamitos, June 15–18 1998. IEEE Computer Society.

[BGS75]   Theodore Baker, John Gill, and Robert Solovay. Relativizations of the $\mathcal{P} =?\ \mathcal{NP}$ question. *SIAM Journal on Computing*, 4(4):431–442, December 1975.

[BK95]   M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, January 1995.

[Blu67]   M. Blum. A machine-independent theory of the complexity of recursive functions. *JACM: Journal of the ACM*, 14, 1967.

[Blu82]   M. Blum. Coin flipping by telephone. In *Proc. 1982 IEEE COMPCON, High Technology in the Information Age*, pages 133–137, 1982. See also SIGACT News, Vol 15, No. 1, 1983.

[Blu84]   M. Blum. Independent unbiased coin flips from a correlated biased source: a finite state Markov chain. In *25th Annual Symposium on Foundations of Computer Science*, pages 425–433. IEEE, 24–26 October 1984.

[BM84]   M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13(4):850–864, Nov 1984.

[BM88]   László Babai and Shlomo Moran. Arthur-Merlin games: A randomized proof system, and a hierarchy of complexity classes. *Journal of Computer and System Sciences*, 36(2):254–276, April 1988.

[BNS]   L. Babai, N. Nisan, and M. Szegedy. Multiparty protocols, pseudorandom generators for logspace, and time-space trade-offs.

[Bru04]   Colin Bruce. *Schrodinger's Rabbits: Entering The Many Worlds Of Quantum*. Joseph Henry Press, 2004.

[BS90]   R.B. Boppana and M. Sipser. The complexity of finite functions. In *Handbook of Theoretical Computer Science, Ed. Jan van Leeuwen, Elsevier and MIT Press (Volume A (= "1"): Algorithms and Complexity)*, volume 1. 1990.

[BS96]   Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory — Efficient Algorithms*, volume I. MIT Press, Cambridge, USA, 1996.

[BT94]   Richard Beigel and Jun Tarui. On ACC. *Computational Complexity*, 4(4):350–366, 1994. Prelim version in IEEE FOCS 1991.

[BV97]   E. Bernstein and U. Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, October 1997. Prelim version in STOC 1993.

[Chu36]   Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *Amer. J. Math.*, 58(2):345–363, 1936.

DRAFT

[CK00]   Pierluigi Crescenzi and Viggo Kann. A compendium of np optimization problems. http://www.nada.kth.se/~viggo/problemlist/, 2000. Website tracking the tractability of many NP problems.

[CLRS01]   T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2001.

[Cob64]   A. Cobham. The intrinsic computational difficulty of functions. In Yehoshua Bar-Hillel, editor, *Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science*, pages 24–30. Elsevier/North-Holland, 1964.

[Coo71]   S. A. Cook. The complexity of theorem proving procedures. In *Proc. 3rd Ann. ACM Symp. Theory of Computing*, pages 151–158, NY, 1971. ACM.

[Coo73]   Stephen A. Cook. A hierarchy for nondeterministic time complexity. *Journal of Computer and System Sciences*, 7(4):343–353, August 1973.

[CW89]   Aviad Cohen and Avi Wigderson. Dispersers, deterministic amplification, and weak random sources (extended abstract). In *30th Annual Symposium on Foundations of Computer Science*, pages 14–19, 30 October–1 November 1989.

[CW90]   Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, March 1990.

[Dav65]   Martin Davis. *The Undecidable*. Dover Publications, 1965.

[Deu85]   D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London Ser. A*, A400:97–117, 1985.

[DH76]   W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(5):644–654, November 1976.

[DJ92]   D. Deutsch and R. Jozsa. Rapid solution of problems by quantum computation. *Proc Roy Soc Lond A*, 439:553–558, October 1992.

[DK00]   D.Z. Du and K. I. Ko. *Theory of Computational Complexity*. Wiley, 2000.

[DPV06]   S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani. *Algorithms*. McGraw Hill, 2006. Draft available from the authors' webpage.

[Edm65]   J. Edmonds. Paths, trees, and flowers. *Canad. J. Math*, 17:449–467, 1965.

[ER60]   P. Erdős and R. Rado. Intersection theorems for systems of sets. *J. London Math. Soc.*, 35:85–90, 1960.

[Fey82]   R. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6&7):467–488, 1982.

DRAFT

[FSS84]   M. Furst, J. Saxe, and M. Sipser. Parity, circuits, and the polynomial time hierarchy. *Mathematical Systems Theory*, 17:13–27, 1984. Prelim version in IEEE FOCS 1981.

[GG00]   O. Goldreich and S. Goldwasser. On the limits of nonapproximability of lattice problems. *JCSS: Journal of Computer and System Sciences*, 60, 2000.

[GGM86]   O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986. Prelim. version in IEEE FOCS 1984.

[Gil74]   John T. Gill. Computational complexity of probabilistic Turing machines. In ACM, editor, *Conference record of sixth annual ACM Symposium on Theory of Computing: papers presented at the symposium, Seattle, Washington, April 30–May 2, 1974*, pages 91–95, New York, NY, USA, 1974. ACM Press.

[Gil77]   John Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6(4):675–695, December 1977.

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

[GM84]   Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, April 1984.

[GMR89]   S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, February 1989. Prelim version in ACM STOC 1985.

[GMW87]   O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game — A completeness theorem for protocols with honest majority. In ACM, editor, *Proceedings of the nineteenth annual ACM Symposium on Theory of Computing, New York City, May 25–27, 1987*, pages 218–229, New York, 1987. ACM Press.

[Gol04]   Oded Goldreich. *Foundations of Cryptography, Volumes 1 and 2*. Cambridge University Press, 2001,2004.

[GS87]   S. Goldwasser and M. Sipser. Private coins versus public coins in interactive proof systems. In S. Micali, editor, *Randomness and Computation*. JAI Press, Greenwich, CT, 1987. Extended Abstract in *Proc. 18th ACM Symp. on Theory of Computing*, 1986.

[Has86]   Johan Hastad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 6–20, Berkeley, California, 28–30 May 1986.

[HILL99] Johan Håstad, Russell Impagliazzo, Levin Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, August 1999.

DRAFT

[HMU01]   J. E. Hopcroft, R. Motwani, and J. D. Ullman. *"Introduction to Automata Theory, Language, and Computation"*. Addison–Wesley, 2nd edition edition, 2001.

[Hod83]   Andrew Hodges. *Alan Turing: the enigma.* Burnett Books, London, 1983.

[HS65]   J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[HS66]   F. C. Hennie and R. E. Stearns. Two-tape simulation of multitape Turing machines. *Journal of the ACM*, 13(4):533–546, October 1966.

[HVV04]   Alexander Healy, Salil Vadhan, and Emanuele Viola. Using nondeterminism to amplify hardness. In *Proceedings of the thirty-sixth annual ACM Symposium on Theory of Computing (STOC-04)*, pages 192–201, New York, June 13–15 2004. ACM Press.

[ILL89]   Russell Impagliazzo, Leonid A. Levin, and Luby Luby. Pseudo-random generation from one-way functions. In *Proceedings of the 21st Annual Symposium on Theory of Computing (STOC '89)*, pages 12–24, New York, May 1989. ACM Association for Computing Machinery.

[Imp95]   Russell Impagliazzo. A personal view of average-case complexity. In *Structure in Complexity Theory Conference*, pages 134–147, 1995.

[INW94]   Russell Impagliazzo, Noam Nisan, and Avi Wigderson. Pseudorandomness for network algorithms. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on the Theory of Computing*, pages 356–364, 23–25 May 1994.

[IW01]   Russell Impagliazzo and Avi Wigderson. Randomness vs time: Derandomization under a uniform assumption. *JCSS: Journal of Computer and System Sciences*, 63, 2001. Prelim. version in IEEE FOCS 1998.

[Kan82]   R. Kannan. Circuit-size lower bounds and non-reducibility to sparse sets. *Information and Control*, 55(1–3):40–56, 1982. Prelim version in IEEE FOCS 1981.

[Kar72]   R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, New York, 1972.

[KI03]   Valentine Kabanets and Russell Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. In ACM, editor, *Proceedings of the Thirty-Fifth ACM Symposium on Theory of Computing, San Diego, CA, USA, June 9–11, 2003*, pages 355–364, New York, NY, USA, 2003. ACM Press.

[KL82]   R. Karp and R. Lipton. Turing machines that take advice. *L' Ensignement Mathématique*, 28:191–210, 1982.

[KN97]   E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

DRAFT

[Koz97]   Dexter C. Kozen. *Automata and Computability*. Springer-Verlag, Berlin, 1997.

[KRW95]   Mauricio Karchmer, Ran Raz, and Avi Wigderson. Super-logarithmic depth lower
          bounds via the direct sum in communication complexity. *Computational Complexity*,
          5(3/4):191–204, 1995.

[KT06]   Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson Studium, Boston-San
          Francisco-New York-London-Toronto-Sydney-Tokyo-Singapore-Madrid-Mexico City-
          Munich-Paris-Cape Town-Hong Kong-Montreal, 2006.

[KUW86]   R.M. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random
          NC. *COMBINAT: Combinatorica*, 6:35–48, 1986.

[KV94]   M. Kearns and L. Valiant. Cryptographic limitations on learning Boolean formulae and
          finite automata. *J. ACM*, 41(1):67–95, 1994. Prelim. version in 21th STOC, 1989.

[KVVY93] R. Kannan, H. Venkateswaran, V. Vinay, and A. C. Yao. A circuit-based proof of Toda's
          theorem. *Information and Computation*, 104(2):271–276, 1993.

[KW90]   Mauricio Karchmer and Avi Wigderson. Monotone circuits for connectivity require
          super-logarithmic depth. *SIAM Journal on Discrete Mathematics*, 3(2):255–265, May
          1990. Prelim version in ACM STOC 1988.

[Lea05]   David Leavitt. *The man who knew too much: Alan Turing and the invention of the
          computer*. Great discoveries. W. W. Norton & Co., New York, NY, USA, 2005.

[Lei92]   F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays •
          Trees • Hypercubes*. Morgan Kaufmann, 1992.

[Lev]   L. Levin. The tale of one-way functions (english translation). *Problemy Peredachi
          Informatsii*, 39(1).

[Lev73]   L. Levin. Universal sequential search problems. *PINFTRANS: Problems of Information
          Transmission (translated from Problemy Peredachi Informatsii (Russian))*, 9, 1973.

[LFK92]   C. Lund, L. Fortnow, and H. Karloff. Algebraic methods for interactive proof systems.
          Journal of the ACM, 39(4):859–868, October 1992.

[Lip90]   Richard J. Lipton. Efficient checking of computations. In *7th Annual Symposium on
          Theoretical Aspects of Computer Science*, volume 415 of *lncs*, pages 207–215, Rouen,
          France, 22–24 February 1990. Springer.

[Lip91]   R. J. Lipton. New directions in testing. In *Distributed Computing and Cryptogra-
          phy*, volume 2 of *DIMACS Series in Discrete Mathematics and Theoretical Computer
          Science*, pages 191–202. American Mathematics Society, 1991.

[LL06]   Eric Lehman and Tom Leighton. Mathematics for computer science. Technical report,
          2006. Lecture notes.

[LLKS85]   E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem.* John Wiley, New York, 1985.

[LMSS56]   K. de Leeuw, E. F. Moore, C. E. Shannon, and N. Shapiro. Computability by probabilistic machines. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 183–212. 1956.

[Lov78]   L. Lovàsz. Kneser's conjecture, chromatic number, and homotopy. *J. Combin. Theory Ser. A*, 1978.

[Lov79]   L. Lovàsz. On determinants, matchings, and random algorithms. In L. Budach, editor, *Fundamentals of Computation Theory FCT '79*, pages 565–574, Berlin, 1979. Akademie-Verlag.

[LR01]   Oded Lachish and Ran Raz. Explicit lower bound of $4.5n - o(n)$ for Boolean circuits. In ACM, editor, *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing: Hersonissos, Crete, Greece, July 6–8, 2001*, pages 399–408, New York, NY, USA, 2001. ACM Press.

[LRVW03]   Chi-Jen Lu, Omer Reingold, Salil Vadhan, and Wigderson Wigderson. Extractors: optimal up to constant factors. In ACM, editor, *Proceedings of the Thirty-Fifth ACM Symposium on Theory of Computing, San Diego, CA, USA, June 9–11, 2003*, pages 602–611, New York, NY, USA, 2003. ACM Press.

[MR95]   R. Motwani and P. Raghavan. *Randomized algorithms.* Cambridge University Press, 1995.

[MS82]   K. Mehlhorn and E.M. Schmidt. Las Vegas is better than determinism in VLSI and distributed computing (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 330–337, San Francisco, California, 5–7 May 1982.

[MU05]   Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, 2005.

[MVV87]   K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7:105–113, 1987.

[MZ02]   Jiri Matousek and Günter M. Ziegler. Topological lower bounds for the chromatic number: A hierarchy, November 24 2002. Comment: 16 pages, 1 figure. Jahresbericht der DMV, to appear.

[NC00]   M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, 2000.

[Nis92]   N. Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12, 1992. Prelim version in ACM STOC 1990.

[NW94]   Noam Nisan and Avi Wigderson. Hardness vs randomness. *Journal of Computer and System Sciences*, 49(2):149–167, October 1994. Prelim version in IEEE FOCS 1988.

[NZ96]   Noam Nisan and David Zuckerman. Randomness is linear in space. *Journal of Computer and System Sciences*, 52(1):43–52, 1996. Prelim. version in ACM STOC 1993.

[O'D04]   O'Donnell. Hardness amplification within NP. *JCSS: Journal of Computer and System Sciences*, 69, 2004.

[Pap85]   Christos H. Papadimitriou. Games against nature. *J. Comput. System Sci.*, 31(2):288–301, 1985.

[Pap94]   Christos H. Papadimitriou. *Computatational Complexity*. Addison-Wesley, Reading, Mass., 1994.

[PS84]   Christos H. Papadimitriou and Michael Sipser. Communication complexity. *Journal of Computer and System Sciences*, 28(2):260–269, April 1984. Prelim version in ACM STOC 1982.

[PV06]   Christos Papadimitriou and Umesh Vazirani. Lecture notes on discrete mathematics for computer science, 2006. Available from the authors' home pages.

[PY82]   C. H. Papadimitriou and M. Yannakakis. The complexity of facets (and some facets of complexity). *Journal of Computer and System Sciences*, 28(2):244–259, 1982.

[PY91]   C. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43(3):425–440, December 1991. Prelim version STOC 1988.

[Rab80]   M. O. Rabin. A probabilistic algorithm for testing primality. *Journal of Number Theory*, 12, 1980.

[Raz85a]   A lower bound on the monotone network complexity of the logical permanent. *Matematicheskie Zametki*, 37(6):887–900, 1985. In Russian, English translation in Mathematical Notes of the Academy of Sciences of the USSR 37:6 485-493.

[Raz85b]   Alexander A. Razborov. Lowerbounds on the monotone complexity of some boolean functions. *Doklady Akademii Nauk. SSSR*, 281(4):798–801, 1985. Translation in Soviet Math. Doklady 31, 354–357.

[Raz87]   Alexander A. Razborov. Lower bounds on the size of bounded depth circuits over a complete basis with logical addition. *MATHNASUSSR: Mathematical Notes of the Academy of Sciences of the USSR*, 41, 1987.

[Raz89]   A. A. Razborov. On the method of approximations. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 167–176, 15–17 1989.

[Raz92]   A.A. Razborov. On submodular complexity measures. In *Boolean Function Complexity, (M. Paterson, Ed.)*, pages 76–83. London Mathematical Society Lecture Note Series 169, Cambridge University Press, 1992.

[Raz98]   Ran Raz. A parallel repetition theorem. *SIAM Journal on Computing*, 27(3):763–803, June 1998. Prelim version in ACM STOC'95.

[Raz00]   R. Raz. The bns-chung criterion for multi-party communication complexity. *Computational Complexity*, 9(2):113–122, 2000.

[Raz03]   Alexander A. Razborov. Pseudorandom generators hard for k-DNF resolution and polynomial calculus resolution, 2003.

[Raz04]   Ran Raz. Multi-linear formulas for permanent and determinant are of super-polynomial size. In *Proceedings of ACM Symposium on Theory of Computing*. ACM Press, 2004.

[Ros06]   Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, 2006.

[RR97]   Alexander A. Razborov and Steven Rudich. Natural proofs. *Journal of Computer and System Sciences*, 55(1):24–35, August 1997. Prelim version ACM STOC 1994.

[RR99]   Ran Raz and Omer Reingold. On recycling the randomness of states in space bounded computation. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing (STOC'99)*, pages 159–168, New York, 1999. Association for Computing Machinery.

[RS95]   R. Raz and B. Spieker. On the "log rank"-conjecture in communication complexity. *Combinatorica*, 15, 1995. Prelim version in IEEE FOCS 1993.

[RSA78]   R. L. Rivest, A. Shamir, and L. Adelman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[RVW00]   O. Reingold, S. Vadhan, and A. Wigderson. Entropy waves, the zig-zag graph product, and new constant-degree expanders and extractors. In IEEE, editor, *41st Annual Symposium on Foundations of Computer Science: proceedings: 12–14 November, 2000, Redondo Beach, California*, pages 3–13. IEEE Computer Society Press, 2000.

[RW93]   Alexander Razborov and Avi Wigderson. $n^{\Omega(\log n)}$ Lower bounds on the size of depth-3 threshold circuits with AND gates at the bottom. *Information Processing Letters*, 45(6):303–307, 16 April 1993.

[Sav70]   W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *JCSS: Journal of Computer and System Sciences*, 4, 1970.

[Sav72]   J. E. Savage. Computational work and time on finite machines. *Journal of the ACM*, 19(4):660–674, October 1972.

DRAFT

[Sha83]   A. Shamir. On the generation of cryptographically strong pseudorandom sequences. *ACM Trans. on Computer Sys.*, 1(1):38–44, 1983. Prelim version presented at Crypto'81 and ICALP'81.

[Sha92]   Adi Shamir. IP = PSPACE. *Journal of the ACM*, 39(4):869–877, October 1992.

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

[Sip83]   Michael Sipser. A complexity theoretic approach to randomness. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 330–335, Boston, Massachusetts, 25–27 April 1983.

[Sip88]   Michael Sipser. Expanders, randomness, or time versus space. *Journal of Computer and System Sciences*, 36(3):379–383, June 1988. Prelim. version in Proc. IEEE Structure in Complexity '86.

[Sip92]   M. Sipser. The history and status of the P versus NP question. In ACM, editor, *Proceedings of the twenty-fourth annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 4–6, 1992*, pages 603–618, New York, NY, USA, 1992. ACM Press.

[SIP96]   M. SIPSER. *Introduction to the Theory of Computation*. PWS, Boston, MA, 1996.

[Smo87]   R. Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In *Proceedings of the nineteenth annual ACM Symposium on Theory of Computing, New York City, May 25–27, 1987*, pages 77–82, New York, NY, USA, 1987. ACM Press.

[SS77]   R. Solovay and V. Strassen. A fast Monte Carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, 1977.

[Sto77]   L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.

[Str69]   Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[STV]   M. Sudan, L. Trevisan, and S. Vadhan. Pseudorandom generators without the XOR lemma. *JCSS: Journal of Computer and System Sciences*, 62(2).

[SU02a]   M. Schaefer and C. Umans. Completeness in the polynomial-time hierarchy: Part I. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 33, September 2002.

[SU02b]   M. Schaefer and C. Umans. Completeness in the polynomial-time hierarchy: Part II. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 33, December 2002.

[SZ99a]   M. Saks and S. Zhou. $BP_H SPACE(S) \subseteq DSPACE(S^{3/2})$. *JCSS: Journal of Computer and System Sciences*, 58(2):376–403, 1999.

[SZ99b]   Aravind Srinivasan and David Zuckerman. Computing with very weak random sources. *SIAM Journal on Computing*, 28(4):1433–1459, August 1999. Prelim. version FOCS 1994.

[Tar88]   Éva Tardos. The gap between monotone and non-monotone circuit complexity is exponential. *Combinatorica*, 8(1):141–142, 1988.

[Tod91]   S. Toda. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 20(5):865–877, 1991.

[Tra84]   B. A. Trakhtenbrot. A survey of Russian approaches to perebor (brute-force search) algorithms. *Annals of the History of Computing*, 6(4):384–400, October/December 1984. Also contains a good translation of [Lev73].

[Tre01]   Luca Trevisan. Extractors and pseudorandom generators. *Journal of the ACM*, 48(4):860–879, July 2001. Prelim version in ACM STOC 1999.

[Tur36]   A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings, London Mathematical Society,*, pages 230–265, 1936. Published as Proceedings, London Mathematical Society,, volume 2, number 42.

[Uma01]   C. Umans. The minimum equivalent DNF problem and shortest implicants. *JCSS: Journal of Computer and System Sciences*, 63, 2001.

[Vad00]   Salil Vadhan. On transformation of interactive proofs that preserve the prover's complexity. In ACM, editor, *Proceedings of the thirty second annual ACM Symposium on Theory of Computing: Portland, Oregon, May 21–23, [2000]*, pages 200–207, New York, NY, USA, 2000. ACM Press.

[Val75]   Leslie G. Valiant. On non-linear lower bounds in computational complexity. In *STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing*, pages 45–53. ACM Press, 1975.

[Val79a]   L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

[Val79b]   Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[Vaz01]   Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin-Heidelberg-New York-Barcelona-Hong Kong-London-Milan-Paris-Singapur-Tokyo, 2001.

[VG99]   Joachim Von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 1999.

[vN45]   John von Neumann. First draft of a report on the EDVAC. University of pennsylvania report for the u.s. army ordinance department, 1945. Reprinted in part in Brian Randell, ed. The Origins of Digital Computers: Selected Papers, Springer Verlag, 1982.

[von61]   J. von Neumann. *Probabilistic logics and synthesis of reliable organisms from unreliable components*, volume 5. Pergamon Press, 1961.

[Yam97]   P. Yam. Bringing schroedinger's cat back to life. *Scientific American*, pages 124–129, June 1997.

[Yan91]   M. Yannakakis. Expressing combinatorial optimization problems by linear programs. *Journal of Computer and System Sciences*, 43(3):441–466, 1991. Prelim version in ACM STOC 1988.

[Yao79]   A.C.C. Yao. Some complexity questions related to distributive computing(preliminary report). In *STOC '79: Proceedings of the 11th ACM STOC*, pages 209–213. ACM Press, 1979.

[Yao82]   Andrew C. Yao. Theory and applications of trapdoor functions (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 80–91. IEEE, 3–5 November 1982.

[Yao85]   A. C.-C. Yao. Separating the polynomial-time hierarchy by oracles. In *26th Annual Symposium on Foundations of Computer Science (FOCS '85)*, pages 1–10, Los Angeles, Ca., USA, 1985. IEEE Computer Society Press.

[Yao86]   Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE, 27–29 October 1986.

[Yao90]   Andrew Chi-Chih Yao. On ACC and threshold circuits. In *31st Annual Symposium on Foundations of Computer Science*, volume II, pages 619–627. IEEE, 22–24 October 1990.

[Yao93]   A. Yao. Quantum circuit complexity. In *34th Annual Symposium on Foundations of Computer Science*, pages 352–361, Palo Alto, California, 1993. IEEE.

[Zak83]   S. Zak. A Turing machine time hierarchy. *Theoretical Computer Science*, 26(3):327–333, October 1983.

[Zuc90]   David Zuckerman. General weak random sources. In *31st Annual IEEE Symposium on Foundations of Computer Science*, volume II, pages 534–543, 22–24 October 1990.