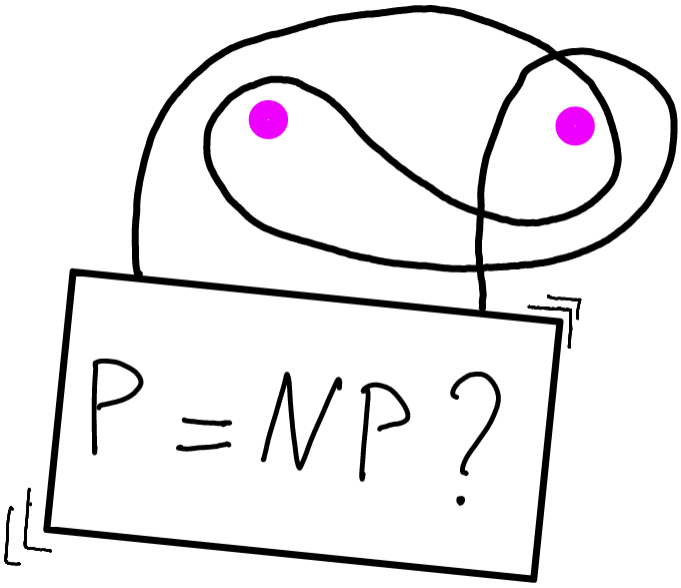
MATHEMATICS OF THE IMPOSSIBLE

THE COMPLEXITY OF COMPUTATION

Working draft compiled on March 16, 2026

Emanuele Viola



Contents

Per ardua ad astra

Chapter 0

Introduction

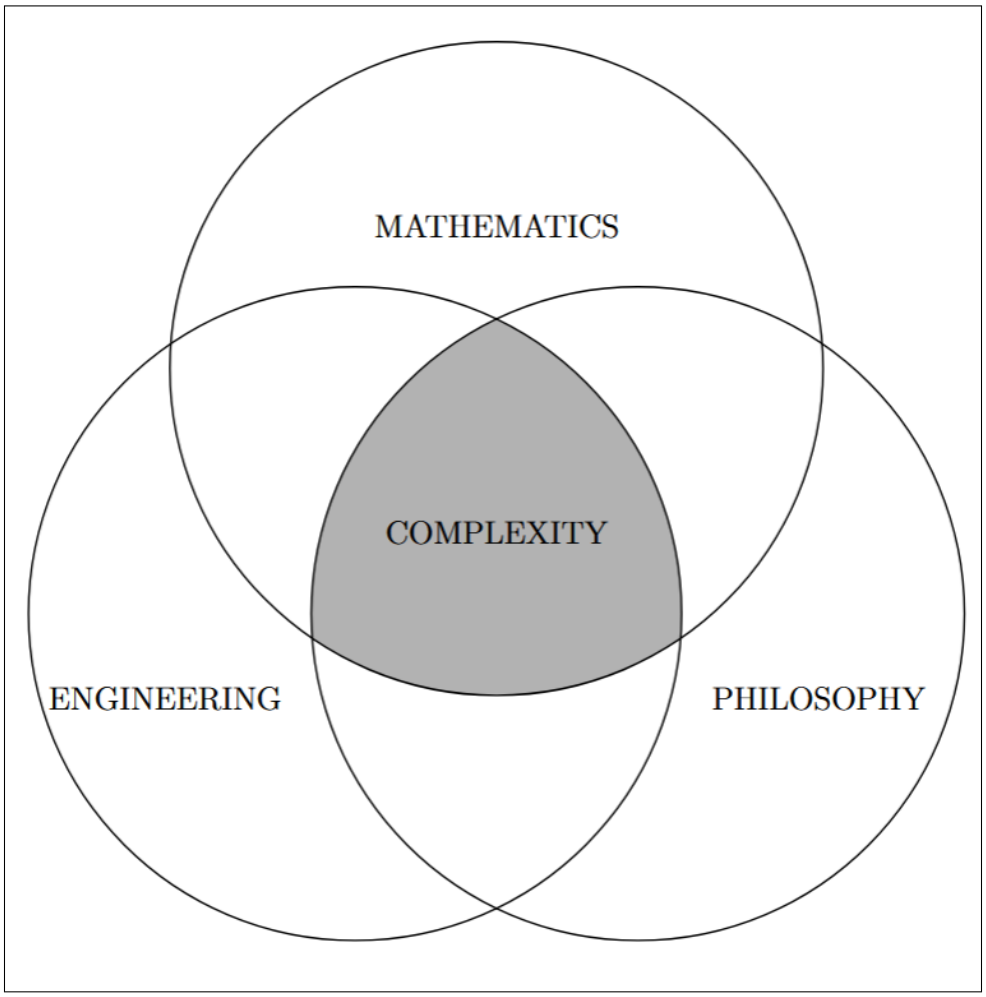


Figure 1: Complexity theory.

Enter *complexity*, the theory of efficient computation. Here "computation" is meant broadly, encompassing any type of information or data processing, even when not tied to physical hardware. Complexity theory lies at the intersection of mathematics, engineering, and philosophy (see figure 1). It is rigorous like mathematics, proceeding through precise claims and proofs, but its main objects of study are computational models such as programs and circuits. Like engineering, it focuses on performing specified tasks under resource constraints. Yet the theory transcends the models, addressing broader, fundamental philosophical questions, including: What can be computed? What is knowledge? What is a proof? What is randomness? To each of these questions, complexity theory provides fascinating answers. The next "teasers" illustrate some of these answers, and more generally aim to introduce the reader to the excitement, beauty, richness, and breadth of the field.

0.1 Teasers

One of the hallmarks of complexity theory is the surprising, often counterintuitive nature of its results. This idea runs throughout the book and is evident in each of the upcoming teasers.

0.1.1 Computing with three bits of memory

Consider a computer with *three* bits of memory. There's also a clock, ticking $1, 2, 3, \ldots$ In one clock cycle the computer can read one bit of the input and update its memory, or stop and return a value. These actions depend only on the clock, the three memory bits, and the length of the input. Let's give a few examples of what such a computer can do. First, it can compute the And function on n bits:

```
Computing And of (x_1,x_2,...,x_n)

For i = 1, 2, ... until n
   Read x_i
   If x_i = 0 return 0
Return 1
```

We didn't really use the memory. Let's consider a slightly more complicated example. A word is a *palindrome* if it reads the same both ways, like *racecar*, *non*, *anna,* and so on. Similarly, examples of palindrome bit strings are $11, 0110$, and so on.

Let's show that the computer can decide if a given string is palindrome quickly, in n steps:

```
Deciding if (x_1, x_2, ..., x_n) is palindrome:

For i = 1, 2, ... until i > n/2
   Read x_i and write it in memory bit m
   If m ≠ x_{n-i} return 0
Return 1
```

That was easy enough. Now consider the Majority function on n bits, which is 1 iff the sum of the input bits is $> n/2$ and 0 otherwise. Majority, like any other function on n bits, can be computed on such a computer in time *exponential* in n. This is done by checking in turn if the input is any of the strings that make Majority equal one. This approach works for any function, but it's terribly inefficient. Can we do better for Majority? Can we compute it in time which is just a power of n? It is easy to convince oneself that this is impossible: After all, if you start counting bits, you'll soon run out of memory! Indeed, this was conjectured to be impossible (see Chapter 18).

And yet, we have the following surprising result.

Theorem 0.1. Majority can be computed on such a computer in time a power of n.

The proof of Theorem 0.1 is presented in Chapter 7 (see Theorem 7.21). It has deep ties with group theory, picture-hanging puzzles, and the relationship between time and memory in computation (all discussed in Chapter 7).

Moreover, Theorem 0.1 is not a trick tailored to majority. Many other problems, apparently much more complicated, have the same programs. In fact, such programs appear so powerful that it is consistent with our knowledge that every "textbook algorithm" can be solved in time n^c on such a computer! Proving a limitation on the power of such programs is an example of the Grand Challenge of complexity theory, which seeks to prove that natural computational tasks are hard to solve on various computational models. Results such as Theorem 0.1 suggest why the Grand Challenge is so hard to solve. It will be hard to rule out efficient programs, since they are so powerful and counterintuitive. In fact, complexity theory takes this a notch further. The theory shows that models such as the above can compute functions that *escape the reach of current mathematical proofs...* if you believe certain things, like that it's hard to factor numbers. This is discussed in Chapter 17 and enters some of the "mysticism" that surrounds complexity theory, where different beliefs and conjectures are pitted against each other in a battle for ground truth.

0.1.2 Randomness and derandomization

Suppose I tell you that I've tossed a coin 40 times and got this sequence of heads (0) and tails (1):

$$0101010101010101010101010101010101010101.$$

You would probably think this cannot be true. But suppose instead I claim that I got

$$1000111110100010011110100101111101100100.$$

Would you then think this is possible? But why feel this way? For a fair coin, the two strings have the same probability of 2^{-40}.

This example leads to a very interesting question: *what is randomness?*

Let's consider three possible notions of randomness:

1. **Classical:** Any string of n bits has probability 2^{-n}. The example above shows that this classical viewpoint doesn't capture our intuitive notion of randomness.

2. **Intrinsic (or ontological):** A string is less random the shorter the description it has. The first string has the short program "Print '01' for 20 times," while the shortest program for the second seems to be "Print '10001111101000100111101001011111011001 00'."

3. **Behavioristic:** Randomness is in the eyes of the beholder: A distribution on strings is random if you can't distinguish it from a uniform string.

The behavioristic turns out to be the most useful. For example, in a cryptographic context, we want secret messages to "look random" to an adversary, while still conveying information. Next we detail another application.

Consider the task of computing a prime number with n digits, for large n. There is a simple randomized algorithm for this:

Pick uniformly at random a number with n digits. Check if it is prime. If it isn't, repeat.

Classical results on the density of prime numbers (proved later, see Lemma 3.11) guarantee that this algorithm is efficient. However, it uses randomness and, despite much work, no efficient deterministic algorithm is known. This is a significant drawback. A deterministic algorithm gives a compact description of large primes, whereas a randomized algorithm does not, due to the random choices. This negatively affects a number of applications.

However, an amazing development of complexity theory says that *a deterministic algorithm exists under widely believed assumptions.* Moreover, this is true not only for finding primes, but for *every* efficient randomized algorithm (i.e., P = BPP). The widely believed assumptions are the existence of programs that are "hard" for *circuits,* a model of computation close to the hardware of a computer. Specifically, one needs programs that cannot be sped up by circuits – it would be quite surprising if such programs did not exist, i.e., if *every* program could be sped up by a circuit. This connection is known as "hardness vs randomness" and is proved by leveraging the program to construct randomness in a behavioristic sense. More specifically, we construct a *pseudorandom generator* whose output cannot be distinguished from truly random by any efficient algorithm.

To summarize:

Either randomized algorithms have efficient deterministic simulations, or programs can be sped up by circuits.

We prove this in Chapter 11 (see Theorem 3.15 and Problem 3.5).

Interestingly, this technology does not just give implications; it also gives *unconditional* results (for certain classes of algorithms). We also prove this in chapters 9 and 11 (see Corollary 11.35).

0.1.3 Proofs and delegating computation

The classical notion of a mathematical proof that of a (possibly very long) sequence of carefully justified steps that, taken together, guarantee the truth of a statement. Miss even a single step, and you risk accepting a bogus argument. Complexity theory has revolutionized this concept of proof, with far-reaching consequences — for example in a number of deployed systems from cryptography to digital currency and cloud computing.

One of the most striking examples is the PCP theorem, where PCP stands for *probabilistically checkable proofs*, which asserts the following:

Any proof can be written in a way such that inspecting only a constant number of randomly selected bits suffices to gain 99% confidence in its correctness.

At first glance, this seems impossible — how could one verify a proof without reading it all? One can easily concoct a mathematical proof that is correct except in a single step, and many great "discoveries" have failed in such a way. Indeed, a result like the PCP theorem was considered highly unlikely (cf Chapter 18). In Chapter 12 we will develop the key machinery that makes it possible, and provide an overview of the proof.

The PCP theorem is related to another main development in the theory of proofs, the introduction of *interactive* proofs where an efficient prover aims to check the validity of a proof by asking questions. We will prove in Chapter 10 that interactive proofs are much more powerful than traditional proofs. They allow us to verify efficiently the correctness of statements for which efficient (non-interactive) proofs are not known (IP = PSpace).

An instance of this is *delegating computation*. Let us illustrate it via an example. Suppose wanting to count the number of *triangles* in a given graph. See for example the graph in figure 2 where a triangle is highlighted. If the graph has n nodes and is given say as an $n \times n$ adjacency matrix, we can solve this in time about n^3, simply by trying all triples of nodes. Some improvements are possible, but it is not known if one can count the triangles in time close to the size n^2 of the graph. (Finer analyses would take into account the number of edges.)

However, we can *delegate this computation*. An external prove can run the count of triangles and provide a proof that the count is correct; *and we shall be able to check the proof in time about* n^2, which is close to optimal and much faster than if we counted the triangles ourselves. Moreover, the work of the external prover is still going to be feasible.

This example can be greatly generalized to a large class of problems, for example including linear algebra, and forms the basis for exciting developments in cloud computing. We prove these results about delegating computation in section §10.4.

0.1.4 Changing base losing no time and no space

Finally, here's a cute and fundamental data-structure problem with a counterintuitive solution. Given as input m *ternary* elements, called *trits*, the problem is to store them into

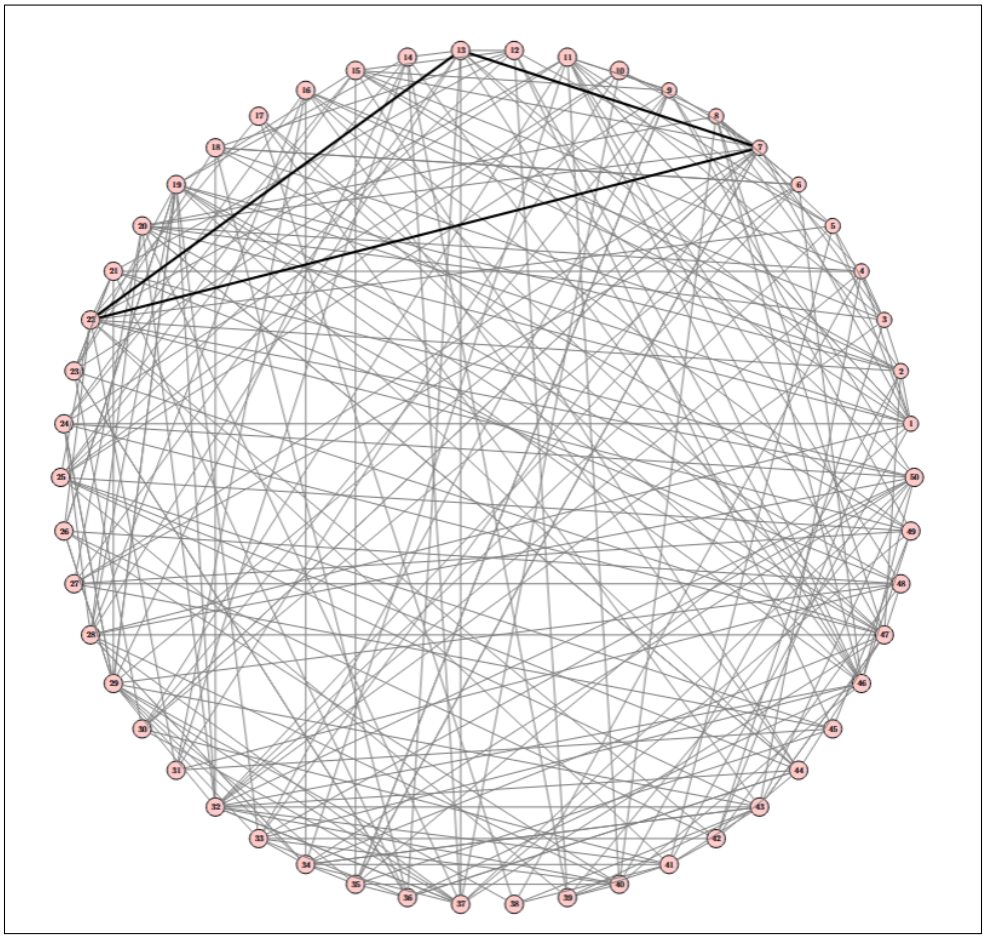


Figure 2: A graph with a triangle high-lighted.

memory, using bits, so that each trit can be retrieved quickly. There are two easy and antipodal solutions.

The first and simplest solution is to use 2 bits per trit. Viewing a trit as an element in $[3] = \{0, 1, 2\}$, we would map

$$0 \to 00$$
$$1 \to 01$$
$$2 \to 10.$$

This way we can retrieve each trit in constant time. However, we need to use $2m$ bits.

The other solution is to think of the m trits as a single number in $[3^m] = \{0, 1, 2 \ldots, 3^m - 1\}$, and simply write down its binary representation. For this, we only need

$$\lceil \log_2 3^m \rceil = \lceil m \cdot \log_2 3 \rceil \approx m \cdot 1.584 \ldots.$$

This is substantially less than the $2m$ space we used above. On the other hand, how to retrieve a trit? It seems very complicated to do so, and require to read the entire memory.

Thus, the first solution optimized time, the other space. Can we get the best of both worlds? Surprisingly, we can:

Theorem 0.2. We can store m trits using $\lceil \log_2 3^m \rceil$ bits so that each trit can be retrieved in constant time.

Sounds impossible? We will prove this in section 15.1.3.

This technology extends to a number of other problems, and can be seen as a computational strengthening of classical results in *information theory* about *data compression* and *source coding.*

0.2 Notation

The c notation. The mathematical symbol c has a special meaning in this text. Every *occurrence* of c denotes a real number > 0. There exist choices for these numbers such that the claims in this book are (or are meant to be) correct. Let us illustrate via few examples:

– "For all sufficiently large n" can be written as $n \geq c$.

– "It is an open problem to show that some function in NP requires circuits of size cn." This is a correct statement at the moment of this writing: one can replace this occurrence with 5.

– "$c > 1 + c$" is correct if we assign 2 to the first occurrence, 1 to the second.

– "$100n^{15} < n^c$" is correct for all large enough n. Assign $c = 16$.

– "$c < 1/n$ for every n". This is not true: No matter what we assign c to, we can pick a large enough n. Note the assignment to c is absolute, independent of n.

More generally, when subscripted this notation indicates a function of the subscript. For example:

– "For every ϵ and all sufficiently large n" can be written as $n \geq c_\epsilon$.

This notation replaces the big-Oh notation. For readers who replace the latter, a quick and dirty fix is to replace every occurrence of c in this book with $O(1)$.

Logarithm. log denotes logarithm with base 2 by default.

Polynomial. It is customary in complexity theory to bound quantities by a polynomial, as in *polynomial time*, when in fact only one monomial matters. It seems to me this makes some statements cumbersome, and lends itself to confusion since polynomials with many terms are useful for many other things. I use *power* instead of polynomial, as in power time (similarly to say "power law").

Proof details. The key aspects of a proof are sometimes obscured by low-level details, such as ensuring that parameters are integers or sufficiently large. In the early chapters, I try to address most of these details in proofs; in later chapters, as the proofs become more involved, I sometimes skip such details to focus the exposition.

Mathematical notation

$[i..j]$	The integer interval $\{i, i+1, i+2, \ldots, j\}$
$[i]$	$[0..i-1] = \{0, 1, 2, \ldots, i-1\}$
$[2]^n$	Binary strings of length n
$[2]^*$	Binary strings of any length
$[a,b]$,(a,b),$(a,b]$,...	Usual real intervals
$\lvert a\rvert$, $a \in \mathbb{N}$	Bit length, minimum number of bits to write a. We have $\lvert a\rvert = \lceil \log(a+1) \rceil$, with the exception $\lvert 0\rvert = 1$.
$\lvert x\rvert$, $x \in [2]^n$	n
$\lvert x\rvert$ x a tuple, program, circuit, etc.	Bit length of x in suitable encoding
f_n	The restriction of f to inputs of n bits
$i \mid j$	i divides j
$\mathbb{E}$	Expectation
$\mathbb{F}_q$	Field with q elements
$\mathbb{N}$	Natural numbers $\{0, 1, 2, \ldots\}$
$\mathbb{P}$	Probability
$\mathbb{Q}$	Rational numbers (from *quotient*)
$\mathbb{R}$	Real numbers
U	Uniform distribution (over suitable domain such as $[2]^n$)
$\mathbb{Z}$	Integer numbers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ (from *Zahlen*)
$\mathbb{Z}_m$	Integers $[m]$ with operations modulo m
§	Section

Abbreviations

a.k.a.	also known as
cf	compare
e.g.	as an example (*exempli gratia*)
i.e.	that is (*id est*)
iff	if and only if
lhs	left-hand side
prob.	probability
rhs	right-hand side
r.v.	random variable
s.t.	such that
w.h.p.	with high prob.
w.l.o.g.	without loss of generality

0.3 Contents and comparisons

This is an iconoclastic book that overhauls computational complexity theory, featuring recent breakthroughs, neglected gems, and simpler expositions of the classics. Details follow:

1. The presentation is also *geared towards an algorithmic audience*. Our default model is that of *word programs* (a.k.a. word RAM) (Chapter 1), the standard model for analyzing algorithms. This is in contrast with other texts which focus on tape machines. I reduce computation directly to quasi-linear 3Sat using sorting algorithms, and cover the relevant sorting algorithm. Besides typical reductions from the theory of NP completeness, I also present a number of other reductions, for example related to the 3SUM problem and the exponential-time hypothesis (ETH). This is done not only to showcase the wealth of settings, but because these reductions are central to algorithmic research. Also, I include a chapter on *data structures*, which are typically studied in algorithms yet omitted from complexity textbooks. I hope this book helps to reverse this trend; impossibility results for data structures squarely belong to complexity theory. Finally, a recurrent theme in the book is the power of restricted computational models. I expose *surprising algorithms which challenge our intuition* in a number of such models, including space-bounded algorithms, boolean and algebraic circuits, tape machines, and communication protocols.

2. The book contains a number of recent, exciting results which are not covered in available texts, including: catalytic computation (section §6.10), space-efficient simulations (Chapter 6), cryptography in NC^0 (section §7.7), amplifying lower bounds by means of self-reducibility (section §8.3), delegating computation (section §10.4), resampleability (Claim 11.27), elementary constructions of expanders avoiding iterated recursion (Chapter 12), number-on-forehead communication protocols for composed functions (section §13.2.2), succinct data structures (section 0.1.4), impossibility results for arith-

metic circuits (Chapter 14) and for ACC^0 (9.7), and natural-proof barriers that are informed by deployed cryptosystems (Chapter 17).

3. I also present several little-known but important results. This includes factoring efficiently using large words(Theorem 1.22), cosmological bounds (Theorem 2.14), the complexity of computing integers and its connections to factoring (section §14.2), and several results on pointer chasing section 13.2.3) and tape machines (Theorem 16.27, section §16.7).

4. A number of well-known results are presented in a different way, with the goal of demystifying them, exposing illuminating connections, and making the results more accessible by removing unnecessary mathematical barriers. Some of this was discussed above in 1. In addition, I work with partial (as opposed to total) functions by default. This affects many things, for example hierarchy theorems, see section §3.5. Unlike other texts where they appear later, here I present *circuits* and *randomness* right away, and weave them through the narrative henceforth. I introduce BPP and showcase its power via intuitive problems which require minimal mathematical background. Key results such as $\mathrm{BPP} \subseteq \Sigma_2\mathrm{P}$ and $\mathrm{PH} \subseteq \mathrm{BP}\oplus\mathrm{P} \subseteq \mathrm{P}^{\#\mathrm{P}}$ are presented through the lens of the circuit model and are connected to pseudorandom generators. In fact I replace the class $\mathrm{P}^{\#\mathrm{P}}$ with SymP which is easier to define while giving a stronger result that is more in line with a circuit-based exposition. In Chapter 11 I give a very simple proof of the XOR lemma. Also, I use the BIG-HIT generator to give constructions of pseudorandom generators from hard functions, avoiding some of the steps or technical machinery of previous constructions. Reductions are presented *before* completeness rather than later as in most texts. This, I believe, demystifies their role and leads to a transparent exposition as stand-alone results. Also, reductions are presented first as *implications*, then specialized in various ways. This affects several things, including the definition of NP-intermediate problems, see Exercise 5.12. On the other hand, implications are the most flexible type of reductions, and I think understanding implication is more fundamental than understanding more constrained types of reductions. The construction of expanders and the proof that undirected reachability is in L (both in Chapter 12) avoid eigenvalues and thus don't require non-trivial mathematical background such as the eigenbasis theorem. I strip down of information theory the communication lower bound for pointer chasing Theorem 13.7. Many results are presented in the context of circuits which retains the important points while avoiding some details of uniform models.

5. This book *challenges traditional assumptions and viewpoints*. For example, I discuss my reasons for not believing $\mathrm{P} \neq \mathrm{NP}$. In particular, I catalog and contextualize for the first time conjectures in complexity lower bounds which were later disproved (Chapter 18). Also, I emphasize that several available impossibility results may be "strong" rather than "weak" as commonly believed because they fall just short of proving major separations (e.g. section §6.3), and I expose the limitations of standard tools such as the hard-core set lemma (section 11.3.2). Finally, the notes put results in perspective.

To put results in context I routinely investigate what happens under slightly different assumptions. Finally, I present proofs in a "top down" fashion rather than "bottom up," starting with the main high-level ideas and then progressively opening up details, and I try to first present the smallest amount of machinery that gives most of the result. I decided to relegate references to the notes of each chapter and not spell out names of authors. Central results, such as PCP theorems or switching lemmas or derandomization, are co-authored by many people or span several papers, so that their history is better explained in the notes. To save some of the thrill of name-splashing, names appear in select portions which bend to the historical. Names also appear in the index, so one can for example look up "Markov's inequality" there.

Asymptotics and parameters Results in complexity theory are often stated in a way that hides parameter dependencies, for example though asymptotic or closure notation built into the underlying definitions. For example, a typical definition of the time complexity class $\text{Time}(t)$ corresponds to $a \cdot t$ steps on inputs of length $n \geq n_0$ performed by some program – this involves *three* additional parameters (a, n_0, and the program) as well as an asymptotic notion; moreover, the class P is often defined as the closure of such classes. The choice of hiding parameter dependencies may be a heritage from classical branches of mathematics, e.g., the limit notation in calculus. In complexity theory, it is sometimes a source of confusion and an obstacle. This text attempts to mitigate this, while preserving established definitions, in the following ways:

1. I use the c notation (see section §0.2) instead of asymptotic notation such as big-Oh $O(.)$ and big-Omega $\Omega(.)$, making all parameter dependencies explicit.

2. I give alternative definitions which reduce the number of parameters and/or avoid asymptotics. Continuing the previous example, I define $\text{Time}(t)$ as (exactly) t steps on a program P for all input length $n \geq |P|$; this removes two parameters and clarifies that the asymptotic input length can be the same as the program size. I make similar choices for circuits, space-bounded machines, tape machines, etc.

3. I make parameter dependencies (more) explicit throughout the text, by quantifying the parameter losses in simulations, inclusions, reductions, etc. This happens already in Chapter 1 (e.g., Theorem 1.19, Claim 1.21, Theorem 1.30) and is even more pervasive in later chapters covering more advanced results.

Summary of some terminological and not choices

Some other sources	this book
$O(1)$, $\Omega(1)$	c
polynomial time	power time
superpolynomial	superpower
mapping reduction (sometimes)	A reduces to B in P means $B \in \mathrm{P} \Rightarrow A \in \mathrm{P}$
NP-complete	complete for P vs NP
Extended Church-Turing thesis	Power-time computability thesis
pairwise independent	pairwise uniform
Turing machine	tape machine
TM with any alphabet	TM with fixed alphabet
classes have total functions	classes have partial functions
promise-P, promise-BPP, ...	P, BPP, ...
P/poly	CktP
L/poly	BrL
$\{0,1\}$	[2]
Time(t) $\to at(n)$ steps for $n \geq n_0$	$t(n)$ steps for $n \geq \lvert P \rvert$
PH $\subseteq \mathrm{P}^{\#\mathrm{P}}$	PH $\subseteq$ SymP

Further thoughts on notation. I avoid the term "random-access" which leads to strange expressions like "randomized random-access." I rarely use RAM in the main text, and don't expand it, but I think of "rapid-access machines." Finally, writing $f : 2^n \to 2$, thinking $2 = \{0,1\}$, could be beneficial, but I fear this is pushing it a little.

0.4 How to use this book

This book is intended both as a textbook and as a reference book. The intended audience includes students at all levels, and researchers, in both computer science and related areas such as mathematics, physics, data science, and engineering. The text is interspersed with exercises which serve as quick concept checks, for example right after a definition. More advanced problems are collected at the end of each chapter. Solutions or hints for both exercises and problems are provided as separate manuals. I assume no background in theory of computation, only some "mathematical maturity" as can arise for example from typical introductory courses in discrete mathematics. In fact, a point of this book is to strip results of complicated background and present them in a new and more accessible way (cf section §0.3). All other mathematical background is covered in Appendix A.

The book can be used in several different types of courses.

- For an introductory course in complexity theory, suitable for an undergraduate student who has had some prior exposure to algorithms and mathematical thinking, one can

cover Chapters 1 to 5. At the same time the text can expose the interested students to more advanced topics, and stimulate their critical thinking.

- For a broader course in complexity, suitable for advanced undergraduate students or for graduate students, one can add Chapters 5.5 to 10. Such a course can be supplemented with isolated topics from later chapters. For example, in my offerings of cross-listed undergraduate/graduate PhD complexity theory, I typically cover Chapters 1 to 10 and then one or two select chapters from 11 to 17. The pace is about one chapter a week, and I ask the students to attempt all exercises.

- For a special-topics course or seminar one can use Chapters 11 to 17. One possibility, which I tested, is covering all these chapters.

Chapters 1 to 11 are best read in order. Chapters 12 to 17 have fewer dependencies and can be read more or less in any order.

0.5 Acknowledgments

Obviously, this work would not have been possible had I not had the privilege of interacting with many of the leading complexity theorists over the years. Many people helped in various ways specifically with this book, including: Eric Allender, Noga Alon, Ravi Boppana, Sam Buss, Peter Bürgisser, Bill Gasarch, Oded Goldreich, Songhua He, Valentine Kabanets, Michal Koucký, Kasper Green Larsen, Nutan Limaye, Bruno Loff, Ramprasad Saptharishi, Oded Schwartz, Srikanth Srinivasan, Justin Thaler, Salil Vadhan, Avi Wigderson. I am also indebted to the students in my classes and the readers of my blog, where some of this material appeared first, beginning in January 2023.

Chapter 1

Time

> *Concluding, I view the mystery of the difficulty of proving (even the slightest non-trivial) computational difficulty of natural problems to be one of the greatest mysteries of contemporary mathematics.*

In this chapter we introduce our first – and a main – model of computation. In general, computation has two fundamental features:

- **Locality.** Computation proceeds in small, quick, and local steps. Each step only depends on and affects a small amount of "data." For example, in the grade-school algorithm for addition, each step only involves a constant number of digits.

- **Generality.** The computational process is general in that it applies to many different inputs. At one extreme, we can think of a single algorithm which applies to an infinite number of inputs. This is called *uniform* computation and is explored in this chapter. Or we can design algorithms that work on a finite set of inputs. This makes sense if the description of the algorithm is much smaller than the description of the inputs that can be processed by it. This setting is usually referred to as *non-uniform* computation and is explored in Chapter 2.

We seek a model that is realistic: it should be close enough to how computers work and are programmed. At the same time, it should not be overly complicated to define and to reason about.

1.1 Word programs

Our model can be seen as a simple machine language, or assembly, not unlike that used by microprocessors. We think of computing over *words* of w bits. Memory is organized as an array M of such words. Also, there is an array R of words corresponding to a constant number of *registers*. We allow basic arithmetic operations among registers, and read from and write to memory. We have a basic control-flow instruction, GOTO...IF which allows

us to create loops similar to the *for* or *while* loops in many programming languages. In addition, we include the instruction MALLOC that increases the number of memory words.

Definition 1.1. A *word program* of size ℓ is a sequence of ℓ instructions of the following type:

- $R[i] := R[j]$, where $i, j \in [\ell]$ (copying a register)
- $R[i] := b$, where $i \in \ell$ and $b \in [2] = \{0, 1\}$ (setting a register to constant)
- $R[i] := E$, where $i \in [\ell]$ and E is a *basic expression,* which is one of the following operations or relations, among two registers or one register and the constant 1:
  - $+, -, *, \%$ addition, subtraction, multiplication, and modular remainder
  - $<<, >>$, left and right bit shift
  - $\wedge, \vee, \neg$ bit-wise and, or, and negation
  - $>, <, \leq, \geq, =$, order relations;
- $R[i] := M[R[j]]$, where $i, j \in [\ell]$ (reading from memory)
- $M[R[i]] := R[j]$, where $i, j \in [\ell]$ (writing to memory)
- GOTO i IF E, where $i \in [\ell]$, and E is a basic expression as above
- MALLOC (increasing memory)
- STOP

Word programs are sometimes called word RAMs, cf section §0.2.

Note the above definition is redundant. For example, we can implement $R[i] := R[j]$ as $R[i] := R[j] * 1$. Many other simplifications are possible. However, we keep the above syntax for clarity. Later we will consider various minimalistic programs, including some that only have two instructions (see section §7.5).

An important concept that we will see again and again is that of a *configuration*. A configuration is a "snapshot" of the execution of a program at a certain point that contains all the information required to carry on the computation after that point, assuming the program is known.

Definition 1.2. The *configuration* C of a program of size ℓ is a tuple

$$C = (i, m, w, R, M)$$

where $i \in [\ell]$ is the *program counter*, $m \in \mathbb{N}$ is the *space*, $w \in \mathbb{N}$ is the word length, $R \in [2^w]^\ell$ is the array of registers, and $M \in [2^w]^m$ is the memory.

As indicated earlier, a computation step makes *local changes*. To indicate such changes, the following notation is useful. For a vector V (e.g., R, M) we write $V[i \leftarrow x]$ for the vector V' where $V'[i] = x$ and $V'[j] = V[j]$ for all $j \neq i$. Regarding memory, we shall maintain the invariant that the word length is sufficient to write the number m of memory words, so w is at least the *bit length* of m, denoted $|m|$. In particular this allows us to index the memory words (for which $w \geq |m-1|$ suffices). (Recall from section §0.2 that $|m| = \lceil \log(m+1) \rceil$ for $m \geq 1$.)

We can now formally define computation.

Definition 1.3. Let P be a word program of size ℓ. Let $C = (p, m, w, R, M)$ be a configuration, and let instruction p of P be I. Then C *yields* configuration C' as follows:

- If $I =$ "$R[i] := E$" then $C' = (p+1, m, w, R[i \leftarrow E], M)$. Here the operations in E are performed modulo 2^w: The operation is performed over naturals, but only the least significant w bits are stored in $R[i]$. Subtraction $R[j] - R[k]$ is defined as 0 if $R[j] < R[k]$. Modular remainder $R[i]\%0$ is defined as $R[i]$. Relations (e.g., $R[i] < R[j]$) take value 1 if true, 0 otherwise.

- If $I =$ "$R[i] := M[R[j]]$" then $C' = (p+1, m, w, R[i \leftarrow M[R[j]]], M)$, where $M[R[j]]$ is defined to be 0 if $R[j] \geq m$.

- If $I =$ "$M[R[i]] := R[j]$" then $C' = (p+1, m, w, R, M[R[i] \leftarrow R[j]])$ if $i < m$, otherwise M is unchanged.

- If $I =$ "MALLOC" then $C' = (p+1, m+1, \max\{w, |m+1|\}, R, M[m \leftarrow 0])$.

- If $I =$ "GOTO i IF E" then $C' = (i, m, w, R, M)$ if $E \geq 1$, and $C' = (p+1, m, w, R, M)$ if $E = 0$.

A *computation* is a sequence of configurations corresponding to a program.

Definition 1.4. A *computation* of a word program P is a sequence of configurations $C_0, C_1, \ldots, C_t$ such that C_i yields C_{i+1} for $i \in [t]$. We say that P computes C_t from C_0.

In general, configuration C_0 starts with some information, or input, and C_t contains some additional information, written in the registers, or memory, or some combination.

Example 1.5. We give a program for *CountingSort* starting from suitable configurations. The space is $m := ct$ (recall the "c" notation from section §0.2), the first t memory words $M[0..t-1]$ contain numbers in $[t]$, the other memory words are 0, the word size is $c\lceil \log m \rceil$, and $R[0] := t$. The following program will place the sorted sequence of numbers into words $M[2t..3t-1]$. The words $M[t..2t-1]$ are auxiliary words used to count the number of occurrences. Typically, these two sub arrays of M would be given different names; this example illustrates how adding suitable shifts to memory indexes, we can use a single array to simulate many arrays.

/* Loop to set $M[t+i]$ to the number of input words equal to i:

```
   For (R[1] = 0; R[1] < R[0]; R[1] + +)
     M[R[1] + R[0]] + +;
*/
0: R[2] := M[R[1]]
1: R[2] := R[2] + R[0]
2: R[3] := M[R[2]]
3: R[3] := R[3] + 1
4: M[R[2]] := R[3]
5: R[1] := R[1] + 1;
6: GOTO 0 IF R[1] < R[0]
/* Loop to set M[t + i] to the number of input words equal or less than i:
   For (R[1] = 1; R[1] < R[0]; R[1] + +)
     M[R[1] + R[0]]+ = M[R[1] + R[0] − 1];
*/
7: R[1] := 1
8: R[2] := R[1] + R[0]
9: R[3] := R[2] − 1
10: R[4] := M[R[2]]
11: R[5] := M[R[3]]
12: R[6]:=R[4] + R[5]
13: M[R[2]] := R[6]
14: R[1] := R[1] + 1;
15: GOTO 8 IF R[1] < R[0]
/* Loop to place each number at right location.
   For (R[1] = 0; R[1] < n; R[1] + +) {
     M[M[M[R[1]] + R[0]] + 2 ∗ R[0]] = M[R[1]];
     M[M[R[1]] + R[0]] − −;
   }
*/
16: R[1] := 0
// M[M[M[R[1]] + R[0]] + 2 ∗ R[0]] = M[R[1]];
17: R[3] := M[R[1]]
18: R[2] := R[3] + R[0]
19: R[2] := M[R[2]]
20: R[2] := R[2] + R[0]
21: R[2] := R[2] + R[0]
22: M[R[2]] := R[3]
// M[M[R[1]] + R[0]] − −;
23: R[3] := R[3] + R[0]
24: R[4] := M[R[3]]
25: R[4] := R[4] − 1
26: M[R[3]] := R[4]
```

27: $R[1] := R[1] + 1$;
28: GOTO 17 IF $R[1] < R[0]$
29: STOP

The purpose of this example was to convince ourselves that standard algorithms can be implemented as word programs. Going forward, we will rarely need to write down programs explicitly; a high-level description suffices.

1.2 Complexity classes

In Example 1.5 we have started from a somewhat *ad hoc* configuration: The input was given in words, we had enough memory for the scratch array, and the output was written after that. If we didn't have enough memory, we would have needed to use MALLOC to allocate it. If the input was given in bits, we would have needed to translate it into words. To talk about the complexity of computing arbitrary functions, we now fix some input-output conventions.

The input is going to be a bit-string $x = (x_0, x_1, \ldots, x_{n-1}) \in [2]^n$. We use $|x|$ to denote n. We often need to work with *more structured* objects, like tuples, graphs, matrices, programs, etc. One can always encode such objects in binary, and often ignore the details of such encodings. For a simple example, we can encode a pair (x, y) where $x, y \in [2]^*$ by inserting a 0 in front of each bit of x except the last which has a 1 in front:

$$(x, y) \to 0x_0 0x_1 \cdots 0x_{|x|-2} 1x_{|x|-1} y. \tag{1.1}$$

Such a string uniquely specifies x and y. The length of this representation is $2|x|+|y|$. We are doubling the bits in x, but this blow-up hardly concerns us. Still, succincter encodings exist, see Problem 1.1. At the beginning of the computation, the input is in words $M[0..n-1]$, one bit per word. As mentioned earlier, the word length w is set to the minimum that can write the number m of memory words, and at the beginning $m = n$, so $w := |n|$. Other choices for the word size are discussed in section §1.7. For some problems (e.g., sorting, see Example 1.5) it is actually natural to given the input in words, not bits. Typically, one can quickly convert between the two representations, so this distinction will not make a difference in most settings, while working with bits suffices for and simplifies most of the presentation.

Exercise 1.6. Extend the encoding above (Equation (1.1)) to tuples; then give a high-level description of a word program that converts an input tuple $x \in [m]^m$ (given as above one bit per memory word) into the format expected in Example 1.5.

Registers are initialized at 0, except for $R[0] = n$. At the end, the output is written in $M[0], M[1], \ldots$, again one bit per cell, and $R[0]$ is the length of the output.

Definition 1.7. We say that a word program P computes $y \in [2]^*$ on input $x \in [2]^*$ in *time* t if there is a computation $C_0, C_1, \ldots, C_t$ such that:

- C_0 is the start configuration $(0, n, w, R, M)$ where $n = |x|$, $w := |n|$, $R[0] = n$ and the other $R[i]$ are 0, and $M[i] = x_i$ for $i \in [n]$ while $M[i] = 0$ for $i \in [n..2^w - 1]$.

- C_t is a configuration (p, m, w, R, M) where instruction p of P is STOP, $R[0] = |y| \leq m$, and $M[0] = y_0, M[1] = y_1, \ldots, M[|y| - 1] = y_{|y|-1}$.

Example 1.8. The following program on input $x \in [2]^*$ outputs xx, i.e., two copies of x. We use $R[1]$ to loop from 0 to $n - 1$, and $R[2]$ to loop from n to $2n - 1$; each time we copy $M[R[1]]$ into $M[R[2]]$ using scratch register $R[3]$. We call MALLOC to allocate memory words for the output.

0: $R[2] := R[1] + R[0]$ // $R[2]$ points to next symbol of 2nd copy
1: $R[3] := M[R[1]]$
2: MALLOC
3: $M[R[2]] := R[3]$
4: $R[1] := R[1] + 1$
5: GOTO 0 IF $R[1] < R[0]$
6: $R[0] := R[0] + R[0]$ //Double $R[0]$ to indicate output length
7: STOP

Next we define complexity classes. First, some remarks.

- We allow *partial functions,* i.e., functions with a domain X that is a strict subset of $[2]^*$, as opposed to *total functions* which are defined over the entire $[2]^*$. Partial functions are a natural choice for many problems.

- We measure the running time of the machine in terms of the *input length,* denoted n. Running times can be complicated expressions such as $t(n) = 2n^3 \log(n + 1) + 3n + 15$ which are hard to make sense of and depend on irrelevant details of the model, e.g., is the constant 2 allowed or do we need to build it via the expression $1 + 1$? We'd like to focus on the leading terms only, and say for example that the former expression is $\leq 3n^3 \log n$. This is true for large enough n, but not for small n. For example, if $n = 1$ then $\log n = 0$, so the expressions $3n^3 \log n$ would be 0, but $t(n)$ is not 0. As these details do not add to our understanding, we adopt the convention that the running time bound only has to hold for inputs larger than a constant. Jumping ahead, it will sometimes be useful to allow more time than the input length. For example, we may want to allow time k for some parameter k. This can be accomplished by including in the input a "pad" of k ones, denoted 1^k.

- Depending on the context, it is convenient to work with complexity classes of *boolean* functions, i.e., range [2], or with functions with multi-bit output, i.e., range $[2]^*$. The multi-bit variant is denoted with an "F" in front. More generally, we will be interested in computing not just functions but *relations*. That is, given an input x there will be a *set* $f(x)$ of several possible outputs, and we just want to compute any y in $f(x)$. The case of functions is when $f(x)$ is a singleton set $\{y\}$, in which case we simply write $y = f(x)$. The "F" variant will be defined for this general case of relations. Many natural problems naturally give relations, one example being factoring (see 1.14).

Definition 1.9. [Time complexity classes] Let $t : \mathbb{N} \to \mathbb{N}$ be a function. Time(t) is the set of functions that map a subset $X \subseteq [2]^*$ to $[2]$ for which there exists a word program P that computes $f(x)$ in time $\leq t(|x|)$ on any input $x \in X$ of length $\geq |P|$.

We also define

$$\text{Linear-Time} := \bigcup_{d \geq 1} \text{Time}(dn),$$

$$\text{Quasi-Linear-Time} := \bigcup_{d \geq 1} \text{Time}(n \log^d n),$$

$$\text{Power-Time P} := \bigcup_{d \geq 1} \text{Time}(n^d),$$

$$\text{Exponential-Time Exp} := \bigcup_{d \geq 1} \text{Time}(2^{n^d}).$$

Correspondingly, FTime(t) is the set of functions that map a subset $X \subseteq [2]^*$ to subsets of $[2]^*$ for which there exists a word program P that computes some $y \in f(x)$ in time $\leq t(|x|)$ on any input $x \in X$ of length $\geq |P|$. We simply write $y = f(x)$ for the case of functions (where $f(x) = \{y\}$). Also,

$$\text{FP} := \bigcup_{d \geq 1} \text{FTime}(n^d).$$

When computing a boolean function we can also say that the program *accepts* an input if it outputs 1 and *rejects* if it outputs 0. We sometimes use corresponding instructions Accept and Reject in high-level programs for brevity.

Remark 1.10. Here $|P|$ is the length of the program. This Definition 1.9 does not change if "length $\geq |P|$" is relaxed to "length $\geq n_0$" for another constant n_0 depending on P. Indeed, we can easily pad the length of P so that it's larger than n_0, for example by adding n_0 useless instructions $R[0] := R[0]$ at the end. The relaxed definition can be more useful when writing programs, as we wouldn't want to write the padding. But this book is mostly about *analyzing* programs, and the choice $n_0 = |P|$ shortens some definitions and proofs (e.g., time hierarchies). Similarly, Definition 1.9 does not change if we insist that P computes f correctly even on inputs of length $< |P|$ (in some time). Indeed, we can modify the program so that it contains the value of f for all those inputs. Depending on the circumstance, we will use the variant that is most convenient and makes the arguments more transparent. Finally, sometimes Time(t) is defined to correspond to at steps for a constant a, as opposed to $\leq t$ steps here. Our choice is perhaps more in line with the way time is stated in algorithms, and fits well with certain results such as the hierarchy Theorem 1.30.

An analysis of the program in Example 1.8 shows that duplicating a string is in FTime(cn). As mentioned earlier, we usually do not write programs but give higher-level descriptions, as illustrated by the next examples.

Example 1.11. Computing the Or of n bits is in Time(cn). Indeed, we can scan the input once, keeping track of the Or of the bits seen so far in a register, and in the end write that register to memory. This takes time $cn + c$, which is $\leq cn$ for large enough n.

Example 1.12. Addition of 2 naturals is in FTime(cn). We can implement the grade-school algorithm. Given x, y, we compute their sum bit-by-bit, starting with the least-significant bit and keeping track of the carry. This requires constant time per bit. This sum is written in a new memory array, then copied in the output. This takes linear time. Overall, the time is $\leq cn$ for large enough n.

Exercise 1.13. Prove that multiplication of 2 naturals in in FP.

We note that the definitions of complexity classes allow for arbitrary constants (e.g., the exponent of log in the definition of Quasi-Linear-Time). In practice, natural problems falling within these classes tend to have reasonable constants leading to algorithms that can be deployed. On the other hand, the constants can lead to unpractical results. This point is well illustrated by Theorem 1.19 and Problem 1.2.

For many important problems, whether they are in FP or not is not known. We will encounter many such problems in this book, e.g. in Chapter 4. For the discussion in this chapter, it suffices to consider factoring:

Definition 1.14. The Factoring problem: Given $x \in \mathbb{N}$, compute a prime factor of x.

It is not known if Factoring is in FP. A common conjecture is that it is not; moreover many deployed cryptographic systems (for electronic commerce, etc.) rely on the infeasibility of factoring numbers, in fact products of two primes, of about 1000 digits.

Conjecture 1.1. Factoring is not in FP.

The fastest algorithm that we have runs in exponential time 2^{n^c}.

1.3 You don't need much to have it all

How powerful are word programs? Perhaps surprisingly, they are all-powerful.

Power-time computability thesis. Power-Time, P, is the same for word programs as for any "realistic" model of computation.

This is a *thesis*, not a *theorem*. The meaning of "realistic" is a matter of debate, and one challenge to the thesis is discussed in Chapter 3. However, the thesis can be proved for many standard computational models, which include all modern programming languages. The proofs aren't hard, and amount to designing suitable *compilers*. Essentially, one just goes through each instruction or construct in the language and gives a implementation in a word program. In Example 1.5 we have basically done this: We have shown how to write "for" loops and complicated expressions as word programs. Less obvious is that even models that appear much more restricted are equivalent to word programs. We will see this in Chapter 16.

A main source for our interest in word programs is that one can formulate a bolder thesis, namely that word programs capture efficient computation up to lower-order factors:

Quasi-Linear-Time computability thesis. Quasi-Linear-Time is the same for word programs as for any "realistic" model of computation.

Example 1.5 supports the Quasi-Linear-Time computability thesis, because the word-program implementation of quick sort runs in linear time.

Because of these theses, word programs will be often identified with and simply called "algorithms," and we can appeal to "algorithmic intuition" when asserting the existence of a program. To develop such intuition, next we show how some intuitive composition properties of algorithms work for word programs.

1.4 Composing programs

Programs are usually made by composing many simpler programs. In this section we prove some composition results which will be used frequently, often implicitly. This serves as an illustration of the model and increase our confidence in the computability theses from section 1.3.

The next result illustrates a basic property of FP: Closure under composition. We mostly use this for functions, where the composition $g \circ f$ is simply defined as $x \to g(f(x))$. But it works just the same for relations, where the notation $g(f(x))$ stands for $\bigcup_{y \in f(x)} g(y)$.

Claim 1.15. If f and g are in FP then the composition $h(x) := g(f(x))$ is in FP.

Proof. Let $f \in \text{FTime}(n^a)$ and $g \in \text{FTime}(n^b)$ and let P_f and P_g be corresponding programs as in Definition 1.9. The idea is simply to run P_f and then P_g. The output length of f on an input of length n is $\leq n^a + n$, because P_f can only execute so many MALLOC instructions, and the output length is bounded by the memory available, see Definition 1.7. Hence $P_g(f(x))$ will run in time $\leq (n^a + n)^b$ if the $|f(x)| \geq |P_g|$. Otherwise, P_g will take at most a number s of steps, the maximum over all inputs of length $\leq |P_g|$. Overall, the running time of the combined program on an input of length n is

$$n^a + (n^a + n)^b + s \leq n^{ab+1}$$

for large enough n, as desired.

However, P_f may stop in a configuration with more memory than the start configuration of P_g, and we do not know how P_g behaves in that case. To address this, we modify P_g as follows. We use registers to keep track of a new word size w' and space m', initialized to the values $||f(x)||$ (i.e., the number of bits needed to write down the length $|f(x)|$ of $f(x)$) and $|f(x)|$, as in the starting configuration on input $f(x)$. After each instruction we truncate the registers to word size w', using bit-wise And. When accessing memory, the index is compared to m' to determine if it is out of boundary. When MALLOC is executed, we increase m', and adjust w' accordingly. These changes allows us to simulate the behavior of P_g on the input $f(x)$, and only increase the time by a constant factor. **QED**

Typical programs (including the ones in Example 1.5 and Example 1.8) work just as well when started from a configuration with larger memory and word size. We call such programs *compatible*. For compatible programs, the above proof can be simplified by skipping the last paragraph, i.e., we can simply run the programs one after the other as they are. Alternatively, we could have used a FREE instruction that is the opposite of MALLOC; but the approach above is more flexible.

In addition, we often need to run a program within the context of another computation, i.e., as a *subroutine*. That is, we are in some configuration and want to compute a function f on some input x which is in memory, while keeping the memory intact (so we can run other programs). This situation is captured in the following claim.

Claim 1.16. Let $f \in \text{FP}$. Then the function $f'(x, y) := (x, y, f(x))$ is in FP.

Proof. We begin by copying x into $|x|$ new memory words, starting at word $n = |(x, y)|$. Now we run a program P for f on this copy, but modify it by adding n to all memory addresses accessed. This would be the end of the proof if P was compatible. Otherwise, as in Claim 1.15 we need to address the fact that P was designed to start with $|x|$ words of memory; we can use the same solution. **QED**

Example 1.17. Given a natural x, computing x^4+x^2 is in FP. Indeed, we know that squaring is in FP by Exercise 1.13. Using Claim 1.16 we can compute $x \to (x, x^2)$ in FP. Composing this via Claim 1.15 with squaring and addition (Example 1.12) we obtain $x \to x^4 + x^2$.

1.5 Universal programs

Universal programs can simulate any other program on any input. These programs play a critical role in some results we will see shortly. They also have historical significance: before them machines were tailored to specific tasks. One can think of universal programs as epitomizing the victory of *software* over *hardware:* A single machine can be programmed to simulate any other machine. A universal program will thus take as input both another program Q and an input x for Q (in a standard encoding such as equation (1.1)).

Lemma 1.18. There is a word program U such that for any word program $Q \in [2]^*$ and input $x \in [2]^*$ we have: If Q computes $y \in [2]$ from x in time t then U on input Q and x computes y in time $c(t + |Q| + |y|)$.

Proof. To achieve the claimed fast simulation, it is convenient to represent word programs in a specific format. A program of size ℓ will be represented as a list of 4ℓ words. For each instruction we have 4 words. The first has a constant range and denotes the type of the instruction. The other 3 words contain numbers in $[\ell]$ which represent the indices or the program counter in the instruction. E.g., for the instruction GOTO 15 IF $R[7] < R[2]$ the three numbers would be $15, 7, 2$.

The program U first computes this representation for Q. This can be done in linear time, by scanning the representation of Q in a read-once fashion, as in Exercise 1.6.

After that, U arranges the memory so that the first words contain the above representation of Q, followed by the starting configuration of Q on input x. Then U simulates the instructions of Q, one at the time. Each instruction of Q takes constant time to simulate. U first reads from the configuration the program counter, then reads the corresponding instruction of Q and the corresponding indices, if any. Then it reads or writes the corresponding words from the configuration of Q.

The remaining details are the same as for Claim 1.16. We map memory word i of Q to memory word $i + s$ of U, where s is the index of the first memory word in the configuration of Q. Also, while simulating instructions of Q, the instructions of U use the larger word size, but then U truncates the output to the word size of Q.

E.g., if the instruction was the memory-read $R[5] := M[R[3]]$ then U reads the content of $R[3]$ from the configuration. Let this be v. It then reads the memory at location $s + v$, and finally updates the content of $R[5]$ in the configuration, truncating it to the word size of Q.

At the end, U can copy the output y of Q in the first $|y|$ words. **QED**

Some variants of this result will be used. First, we will often use a "clocked" version of universal machines, where the machine starts with a time bound t in a register, and only carries the simulation for t steps. This can be easily implemented within the same time, by decreasing the clock at every instruction. Second, we can insist that U is a total function, that is, it takes as input any bit string. Not all strings would immediately correspond to valid programs, but it is easy to tell which ones are, and we can interpret any instruction which does not parse as, say, STOP. This way every string corresponds to a program.

1.6 The fastest algorithm for Factoring

A curious fact about some problems is that we know of an algorithm which is, in an asymptotic sense to be discussed now, essentially the fastest possible algorithm. This algorithm proceeds by simulating every possible program. When a program stops and outputs the answer, we can *check it* efficiently. Naturally, we can't just take any program and simulate it until it ends, since it may never end. So, we will use a clocked simulation. There is a particular simulation schedule which leads to efficient running times. For concreteness we state this for Factoring, but the same ideas will apply to other problems.

Theorem 1.19. There is a word program U that computes Factoring and has the following property: For any word program P for Factoring, and every input x, if P runs in time t on x then U runs in time $c_P t + c_P |x|^c \log t$.

This is a striking result. There is a single algorithm that does nearly as well as any other. In particular, Factoring is in FP iff U runs in power time. Moreover, the algorithm U is explicitly given: we can program it and run it. Naturally, the catch is that it is not

practical. Indeed, the result nicely illustrates how "constant factors" can lead to impractical results because, of course, the problem is that the constant in front of t can be enormous, see Problem 1.2.

Proof. For $i = 1, 2, \ldots$ we simulate program i for 2^i steps. As discussed in section 1.5, a modification of Lemma 1.18 guarantees that for each i the simulation takes time $c2^i$. If program i stops and outputs y, then U checks in time $|x|^c$ if y is prime and divides x. For this we use that deciding primality is in P (see the notes).

Now let P be a word program computing Factoring. In the enumeration of programs, each program appears infinitely often. For example, one can add useless STOP instructions. So, if P has a description of length c_P, it also has descriptions of length $c_P + j$ for any j. (Note that adding a STOP instruction may take more than 1 bit, but again as discussed in section 1.5 anything which doesn't parse can be interpreted as STOP, so indeed we can have descriptions of length $c_P + 1, c_P + 2$,etc.) Let us take the shortest description which has length $\geq \log t$, which suffices to terminate the simulation. Let ℓ be the length of this description, and note $\ell \leq c_P + c \log t$.

The time spent by U for a fixed i is $\leq c \cdot 2^i + |x|^c$. Hence the total running time of U is

$$\leq \sum_{i=1}^{\ell} \left(c2^i + |x|^c \right) \leq c_P 2^{\ell+1} + \ell |x|^c \leq c_P t + c_P |x|^c \log t.$$

QED

1.7 On the word size

Having defined and illustrated the model, we return to an issue related to its definition: the word size. While the MALLOC operation allows to increase the word length, the latter does not get too big. Specifically, if a word program runs in time t it can only use $\leq t$ MALLOC instructions, and hence $\leq n + t$ memory words. In particular, the word length w will remain $\leq |n + t|$. For some problems it is natural to start with slightly larger word size, like $10\,|n|$. (Recall from section §0.2 that $|n|$ is approximately $\log n$.) This allows for example to compute an integer like n^2 in a register. However, one can simulate such a register using word size $\log n$ with only a constant factor increase in number of registers and running time. We now show this. For concreteness let us first define a model with larger word size.

Definition 1.20. We say that a word program computes y on input x *with word size* $b(n)$ if it computes y as in Definition 1.7, except that the start configuration in Definition 1.7 has word size $w := b(n)$ (as opposed to $w := |n|$ in Definition 1.7). The classes $\text{FTime}(t(n))$ with word size $b(n)$ are defined correspondingly.

The next claim shows that increasing the word size by any constant only impacts the running time by a constant factor.

Claim 1.21. $\text{FTime}(t(n))$ with word size $a \cdot |n| \subseteq \text{FTime}(c_a t(n))$, for every $a \in \mathbb{N}$.

Proof. We only sketch this proof for $a = 2$. We use 4 w-bit registers to represent 1 register R' of $2w$ bits, where each w-bit register is *half-empty*: has non-zero bits only in the least significant $w/2$ bits:

$$R' = 2^{3w/2}R[3] + 2^w R[2] + 2^{w/2}R[1] + R[0].$$

To simulate addition between $2w$-bit registers, we first add the corresponding 4 w-bit registers. The results will fit in our w-bit registers, since the registers are half-empty. To maintain half-emptiness, we pick the most significant $w/2$ bits from the $R[0]$, add them to $R[1]$ and remove them from $R[0]$. Now $R[0]$ is half-empty. We do the same for $R[1]$, and so on.

A similar approach works for the other instructions, see Problem 1.3. **QED**

So we can typically assume we have such longer words when designing algorithms, but restrict to the smaller word size when analyzing.

What about even larger words? One is tempted to brush aside details and consider *unbounded* word sizes, where all computation is performed over integers. Indeed, this model is sometimes a useful abstraction when writing algorithms. However, some care is needed because the ability to perform arithmetic with unbounded (or very large) integers gives surprising power. As we now show, this allows us to factor integers efficiently!

Theorem 1.22. There is a function $b(n)$ s.t. factoring $\in \text{FTime}(n^c)$ with word size $b(n)$.

1.7.1 Factoring with large words

In this section we prove Theorem 1.22. The proof showcases and ties together in an unusual but not overly complicated way many fundamental algorithms, including repeated squaring, recursion, binary search, and greatest common divisor. We break it up in a series of steps, interesting in their own right. We show that given an n-bit integer x we can compute the following in time n^c, *into a register* (i.e., we reach a configuration where the desired quantity is in a register):

(0) x (i.e., x is "loaded" into a register).

(1) Exponentiation y^x, given any integer y in a register (with any number of bits),

(2) Factorial $x!$

(3) A prime factor of x.

We note that (1) and (2) involve numbers with $\geq 2^n$ bits. Here is where we leverage the large word size. Next we prove these points in turn.

Exercise 1.23. Prove (0).

(1) This is done via *repeated squaring*. Namely, write

$$x = \sum_{i=0}^{b} 2^i x_i$$

and note

$$y^x = \prod_{i=0:x_i=1}^{b} y^{2^i},$$

where a product of no terms is defined to be 1. The numbers y^{2^i} can be computed by repeated squaring in time cb, because $(y^{2^i})^2 = y^{2^{i+1}}$. Then we can just multiply together those corresponding to $x_i = 1$.

(2) We give a recursive algorithm.

If x is odd we reduce to the case of x even using $x! = x \cdot (x-1)!$.

If x is even we use

$$x! = \binom{x}{x/2}(x/2)!^2.$$

The factorial in the rhs is computed recursively. To compute the binomial we use the binomial theorem to write, for any ℓ:

$$(2^\ell + 1)^x = \sum_{j=0}^{x} \binom{x}{j} 2^{\ell \cdot j}.$$

Note the binary representation of the rhs contains all the binomials $\binom{x}{j}$ spaced out. Specifically, $\binom{x}{j}$ starts at bit $\ell \cdot j$. Since each binomial is $\leq 2^x$ and thus takes $\leq x$ bits, picking $\ell := x$ ensures that the binary representations of the binomials do not overlap in the binary representation of the lhs. Hence, the bits for the binomial $\binom{x}{x/2}$ are an interval of the bits of the lhs.

For example, for $\ell = x = 6$, the binary representation of $(2^\ell + 1)^x$ is

$$1\ 000110\ 001111\ 010100\ 001111\ 000110\ 000001$$

and you can verify that block i of 6 bits is the binary representation of $\binom{6}{i}$. For example, $\binom{6}{4} = 1111$ in binary.

To compute these bits, note the lhs can be written y^x for $y = 2^\ell + 1$. Now, y can be computed efficiently by (1), and so again by (1) one can compute y^x efficiently. To extract the bits of y^x we divide by $2^{\ell x/2}$ (implemented using the bit-shift operation >>) then take the ℓ least significant bits (implemented as bit-wise And with $2^\ell - 1$).

Exercise 1.24. Prove (3). Guideline: The main idea is that computing the *greatest common divisor* (gcd) of x and $y!$ tells us whether x has a factor $\leq y$ or not. Use that the gcd of x and y can be computed in time linear in the bit length of $\min\{x, y\}$, see Fact A.2 and recall word programs have the modular remainder operation.

From (3), the factor can also be output to memory in linear time, concluding the proof of Theorem 1.22.

1.8 The grand challenge

The grand challenge of computational complexity theory is to prove tight lower bounds on the running times required to solve "natural" computational problems on general computational models, such as word programs. As mentioned in Chapter 0, our ability to prove such impossibility results appears limited. Indeed, it is consistent with our knowledge that all such problems are in FTime(cn). To point to specific problems, we can say that it is consistent with our knowledge that FTime(cn) contains Factoring, and all the problems in chapters 3 and 4.

For these problems, essentially only trivial impossibility results are known. For example, if a function depends on all the input bits, we need time $\geq n$ to read all the input. One can optimize the constants, see Problem 1.4. But even a bound of cn is unknown.

In this section we present some impossibility results for somewhat artificial problems. We will cover a variety of techniques which will be used later also for more natural problems, at the price of some extra restrictions on the computational model. For example, these techniques will be used in Theorem 6.49, stating that any algorithm solving the natural "sat" problem requires either much time or space.

1.8.1 Undecidability and diagonalization

First, we briefly discuss *undecidable* problems: problems that cannot be solved by computers, regardless of time. The main technique for showing this is known as *diagonalization* and will feature later as well. This technique can be illustrated informally via the following riddle:

Can there be a computer that answers every question correctly with a "yes" or "no"?

We claim that such a computer cannot exist. The argument is as follows. Suppose such a computer exists and call it M. Then ask it the following question:

Does M answer "no" to *this* question?

We claim that M cannot produce an answer to this question. Indeed, if M answers "yes," then since M is correct it should have answered "no" which is a contradiction. While if M answers "no" then it is not true that M answers "no" to the question, hence M answers "yes" which is again a contradiction. Hence, there is a contradiction either way, and we conclude that the computer M does not exist.

Perhaps surprisingly at first sight, this argument can be formalized. The main idea is simply that if we have a program, we can give it as input its own description.

Theorem 1.25. [An undecidable language] There is $f : [2]^* \to [2]$ that cannot be computed by any word program, regardless of time.

Proof. The problem f takes as input a description P of a program. The output is defined as follows. If P run on its own description outputs 1, then $f(P) := 0$. Otherwise (including if P isn't a valid description, or if P does not stop), $f(P) := 1$.

Assume towards a contradiction that a program Q exists that computes f. Consider the output $Q(Q)$ of Q on its own description. If $Q(Q) = 0$ then $f(Q) = 0$, which means that Q run on its own description outputs 1, i.e., $Q(Q) = 1$. This is a contradiction. Similarly, if $Q(Q) = 1$ then $f(Q) = 1$. Since Q is a valid program that stops, this means that Q run on its own description does not output 1. This is again contradiction. Hence, Q cannot exist. **QED**

Using similar ideas, one can prove that several other problems are undecidable. For example, the generic problem of *program verification*, i.e., does this program produce this output, is undecidable.

To understand why this technique is called diagonalization consider the matrix M where the rows are programs and the columns inputs, and entry P, x of M is 1 if $P(x)$ outputs 1, 0 if it outputs 0, and is blank otherwise. Then, the function f in the proof is designed to be different from the diagonal. That is, $f(Q) \neq M_{Q,Q}$.

1.8.2 The hierarchy of Time

Can you solve more problems if you have more time? For example, can you write a program that runs in time $t'(n) = n^2$ and computes something that cannot be computed in time $t(n) = n^{1.5}$? To set the stage, we note that the answer is yes for trivial reasons if we allow for functions with long output lengths. Indeed, in time n^2 we can output a string with $\geq cn^2$ ones, but in time $n^{1.5}$ we can output at most $n^{1.5}$ ones. This shows $\text{FTime}(n^2)$ is not contained in $\text{FTime}(n^{1.5})$.

The answer is more interesting and useful if the functions are boolean. Such results are known as *time hierarchies*, and a generic technique for proving them is *diagonalization.* The argument allows us to show that even increasing running time by a constant factor yields more computational power. There is however a technicality due to the fact that for some pathological running times $t(n)$ the result is not true:

Claim 1.26. There exists a function $t(n)$ such that for every function $t'(n) \geq t(n)$ we have $\text{Time}(t'(n)) = \text{Time}(t(n))$.

Proof. Let $t(n)$ be the so-called *busy-beaver* function: the maximum over all word programs P of length $\leq n$ and over all $x \in [2]^n$ of length n of the number of steps it takes for P to stop on input x. If P does not stop on x we define this number to be 0, so that the maximum is well defined.

Now let $f \in \text{Time}(t'(n))$ and let P be a corresponding program as in Definition 1.9. Note that P on inputs of length n stops within $t(n)$ steps by definition of $t(n)$. Hence $f \in \text{Time}(t(n))$. **QED**

The time bound $t(n)$ in Claim 1.26 is strange; in particular it is hard to compute. Next we show that for time bounds that are easy to compute, including all "standard" time bounds like $n, n \log n, n^2, 2^n$, etc., the time hierarchy does hold.

Definition 1.27. A function $t : \mathbb{N} \to \mathbb{N}$ is *time-constructible* if $t(n) \geq n$ and there is a word program that does not use memory reads or writes, and starting in a configuration with register $R[0] = n$ and all other registers equal to 0, reaches a configuration where $R[1] = t(n)$ in time $\leq a \cdot t(n)$ for some constant $a \in \mathbb{N}$.

Most time bounds of interest are time constructible. We illustrate a few examples.

Claim 1.28. The following functions of n are time-constructible:

1. an for any $a \in \mathbb{N}$
2. $n \cdot |n|$.
3. n^a for any $a \in \mathbb{N}$.
4. 2^n.

Proof. 1. The program loops from 0 to $n-1$. In each iteration, it increases $R[1]$ by a, and also calls MALLOC a times. Calling MALLOC is necessary to have word size large enough to store the value an.

4. Write 1 in $R[1]$. The program loops n times; at each iteration it doubles the value in $R[1]$ using the same approach in 1. for $a = 2$. Iteration i takes $c2^i$ steps. The total number of steps is thus $\sum_{i \in [n]} c2^i \leq c2^n$, as desired. **QED**

Exercise 1.29. Prove 2. and 3.

We can now state and prove the time-hierarchy theorem that for every time-constructible function t there is a constant a depending only t, and a function in $\text{Time}(at)$ that is not in $\text{Time}(t)$.

Theorem 1.30. [Time hierarchy] $\text{Time}(c_t t) \not\subseteq \text{Time}(t)$, for any time-constructible t.

Proof. We define f by giving a program for it. The input is itself a program Q. On that input, first use time-constructibility to compute $t(|Q|)$ into a register called *counter*. Then, make two copies of Q, e.g. using the program from Example 1.8 (but adding a delimiter as required by the next step). Now run the universal machine from Lemma 1.18 to simulate Q on its own description, but modify the simulation as follows. For every step simulated, decrease the counter. If Q outputs 1 before the counter stops, output 0 instead. Otherwise, including the cases where Q is not a correct program description, or the counter reaches 0 before Q stops, output 1. (The universal machine in Lemma 1.18 works in the same way

even if started with larger word length, as may arise after computing t; alternatively we can use the technique in Claim 1.15.)

The running time of this program is

$$at(n) + cn + ct(n)$$

where the terms correspond to computing t, duplicating Q, and running the simulation. In all, the program takes time $at(n)$ for a constant a.

We now show that such an f is not in Time($t(n)$). Suppose towards a contradiction that $f \in$ Time($t(n)$) and let P be a corresponding program as in Definition 1.9. Consider running P on its own description, as in Theorem 1.25. We have $P(P) = f(P)$, but the simulation in the definition of f stops on input P, and so $f(P)$ is the complement of $P(P)$, which is a contradiction. Hence $f \notin$ Time($t(n)$). **QED**

Corollary 1.31. Exp $\not\subseteq$ P.

Proof. Let $t(n) = 2^n$, which is time constructible by Claim 1.28. Then use that P $\subseteq$ Time($2^{n/2}$) $\subsetneq$ Time(2^n) $\subseteq$ Exp, where the first and last inclusion are by definition, and the middle separation is by Theorem 1.30. **QED**

1.9 Problems

Problem 1.1. [Encoding pairs] An encoding of pairs is a 1-1 map $f : [2]^* \times [2]^* \to [2]^*$. The encoding is *efficient* if both f and the inverse of f are in FP. Describe efficient encodings with the following upper bounds on $|f(x, y)|$ (recall the notation $||$ for bit length from section §0.2):

1. $2|x| + |y|$.
2. $2\,||x|| + |x| + |y|$.
3. $2\,|||x||| + ||x|| + |x| + |y|$.

Prove that for any efficient encoding, efficient or not, and for all arbitrarily large n there are strings x, y with $|x| + |y| = n$ such that $|f(x, y)| \geq n + \log(n + 1) - 1 \geq n + |n| - c$, nearly matching the above.

Problem 1.2. Suppose there exists a such that Theorem 1.19 holds with the running time of U replaced with $(|P| \cdot t \cdot |x|)^a$. (That is, the dependence on the program description improved to power; on the other hand we allow even weaker dependence on t and $|x|$.) Prove that Factoring $\in$ FP.

Problem 1.3. Explain how to simulate multiplication between registers with $2w$ bits using registers with w bits.

Problem 1.4. Recall from Example 1.11 that Or is in Time(cn). Show that Or is not in Time($2n - c$).

1.10 Notes

It's all over.

(Reportedly said upon hearing of [128], cf [125].)

The fundamental work on complexity is [128]. That work formalized computation for the first time, and discovered its self-referential ability, essentially inventing universal machines, the diagonalization technique, and proving Theorem 1.25. (The focus of [128] was slightly different, so the theorem is not quite stated there.) Of course, [128] did not come out of nowhere, but was in fact a reaction to a program of automating mathematics, and it built on logical formalizations of mathematics; and diagonalization has its roots in (and takes the name from) the proof that the real numbers are uncountable. Also, there are several previous works aimed at formalizing computation in various branches of science. See [257] for an account of this compelling history. Still, if a fundamental work must be picked, [128] seems appropriate, for it can be considered the first work on impossibility results about general computation.

The formalization of computation in [128] is in terms of recursive functions, not unlike modern functional programming languages. Many other equivalent formalizations came about later. The model of *tape machines,* discussed in Chapter 16, was introduced in [345]. This model is closer to computer hardware or imperative programming languages, and makes it a little more intuitive how to measure time and space in computation. Historically, complexity theory was developed over such tape machines, with complexity classes defined in [161]. This paper also proves a first hierarchy result, later improved, see Chapter 16.

The fact that the hierarchy theorem does not hold for some (non time-constructible) function $t(n)$ was proved in [339, 65]. Their result is stronger than Claim 1.26 because it yields a computable $t(n)$.

Word programs with unbounded word size are from [94], where a time hierarchy is also proved. The first paper to consider word programs (with bounded word size) seems to be [118], which has: "*What seems to be needed is a computational model that avoids the potential abuses of the unit cost random access machine, but allows for unit cost operations among operands of "reasonable" size, i.e., operands of size commensurate with the sizes of the numbers to be sorted.*" Several other works have put forth variants of word programs.

The power-time computability thesis is an efficient version of the computability thesis from logic. For a proof of a formalization of the latter, and related discussion, see [152].

Theorem 1.22 is from [308]. The algorithm for the greatest common divisor is very old, see [109], but there was no bound on its efficiency until [223], which provides the bound we need. Primality was placed in P in [9].

Problem 1.2 is from [343].

A brief history of the impossible. Historically, impossibility results have been some of the most consequential. An early example is the proof that the diagonal of a square is irrational, about 2500 years ago. This can be seen as a statement about computation, where the computational model are rational numbers. The target object is "natural" or occurs "in

the wild." Another famous example is that there is no closed-form algebraic expression for polynomial equations of degree 5. In the first half of the 20th century undecidability results in logic and mathematics such as Theorem 1.25 began to appear. Complexity theory takes a more quantitative angle on impossibility. It focuses on a resource such as time, considers problems that can be solved with enough of the resource, and tries to *bound from below the amount of resource that's needed*. Because of this, impossibility results are also known as "*lower bounds*." The first such results were proved in the second half of the 20th century, and many have not been improved since. The first results on tape machines are from the 60s, and so are the first results on circuits such as [263, 254], though circuit complexity started already in [312]. (Circuits are introduced in Chapter 2.) [340] provides a regional survey of research on lower bounds. A fresh wave of mathematical ideas and results in circuit complexity came in the 80's and 90's, discussed in Chapter 7. This wave soon "hit the wall," for example the "wall" of constant-depth circuits with mod 6 gates, see section §9.6, and stalled. Still the results we do know are very consequential. For example, a lower bound for small-depth circuits from [254] is often cited as a key factor in ushering the "AI winter" of the 1970s and 80s, see the quote from [279] in Chapter 7.

Two lines of work have moved more or less parallel to developments in boolean circuit complexity. The first is *algebraic* complexity, see Chapter 14. The second is a line of works that has devised increasingly sophisticated ways to leverage uniformity and diagonalization, producing results such as [282, 113, 115, 384] (see Theorem 6.49, Theorem 6.53, and Theorem 9.54) that do not have a non-uniform counterpart.

Another active line of research starting in the 70's has proposed several "barriers" to explain the lack of progress, see Chapter 17.

The quote at the beginning of the chapter is from [380].

Chapter 2

Circuits

In this chapter we introduce *circuits*, a *non-uniform* model of computation which is close to the hardware of an actual computer. We can think of a circuit as a graph, or as a restricted program without control-flow operation, called straight-line program.

Definition 2.1. A *circuit* or *straight-line program* of size s with n input bits and m output bits is a sequence of s assignment instructions to gates g_i where assignment $i \in [s]$ is of one of the following types:

- $g_i := b$, for $b \in [2]$,
- $g_i := t \wedge t'$, where each of the terms t and t' is either a gate g_h with $h < i$ or a literal (an input bit x_i or its negation $\neg x_i$ for $i \in [n]$)
- $g_i :=t \vee t'$, where t and t' are as for $g_i := t \wedge t'$
- $g_i := \neg g_h$ where $h < i$.

The *fan-out* of a gate is the number of its occurrences in other instructions. A circuit computes a function on $[2]^n$ in the natural way, executing the instructions in order. The last m gates constitute the output.

We can also visualize a circuit as an acyclic directed graph with nodes representing gates and directed edges representing wires from the output of one gate to the input of the next.

As for word programs, the instruction set in Definition 2.1 is somewhat arbitrary and redundant. For example, we can write $g_i = 0$ as $g_i = x_j \wedge \neg x_j$, etc. Also, a single gate type, like NAnd which computes the negation of And, suffices to simulate all others (since we can write $\neg x$ as $x\mathrm{NAnd}x$). Less obviously, gates of the type $g_i = \neg g_j$ can be removed at little cost in size (Problem 2.2). Other types of gates are often useful. In fact, it is often useful to allow *any* gate on two inputs. Again, this comes at little cost in size, as we shall prove shortly in Theorem 2.5. The basis we picked is however natural in some contexts so we picked it for concreteness.

Example 2.2. Here's a circuit computing the Xor of two bits:

$$(x_0 \wedge \neg x_1) \vee (\neg x_0 \wedge x_1).$$

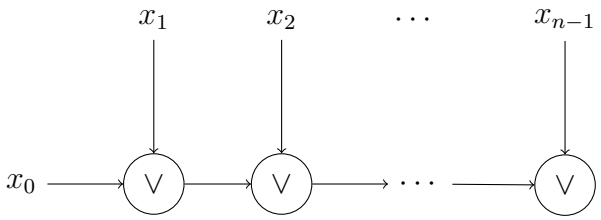
This circuit has size 3 and one output gate. The corresponding sequence of gates is:
$g_0 := x_0 \wedge \neg x_1$
$g_1 := \neg x_0 \wedge x_1$
$g_2 := g_0 \vee g_1$.

Definition 2.3. Let $g : \mathbb{N} \to \mathbb{N}$ be a function. We denote by $\text{CktSize}(g(n))$ the set of functions $f : X \subseteq [2]^* \to [2]^*$ for which there exists $n_0 \in \mathbb{N}$ s.t. for every $n \geq n_0$ there is a circuit of size $g(n)$ that computes f on every input in X of length n. We also define

$$\text{CktP} := \bigcup_d \text{CktSize}(n^d).$$

Note we only gave a definition for non-boolean functions, as the distinction between boolean and non-boolean will not be too important for circuits.

Example 2.4. The function Or on n bits, which outputs 1 if one of the bits is 1, and 0 otherwise, is in $\text{CktSize}(cn)$. A corresponding circuit can be drawn as



The same result holds for And.

Often we will consider computing functions on *small inputs*. In such cases, we can often use the following general result. In a way, the usefulness of the result goes back to the locality of computation. The result will be used extensively.

Theorem 2.5. Every function $f : [2]^n \to [2]^m$, for $n, m \geq 1$, can be computed by a circuit of size $\leq cm2^n/n$.

Proof. It suffices to consider $m = 1$. The case of larger m follows by applying the solution for $m = 1$ to each output bit of f. So let $f : [2]^n \to [2]$. We give several different proofs showcasing a wealth of ideas and yielding circuits for f of various sizes.

Size $n2^n$: Write

$$f(x) = \bigvee_{a \in [2]^n : f(x)=1} [x = a],$$

where recall $[x = a]$ is 1 if $x = a$ and 0 otherwise. In turn we can write

$$[x = a] = \left(\bigwedge_{i \in [n]: a_i = 1} x_i \right) \wedge \left(\bigwedge_{i \in [n]: a_i = 0} \neg x_i \right).$$

Using the circuit for And from Example 2.4, $[x = a]$ has circuits of size $\leq n$. Plugging these circuits in the expression for f above, and again using the circuit for Or from Example 2.4 we obtain a circuit of size $n2^n$.

Size $c2^n$:

Let f_0 and f_1 be functions on $n-1$ bits corresponding to setting the last input bit x_{n-1} of f to 0 or 1. Then we have

$$f(x) = (f_0(x_0, x_1, \ldots, x_{n-2}) \wedge \neg x_{n-1}) \vee (f_1(x_0, x_1, \ldots, x_{n-2}) \wedge x_{n-1}). \tag{2.1}$$

This shows that the maximum circuit size $S(n)$ of functions on n bits satisfies

$$S(n) \leq 2S(n-1) + 3,$$

also, $S(0) = 1$ as a function on 0 bits is just a constant. Opening up the recursion we have $S(n) \leq 2^n + 32^{n-1} + 32^{n-2} + \ldots + 3 \leq c \cdot 2^n$ (see Fact A.4).

Size $c2^n/n$: Perform the recursion in equation (2.1) until we have functions on k bits, for a k to be determined. This gives a "circuit" C' for f of size $c2^{n-k}$ where some of the gates compute functions on k bits corresponding to fixings of the last $n-k$ input bits of f. The savings in size comes from this cool observation: While there are 2^{n-k} gates computing functions on k bits, for a small k there are much fewer functions on k bits, so many of these gates actually computed the same function. So we can re-use the same circuit for many gates, leading to a better bound on the circuit size than we would get by continuing the recursion all the way to the base case. Here we critically use that the circuit model allows large fan-out, which is also cool (in other models this can't be done).

Specifically, there are 2^{2^k} functions on k bits. By an approach above we can compute each of them by a circuit of size $c2^k$. So overall all these functions can be computed by a circuit of size $c2^{2^k+k}$. Combining this circuit with C', we see that for any k there is a circuit for f of size

$$\leq c\left(2^{n-k} + 2^{2^k+k}\right).$$

To minimize this quantity we pick k that balances the two summands. In particular, for $k = \lceil \log n \rceil - c$ and $n \geq c$, the quantity is

$$\leq c\left(2^{n-\lceil \log n \rceil + c} + 2^{n/4 + \log n}\right) \leq c\left(2^n/n + 2^{n/2}\right) \leq c2^n/n.$$

QED

Exercise 2.6. [Simple indexing circuit] The Indexing problem: Given $x \in [2]^n$ and $i \in [2]^{\log n}$, where n is a power of 2, compute bit i of x. Prove that Indexing is in CktSize($cn \log n$). Use this to give another proof of the $cn2^n$ bound in Theorem 2.5.

Exercise 2.7. Prove that the sum of two n-bit integers is in CktSize(cn). Hint: Consider a circuit for the sum of three bits $f : [2]^3 \to [2]^2$, $f(x, y, z) := x + y + z$. Apply this result repeatedly.

2.1 The grand challenge for circuits

With regard to the the grand challenge, the situation for circuits is analogous to that for word programs. Even a bound of cn is unknown, e.g. for the problems mentioned in section §1.8, and we can only prove modest-looking bounds for smaller constants. The arguments are a little more complicated and useful for circuits. For example, Or has circuits of size $\leq n$ as we saw in Example 2.4, whereas it requires time $\geq 2n - c$ (Problem 1.4). So proving a circuit-size bound $\geq (1 + c)n$ requires a different function. We illustrate these techniques with a simple example; a line of works has optimized constants with increasingly complicated case-analyses, see the notes.

Theorem 2.8. Let $\text{Thr}_2 : [2]^n \to [2]$ output 1 iff at least two input bits are 1. Then $\text{Thr}_2 \notin \text{CktSize}(2n - c)$.

The proof uses the far-reaching *restrict-and-simplfy method.* A *restriction* is just a fixing of some input variables to constants. The method shows that *after applying a suitable restriction the circuit simplifies.* One then iterates the argument – fixing more input variables and simplifying more the circuit – until the circuit trivializes. The important point is that the circuit trivializes *faster* than the target function; in the end, the restricted circuit has become say a constant, but the restricted target function is not, concluding the proof. In this first, basic instantiation of the method, both the restriction and the simplification are modest: We show that we can fix *one* input variable and remove *two* gates. Albeit modest, this simplification is not trivial. This method will be studied in Chapter 7 for more specialized types of circuits exhibiting dramatic simplifications.

Proof. We prove it for the stronger circuit model where we allow as gates *any* functions on two bits. To apply the restrict-and-simplify method, suppose there was a circuit of size $\leq 2n - c$ computing Thr_2 on $n \geq 2$ bits. Let g be a gate connected to two literals ℓ_i, ℓ_j of distinct variables $x_i \neq x_j$. Such a gate exists when $n \geq 2$, for else the output is either a constant or a literal, which isn't true. We claim that either x_i or x_j appears in another gate. This is because ranging over the 4 possible fixings $x_i = b_i \in [2]$ and $x_j = b_j \in [2]$, the function Thr_2 ranges over 3 different functions of the remaining $n - 2$ variables (depending on the value $b_i + b_j \in [3]$), for $n \geq 2$. On the other hand, if both x_i and x_j were connected only to g, there would be only 2 possible resulting circuits, because g only takes two values. Hence, fixing either $x_i = 0$ or $x_j = 0$ allows us to remove two gates, while still computing Thr_2 on $n - 1$ bits. Any gate that took as input one of those two removed gates is changed to compute the resulting function. The exception is when we are removing the output gate, in which case we simplify the circuit to a single gate. We iterate this argument $\leq n-2$ times until the restricted circuit is a single gate. Since there are still ≥ 2 input variables unfixed, this is not possible. **QED**

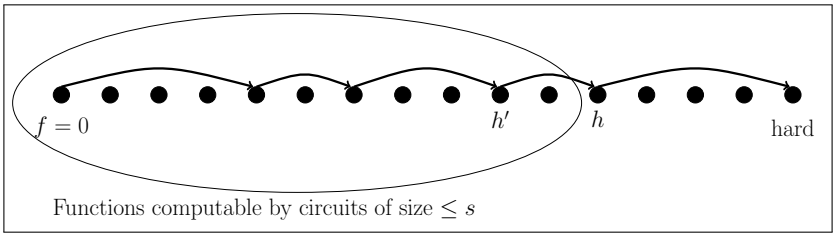


Figure 2.1: Illustration of the proof of Theorem 2.11.

As we said, we don't know how to prove that natural functions require large circuits. However, we can prove the *existence* of functions requiring large circuits, using counting arguments. (Counting arguments can be used for word programs as well, but Theorem 1.25 gave a more concrete problem.)

Theorem 2.9. There are functions $f : [2]^n \to [2]$ that require circuits of size $\geq c2^n/n$, for every n.

This bound matches Theorem 2.5.

Proof. First we claim that a circuit $C : [2]^n \to [2]$ of size $s \geq n$ can be described by $cs \log s$ bits. On the other hand there are 2^{2^n} functions on n bits. Since each circuit computes at most one function, if every function is computable by a circuit of size s we have

$$cs \log s \geq 2^n.$$

This is not possible if $s \leq c2^n/n$, as the lhs is $\leq c(2^n/n) \cdot n < 2^n$. **QED**

Exercise 2.10. Prove the claim.

We now turn to proving the analogue of the Time Hierarchy (Theorem 1.30) in the circuit model. Specifically, we'd like to show that increasing the size of circuits allows us to compute more functions. One can prove such a result by combining Theorem 2.9 and Theorem 2.5. But a stronger and more enjoyable argument exists.

Theorem 2.11. [Hierarchy for circuit size] For every n and $s \leq c2^n/n$ there is a function $f : [2]^n \to [2]$ that can be computed by circuits of size $s + n + c$ but not by circuits of size s.

Proof. Refer to figure 2.1. Consider a path from the all-zero function $f = 0$ to a hard function which requires circuits of size $\geq s$, guaranteed to exist by Theorem 2.9, changing the output of the function on one input at each step of the path. Let h be the first function that requires size $> s$, and let h' be the function right before that in the path. (Note $h \neq f$, since f has circuit size 1.) By construction, h' has circuits of size $\leq s$, and h can be computed from h' by changing the value on a single input. The latter can be implemented by circuits of size $n + c$. **QED**

Compared to the Time Hierarchy Theorem 1.30, the above circuit hierarchy is finer. The gap in the latter is of the order of the input length, whereas in the former it is of the order of the time bound. In fact, a surprising, much finer hierarchy for circuit size exists, where the gap is *just one gate*, see Problem 2.3.

2.2 Circuits vs. Time

In this section we explore the relationship between circuit and time classes. First, we show that circuits can simulate word programs. It isn't quite true that FP $\subseteq$ CktP, since a function in FP can have different output lengths for inputs of the same input length, whereas a circuit has a fixed number of output bit. But the containment is true for boolean functions (or more generally for functions whose output length depends only on the input length).

Theorem 2.12. P $\subseteq$ CktP.

Proof. Let $f \in \text{Time}(n^a)$ for a natural $a \geq 1$, let P be a corresponding word program of length ℓ, and let $n \geq c_{a,\ell}$. Because P runs in time $\leq n^a$, it uses $\leq n + n^a \leq 2n^a$ memory words, and the word length is $\leq c_a \log n$. Therefore, we can represent a configuration of P using $\leq c_{a,P} n^a \log n \leq n^{c_a}$ bits.

It suffices to build a circuit S that given as input a configuration $C = (i, m, w, R, M)$ computes the configuration C' that C yields. Indeed, we can then stack together n^a copies of such a circuit, and output the first bit of $M[0]$ in the last configuration. To build S we first construct a *memory-read circuit* that given a configuration and a pointer i to a memory word, computes $M[i]$. This can be done with size n^{c_a} using $c_a \log n$ copies of the indexing circuit from Exercise 2.6, one for each bit of $M[i]$.

Returning to the construction of S, we first apply ℓ such memory-read circuits to fetch the ℓ memory words $M[R[i]]$, $i \in [\ell]$ indexed by the ℓ registers. At this point, each bit of the next configuration C' can be computed from the fetched values and the information in C *excluding* the memory M. Overall, each bit of C' is a function of just $\ell c_a \log n$ bits. By Theorem 2.5, this bit of C' can be computed by a circuit of size $n^{c_{\ell,a}}$. And so all of C' can be computed by in size $n^{c_{\ell,a}}$. **QED**

When starting with a function in Time(n^a), this argument produces circuits of size n^b for a somewhat large b. This is mainly due to the use of the brute-force circuit from Theorem 2.5. On the other hand, we used no knowledge of the specific instructions of the word program; the same proof applies to any instructions as long as they are local. By contrast, one can argue that the specific instructions from Definition 1.1 have efficient circuit implementations and optimize the argument to obtain b about $2a$. But $b < 2a$ is open.

It is not known if Exp $\subseteq$ CktP. It is widely conjectured that Exp is not in CktP, and in fact that Exp requires circuits of exponential size. But all that we can prove is:

Theorem 2.13. For every k, Exp $\not\subseteq$ CktSize(n^k).

Proof. On an input x of length n, the function f is defined to be a function h applied to only the first $m := (k+1)\lceil \log n \rceil$ input bits. By Theorem 2.9, there is a such function h on m bits that requires circuits of size $\geq c2^m/m > n^k$ for $n \geq c_k$. In exponential time we can enumerate over all 2^{2^m} functions h on m bits. For each such function, we also enumerate over all circuits of size $\leq n^k$ to determine if it can be computed by such a circuit. If it can't, we have found h and hence f. Otherwise, we try the next h. Recall that, as in the proof of Theorem 2.9, a circuit of size n^k can be described with $\leq ckn^k \log n \leq n^{k+1}$ bits, for $n \geq c_k$. So the the entire enumeration is feasible in exponential time.

Note we have found f without even looking at the input x, only its length. Finally, we output $f(x)$. **QED**

2.3 Cosmological, non-asymptotic impossibility

Most of the results in this book are *asymptotic*, i.e., they only apply to sufficiently large inputs. As stated, these results don't say anything for inputs of a fixed length. For example, any function on 10^{100} bits (or any other constant-size set) is in Time(0), according to Definition 1.7. (The time bound only needs to apply to large enough inputs.)

However, it is important to note that all the proofs are *constructive* and one can work out non-asymptotic results. We state next one representative example when this was done. It is a problem in logic, an area which often produces very hard problems. On an alphabet of size 63, we consider formulas with first-order variables $x, y, \ldots$ that range over $\mathbb{N}$, second-order variables $S, T, \ldots$ that range over finite subsets of $\mathbb{N}$, the predicates "$y = x+1$" and "$x \in S$," and standard quantifiers, connectives, constants, and order relations, and set equality. For example one can write things like "every finite set has a maximum:" $\forall S \exists x \in S \forall y \in S, y \leq x$.

Deciding the truth of such formulas is extremely hard:

Theorem 2.14. Any circuit deciding the truth of logical formulas of length at most 610 in the language above requires $\geq 10^{125}$ gates.

So even if each gate were the size of a proton, the circuit would not fit in the known universe. The proof, referenced in the notes, is a variant of the techniques we saw in this chapter and Chapter 1.

2.4 Problems

Problem 2.1. Prove that Indexing (as defined in Exercise 2.6) is in CktSize(cn).

Problem 2.2. [Pushing negations to the input] Show that for any circuit $C : [2]^n \to [2]$ of size s there is an equivalent circuit C' of size $2s$ without Not gates $g_i := \neg g_j$.

Problem 2.3. [Finer hierarchy for circuit size]. Prove that for all n and $s \leq c2^n/n$, there is a function $f : [2]^n \to [2]$ that can be computed by circuits of size $s+1$ but not by circuits of size s.

2.5 Notes

The existence of hard functions for circuits via counting arguments, Theorem 2.9, goes back to [312], Theorem 7, "*Are most functions simple or complex?*" So does the brute-force computation of functions via circuits. The bound in Theorem 2.5 is from [239]. Further works have optimized the constants.

Theorem 2.14 is from [325], see the reference for the history of the result.

A number of papers have proved circuit lower bounds of the form an for various constants a, for functions in P. Over the full basis, i.e., when the computation gates can compute arbitrary functions (the model of Theorem 2.8), see [229]; over the basis $\neg, \vee, \wedge$, see [195] for slightly better constants.

The fine hierarchy for circuit size, Problem 2.3, is from [342].

Chapter 3

Randomness

Alea iacta est

Today, there's a significant challenge to the computability thesis. This challenge comes from... I know what you are thinking: *Quantum computing, superposition, factoring.* Nope. *Randomness.*

The last century or so has seen an explosion of randomness affecting much of science, and computing has been a leader in the revolution. Today, randomness permeates computation. Except for basic "core" tasks, using randomness in algorithms is standard, and considered "feasible." So let us augment our model with randomness.

Definition 3.1. A *randomized word program* is a word program with the extra instruction

- $R[i] :=$ Rand, where $i \in [\ell]$ and the ℓ is the number of registers of the program.

This instruction sets $R[i]$ to a uniform value in $[2]^w$ independent of all previous values, where w is the current word length.

We denote by *BPTime*$(t(n))$ *with error* $\epsilon(n)$ the set of functions f that map a subset $X \subseteq [2]^*$ to $[2]$ for which there exists a randomized word program P that, on any input $x \in X$ of length $\geq |P|$, stops within $t(|x|)$ steps and has

$$\mathbb{P}[P(x) = f(x)] \geq 1 - \epsilon(|x|),$$

where $\mathbb{P}$ denotes probability, taken over the values given by Rand.

The set FBPTime with error $\epsilon(n)$ is defined in the same way except that f outputs subsets and the condition is $\mathbb{P}[P(x) \in f(x)] \geq 1 - \epsilon(|x|)$.

If the error ϵ is not specified then it is assumed to be $1/3$. We also define

$$\text{BPP} := \bigcup_a \text{BPTime}(n^a),$$
$$\text{FBPP} := \bigcup_a \text{FBPTime}(n^a),$$

where BP stands for *bounded (error) probability.*

Exercise 3.2. Does the following algorithm show that deciding if a given integer x is prime is in BPP? "Pick a uniform integer $y \in [2..x-1]$. If y divides x output NOT PRIME, else output PRIME."

The introduction of randomness in our model raises several fascinating questions. Does randomness allow us to solve problems faster? Is BPP = P? Does randomness exist "in nature?" Do we need "perfect" randomness for computation? We begin to explore these questions in this chapter, and will return to them later.

First, in Definition 3.1 we have assumed that we have "perfect" randomness, i.e., that Rand returns a *uniform* value. "Imperfect" sources of randomness could be a better model for the randomness sources available "in nature." Problem 3.2 asks you to prove that a simple imperfect source, where the bits are biased, suffices for BPP. A large body of research has been devoted to greatly generalize this setting to pinpoint the imperfect sources of randomness that suffice for simulating BPP.

Nest, we begin by investigating the error bound in the definition of BPTime. This bound is somewhat arbitrary, because it can be reduced.

3.1 Error reduction for one-sided algorithm

We first discuss how to reduce the error in the important special case of *one-sided errors*. Suppose we have a randomized algorithm for a boolean f with the extra condition that it never makes mistakes when $f(x) = 0$. In other words:

- if $f(x) = 0$ then $\mathbb{P}[P(x) = 0] = 1$, while
- if $f(x) = 1$ then $\mathbb{P}[P(x) = 1] \geq 1 - \epsilon(|x|)$.

The corresponding classes are called RTime and RP.

The error probability of such a one-sided algorithm can be reduced by running it r times independently, and taking the Or of the outcomes. In the case $f(x) = 0$, all r runs will give 0, and so the Or will be 0. In the case that $f(x) = 1$, the prob. of a mistake is the prob. that all the r runs give 0. Because the runs are independent, this equals

$$\mathbb{P}[P(x) = 0]^r \leq \epsilon(|x|)^r.$$

For example, if ϵ is a constant, the probability decays exponentially fast with r.

In this one-sided case we can in fact even allow the error parameter ϵ to be very close to 1: If $\epsilon = 1 - p$ then the error bound is (by Fact A.5)

$$(1-p)^r \leq e^{-pr}.$$

In particular, we can start with error prob. ϵ as large as $1 - 1/n^a$ and still drive it to 2^{-n^b}, for any constants a and b, at the price of running the algorithm $r = n^{a+b}$ times.

3.2 Error reduction for BPTime

We now discuss error-reduction for (two-sided) BPP algorithms. We have the following result.

Theorem 3.3. [Error reduction for BPP] Definition 3.1 of BPP remains the same if the error $1/3$ is replaced with $1/2 - 1/n^a$ or $1/2^{n^a}$, for any constant a.

The idea in the proof is natural: You repeat the algorithm r times for a large r, but instead of computing Or as in section §3.1, you take a majority vote. The analysis of this is slightly more involved than for the case of one-sided algorithms. The math fact that powers the proof is this:

$$(1-\epsilon)(1+\epsilon) = 1-\epsilon^2. \tag{3.1}$$

It's a non-trivial fact: We are multiplying a quantity less than 1 by another which is bigger than 1, and the product is less than 1. In the context of randomized algorithms, this will allow us to show that making $r/2$ mistakes is unlikely.

Proof of Theorem 3.3. Suppose that f is in BPP with error $\epsilon \leq 1/2 - 1/n^a$ and let P be a corresponding randomized word program. On an input x, let $1/2 - p$ be the error probability on x, where $p \geq 1/n^a$. Let us run P for $r := 8n^{2a} \cdot n^b$ times, each time with fresh randomness, and take a majority vote. This new algorithm makes a mistake only if at least $r/2$ runs of P make a mistake. (We consider a tie a mistake, so it doesn't matter how majority is defined for ties.)

To bound this error probability, consider a sequence $s \in [2]^r$ of possible outcomes, where $s_i = 1$ means that run i of the algorithm made a mistake, and denote by $w(s)$ the total number of mistakes (i.e., the number of ones in s, or the weight). Let p_s be the probability of the sequence of outcomes, and note

$$p_s = (1/2-p)^{w(s)}(1/2+p)^{r-w(s)}.$$

Our goal is to bound

$$\sum_{s \in [2]^r : w(s) \geq r/2} p_s.$$

Each term p_s in the sum is $\leq (1/2-p)^{r/2}(1/2+p)^{r/2}$, because $w(s) \geq r/2$ and $p > 0$. The total number of possible outcomes is 2^r. So the above sum is

$$\leq 2^r(1/2-p)^{r/2}(1/2+p)^{r/2} = (1-2p)^{r/2}(1+2p)^{r/2} = (1-4p^2)^{r/2} \leq e^{-4p^2 r/2} \leq 2^{-n^b},$$

as desired. The second equality in the derivation is equation (3.1) and then we use Fact A.5 and our choice of r. **QED**

The probabilistic analysis in the proof is a special case of deviation bounds for the sum of random variables, a far-reaching topic which we discuss in section §A.5.1. The analysis above is more self-contained and elementary.

3.3 The power of randomness

In this section we explore various computing paradigms that are enabled by randomness and that allow us to solve problems faster than the best known deterministic algorithm. The techniques we see will be used many times later. We present two problems, both involving checking identities over various domains, a setting where the power of randomness really shines.

3.3.1 Verifying matrix multiplication

We denote by $\mathbb{F}_2$ the set [2] equipped with addition and multiplication modulo 2. This is the finite field $\mathbb{F}_2$; finite fields are discussed more in section §A.9.

Definition 3.4. The MatrixMultiplicationVerification (MMV) problem: Given 3 square matrices A, B, C over $\mathbb{F}_2$, is $AB = C$?

This problem can be solved (deterministically) simply by performing the matrix multiplication AB and checking equality with C. But the fastest known algorithm to multiply two $d \times d$ matrices runs in time $\geq d^{2+c}$, leading to a running time $\geq n^{1+c}$ for MMV. By contrast, a simple randomized algorithm runs in linear time.

Theorem 3.5. MMV $\in$ BPTime(cn).

The proof uses a fact which is as simple as it is useful:

Fact 3.6. [*Random subset* or *random parity principle*] Suppose $x \in [2]^n$ is non-zero. Then the parity of a uniformly-selected subset of x is a uniform bit. Equivalently, for any non-zero $x \in \mathbb{F}_2^n$, if $A \in \mathbb{F}_2^n$ is uniform then the *inner product* $\sum_i A_i x_i$ is uniform in $\mathbb{F}_2$.

Proof of Theorem 3.5. Let $d = c\sqrt{n}$ be the dimension of the matrices. Pick U uniformly in $\mathbb{F}_2^d$. Compute the matrix-vector products $(AB)U$ and CU in time cn; and check if they are equal. To compute $(AB)U$ exploit the associativity rule $(AB)U = A(BU)$: compute BU first, then multiply by A.

If $AB = C$ the check always passes. In case $AB \neq C$, one of the rows is different. Say $(AB)_i \neq C_i$, where M_i is row i of M. The check only passes if $(AB)_i U = C_i U \iff ((AB)_i - C_i)U$. By the random parity Fact 3.6, the check passes with prob. $\leq 1/2$. We can reduce the error as in section §3.1. **QED**

3.3.2 Checking if a circuit represents zero

We now present a deceptively simple problem which is in BPP but is not known to be in P. This problem asks if a given integer is zero; the twist is that the circuit is not given explicitly, but succinctly, via a circuit.

Definition 3.7. An *integer circuit* of size s is a sequence of s instructions (or gates) $g_0, g_1, \ldots$ of the following types:

- $g_i := b$, with $b \in \{-1, 1\}$
- $g_i := g_j + g_k$, where $j, k < i$
- $g_i := g_j \times g_k$, where $j, k < i$.

All operations are over $\mathbb{Z}$. The circuit represents the integer computed by the last instruction.

Note an integer circuit has no inputs. It does not compute a function but represents a single integer.

Definition 3.8. The *Zero-Integer-Circuit* (ZIC) problem: Given an integer circuit, decide if it represents 0.

It is not known if ZIC $\in$ P. One cannot simply evaluate the circuit, since the integers computed at the gates can be doubly exponential in n, and thus require an exponential number of bits to represent in binary.

Exercise 3.9. Give an integer circuit with s gates representing an integer $\geq 2^{2^{s-c}}$.

Nevertheless, we have:

Theorem 3.10. ZIC $\in$ BPP.

The technique in the proof is to pick a random prime number p in a suitable range, compute the circuit modulo p, and check if the output is 0. In other words, we are computing in the finite field $\mathbb{F}_p$ consisting of the set of integers $[p]$ with operations modulo p, see section §A.9 for more on fields. Because we are working modulo p, the integers represented by the gates stay small and we can compute them in FP. If the original circuit represents 0 then it will also represent 0 modulo any prime. Less obviously, if for many primes p the circuit represents 0 modulo p then in fact it also represents 0 as an integer. This technique is called *fingerprinting* or *modular hashing*.

For the analysis we need to bound the number of primes in a specific interval. The so-called *Prime Number Theorem* shows that $\pi(t)$ approaches $t/\log_e t$. The following Lemma 3.11 weaker bound has a significantly simpler proof, a gem of number theory, given in section §A.2.

Lemma 3.11. [Weak prime number theorem] The number $\pi(t)$ of primes in $[t]$ is $\geq ct/\log t$, for $t \geq c$.

We also need the following bound on the maximum integer represented by a circuit.

Claim 3.12. Prove that an integer circuit with s gates represents an integer whose absolute value is $\leq 2^{2^s}$.

Proof. By induction on s. For $s = 0$ the circuit consists of a constant only, whose absolute value is $1 \leq 2^1$. For the inductive step, if the output gate is $\times$, by induction its children represent integers $\leq 2^{2^{s-1}}$ in absolute value, so the output represents an integer $\leq 2^{2^{s-1}} \cdot 2^{2^{s-1}} = 2^{2 \cdot 2^{s-1}} = 2^{2^s}$ in absolute value. Ditto for the + gate. **QED**

We can now solve ZIC in BPP.

Proof of Theorem 3.10.. The algorithm picks a uniform prime p less than 2^{2n}, using the process explained below, then evaluates the circuit modulo p, and if the latter is 0 it outputs that the represented integer is zero; otherwise it outputs it is non-zero.

If the circuit represents 0 the algorithm is always correct.

If the circuit does not represent 0, then by Claim 3.12 it represents an integer in absolute value $\leq 2^{2^n}$. Such a value can be divisible by at most 2^n primes, since each prime is ≥ 2. Since by Lemma 3.11 the number of primes in the chosen range is $\geq 2^{2n}/n^c$, the probability of selecting a p that divides y will be $\leq 2^n \cdot n^c/2^{2n}$.

There only remains to pick a uniform prime. This can be done in FBPP as follows. Consider picking a uniform integer at most 2^{2n}. By the bound on the number of primes, Lemma 3.11, we have at least a $1/n^c$ chance of getting a prime, and if we do get a prime it will be a uniform prime by construction. We can test if a number is prime, and if it is not we can repeat. By the same analysis in section §3.1, if we try n^c times the chance of never getting a prime would be

$$(1 - 1/n^c)^{n^c} \leq 2^{-n}.$$

Summing the two error bounds above, we obtain that the overall error probability is (much) less than 1/3. **QED**

3.4 Does randomness really buy time?

In this section we discuss the relationships between Time and BPTime. We have the following basic inclusions.

Theorem 3.13. $\mathrm{P} \subseteq \mathrm{BPP} \subseteq \mathrm{Exp}$.

Proof. The first inclusion is by definition. The idea for the second inclusion is to brute-force the random choices in exponential time. A randomized word program M running in time n^a uses words of $\leq c_a \log n$ bits. Each Rand operation gives a uniform word, so the program uses $\leq n^a \cdot c_a \log n \leq n^{c_a}$ random bits. We can enumerate over all $2^{n^{c_a}}$ choices for these bits, run M for each of them, and output the majority of the outcomes. This takes time $2^{n^{c_a}}$. **QED**

Exercise 3.14. Generalize Theorem 3.13 by proving $\mathrm{Time}(t) \subseteq \mathrm{BPTime}(t) \subseteq \mathrm{Time}(c^{t \log t})$, for any function $t = t(n)$.

Now, two surprises. First, BPP $\subseteq$ Exp is the fastest deterministic simulation we can *prove*. On the other hand, what may come as a bigger surprise, despite the examples in section §3.3 it is widely believed that in fact P = BPP! And not only that, it is even believed that the overhead to simulate randomized computation deterministically is very small. The gap between our ability and common belief is abysmal.

One of the exciting developments of complexity theory has been the connection between the P $=^?$ BPP question and the "grand challenge" from section §1.8. At a high level, it has been shown that explicit functions that are hard for circuits can be used to *de-randomize* computation. In a nutshell, the idea is that if a function is hard to compute then its output is "random," so can be used instead of randomness. Quantitatively, the harder the function the more randomness it provides. At one extreme, we have the following striking connection:

Theorem 3.15. Suppose for some $a > 0$ there is a function in Time(2^{an}) which on inputs of length n cannot be computed by circuits with $2^{n/a}$ gates, for all large enough n. Then P = BPP.

In other words, either randomness is useless for power-time computation, or else circuits can speed up exponential-time uniform computation! We will prove this in Chapter 11.

While we don't know if P = BPP, we can prove that, like P, BPP has power-size circuits.

Theorem 3.16. BPP $\subseteq$ CktP.

The proof is a nice application of the *probabilistic method,* where a mathematical object with a certain property is proved to exist by showing that a random object satisfies the property with non-zero probability. In this application, the object is an outcome for the randomness of the randomized word program, and the property is that it should work for *all* the inputs of a certain length. Once we have such an outcome, we can hardwire it in the circuit.

To show that a random object satisfies the property, we drive the error to exponentially small by Theorem 3.3. Once the error is smaller than the number 2^n of inputs we are considering, we can afford a *union bound*. This technique of combining tail and union bounds is often used in the probabilistic method.

To make the proof more transparent it is convenient to adopt the following notation. For a randomized program P we write $P(x, R)$ for the execution of P on input x where $R = (R_0, R_1, R_2, \ldots)$ are the values given by Rand. The j-th instruction $R[i] :=$ Rand is replaced with $R[i] := R_j$, truncated to the current word length. Recall that the word length is dynamic; for simplicity we can assume that each R_i is longer than the maximum word length.

Proof. Let $f : X \subseteq [2]^* \to [2]$ be in BPP. By Theorem 3.3 we can assume that the error is $\epsilon < 2^{-n}$, and let P be a corresponding program. Note

$$\mathbb{P}_R\left[\exists x \in [2]^n : P(x, R) \neq f(x)\right] \leq \sum_{x \in [2]^n} \mathbb{P}_R[P(x, R) \neq f(x)] \leq 2^n \cdot \epsilon < 1,$$

where the first inequality is a union bound.

Therefore, there is a fixed choice for R that gives the correct answer for every input $x \in [2]^n$. This choice can be hardwired in the circuit, and the rest of the computation can be written as a circuit by Theorem 2.12. **QED**

3.5 The hierarchy of BPTime

In this section we prove the analogous of the Time Hierarchy Theorem 1.30 for BPTime.

Theorem 3.17. $\mathrm{BPTime}(c_t t(n+1)) \not\subseteq \mathrm{BPTime}(t(n))$, for any time-constructible non-decreasing t.

The proof uses a variant of the diagonalization technique which is known as *delayed diagonalization.* To explain how it fits in, recall that one of the issues that arises in the proof of Theorem 1.30 is that the input program may not run in a bounded amount of time. The solution is to use a *clocked simulation* of the program: We simulate the program, but if it runs for too long, we stop it. A more serious complication arises for randomized programs: The program may have acceptance probability very close to 1/2, whereas for BPTime we want probability $\geq 2/3$ (or bounded away from 1/2). No clocking mechanism works to fix this issue. Instead, we are going to compute the acceptance probability in *brute-force*, similarly to the proof of Theorem 3.13, and then "round it." This takes exponential time, so we can only afford it on very small inputs. This is where delayed diagonalization kicks in. It allows us to spread the effort over many different inputs, so that the expensive brute-force step becomes feasible.

Proof. We shall pad programs to increase the length of their description (this should not be confused with the padding mentioned in 1.10, which serves a different purpose). We write P_i for a program P which is padded with i extra bits (as we did, say, in the proof of Theorem 1.19).

We define the following set of inputs X and function $f : X \to [2]$. The set X is a subset of padded programs P_i. The definition depends on (1) how i compares with $L(|P|)$, where $L(k) := 2^{t^c(k)}$, and (2) whether P is *bounded on an input x,* which means it runs in $\leq t(|x|)$ steps on x, and moreover $\mathbb{P}[P(x) = b] \geq 2/3$ for some $b \in [2]$.

If $i < L(|P|)$ then $P_i \in X$ if $P(P_{i+1})$ is bounded, and we define $f(P_i) := b$ if $\mathbb{P}[P(P_{i+1}) = b] \geq 2/3$.

If $i = L(|P|)$ then $P_i \in X$ always and $f(P_i)$ is intuitively defined as $\neg P(P)$. More precisely, if P on input P accepts with prob. $\geq 2/3$ within $t(|P|)$ steps, then $f(P_i) := 0$. In all other cases, including if P is not bounded on P, $f(P_i) := 1$.

If $i > L(|P|)$ then $P_i \notin X$.

First we claim that $f \in \mathrm{BPTime}(c_t t(n))$. On input P_i of length n we first decide if $i = L(|P|)$. For this we compute $\lfloor \log^c i \rfloor$; this can be done in linear time . Then we use time-constructibility of t to compare this value with $t(|P|)$. Computing $t(|P|)$ takes time $\leq c_t t(|P|) \leq c_t t(n+1)$ because t is non-decreasing.

If $i < L(|P|)$ we run P on P_{i+1} and output its answer. This uses the universal program Lemma 1.18, and we don't even need to clock the simulation, since P is bounded on P_{i+1}. This takes $ct(|P_{i+1}|) = ct(n+1)$ steps.

If $i = L(|P|)$ we run P on P for $t(|P|)$ steps, for any possible choice of the coin tosses. This takes $\leq L(|P|)$ steps, which is less than n, and hence less than $t(n+1)$.

Next we claim that $f \notin \text{BPTime}(t(n))$. Suppose otherwise and let P be a corresponding program. Now consider the behavior of P on inputs $P_0 = P, P_1, P_2, \ldots, P_{L(|P|)}$. There are a couple of cases. First suppose P is bounded on all these P_i. Then all the P_i are in X we have the inequalities

$$\begin{array}{ccccccc} P(P_0) & & P(P_1) & & & & P(P_{L(|P|)}) \\ \| & \not\parallel & \| & \not\parallel & \ldots & \not\parallel & \| \\ f(P_0) & & f(P_1) & & & & f(P_{L(|P|)}) \end{array} .$$

These hold by correctness of P and the definition of f. However, also by definition of f we have $f(P_{L(|P|)}) \neq P(P_0)$, which is a contradiction.

Otherwise, let i be the largest s.t. P is not bounded on P_i. It cannot be that $i = L(|P|)$, because $P_{L(|P|)} \in X$ by definition and so P would not correctly compute f. In the other case, P is bounded on P_{i+1}. But then $P_i \in X$ by definition, and again this means P is not computing f on P_i correctly. **QED**

Whereas Theorem 1.30 gave a separation for total functions, here we only get a separation for partial functions; it is an open problem to prove a time hierarchy for total functions.

3.6 Problems

Problem 3.1. Prove $\text{BPTime}(10) \not\subseteq \text{Time}(n/10)$. Hint: Recall we allow partial functions.

Problem 3.2. Consider *biased* randomized word programs where the operation Rand returns one bit which, independently from all previous calls to Rand, is 1 with probability 1/3 and 0 with probability 2/3. Show that BPP does not change for sucbiased word programs.

Problem 3.3. [ZPP] Show that the following three conditions for $f : X \subseteq [2]^* \to [2]$ define the same class of boolean functions, called ZPP for *zero (error) probability*:

1. There exists a randomized algorithm that always computes f correctly and whose *expected* running time is n^a for some $a \in \mathbb{N}$ and large enough n.

2. There exists a randomized algorithm that on every input x outputs either $f(x)$ or '?' in time n^a for some $a \in \mathbb{N}$ and large enough n, and the probability of outputting '?' is $\leq 1/2$.

3. There exists a randomized *one-sided* algorithm for f and also a randomized one-sided algorithm for $1 - f$, both of which run in time n^a for some $a \in \mathbb{N}$ and large enough n. Here "one-sided" is as in section §3.1.

Problem 3.4. For a circuit C on n bits denote by p_C the probability $\mathbb{P}_x[C(x) = 1]$.

1. Show that the CAP (Circuit Acceptance Probability) problem is in FBPP: Given a circuit C and $\epsilon > 0$ written in unary (for example, as a string of $1/\epsilon$ ones) compute p s.t. $|p - p_C| \leq \epsilon$. Hint: Use Lemma A.15.

2. Show that the following decision version of CAP is in BPP: Given a circuit C, a number p (written in binary), and $\epsilon > 0$ written in unary, such that $|p_C - p| \geq \epsilon$, decide if $p_C \geq p$.

Problem 3.5. Assume P = BPP. Prove (see Problem 3.4 for the definition of CAP and p_C):

1. CAP is in FP.

2. Given a circuit C and ϵ written in unary s.t. $p_C \geq \epsilon$ we can compute $x : C(x) = 1$ in FP.

3. Given n in unary we can compute an n-bit prime in FP.

Problem 3.6. [Existence of good error-correcting codes] A *binary error-correcting code* with block length n, message length k, and minimum distance d, called an $(n, k, d)_2$ code for short, is a subset $C \subseteq [2]^n$ of size 2^k s.t. for any distinct $x, y \in C$, x and y differ in $\geq d$ coordinates. Use the probabilistic method as in the proof of Theorem 3.16 to show the existence of $(n, \epsilon n, \epsilon n)_2$ codes, for a constant $\epsilon > 0$ and every n. Such (families of) codes are called *good*.

3.7 Notes

Randomized tape machines were studied already in [101] (cf section §16.6). Randomized complexity classes, including BPP, originate in [184]. For more on deviation bounds see section §A.5.1 and section §A.12.

The simulation of BPP by circuits, Theorem 3.16, is from [7].

The randomized algorithm for verifying matrix multiplication, Theorem 3.5, is from [119]. The randomized algorithm for the zero integer circuit problem, Theorem 3.10, is from [304].

The derandomization BPP = P assuming the existence of hard functions in E, Theorem 3.15, is discussed in section §11.7.

The hierarchy for BPTime is from [171], where it is attributed to folklore. See [171] for further discussion.

Chapter 4

Reductions

One can relate the complexity of functions via *reductions*. This concept is so ingrained in common reasoning that giving it a name feels, at times, strange. For in some sense pretty much everything proceeds by reductions. In any algorithms textbook, the majority of algorithms can be cast as reductions to algorithms presented earlier in the book, and so on. And it is worthwhile to emphasize now that, as we shall see below, reductions, even in the context of computing, have been used for millennia. For about a century reductions have been used in the context of undecidability in a modern way, starting with the incompleteness theorem in logic, whose proof reduces questions in logic to questions in arithmetic.

Perhaps one reason for the more recent interest in complexity reductions is that we can use them to relate problems that are tantalizingly close to problems that today we solve routinely on somewhat large scale inputs with computers, and that therefore appear to be just out of reach. By contrast, reductions in the context of undecidability tend to apply to problems that are completely out of reach, and in this sense remote from our immediate worries.

The "web" of reductions is immense. Many reductions are obtained by constructing suitable *gadgets* that allow us to simulate behavior in one setting within another. The invention of these gadgets is ingenious and technical – a form of "art" illustrated for example later in figure 4.4.

4.1 Types of reductions

Informally, a reduction from a function f to a function g is a way to compute f given that we can compute g. One can define reductions in different ways, depending on the overhead required to compute f given that we can compute g. The most general type of reduction is simply an *implication*.

General form of reduction from f to g:
If g can be computed with resources X **then** f can be computed with resources Y.

A common setting is when $X = Y$. In this case the reduction allows us to stay within the same complexity class.

Definition 4.1. We say that f reduces to g in X (or under X reductions) if

$$g \in X \Rightarrow f \in X.$$

A further special and noteworthy case is when $X = \mathrm{P}$; in these cases the reduction can be interpreted as saying that if g is easy to compute than f is too. But in general X may not be equal to Y. We will see examples of such implications for various X and Y.

If $g \notin X$ this definition trivializes, since then everything reduces to g. But the point of this definition is that it allows us to draw connections between problems *whose complexity is not known*. Still, it is sometimes useful to be more specific about how the implication is proved. For example, this is useful when inferring various properties of f from properties of g, something which can be obscured by a stark implication.

A more constrained type of reduction which is sometimes more suitable for a fine-grained analysis is the following:

Definition 4.2. We say that f *map reduces* to g in X (or via a map in X) if there is $M \in X$ such that $f(x) = g(M(x))$ for every x.

Exercise 4.3. Suppose that f map reduces to g in X.
(1) Suppose $X = \mathrm{FP}$. Show f reduces to g in X.
(2) Suppose $X = \bigcup_d \mathrm{FTime}(d \cdot n^2)$. Can you still show that f reduces to g in X?

In general, the harder the problems we are reducing the more constrained the type of reduction and the class X can be. In several interesting cases, map reductions with an extremely constrained class X suffice. But in other settings, the reductions we shall see are not even mapping reductions. In fact, our first example is not a mapping reduction.

4.2 Multiplication

Summing two n-bit integers is in $\mathrm{CktSize}(cn)$ (Exercise 2.7). But the smallest circuit known for multiplication has $\geq cn \log n$ gates. (For multiplication on word programs see the notes.) It is a long-standing question whether we can multiply two n-bit integers with a linear-size circuit.

What about squaring integers? Is that harder or easier than multiplication? Obviously, if we can multiply two numbers we can also square a number: simply multiply it by itself. This is a trivial example of a reduction. What about the other way around? We can use a reduction established millennia ago by the Babylonians. They employed the equation

$$a \cdot b = \frac{(a+b)^2 - (a-b)^2}{4} \tag{4.1}$$

to reduce multiplication to squaring, plus some easy operations like addition and division by four. In our terminology we have the following.

Definition 4.4. The Multiplication problem: Given two n-bit integers, output their product. The Squaring problem: Given an n-bit integer, output its square.

Theorem 4.5. Multiplication reduces to Squaring in linear circuit size, i.e., in $\bigcup_a \text{CktSize}(an)$.

Proof. Suppose C is a linear-size circuit computing Squaring. Then we can multiply using equation (4.1). Specifically, given a and b we use Exercise 2.7 to compute $a+b$ and $a-b$. (We haven't seen subtraction or negative integers, but it's similar to addition.) Then we run C on both of them. Finally, we again use Exercise 2.7 for computing their difference. It remains to divide by four. In binary, this is accomplished by ignoring the last two bits – which costs nothing on a circuit. **QED**

4.3 3Sum

Definition 4.6. The 3Sum problem: Given a list of integers, are there three that sum to 0?

It is easy to solve 3Sum in time $n^2 \log^c n$ with a word program, at least if the numbers have $\leq \log^c n$ bits: We can first sort the integers and then for each pair (a, b) we can do a binary search to check if $-(a+b)$ is also present. Let's convince ourselves that the bit length of the integers does not affect this algorithm:

Exercise 4.7. Prove 3Sum$\in \text{Time}(n^2 \log^c n)$. Hint: Consider the case where each integer takes w bits.

3Sum has been conjectured to require quadratic time.

Definition 4.8. $\text{SubquadraticTime} := \bigcup_{\epsilon>0} \text{Time}(n^{2-\epsilon})$.

Conjecture 4.1. 3Sum $\notin$ SubquadraticTime.

One can reduce 3Sum to a number of other interesting problem to infer that, under Conjecture 4.1, those problems require quadratic time too.

Definition 4.9. The Collinearity problem: Given a list of points in the plane, are there three points on a line?

Theorem 4.10. 3Sum reduces to Collinearity in SubquadraticTime.

Proof. We give a proof assuming that the input to 3Sum consists of distinct integers, and using the fact that multiplication is in quasilinear time (which we do not prove). The case where some integer is repeated is left as exercise. Refer to Figure 4.1 for an illustration. We map instance $a_1, a_2, \ldots, a_t$ of 3Sum to the points

$$(a_1, a_1^3), (a_2, a_2^3), \ldots, (a_t, a_t^3),$$

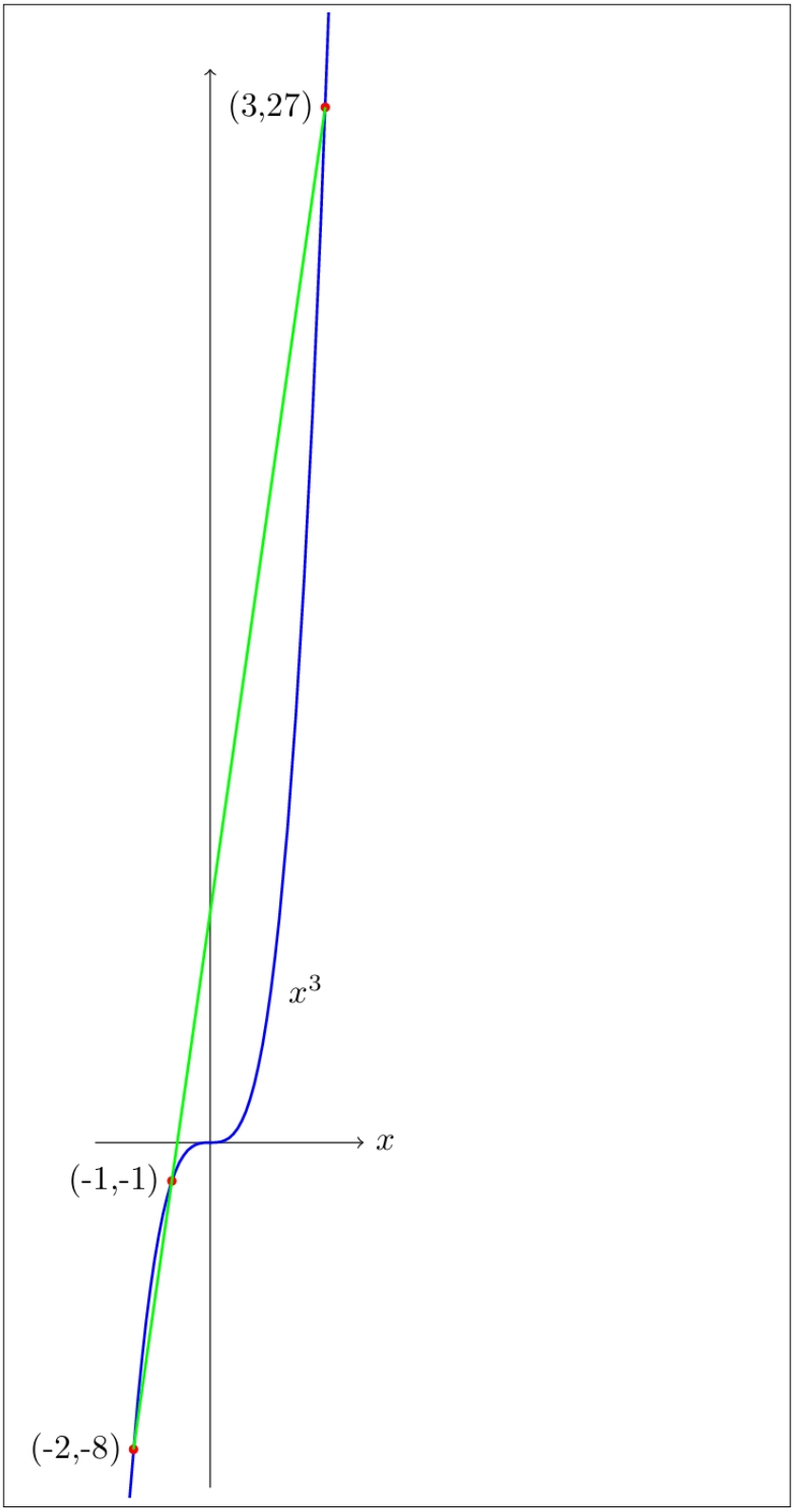


Figure 4.1: Illustration of the proof of Theorem 4.10 for the 3Sum instance $(-2,-1,3)$. Because $-2-1+3=0$, there is a straight line through the points $(-2,-2^3),(-1,-1^3),(3,3^3)$.

and solve Collinearity on those points. Computing this map takes quasi-linear time.

Let $(x, x^3), (y, y^3)$, and (z, z^3) be three points with x, y, z all distinct. Note these points are on a line iff

$$\frac{y^3 - x^3}{y - x} = \frac{z^3 - x^3}{z - x}.$$

Because $y^3 - x^3 = (y - x)(y^2 + yx + x^2)$, this condition is equivalent to

$$y^2 + yx + x^2 = z^2 + zx + x^2 \Leftrightarrow (x + (y + z))(y - z) = 0.$$

This is equivalent to $x + y + z = 0$ because $y \neq z$.

Note that the Collinearity instance has length linear in the 3Sum instance, and the result follows. **QED**

We now give a reduction in the other direction: We reduce a problem to 3Sum.

Definition 4.11. The 3Cycle problem: Given the adjacency list of a directed graph, is there a cycle of length 3?

The fastest known algorithm for 3Cycle runs in time $\geq n^{1.4}$. We can show that an improvement follows if 3Sum $\in$ Time$(n^{1+\epsilon})$ for a small enough ϵ. We give a randomized reduction. While a deterministic reduction is also possible, the randomized reduction is simpler.

Theorem 4.12. 3Sum $\in$ Time$(t(n)) \Rightarrow$ 3Cycle $\in$ BPTime$(ct(n))$, for any $t(n) \geq n$.

Proof. We assume we are given a list of edges where each node is represented using $\log n$ bits. The reduction assigns random numbers r_x with $4 \log n$ bits to each node x in the graph. The 3Sum instance consists of the integers $r_x - r_y$ for every edge $x \to y$ in the graph. Its size is linear in the input length.

To verify correctness, suppose that there is a cycle

$$x \to y \to z \to x$$

in the graph. Then we have $r_x - r_y + r_y - r_z + r_z - r_x = 0$, for any random choices.

Conversely, suppose there is no cycle, and consider any three numbers $r_{x1} - r_{y1}, r_{x2} - r_{y2}, r_{x3} - r_{y3}$ from the reduction and its corresponding edges. Some node xi has unequal in-degree and out-degree in those edges. This means that when summing the three numbers, the random variable r_{xi} will not cancel out. When selecting uniform values for that variable, the probability of getting 0 is at most $1/t^4$.

By a union bound, the probability there there are three numbers that sum to zero is $\leq t^3/t^4 < 1/3$. **QED**

Many other clusters of problems exist, for example based on matrix multiplication or all-pairs shortest path.

4.4 Satisfiability

In this section we begin to explore an important cluster of problems not known to be in P. What's special about these problems is that in Chapter 5 we will show that we can reduce *arbitrary computation* to them, while this is unknown for the problems in the previous section.

The basic problem in this class is to determine, given a circuit C, if there is an input x s.t. $C(x) = 1$. In fact, for this chapter it suffices to work with circuits which are made of just a couple of layers of And, Or gates, with *unbounded* fan-in. This special case is defined next and linked to the general problem in the next Chapter 5.

Definition 4.13. A *literal* is a variable x or its negation $\neg x$. A *clause* is an Or of literals. A *conjunctive normal form formula (CNF)* is an And of clauses. A *kCNF* is a CNF where each clause has k literals. A CNF is *satisfiable* if there exists a boolean assignment to the variables on which the circuit outputs 1. We also call 0 *false* and 1 *true* in this context.

The 3Sat problem: Given a 3CNF ϕ, is there an assignment x s.t. $\phi(x) = 1$?

Example 4.14. The formula $(x \vee y \vee z) \wedge (z \vee \neg y \vee \neg x)$ is a 3CNF. This CNF is satisfiable because the assignment $x = 1, y = 0, z = 0$ gives $(1 \vee 0 \vee 0) \wedge (0 \vee 1 \vee 0) = 1 \wedge 1 = 1$.

On the other hand, the 3CNF $(x \vee x \vee x) \wedge (\neg x \vee \neg x \vee \neg x)$ is not satisfiable: no matter how x is set, one of the two clauses will be 0.

There are two main reasons why 3Sat is important. First, a number of important problems are routinely reduced to 3Sat and then solved using sat solvers. Second, we will show in the next Chapter 5 that is captures *arbitrary computation*.

One can solve 3Sat in exponential time via *brute-force*: Try all assignments to the variables. Despite much effort no significantly faster algorithm is known. In particular, the following is a leading conjecture of complexity theory.

Conjecture 4.2. 3Sat $\notin$ P.

In fact, stronger conjectures have been made, parameterized by the number of variables, basically asserting that brute-force search is the best that we can do.

Conjecture 4.3. [Exponential time hypothesis (ETH)] There is $\epsilon > 0$ such that there is no algorithm that on input a 3CNF ϕ with v variables and cv^3 clauses decides if ϕ is satisfiable in time $2^{(\epsilon+o(1))v}$.

Conjecture 4.4. [Strong exponential-time hypothesis (SETH)] For every $\epsilon > 0$ there is k such that there is no algorithm that on input a kCNF ϕ with v variables and cv^k clauses decides if ϕ is satisfiable in time $2^{(1-\epsilon+o(1))v}$.

It is known that SETH $\Rightarrow$ ETH, but the proof is not immediate.

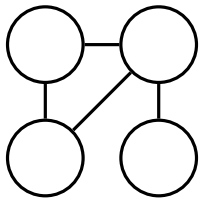
We now give reductions from 3Sat to several other problems. The reductions are in fact mapping reductions. Moreover, the reduction map can be extremely restricted, see Problem 4.4. In this sense, therefore, these reductions can be viewed as direct translations of the

problems, and maybe we shouldn't really be thinking of the problems as different, even if they at first sight refer to different objects (formulas, graphs, numbers, etc.). More on this perspective is in Chapter 18.

4.4.1 3Sat to Clique

Definition 4.15. The Clique problem, given a graph G and an integer t, are there t nodes in G that are all connected? The latter is called a *clique* of size t.

Example 4.16. The following graph has a clique of size 3 but not of size 4:



Theorem 4.17. 3Sat reduces to Clique in P.

Proof. Given a 3CNF φ with k clauses, we construct a graph G with $3k$ nodes where we have a node for each literal occurrence. We then connect all except

(A) Nodes in same clause, and

(B) Contradictory nodes, such as x and $\neg x$.

The construction is in FP.

We claim that φ is satisfiable iff G has a clique of size k.

Only if: Given a satisfying assignment, collect exactly one node which is satisfied in each clause. This makes $t = k$ nodes. For any pair of such nodes, (A) does not hold by construction, and (B) because they correspond to an assignment.

If: Given a clique of size t, pick any assignment that makes the corresponding literals true. This is a valid definition by (B). Also, because of (A), there is at least one true literal in each clause. **QED**

An example of the reduction is illustrated in figure 4.2.

4.4.2 3Sat to Subset-Sum

Definition 4.18. The Subset-sum problem: Given n integers a_i and a target t, is there a subset of the a_i that sums to t?

Example 4.19. There is a subset of $5, 2, 14, 3, 9$ summing to $t := 25$ $(2 + 14 + 9 = 25)$. But there is no subset of $1, 3, 4, 9$ summing to $t := 15$.

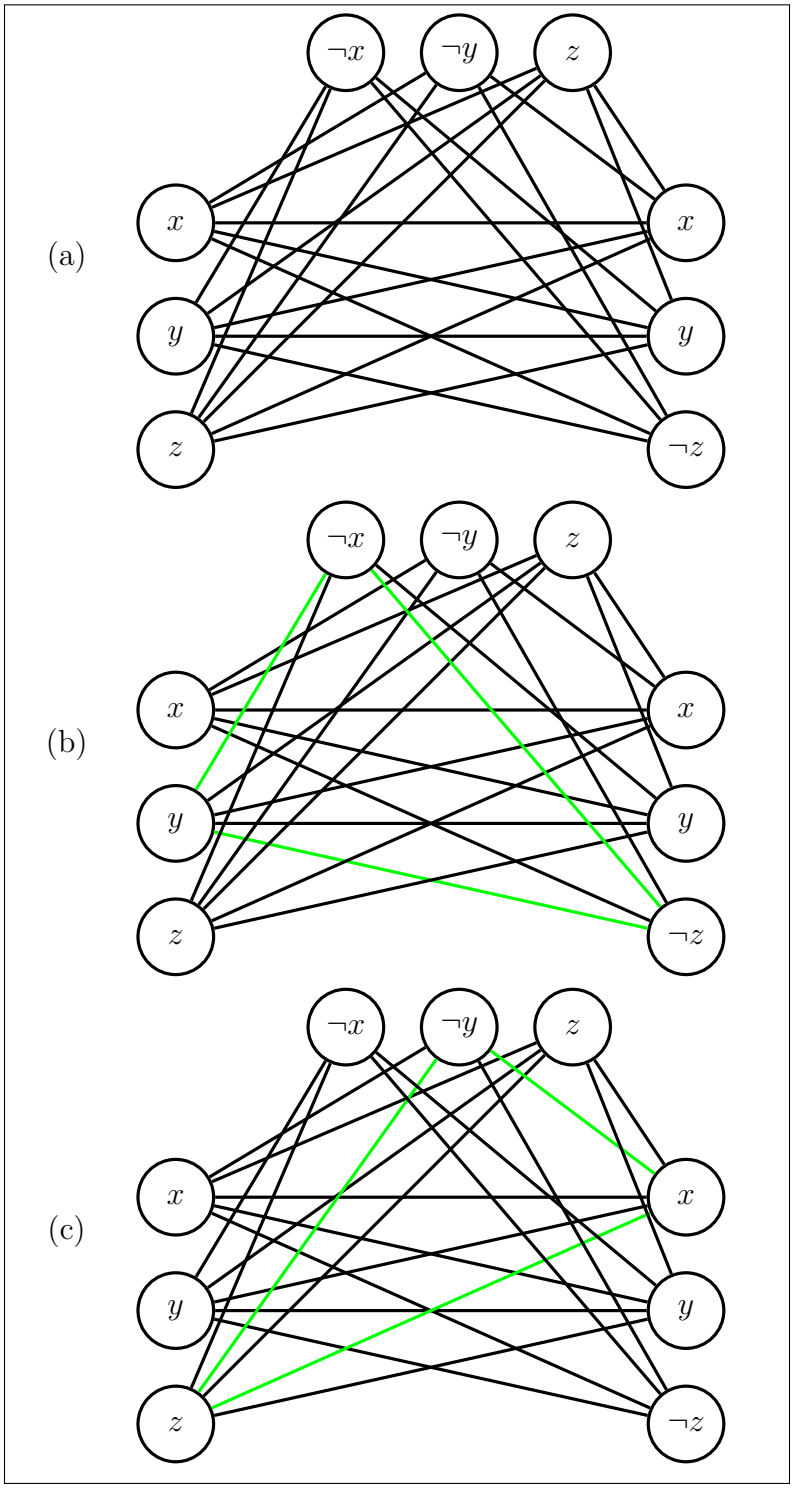


Figure 4.2: Example of reduction from 3Sat to Clique, proof of Theorem 4.17. Graph (a) is the image of the reduction on 3CNF $(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z)$. We seek cliques of size $t = k = 3$, a.k.a. triangles. A satisfying assignment is $x = 0; y = 1; z = 0$. The corresponding clique is shown in (b). Another satisfying assignment is $x = 1; y = 0; z = 1$. The corresponding clique is shown in (c).

Subset-sum is also a very interesting problems. If the numbers are small it can be solved in power time via dynamic programming. Hence the next reduction capitalizes on the magnitude of the numbers.

Theorem 4.20. 3Sat reduces to Subset-sum in P.

Proof. On input φ with v variables and k clauses we produce a list of numbers with $v + k$ digits. The most significant v correspond to variables; the other k to clauses. For each variable x include number a_x^T which has 1 in the digit corresponding to x, and a 1 in every digit of a clause where x appears without negation. Similarly, include number a_x^F which also has a 1 in the digit corresponding to x, and now a 1 in every digit of a clause where x appears negated.

Also, for each clause C, include twice the number a_C which has a 1 in the digit corresponding to C, 0 in others.

Set t to be 1 in first v digits, and 3 in rest k digits.

This construction is in FP.

Now suppose φ has satisfying assignment. Pick a_x^T if x is true, a_x^F if x is false. The sum of these numbers yield 1 in first v digits by construction. It also yields 1, 2, or 3 in each of the last k digits because each clause has a true literal. By picking appropriate subset of the numbers a_C we can reach t.

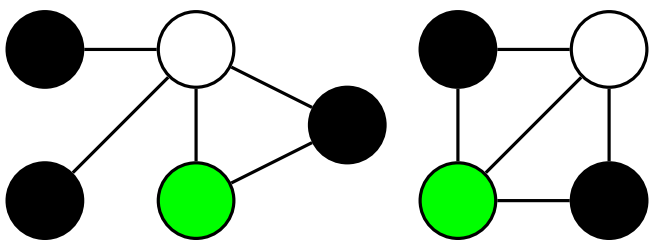
Conversely, given a subset, note that there is no carry in sum, because there are only 3 literals per clause. So digits behave "independently." For each pair a_x^T, a_x^F exactly one is included, otherwise would not get 1 in that digit. Define x true if a_x^T included, false otherwise. For any clause C, the a_C contribute ≤ 2 in that digit. So each clause must have a true literal otherwise sum would not get to 3 in that digit. **QED**

An example of the reduction is illustrated in figure 4.3.

4.4.3 3Sat to 3Color

Definition 4.21. For $k \in \mathbb{N}$, a *k-coloring* of a graph is a coloring of each node using at most k colors, such that no adjacent nodes have the same color. The kColor problem: Given a graph G, does it have a k coloring?

Example 4.22. The following graphs have a 3-coloring, shown:



(a)

	var x	var y	var z	clause 1	clause 2	clause 3
$a_x^T =$	1	0	0	1	0	1
$a_x^F =$	1	0	0	0	1	0
$a_y^T =$	0	1	0	1	0	1
$a_y^F =$	0	1	0	0	1	0
$a_z^T =$	0	0	1	1	1	0
$a_z^F =$	0	0	1	0	0	1
$a_{c1} =$	0	0	0	1	0	0
$a_{c2} =$	0	0	0	0	1	0
$a_{c3} =$	0	0	0	0	0	1
$t =$	1	1	1	3	3	3

(b)

	var x	var y	var z	clause 1	clause 2	clause 3	
$a_x^T =$	1	0	0	1	0	1	
$a_x^F =$	1	0	0	0	1	0	
$a_y^T =$	0	1	0	1	0	1	
$a_y^F =$	0	1	0	0	1	0	
$a_z^T =$	0	0	1	1	1	0	
$a_z^F =$	0	0	1	0	0	1	
$a_{c1} =$	0	0	0	1	0	0	(twice)
$a_{c2} =$	0	0	0	0	1	0	(twice)
$a_{c3} =$	0	0	0	0	0	1	
$t =$	1	1	1	3	3	3	

(c)

	var x	var y	var z	clause 1	clause 2	clause 3	
$a_x^T =$	1	0	0	1	0	1	
$a_x^F =$	1	0	0	0	1	0	
$a_y^T =$	0	1	0	1	0	1	
$a_y^F =$	0	1	0	0	1	0	
$a_z^T =$	0	0	1	1	1	0	
$a_z^F =$	0	0	1	0	0	1	
$a_{c1} =$	0	0	0	1	0	0	
$a_{c2} =$	0	0	0	0	1	0	(twice)
$a_{c3} =$	0	0	0	0	0	1	
$t =$	1	1	1	3	3	3	

Figure 4.3: Example of reduction from 3Sat to Clique, proof of Theorem 4.20. Graph (a) is the image of the reduction on 3CNF $(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z)$. The numbers a_{c1}, a_{c2}, a_{c3} have two occurrences each. A satisfying assignment is $x = 0, y = 1, z = 0$. The corresponding subset is shown in (b). Another satisfying assignment is $x = y = z = 1$ with corresponding subset shown in (c).

An example of a graph that cannot be 3-colored is a clique of size 4.

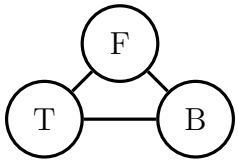
Exercise 4.23. Reduce 3-Color to 4-Color in P.

The main result in this section is the following.

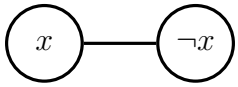
Theorem 4.24. 3Sat reduces to 3Color in P.

Proof. Given a 3CNF ϕ, we construct a graph G as follows.

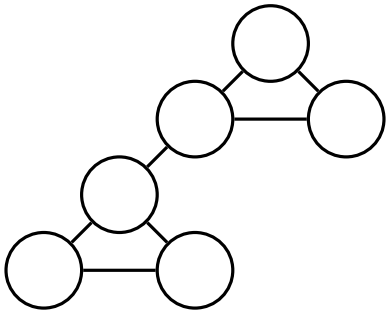
Add a clique called the "palette" with 3 special nodes, T for True, F for False, and B for Base:



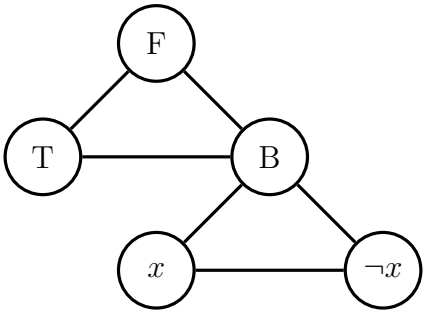
For each variable add 2 literal nodes with an edge between them



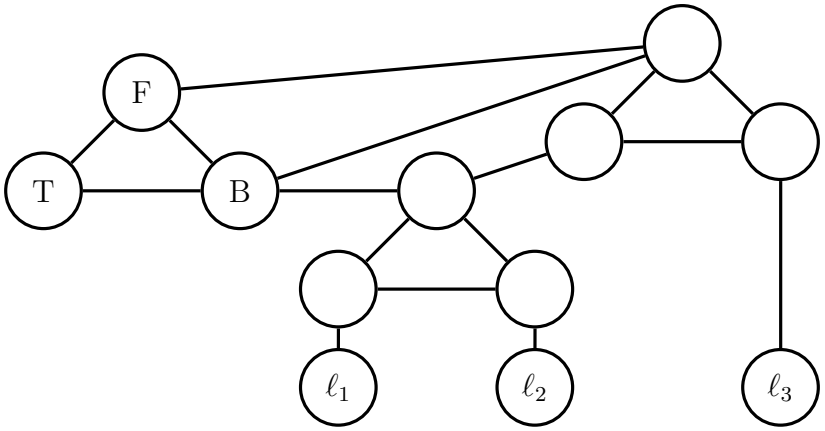
For each clause add the following gadget with 6 nodes



Connect each literal node to node B in the palette



For each clause (ℓ_1, ℓ_2, ℓ_3) connect the clause gadget to the palette and to the nodes ℓ_i as follows:



The construction of G is in P. We now prove that φ is satisfiable iff G is 3 colorable. We begin with some preliminary remarks. In the palette, T's color represents TRUE, and F's color represents FALSE. Note in a 3-coloring, all variable nodes must be colored T or F because they are connected to B. Also, x and $\neg x$ must have different colors because they are connected. So we can "translate" a 3-coloring of G into a true/false assignment to variables of φ.

The important claim is that a clause gadget can be 3-colored iff any of the literals connected to it is colored True. This holds because each of the two triangles in a the clause gadget is computing "Or:" In a triangle, the top node is colored according to the Or of the two literals connected to the bottom two nodes in the triangle. For example, if the literals are both F, then the bottom nodes in the triangle must be colored T and B, and so the top is F.

The result follows. Given a satisfying assignment, we can pick the corresponding coloring of the literal nodes and extend it to a 3 coloring of the entire graph. Vice versa, given a 3 coloring of the graph we can infer an assignment to the variables and note that each clause has a true literal since each clause gadget is 3 colored. **QED**

An example of the reduction is illustrated in figure 4.4.

4.4.4 More

The web of reductions from 3Sat is immense, see the notes for various compilations. Amusingly, there are reductions from 3Sat to (generalized versions of) several popular videogames, including *Tetris*, *Lemmings*, and *Super Mario*. The basic mechanism used in many of these results (including the latter two) is a *one-way commitment gadget* that forces the player to commit to a boolean choice. This mechanism is illustrated in a simple setting in Problem 4.1.

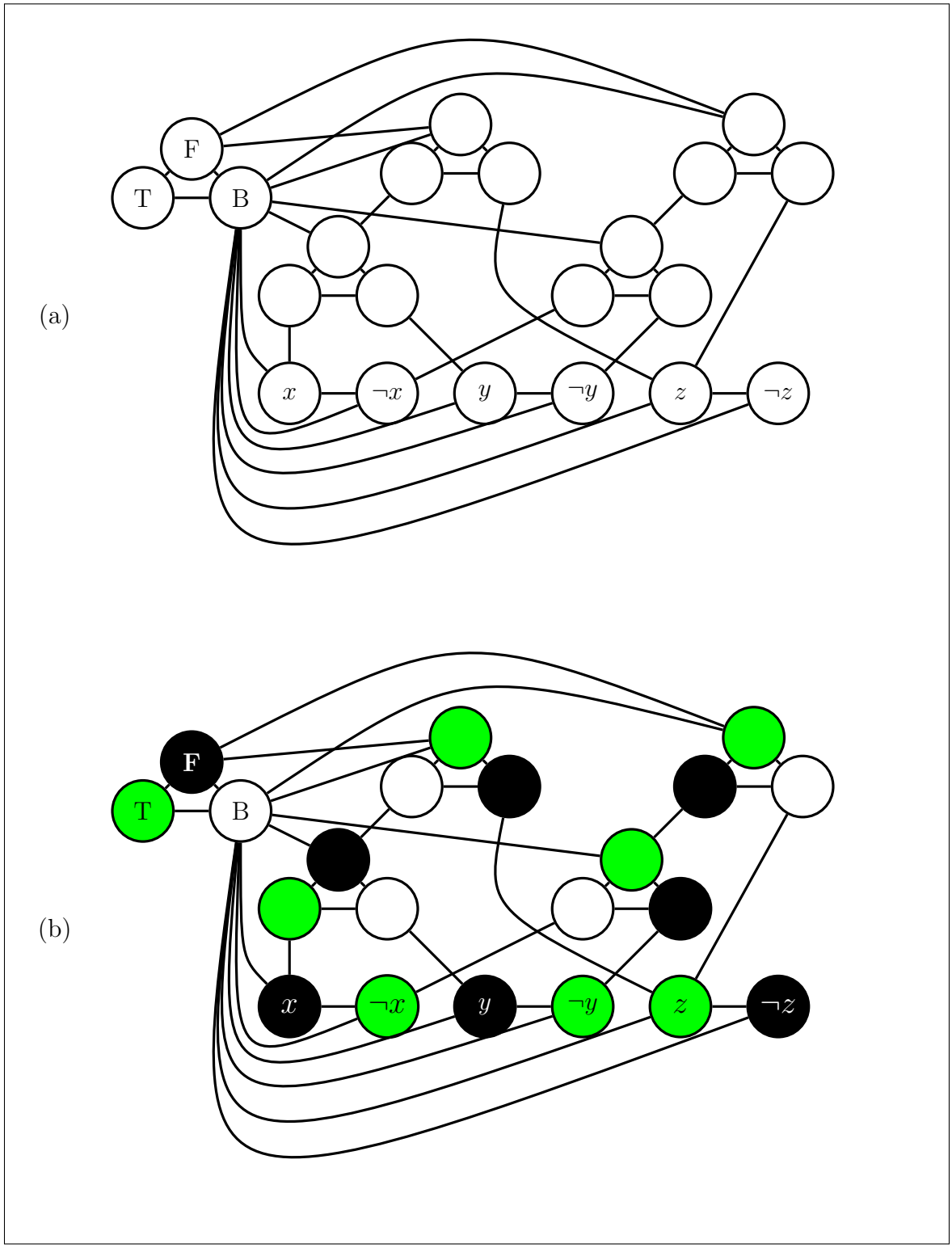


Figure 4.4: Example of reduction from 3Sat to 3Color, proof of Theorem 4.24. Graph (a) is the image of the reduction on 3CNF $(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee z)$. A satisfying assignment is $x = y = 0$ and $z = 1$. The corresponding coloring is shown in (b).

While in this section we have focused on reductions from 3Sat, it is important to note that reductions in the opposite direction exist as well, and so in fact the problems in this section form a cluster of *power-time equivalent* problems: Any one of the problems is in P iff all the others are. We will see a generic reduction in the next chapter. For now, we illustrate this equivalence in a particular case.

Exercise 4.25. Reduce 3Color to 3Sat in P, following these steps:

1. Given a graph G, introduce variables $x_{i,d}$ representing that node i has color d, where d ranges in the set of colors $C = \{g, r, b\}$. Describe a set of clauses that is satisfiable if and only if for every i there is exactly one $d \in C$ such that $x_{i,d}$ is true.
2. Introduce clauses representing that adjacent nodes do not have the same color.

For more reductions see the problems section.

4.5 Power hardness from SETH

Assuming SETH it has been shown that for a number of prominent problems, such as *longest common subsequence,* or *edit distance*, the classical algorithms that run in quadratic time are the best possible: The problems cannot be solved in subquadratic time. In this section we prove a result of this flavor, which is in fact the building block of the results we just mentioned.

Definition 4.26. The Or-Vector problem: Given two lists A and B of strings of the same length, determine if there is $a \in A$ and $b \in B$ such that the bit-wise Or $a \vee b$ equals the all-one vector.

Similarly to 3Sum, the Or-Vector problem is in Time(cn^2), in fact the setting is slightly easier as it's more natural to assume that the bit vectors have the same length here, see Exercise 4.7. We can show that a substantial improvement would disprove SETH.

Theorem 4.27. Or-Vector $\in$ SubquadraticTime $\Rightarrow$ SETH is false.

Proof. Given a kCNF ϕ of size n with v variables and $d \leq n$ clauses, divide the v variables in two blocks of $v/2$ each. For each assignment to the variables in the first block construct the vector in $[2]^d$ where bit i is 1 iff clause i is satisfied by the variables in the first block. Call A the resulting set of vectors. Let $N := 2^{v/2}$ and note $|A| = N$. Do the same for the other block and call the resulting set B.

Note that ϕ is satisfiable iff Or-Vector(A, B) is true, i.e., $\exists a \in A, b \in B$ such that $a \vee b = 1^d$.

Constructing these sets takes time cNn: For each of the N assignments, we scan the input to construct the vector.

The instance length of Or-Vector is Nd. If Or-Vector $\in$ SubquadraticTime we can then solve the kCNF instance in time $cNn + (Nd)^{2-\epsilon} \leq c2^{n/2}n + 2^{n(2-\epsilon)/2}n^{2-\epsilon}$, for some $\epsilon > 0$. Taking $k = c_\epsilon$ then rules out SETH. **QED**

This proof is an interesting example of how we can connect different parameter regimes. SETH is stated in terms of exponential running times, but we can get an improvement by "scaling up" an algorithm for Or-Vector. In general, "scaling" parameters is a powerful technique in the complexity toolkit.

4.6 Search problems

Most of the problems in the previous sections ask about the *existence* of solutions. For example 3Sat asks about the existence of a satisfying assignment. It is natural to ask about computing such a solution, if it exists. Such non-boolean problems are known as *search problems*.

Next we show that in some cases we can reduce a search problem to the corresponding boolean problem.

Definition 4.28. The Search-3Sat problem: Given a satisfiable 3CNF formula, output a satisfying assignment.

Theorem 4.29. Search-3Sat reduces to 3Sat in FP.

Proof. We construct a satisfying assignment one variable at the time. Given a satisfiable 3CNF, set the first variable to 0 and check if it is still satisfiable with the assumed algorithm for 3Sat. If it is, go to the next variable. If it is not, set the first variable to 1 and go to the next variable. **QED**

Exercise 4.30. The Search-Clique problem: Given a graph G and an integer t s.t. G has a clique of size t, output one such clique. Show that Search-Clique reduces to Clique in FP.

Recall that in Theorem 1.19 we gave the "fastest" algorithm for Factoring. Similar algorithms exist for other search problems. The next exercise asks you to verify this for 3Sat.

Exercise 4.31. There is a word program U that computes Search-3Sat and has the following property: For any other word program P for Search-3Sat, and every input x, if P runs in time t on x then U runs in time $c_P t + c_p |x| \log^c |x| \log t$. Note and explain why the time bound is better than Theorem 1.19.

4.7 Gap-Sat: The PCP theorem

> *Furthermore, most problem reductions do not create or preserve such gaps. There would appear to be a last resort, namely to create such a gap in the generic reduction [...]. Unfortunately, this also seems doubtful. The intuitive reason is that computation is an inherently unstable, non-robust mathematical object, in the sense that it can be turned from non-accepting by changes that would be insignificant in any reasonable metric – say, by flipping a single state to accepting.*

One of the most exciting, consequential, and technical developments in complexity theory of the last few decades has been the development of reductions that create *gaps.*

Definition 4.32. The γ-Gap-3Sat problem: Solve 3Sat on formulas that are either satisfiable or are such that any assignment satisfies at most a γ fraction of clauses.

Note that 3Sat is equivalent to γ-Gap-3Sat for $\gamma = 1 - 1/n$, since a formula of size n has at most n clauses. At first sight it is unclear how to connect the problems when γ is much smaller. Surprisingly, and contrary to the quote above, it is possible to obtain a constant γ. This result is known as the PCP theorem, where PCP stands for probabilistically-checkable-proofs. The connection to proof systems will be discussed in Chapter 10.

Theorem 4.33. [PCP] 3Sat map reduces to $(1-c)$-Gap-3Sat in P.

We give an overview of the proof in section §12.5.

This PCP Theorem 4.33 has several exciting consequences. As mentioned before, we shall prove in Chapter 5 that arbitrary computation can be reduced to 3Sat. Hence the PCP theorem gives us a way to *robustify* computation and make it more stable. This is discussed more in Chapter 5.

Another consequence of the PCP Theorem 4.33 is that, if we believe Conjecture 4.2, not only we cannot determine if a given 3CNF is satisfiable, but we can't even distinguish the case in which every clause can be satisfied from the case in which at most $1-c$ fraction of clauses can be satisfied. A consequence of this is that we cannot compute an *approximation* to the maximum number of clauses that can be satisfied. In this sense, 3Sat is *inapproximabile.* It has been a major line of research to obtain tight inapproximability results for a variety of problems. This is well-motivated by the fact that we often don't need to compute an optimal solution, but a good enough solution will do. Similar inapproximability results can be established for other problems such as 3Color and Clique. For some problems such as Clique this follows from the reduction we saw, as the next exercise asks you to verify.

Definition 4.34. The γ-Gap-Clique problem: Solve Clique on pairs (G, t) where G either has a clique of size $\geq t$ or has no clique larger than γt.

Exercise 4.35. Assuming Theorem 4.33, reduce 3Sat to γ-Gap-Clique for a constant $\gamma < 1$.

In general, however, reductions among problems may not preserve gaps.

4.8 Problems

Exercise 4.36. The problem System: Given a systems of linear inequalities, does it have an integer solution?

For example,

$$(x + y \geq z, x \leq 5, y \leq 1, z \geq 5) \in \text{System};$$
$$(x + y \geq 2z, x \leq 5, y \leq 1, z \geq 5) \notin \text{System}.$$

Reduce 3Sat to System in P.

Problem 4.1. The problem Game: The input is a directed graph with a special source node s, m destination nodes $t_1, t_2, \ldots, t_m$, and a subset B of *collapsing nodes*. Decide whether there are m paths from s to each of the destination nodes. The paths can share edges, but any two paths entering a collapsing node must leave through the same outgoing edge.

Reduce 3Sat to Game in P.

Problem 4.2. The Quad-Sys problem: Given a system of quadratic equations over $\mathbb{F}_2$, decide if it has a solution. Reduce 3Sat to Quad-Sys in P.

Problem 4.3. Reduce Search-3Color to 3Color in P.

Problem 4.4. Specify suitable encodings of 3Sat and 3Color and prove that 3Sat map reduces to 3Color in ProjectionsP (see Definition 5.10).

4.9 Notes

Circuits of size $cn \log n$ for multiplication were obtained in [162]. It has been shown in [8] that this size is tight based on a conjecture in network coding. The paper [305] gives a linear-time algorithm for multiplication on a "storage modification machine." It *seems* this algorithm can be written as a word program, but I don't know this has been verified. See also discussion in section 4.3.3.C in [215].

Following [93] (discussed in the next chapter), [206] established reductions from satisfiability of (general) boolean formulas to 21 problems, including 3Sat, Clique, Cover-by-vertexes, 3Color, and Subset Sum. This opened the floodgates, see [126] for a starter, or the list on wikipedia, [382]. For one exposition of a reduction to a videogame see the video [54]. For videos covering the reductions in section §4.4, see videos 29, 30, 31, and 32 from [358] (the videos use the terminology "polynomial time" instead of "power time" here).

The ETH and the SETH are from [189] and [191]. Again, a large number of reductions involving these hypotheses exists. In particular, tight hardness results based on SETH have been established for several well-studied problems, including longest-common subsequence [5] and edit distance [41]. See also [40].

The web of reductions of 3Sum, including Theorem 4.10, was first spun in [122] and has grown ever since. Theorem 4.12 is from [367].

Theorem 1.19 is from [228].

The quote at the beginning of section §4.7 is from [277]. The PCP theorem as stated in Theorem 4.33 is from [30]. A sequence of exciting works preceded and followed it. For an account, as well as a proof of the PCP theorem, see [29].

Problem 1.2 is from [343].

Chapter 5

Nondeterminism

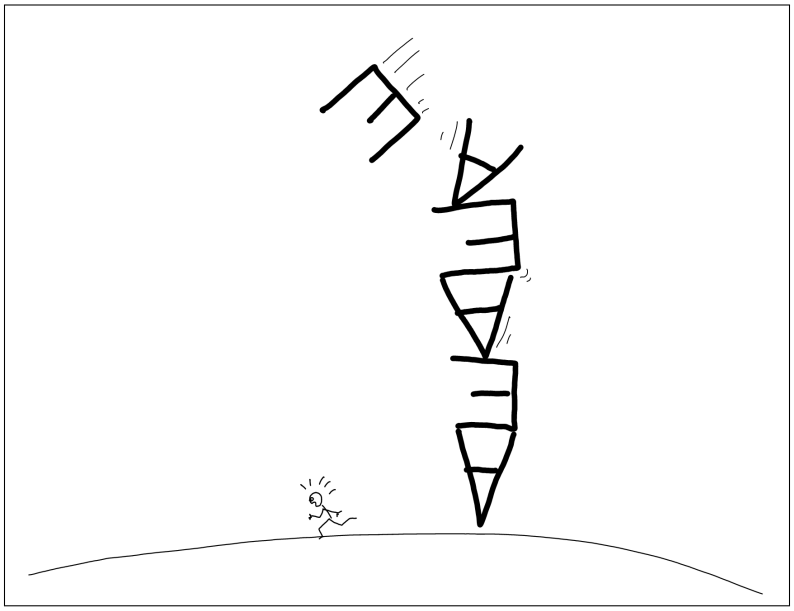


Figure 5.1: Does the PH collapse?

In this chapter we show how to reduce *arbitrary computation* to 3Sat (and hence to the other problems we showed in Section §4.4 3Sat reduces to). While in Chapter 4 we already showed that some problems like 3Color can be reduced to 3Sat, we now show how to reduce *any* problem computable in a certain class to 3Sat (the class includes 3Color and many other problems). This is a *generic, model-based* form of reducibility.

What powers everything is the following landmark and, in hindsight, simple result which reduces circuit computation to 3Sat. Specifically, given a circuit we can construct a formula whose satisfiability is equivalent to the output of the circuit.

Theorem 5.1. Given a circuit $C : [2]^n \to [2]$ with s gates we can compute in FP a 3CNF

formula φ_C in $n+s$ variables such that for every $x \in [2]^n$:

$$C(x) = 1 \Leftrightarrow \exists y \in [2]^s : \varphi_C(x, y) = 1.$$

The key idea to *guess computation and check it efficiently, using that computation is local.* The additional s variables one introduces contain the values of the gates during the computation of C on x. We simply have to check that they all correspond to a valid computation, and this can be written as 3CNF because each gate depends on at most two other gates.

Proof. Introduce a variable y_i for each computation gate g_i in C. The value of y_i is intended to be the value of gate g_i during the computation. Whether the value of a gate g_i is correct is a function γ of 3 variables: y_i and the ≤ 2 gates that input g_i, some of which could be input variables. This γ can be written as a 3CNF by the first constructions in the proof of Theorem 2.5. (We can apply that construction to write $\neg\gamma$ as a 3DNF, then complement that to obtain a 3CNF for γ.) Finally, take an And of all these 3CNFs, and add clause y_o for the output gate g_o. **QED**

Exercise 5.2. Write down the 3CNF for the circuit in Example 2.2, as given by the proof of Theorem 5.1.

We can combine Theorem 5.1 with the simulation of word programs by circuits (Theorem 2.12) to reduce time to 3Sat. We present this in Theorem 5.13. However, this simulation has a power loss, and using it we end up reducing time t to 3CNFs of size $\geq t^2$. Later in section §5.3 we obtain a quasi-linear simulation using an enjoyable argument which bypasses Theorem 2.12.

In fact, these simulations apply to a more general, *non-deterministic*, model of computation. We define this model next.

5.1 Nondeterministic computation

In the concluding equation in Theorem 5.1 there is an $\exists$ quantifier on the right-hand side, but there isn't one on the left, next to the circuit. However, because the simulation works for every input, we can "stick" a quantifier on the left and have the same result. The resulting circuit computation $C(x, y)$ has two inputs, x and y. We can think of it as a *non-deterministic* circuit, which on input x outputs 1 iff $\exists y : C(x, y)$. The message here is that – if we allow for an $\exists$ quantifier, or in other words consider nondeterministic computation – efficient computation is *equivalent* to 3CNF! This is one motivation for formally introducing a *nondeterministic* computational model.

Definition 5.3. NTime($t(n)$) is the set of functions $f : X \subseteq [2]^* \to [2]$ for which there is a word program P that for every $x \in X$ of length $\geq |P|$ satisfies:
- $f(x) = 1$ iff $\exists y \in [2]^{t(n)}$ such that $P(x, y) = 1$, and
- $P(x, y)$ stops within $t(n)$ steps for every $y \in [2]^{t(n)}$.

We also define

$$\text{Quasi-Linear-NTime} := \bigcup_{d \geq 1} \text{NTime}(n \log^d n),$$

$$\text{Non-deterministic-Power-Time, NP} := \bigcup_{d \geq 1} \text{NTime}(n^d),$$

$$\text{Non-Deterministic-Exponential-Time, NExp} := \bigcup_{d \geq 1} \text{NTime}(2^{n^d}).$$

Note that the running time of P is a function of $|x|$, not $|(x, y)|$. This difference is inconsequential for NP, since the composition of two powers is another power. But it is important for a more fine-grained analysis. For example, the proof of Theorem 5.19 uses that the running time is a function of $|x|$.

We refer to a word program as in Definition 5.3 as a *nondeterministic program*, and to the y in $P(x, y)$ as the *nondeterministic choices,* or *guesses,* of the machine on input x.

We can also define NTime in a way that is similar to BPTime, Definition 3.1. That is, we could use randomized word programs, and say that the output is 1 if the probability of outputting 1 is > 0. The instruction Rand in this case should be better be called Guess. The two definitions are equivalent for typical time bounds (but $t(n)$ is always easy to compute from the input length in Definition 5.3, whereas it could be hard to compute in the other definition). My choice for BPTime is motivated by the identification of BPTime with computation that is actually run. For example, in a programming language one uses an instruction like Rand to obtain random values; one does not think of the randomness as being part of the input. By contrast, NTime is a more abstract model, and the definition with the nondeterministic guesses explicitly laid out is closer in spirit to a 3CNF.

All the problems we studied in Section §4.4 are in NP.

Fact 5.4. 3Sat, Clique, SubsetSum, and 3Color are in NP.

Proof. For a 3Sat instance f, the variables y correspond to an assignment. Checking if the assignment satisfies f is in P. This shows that 3Sat is in NP. **QED**

Exercise 5.5. Finish the proof by addressing the other problems in Fact 5.4

We can think of NP as the problems which admit a solution that can be verified efficiently, namely in P. For example for 3Sat it is easy to verify if an assignment satisfies the clauses, for 3Color it is easy to verify if a coloring is such that any edge has endpoints of different colors, for SubsetSum it is easy to verify if a subset has a sum equal to a target, and so on.

The theory of NP completeness shows that *the proof verification can be implemented in a restricted model*, namely a 3CNF. So we don't have to think of the verification step as using the full power of P: Any proof can be turned into another proof that can be verified by a 3CNF.

The PCP theorem takes this one step further: *The proof verification can be implemented in constant randomized time*. For Gap-3Sat, you pick a uniform clause, and check if it's satisfied.

P vs. NP

The flagship question of complexity theory is whether P = NP or not. This is a young, prominent special case of the grand challenge (section §1.8). Contrary to the belief that BPP = P, see section 3.4, the general belief seems to be that P ≠ NP. Similarly to BPP, see Theorem 3.13, the best deterministic simulation of NP runs in exponential time by trying all nondeterministic guesses. This gives the middle inclusion in the following fact; the other two are by definition.

Fact 5.6. P ⊆ NP ⊆ Exp ⊆ NExp.

Recall that a consequence of the Time Hierarchy Theorem 1.30 is that P ≠ Exp (Corollary 1.31). From the inclusions above it follows that

$$\text{P} \neq \text{NP or NP} \neq \text{Exp, possibly both.}$$

Thus, we are not completely clueless, and we know that at least one important separation is lurking somewhere. Most people appear to think that *both* separations hold, but we are unable to prove *either*.

Randomness vs. NP

Essentially by definition, NP contains the one-sided version of BPP, called RP, see Section §3.1.

Exercise 5.7. Prove this.

The relationship with BPP is not clear. However, we have the following result.

Theorem 5.8. P = NP ⇒ P = BPP.

Thus, if nondeterminism can be dispensed with, so can (double-sided) randomness. Theorem 5.8 is the combination of two results discussed later: Theorem 5.37 and Theorem 5.36.

5.2 Completeness

We now go back to the question at the beginning of this chapter about reducing arbitrary computation to 3Sat. We shall reduce all of NP to 3Sat. Problems admitting such reductions deserve a definition. Since we will encounter similar reductions in a variety of contexts, we give a general definition.

Definition 5.9. Let X, Y, R be sets of functions. We call a function f:
hard for X vs Y if $f \in X \Rightarrow X = Y$;
complete for X vs Y if $f \in Y$ and f is hard for X vs Y;
hard for Y under map reductions in R if every $g \in Y$ map reduces to f in R;
complete for Y under map reductions in R if $f \in Y$ and f is hard for Y under map reductions in R.

Just like for reductions (see section §4.1) the definitions with map reductions are more constrained, and they imply the others as as long as X is closed under R. For example, if f is hard for NP under map reductions in P then f is hard for P vs NP. Many problems we encounter (including 3Sat) are in fact hard under surprisingly constrained reductions, basically just duplicating or fixing bits. We define one such type of reductions next.

Definition 5.10. ProjectionsP is the subset of CktP where each output bit is either 0, 1, or an input literal (a variable x_i or its negation).

See Problem 4.4 for an example of a reduction under projections. However for simplicity we often state the results for more general classes of reductions, like FP.

Complete problems are the "hardest problems" in the class, as formalized in the following fact.

Fact 5.11. Suppose f is complete for P vs NP. Then $f \in \mathrm{P} \Leftrightarrow \mathrm{P} = \mathrm{NP}$.

Proof. ($\Leftarrow$) This is because $f \in \mathrm{NP}$.

($\Rightarrow$) Let $f' \in \mathrm{NP}$. Because f is hard we know that $f \in \mathrm{P} \Rightarrow f' \in \mathrm{P}$. **QED**

Exercise 5.12. Suppose $\mathrm{P} = \mathrm{NP}$. Prove that any problem in NP is complete for P vs NP.

Suppose instead $\mathrm{P} \neq \mathrm{NP}$. Let $f \in \mathrm{NP}$. Prove f is complete for P vs NP iff $L \notin \mathrm{P}$.

Fact 5.11 points to an important interplay between problems and complexity classes. We can study complexity classes by studying their complete problems, and vice versa.

The central result in the theory of NP completeness is the following.

Theorem 5.13. 3Sat is complete for P vs NP.

Proof. 3Sat is in NP by Fact 5.4. Next we prove hardness. As mentioned earlier, the main idea is to combine Theorem 5.1 and Theorem 2.12; the rest of the proof mostly amounts to opening up definitions.

Let $f \in \mathrm{NP}$ and let P be a corresponding word program which runs in time n^d on inputs (x, y) where $|x| = n$ and $|y| = n^d$, for some constant d. By the simulation of word programs by circuits, Theorem 2.12, we can compute in FP a circuit $C(x, y)$ of size $n^{c_{d,P}}$ such that for any x, y we have $P(x, y) = 1 \Leftrightarrow C(x, y) = 1$.

Now, suppose we want to compute f on an input w. This is equivalent to deciding if $\exists y : C(w, y) = 1$ by what we just said. We can "hard-wire" w into C to obtain the circuit $C_w(y) := C(w, y)$ only on the variables y. Here by "hard-wire" we mean replacing the input gates x with the bits of w. The size of C_w is at most the size of C plus 2, in case we need to add the constant gates. Now we can apply Theorem 5.1 to this new circuit to produce a 3CNF φ_w on variables y and new variables z such that $C_w(y) = 1 \Leftrightarrow \exists z : \varphi_w(y, z) = 1$, for any y. The size of φ_w and the number of variables z is power in the size of the circuit.

We have obtained:

$$f(w) = 1 \Leftrightarrow \exists y : P(w, y) = 1 \Leftrightarrow \exists y : C_w(y) = 1 \Leftrightarrow \exists y, z : \varphi_w(y, z) = 1 \Leftrightarrow \varphi_w \in \mathrm{3Sat}.$$

Hence if 3Sat is in P we can compute $f(w)$ efficiently by deciding if φ_w is satisfiable. **QED**

In sections §4.4 we reduced 3Sat to other problems which are also in NP by Fact 5.4. This implies that all these problems are NP-complete. Here we use that if problem A reduces to B in P, and B reduces to C, then also A reduces to C. This is because if $C \in \mathrm{P}$ then $B \in \mathrm{P}$, and so $A \in \mathrm{P}$.

Corollary 5.14. Clique, Subset-sum, and 3Color are complete for P vs NP.

It is important to note that there is nothing special about the *existence* of complete problems. The following is a simple such problem that does not require any of the machinery in this section.

Exercise 5.15. The Simu problem: Given a word program P, an input x, and $t \in \mathbb{N}$, where t is written in unary, decide if there is $y \in [2]^t$ such that $P(x, y) = 1$ in t steps.

Prove that Simu is complete for P vs NP.

What if t is written in binary?

The interesting aspect of complete problems such as 3Sat and those in Corollary 5.14 is that they are very simple and structured. This makes them suitable for reductions, and for inferring properties of their complexity class which are less evident from a model-based definition.

5.3 From programs to 3Sat in quasi-linear time

The framework in the previous section is useful to relate membership in P of different problems in NP, but it is not suitable for a more fine-grained analysis. For example, under the assumption that 3Sat is in Time(cn) we cannot immediately conclude that other problems in NP are solvable in this time or in about this time. We can only conclude that they are in P. In particular, the complexity of 3Sat cannot be related to that of other central conjectures, such as whether 3Sum is in subquadratic time, Conjecture 4.1.

The culprit is the power loss in reducing word programs to circuits, mentioned at the beginning of the chapter. We now remedy this situation and present a quasi-linear reduction. As we did before, see Theorem 5.1 and Theorem 5.13, we first state a version of the simulation for (deterministic) computation which contains all the main ideas, and then we note that a completeness result follows.

Theorem 5.16. Given an input length $n \in \mathbb{N}$, a time bound $t \in \mathbb{N}$, and a word program P that runs in time t on inputs of n bits, we can compute in time $t' := c_P t \log^c t$ a 3CNF φ on variables (x, y) where $|y| \leq t'$ such that for every $x \in [2]^n$:

$$P(x) = 1 \iff \exists y : \varphi(x, y) = 1.$$

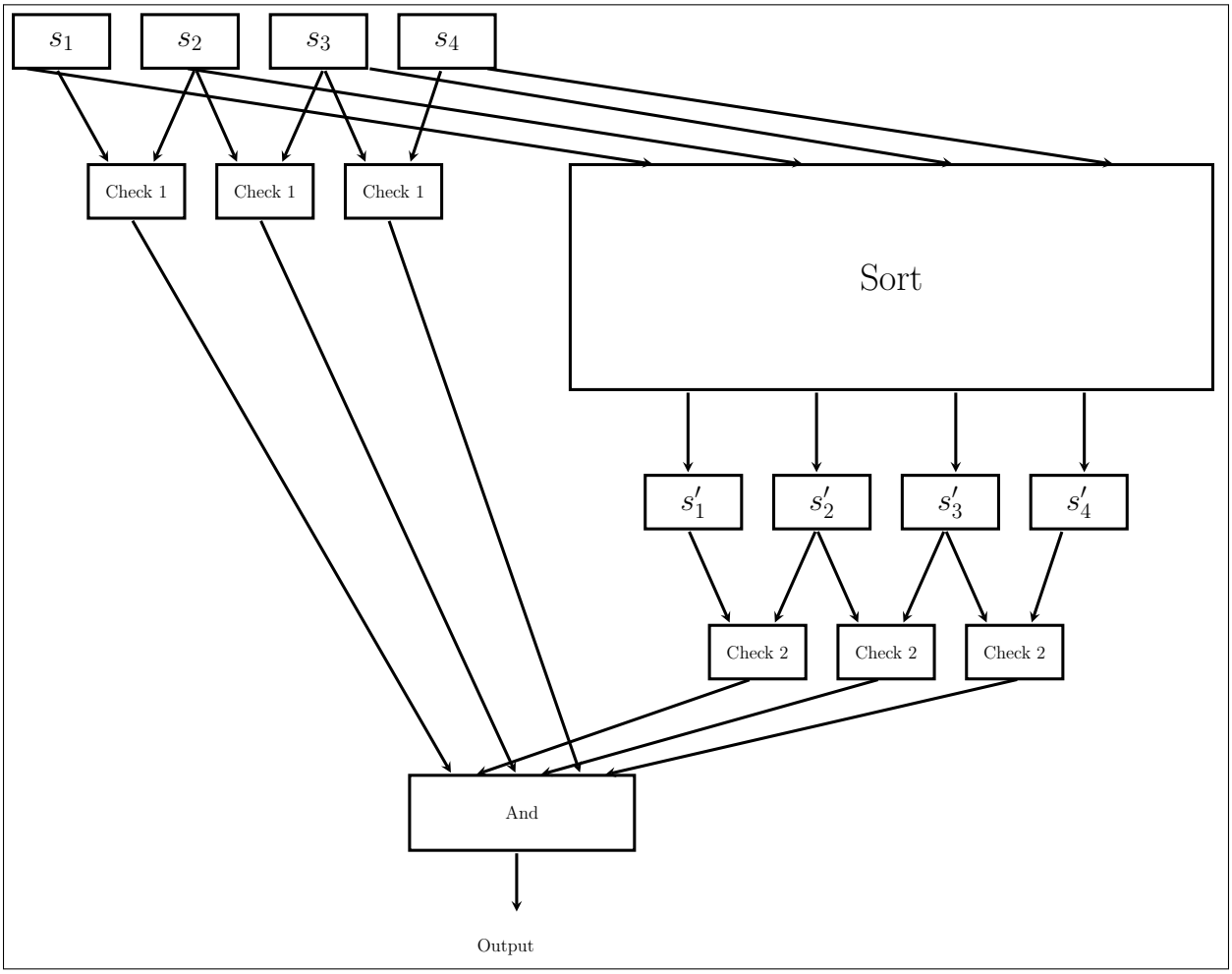


Figure 5.2: Circuit in the proof of Theorem 5.16.

We now present the proof of this amazing result.

At the high level, the approach is like in Theorem 5.1: We are going to *guess* computation and check it efficiently. However, we can't quite guess t configurations (Definition 1.2) of a word program, since each configuration contains the memory as well and so can have size about t. This would result in a quadratic blow-up, which we seek to avoid. Instead, we will guess *internal configurations* which are like configurations *except* that we don't include the entire memory, but only one word in case of Read/Write operations. This is similar to the data we have after fetching memory locations in the simulation of P by circuits, Theorem 2.12.

Definition 5.17. The *internal configuration,* abbreviated IC*,* of a word program specifies:

- the program counter,
- the word length w,
- its registers, and
- if the current instruction is a Read $R[i] := M[R[j]]$ or Write $M[R[j]] := R[i]$ then the IC includes the word $M[R[j]]$.

If a program P runs in time $t \geq n$ on an input of length n, its ICs can be described by $c_P \log t$ bits. Again, this is as in the proof of Theorem 2.12.

As we mentioned, we are going to guess the ICs, and then we need to check them efficiently by a circuit. This is not immediate, since, again, the word program can read and write in memory at arbitrary locations, something which is not easy to do with a circuit. It is convenient to view the checking as two separate checks, only one of which involves memory. If both checks pass, then the computation is correct.

More precisely, a sequence of internal configurations $s_1, s_2, \ldots, s_t$ corresponds to the computation of the program on input x iff for every $i < t$:

1. If s_i does not access memory, then s_{i+1} has its registers, program counter, and word length updated according to the instruction executed in s_i,
2. If s_i is computing a read operation $R[i] := M[R[j]]$ then in s_{i+1} register $R[i]$ contains *the most recent value written in memory* word $R[j]$. In case this word was never written, then $R[i]$ should contain $x_{R[j]}$ if $R[j] \in [n]$, and 0 otherwise. The program counter in s_{i+1} also points to the next instruction.

Rather than directly constructing a 3CNF that implements these checks, we construct a circuit and then appeal to Theorem 5.1. The circuit is illustrated in Figure 5.2. To construct a circuit for Check 1, note that for each i the check between ICs s_i and s_{i+1} can be implemented by a circuit of size $c_P \log^c t$. For example, this circuit may check that $R[0] := R[1] + R[2]$. This can be done via the circuit in Exercise 2.7. And the same holds for other operations. Alternatively, we can argue that these check are in P (note the input length is the length

of two ICs), and then appeal to Theorem 2.12 to obtain a circuit. Taking an And of these circuits over the choices of i gives a circuit of the desired size for Check 1.

The difficulty lies in Check 2, because the circuit needs to find "the most recent value written." The clever solution is to *sort the ICs by the memory address in Read/Write operations*. After sorting, we can implement Check 2 as easily as Check 1, since we just need to check adjacent pairs of ICs.

The emergence of sorting in the theory of NP-completeness cements the pivotal role this operation plays in computer science.

To implement this idea we need to be able to sort the ICs by their memory addresses with a quasi-linear size circuit. Several quasi-linear time sorting algorithms are well known (such as CountingSort, see Example 1.5), but implementing them as quasi-linear-size circuits is not immediate. Still, such sorting circuits exist:

Lemma 5.18. Given t and m we can compute in time $t' := t \cdot (m \log t)^c$ a circuit (of size $\leq t'$) that sorts t integers of m bits.

Because this reduction is so fundamental, for completeness we give a proof of Lemma 5.18 in section §5.3.1.

We summarize the key steps in the proof.

Proof of Theorem 5.16. We construct a circuit C as in Figure 5.2 (for $t = 4$) and then appeal to Theorem 5.1. The extra variables y correspond to t ICs $s_1, s_2, \ldots, s_t$. An IC takes $c_P \log t$ bits to specify, so we need $\leq c_P t \log t$ variables y. The circuit C first performs Check 1 above for each adjacent pair (s_i, s_{i+1}) of ICs. This takes size $c_P \log^c t$ for each pair, and so size $c_P t \log^c t$ overall.

Then C sorts the ICs by memory addresses, producing sorted ICs $s'_1, s'_2, \ldots, s'_t$. This takes size $t \cdot \log^c t$ by Lemma 5.18, using that the memory addresses have $m \leq c \log t$ bits. Then the circuit performs Check 2 for each adjacent pair (s'_i, s'_{i+1}) of ICs. The circuit size required for this is no more than for Check 1.

Finally, the circuit takes an And of the results of the two checks, and also checks that s_t outputs 1.

This concludes the proof, except for some low-level details related to sorting which I discuss now. The sorting Lemma 5.18 refers to integers, whereas as common in algorithms we need to sort *unstructured objects* (the ICs) by *keys* (the memory addresses). However, we can just view each IC as an integer with the memory address written in the most significant digits, to have the same effect. Also, the sorting should be *stable*: The relative order of ICs with the same key should be maintained in the output. This can again be easily accomplished by including a timestamp, i.e., the original order as a number in $[t]$, written next in significance after the memory address. (So if two ICs address the same memory location, their order will be decided based on the timestamp.) This timestamp increases the key by $\leq c \log t$ bits and so can be afforded. **QED**

We can now prove completeness in a manner similar to Theorem 5.13, with a relatively simple extension of Theorem 5.16.

Theorem 5.19. 3Sat is complete for NTime($t(n)$) under map reductions in Time($t \log^c t$), for every time-constructible $t(n) \geq n \log^c n$.

Recall the assumption on t is satisfied by all standard functions – see discussion in section 1.8.2.

Proof. First let us show that 3Sat is in NTime($t(n)$). Since $t(n) \geq n \log^c n$ it is enough to show that 3Sat is in quasilinear-NTime. Consider a 3CNF instance φ of length n. This instance has at most n variables, and we can guess an assignment y to them within our budget of non-deterministic guesses. There remains to verify that y satisfies φ. For this, we can do one pass over the clauses. For each clause, we access the bits in y corresponding to the 3 variables in the clause, and check if the clause is satisfied. This takes constant time per clause, and so time cn overall.

Now let us prove hardness. Let $f \in$ NTime(t) and P be a corresponding word program. Given an input $w \in [2]^n$ we have to compute in time $t \log^c t$ a 3CNF φ such that

$$\exists y \in [2]^{t(n)} : P(x, y) = 1 \iff \exists y \in [2]^{t(n) \log^c t(n)} : \varphi(y) = 1.$$

First we compute $t(n)$, using the assumption. We now apply Theorem 5.16, but on a new input length $n' := c(n + t) \leq ct$, to accommodate for inputs of the form (x, y). This produces a formula φ of size $c_P t \log^c$ in variables (x, y) and new variables z. We can now set the variables x' to the input w to conclude the proof. The running time is dominated by that of the reduction in Theorem 5.16, which is $t \log^c t$. **QED**

We can now give the following quasi-linear version of Fact 5.11. The only extra observation for the proof is again that the composition of two quasi-linear functions is quasi-linear.

Corollary 5.20. 3Sat $\in$ Quasi-Linear-Time $\Leftrightarrow$ Quasi-Linear-NTime = Quasi-Linear-Time.

Exercise 5.21. Prove that Theorem 5.19 holds with 3Color instead of 3Sat using the reduction in section §4.4. Can you obtain a similar result for Clique and Subset-sum using the reductions in section §4.4?

Exercise 5.22. Prove that 3Sum reduces to 3Sat in Subquadratic time.

5.3.1 Efficient sorting circuits

In this section we prove Lemma 5.18. We present an efficient sorting algorithm for an array $A[0..n-1]$ which enjoys the *CE property*: *the only way in which the input is accessed is via* Compare-Exchange *operations*. Compare-Exchange takes two indexes i and j and swaps $A[i]$ and $A[j]$ if they are in the wrong order. It has the following code:

```
Compare-Exchange(Array A[0..n-1] and indexes i and j with i < j):
if A[i] > A[j]
 swap A[i] and A[j]
```

Why care about this property? It makes the comparisons *independent from the input,* and this allows us to implement the algorithm with a network – a *sorting network* – of fixed Compare-Exchange operations. In particular, we will get a circuit.

The algorithm is called CE-Sort. It has a basic recursive structure similar to MergeSort:

```
CE-Sort(A[0..n − 1]):
if n ≥ 1 {
 CE-Sort(A[0..n/2 − 1])
 CE-Sort(A[n/2..n − 1])
 CE-Merge(A[0..n − 1])
}
```

Throughout, we assume that n is a power of 2.

Algorithm CE-Merge(A) merges the two already sorted halves $A[0..n/2-1]$ and $A[n/2-1..n-1]$ of A, resulting in a sorted output sequence. CE-Merge replaces the Merge operation in Mergesort which does not have the CE property. It works in a remarkable and startling way. First it merges the *odd* subsequence of the entire array A, then the *even*, and finally it makes a series of Compare-Exchange operations.

```
CE-Merge(A[0..n − 1]):
if n = 2
 Compare-Exchange(A, 0, 1)
else {
 CE-Merge(A[0, 2, 4, . . . , n − 2]) //the even subsequence
 CE-Merge(A[1, 3, 5, . . . , n − 1]) //the odd subsequence
 for i ∈ {1, 3, 5, 7, . . . , n − 3}
  Compare-Exchange(A, i, i + 1)
}
```

We shall now argue that this algorithm is correct.

Lemma 5.23. If $A[0..n/2-1]$ and $A[n/2..n-1]$ are sorted, then CE-Sort($A[0..n-1]$) outputs a sorted array.

Proof. To prove this lemma we invoke the 0-1 *principle*. This principle says that CE algorithms sort integers correctly iff they sort correctly integers in [2]. In this particular case, we can apply this principle to say that it suffices to prove the lemma when each $A[i] \in [2]$ for every i.

For completeness we prove this principle in now. Let A be an input and B an output produced by a CE algorithm S. Suppose B is not sorted let k be the smallest such that $B[k] > B[k+1]$. Define a function f such that $f(x) = 1$ if $x \geq b_k$ and $f(x) = 0$ otherwise. For an array X let $f(X)$ denote the array obtained by applying f to every entry of X.

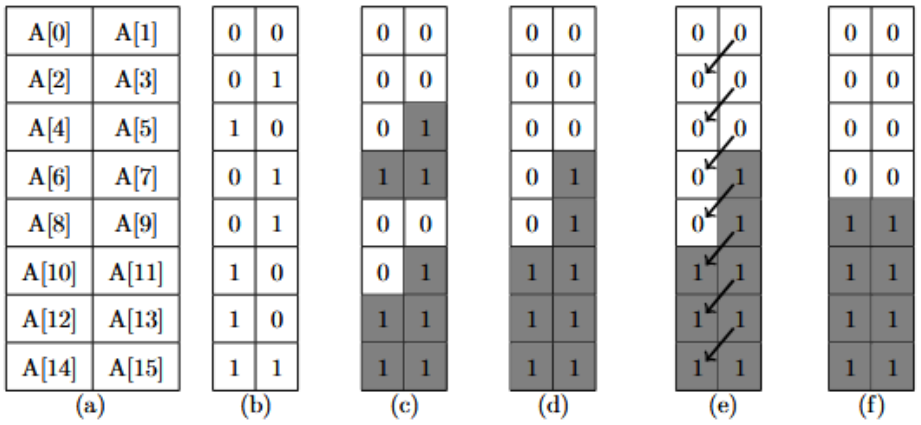


Figure 5.3: CE-Merge.

Observe that $f(B)$ is not sorted. However (see Exercise 5.24) f commutes with any Compare-Exchange operation applied to any sequence X:

$$f(\text{Compare-Exchange}(X, i, j)) = \text{Compare-Exchange}(f(X), i, j). \tag{5.1}$$

If the algorithm S is just a sequence of Compare-Exchange, we have

$$f(B) = f(S(A)) = S(f(A)).$$

So the algorithm fails to correctly merge the 0-1 sequence $f(A)$.

Having established this principle, we now prove the lemma for $A \in [2]^n$ by induction on n, following the recursive definition of CE-Merge. Refer to figure 5.3.

The base case $n = 2$ is clear. Assume that CE-Merge correctly merges any two sorted 0-1 sequences of size $n/2$. We view an input sequence of n elements as an $n/2 \times 2$ matrix, with the left column corresponding to elements at the even-indexed positions $0, 2, \ldots, n-2$ and the right column corresponding to elements at the odd-indexed positions $1, 3, \ldots, n-1$ (Figure 5.3(a)). 5.3(b) shows a corresponding 0-1 input, which we can assume w.l.o.g. because of the zero-one principle. Figure 5.3(c) shows the matrix after the recursive calls to the sorting. Since the upper half of the matrix is sorted by assumption, the right column in the upper half has the same number or exactly one more 1 than the left column in the upper half. The same is true for the lower half. Because each (length-$(n/4)$) column in each half of the matrix is also individually sorted by assumption, the induction hypothesis guarantees that after the two calls to CE-Merge both the left and right (length-$(n/2)$) columns are sorted (Figure 5.3(d)).

At this point only one of 3 cases arises:

1) The odd and even subsequences have the same number of 1s.

2) The odd subsequence has a single 1 more than the even subsequence.

3) The odd subsequence has two 1s more than the even subsequence.

In the first two cases, the sequence is already sorted. In the third case, the Compare-Exchange operations (Figure 5.3(e)) yield a sorted sequence (Figure 5.3(f)). **QED**

Exercise 5.24. Prove Equation (5.1).

To conclude the proof of Lemma 5.18 about sorting circuits, it only remains to argue efficiency. Let $S_M(n)$ denote the number of Compare-Exchange operations for CE-Merge for an input sequence of length n. We have the recurrence

$$S_M(n) = 2 \cdot S_M(n/2) + n/2 - 1,$$

which yields $S_M(n) \leq cn \log n$.

Finally, let $S(n)$ denote the number of calls to Compare-Exchange for CE-Sort on an input sequence of length n. Then we have the recurrence

$$S(n) \leq 2 \cdot S(n/2) + cn \log n$$

which yields $S(n) = cn \cdot \log^2 n$.

To conclude the proof, note that Compare-Exchange for inputs with m bits can be implemented by a circuit of size m^c.

5.4 Power from completeness

The realization that arbitrary computation can be reduced to 3Sat and other problems is powerful and liberating. In particular in this section wee demonstrate that it allows us to significantly widen the net of reductions.

5.4.1 Max-3Sat

The 3Sat asks if all the clauses can be satisfied. The Max-3Sat asks to compute the maximum number of clauses that can be satisfied.

Definition 5.25. The Max-3Sat problem: Given a 3CNF, compute the maximum number of clauses that can be satisfied by an assignment to the variables.

3Sat trivially reduces to Max-3Sat in FP. The converse will be shown next.

Theorem 5.26. Max-3Sat reduces to 3Sat in FP.

Proof. Consider the problem Atleast-3Sat: Given a 3CNF formula and an integer t, is there an assignment that satisfies at least t clauses? This is in NP and so can be reduced to 3Sat in P. This is the step that's not easy without "thinking completeness:" given an algorithm for 3Sat it isn't clear how to use it directly to solve Atleast-3Sat.

Hence, if 3Sat is in P so is Atleast-3Sat. On input a 3CNF ϕ for Max-3Sat, we can use binary search and Atleast-3Sat to find the largest t s.t. $(\phi, t) \in$ Atleast-3Sat. We then output t. **QED**

Having found the maximum, one can also compute a corresponding satisfying assignment fixing one variable at the time as in the proof of Theorem 4.29.

5.4.2 NP is as easy as detecting unique solutions

In this section we present a beautiful randomized reduction. A satisfiable 3CNF can have multiple satisfying assignments. On the other hand some problems and puzzles have unique solutions. It is natural to ask if the hardness of 3Sat stems from multiple assignments – perhaps a unique satisfying assignment would be easy to find. In this section we show that, in fact, deciding the satisfiability of 3CNFs with a unique satisfying assignment is no easier, if randomized reductions are allowed.

We will be working with satisfiability of general circuits, and infer the result for 3CNFs as a consequence via completeness. We now define these problems and then state the main result.

Definition 5.27. The Unique-CktSat problem: Given a circuit C s.t. there is at most one input x for which $C(x) = 1$, decide if such an input exists.

Unique-3Sat is the Unique-CktSat problem restricted to 3CNF circuits.

Theorem 5.28. 3Sat reduces to Unique-3Sat in BPP.

The proof is another example of the power of randomness, and introduces an important paradigm in randomized computing: *hashing*. We will use hash functions to "isolate" assignments. First we define the type of hash functions we will need, then we construct them, and finally we state and prove the isolation lemma.

Definition 5.29. A distribution H on functions mapping $S \to T$ is called *pairwise uniform* if for every distinct $x, x' \in S$ and every $y, y' \in T$ one has

$$\mathbb{P}_H[H(x) = y \wedge H(x') = y'] = 1/|T|^2.$$

This is saying that on every pair of distinct inputs H behaves like a uniform function. Yet unlike completely uniform functions, the next lemma shows that pairwise uniform functions can have a short description, which makes them suitable for use in algorithms.

Lemma 5.30. Let $H : [2]^n \to [2]^t$ be the random function $H(x) := Ax + B$ where A is a uniform $n \times t$ matrix over $\mathbb{F}_2$ and B is a uniform vector in $\mathbb{F}_2^t$. Then H is pairwise uniform.

Exercise 5.31. Prove that the lemma follows from the lemma with $t = 1$.

Proof. Let $t = 1$ and consider any $x \neq x'$. Let $i \in [n]$ be a coordinate s.t. $x_i \neq x'_i$. We aim prove that the pair $(H(x), H(x'))$ is uniformly distributed over $\mathbb{F}_2^2$. In fact, we will prove the stronger result that this is true even for any fixing of all the coordinates in A except i.

Assuming without loss of generality that $x_i = 0$ (and $x'_i = 1$), we have that

$$(H(x), H(x')) = \left(\sum_{j \neq i} A_j x_j + B, \sum_{j \neq i} A_j x'_j + A_i x'_i + B \right) = (a + B, b + A_i + B)$$

where a and b are fixed values in $\mathbb{F}_2$. This distribution is uniform over $\mathbb{F}_2^2$ iff the distribution $(B, A_i + B)$ is; and the latter is indeed uniform. **QED**

There are constructions of pairwise-uniform with smaller support, see Problem 5.1.

The next result shows how we can use pairwise uniformity to "isolate" an element.

Lemma 5.32. [Isolation] Let H be a pairwise uniform function mapping $S \to T$, and let $1 \in T$. The probability that there is a unique element $s \in S$ such that $H(s) = 1$ is

$$\geq \frac{|S|}{|T|} - \frac{|S|^2}{|T|^2}.$$

In particular, if $|T|/8 \leq |S| \leq |T|/4$ this prob. is $\geq \frac{1}{8} - \frac{1}{16} \geq 1/8$.

Proof. For fixed $s \in S$, the probability that s is the unique element mapped to 1 is at least the prob. that s is mapped to 1 minus the prob. that both s and some other $s' \neq s$ are mapped to 1. This is

$$\geq \frac{1}{|T|} - \frac{|S| - 1}{|T|^2}.$$

These events for different $s \in S$ are disjoint; so the target probability is at least the sum of the above over $s \in S$. **QED**

We can now present the reduction from 3Sat to Unique-3Sat.

Proof of Theorem 5.28. Given a 3Sat instance ϕ with $\leq n$ variables x, pick a uniform $i \in [n + c]$. We then sample a pairwise uniform function mapping $[2]^n$ to $[2]^i$, say from Lemma 5.30, and consider a circuit computing

$$C(x) := \phi(x) \wedge H(x) = 0^i.$$

Computing H as in Lemma 5.30 yields a circuit of size n^c.

If ϕ is not satisfiable, C is not satisfiable, for any choice for H.

Now suppose that ϕ has $s \geq 1$ satisfying assignments. With prob. $\geq c/n$ over the choice of i we will have $2^{i-3} \leq s \leq 2^{i-2}$, in which case Lemma 5.32 guarantees that C has a unique satisfying assignment with prob. $\geq c$ over the choice of H. Overall, a Unique-CktSat algorithm on C outputs 1 with prob. $\geq c/n$. By the same analysis of section §3.1, if we repeat this process cn times, with independent random choices, the Or of the outcomes gives the correct answer with prob. $\geq 2/3$. **QED**

Exercise 5.33. Conclude the proof reducing Unique-CktSat to Unique-3Sat.

5.5 Alternation

We placed one quantifier "in front" of computation and got something interesting: NP. So let's push the envelope and place more. As we will see the corresponding classes turn out to be extremely useful, with deep ties to impossibility results, the P vs. BPP question, circuits, and much more. The proofs of several results that do not *prima facie* involve multiple quantifiers excurse into multiple quantifiers. Examples include new reductions to 3Sat (Exercise 5.38), impossibility results for 3Sat established in Theorem 6.49 and Theorem 6.53, and P = NP ⇒ P = BPP (see Theorem 5.8 and Exercise 5.39). The first of these will be proved in this chapter.

The next definition is like the Definition 5.3 of NTime, except there are multiple quantifiers.

Definition 5.34. $\Sigma_i\mathrm{Time}(t(n))$ is the set of functions $f : X \subseteq [2]^* \to [2]$ for which there is a word program P that for every $x \in X$ of length $\geq |P|$ satisfies:
- $f(x) = 1$ iff $\exists y_1 \forall y_2 \exists y_3 \forall y_4 \ldots Q_i y_i \in [2]^{t(|x|)} : P(x, y_1, y_2, \ldots, y_i) = 1$, and
- $P(x, y_1, y_2, \ldots, y_i)$ stops within $t(n)$ steps for every $y_1, y_2, \ldots, y_i \in [2]^{t(n)}$.

$\Pi_i\mathrm{Time}(t(n))$ is defined similarly except that we start with a $\forall$ quantifier. We also define

$$\Sigma_i\mathrm{P} := \bigcup_d \Sigma_i\mathrm{Time}(n^d),$$

$$\Pi_i\mathrm{P} := \bigcup_d \Pi_i\mathrm{Time}(n^d), \text{ and}$$

$$\text{the Power Hierarchy, PH} := \bigcup_i \Sigma_i\mathrm{P} = \bigcup_i \Pi_i\mathrm{P}.$$

We refer to such computation and the corresponding programs as *alternating,* since they involve an alternation of quantifiers. The PH is the power-time analogue of the older arithmetical hierarchy from computability theory or logic in which nondeterministic time plays the role of listable (a.k.a. computably enumerable, etc.).

Similarly to NP, we have PH ⊆ Exp. We leave this as an exercise; we'll prove a stronger fact later, see Theorem 6.6.

Exercise 5.35. The Min-Ckt problem: Given a circuit C, decide if there is no smaller circuit equivalent to C. Prove that Min-Ckt is in Π_2P.

The following key result says that alternations can simulate randomness. More precisely, two simulations. The first optimizes the number of quantifiers, the second the time. This should be contrasted with various *conditional* results (such as Theorem 3.15) suggesting that in fact a quasilinear deterministic simulation (with no quantifiers) is possible.

Theorem 5.36. BPP ⊆ PH.

More in detail, for every function t we have:
(1) $\mathrm{BPTime}(t) \subseteq \Sigma_2\mathrm{Time}(t^2 \log^c t)$, and
(2) $\mathrm{BPTime}(t) \subseteq \Sigma_3\mathrm{Time}(t \log^c t)$.

The proof is best understood in terms of constant-depth circuits and is therefore postponed to Chapter 9.

5.5.1 Does the hierarchy collapse?

We refer to the event that $\exists i : \Sigma_i \mathrm{P} = \mathrm{PH}$ as "the PH collapses." It is unknown if the PH collapses. A common belief appears to be that it does not. According to this belief, statements of the type

$$X \Rightarrow \text{PH collapses}$$

constitute evidence that X is false. Examples of such statements are discussed next.

Theorem 5.37. $\mathrm{P} = \mathrm{NP} \Rightarrow \mathrm{P} = \mathrm{PH}$.

The idea in the proof is simply that if you can remove one quantifier then you can remove more.

Proof. We prove by induction on i that $\Sigma_i \mathrm{P} \bigcup \Pi_i \mathrm{P} = \mathrm{P}$.

The base case $i = 1$ follows by assumption and the fact that P is closed under complement.

Next we do the induction step. We assume the conclusion is true for i and prove it for $i + 1$. We will show $\Sigma_{i+1}\mathrm{P} = \mathrm{P}$. The result about $\Pi_{i+1}\mathrm{P}$ follows again by complementing.

Let $f \in \sum_{i+1} \mathrm{P}$, so $\exists a \in \mathbb{N}$ and a power-time program P such that for any input x of length n:

$$f(x) = 1 \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \forall y_4 \ldots Q_{i+1} y_{i+1} \in [2]^{n^a} : P(x, y_1, y_2, \ldots, y_{i+1}) = 1.$$

(As discussed after Definition 5.34 we don't need to distinguish between time as a function of $|x|$ or of $|(x, y_1, y_2, \ldots, y_{i+1})|$ when considering power times as we are doing now.)

Now the creative step of the proof is to consider f' defined as

$$f'(x, y_1) = 1 \iff \forall y_2 \in [2]^{n^a} \ldots Q_{i+1} y_{i+1} \in [2]^{n^a} : P(x, y_1, y_2, \ldots, y_{i+1}) = 1.$$

Note $f' \in \Pi_i \mathrm{P}$. By induction hypothesis $f' \in \mathrm{P}$. So let P' compute f' in power time. We have $f(x) = 1 \iff \exists y_1 \in [2]^{n^a} : P'(x, y_1) = 1$. And so $f \in \mathrm{NP} = \mathrm{P}$, again using the hypothesis. **QED**

Exercise 5.38. Prove that Min-Ckt reduces to 3Sat in P.

Exercise 5.39. Prove that $\mathrm{P} = \mathrm{NP} \Rightarrow \mathrm{P} = \mathrm{BPP}$ (Theorem 5.8).

Exercise 5.40. Prove the following variant of Theorem 5.37: Suppose $\forall \epsilon > 0\ \mathrm{NTime}(n^{1+\epsilon}) \subseteq \mathrm{Time}(n^{1+2\epsilon})$. Then $\Sigma_i \mathrm{Time}(n^{1+\epsilon}) \subseteq \mathrm{Time}(n^{1+4^i\epsilon})$ for every $i \in \mathbb{N}, \epsilon > 0$.

We have shown that if you can remove one quantifier you can remove all of them – Theorem 5.37. Next we show that if you can *swap* a quantifier you can swap and then collapse all of them. For later uses we prove this in the general case where the swap happens at some level of the hiearchy – a similar extension of Theorem 5.37 holds as well.

Theorem 5.41. $\Sigma_i\mathrm{P} = \Pi_i\mathrm{P} \Rightarrow \mathrm{PH} = \Sigma_i\mathrm{P}$.

Proof. We prove that for every $j \geq i$ we have $\Sigma_j\mathrm{P} \bigcup \Pi_j\mathrm{P} = \Sigma_i\mathrm{P} \bigcap \Pi_i\mathrm{P}$. We proceed by induction on j. The base base $j = i$ is given by hypothesis. For the inductive step let $f \in \sum_{j+1}\mathrm{P}$, the case $f \in \Pi_{j+1}\mathrm{P}$ being symmetrical. By definition, $\exists a \in \mathbb{N}$ and a power-time program P such that for any input x of length n,

$$f(x) = 1 \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \forall y_4 \ldots Q_{j+1} y_{j+1} \in [2]^{n^a} : P(x, y_1, y_2, \ldots, y_{j+1}) = 1.$$

Similarly to the proof of Theorem 5.37 consider

$$f'(x, y_1) = 1 \iff \forall y_2 \in [2]^{n^a} \ldots Q_{i+1} y_{i+1} \in [2]^{n^a} : M(x, y_1, y_2, \ldots, y_{i+1}) = 1.$$

Note $f' \in \Pi_j\mathrm{P}$. By induction hypothesis $f' \in \Sigma_i\mathrm{P}$. So there is a power-time program P' and $a' \in \mathbb{N}$ such that

$$f(x, y_1) = 1 \Leftrightarrow \exists z_1 \forall z_2 \exists z_3 \forall z_4 \ldots Q_i z_i \in [2]^{n^{a'}} : P'(x, y_1, z_1, z_2, \ldots, z_i) = 1.$$

W.l.o.g. $a' \geq a$. Hence we have

$$f(x) = 1 \Leftrightarrow \exists y_1 \exists z_1 \forall z_2 \exists z_3 \forall z_4 \ldots Q_i z_i \in [2]^{n^{a'}} : P'(x, y_1, z_1, z_2, \ldots, z_i) = 1.$$

We can now merge or collapse the two quantifiers $\exists y_1$ and $\exists z_1$ into a single one, which shows that $f \in \Sigma_i\mathrm{P} = \Pi_i\mathrm{P}$, where the equality holds by assumption. **QED**

It is not known if NP has power-size circuits. However, the following result shows that if it does then the PH collapses. This is an interesting connection between non-uniform and uniform computational models.

Theorem 5.42. $\mathrm{NP} \subseteq \mathrm{CktP} \Rightarrow \mathrm{PH} = \Sigma_2\mathrm{P}$.

Proof. We'll show $\Pi_2\mathrm{P} \subseteq \Sigma_2\mathrm{P}$ and then appeal to Theorem 5.41. Let $f \in \Pi_2\mathrm{Time}(n^d)$ and P be a corresponding program s.t.

$$f(x) = 1 \Leftrightarrow \forall y_1 \in [2]^{n^d} \exists y_2 \in [2]^{n^d} : P(x, y_1, y_2) = 1.$$

We claim the following equivalent expression for the right-hand side:

$$\forall y_1 \in [2]^{n^d} \exists y_2 \in [2]^{n^d} : P(x, y_1, y_2) = 1 \Leftrightarrow \exists C \, \forall y_1 \in [2]^{n^d} : P(x, y_1, C(x, y_1)) = 1,$$

where C ranges over circuits of size $|x|^{d'}$ for some d'. If the equivalence is established the result follows, since evaluating a circuit can be done in power time.

To prove the equivalence, first note that the $\Leftarrow$ direction is obvious, by setting $y_2 := C(x, y_1)$. The interesting direction is the $\Rightarrow$. Consider the problem Search-CktSat which is like Search-3Sat but for general circuits rather than 3CNFs. Now, because CktSat is in NP, by assumption it has power-size circuits. By the reduction in Theorem 4.29, Search-CktSat has power-size circuits S as well. Hence, the desired circuit C may, on input x and y_1 produce a new circuit W mapping an input y_2 to $P(x, y_1, y_2)$ (as in Theorem 2.12), and run S on W to produce y_2. **QED**

Exercise 5.43. Prove that $\mathrm{PH} \not\subseteq \mathrm{CktSize}(n^k)$, for any $k \in \mathbb{N}$. (Hint: Theorem 2.13.)

Improve this to $\Sigma_2\mathrm{P} \not\subseteq \mathrm{CktSize}(n^k)$.

5.6 Problems

Problem 5.1. Let $\mathbb{F}_q$ be a finite field. Define the random function $H : \mathbb{F}_q \to \mathbb{F}_q$ as $H(x) := Ax + B$ where A, B are uniform in $\mathbb{F}_q$. Prove that H is pairwise uniform. Explain how to use H to obtain a pairwise uniform function from $[2]^n$ to $[2]^t$ for any given $t \leq n$.

Problem 5.2. In this problem you'll prove non-deterministic time hierarchies. We focus on computation with one quantifier only, but similar results hold for any level of the hierarchy. Note we don't even need $t(n)$ to be time-constructible.

(1) Prove $\Sigma_1\text{Time}(t(n)) \not\subseteq \Pi_1\text{Time}(ct(n))$, for any $t(n) \geq n$. Hint: Follow the Time Hierarchy Theorem 1.30.

(2) Prove $\text{NTime}(ct(n+1)) \not\subseteq \text{NTime}(t(n))$, for any non-decreasing $t(n) \geq n$. Hint: Follow the randomized Time Hierarchy Theorem 3.17.

Problem 5.3. Prove $\text{Exp} \subseteq \text{CktP} \Rightarrow \text{Exp} = \Sigma_2\text{P}$.

Problem 5.4. [Arithmetic on truth tables] Let $f : [2]^* \to [2]$ be a function. Denote by $a_f(n)$ the 2^n-bit integer whose binary representation are the evaluations of f on every input of n bits. For example,

$$a_f(2) = f(11)f(10)f(01)f(00).$$

Define f' to be the function s.t. $a_{f'}(n) = a_f(n) + 1 \mod 2^{2^n}$, for every n.

Show that if $f \in \text{PH}$ then $f' \in \text{PH}$.

5.7 Notes

NP-completeness and Theorem 5.13 originates in the fundamental works [93, 228]. The first paper proves a version of Theorem 5.1 for tape machines, for a more recent and similar exposition see [319]. Theorem 5.16 is from [153, 295]. The first work focuses on an equivalence between computational models, while the second explicitly constructs a 3CNF formula. We presented the proof in a slightly different way, using the sorting circuits from [47] and following the exposition in [265].

The reduction to unique-3Sat, Theorem 5.28, is from [353] from which we borrowed the section title which, interestingly, emphasizes how easy NP could be, see Chapter 18.

Pairwise uniformity was studied as least since [286] and [78]. For background see [347, 169].

The PH was identified in [326], where Theorem 5.37 is also proved. Theorem 5.42 is from [207].

For a compendium of problems complete for various levels of the PH, in the style of [126], see [303].

The first item in the simulation of BPP Theorem 5.36 is from [318]; the second is from [364].

Chapter 6

Space

Time is only one of many resources we wish to bound. Another important one is *space*, which is another name for the *memory* of the machine. Computing under space constraints could be slightly less familiar to us than computing under time constraints, and many surprises lay ahead in this chapter that challenge our intuition of space-efficient computation.

We shall consider both space bounds bigger than the input length and smaller. For the latter, we allow the program to *read* the input words, but not *write* on them. We also want to compute functions f whose output is more than 1 bit. One option is to have some memory words which are *write-only*. I prefer instead to reduce this to boolean functions by requiring that given x, i output bit i of $f(x)$ is computable efficiently. To make sense of this we'll also require that we can decide if i is out of range, i.e., $i \geq f(x)$. Given this choice, there isn't much use in writing the output in memory, so we'll simply compute it in a register. Finally, we will follow the custom of measuring space in *bits*; measuring in terms of words would also be natural, as discussed right after the definition.

With this in mind, we give the following space analogue of Definition 1.7 of time-bounded computation.

Definition 6.1. We say that a word program P computes $y \in [2]$ on input $x \in [2]^*$ in space s if for some t it computes y into register $R[0]$ in time t as in Definition 1.7 (except the output is written in $R[0]$), and in addition:

- P never writes in words $0..n-1$.
- The total number of bits in the memory words used by P, excluding the input, is $\leq s$. In other words, every configuration (p, m, w, R, M) occurring in the computation has $(m-n)w \leq s$.

We now define the corresponding classes.

Definition 6.2. We denote by $\mathrm{Space}(s(n))$ the set of functions $f : X \subseteq [2]^* \to [2]$ for which there is a word program P that for every $x \in X$ of length $\geq |P|$ computes $f(x)$ in space $s(|x|)$.

We define:

$$\text{logarithmic space, L} := \bigcup_d \text{Space}(d \log n),$$
$$\text{power space, PSpace} := \bigcup_d \text{Space}(n^d).$$

We denote by $\text{FSpace}(s(n))$ the set of functions $f : X \subseteq [2]^* \to [2]^*$ s.t.

1. given $x \in X$ and $i < |f(x)|$ computing bit i of $f(x)$ is in $\text{Space}(s(n))$,
2. given $x \in X$ and i computing $[i < f(x)]$ is in $\text{Space}(s(n))$, and
3. $|f(x)| \leq 2^{s(|x|)}$ for every $x \in X$.

FL and FPspace are defined similarly.

Note that functions in FL are restricted to have power-length output (like functions in FP have). We did not define FSpace for relations, only functions, as that will be sufficient.

The letter L stands for *logarithmic* which is the number of *bits* of space. If we measure space in words, the number is *constant*. Such programs do not need to use memory at all, they can simulate it using registers only:

L is what's computable *without writing in memory, only in registers.*

It is not clear how to make sense of sub-logarithmic space on word programs. But a surprisingly rich theory will be presented over a more restricted model in section §16.7.

Exercise 6.3. Show that the complexity class L does not change if we drop multiplication ($*$) and modular remainder ($\%$) from the instruction set of word programs (Definition 1.1).

As for time, a central notion is that of a *configuration* (Definition 1.2). Once a program P and an input x are fixed, for space-bounded computation we do not need to include the read-only memory words $M[0..|x|-1]$, since they cannot change. This leads to an accurate count of the number of possible configurations which will be useful several times starting now. We investigate next the relationship between space and time. We begin with some basic simulations; a slight improvement to 1. is known, see the notes.

Theorem 6.4. For every functions $t = t(n) \geq n$ and $s = s(n) \geq \log n$:

1. $\text{Time}(t) \subseteq \text{Space}(ct \log t)$,
2. $\text{Space}(s) \subseteq \bigcup_{a \in \mathbb{N}} \text{Time}(a^s)$.

Proof. 1. The space-bounded simulation will first copy the input in read-write memory words, and then simulate the time-bounded computation. Because in time t we can only allocate t extra memory words, and $t(n) \geq n$, we only need ct words of space, for a total of $\leq ct \log t$ bits. (As in Claim 1.15, one detail is that the time-bounded and space-bounded programs start off with different word size; the solution is the same as before.)

2. Let P be a word program computing an output from an input x using space s. To describe a configuration we need the contents of all the read/write memory words, which by assumption is $\leq s$ bits. We also need the program counter, the word length, and the contents of the registers. This is at most c_P times the maximum word length. The word length is always $\leq c \log(m)$ where m is the total number of memory words, and $m \leq n + s$. In total, we need $\leq c_P(s + \log(n + s))$ bits to write down a configuration. When $s \geq \log n$ this is $\leq c_P s$ bits, for a total of $\leq c_P^s$ configurations. Since the machine ultimately stops on valid inputs, configurations cannot repeat, hence the same machine (with no modification) will stop in the desired time. **QED**

Exercise 6.5. Show FL $\subseteq$ FP.

Similar results hold for alternating time as well. For simplicity we only state it for PH.

Theorem 6.6. PH $\subseteq$ PSpace.

The proof is our first example of a general philosophy:

Unlike time, space can be reused.

Proof. Let us first illustrate why NP $\subseteq$ PSpace. So let $f \in$ NP and P be a corresponding program as in De 5.3. On input x, we have to determine if there is $y \in [2]^{n^a} : P(x, y) = 1$, for a constant a depending on P. We shall enumerate over all choices of y, run $P(x, y)$ on all of them, and accept if any run outputs 1. The key point is that we can *reuse* the same space for different runs.

To prove the more general result in the theorem statement, PH $\subseteq$ PSpace, one can proceed by induction similar to Theorem 5.37. The details are left as exercise. **QED**

Gathering this information, we have:

$$\text{L} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PH} \subseteq \text{PSpace} \subseteq \text{Exp}.$$

Just like for Time, for space one has universal programs and a hierarchy theorem. The proofs are very similar to the ones in Chapter 1 and are omitted. The main observation is that the universal program in Lemma 1.18 is space efficient, which can be verified by inspection. The hierarchy theorem implies L $\neq$ PSpace. Hence, analogously to the situation for Time and NTime (section §5.1), we know that at least one of the inclusions above between L and PSpace is strict. Most people seem to think that all are strict, but nobody can prove that any specific one is.

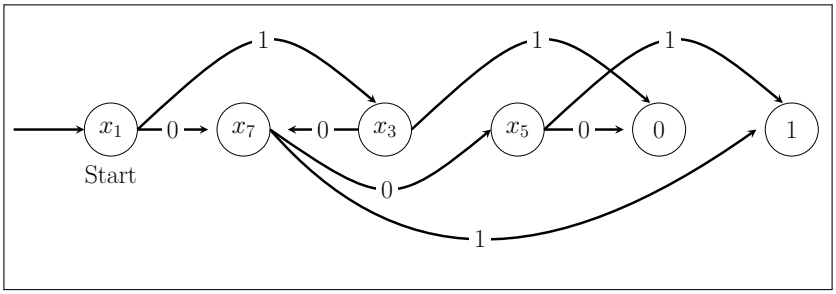


Figure 6.1: Illustration of branching program (Definition 6.7).

6.1 Branching programs

Branching programs are the non-uniform counterpart of Space, just like circuits are the non-uniform counterpart of Time.

Definition 6.7. A *branching program* is a directed graph. Each node is labeled by either an input variable, in which case it has two outgoing edges labeled 0 and 1, or else it is labeled with a 0 or a 1, in which case it has no outgoing edges. One special node is the *start* node. A branching program computes a function $f : X \to [2]$ if for every $x \in X$, starting from the start node and following edge labels corresponding to x we reach $f(x) \in [2]$. The *size* of the program is the number of nodes, the *space* is the logarithm of the size.

We denote by $\text{BrSize}(s(n))$ the set of functions $f : X \subseteq [2]^* \to [2]^*$ s.t. there is n_0 and for every $n \geq n_0$ there are m branching programs b_i s.t. $f(x) = (b_0(x), b_1(x), \ldots, b_{m-1}(x))$, and the sum of the sizes of the b_i is $\leq s(n)$.

Finally, $\text{BrL} := \bigcup_{d \in \mathbb{N}} \text{BrSize}(n^d)$.

See the notes for variants of this definition.

Just like CktP is the non-uniform analogue of P in that circuits can simulate word programs (Theorem 2.12), BrL is the non-uniform analogue of L in that branching programs can simulate space-bounded computation.

Theorem 6.8. $\text{L} \subseteq \text{BrL}$.

Proof. Each node in the program corresponds to a configuration – that's the 1-line proof.

Let us add some details. Let P be a word program for f that runs in space $s(n) = a \log(n)$ for a constant a. On inputs of length n, we create a branching program B whose nodes are the configurations of P as described in the proof of Theorem 6.4. The number of configurations is $\leq c_P^s$, so the space of B is as desired.

The start node of B is the start configuration of P.

Configurations with instruction STOP are labeled 0 or 1 depending on the value of $R[0]$ (which recall from Definition 6.2 contains the output) , and have no outgoing edges.

Configurations with instruction $R[i] := M[R[j]]$ with $R[j] \in [n]$, i.e., reading an input bit, query bit $R[j]$ of the input, and have two outgoing edges, leading to configurations where $R[i]$ is modified accordingly.

The other configurations don't need to query any input variable and can simply proceed to the next configuration (or can be shortcut altogether). **QED**

A similar result holds for FL, as long as the output length is bounded by a power. We leave this formulation as an exercise.

Definition 6.9. The branching program given by Theorem 6.8 is called the *configuration graph* of P on inputs of length n.

A somewhat space-efficient simulation of circuits by branching programs is known (unlike for Time, see Theorem 6.4 and the notes). Essentially, circuits of size s can be simulated by branching programs using space $\sqrt{s}$.

Theorem 6.10. $\text{CktSize}(s(n)) \subseteq \text{BrSize}(c^{\sqrt{s(n)\log s(n)}})$.

See section 7.8.1 for the proof.

6.2 The power of L

Computing without memory, as in L, seems quite difficult. It turns out that L is a powerful class capable of amazing computational feats that challenge our intuition of efficient computation. Moreover, these computational feats hinge on far-reaching mathematical techniques. To set the stage, we begin with a composition result. In the previous sections we used several times the intuitive fact that the composition of two maps in FP is also in FP, Claim 1.15. This is useful as it allows us to break a complicated algorithm in small steps to be analyzed separately – which is a version of the *divide et impera* paradigm. A similar composition result holds and is useful for space, but the argument is somewhat less obvious.

Lemma 6.11. Let $f_1, f_2 \in \text{FL}$. Suppose $|f_1(x)| \leq |x|^a$ for a constant a. Then $f_2 \circ f_1 \in \text{FL}$.

The basic idea is to run the program for f_2, but whenever it reads a bit from the input, run the program for f_1 to compute this bit "on the fly," reusing the same space for each computation of f_1.

Proof. We are given as input x and i and we wish to compute bit i of $f_2(f_1(x))$. First we determine $|f_1(x)|$. This is done by running the program that decides if $|f_1(x)| < j$ for $j = 1, 2, \ldots$. Each run of f_1 re-uses the same registers. (Let us discuss some details of this step. Note we can't write down $|f_1(x)|$ in a register, but by our assumption on $|f_1(x)|$, we only need $\leq c_a \log n$ bits to represent j. Hence we shall use c_a registers to store j, as in Claim 1.21. Also, when j becomes larger than i, the starting configuration on input (x, j) may have larger word length than the starting configuration on our input, which is (x, i). To address this, we use two registers to simulate one register in the program for f_1. The

important thing is that each run of f_1 re-uses the same registers, so we can run all these simulations for various j using c_a registers. At the end, we have computed $|f_1(x)|$, in c_a registers.)

Then we simulate the program for f_2 as if it was run on input $(f_1(x), i)$. Recall the starting configuration has $|f_1(x)|$, which we computed above. Here we again use c_a registers for each register in the program for f_2, as above. Whenever the program for f_2 reads bit j from $f_1(x)$, we supply it "on the fly" by simulating f_1 on input (x, j), re-using the same space for each evaluation of f_1. **QED**

6.2.1 Arithmetic

A first example of the power of L is given by its ability to perform basic arithmetic. Grade school algorithms use a lot of space, for example they employ space $\geq n$ to multiply two n-bit integers.

Theorem 6.12. The following arithmetic problems are in FL:

1. Addition of two naturals.
2. Iterated addition: Addition of any number of naturals.
3. Multiplication of two naturals.
4. Iterated multiplication: Multiplication of any number of naturals.
5. Division of two naturals.

Iterated multiplication is of particular interest because it can be used to compute "pseudorandom functions." Such objects shed light on our ability to prove impossibility results via the "Natural Proofs" connection which we will see in Chapter 17.

Proof of 1. in Theorem 6.12. We are given as input $x, y \in [2]^*$ and an index i and need to compute bit i of $x + y$. Starting from the least significant bits, we add the bits of x and y, storing the carry of 1 bit in memory. Output bits are discarded until we reach bit i, which is output. We only need memory for pointers to x and y, and the carry. Recall from Definition 6.2 we also need to decide if i is less than the output length $|x + y|$. This can be done in a similar way. We do a pass on the input keeping track of the carry and determine how many output bits are needed. **QED**

Exercise 6.13. Prove 2. and 3. in Theorem 6.12.

Proving 4. and 5. is more involved and requires some of those far-reaching mathematical techniques we alluded to before. Division can be reduced to iterated multiplication. In a nutshell, the idea is to use the expansion

$$\frac{1}{x} = \sum_{i \geq 0} (-1)^i (x-1)^i.$$

We omit details about bounding the error. Instead, we point out that this requires computing powers $(x-1)^i$ which is an example of iterated multiplication (and in fact is no easier). So for the rest of this section we focus on iterated multiplication. Our main tool for this is the *remaindering representation* of integers, abbreviated RR.

Theorem 6.14. Let $p_1, ..., p_\ell$ be distinct primes and $m := \prod_i p_i$. Then $\mathbb{Z}_m$ is isomorphic to $\mathbb{Z}_{p_1} \times \ldots \times \mathbb{Z}_{p_\ell}$.

The forward direction of the isomorphism is given by the map

$$x \in \mathbb{Z}_m \to (x \mod p_1, x \mod p_2, ..., x \mod p_\ell) \in \mathbb{Z}_{p_1} \times ... \times \mathbb{Z}_{p_\ell}.$$

For the converse direction, there exist integers $e_1, ..., e_\ell \leq m^c$, depending only on the p_i such that the converse direction is given by the map

$$(x \mod p_1, x \mod p_2, ..., x \mod p_\ell) \in \mathbb{Z}_{p_1} \times ... \times \mathbb{Z}_{p_\ell} \to x := \sum_{i=1}^{\ell} e_i \cdot (x \mod p_i).$$

Each integer e_i is 0 mod p_j for $j \neq i$ and is 1 mod p_i.

Example 6.15. $\mathbb{Z}_6$ is isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_3$. The equation $2 + 3 = 5$ corresponds to $(0, 2) + (1, 0) = (1, 2)$. The equation $2 \cdot 3 = 6$ corresponds to $(0, 2) + (1, 0) = (0, 0)$. Note how addition and multiplication in RR are performed in each coordinate separately; how convenient.

To compute iterated multiplication the idea is to move to RR, perform the multiplications there, and then move back to standard representation. A critical point is that each coordinate in the RR has a representation of only $c \log n$ bits, which makes it easy to perform iterated multiplication one multiplication at the time, since we can afford to write down intermediate products.

The algorithm is as follows:

Computing the product of input integers $x_1, \ldots, x_t$.

1. Let $\ell := n^3$ and compute the first ℓ prime numbers $p_1, p_2, \ldots, p_\ell$.
2. Convert the input into RR: Compute $(x_1 \mod p_1, \ldots, x_1 \mod p_\ell), \ldots, (x_t \mod p_1, \ldots, x_t \mod p_\ell)$.
3. Compute the multiplications in RR: $(\Pi_{i=1}^t x_i \mod p_1), \ldots, (\Pi_{i=1}^t x_i \mod p_\ell)$.
4. Convert back to standard representation.

Exercise 6.16. Prove the correctness of this algorithm.

Now we explain how steps 1, 2, and 3 can be implemented in FL. Step 4 can be implemented in FL too, but showing this is somewhat technical due to the computation of the numbers e_i in Theorem 6.14. However these numbers only depend on the input length, and so we will be able to give a self-contained proof that iterated multiplication is in BrL. The composition of the steps is then in FL by Lemma 6.11.

Step 1

By Lemma 3.11, the primes p_i have magnitude $\leq n^c$ and so can be represented with $c \log n$ bits. We can enumerate over integers with $\leq c \log n$ bits in logarithmic space. For each integer x we can test if it's prime by again enumerating over all integers y and z with $\leq c \log n$ bits and checking if $x = yz$, say using the space-efficient algorithm for multiplication in Theorem 6.12. (The space required for this step would in fact be $c \log \log n$.)

Step 2

We explain how given $y \in [2]^n$ we can compute $(y \mod p_1, \ldots, y \mod p_\ell)$ in FL. If y_j is bit j of y we have that

$$\begin{aligned} y \mod p_i &= \left[\sum_{j=0}^{n-1} (2^j y_j)\right] \mod p_i \\ &= \left[\sum_{j=0}^{n-1} (2^j \mod p_i) y_j\right] \mod p_i. \end{aligned}$$

Note that the values $a_{i,j} := 2^j \mod p_i$ can be computed in FL and only take $c \log n$ bits. Multiplying by y_j is also in FL by Theorem 6.12. Hence the problem reduces to iterated addition of n numbers which is in FL by Theorem 6.12.

Step 3

This is a smaller version of the original problem: for each $j \leq \ell$, we want to compute $(\Pi_{i=1}^t x_i \mod p_j)$ from $x_1 \mod p_j, \ldots, x_t \mod p_j$. However, as mentioned earlier, each $(x_i \mod p_j)$ is at most n^c in magnitude and thus has a representation of $c \log n$ bits. Hence we can just perform one multiplication at the time.

Step 4

By Theorem 6.14, to convert back to standard representation from RR we have to compute the map

$$(y \mod p_1, \ldots, y \mod p_\ell) \to \sum_{i=1}^{\ell} e_i \cdot (y \mod p_i).$$

Assuming we can compute the e_i, this is just multiplication and iterated addition, which is in FL by Theorem 6.12.

Putting the steps together

Combining the steps together we can compute iterated multiplication in FL as long as we are given the integers e_i in Theorem 6.14.

Theorem 6.17. Given integers $x_1, x_2, \ldots, x_t$, and given the integers $e_1, e_2, \ldots, e_\ell$ as above, we can compute $\prod_i x_i$ in FL.

In particular, because the e_i only depend on the input length, but not on the x_i they can be hardwired in a branching program.

Corollary 6.18. Iterated multiplication $\in$ BrL.

Exercise 6.19. Show that given integers $x_1, x_2, \ldots, x_t$ and $y_1, y_2, \ldots, y_t$ one can decide if

$$\prod_{i=1}^{t} x_i = \prod_{i=1}^{t} y_i$$

in L. You cannot use the fact that iterated multiplication is in FL, a result which we stated but not completely proved.

Exercise 6.20. Show that for every $d \in \mathbb{N}$ iterated multiplication of $d \times d$ matrices over the integers $\in$ BrL.

6.2.2 Graphs

We now give another example of the power of L.

Definition 6.21. The undirected reachability problem: Given an undirected graph G and two nodes s and t, determine if there is a path from s to t.

Standard time-efficient algorithms to solve this problem *mark* nodes in the graph. In logarithmic space we can keep track of a constant number of nodes, but it is not clear how we can avoid repeating old steps forever. Surprisingly, this is possible:

Theorem 6.22. Undirected reachability is in L.

The idea behind this result is that a *random walk* on the graph will visit every node, and can be computed using small space, since we just need to keep track of the current node. Then, one can *derandomize* the random walk and obtain a deterministic walk, again computable in small space. I give a self-contained proof in Chapter 12.

6.2.3 Linear algebra

Our final example comes from linear algebra. Familiar methods for solving a linear system

$$Ax = b$$

require a lot of space. For example using elimination we need to rewrite the matrix A. Similarly, we cannot easily compute determinants using small space. However, a different method exists.

Theorem 6.23. Deciding if an integer linear system has a solution $\in \text{Space}(c \log^2 n)$.

We sketch the proof in Theorem 7.8. For a complete proof see the notes.

6.3 Checkpoints

The *checkpoint* technique is a fundamental tool in the study of space-bounded computation. Let us illustrate it at a high level. Let us consider a graph G, and let us write $u \rightsquigarrow^t v$ if there is a path of length $\leq t$ from u to v. The technique allows us to *trade the length of the path with quantifiers*. Specifically, for any parameter b, we can break down paths from u to v in b smaller paths that go through $b-1$ checkpoints. The length of the smaller paths needs be only t/b (assuming that b divides t). We can guess the breakpoints and verify each smaller path separately, at the price of introducing quantifiers but with the gain that the path length got reduced from t to t/b. The checkpoint technique is thus an instantiation of the general paradigm of guessing computation and verifying it locally, introduced in Chapter 5. One difference is that now we are only going to guess *parts* of the computation.

The checkpoint technique
in As
$u \rightsquigarrow^t v \Leftrightarrow \exists p_1, p_2, \ldots, p_{b-1} : \forall i \in [b] : p_i \rightsquigarrow^{t/b} p_{i+1}$,

where we denote $p_0 := u$ and $p_b := v$.

Two aspects are important of this technique. First, it can be applied *recursively* to the problems $p_i \rightsquigarrow^{t/b} p_{i+1}$. We need to introduce more quantifiers, but we can reduce the path length to t/b^2, and so on. Second, it goes hand in hand with the philosophy of re-using space: We can re-use space for different i.

We will see several instantiations of this technique, for various settings of parameters, ranging from $b = 2$ to $b = n^c$. As a first example, we use the checkpoint technique to speed up space-bounded computation with alternations.

Theorem 6.24. $\mathrm{L} \subseteq \bigcup_{i \in \mathbb{N}} \Sigma_i \mathrm{Time}(n^\epsilon)$, for any $\epsilon > 0$.

Proof. Let $f \in \mathrm{L}$. For any given $\epsilon > 0$ we will show that $f \in \Sigma_i \mathrm{Time}(n^{c\epsilon})$ for some i. Since this holds for any ϵ, the result follows. Let G be the configuration graph (Definition 6.9) corresponding to (a program for) f, with $t := n^a$ nodes for inputs of length n, some $a \in \mathbb{N}$. On input $x \in [2]^n$ we need to decide if the start node reaches the node labeled 1 within t steps (w.l.o.g. we assume there is a unique node labeled 1). We apply the checkpoint technique recursively, with parameter $b := n^\epsilon$ (we assume w.l.o.g. this and later quantities are integer). Each application reduces the path length by a factor b. Hence with a/ϵ applications we can reduce the path length to

$$\frac{t}{b^{a/\epsilon}} = \frac{t}{n^a} = 1.$$

Each quantifier ranges over $\leq cb \log n^a$ bits, which is sufficient to guess b configurations. The total number of quantified bits is then $m := c(a/\epsilon) \cdot b \log n^a \leq n^{c\epsilon}$, the latter holding for n large enough.

There remains to check a path of length 1, i.e., an edge. The endpoints of this edge are two configurations u and v which depend on the quantified bits. It suffices to say that they

can be computed in time $\leq m^c \leq n^{c\epsilon}$. Once we have computed u and v we can check if u leads to v in one step, which may involve reading one bit of the input x. This check is even faster, can be done in time depending on the program only. **QED**

6.4 The grand challenge for space

We now discuss impossibility results for space, paralleling Chapter 1. Again, we can use diagonalization to prove a *space hierarchy*. The statement and the proof follow closely Theorem 1.30 without adding new ideas, so they are omitted. However, we mention that, again, from the space hierarchy it follows that $\mathrm{L} \neq \mathrm{PSpace}$. Hence in particular either $\mathrm{P} \neq \mathrm{L}$ or $\mathrm{P} \neq \mathrm{PSpace}$. Thus, we know that at least one important separation between time and space holds. Most people appear to think that *both* hold, but we are unable to prove *either*.

We now turn to non-uniform impossibility results. Whereas it is consistent with our knowledge that P or even NP have linear-size circuits, for branching program we can rule this our and establish a nearly quadratic lower bound for branching programs. A quadratic bound remains open, even for NP.

Definition 6.25. The Element-Distinctness problem: Given n/w vectors of length $w := 2 \log n$, decide if they are all distinct.

Exercise 6.26. Prove that Element-Distinctness $\in \mathrm{BrSize}\left(cn^2/\log n\right)$.

Theorem 6.27. Element-Distinctness $\notin \mathrm{BrSize}\left(n^2/\log^c n\right)$.

The proof is another example of the restrict and simplify method, see section §2.1. A suitable restriction reduces the size of the branching program; at the same time, we can guarantee that the restricted branching program still computes many functions. Comparing the number of functions to the size of the restricted program completes the argument.

Proof. Let B be a branching program computing Element-Distinctness with S variable nodes (and 2 nodes for 0 and 1). Partition its variable nodes in n/w sets depending on which (bit in a) vector they query. One of the sets has size $S' \leq Sw/n$. For any fixing a of the other $n/w - 1$ vectors, we obtain a branching program B_a with S' variable nodes, as the nodes corresponding to fixed variables can be shortcut. The critical observation is that any fixing a to distinct vectors can be recovered, up to permutation, from B_a: The vectors in a are those s.t. $B_a(a) = 1$. In particular, B_a encodes any subset of $[2]^w$ of size $n/w - 1$.

On the other hand, as in the proof of Theorem 2.9 (see Exercise 2.10), we can bound the number of branching programs of size $S' + 2$ by

$$\leq 2^{cS' \log S'} \leq 2^{(S/n) \cdot \log^c n}.$$

Indeed, for each node it suffices to indicate which variable it queries, or if it is a constant, and the two neighbors among $S' + 2$ possible nodes. For each node this takes $\leq c(\log n + \log(S' + 2)) \leq c \log S'$ (assuming w.l.o.g. $S' \geq n$). Overall, we can describe a branching program with $cS' \log S'$ bits.

Combining these two facts and taking logs we get

$$(S/n) \cdot \log^c n \geq \log \binom{2^w}{n/w-1}.$$

Recalling $w = 2 \log n$, the binomial in the right-hand side is $\geq \binom{n^2}{n/c \log n} \geq n^{cn/\log n} \geq 2^{cn}$ (see Fact A.7) and the result follows. **QED**

No quadratic bound is known for NP.

6.5 Randomness

Analogously to what we did for time-bounded computation in Chapter 3, we can consider randomized space-bounded computation. In particular, we denote by BPL the analogue of BPP for space. As in Definition 3.1 of the latter we ask that the program stops in power time. We will give a surprising example of the power of space-bounded machine that can run for a long time in Theorem 16.35.

Definition 6.28. BPL is defined as BPP Definition 3.1 except that in addition the word program runs in space $\leq |P| \log |x|$ (where P and x are as in Definition 3.1).

The analogous of the "BPP = P?" question for space is "BPL = L?."

It can be shown that BPL $\subseteq$ P (exercise). In terms of space, one can simulate BPL deterministically using quadratic log space, using a proof similar to that of Theorem 6.46. One can get the best of both worlds and obtain a single program that simultaneously runs in power time and quadratic log space, see Theorem 6.51. If only space is counted, the following better simulation is known.

Theorem 6.29. BPL $\subseteq$ Space$(\log^{3/2} n)$.

For the proof see the notes.

6.6 Reductions

As in chapters 4 and 5, we can use reductions to relate the space complexity of problems, and we can identify complete problems. We discuss this in turn for PSpace and L.

6.6.1 P vs. PSpace

Definition 6.30. A *quantified boolean formula* (QBF) is a boolean formula where we allow both $\exists$ and $\forall$ quantifiers. The QBF problem: Given a QBF, is it true?

Example 6.31. $\exists x \forall y \exists z : (x \vee y) \wedge (\neg x \vee z)$ is a true QBF.

Theorem 6.32. QBF is complete for P vs PSpace.

Proof. To prove that QBF is in PSpace we use a natural recursive algorithm. Given a QBF, we consider one quantifier Qx and we recursively determine the truth of the formula with $x = 0$ and $x = 1$, reusing the space among recursive calls. If $Q = \exists$ we return *true* if at least one assignment resulted in a true formula, if $Q = \forall$ if both. The space $S(n)$ satisfies

$$S(n) \leq n^c + S(n-1)$$

with solution $S(n) \leq n^c$.

To prove hardness, we use the checkpoint technique to recursively halve the computation time, and we note that we can express this as a QBF. Details follow. Let $f \in \text{Space}(n^a)$ and P be a corresponding program. Let x be an input. As follows from the proof of Theorem 6.4, the configurations of P take $\leq cn^a$ bits and so the running time of P is $\leq t := c^{n^a}$.

To know if P goes from configuration C to C' in $\leq t$ steps, we guess a middle configuration C_m and check if it goes from C to C' in $t/2$ steps, and from C_m to C_2 in $t/2$ steps.

This introduces an $\exists$ quantifier over cn^a bits, and a $\forall$ quantifier on 1 bit. Repeating $\log(t) \leq cn^a$ times, we have reduced the time to 1. The final check to do is to verify if a configuration C goes to a configuration C' in 1 step. This (including computing C and C') can be computed in power time from the quantified bits and the input x. Hence by Theorem 2.12 and Theorem 5.1 we can introduce another $\exists$ quantifier on n^{c_a} bits and write this computation as a 3CNF. **QED**

As for NP, many natural problems are known to be PSpace complete. Among them are many popular games. Recall from Chapter 5 that popular *single-player* games and puzzles are NP-complete (when suitably generalized). By contrast, PSpace complete problems include many popular *two-player* games. This is not surprising, as QBF itself can be seen as a game between player $\forall$ and $\exists$ who alternate setting variables. The game instance consists of the QBF, and player $\exists$ wins if they can make the formula true. For a list of PSpace complete problems, including many games, see the notes.

6.6.2 L vs. P

A very basic problem is complete for L vs P. We are given a circuit *with no variables*, just constants, and we want compute its output. This is in the same spirit of the zero-integer circuit problem from Definition 3.8. But here the setting is even simpler, as we work with plain boolean circuits, as in Chapter 2.

Definition 6.33. The circuit value problem: Given a circuit C with no variables, compute $[C]$.

We can easily solve this problem in power time by computing one gate at the time. The straightforward way of doing this would require much space to store the values of intermediate gates. The next result shows that if we could dramatically decrease the space then L = P would follow.

Theorem 6.34. Circuit value is complete for L vs P.

Proof. Circuit value is in P since we can evaluate one gate at the time. Now let $f \in \mathrm{P}$. We can reduce computing f on input x to a circuit value instance, as in Theorem 2.12. The important point is that this reduction is in FL: given an index to a gate in the circuit, we can compute the type of the gate and the indexes to its children in FL. This can be verified by inspection of the circuit in the proof of Theorem 2.12. The details are in fact easier to visualize for the model of *tape machines*, see the proof of Theorem 16.19. **QED**

Definition 6.35. The monotone circuit value problem: Given a circuit C with no variables and no negations, compute $[C]$.

Exercise 6.36. Prove that monotone circuit value is complete for L vs P. Hint: Problem 2.2.

Recall from section 6.2.3 that finding solutions to linear systems

$$Ax = b$$

has space-efficient algorithms. Interestingly, if we generalize equalities to inequalities the problem becomes P complete. This set of results illustrates a sense in which "linear algebra" is easier than "optimization."

Definition 6.37. The linear inequalities problem: Given a $d \times d$ matrix A of integers and a d-dimensional vector of integers, determine if the system $Ax \leq b$ has a solution over the reals.

Theorem 6.38. Linear inequalities is complete for L vs P.

Proof. The ellipsoid algorithm shows that Linear inequalities is in P, but we will not discuss this classic result; for a reference see the notes. Instead, we focus on showing how given a circuit C with no variables and one output gate we can construct a set of inequalities that are satisfiable iff C outputs 1. We shall have as many variables v_i as gates g_i in the circuit. The inequalities are as follows (recall Definition 2.1 of circuit):

- For a constant gate $g_i := b \in [2]$ add $v_i = b$ (which can be written as $v_i \geq b$ and $v_i \leq b$).
- For a Not gate $g_i := \neg g_j$ add $v_i = 1 - v_j$.
- For an And gate $g_i := g_j \wedge g_k$ add

$$\begin{aligned} v_i &\in [0,1] \\ v_i &\leq v_j \\ v_i &\leq v_k \\ v_i &\geq v_j + v_k - 1. \end{aligned}$$

- For an Or gate we can have a similar set of inequalities, or we can write it using Not and And (Fact A.1).
- For the output gate g_i add $v_i = 1$.

We claim that in every solution to the system above the value of v_i is the value in [2] of gate g_i on input x. This can be proved by induction. For constant and Not gates this is immediate. For an And gate $g_i = g_j \wedge g_k$, note that if $v_j = 0$ then $v_i = 0$ as well because of the equations $v_i \geq 0$ and $v_i \leq v_j$. The same holds if $v_k = 0$. If both v_j and v_k are 1 then v_i is 1 as well because of the equations $v_i \leq 1$ and $v_j + v_k - 1 \leq v_i$. **QED**

6.7 Nondeterministic space

Because of the insight we gained from considering non-deterministic time-bounded computation in section §5.1, we are naturally interested in non-deterministic space-bounded computation. In fact, perhaps we will gain even more insight, because this notion will really challenge our understanding of computation.

How to define non-deterministic space-bounded computation? A naive approach is to do a local change to Definition 5.3 of NTime. This is an ill-fated choice:

Exercise 6.39. Define $\exists \cdot \mathrm{L}$ as (NP Definition 5.3), except that the program P uses logarithmic space. Prove that $\exists \cdot \mathrm{L} = \mathrm{NP}$.

Instead, the definition of non-deterministic space is a variant of the *alternative* definition of NTime with Guess instructions (see discussion in section §5.1).

Definition 6.40. A *nondeterministic* word program is a word program with the extra instruction

- $R[i] :=$ Guess, where $i \in [\ell]$ and ℓ is the number of registers of the program.

This instruction sets $R[i]$ to a value in $[2]^w$, where w is the current word length.

A function $f : X \subseteq [2]^* \to [2]$ is computable in $\mathrm{NSpace}(s(n))$ if there is a nondeterministic word program P that for every $x \in X$ of length $\geq |P|$ satisfies:

- If $f(x) = 1$ then there exists a sequence of values for the Guess instructions such that $P(x)$ outputs 1 in space s, and

- If $f(x) = 0$ then for every sequence of values for the Guess instructions $P(x)$ outputs 0 in space s.

We define

$$\mathrm{NL} := \bigcup_d \mathrm{NSpace}(d \log n),$$
$$\mathrm{NPSpace} := \bigcup_d \mathrm{NSpace}(n^d).$$

How can we exploit this non-determinism? Recall from section 6.2.2 that reachability in *undirected* graphs is in L. It is unknown if the same holds for directed graphs. However, we can solve it in NL.

Definition 6.41. The directed reachability problem: Given a directed graph G and two nodes s and t, decide if there is a path from s to t.

Theorem 6.42. Directed reachability is in NL.

Proof. The proof simply amounts to guessing a path in the graph. The pseudocode is as follows.

```
On input G, s, t:
v := s
For i = 0 to |G|:
  If v = t then Accept
  Guess a neighbor w of v
  v := w
Reject
```

The space needed is $c(|v| + |i|) = c \log n$. **QED**

We can investigate completeness for NL similarly to NP, and have the following result.

Theorem 6.43. Directed reachability is complete for L vs NL.

Exercise 6.44. Prove this.

The simulation of space by time in Theorem 6.4 holds even for non-deterministic space.

Theorem 6.45. $\text{NSpace}(s(n)) \subseteq \bigcup_a \text{Time}(a^s)$, for any $s(n) \geq \log n$.

Proof. On input x, we compute the configuration graph of the program on input x. The nodes are the configurations, and there is an edge from u to v if the machine can go from u to v in one step. Unlike the deterministic case, this graph now has large out-degree (corresponding to the Guess instruction). Still, the size of this graph is a^s for a constant a depending on the program. This graph can be written down in power time in its size. Once we have the graph we solve reachability on this graph in power time, using say breadth-first-search.

A detail is that we may not know what s is. So we try to write down the graph with $s = \log n$. If we can't, because some configuration points to a larger configuration, we increase s by 1, and repeat. We continue until the graph can be computed; this doesn't affect the time bound. **QED**

The next theorem shows that non-deterministic space is not much more powerful than deterministic space: it buys at most a square. Contrast this with the P vs. NP question! The best deterministic simulation of NP that we know is the trivial $\text{NP} \subseteq \text{Exp}$. Thus the situation for space is entirely different.

Theorem 6.46. $\text{NSpace}(s) \subseteq \bigcup_{d \in \mathbb{N}} \text{Space}(ds^2)$, for every $s = s(n) \geq \log n$. In particular, NPSpace = PSpace.

Proof. We use the checkpoint technique with parameter $b = 2$, and re-use the space to verify the smaller paths. Let $f \in \text{NSpace}(s(n))$ and N a corresponding non-deterministic word program. We aim to construct a (deterministic) word program that on input x computes

$$\text{Reach}(C_{\text{start}}, C_{\text{accept}}, c_N^{s(n)}),$$

where $\text{Reach}(u, v, t)$ decides if v is reachable from u in $\leq t$ steps in the configuration graph of N on input x, C_{start} is the start configuration, C_{accept} is the configuration outputting 1, and $c_N^{s(n)}$ is the number of configurations of N, as in the proof of Theorem 6.4.

The key point is how to implement Reach.

```
Computing Reach(u, v, t)
If t = 1 then Accept if u yields v, otherwise Reject
\\Otherwise, t > 1
For all ‘‘middle’’ configurations m
  If both Reach(u, m, t/2) = 1 and Reach(m, v, t/2) = 1 then ACCEPT
Reject
```

Let $S(t)$ denote the space needed for computing $\text{Reach}(u, v, t)$. We have $S(1) = c_N s(n)$, while for $t > 1$:

$$S(t) \leq c_N s(n) + S(t/2).$$

This is because we can re-use the space for two calls to Reach. Therefore, the space for $\text{Reach}(C_{\text{start}}, C_{\text{accept}}, c_N^{s(n)})$ is

$$\leq c_N s(n) + c_N s(n) + \ldots + c_N s(n) \leq c_N s^2(n).$$

As in the proof of Theorem 6.45, a detail is that s may be hard to compute. Again, we can run the above procedure with incremental space bounds $s = \log n, \log n + 1, \ldots$ without affecting the final space bound. **QED**

To set the stage for the next result, recall that we do not know if NP is closed under complement. It is generally believed not to be, and if it is then the PH collapses (Theorem 5.41).

What about space? Theorem 6.46 shows $\text{NSpace}(s) \subseteq \text{Space}(cs^2)$. Because the latter is closed under complement, up to a quadratic loss in space, non-deterministic space is closed under complement.

Can we avoid squaring the space? Yes! This is weird!

Theorem 6.47. The complement of Directed Reachability is in NL. In particular, NL is closed under complement.

Proof. We want a non-deterministic word program that given a graph G with $\leq n$ nodes , and s and t accepts iff there is *no* path from s to t. For starters, assume the program has somehow computed the number C of nodes reachable from s. *The key idea is that there is no path from s to t iff there are C nodes different from t reachable from s.* Thus, knowing C we can solve the problem as follows:

```
Algorithm for deciding if there is no path from s to t, given C:
Count:= 0
For all nodes v ≠ t :
  Guess a path from s of length n
  If path reaches v, increase Count by 1
If Count=C Accept else Reject
```

There remains to compute C. Let A_i be the nodes at distance $\leq i$ from s, and let $C_i := |A_i|$. Note $A_0 = \{s\}, c_0 = 1$. We seek to compute $C = C_n$. To compute C_{i+1} from C_i, enumerate nodes v (candidates in A_{i+1}). For each v, enumerate over all nodes w in A_i, and check if $w \to v$ is an edge. If so, increase C_{i+1} by 1. The enumeration over A_i is done guessing C_i nodes and paths from s. If we don't find C_i nodes, we reject.

If this procedure runs all the way to C_n (without ever rejecting) then we have computed C_n correctly. **QED**

Exercise 6.48. Given a graph G and nodes s, t, and the count C of the number of nodes reachable from s, explain how to compute in FL a graph G' and nodes s', t' s.t. there is no path from s to t in G iff there is a path from s' to t' in G', and $|G'| \leq |G|^c$.

Give a direct "algorithmic" proof, based on the algorithm in Theorem 6.47, but without using the result as a black-box, or mentioning configuration graphs or completeness.

6.8 An impossibility result for 3Sat

We should turn back to a traditional separation technique – diagonalization.

We put together many of the techniques we have seen to obtain a remarkable impossibility results for 3Sat:

Theorem 6.49. Either 3Sat $\notin$ L or 3Sat $\notin$ Time($n^{1+\epsilon}$) for some constant ϵ.

Note that we don't know if 3Sat $\in$ L or if 3Sat $\in$ Time($n \log^{10} n$). But Theorem 6.49 implies that one of the two is false. One can optimize the methods to push ϵ close to 1, but even establishing $\epsilon = 1$ seems out of reach, and there are known barriers for current techniques (see the notes).

Proof. We assume that what we want to prove is not true and derive the following striking contradiction with the hierarchy Theorem 1.30:

$$\begin{aligned} \text{Time}(n^2) &\subseteq \text{L} \\ &\subseteq \bigcup_d \Sigma_d \text{Time}(n) \\ &\subseteq \text{Time}(n^{1.9}). \end{aligned}$$

The first inclusion holds by the assumption that 3Sat $\in$ L and the fact that any function in Time(n^2) can be reduced to 3Sat in FL, by Theorem 5.1 and the discussion after that, and the composition result for space, Lemma 6.11.

The second inclusion is Theorem 6.24.

For the third inclusion, we start similarly to the first. We first claim that NTime($n^{1+\epsilon}$) $\subseteq$ Time($n^{1+2\epsilon}$) for every ϵ. Then the inclusion follows from Exercise 5.40.

The claim follows from the completeness of 3Sat, Theorem 5.19. More in detail, we can reduce NTime($n^{1+\epsilon}$) to a 3Sat instance of length $m := n^{1+\epsilon} \log^c n$. Then applying the algorithm from 3Sat in the assumption, with running time $n^{1+\epsilon/2}$, we can solve this instance in time $m^{1+\epsilon} = n^{(1+\epsilon)(1+\epsilon/2)} \log^{c(1+\epsilon/2)} n$. For large enough n, this is $\leq n^{1+2\epsilon}$. **QED**

6.9 TiSp

So far in this chapter we have focused on bounding the space usage. It is natural to consider algorithms that operate in little time *and* space.

Definition 6.50. TiSp(t, s) denotes the functions computable by a word program that uses space s as in Definition 6.1 and simultaneously time t.

To illustrate the relationship between TiSp, Time, and Space, consider undirected reachability. It is solvable in Time($n \log^c n$) by breadth-first search, and in logarithmic space by Theorem 6.22. But it isn't known if it is in TiSp($n \log^a n, a \log n$) for some constant a. Also, the following result is known, improving on simpler simulations mentioned in section §6.5.

Theorem 6.51. BPL $\subseteq$ TiSp($n^a, a \log^2 n$).

For a proof see the notes.

The following is a non-uniform version of TiSp.

Definition 6.52. A branching program of length t and width W is a branching program where the nodes are partitioned in t layers $L_1, L_2, \ldots, L_t$ where nodes in L_i only lead to nodes in L_{i+1}, and $|L_i| \leq W$ for every i.

Thus t represents the time of the computation, and $\log W$ the space. Recall that Theorem 9.32 gives bounds of the form $\geq cn^2/\log n$ on the size of branching program (without distinguishing between length and width). For branching programs of length t and width W this bound gives $t \geq cn^2/W \log n$. Note this gives nothing for power width like $W = n^2$. The state-of-the-art for power width is $t \geq cn\sqrt{\log n/\log\log n}$, which in fact holds even for subexponential width, see the notes.

We now prove an impossibility result for 3Sat similar to Theorem 6.49, but for TiSp. We seek to rule out algorithms for 3Sat that simultaneously use little space and time, whereas in Theorem 6.49 we even ruled out the possibility that there are two distinct algorithms, one optimizing space and the other time. The main gain is that we will be able to handle much larger space: power rather than log.

Theorem 6.53. $3\text{Sat} \notin \text{TiSp}(n^{1+c_\epsilon}, n^{1-\epsilon})$, for any $\epsilon > 0$.

Proof. We follow closely the proof of Theorem 6.49. We assume that what we want to prove is not true and derive the following contradiction with the time hierarchy Theorem 1.30:

$$\begin{aligned}\text{Time}\left(n^{1+\epsilon}\right) &\subseteq \text{TiSp}\left(n^{1+c_\epsilon}, n^{1-\epsilon^2/2}\right)\\ &\subseteq \Sigma_{c_\epsilon}\text{Time}(n)\\ &\subseteq \text{Time}\left(n^{1+\epsilon/2}\right).\end{aligned}$$

We only sketch the proof of the first inclusion. It follows from the fact that 3Sat is complete under reductions s.t. each bit is computable in time (and space) $\log^c n$; this follows from an inspection of the proof of Theorem 5.16 whose details we omit.

The second inclusion is Exercise 6.54.

The last is the same as in the proof of Theorem 6.49. **QED**

Exercise 6.54. Prove the following version of Theorem 6.24: $\text{TiSp}(n^a, n^{1-\alpha}) \subseteq \Sigma_{ca/\alpha}\text{Time}(n)$ for any $a \geq 1$ and $\alpha > 0$.

6.10 Computing with a full memory: Catalytic space

> *Imagine the following scenario. You want to perform a computation that requires more memory than you currently have available on your computer. One way of dealing with this problem is by installing a new hard drive. As it turns out you have a hard drive but it is full with data, pictures, movies, files, etc. You don't need to access that data at the moment but you also don't want to erase it. Can you use the hard drive for your computation, possibly altering its contents temporarily, guaranteeing that when the computation is completed, the hard drive is back in its original state with all the data intact? [...] Can you still make good use of this additional space?*

Turns out you can. We illustrate this in a simple scenario. First, let us define the model.

Definition 6.55. Catalytic log-space (CL) is the class of problems that can be solved in logarithmic space and power time by a machine equipped with an extra power-size memory. For every input and any possible setting of the extra memory, the machine needs to compute the output. At the end of the computation, the extra memory must be in the same setting it was at the beginning.

Theorem 6.56. $\text{NL} \subseteq \text{CL}$.

Proof. We are given a graph G with n nodes, and two nodes s and t and we'd like to know if there is a path from s to t. By adding self-loops on t, this is equivalent to asking if there is a path of length exactly n.

Let $W_{u,i}$ for $u \in [n], i \in [n+1]$ be a memory word of $n^c = \log q$ bits, which we view as an element of $\mathbb{Z}_q$, so sums will be modulo q. These words are in the large memory, which is initialized to values we don't know. Consider the procedure Propagate:

```
Propagate:
For i = 0, 1, ..., n − 1
  For all edges u → v in G
n     W_{v,i+1} += W_{u,i}
```

Note Propagate can be easily reversed, by subtracting values in reverse order. We call the corresponding operation ReversePropagate

At the end of Propagate, the value $W_{t,n}$ contains the number of paths leading to t modulo q, *plus something which depends on the initial values of the* W. To get rid of the latter, consider running propagate after increasing $W_{s,0}$ by 1. The change in $W_{t,n}$ will be the number of paths from s to t modulo q. Since the number of such paths is $\leq n^n$, by our choice of q this this change will be 0 iff there is a path from s to t. We can check if this change is 0 by comparing the two values bit by bit.

More specifically, we can decide connectivity as follows

```
For i ∈ [log q] {
  Propagate
  Store bit i of W_{t,n} in b
  ReversePropagate
  W_{s,0} += 1
  Propagate
  Xor bit i of W_{t,n} with b
  ReversePropagate
  W_{s,0} −= 1
  If b = 1 then output CONNECTED and STOP
}
Output NOT CONNECTED
```

When the program stops the words W are repristinated. By inspection the program runs in power time. **QED**

In fact, CL is believed to be larger than NL, see Problem 7.4.

6.11 Problems

Problem 6.1. [Acyclic branching programs] We define the *graph* of a branching program as the graph obtained by removing all labels.

(1) Give an example of a branching program that computes a non-trivial function and whose graph is not acyclic.

(2) Prove that any branching program of size s has an equivalent branching program of size s^2 whose graph is acyclic.

Problem 6.2. Prove that L map reduces in quasi-linear time to QBF where instances of length n have $\leq \log^c n$ variables.

Problem 6.3. Consider the class $\Sigma_{a(n)}\text{Time}(t(n))$ where the number of alternations is $a(n)$ on inputs of length n (as opposed to being fixed to i for every input as in $\Sigma_i\text{Time}(t(n))$.

Prove $\text{L} \neq \Sigma_{a(n)}\text{Time}(n)$ for any growing $a(n)$.

Problem 6.4. A beautiful illustration of the power of L. Consider strings over the alphabet $\{u, v, u^{-1}, v^{-1}\}$ (a.k.a. words in the free group with 2 generators). A string can be simplified by removing adjacent pairs of the type $uu^{-1}, u^{-1}u, vv^{-1}, v^{-1}v$. For example, $uv^{-1}vu^{-1}$ simplifies to uu^{-1} and then to the empty string. On the other hand, the string $uv^{-1}u^{-1}$ cannot be simplified to the empty string. The Simplify problem: Given a string, does it simplify to the empty string?

Prove Simplify is in L. Guideline:

(1) For integers i consider the matrices

$$U_i := \begin{bmatrix} 1 & 2i \\ 0 & 1 \end{bmatrix}, V_i := \begin{bmatrix} 1 & 0 \\ 2i & 1 \end{bmatrix}.$$

Show that $U_iU_j = U_{i+j}$ and so in particular $U_i^{-1} = U_{-i}$; and show the same for the V_i.

(2) Let $\begin{bmatrix} x \\ y \end{bmatrix} \in \mathbb{Z}^2$ be a vector and let $i \neq 0$. Show that if $|x| < |y|$ then $\begin{bmatrix} x' \\ y' \end{bmatrix} := U_i \begin{bmatrix} x \\ y \end{bmatrix}$ has $|x'| > |y'|$. Conversely, show that if $|x| > |y|$ then $\begin{bmatrix} x' \\ y' \end{bmatrix} := V_i \begin{bmatrix} x \\ y \end{bmatrix}$ has $|x'| < |y'|$. (This is the so-called ping-pong lemma.)

(3) Show that an alternating product of matrices $U_{i_1}V_{i_2}U_{i_3}V_{i_4}\cdots U_{i_k}$ where the i_j are not zero is not equal to the identity matrix. Note that we begin and end with a U matrix, and we alternate between U and V.

(4) Show that a product of matrices $U_{i_1}V_{i_2}U_{i_3}\cdots V_{i_k}$ where the i_j are not zero is not equal to the identity matrix. Note that we begin with U but end with a V matrix, and as before we alternate U and V matrices. (Hint: Reduce to (3) by multiplying on the left by M^{-1} and on the right by M.)

(5) Show that Simplify is in L.

(6) Consider the generalization of simplify to k elements u_i (corresponding to the free group with k generators). Show that it is in L for every k. Hint: Reduce to the $k = 2$ case by mapping u_i to u^ivu^{-i} (and u_i^{-1} to $u^{-i}vu^i$).

6.12 Notes

A non-trivial simulation of time by space is in [158]. I suspect this should also save a log factor in Theorem 6.4, but I don't know it has been verified for word programs. Theorem 6.10 is from [387], see also the follow-up [307].

Theorem 6.22 is from [292]. The time must have been "ripe:" a concurrent, different proof [344] gives the only slightly weaker space bound $c \log n \log \log n$. A variant of the proof in [292] appeared later in [298]. See section §12.7 for discussion.

Theorem 6.23 follows from [97] which in fact establishes a stronger result, for a class we encounter in Chapter 7.

The use of RR for arithmetic is from [50], which also contains several reductions among arithmetical problems. Some of the steps are from the earlier work [247]. For a discussion of the complexity of division, see [18].

Theorem 6.27 is from [263].

The hierarchy for space was proved in [324].

For the simulation of BPL using space $\log^{3/2} n$, Theorem 6.29, see [299, 182], the latter achieving a slightly better space bound than Theorem 6.29. For Theorem 6.51, see [268].

For a compendium of P-complete problems see [145], for PSpace see [381].

Theorem 6.24 goes back to [264].

Theorem 6.46 is from [302].

Theorem 6.47 was obtained independently in [185, 334]. An earlier surprising collapse paved the way, at least for one of the proofs, cf [224]. The proof in [185] uses a logical formalism, the proof we presented is closer to the one in [334]. The question of whether NL is closed under complement was a.k.a. the "second LBA problem," where LBA stands for linear-bounded automaton, see [221]. As usual, had the answer been different, it would have had applications to the first LBA problem, which is the question whether Theorem 6.46 is tight, and remains open.

See [13, 52] for the state-of-the-art bounds for power-width branching programs.

The introductory quote to section §6.8 is from [113], where Theorem 6.49 is proved. This influential work ignited a whole research area. For a survey (not up to date) see [355] or [383]. For the limitations of this type of results, see [77].

That linear inequalities is in P was shown first in [210].

Catalytic computation was introduced in [69]. The text at the beginning of section §6.10 is their introductory paragraph. They also prove that CL contains NL (Theorem 6.56) and supposedly larger classes as well. The proof of Theorem 6.56 that we presented is from [92].

Regarding Definition 6.7 of branching programs, and cycles. Some texts require that the graph of a branching program, ignoring all labels, is acyclic. This can be thought of as a *syntactic* condition which makes it easy to decide if a graph corresponds to a branching program. By contrast, my definition is *semantic*: I ask that the path induced by an input does not contain a cycle. One would *a priori* need to examine all $x \in X$ to decide if this is the case. The semantic definition appears more in line with the way we defined other resources. For example, given a program it is not immediate to decide if it runs in a specific time bound. Moreover, it makes the proof of Theorem 6.8 slightly easier. For a comparison of the two definitions see Problem 6.1.

Chapter 7

Depth

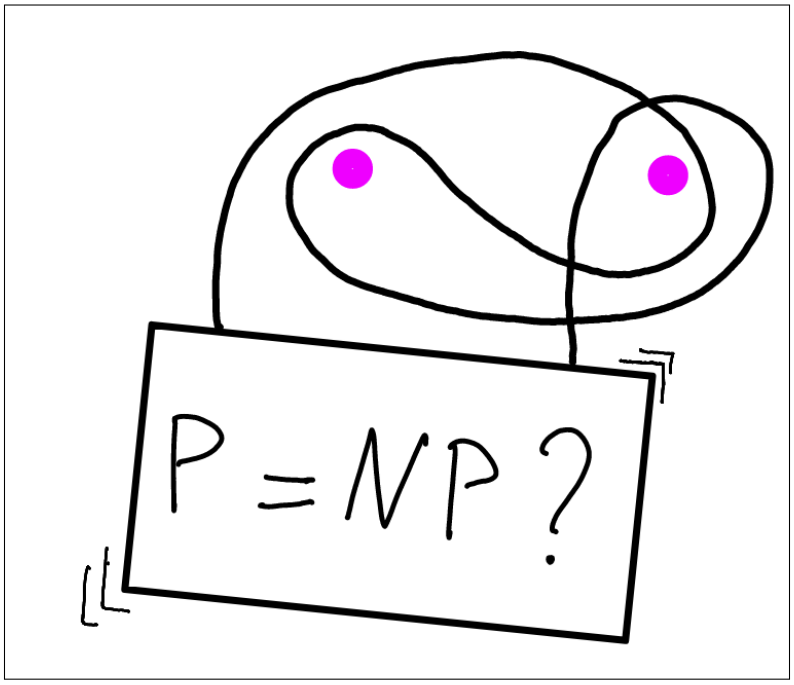


Figure 7.1: A picture hung on two nails so that removing either one causes the picture to fall. The Grand Challenge hangs in the balance.

In this chapter we study circuits of small depth. Many surprises lay ahead, including a solution to the teaser in Chapter 0!

Definition 7.1. The *depth* of a circuit is the length d of a longest subsequence (or path) $g_1, g_2, \ldots, g_d$ of its gates where g_i is input to g_{i+1} for each $i \in [1..d-1]$.

For $i \in \mathbb{N}$, NC^i is the class of functions $f : X \subseteq [2]^* \to [2]^*$ for which there is $a \in \mathbb{N}$ s.t. for every $n \geq 2$ the function on inputs of length n is computable by a circuit that has depth $a \log^i n$ and size n^a.

Finally,

$$\mathrm{NC} := \bigcup_d \mathrm{NC}^d.$$

For convenience we also use the term "NC^i circuit" to refer to a circuit whose depth is bounded by $a \log^i n$ for an absolute constant a. For example we talk of an NC^0 circuit, and so on.

Let us begin slowly with some basic properties of small-depth circuits so as to get familiar with them. The next exercise implies that circuits of depth $d \log n$ for a constant d also have power size, so we don't need to bound the size separately.

Exercise 7.2. A circuit of depth d with m output gates has an equivalent circuit of size $\leq mc^d$.

The next exercises shows how to compute several simple functions by log-depth circuits.

Exercise 7.3. Prove that the Or, And, and Parity functions are in NC^1.

The class NC^0 is also of great interest. It can be more simply defined as the class of functions where each output bit depends on a constant number of input bits. We will see many surprising, useful things that can be computed NC^0, for example section §7.7.

Exercise 7.4. Prove that $\mathrm{NC}^0 \neq \mathrm{NC}^1$.

7.1 Depth vs space

Now let us compare depth and space.

Theorem 7.5. $\mathrm{NL} \subseteq \mathrm{NC}^2$.

Proof. We show that directed reachability is in NC^2, from which the result follows (exercise). On input a directed graph G on $\leq n$ nodes and two nodes s and t, let M be the $n \times n$ transition matrix corresponding to G, where $M_{i,j} = 1$ iff edge $j \to i$ is in G. Transition matrices are multiplied similarly to matrices over $\mathbb{Z}$, except that "+" is replaced with "$\vee$," which suffices to determine connectivity. To answer directed reachability we compute entry t of $M^n v$, where v has a 1 corresponding to s and 0 everywhere else. (We can modify the graph to add a self-loop on node t so that we can reach t in exactly u steps iff we reach t in any number of steps.)

Computing M^n can be done by squaring $c \log n$ times M. Each squaring can be done in depth $c \log n$, by Exercise 7.3. **QED**

Conversely, we have the following.

Theorem 7.6. $\mathrm{NC}^1 \subseteq \mathrm{BrL}$.

Below in Theorem 7.30 we prove the stronger and much less obvious result that the branching programs can be taken to have width 5.

Proof. We prove by induction that circuits of depth d with 1 output gate have branching programs of size c^d, from which the result follows. The case of depth 1 is immediate. For the induction step, suppose the circuit C has the form $C_1 \wedge C_2$. By induction, C_1 and C_2 have branching programs $B_1$and B_2 each of size c^{d-1}. A branching program B for C of size 2^d is obtained by rewiring the edges leading to states labelled 1 in B_1 to the start state of B_2. The start state of B is the start state of B_1. Its size is $\leq 2 \cdot c^{d-1} \leq c^d$. **QED**

Exercise 7.7. Finish the proof by analyzing the case $C = C_1 \vee C_2$ and $C = \neg C_1$.

7.2 The power of NC^2: Linear algebra

We can informally think that *linear algebra* problems are solvable in NC^2. In particular, we can compute matrix inversion. This can be done over any field. We sketch the ideas over $\mathbb{Q}$ and refer to the notes for other fields. Rationals are represented as pairs of integers. This representation may not be unique, as it is not known how to compute the greatest common divisor in NC.

Theorem 7.8. Given an invertible matrix over $\mathbb{Q}$, we can compute its inverse NC^2.

Proof sketch.. Let A be a $d \times d$ matrix. The key is to compute the (coefficients of the) characteristic polynomial of A,

$$p_A(x) = \det(xI - A) = x^d - s_1 x^{n-1} + s_2 x^{n-2} - \cdots \pm s_n$$

where $s_i \in \mathbb{Q}$. Once we have this polynomial we use that $p_A(A) = 0$ by Fact A.44. This gives

$$A^d - s_1 A^{n-1} + s_2 A^{n-2} - \cdots \pm s_n I = 0.$$

Multiplying this equation by A^{-1} we see that we can compute A^{-1} in NC^2 given the powers A^i and the s_i. The powers A^i can be computed in NC^2 by repeated squaring. To compute the s_i we use that

$$p_A(x) = \prod_{i=1}^{d} (x - \lambda_i)$$

where the λ_i are the *eigenvalues* of A. Using Fact A.49 we can compute in NC^2 the coefficients of $p_A(x)$ from the power sums

$$p_j(\lambda_1, \lambda_2, \ldots, \lambda_d) = \sum_{i=1}^{d} \lambda_i^j = \mathrm{trace}(A^j).$$

Again, A^j can be computed in NC^2 and the trace is just the sum of the diagonal elements. **QED**

7.3 Formulae

We can equivalently think of NC^1 as power-size *formulae*. A formula is a circuit that you can actually write down on a line, such as $(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_1 \wedge x_3)$. By contrast, a circuit can reuse gates, so in general it is more complicated to write down.

Definition 7.9. A *formula* is a circuit where the fan-out is 1(i.e., each gate is input to at most another gate). FormulaSize and FormulaP are defined similarly to CktSize and CktP.

Formulae can have large depth, as in $x_1 \wedge (x_2 \wedge (x_3 \wedge \ldots)\ldots))$. Yet we have:

Theorem 7.10. $\mathrm{NC}^1 = \mathrm{FormulaP}$.

Proof. It suffices to prove this in the case of one output gate. Given a circuit of depth d, consider the formula where there is a gate g for every path from the output to a gate h. The type of gate g is the same as h. Connect (the gate corresponding to) path p to path p' if p' extends p by one edge. The fan-out is 1 by definition, and the number of paths is $\leq c^d$. If d is logarithmic then the formula has power size.

Conversely, we prove by induction on s that any formula f of size $s \geq c$ has circuits of depth $c \log s$. Find a subformula g of size $\in [0.3s, 0.7s]$. For $b \in [2]$ let g_b be f with g replaced by b. Then we have

$$f = (g \wedge g_1) \vee (\neg g \wedge g_0).$$

Note that g, g_0, g_1 all have size $\leq cs$, and the depth of the rhs is at most c plus the maximum depth of g, g_0, g_1. Applying the induction hypothesis completes the proof. **QED**

Exercise 7.11. Prove that g exists.

7.3.1 The grand challenge for formulae

For formulae the best known size lower bound is cubic. We present a weaker and simpler argument. It is convenient to express the bound in terms of the occurrences of variables x_i in the formula, from which a bound for FormulaSize follows (exercise).

Definition 7.12. For a function $f : [2]^n \to [2]$ we denote by $L(f)$ the minimum number of occurrences of input variables x_i in a formula computing f.

The lower bound on $L(f)$ is established via another instance of the restrict and simplify method (section §2.1). Specifically, the next lemma shows that by fixing an input bit of f we can reduce $L(f)$ somehow. Note that there is always a variable occurring $\geq L(f)/n$ times, and fixing that reduces $L(f)$ by $L(f)/n$; but this is not very helpful. The lemma improves this by a factor 1.5.

Lemma 7.13. Let $f : [2]^n \to [2]$ be a function depending on ≥ 2 input bits. We can fix an input bit so that the restricted formula $f' : [2]^{n-1} \to [2]$ has $L(f') \leq L(f)(1 - 1.5/n)$.

Proof. Let ϕ be a formula for f with $L(f)$ literals. Using Fact A.1 we can assume there are no Not gates in the formula. (In other words, the transformation in Problem 2.2 comes at no cost for formulae.) We claim that for every And gate

$$g_i = g_j \wedge \ell_k$$

where ℓ_k is a literal (x_k or $\neg x_k$) the sibling of ℓ_k is connected to a variable different from x_k. The same holds for Or gates.

For some i, there are $\geq L(f)/n$ occurrences of x_i. So there is a fixing of x_i that removes $L(f)/n$ input gates as wells as $\geq$ half of the siblings. Since the siblings contain at least another input gate, the result follows. **QED**

Exercise 7.14. Prove that the sibling contains an input gate x_j different from x_i.

Corollary 7.15. $L(\text{Parity}_n) \geq n^{1.5}$, where Parity_n is the parity function on n bits.

Proof. Note that $(1 - 1.5/n) \leq (1 - 1/n)^{1.5} = ((n-1)/n)^{1.5}$ by Fact A.8, and that the restriction of parity is parity again, possibly complemented (which does not change L). Applying Lemma 7.13 for $n - 1$ times:

$$1 \leq L(\text{Parity}_1) \leq L(\text{Parity}_n) \left(\frac{n-1}{n}\right)^{1.5} \left(\frac{n-2}{n-1}\right)^{1.5} \cdots \left(\frac{1}{2}\right)^{1.5} = L(\text{Parity}_n) \left(\frac{1}{n}\right)^{1.5}.$$

QED

A different argument gives a quadratic lower bound which is tight, see the notes.

7.4 The power of NC^1: Arithmetic

In this section we illustrate the power of NC^1 by showing that the same basic arithmetic which we saw is doable in L (Theorem 6.12) can in fact be done in NC^1 as well.

Theorem 7.16. The arithmetic problems in Theorem 6.12 are in NC^1.

Next we prove that addition, iterated addition, and iterated multiplication are in NC^1 (see Theorem 6.12 for the definitions). Multiplication is a special case; we do not address division. Addition is already non-trivial, as the computation of the carries appears sequential.

Proof that addition is in NC^1.. The approach is to compute all the carries in parallel using *carry look-ahead*. Specifically we note that the i-th carry of the sum $x + y$ is 1 iff there is some less significant position $j < i$ where the carry is generated and it is propagated up to i. This can be written as

$$c_i = 1 \iff \bigvee_{j<i} \left(x_j = 1 \wedge y_j = 1 \bigwedge_{k=j+1}^{i-1} (x_k = 1 \vee y_k = 1) \right).$$

QED

Exercise 7.17. Conclude the proof.

Iterated addition is surprisingly non-trivial. We can't use the methods from the proof of Theorem 6.12. Instead, we rely on a new and very clever technique.

Proof that iterated addition is in $\mathbf{NC^1}$.. We use "2-out-of-3:" Given 3 integers X, Y, Z, we compute 2 integers A, B such that

$$X + Y + Z = A + B,$$

where each bit of A and B only depends on three bits, one from X, one from Y, and one from Z. Thus A and B can be computed in NC^0. If we can do this, then to compute iterated addition we construct a tree of logarithmic depth to reduce the original sum to a sum 2 terms, for which we can use the previous result about addition.

Here's how it works. Note $X_i + Y_i + Z_i \leq 3$. We let A_i be the least significant bit of this sum, and B_{i+1} the most significant one. Note that A_i is the XOR $X_i + Y_i + Z_i$, while B_{i+1} is the majority of X_i, Y_i, Z_i. **QED**

The following corollary will also be used to solve the teaser in Chapter 0.

Corollary 7.18. Majority is in NC^1.

Exercise 7.19. Prove it.

Next we turn to iterated multiplication. The idea is to follow the proof for L in section 6.2.1. We shall use RR again. The problem is that we still had to perform iterated multiplication, albeit only in $\mathbb{Z}_p$ for $p \leq n^c$. One more mathematical fact is useful now: $(\mathbb{Z}_p - \{0\})$ is a cyclic group (Fact A.33).

Proof that iterated multiplication is in $\mathbf{NC^1}$.. We follow the proof for L in section 6.2.1. To compute iterated product of integers $r_1, r_2, \ldots, r_t$ modulo p, use Fact A.33 to compute the logarithms $\ell_1, \ell_2, \ldots, \ell_t$ s.t.

$$r_i = g^{\ell_i}.$$

Then $\prod_i r_i \mod p = g^{\sum_i \ell_i}$. We use the previous result to compute the iterated addition of the exponents in NC^1. Note that computing the exponent of a number mod p, and vice versa, can be done in log-depth since the numbers have $c \log n$ bits (as follows for example by a construction in Theorem 2.5 and Exercise 7.3). **QED**

Recall that the remaindering representation was also used to show that the iterated product of constant-dimension matrices over the naturals is in L (see Exercise 6.20). It is unknown if this problem is in NC^1, though one can make the circuit depth very close to logarithmic, see Problem 7.3.

7.5 Computing with 3 bits of memory

We now present a surprising result that in particular strengthens Theorem 7.6. For a moment, let's forget about circuits, branching programs, etc. and instead consider a new, minimalistic type of programs. Like straight-line programs (Definition 2.1), we do not have

control-flow operations or tests. In addition, this new type of program will be *memoryless:* It only has *three* registers, each holding 1 bit.

Definition 7.20. A *memoryless straight-line program* (abbreviated *MSLP*) on n bits is a sequence of operations of the type

$$R_i+ = R_j \text{ or}$$
$$R_i+ = R_j x_k$$

where x_k represents an input bit and $k \in [n]$, and $i, j \in [3]$. Here $R_i+ = R_j$ means to add the content of R_j to R_i, while $R_i+ = R_j x_k$ means to add $R_j x_k$ to R_i, where $R_j x_k$ is the product (a.k.a. And) of R_j and x_k. All operations are modulo 2.

For i, j and $f : [2]^n \to [2]$ we say that the program is *for* (or *equivalent to*)

$$R_i+ = R_j f$$

if for every input x and initial values of the registers, executing the program is equivalent to the instruction $R_i+ = R_j f(x)$, where note that R_j and R_k are unchanged.

Note that if we repeat twice a program for $R_i+ = R_j f$ then no register changes (recall the sum is modulo 2, so $1 + 1 = 0$). This feature is critically exploited later to "clean up" computation. We now state and prove the surprising result.

Theorem 7.21. Suppose $f : [2]^n \to [2]$ is computable by circuits of depth d. There is a memoryless straight-line program of length $\leq c4^d$ for

$$R_i+ = R_j f,$$

for every $i \neq j$.

Once such a program is available, we can start with register values $(0, 1, 0)$ and $i = 0, j = 1$ to obtain $f(x)$ in R_0.

Proof. It is convenient to work with circuits with Xor instead of Or gates. This is without loss of generality since $x \vee y = x + y + x \wedge y$. Moreover, we assume that each Xor and Or gate take as input two previous gates (at the cost of introducing extra gates computing input literals).

We proceed by induction on d. When $d = 1$ the output gate is a constant or a literal ℓ_k. For the constant zero we can use the empty program, for the constant 1 we use $R_i+ = R_j$, for x_k we use $R_i+ = R_j x_k$ and for $\neg x_k$ we use $R_i+ = R_j x_k$ followed by $R_i+ = R_j$.

For the induction step, a program for $R_i+ = R_j(f_1 + f_2)$ is simply given by the concatenation of (the programs for)

$$R_i+ = R_j f_1$$
$$R_i+ = R_j f_2.$$

Less obviously, a program for $R_i+ = R_j(f_1 \wedge f_2)$ is given by

$$\begin{aligned} R_i+ &= R_k f_1 \\ R_k+ &= R_j f_2 \\ R_i+ &= R_k f_1 \\ R_k+ &= R_j f_2. \end{aligned}$$

QED

Exercise 7.22. Prove that the program for $f_1 \wedge f_2$ in the proof works. Write down the contents of the registers after each instruction.

A similar proof works over other fields as well.
We can now address the teaser Theorem 0.1 from Chapter 0.

Proof of Theorem 0.1. Combine Corollary 7.18 with Theorem 7.21. **QED**

We now address the complexity of computing iterated product of matrices.

Definition 7.23. The iterated product of matrices problem: Given square matrices, output the first entry of their product.

We will consider this problem over different domains and for various dimensions.

Corollary 7.24. Iterated product of 3x3 matrices over $\mathbb{F}_2$ is complete for NC^1 under map-reductions in ProjectionsP (Definition 5.10).

Exercise 7.25. Prove this.

Recall from Chapter 6 (see in particular section 6.7) that various graph reachability problems are complete for space-bounded computation. In particular, we have:

Exercise 7.26. Show that iterated multiplication of matrices (of any dimension) over $\mathbb{F}_2$ is hard for BrL under map reductions in ProjectionP.
Does your proof apply to NL as well?

Hence major open questions about computation are related to the following "purely mathematical question" that doesn't make any direct reference to computation:

Open question 7.27. *Can iterated product of matrices be reduced in ProjectionsP to iterated product of* 3×3 *matrices? (All matrices over* $\mathbb{F}_2$*.)*

Note this means writing (any one entry of) the iterated product of matrices over variables x_i as (one entry of) a power-length product of 3×3 matrices where each entry is a literal or a constant. If the answer is positive then by above $\mathrm{BrL} \subseteq \mathrm{NC}^1$, if negative then a natural problem (in P) is not in NC^1. A breakthrough either way! More precisely, the question is equivalent to $\mathrm{NC}^1 = \oplus\mathrm{BrL}$, where the latter are "parity branching programs."

7.6 Group programs

The results in section §7.5 are relatively easy to present, but may feel a little *deus ex machina*. Now we present an alternative proof in the framework of groups. The setup may be slightly more convoluted, but the steps in the proof might be a bit more transparent. As is often the case, having two perspectives on a problem is beneficial.

> *It is better to solve one problem five different ways than to solve five problems one way.*

Definition 7.28. A group program π of length ℓ over a group G is a word of length ℓ where each element is raised to an input bit. More formally, it is given by a word $(g_1, g_2, \ldots, g_\ell)$ where $g_i \in G$, an additional element $h \in G$, and a sequence $(k_1, k_2, \ldots, k_\ell) \in [n]^\ell$ of indices to input bits. On input x the program π computes $\pi(x) = \left(\prod_{i=1}^{\ell} g_i^{x_{k_i}}\right) h \in G$. We say π α-computes $f : [2]^n \to [2]$ if $\forall x : \pi(x) = \alpha^{f(x)}$.

That is, the bits of the input x specify which subset of elements in the word to multiply. This simple formulation requires the extra element h; otherwise we can't meaningfully compute $f(0) = 1$.

Exercise 7.29. Consider the following alternative definition of group programs:

The program is given by three words in G^ℓ: $(g_1^{(b)}, g_2^{(b)}, \ldots, g_\ell^{(b)})$ for $b \in [2]$ and $(h_1, h_2, \ldots, h_\ell)$, as well as the sequence $(k_1, k_2, \ldots, k_\ell) \in [n]^\ell$ of indices to input bits, and $h \in G$; on input x the output is $\left(\prod_{i=1}^{\ell} h_i g_i^{(x_{k_i})}\right) h$.

Prove that this alternative definition is equivalent to Definition 7.28.

Computing And. Computing the And of two bits is akin to the mathematical puzzle of hanging a picture with two nails so that removing any one of them makes the picture fall (the solution is in figure 7.1). Actually, this works over any non-abelian group. Indeed, G being non-abelian is the same as saying that there are $a, b \in G$ s.t.

$$ab \neq ba.$$

This is equivalent to saying that the commutator

$$aba^{-1}b^{-1}$$

is not the identity. The following group program then computes the And of bits x and y:

$$a^x b^y a^{-x} b^{-y}.$$

Note that if $x = y = 1$ then we get the commutator which as we just said is not 1. Otherwise, either the as or the bs disappear, and the program evaluates to 1.

To compute circuits we naturally have to iterate this procedure. We can do so if we have non-trivial commutators that can themselves be used as elements in commutators. This approach works for any non-solvable group, a class which includes the group of matrices in Corollary 7.24. This is worked out in Problem 7.2. But now for concreteness we present a closely related construction over a specific group.

A solution over S_5. For concreteness we work with S_5, the group of permutations of 5 elements. Later we discuss other groups. Abusing notation we say that a permutation $g \in S_5$ is a *cycle* if its graph consists of exactly one cycle of length 5. For example, $1 \to 5 \to 2 \to 3 \to 4 \to 1$ is a cycle. We write it compactly as (15234).

Theorem 7.30. Let $f : [2]^n \to [2]$ be computable by a circuit of depth d. Then for any cycle $\alpha \in S_5$, f is α-computed by a group program of length c^d over S_5. In particular, NC^1 is equivalent to power-size branching programs of width 5.

Compare this to the weaker equivalence in Theorem 7.6.

Exercise 7.31. Prove the in particular part, assuming the first part.

To prove this theorem we begin with a lemma stating that the choice of the cycle is immaterial.

Lemma 7.32. Let $\alpha, \beta \in S_5$ be two cycles, let $f : [2]^n \to [2]$. Over the group S_5, f is α-computable with length $\ell \Leftrightarrow$ f is β-computable with length ℓ.

Proof. First recall that any two cycles $\alpha = (\alpha_1 \alpha_2 \dots \alpha_5)$ and $\beta = (\beta_1 \beta_2 \dots \beta_5)$ are conjugate, that is $\alpha = \rho^{-1} \beta \rho$ for

$$\rho := (\alpha_1 \to \beta_1, \alpha_2 \to \beta_2, ..., \alpha_5 \to \beta_5).$$

Hence if $\left(\prod_{i=1}^{\ell} g_i^{x_{k_i}} \right) h$ α-computes f then

$$\rho^{-1} \left(\prod_{i=1}^{\ell} g_i^{x_{k_i}} \right) h \rho$$

β-computes f. We can write this as in Definition 7.28 by Exercise 7.29. **QED**

Lemma 7.33. If $f : \{0,1\}^n \to \{0,1\}$ is α-computable by a group program of length ℓ, so is $1 - f$.

Proof. First apply the previous lemma to α^{-1}-compute f. Then replace the h in Definition 7.28 with $h\alpha$. **QED**

Lemma 7.34. If f is α-computable with length ℓ and g is β-computable with length ℓ then $(f \wedge g)$ is $(\alpha\beta\alpha^{-1}\beta^{-1})$-computable with length 4ℓ.

Proof. Concatenate 4 programs computing as follows:

- α-compute f
- β-compute g

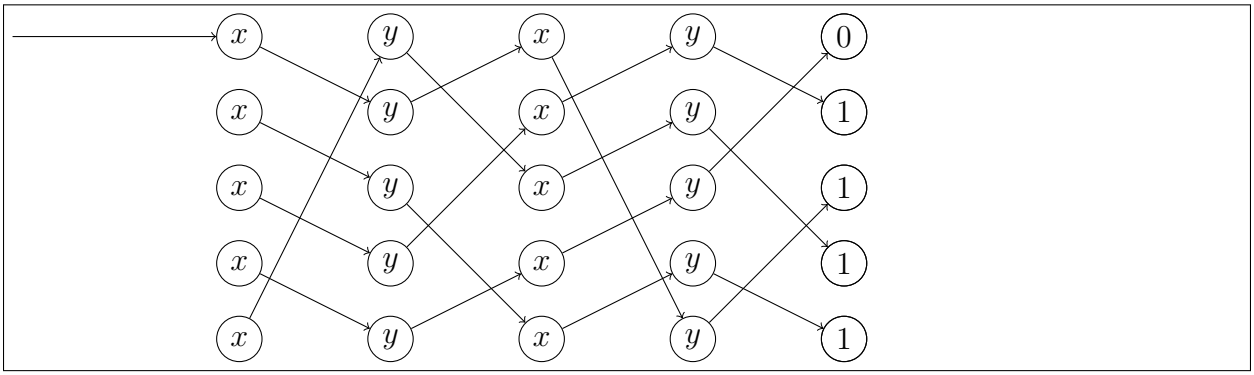


Figure 7.2: The width-5 permutation branching program for And on two bits x and y from the proof of Fact 7.35. All edges drawn have label 1. Every variable node has an edge with label 0 going to the corresponding node in the next column on the right.

- α^{-1}-compute f
- β^{-1}-computes g.

If $f(x) = g(x) = 1$ then the concatenated program evaluates to $(\alpha\beta\alpha^{-1}\beta^{-1})$; otherwise it evaluates to 1. **QED**

As mentioned earlier, to iterate this lemma we need to check that the commutator itself is a cycle.

Fact 7.35. There cycles α and β in S_5 whose commutator $\alpha\beta\alpha^{-1}\beta^{-1}$ is also a cycle.

Proof. Let $\alpha := (12345)$, $\beta := (13542)$, we can check that $\alpha\beta\alpha^{-1}\beta^{-1}$ is a cycle. **QED**

Exercise 7.36. Check it.

For an illustration, see figure 7.2. Note that it is easy to compute And with a branching program, but the gain is that this is a *permutation* branching program.

Proof of first part of Theorem 7.30. By induction on d using previous lemmas. **QED**

Exercise 7.37. Give the details of the proof.

The results work over other groups as well. In particular, we can recover the results in section §7.5, see Corollary 7.24, working with the group $GL(3, 2)$ of invertible 3×3 matrices over $\mathbb{F}_2$, see Problem 7.2. This group has size 168. The smallest group size that allows for this simulation is 60, met by the alternating group $A_5 \subseteq S_5$.

7.7 The power of NC^0: Cryptography

The results in this section came as a shock, and demonstrate the unsuspected power of NC^0 (and perhaps the weakness of our intuition).

Let us first illustrate the technique in the simplest possible setting. Parity is not in NC^0, since it depends on all the input bits. Nevertheless, NC^0 can do something almost as good: It can generate uniformly distributed input-output pairs of parity. In other words, this is a problem that NC^0 can't solve, but for which nevertheless NC^0 can create instances together with their solutions.

Exercise 7.38. Show that there are constant-depth circuits $C : [2]^n \to [2]^{n+1}$ whose output distribution over a uniform input is the same as $(X, \mathrm{parity}(X))$ for uniform $X \in [2]^n$.

This technique extends to any group. Given the results in the previous sections (such as Theorem 7.30) it is natural to apply it to S_5.

One-way functions are functions that are easy to compute (e.g., in power time) but hard to invert (e.g., even in some superpower time). Most modern cryptography is based on the existence of one-way functions. The theory of one-way functions is rich, see the notes and section 11.5.2. Common candidate one-way functions are computable in NC^1. For example, a candidate one-way function views x of $2n$ bits as two naturals a, b of n bits and outputs their product. This candidate is in NC^1 by Theorem 7.16 and it can be related to the hardness of factoring. It is not clear how to implement this and other candidates in complexity classes that cannot perform multiplication. Yet surprisingly we will show that the existence of one-way functions in NC^1 implies the existence of one-way functions in NC^0. We illustrate this result in a basic setting which suffices to make our points.

We say that a function $f : X \subseteq [2]^n \to [2]^m$ is ϵ-hard to invert for size s if for every circuit C of size s we have

$$\mathbb{P}_{x \in X} \left[f(C(f(x))) = f(x) \right] \leq \epsilon,$$

i.e., given a uniform image $f(x)$ the circuit cannot compute a preimage x' except with prob. ϵ.

Theorem 7.39. [One-way functions in $\mathrm{NC}^1 \Rightarrow$ one-way functions in NC^0.] Suppose $f : X \subseteq [2]^n \to [2]^m$ is computable by circuits of depth d and is ϵ-hard to invert for size s. Then there is $f' : X' \subseteq [2]^{n'} \to [2]^{m'}$ that is computable by circuits of depth c and is ϵ-hard to invert for size s', where $n' \leq n + m \cdot c^d$, $m' \leq m \cdot c^d$ and $s' \geq s - m \cdot c^D$.

Note that if the depth d is logarithmic then the blow-up in parameters is only a power. We emphasize that the depth of f' is a constant independent of d.

The main technique for proving this is to use that group products are complete for NC^1 (Corollary 7.24), and then use that group products enjoy *random self-reducibility*. The latter means that given a sequence of group elements

$$g_1, g_2, g_3, \ldots, g_n$$

one can easily sample a sequence of group elements that is uniform except that it has the same product as the g_i. To do this, pick $r_1, r_2, \ldots, r_{n-1}$ uniformly in G and output

$$g_1 r_1^{-1}, r_1 g_2 r_2^{-1}, r_2 g_3 r_3^{-1}, \ldots, r_{n-1}^{-1} g_n. \tag{7.1}$$

Note that the first $n-1$ elements are uniformly distributed, while the product of the n elements is the same as $\prod g_i$. Thus, we are "masking" the group program, leaving only its output unchanged. If we think of the program as encoding some "secret" (for example, the input to the one-way function) this allows us to reveal a "masked" version of the program which has the same functionality, but hides the input. Moreover, this rerandomization is done *locally*, and so can be implemented in NC^0. We will later see other applications of this re-randomization property (e.g., Problem 8.1).

Proof. By Theorem 7.30, each output bit of f is α-computable by a group program of length $D := c^d$ over S_5. Note that the group program multiplies to 1 or α, so we can think of it as a boolean value. Setting $h = 1$ in Definition 7.28 without loss of generality, we can write

$$f : x \to \begin{bmatrix} \prod_{i \in [D]} g_{1,i}^{x_{k_{1,i}}} \\ \prod_{i \in [D]} g_{2,i}^{x_{k_{2,i}}} \\ \cdots \\ \prod_{i \in [D]} g_{m,i}^{x_{k_{m,i}}} \end{bmatrix}.$$

Consider the matrix-valued function which outputs each term in the product:

$$f'' : x \to \begin{bmatrix} g_{1,1}^{x_{k_{1,1}}} & g_{1,2}^{x_{k_{1,2}}} & \cdots & g_{1,D}^{x_{k_{1,D}}} \\ g_{2,1}^{x_{k_{2,1}}} & g_{2,2}^{x_{k_{2,2}}} & \cdots & g_{2,D}^{x_{k_{2,D}}} \\ & & \cdots & \\ g_{m,1}^{x_{k_{m,1}}} & g_{m,2}^{x_{k_{m,2}}} & \cdots & g_{m,D}^{x_{k_{m,D}}} \end{bmatrix}.$$

This function is computable in constant depth. In fact each output element only depends on one input bit. However, it is clearly not one way: The output reveals the input.

So we simply rerandomize f'' using equation (7.1):

$$f' : x, r \to \begin{bmatrix} g_{1,1}^{x_{k_{1,1}}} r_{1,1}^{-1} & r_{1,1} g_{1,2}^{x_{k_{1,2}}} r_{1,2}^{-1} & \cdots & r_{1,D-1} g_{1,D}^{x_{k_{1,D}}} \\ g_{2,1}^{x_{k_{2,1}}} r_{2,1}^{-1} & r_{2,1} g_{2,2}^{x_{k_{2,2}}} r_{2,2}^{-1} & \cdots & r_{2,D-1} g_{2,D}^{x_{k_{2,D}}} \\ & & \cdots & \\ g_{m,1}^{x_{k_{m,1}}} r_{m,1}^{-1} & r_{m,1} g_{m,2}^{x_{k_{m,2}}} r_{m,2}^{-1} & \cdots & r_{m,D-1} g_{m,D}^{x_{k_{m,D}}} \end{bmatrix},$$

where f' takes as input x as well as a matrix r of $m \times (D-1)$ group elements in S_5.

Note that for every x the output distribution of f' is a uniform matrix whose columns multiply to $f(x)$. So f' is a kind of *randomized encoding* of f. Now, f' is still computable in constant depth, yet it is no easier to invert than f. Indeed, suppose a circuit C' inverts f' with probability ϵ:

$$\mathbb{P}_{x,r}[f'(C'(f'(x,r))) = f'(x,r)] \geq \epsilon.$$

Then consider the distribution on circuits C that on input $y = f(x) \in [2]^m$ first compute a uniform matrix M_y whose columns multiply to y, then output $C'(M_y)$. Note a row of this matrix can be computed by constant-depth circuits of size cD following equation (7.1) (i.e., rerandomized a tuple of elements which are all 1 except one equal to the corresponding bit of y). Each of these circuits has size at most the size of C' plus cmD, and the bound on s' follows.

For every x, the distribution of $M_{f(x)}$ is the same as the distribution of $f'(x, r)$ over uniform r. Hence C' will output a pre-image of f' with probability $\geq \epsilon$. The first n bits of this preimage are a preimage of f. Finally, we can fix the randomness while keeping the inversion probability.

The bounds on n' and m' are by definition. **QED**

Exercise 7.40. Explain which steps in the proof of one-wayness of f' breaks down for f''.

These results can be improved to show that even one-way function in FL (or other classes) imply one-way functions in NC^0, see the notes. Whether the same conclusion follows from the existence of one-way functions in FP remains open.

7.8 Word circuits

In this section we prove a powerful extension of the results in section §7.5 to computation over words. This extension is the main step in the proof of the space-efficient simulation of circuits by branching programs (Theorem 6.10).

Definition 7.41. A *word circuit* (or *word straight-line program)* with word-size w and n input words is a sequence of instructions where instruction i is of the type

$$g_i := g(t, t')$$

where $g : [2]^w \times [2]^w \to [2]^w$ is a function and each of the terms t and t' is either a previous g_j with $j < i$ or an input word x_j with $j \in [n]$. Each instruction can have a different function g. The g_i are called *gates*. A circuit computes a function from $([2]^w)^n$ to $[2]^w$ in the natural way, executing the instructions in order. The last gate is the output. The *depth* of the circuit is defined as in Definition 7.1.

Next we give the memoryless version.

Definition 7.42. A *memoryless straight line word program* on n words of size w is like a memoryless straight line program (Definition 7.20) except that the registers have w bits and we allow instructions

$$R_i += g(t, t')$$

where $g : [2]^w \times [2]^w \to [2]^w$ is a function and each of the terms t and t' is either a register R_k with $k \in [3]$ or an input word x_k with $k \in [n]$. Each instruction can have a different function g.

In the extension, we allow the straight-line program to have word size w' slightly larger than the word size w of the circuit, but we still view it as computing a function on words of w bits (e.g. by padding them with zeroes).

Theorem 7.43. Suppose $f : ([2]^w)^n \to [2]^w$ is computable by a word circuit of depth d. Then there is a memoryless straight-line word program with word size $w \log(cw)$ of length $(cw)^d$ for

$$R_i += f,$$

for every $i \in [3]$.

Proof. We can view words as elements of $\mathbb{F}_2^w$. We extend them to a larger word size $w' = wt$ by viewing each bit as an element in $\mathbb{F}_{2^t}$ and the whole word as an element of $\mathbb{F}_{2^t}^w$, for $t := \log w + c$.

We can write (each output bit of) each function f in the circuit as a polynomial over $\mathbb{F}_2$ of degree $2w$. This is done in the brute-force way as a construction in Theorem 2.5. However, now we can also view f as mapping $\mathbb{F}_{2^t}^w \times \mathbb{F}_{2^t}^w \to \mathbb{F}_{2^t}^w$ (note the value on $\mathbb{F}_2^w \times \mathbb{F}_2^w$ hasn't changed). We will give a simulation for such extended functions.

We proceed by induction on d. For $d = 1$ the output is just $g(x_i, x_j)$ which is a type of instruction we have.

For the induction step, we have programs for

$$R_2 += f_b$$

for $b \in [2]$; we seek a program for $f = g(f_0, f_1)$.

Let α be a generator of the multiplicative group of $\mathbb{F}_{2^t}$ (Fact A.33). The program is this: For each $i \in [2^t - 1]$ we perform

$$\begin{aligned}
R_0 \times &= \alpha^i R_0 \\
R_0 += &\ f_0 \\
R_1 \times &= \alpha^i R_1 \\
R_1 += &\ f_1 \\
R_2 += &\ f(R_0, R_1). \\
R_1 += &\ f_1 \\
R_1 \times &= \alpha^{-i} R_1 \\
R_0 += &\ f_0 \\
R_0 \times &= \alpha^{-i} R_0
\end{aligned}$$

At the end, registers R_0 and R_1 haven't changed, while R_2 has had added the value

$$\sum_{i \in [2^t - 1]} g(\alpha^i R_0 + f_0, \alpha^i R_1 + f_1).$$

We claim that this value equals $g(f_0, f_1)$. It suffices to prove the claim for each output bit. Specifically, we claim that for each polynomial p of degree $\leq 2w$ in $2w$ variables and every $z, y \in \mathbb{F}_{2^t}^{2w}$ we have

$$\sum_{i \in [2^t - 1]} p(\alpha^i z + y) = p(y). \tag{7.2}$$

To match parameters, set $z = (R_0, R_1)$ and $y = (f_0, f_1)$. Note that α^i is a scalar $\in \mathbb{F}_{2^t}$ multiplying the vector $z \in \mathbb{F}_{2^t}^{2w}$ component-wise. For example, we are about to use the fact that coordinate j of $(\alpha^i z)$, denoted $(\alpha^i z)_j$, equals $\alpha^i (z_j)$.

It suffices to verify equation (7.2) when p is a monomial of degree $m \leq 2w$, w.l.o.g. the product of the first m variables). In this case the lhs in equation (7.2) becomes

$$\begin{aligned}\sum_{i \in [2^t-1]} \prod_{j \in [m]} (\alpha^i z + y)_j &= \sum_{i \in [2^t-1]} \sum_{S \subseteq [m]} \left(\prod_{j \in S} \alpha^i z_j \right) \left(\prod_{j \notin S} y_j \right) \\ &= \sum_{S \subseteq [m]} \left(\prod_{j \in S} z_j \right) \left(\prod_{j \notin S} y_j \right) \left(\sum_{i \in [2^t - 1]} \alpha^{Si} \right).\end{aligned}$$

When $S \neq \emptyset$, we have $\alpha^S \neq 1$ because $S \leq 2w$ and α is a generator of a group of size $2^t - 1 \geq 2w$. and by Fact A.4 the last sum equals $(1 - \alpha^{S(2^t-1)})(1 - \alpha^S)$ which is 0 since raising the element α^S to the order $2^t - 1$ of the multiplicative group of $\mathbb{F}_{2^t}$ gives 1 (Fact A.29).

When $S = \emptyset$ the right-hand sum equals $2^t - 1 = 1$. **QED**

Exercise 7.44. Extend Theorem 7.43 to the case of larger fan in. Specifically, for circuits with word size w and fan-in r give a simulation using $r + c$ registers, word size $w \log(cwr)$, and length $(cwr)^d$.

7.8.1 Simulating circuits with square-root space

In this section we use Theorem 7.43 to prove Theorem 6.10. First, we describe a straightforward proof of a result that is weaker but carries the same conceptual message: We prove that we can simulate circuits of size s using space s^{1-c}. Given such a circuit, divide its gates into blocks of consecutive b gates. Each block is only connected to $\leq 2b$ other blocks. Hence we can view this as a word circuit with $w = b$ and fan-in $\leq cw$. The depth of this circuit is $d \leq s/b + 1$. Applying Theorem 7.43 we obtain a simulation with $w + c$ registers of $cw \log w$ bits and length w^{cd}. Each instruction is computable by a branching program of size w^{cw^2}. So overall the size is $w^{cw^2} \cdot w^{cd}$. To balance the exponents, we want $w^2 = d$; so we set $w := cs^{1/3}$ and obtain a total branching-program size of $2^{cs^{2/3} \log s}$.

The better size bound can be obtained in a similar fashion but transforming the circuit into a word circuit with fan-in 2.

7.9 Uniformity

Our Definition 7.1 is for *non-uniform* circuits. Just like PCkt is the non-uniform analogue of P and BrL of L, there is a uniform analogue of NC^1. However, it is somewhat technical, so a common choice is to define uniform NC^i as the class of functions computable by NC^i circuits that can be constructed in FL. I.e., there is a function in FL that given 1^n outputs the circuit on n input bits. This is not completely satisfactory either, but suffices for most of the theory. In particular, all the constructions of circuits in this section turn out to be uniform in this sense. For example, one can verify that the containment $\mathrm{NL} \subseteq \mathrm{NC}^2$, Theorem 7.5, holds for uniform circuits.

For other classes of circuits that we will see later the notion of uniformity is more natural.

7.10 Problems

Problem 7.1. Show that Search-3Sat reduces to 3Sat in NC^1.

Problem 7.2. (cf Theorem 9.53) Let G be a finite non-solvable group. By Fact A.28 it has a non-trivial subgroup H whose commutator subgroup (i.e., the subgroup generated by commutators) is itself. Prove that any function $f : [2]^n \to [2]$ computable by depth-d circuits is α-computable by a program of length $\leq c_G^d$ over G, for any $\alpha \in H$.

Problem 7.3. Prove that iterated multiplication of 3×3 matrices of naturals can be computed by power-size circuits of depth $c \log n \log^* n$. Recall $\log^* n$ is the number of times we need to apply log to n to obtain a value < 2.

Guideline:

(1) Show that it suffices to compute the value mod m for an n-bit number m.

(2) Use remaindering representation to reduce this problem to the problem where m has $c \log n$ bits.

(3) Group the matrices into blocks of $c \log n$ to reduce this problem to many instances of the same problem as in (1).

(4) Formulate and solve a recursion for the depth given by this approach.

Problem 7.4. Show that the iterated product of matrices (of any dimension) over $\mathbb{F}_2$ is in CL (Definition 6.55).

7.11 Notes

Arithmetic in NC^1 was first performed in [50]. Matrix inversion in NC^2, Theorem 7.8, is from [97]. As we saw this algorithm relies on computing the characteristic polynomial. The approach in [97] and in Theorem 7.8 works over fields of large enough characteristic. Subsequently, [59] showed how to compute this (and hence matrix inversion) over any field via a different approach.

Theorem 7.10 was proved in [322] and apparently rediscovered in [67].

The lower bound for parity in Corollary 7.15 is from [331]. This is the first power lower bound for formulas, and apparently the paper that introduced the restrict and simplify method. The tight quadratic bound for parity is from [211]. A series of works has proved increasingly stronger bounds for other functions, culminating in a cubic lower bound [164]. The bounds are obtained by strengthening the restrict-and-simplify Lemma 7.13 and combining it with other ideas.

The "2-out-of-3" idea, in the proof of Theorem 7.16, is from [116].

Group programs were introduced in the 60's in [245, 220] where it was shown that any function on G^n can be computed on a simple non-abelian group. The construction involves the use of commutators to compute And, similar to the classic puzzle of hanging a picture with two nails so that removing any one nail makes it fall [323].

Group programs were rediscovered in the 80's as permutation branching programs of constant width. Using essentially the same construction in [245, 220], it was shown in [255] that NC^1 equals the set of functions computable by power-length group programs, over any non-solvable group. ([255] considers functions over the domain $[2]^n$, so can allow the more general class of non-solvable groups instead of the more restrictive simple non-abelian in [245, 220], where the domain is G^n.) After [255], several related simulations were discovered, such as [58]. section §7.6 and Problem 7.2 are from [255]. Theorem 7.21 is a variant of this result from [58]. The proof we presented follows [90]. The extension to word circuits is from [91] (they do not explicitly word circuits though), see also [134].

The proof I presented of Theorem 6.10 is based on the follow-up [307].

Theorem 7.39 is from [28]. They prove a stronger result, where f can be even in NL or $\oplus$L, and f' remains total if f is. Moreover, their techniques extend to other objects such as cryptographic pseudorandom generators and one-way permutations. For an exposition see [365].

Problem 7.3 is from [199], the outline is from [19].

Chapter 8

Majority

> *Once upon a time two daughter sciences were born to the new science of cybernetics. One sister was natural, with features inherited from the study of the brain, from the way nature does things. The other was artificial, related from the beginning to the use of computers. Each of the sister sciences tried to build models of intelligence, but from very different materials. The natural sister built models (called neural networks) out of mathematically purified neurones. The artificial sister built her models out of computer programs.*
>
> *In their first bloom of youth the two were equally successful and equally pursued by suitors from other fields of knowledge. They got on very well together. Their relationship changed in the early sixties when a new monarch appeared, one with the largest coffers ever seen in the kingdom of the sciences: Lord DARPA, the Defense Department's Advanced Research Projects Agency. The artificial sister grew jealous and was determined to keep for herself the access to Lord DARPA's research funds. The natural sister would have to be slain.*
>
> *The bloody work was attempted by two staunch followers of the artificial sister [...], cast in the role of the huntsman sent to slay Snow White and bring back her heart as proof of the deed. Their weapon was not the dagger but the mightier pen, from which came a book* – Perceptrons – *purporting to prove that neural nets could never fill their promise of building models of mind:* only computer programs could do this. *Victory seemed assured for the artificial sister. And indeed, for the next decade all the rewards of the kingdom came to her progeny, of which the family of expert systems did best in fame and fortune.*
>
> *But Snow White was not dead. What [they] had shown the world as proof was not the heart of the princess; it was the heart of a pig.*

Can *you* kill Snow White? After an AI winter, spectacular progress in artificial intelligence has made it even more apparent that (a type of) small-depth circuits have amazing capabilities ranging from playing chess, to talking, driving cars, and so on. *(Artificial) neural networks* are a computing paradigm that is inspired by the human brain and gives rise to

small-depth circuits. Much of the research focus in artificial intelligence is on *training* such networks, but here we will focus only on their performance after training – a non-uniform model of computation.

Definition 8.1. A *threshold* $f : [2]^n \to [2]$ is a function of the form

$$f(x) := \left[\sum_{i \in [n]} w_i x_i \geq t \right].$$

where the $w_i \in \mathbb{R}$ are *weights* and $t \in \mathbb{R}$ is a *threshold*.

A *threshold circuit* is a circuit made only of threshold gates with unbounded fan-in. We denote by ${\rm TC}_{\mathbb{R}}{}^0$ the class of functions $f : X \subseteq [2]^* \to [2]^*$ for which there is $a \in \mathbb{N}$ s.t. for every $n \geq 1$ the function on inputs of length n is computable by a threshold circuit of depth a and size n^a. Finally, we denote by ${\rm TC}^0$ the same class except that the parametersw_i and t in each threshold gate are integers with n^a bis.

Exercise 8.2. Prove that any function on $n = 1$ bit is computable by a threshold circuit of size 1.

Exercise 8.3. Prove ${\rm TC}^0 \subseteq {\rm NC}^1$.

One can define ${\rm TC}^1$ etc. as in Definition 7.1, but we won't need that.

One can show that without loss of generality the weights and and the threshold in a circuit of size s can be taken to be integers with absolute value $\leq 2^{cs}$. This can even be reduced to s^c at the cost of increasing the size by a power and the depth by one. In particular:

Theorem 8.4. ${\rm TC}_{\mathbb{R}}{}^0 = {\rm TC}^0$.

A striking instantiation of the grand challenge is that it is open to prove impossibility results for threshold circuits of depth 2. Even the status of very simple-looking functions is unknown:

Open question 8.5. Is inner product mod 2 in ${\rm TC}^0_{\mathbb{R}}$ (i.e., a threshold of thresholds)?

As usual, a good explanation for this ignorance is the power of TCs, of which we give several examples next.

Exercise 8.6. A function $f : [2]^* \to [2]$ is *symmetric* if it only depends on the weight of the input. Prove that any symmetric function is in ${\rm TC}^0$. Hint: First do that case when $f(x) = 1$ iff the weight of x is exactly t.

8.1 The power of TC^0: Arithmetic

Theorem 8.7. The arithmetic problems in Theorem 6.12 are in ${\rm TC}^0$.

The proof follows closely that for ${\rm NC}^1$ in section §7.4 (which in turn was based on that for FL). Only iterated addition requires a new idea.

Exercise 8.8. Prove that iterated addition of naturals is in ${\rm TC}^0$. (Hint: Write input as $n \times n$ matrix, one number per row. Divide columns into blocks.)

8.2 Neural networks

Neural networks are made of a "small" number of layers, each consisting of a large number of *(artificial) neurons*. A neuron is a generalization of a threshold: We allow real numbers as input, and rather than checking if the sum surpasses a threshold, we compute an *activation function* of the sum. Several activation functions are considered. In first approximation, we can think of σ as being simply a threshold. Another popular choice is the $ReLU$ (defined next).

Definition 8.9. A *neuron* is a function mapping $(x_1, \ldots, x_m) \in \mathbb{R}^m \to \sigma(\sum_i w_i x_i) \in \mathbb{R}$, where $w_i \in \mathbb{R}$ are weights and $\sigma : \mathbb{R} \to \mathbb{R}$ is an *activation function.* The $ReLU$ (rectified linear unit) activation function is $\sigma(s) := \max\{0, s\}$. A *neural network* is like a threshold circuit but with neurons instead of thresholds.

Using the fact that TC^0 can compute arithmetic, one can show that neural networks can be simulated by threshold circuits (on integer inputs, and when all the weights are integer).

8.3 Amplifying lower bounds by means of self-reducibility

Another great question is whether $\mathrm{TC}^0 = \mathrm{NC}^1$. It is known that Parity cannot be computed by majority circuits of size $n^{1+\epsilon}$ and depth $c \log 1/\epsilon$. Note that *the size bound approaches n exponentially fast with the depth.* This tradeoff is known to be tight for parity, and a natural question is whether we can improve on it and prove stronger bounds, say for iterated multiplication of elements from a group such as S_5, see section §7.6. The following result shows that, in fact, slightly stronger bounds would already imply $\mathrm{TC}^0 \neq \mathrm{NC}^1$. We state this result for TC^0 and S_5 but the proof does not use anything specific about the types of gates or the group: similar results holds for various circuit classes and associative problems.

Theorem 8.10. Suppose $\mathrm{TC}^0 = \mathrm{NC}^1$. Then there is $a \in \mathbb{N}$ s.t. for every ϵ iterated product of n elements in S_5 can be computed by threshold circuits of depth $d' := a \log 1/\epsilon$ and size $d' n^{1+\epsilon}$.

Proof. The problem is in NC^1. By assumption, it has threshold circuits of depth d and size n^k for constants d and k. Exploiting the associativity of the problem, we compute the product recursively according to a regular tree. The root is defined to have level 0. At Level i we compute n_i products of $(n^{1+\epsilon}/n_i)^{1/k}$ elements. At the root ($i = 0$) we have $n_0 = 1$.

By the assumption, each product at Level i has threshold circuits of size $n^{1+\epsilon}/n_i$ and depth d. Hence Level i can be computed by threshold circuits of size $n^{1+\epsilon}$ and depth d.

We have the recursion

$$n_{i+1} = n_i \cdot (n^{1+\epsilon}/n_i)^{1/k}.$$

The solution to this recursion is $n_i = n^{(1+\epsilon)(1-(1-1/k)^i)}$, see below.

For $i = ck\log(1/\epsilon)$ we have $n_i = n^{(1+\epsilon)(1-\epsilon^2)} > n$; this means that we can compute a product of $\geq n$ elements, as required.

Hence the total depth of the circuit is $d \cdot ck\log(1/\epsilon)$, and the total size is $\leq$the depth times $n^{1+\epsilon}$.

It remains to solve the recurrence. Letting $a_i := \log_n n_i$ be the exponent of n_i we have:

$$\begin{aligned} a_0 &= 0 \\ a_{i+1} &= a_i(1-1/k) + (1+\epsilon)/k = a_i\beta + \gamma \end{aligned}$$

where $\beta := (1-1/k)$ and $\gamma := (1+\epsilon)/k$.

This gives

$$a_i = \gamma \sum_{j \leq i} \beta^j = \gamma \frac{1-\beta^{i+1}}{1-\beta} = (1+\epsilon)(1-\beta^{i+1}).$$

QED

8.4 The power of Majority: Boosting correlation

An impossibility result is just one of the things we can ask about a model. A natural strengthening is proving an *average-case* impossibility result. One motivation for this quest is that an impossibility result may be irrelevant to instances that occur "in nature," if the latter follow a specific distribution. Another motivation comes from cryptography, where a random secret key should give a hard cryptosystem.

So, for a distribution D on inputs we can ask how many mistakes are made by any function in a class F to compute h over D. We call this the *hardness* of h. If h is boolean, either the constant 0 or 1 compute it w.p. $\geq 1/2$. So the maximum hardness of h is $1/2$, if these constants belong to F as is typically the case.

When the hardness approaches $1/2$, which is the important setting where functions in F can't compute h much better than random guessing, it is more natural to consider the distance between the hardness parameter and $1/2$, which is called *correlation*. We now define these quantities which will be used extensively in several subsequent chapters.

Multiple occurrences of D in the same expression indicate the same sample from D.

Definition 8.11. Let D a distribution over the input set X.

We say that $h : X \to [2]$ is δ-*hard* for $f : X \to [2]$ w.r.t. D (or over D) if $\mathbb{P}[f(D) \neq h(D)] \geq \delta$, and that it is δ-hard for F w.r.t. D if it is δ-hard for every $f \in F$.

The *correlation* between h and f w.r.t. D is

$$|\mathbb{E}[(-1)^{f(D)+h(D)}]| = |\mathbb{P}[f(D) = h(D)] - \mathbb{P}[f(D) \neq h(D)]| = |1 - 2\mathbb{P}[f(D) \neq h(D)]|.$$

We also introduce the notation $\mathbb{E}e[y]$ for $\mathbb{E}[(-1)^y]$ which allows us to write correlation as

$$|\mathbb{E}e[f(D) + g(D)]|\,.$$

If the range of functions is $\{-1, 1\}$ instead of $[2]$ we can write correlation using multiplicative notation:

$$\left|\mathbb{E}[f(D)g(D)]\right|.$$

We say h has correlation $\leq \delta$ with F w.r.t. D if it has correlation $\leq \delta$ w.r.t. D with any $f \in F$.

If D is not specified it is assumed to be the uniform distribution.

Thus the correlation (or hardness) between two functions is a measure of how often the functions agree. To illustrate parameters, if $h = f$ or $h = 1 - f$ then the correlation is one. The hardness is no larger than 0 in the first case and is 1 in the latter. Typical complexity classes are closed under complement, in which case the fact that we take absolute values in the correlation is immaterial and hardness and correlation are equivalent. If h and f disagree on exactly one input in $[2]^n$ the correlation is $1 - 1/2^n - 1/2^n = 1 - 2/2^n$ and the hardness is $1/2^n$. If they disagree on exactly half the inputs the correlation is zero and the hardness is $1/2$. For any f, most functions h have correlation close to 0 with f.

Exercise 8.12. [Average-case/correlation version of Theorem 2.9] Prove that for every $n \geq c$ there are functions $h : [2]^n \to [2]$ that have correlation 2^{-cn} with circuits of size 2^{cn}.

At first sight, average-case hardness and correlation bounds seem stronger than impossibility. In fact, *impossibility* results are *equivalent* to strong correlation bounds! This is where TC^0 kicks in: The equivalence holds for models that can compute majority.

Theorem 8.13. [Computing $\iff$ correlating over any distribution] Let F be a set of functions mapping $[2]^n$ to $[2]$; and let h be a function.

[$\Leftarrow$] Suppose for every distribution D on $[2]^n$ there is $f \in F$ s.t. $\mathbb{E}e[f(D) + h(D)] \geq \epsilon$. Then there exist cn/ϵ^2 functions $f_i \in F$ s.t.

$$h = \mathrm{Majority}(f_1, f_2, \ldots, f_{cn/\epsilon^2}).$$

[$\Rightarrow$] Suppose $h = \mathrm{Majority}(f_1, f_2, \ldots, f_t)$ for odd t. Then for any distribution D there is i s.t. $\mathbb{E}e[f_i(D) + h(D)] \geq 1/t$.

Exercise 8.14. Prove the [$\Rightarrow$].

In particular, for classes C that include majority and are suitably closed under composition, such as CktP, NC^1, or TC^0, we get that $h \notin \mathrm{C}$ iff there is a distribution D for which any $f \in \mathrm{C}$ has correlation $\leq 1/n^a$ for any a. In other words, *superpower correlation bounds for some distribution are necessary and sufficient for superpower impossibility.*

In the above theorem we need correlation under *every* distribution. Jumping ahead, in section 11.3.2 we will study a similar connection but under the *uniform* distribution. The proofs are related, and they still rely on majority.

We now develop machinery to understand and prove (the [$\Leftarrow$] in) Theorem 8.13. A useful viewpoint, here and elsewhere, is the *equivalence between randomness in the input and having it in the model.* We state it next as a corollary because we obtain (the difficult direction

of) it as a corollary to a powerful and basic result, Theorem A.23, known under a variety of names such as *the min-max theorem from game theory, or linear-programming duality, theorem of the alternatives for linear systems, etc.*

Corollary 8.15. Let h be a function, and F a set of functions. There is a distribution over F s.t. $\mathbb{E}e[F(x) + h(x)] \geq \alpha$ for every input x iff for every distribution D over inputs there is $f \in F$ s.t. $\mathbb{E}e[f(D) + h(D)] \geq \alpha$.

Exercise 8.16. Prove the "only if" direction of Corollary 8.15.

The [$\Leftarrow$] in Theorem 8.13 follows from Corollary 8.15 and tail bounds (Lemma A.15), similarly to the proof of the error-reduction Theorem 3.3 for BPTime.

Proof of [$\Leftarrow$] in Theorem 8.13. Use Corollary 8.15 to get a distribution on F. The majority of cn/ϵ^2 samples from F has error $< 2^{-n}$ (see the proof of Theorem 3.3 or Lemma A.15). By a union bound (as in the proof of Theorem 3.16) we can fix the samples to functions f_i so that the majority is correct on every input of length n. **QED**

By Theorem 8.13 to prove impossibility results for circuits for depth-2 TC^0 it suffices to prove correlation bounds with depth-1 TC^0 circuits, i.e., a single majority gate. Such correlation bounds will be proved in Theorem 13.15 for inner product modulo 2. So it follows that inner product modulo 2 is not in depth-2 TC^0 (cf Question 8.5).

8.5 Uniformity

As in Chapter 7, our definition of TC^0 is for non-uniform circuits. Also as in section §7.9, one can consider FL-uniform TC^0 circuits. However, the right uniform analogue of TC^0 is known as FOM, for *first-order logic with majority*. Let us first define FO. A *first-order (FO) formula* is made with the standard connectives $\neg, \wedge, \vee$, quantifiers $\exists, \forall$, a predicate X, variables $x, y, \ldots$, and the relations $<, +, \times$. Here $+$ is a predicate $+(x, y, z)$ on three variables which is true iff $x + y = z$, and similarly for $\times$. We also denote by FO the set of functions $f : [2]^* \to [2]$ for which there is an FO formula ϕ such that $f_n(x) = 1$ iff ϕ is true when interpreted over the domain $[n]$ and X encodes the input x via $X(i) = x_i$. FO is the uniform counterpart of AC^0, a class we encounter in Chapter 9. First-order with majority (FOM) formulae have in addition majority quantifiers M. The formula $Mx(\phi(x))$ is true if for $\geq n/2$ choices of $x \in [n]$ the formula $\phi(x)$ is true.We also denote by FOM the set of functions computable by FOM formulae, similarly to FO.

As for word programs, Definition 1.1, these definitions are somewhat redundant and arbitrary. Several other choices of relations give the same functions. We emphasize that functions in FO and FOM are each computed by a single formula that works for every input length, just like a word program is a fixed program that works for every input length. For a very simple example, the Or function can be written as the formula

$$\forall i \, (\neg X(i).)$$

For a slightly less trivial example, we show that computing if every 1 is followed by a 0 is in FO. The formula for this is

$$\forall i \left(\exists j \left(j > i \Rightarrow X(i) \Rightarrow \neg X(i+1)\right)\right).$$

The existence of j such that $j > i$ guarantees that $i < n-1$ so that $i+1$ is well defined.

Exercise 8.17. Show how to express $X(i+1)$ without addition.

These examples didn't use the majority quantifier. For an example that uses it see Problem 8.4. It turns out that most of the functions shown to be in TC^0 in this section are in fact in FOM. In particular, the arithmetic problems in Theorem 6.12 are in FOM (see the notes).

In terms of complexity classes, one can show that $\mathrm{FOM} \subseteq \mathrm{TC}^0$ and $\mathrm{FOM} \subseteq \mathrm{L}$.

Exercise 8.18. Show this.

By the space hierarchy (see section §6.4) we have $\mathrm{FOM} \neq \mathrm{PSpace}$. Other than that, just like for TC^0, the power of FOM is not known. While it is widely believed to be a strict subclass of L, the following is open.

Open question 8.19. Is FOM = PH?

8.6 Problems

Problem 8.1. Suppose there is $f \in \mathrm{TC}^0$ s.t. $\mathbb{P}_{g_i \in S_5}[f(g_1, g_2, \ldots, g_n) = \prod_i g_i] \geq 0.51$. Show $\mathrm{NC}^1 = \mathrm{TC}^0$.

Problem 8.2. Prove that iterated product of 2×2 natural matrices is complete for NC^1 vs TC^0. Hint: Use the fact that the group $\mathrm{SL}(2,5)$ of invertible 2×2 matrices over $\mathbb{F}_5$ with determinant 1 is not solvable.

Problem 8.3. Show that the following problems are in TC^0:

(1) Iterated multiplication of univariate polynomials with integer coefficients. (Polynomials are represented as list of coefficients.)

(2) Iterated multiplication of elements in $\mathbb{F}_{2^t}$ (for any field representation as in Example A.32).

(3) Given $x \in \mathbb{F}_{2^t}$ and an n-bit integer a, computing the power $x^a \in \mathbb{F}_{2^t}$, for the specific field representation given by the irreducible polynomial $z^t + z^{t/2} + 1$ where $t = 2 \cdot 3^\ell$ (cf Example A.32).

Problem 8.4. Prove that Parity $\in$ FOM.

Guideline: This is similar to Exercise 8.6, but slightly complicated by the fact that we defined FOM to have majority quantifiers on n values only. First give an FOM formula $\phi(t)$ that is true iff *the first half* of the input bits have weight t. Then give an FOM formula $\phi(t)$ that is true iff (all) the input bits have weight t.

8.7 Notes

The introductory quote is from [279]. The broader history is amusing and may serve as an example of the power of impossibility results. Neural networks were studied since the 40's [246]. In the book [296] it was already pointed out that a constant-depth neural network can compute any function, though there was no good proposal for training such networks. The book referred to in [279] is [254]. What it showed is impossibility results for minimalistic neural networks, consisting of a threshold gate applied to And gates. Specifically, it showed that such models cannot compute parity (or other simple functions), unless the fan-in of the Ands is large. The term "perceptron" is sometimes used to refer to this model, though the meaning varies in the literature. Their result said nothing about networks of larger depth, yet various sources blame it for the onset of the AI winter, during which funding for research in neural networks was particularly hard to find. Perhaps a better explanation is that the hardware, data, and the math weren't yet there. The work [56] "vindicate[d] the reputation of the much maligned perceptron" by showing that small, probabilistic perceptrons can simulate AC^0 (see Definition 9.1).

The $\mathrm{TC}^0_{\mathbb{R}} = \mathrm{TC}^0$ Theorem 8.4 is from [130].

The $n^{1+c\log d}$ lower bound for computing parity by majority circuits is from [190]. A weaker version of Theorem 8.10, which however contains the main message, is from [20]. The stated version is in [85].

The easy direction (Exercise 8.14) of Theorem 8.13 is the "discriminator" lemma in [157].

Corollary 8.15 is from [390] and has been very influential.

For more on FOM and logical characterizations of complexity classes see [256] and the book [186]. [178] shows that the arithmetic problems in Theorem 6.12 are in FOM.

Chapter 9

Alternation

In this chapter we study constant-depth circuits with And, Or, and modular gates of unbounded fan-in. This is perhaps the best understood model of computation, and there is a wealth of far-reaching mathematical techniques surrounding it, as well as deceptively simple open problems. Moreover, this chapter contains an important message regarding the grand challenge. We'll get to it shortly after we define the model.

We have already encountered CNFs in Definition 4.13. A DNF, short for *disjunctive normal form*, is the complement: the output is Or and the other gates are And and are called *terms*. In this chapter we study *alternating circuits* which are a natural generalization of CNFs and DNFs to higher depth.

Definition 9.1. An *alternating circuit* of size s with n input bits and m output bits is a sequence of s assignment instructions to gates g_i where assignment $i \in [s]$ is of one of the following types:

- $g_i := \bigwedge_{j \in [r]} t_j$
- $g_i := \bigvee_{j \in [r]} t_j$

where each t_j is either a gate g_j with $j < i$ or a literal, and $r \in \mathbb{N}$. The last m gates are the output.

We denote by AC^0 the class of functions $f : X \subseteq [2]^* \to [2]^*$ for which there is $d \in \mathbb{N}$ s.t. for every $n \geq 1$ the function on inputs of length n is computable by an alternating circuit of depth d and size n^d.

The name "alternating" comes from the fact that the input gates to an And gate g_i are without loss of generality Or gates g_j, since any And gate g_j can be merged with g_i exploiting that the fan-in is unbounded. And vice versa. Therefore And and Or gates alternate.

Example 9.2. And on n bits is an alternating circuit of depth 1.

DNFs and CNFs are alternating circuits of depth 2.

An (unbounded fan-in) And of DNFs or an Or of CNFs, is an alternating circuit of depth 3.

Addition is in AC^0. We can implement the carry look-ahead construction in the proof of Theorem 7.16.

It will follow from the impossibility results that all the other arithmetic problems in Theorem 6.12 are not in AC^0, see Problem 9.1.

We also consider a generalization where we also allow *modular counting* gates.

Definition 9.3. A Mod-m function $f : [2]^n \to [1]$ is of the form

$$f(x_0, x_1, \ldots, x_{n-1}) = g\left(\sum_{i\in[n]} x_i \mod m\right)$$

for some $g : [m] \to [2]$.

An *alternating circuit with Mod-m gates* is like an alternating circuit but we also have Mod-m gates. The class AC^0-Mod-m is defined analogously to AC^0. Mod-2 functions are also called *parity* and denoted $\oplus$. We denote $\mathrm{ACC}^0 := \bigcup_{m\in\mathbb{N}} \mathrm{AC}^0$-Mod-$m$, for "alternating circuits with counters."

For convenience we also use the term "AC^0 circuit," "AC^0-Mod-2 circuit," "ACC^0 circuit," etc., to indicate alternating circuits with the corresponding types of gates, and with depth bounded by an absolute constant.

We have

$$\mathrm{NC}^0 \subseteq \mathrm{AC}^0 \subseteq \mathrm{AC}^0\text{-Mod-2} \subseteq \mathrm{AC}^0\text{-Mod-6} \subseteq \mathrm{ACC}^0 \subseteq \mathrm{TC}^0$$

with all inclusions being by definition except that last one that holds by Exercise 8.6.

This chapter contains an important message regarding the grand challenge. For alternating circuits, even equipped with parity gates, we can prove exponential impossibility bounds. Even simple functions like Majority cannot be computed in depth d and size $2^{n^{c/d}}$ (Theorem 9.5). At the same time, we will also show (section §9.4) that alternating circuits are powerful enough to simulate L (or even larger classes) with the same tradeoff between depth and size, up to constants. Specifically, any function $f \in \mathrm{L}$ has on inputs of length n alternating circuits of depth d and size $2^{n^{c_f/d}}$. Therefore:

The impossibility results we can prove for alternating circuits are best possible short of proving a major separation such as $\mathrm{L} \neq \mathrm{P}$.

In fact, this message is apparent even for depth 3.

Simple functions like parity require depth 3 circuits of size $2^{c\sqrt{n}}$ (Corollary 9.28). Improving this bound implies new results for branching programs and log-depth circuits (see section §9.4).

This exciting state of affairs is discussed further in Chapter 18.

Another important message of this chapter is that alternating circuits and AC^0 closely correspond to alternating programs and the PH (section §5.5). In particular, AC^0 can be seen as a "scaled down" version of PH. This connection will illuminate the proof that BPP $\subseteq$ PH.

We begin by presenting the impossibility results. To set the stage, let's prove strong results for depth 2, that is, DNFs or CNFs.

Exercise 9.4. Prove that Parity and Majority each require DNFs of size $\geq 2^{cn}$. Hint: What if you have a term with $< n/2$ variables?

Even depth 3 is much more challenging.

9.1 The polynomial method over $\mathbb{F}_2$

In this section we introduce a new and far-reaching technique to prove impossibility results for alternating circuits. This technique departs from the restrict-and-simplify method. Using it, we obtain the following result:

Theorem 9.5. Suppose an alternating circuit of depth $d > 1$ and size s with parity gates computes Majority on n bits. Then $s \geq 2^{cn^{0.5/(d-1)}}$.

The proof uses *the polynomial method* which goes by "simulating" alternating circuits by *low-degree polynomials*. We then show that Majority does not have such simulation. The simulation of circuits by polynomials is not exact, but probabilistic. It works over various fields, and for simplicity we focus on $\mathbb{F}_2$.

Example 9.6. Recall that a polynomial in n variables over $\mathbb{F}_2$ is an object like

$$p(x_1, x_2, \ldots, x_n) = x_1 \cdot x_2 + x_3 + x_7 \cdot x_2 \cdot x_1 + x_2 + 1.$$

Because we are only interested in inputs in $\mathbb{F}_2$ we have $x^i = x$ for any $i \geq 1$ and any variable x, so we don't need to raise variables to powers bigger than one.

The And function can be written as the polynomial

$$\mathrm{And}(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} x_i.$$

The Or function can be written as

$$\mathrm{Or}(x_1, x_2, \ldots, x_n) = 1 + \mathrm{And}(1 + x_1, 1 + x_2, \ldots, 1 + x_n) = 1 + \prod_{i=1}^{n} (1 + x_i).$$

For $n = 2$ we have

$$\mathrm{Or}(x_1, x_2) = x_1 + x_2 + x_1 \cdot x_2.$$

Definition 9.7. A distribution P on polynomials over $\mathbb{F}_2$ computes a function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ with error ϵ if for every x we have

$$\mathbb{P}_P[P(x) = f(x)] \geq 1 - \epsilon.$$

We call P a *probabilistic polynomial*.

The next theorem shows that AC^0-$\oplus$ has low-degree probabilistic polynomials over $\mathbb{F}_2$ (with small error). In particular, it correlates with a low-degree polynomial.

Theorem 9.8. [AC^0-$\oplus$ computable by low-degree probabilistic polynomials.] Let $C : [2]^n \to [2]$ be (the function computed by) an alternating circuit of depth d and size s with parity gates. Then there is a distribution P on polynomials over $\mathbb{F}_2$ of degree

$$(c \log s)^{d-1} \cdot \log 1/\epsilon$$

that computes C with error ϵ.

In particular, there is a polynomial p over $\mathbb{F}_2$ of the same degree such that

$$\mathbb{P}_{x \in [2]^n}[C(x) \neq p(x)] \leq \epsilon.$$

The important point in Theorem 9.8 is that if the depth d is small (e.g., constant) (and the size is not enormous and the error is not too small) then the degree is small as well. For example, for AC^0-$\oplus$ the degree is power logarithmic for constant error.

9.1.1 AC^0 correlates with low-degree polynomials modulo 2

In this section we prove Theorem 9.8.

Lemma 9.9. For every ϵ and n there is a distribution P on polynomials of degree $\log 1/\epsilon$ in n variables over $\mathbb{F}_2$ that computes Or with error ϵ. The same holds for And.

Proof. For starters, pick the following distribution on linear polynomials: For a uniform $A = (A_1, A_2, \ldots, A_n) \in [2]^n$ output the polynomial

$$p_A(x_1, x_2, \ldots, x_n) := \sum_i A_i \cdot x_i.$$

Let us analyze how p_A behaves on a fixed input $x \in [2]^n$:

- If $\mathrm{Or}(x) = 0$ then $p_A(x) = 0$;
- If $\mathrm{Or}(x) = 1$ then $\mathbb{P}_A[p_A(x) = 1] \geq 1/2$.

While the error is large in some cases, a useful feature of p_A is that it never makes mistakes if $\mathrm{Or}(x) = 0$. This allows us to easily reduce the error by taking $t := \log 1/\epsilon$ polynomials p_A and combining them with an Or.

$$p_{A_1, A_2, \ldots, A_t}(x) := p_{A_1}(x) \vee p_{A_2}(x) \vee \cdots \vee p_{A_t}(x).$$

The analysis is like before:

- If $\mathrm{Or}(x) = 0$ then $p_{A_1,A_2,\ldots,A_t}(x) = 0$;
- If $\mathrm{Or}(x) = 1$ then $\mathbb{P}_{A_1,A_2,\ldots,A_t}[p_{A_1,A_2,\ldots,A_t}(x) = 1] \geq 1 - (1/2)^t \geq 1 - \epsilon$.

It remains to bound the degree. Each p_{A_i} has degree 1. The Or on t bits has degree t by Example 9.6. Hence the final degree is $t = \log 1/\epsilon$.

The case of And is analogous and is left as exercise. **QED**

Proof of Theorem 9.8. The "in particular" part follows because averaging over x we have

$$\mathbb{P}_{x,P}[C(x) \neq P(x)] \leq \epsilon.$$

Hence we can fix a particular polynomial p s.t. the probability over x is $\leq \epsilon$.

We now prove the first part of the theorem. We apply Lemma 9.9 with error $\leq \epsilon/s$ to every gate. Literals $\neg x_i$ are written as the polynomial $1 + x_i$ (recall we are working over $\mathbb{F}_2$). By a union bound, the probability that any gate makes a mistake is ϵ, as desired.

The final polynomial is obtained by composing the polynomials of each gate. The composition of a polynomial of degree d_1 with another of degree d_2 results in a polynomial of degree $d_1 \cdot d_2$. Since each polynomial has degree $\leq \log(s/\epsilon) + 1 \leq \log(cs/\epsilon)$, the final degree is $\log^d(cs/\epsilon)$.

To improve this to the stated bound, reason as follows. Apply Lemma 9.9 to every gate *except the output gate* with error c/s. Then apply Lemma 9.9 to the output gate with error c. The composition is then a polynomial of degree $(c \log s)^{d-1}$ which has error $\leq 1/3$. To shrink the error to ϵ, take $c \log 1/\epsilon$ independent samples of the polynomial and compute their majority exactly by a polynomial of degree $c \log 1/\epsilon$. **QED**

Exercise 9.10. Explain why we can handle parity gates as well (as claimed in Theorem 9.8).

9.1.2 Using the correlation to show that Majority is hard

The other step in the proof of the lower bound for majority (Theorem 9.5) is to show that Majority cannot be approximated by such low-degree polynomials. For this the key result is the following:

Lemma 9.11. Every function $f : [2]^n \to [2]$ can be written as $f(x) = p_0(x) + p_1(x) \cdot \mathrm{Maj}(x)$, for some polynomials p_0 and p_1 of degree $\leq n/2$. This holds for every odd n.

Proof. Let M_0 be the set of strings with weight $\leq n/2$, and M_1 the set of strings with weight $\geq n/2$. Assume that for any $b \in [2]$ and every function $f : M_b \to [2]$ there is a polynomial p_b of degree $\leq n/2$ s.t. p_b and f agree on M_b. From this the lemma follows, because we can write

$$f = p_1 \mathrm{Maj} + p_0(1 + \mathrm{Maj}) = p_0 + (p_0 + p_1)\mathrm{Maj}.$$

To prove the assumption, we claim that (the vectors corresponding to) their truth tables over M_0 are linearly independent. This means that any polynomial gives a different function over M_0, and because the number of polynomials is the same as the number of functions, the assumption follows. **QED**

Exercise 9.12. Prove the claim in the proof. Hint: More generally, prove that a non-zero polynomial of degree $\leq k$ is non-zero on a string of weight $\leq k$.

Proof of Theorem 9.5. Apply Theorem 9.8 with $\epsilon = 1/10$ to obtain p. Let S be the set of inputs on which $p(x) = C(x)$. By Lemma 9.11, any function $f : S \to [2]$ ca be written as

$$f(x) = p_0(x) + p_1(x) \cdot p(x).$$

The right-hand side is a polynomial of degree $\leq d' := n/2 + \log^{d-1}(cs)$. The number of such polynomials is the number of possible choices for each monomial of degree i, for any i up to the degree. This number is

$$\prod_{i=0}^{d'} 2^{\binom{n}{i}} = 2^{\sum_i^{d'} \binom{n}{i}}.$$

On the other hand, the number of possible functions $f : S \to [2]$ is

$$2^{|S|}.$$

Since a polynomial computes at most one function, taking logs we have

$$|S| \leq \sum_i^{d'} \binom{n}{i}.$$

The right-hand side is at most $2^n(1/2 + c\log^{d-1}(s)/\sqrt{n})$, since each binomial coefficient is $\leq c2^n/\sqrt{n}$, see Fact A.6.

On the other hand, $|S| \geq 0.9 \cdot 2^n$.

Combining this we get

$$0.9 \cdot 2^n \leq 2^n(1/2 + c\log^{d-1}(s)/\sqrt{n}).$$

This implies

$$0.4 \leq c\log^{d-1}(s)/\sqrt{n},$$

proving the theorem. **QED**

AC^0-$\oplus$ is a particularly interesting class because we can prove exponential impossibility results, but not exponential correlation bounds. The proof technique above gives a correlation bound no better than $1/\sqrt{n}$, for any function. Proving that some function in P has correlation $< 1/\sqrt{n}$ with AC^0-$\oplus$ is an open problem. This open problem is notable because it stands in the way of a number of other long-standing problems, see the notes. One candidate is the Xor of two majorities on disjoint inputs (cf Lemma 11.46).

Open question 9.13. Does AC^0-$\oplus$ have correlation $< 1/\sqrt{n}$ with the Xor of two majorities on $n/2$ bits each?

9.2 The polynomial method over $\mathbb{R}$

In this section we develop the polynomial method over $\mathbb{R}$. We show that we small AC^0 circuits correlate with low-degree polynomials over the reals. Using this, we prove an impossibility result for AC^0 circuits with a Majority gate at the output. Finally, we show that this result implies an exponential correlation bound between parity and AC^0 circuits. Thus the polynomial method is sufficient to prove such strong correlation bounds, which also follow from switching lemmas 9.3.

9.2.1 AC^0 correlates with low-degree real polynomials

The following is a real analogue of Theorem 9.8.

Theorem 9.14. Let $C : [2]^n \to [2]$ be an AC^0 circuit of size $s \geq n$ and depth d. Then for every $\epsilon > 0$ there is a distribution P on polynomials over $\mathbb{R}$ of degree $\log^{cd}(s/\epsilon)$ such that for every $x \in [2]^n$

$$\mathbb{P}_P[P(x) \neq C(x)] \leq \epsilon.$$

As in Theorem 9.8, this implies the existence of a polynomial with good correlation. Note that $P(x)$ may not lie in [2]. Any such output counts as a mistake. Thus the approximating appears quite strong: A real polynomial will output a boolean value in most cases.

Proof. The proof is similar in spirit to the proof of Theorem 9.8. The only difference is how we handle an Or gate. We "guess" the weight of the input, similarly to the proof of Theorem 5.28. Details follow. Consider an Or gate on m bits, assumed to be a power of 2 w.l.o.g.. For any $i = 0, \ldots, \log m$, consider the random set $S_i \subseteq \{1, 2, \ldots, m\}$ obtained by including each element of $\{1, 2, \ldots, m\}$ independently with probability 2^{-i}; in particular, $S_0 := \{1, 2, \ldots, m\}$.

Consider the associated random degree-1 polynomials

$$P_i(x) := \sum_{i \in S_i} x_i$$

for $i = 0, \ldots, \log m$.

Every polynomial always computes Or correctly when the input is 0^m. Now we claim that on any input that is not 0^m, with high probability there is some polynomial that computes Or correctly. Specifically, for all $x \in [2]^m$, $x \neq 0^m$, with probability $\geq c$ over $P_0, \ldots, P_{\log m}$ there is i such that $P_i(x) = 1$.

To show this, let $w := \sum x_i$, $1 \leq w \leq m$, be the weight of x. Let $i, 0 \leq i \leq \log m$, be such that

$$2^{i-1} < w \leq 2^i.$$

Using this bound on w and finally Fact A.5 we obtain

$$\begin{aligned}\mathbb{P}[P_i(x) = 1] &= \mathbb{P}\left[\sum_{i \in S_i} x_i = 1\right] \\ &= w \cdot 2^{-i}(1-2^{-i})^{w-1} \\ &\geq \frac{1}{2} \cdot (1-2^{-i})^{2^i-1} \\ &\geq \frac{1}{2e}.\end{aligned}$$

We now combine the above polynomials into a single one P' as follows:

$$P'(x) := 1 - \prod_{i=0}^{\log m} (1 - P_i(x).)$$

If $x = 0^m$, then again $P'(x) = 0$. If $x \neq 0^m$ then the probability that P' equals 1 is at least the probability that some P_i equals 1 which is $\geq c$ by above. Also, the degree of P' is $c \log m$.

We can reduce the error in P' as in the proof of Theorem 9.8. Explicitly, let

$$P_\epsilon(x) := 1 - \prod_{i=0}^{4 \log 1/\epsilon} (1 - P'_i(x))$$

where $P'_i(x)$ are independent copies of P' above. If $x = 0$ then $P_\epsilon(x) = 0$. If $x \neq 0$,

$$\begin{aligned}\mathbb{P}[P_\epsilon(x) = 1] &\geq \mathbb{P}[\exists i : P'_i(x) = 1] = 1 - \mathbb{P}[\forall i, P'_i(x) \neq 1] \\ &= 1 - \mathbb{P}[P'(x) \neq 1]^{4 \log 1/\epsilon} \geq 1 - (5/6)^{4 \log 1/\epsilon} \geq 1 - \epsilon.\end{aligned}$$

Also, the degree of P_ϵ is $O(\log m \cdot \log 1/\epsilon)$. **QED**

9.2.2 Sign-Approximating Maj-AC^0

An interesting consequence of Theorem 9.14 is that it also gives an approximation of AC^0 circuits augmented with a majority gate.

Definition 9.15. A Maj-AC^0 circuit is an AC^0 circuit except its (only) output gate is Majority.

Such Maj-AC^0 circuits are more robust than what might be apparent at first sight: An AC^0 circuit of size s and depth d with m majority gates (appearing anywhere in the circuit) can be simulated by a Maj-AC^0 circuit of depth $d + c$ and size $2^{m \log^{cd} s}$ (see the notes). For fixed d and $m = \log^c s$, this is a sub-exponential blow-up in size which is negligible compared to the exponential lower bounds we are proving.

The approximation of Maj-AC^0 is by the *sign* of a polynomial. Here, sign : $\mathbb{R} \to \{-1, 1\}$ outputs the sign of a real; the value at 0 is irrelevant for our analysis and can be defined arbitrarily.

Corollary 9.16. Let C be a Maj-AC^0 circuit on n bits of depth d and size s. Then, for every $\epsilon > 0$ there exists a polynomial $p : \{0,1\}^n \to \mathbb{R}$ of degree $\log^{cd}(s/\epsilon)$ such that

$$\mathbb{P}_{x \in \{0,1\}^n} [\text{sign}(p(x)) = e(C(x))] \geq 1 - \epsilon.$$

Proof. Denote the AC^0 subcircuits that feed into the output majority gate by $C_1, \ldots, C_t$, where $t \leq s$. Apply Theorem 9.14 with $\epsilon' := \epsilon/t$ to get polynomials $p_1, \ldots, p_t$ such that for every $i \leq t$ we have $\mathbb{P}_x [p_i(x) = C_i(x)] \geq 1 - \epsilon'$. The degree of each p_i is $\leq \log^{cd}(s/\epsilon')$, since each circuit C_i has size $\leq s$ and depth $\leq d$. Then, define

$$p(x) := \sum_{i \leq t} p_i(x) - \frac{t}{2}.$$

Note that the degree of p is equal to the maximum degree of any p_i, which is $\log^{cd}(w/\epsilon') = \log^{cd}(w/\epsilon)$ (because $t \leq s$). Also note that, whenever we have $p_i(x) = C_i(x)$ for all i, then $\text{sign}(p(x)) = e\left(\text{Majority}(C_1(x), \ldots, C_t(x))\right) = e(C(x))$. By a union bound, this happens with probability at least $1 - t \cdot \epsilon' = 1 - \epsilon$. **QED**

9.2.3 Using the correlation to show impossibility for Maj-AC^0

In this section we use the approximation to show impossibility for Maj-AC^0, and specifically prove:

Theorem 9.17. A Maj-AC^0 circuit of depth d and size s computing parity on n bits has $s \geq 2^{n^{c/d}}$.

The proof has two steps. First we show how to turn any polynomial p that sign-approximates a function f on all but an ϵ fraction of inputs into another polynomial p' that *weakly computes* f, i.e. p' is not the zero polynomial and for every x such that $p'(x) \neq 0$ we have $\text{sign}(p'(x)) = e(f(x))$.

Lemma 9.18. Let $p : [2]^n \to \mathbb{R}$ be a polynomial of degree d and $f : [2]^n \to [2]$ a function such that $\mathbb{P}_x[\text{sign}(p(x)) \neq e(f(x))] \leq \epsilon < 1/2$. Then, there exists a non-zero polynomial $p' : [2]^n \to \mathbb{R}$ such that for every x where $p'(x) \neq 0$ we have $\text{sign}(p'(x)) = e(f(x))$, and the degree of p' is $\leq n - \epsilon_c\sqrt{n} + d$.

The idea is to construct a non-zero polynomial q that is 0 on the ϵ fraction of points on which $\text{sign}(p(x)) \neq e(f(x))$, and then define

$$p'(x) := p(x) \cdot q^2(x).$$

Since q is 0 on any x on which $\text{sign}(p(x)) \neq e(f(x))$, and we square it, p' weakly computes f: on any input where $p'(x) \neq 0$ we have $\text{sign}(p'(x)) = \text{sign}(p(x)) = e(f(x))$.

To zero-out an ϵ fraction of the points, the degree of q will need to be $n/2 - \epsilon_c\sqrt{n}$. Hence the degree of q^2 will be $n - \epsilon_c\sqrt{n}$ and that of p' will be $n - \epsilon_c\sqrt{n}$ plus the degree of p. In

a typical setting of parameters, the degree of p is power-logarithmic in n which makes the degree of p' strictly less than n, which is sufficient for our purposes.

Proof. Let $S \subseteq [2]^n$ be the set of inputs x such that $\text{sign}(p(x)) \neq e(f(x))$. By assumption, $|S| \leq \epsilon 2^n$. We define a non-zero polynomial q such that $q(x) = 0$ for every $x \in S$. Let M be the set of products of at most $n/2 - \epsilon'\sqrt{n}$ variables, for a parameter ϵ' to be chosen later. In other words, M is the set of monic, multilinear monomials of degree $\leq n/2 - \epsilon'\sqrt{n}$:

$$M := \left\{ \prod_{i \in I} x_i : I \subseteq [n], |I| \leq n/2 - \epsilon'\sqrt{n} \right\}.$$

Then, let $q(x) := \sum_{m \in M} a_m \cdot m(x)$ be the weighted sum of these monomials, for a set of weights, or coefficients, $\{a_m\}_{m \in M}$. We want to choose the coefficients in order to make q a non-zero polynomial that vanishes on S. This is equivalent to finding a non-trivial solution to a certain system of equations. Denote $S = \{s_1, \ldots, s_{|S|}\}$ and $M = \{m_1, \ldots, m_{|M|}\}$. Then, the system of equations we would like to solve is

$$\begin{pmatrix} m_1(s_1) & m_2(s_1) & \cdots & m_{|M|}(s_1) \\ m_1(s_2) & m_2(s_2) & \cdots & m_{|M|}(s_2) \\ \vdots & \vdots & \ddots & \vdots \\ m_1(s_{|S|}) & m_2(s_{|S|}) & \cdots & m_{|M|}(s_{|S|}) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_{|M|} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

From linear algebra (Fact A.35), we know that this system has a non-zero solution if there are more variables (the $|M|$ coefficients a_i) than equations (the $|S|$ rows in the matrix). Therefore, we need to show $|M| > |S|$. Using bounds similar to those used in the proof of Theorem 9.5, we have:

$$\begin{aligned} |M| &= \sum_{i=0}^{\lfloor n/2 - \epsilon'\sqrt{n} \rfloor} \binom{n}{i} = \sum_{i=0}^{\lceil n/2 \rceil} \binom{n}{i} - \sum_{i=\lfloor n/2 - \epsilon'\sqrt{n} \rfloor + 1}^{\lceil n/2 \rceil} \binom{n}{i} \geq 2^{n-1} - c\epsilon'\sqrt{n}\binom{n}{\lfloor n/2 \rfloor} \\ &\geq 2^{n-1} - c\epsilon' \cdot 2^n. \end{aligned}$$

By choosing ϵ' sufficiently small compared to $\epsilon < 1/2$, we have

$$|M| > 2^n (1/2 - c\epsilon') > 2^n \cdot \epsilon = |S|.$$

This shows the existence of the desired polynomial q.

Finally, let

$$p'(x) := p(x) \cdot q(x)^2.$$

Since q vanishes on S and q^2 is always positive, $\text{sign}(p'(x)) = e(f(x))$ whenever $p'(x) \neq 0$. Furthermore, p' is not the zero polynomial, since neither p nor q are, and the degree of p' is the degree of p plus twice the degree of q: $d + 2(n/2 - \epsilon'\sqrt{n}) = d + n - 2\epsilon'\sqrt{n}$. **QED**

Now in the second step we show that degree at least n is required to weakly compute the parity function.

Lemma 9.19. Let $p : [2]^n \to \mathbb{R}$ be a non-zero polynomial such that whenever $p(x) \neq 0$ we have $\text{sign}(p(x)) = e(\sum_i x_i)$. Then p has degree at least n.

Proof. Suppose for the sake of contradiction that p has degree $n-1$. We can assume that p is multilinear, since $x^2 = x$ for $x \in [2]$.

Then, for some coefficients $\{a_I\}_{I \subseteq [n]}$, we can write

$$p(x) = \sum_{I \subseteq [n], |I| \leq n-1} a_I \prod_{i \in I} x_i.$$

Let us consider

$$\mathbb{E}_{x \in [2]^n} \left[p(x) \cdot e\left(\sum_i x_i \right) \right].$$

On the one hand, this expectation is > 0. Indeed, the polynomial computes a function that is not identically zero, for else the polynomial, being multilinear, would be zero (Exercise 9.12). But the sign on any non-zero output is $e(\sum_i x_i)$.

On the other hand, by linearity of expectation,

$$\begin{aligned}
\mathbb{E}_{x \in [2]^n} \left(p(x) \cdot e\left(\sum x_i \right) \right) &= \sum_{I \subseteq [n], |I| \leq n-1} a_I \cdot \mathbb{E}_{x \in [2]^n} \left(\prod_{i \in I} x_i \cdot e\left(\sum_i x_i \right) \right) \\
&= \sum_{I \subseteq [n], |I| \leq n-1} a_I \cdot \mathbb{E}_{x \in [2]^n} \left(\prod_{i \in I} x_i \cdot e\left(\sum_{i \in I} x_i \right) \right) \cdot \mathbb{E}_{x \in [2]^n} \left(e\left(\sum_{i \notin I} x_i \right) \right) \\
&= 0,
\end{aligned}$$

where the last equality holds because $|I| < n$, and so the sum $\sum_{i \notin I}$ contains at least one variable, which over the choice of a uniform x will be equally likely to be 0 or 1, making $e\left(\sum_{i \notin I} x_i\right)$ equally likely to be $+1$ and -1, for an expected value of 0. **QED**

Putting the steps together proves Theorem 9.17.

Proof of Theorem 9.17.. Let C be a Maj-AC^0 circuit of size s and depth d. By Corollary 9.16 with $\epsilon := 1/3$ there is a polynomial $p : [2]^n \to \mathbb{R}$ of degree $\log^{cd} s$ such that

$$\mathbb{P}_{x \in [2]^n}[\text{sign}(p(x)) = e(C(x)) = e(\text{parity}(x))] \geq 2/3.$$

Apply Lemma 9.18 to obtain a non-zero polynomial p' of degree $\leq n - \Omega(\sqrt{n}) + \log^{cd} s$ such that, for any x for which $p'(x) \neq 0$, we have $\text{sign}(p'(x)) = e(\text{parity}(x))$.

By Lemma 9.19, the degree of p' must be at least n. Therefore:

$$\begin{aligned}
& n - c\sqrt{n} + \log^{cd} s \geq n \\
\Leftrightarrow\ & \log^{cd} s \geq \sqrt{n},
\end{aligned}$$

concluding the proof. **QED**

9.2.4 AC^0 has small correlation with parity

Using the above results we can prove that AC^0 circuits have exponentially small correlation with parity.

Corollary 9.20. The correlation between parity on n bits and an alternating circuit of depth d and size $s \leq 2^{n^{c_d}}$ is $\leq 1/s$.

By contrast, even a single bit has much larger correlation with Majority, as follows for example by Theorem 8.13.

The proof goes by showing that if an AC^0 circuit C correlates with parity then a Maj-AC^0 circuit computes parity, contradicting Theorem 9.17. On a given input x we construct a distribution on circuits $C_a(x)$ whose output distribution is biased towards parity(x). Taking the majority of a number of independent copies of $C_a(x)$ gives a Maj-AC^0 circuit that computes parity(x) correctly. To construct $C_a(x)$ from C we take advantage of the efficient *random self-reducibility of parity*, the ability to efficiently reduce the computation of parity(x) for any given input x to the computation of parity(a) for a randomly selected input a. The fact that this can be done in AC^0 (in fact, NC^0) is a beautiful, essential component of the proof.

Proof. Let C be a circuit of depth d and size d such that $\left|\mathbb{E}_{x\in[2]^n} e\left[C(x) + \text{parity}(x)\right]\right| \geq 1/s$. Without loss of generality we drop the absolute value signs (since we can complement C at no cost). We now make use of the aforementioned *random self-reducibility* of parity. For $a \in \{0,1\}^n$, consider the circuit C_a defined by

$$C_a(x) := C(a + x) + \text{parity}(a).$$

First of all, notice that for any fixed a, C_a has depth $d + c$ and size $s + cn \leq cs$ (using $s \geq n$ w.l.o.g.). This is because XOR'ing the output with the bit parity(a) can be done with a constant number of gates, while XOR'ing the input by a takes constant depth and cn gates. Now observe that for any fixed $x \in [2]^n$,

$$\begin{aligned}\mathbb{E}_a e\left[C_a(x) + \text{parity}(x)\right] &= \mathbb{E}_a e\left[C(a + x) + \text{parity}(a) + \text{parity}(x)\right] \\ &= \mathbb{E}_a e\left[C(a + x) + \text{parity}(a + x)\right] \\ &= \mathbb{E}_a e\left[C(a) + \text{parity}(a)\right] \geq 1/s.\end{aligned}$$

It is convenient to rewrite the above conclusion as: for any fixed $x \in [2]^n$,

$$\mathbb{P}_a\left[C_a(x) = \text{parity}(x)\right] \geq \frac{1}{2} + \frac{1}{2s}.$$

Now pick t independent $a_1, a_2, \ldots, a_t$ for a parameter t to be determined below, and define the Maj-AC^0 circuit

$$\bar{C}_{a_1,\ldots,a_t}(x) := \text{Maj}\left(C_{a_1}(x), \ldots, C_{a_t}(x)\right).$$

Note that $\bar{C}_{a_1,\ldots,a_t}$ has depth $d+c$ and size $\leq cts$, for every choice of $a_1, \ldots, a_t$. By an analysis similar to the one in the proof of Theorem 3.3, or tail bounds Lemma A.15, we get that for $t := 2w^2n$, for any $x \in [2]^n$, $\mathbb{P}_{a_1,\ldots,a_t}\left[\bar{C}_{a_1,\ldots,a_t}(x) \neq \text{parity}(x)\right] < 2^{-n}$. By a union bound we can fix choices of $a_1, \ldots, a_t$. This results in a Maj-AC^0 circuit of depth $d + c$ and size $\leq cts$ that computes parity on every n-bit input. **QED**

9.3 Switching lemmas

In this section we sharpen the restrict-and-simplify method for alternating circuits, slightly improving the parameters of the bounds established via the polynomial method, Theorem 9.5. For depth d the polynomial method gave a size bound of $s \geq 2^{cn^{0.5/(d-1)}}$. For alternating circuits we can remove the 0.5. In particular, for depth 3 we obtain $s \geq 2^{c\sqrt{n}}$.

Definition 9.21. A *restriction* ρ is an assignment of the variables to $\{0, 1, \star\}$, i.e., some variables are replaced with constants, while those assigned to *star* $\star$ are left "alive." In a *random restriction*, the stars are selected at random according to some distribution, and also each unrestricted variable is set to a uniform bit. For a restriction ρ with s stars and $f : [2]^n \to [2]$ we denote by $f_\rho : [2]^s \to [2]$ the restricted function.

Two natural ways to select the locations for the stars are to select them uniformly among subsets of a fixed size, or to assign each bit to $\star$ independently with the same probability.

A *switching lemma* shows that applying a random restriction to a CNF (or DNF) greatly simplifies it. The simplification is so extreme that one can write a restricted CNF as a DNF and viceversa, whence the name "switching." This switch allows one to write a restricted circuit of depth d as a circuit of depth $d - 1$.

In fact, the restricted CNF (or DNF) can be written as a small-depth decision tree.

Definition 9.22. A function $f : [2]^n \to [2]$ is *w-local* if it depends on $\leq w$ input bits. A *decision tree* is a branching program where each node has in-degree 1. The tree has depth d if every path has $\leq d$ edges.

For example, a decision tree of depth 1 queries one bit.

Exercise 9.23. Let f be computable by a decision tree of depth d. Prove that f is also:

- $2^d - 1$ local,
- A d-DNF,
- A d-CNF.

9.3.1 Switching I

We begin with a switching lemma which is useful in building intuition because its proof makes it evident why switching is at all possible. In terms of parameters, the lemma is sufficient to prove superpower lower bounds, for example establishing that Parity is not in AC^0, but not exponential.

Lemma 9.24. [Switching, I] Let f be an a-DNF on n bits and ρ a random restriction setting each bit to $\star$ independently w.p. $1/n^{\epsilon}$. Then f_ρ is a decision tree of depth $c_{a,\epsilon}$ except with prob. $\leq 1/n^{1/\epsilon}$.

Note the larger the a and the smaller the ϵ the stronger lemma (except for the depth of the tree). To apply a switching lemma to a circuit we view gates at depth 1 as 1-DNFs or 1-CNFs; we illustrate this below.

Proof. We prove it by induction on a for all ϵ.

For base case we can take $a = 0$ which is defined as a constant function.

For the inductive step, suppose f contains k disjoint terms. The prob. p that f_ρ is not constant is $\leq$ the prob. that all the terms in f_ρ are not 1 (as soon as a term is 1 the function is 1). A term is 1 if all the literals are 1. A literal is 1 if it is not mapped to $\star$ and then it is fixed to 1, this has probability $= 0.5(1 - 1/n^{\epsilon}) \geq 1/3$. Hence a term is 1 w.p. $\geq (1/3)^a$ and

$$p \leq (1 - 1/3^a)^k \leq e^{-k/3^a}.$$

For $k = 3^a \log n^{1/\epsilon}$ we have $p \leq 1/n^{1/\epsilon}$ as desired.

Otherwise, we claim that there are ak variables I that intersect *every* term of f. This is a basic and extremely useful combinatorial fact. To prove it, simply collect disjoint terms greedily. We collect $\leq k - 1$, since f does not contain k disjoint terms. No other term is disjoint, which means it contains ≥ 1 of the variables in the terms just collected. Since each term has $\leq a$ variables, we have proved the claim.

Now view ρ as first applying ρ_1 which assigns $\star$ with prob. $1/n^{\epsilon/2}$, and then again applying ρ_2 with the same $\star$ prob. to the remaining variables. Over the choice of ρ_1, the prob. of having $\geq 4/\epsilon^2$ stars in I is by a union bound and Fact A.7

$$\leq \binom{ak}{4/\epsilon^2} \left(\frac{1}{n^{\epsilon/2}}\right)^{4/\epsilon^2} \leq \left(\frac{ca3^a\epsilon^{-1}\log n}{4/\epsilon^2}\right)^{4/\epsilon^2} \frac{1}{n^{2/\epsilon}} \leq \frac{0.5}{n^{1/\epsilon}}$$

for $n \geq c_{a,\epsilon}$.

When there are $\leq 4/\epsilon^2$ stars in I, a decision tree for f_ρ can begin by reading all these variables. After reading them, we get an $(a-1)$-DNF, because the variables intersected every term. The number of possible DNFs thus obtained is $\leq 2^{4/\epsilon^2}$. By a union bound, and the induction hypothesis, the prob. over ρ_2 that one of these $(a-1)$-DNFs is not a decision tree of depth $\leq c_{a,\epsilon}$ is

$$\leq 2^{4/\epsilon^2} \cdot \frac{1}{n^{2/\epsilon}} \leq \frac{0.5}{n^{1/\epsilon}}.$$

Overall, the error probability is $\leq 2 \cdot 0.5/n^{1/\epsilon} \leq 1/n^{1/\epsilon}$ as desired. **QED**

9.3.2 Switching II

Next we state a stronger switching lemma. The proof does not really benefit from the fact that the terms are simple, so we state it for arbitrary functions.

Lemma 9.25. [Switching, II] Let $C : [2]^n \to [2]$ equal the Or of functions $f_i : [2]^n \to [2]$ where each f_i is w-local. Let ρ be a random restriction with s stars. The probability that C_ρ is not a decision tree of depth d is $\leq (cws/n)^d$.

The proof of Lemma 9.25 is below in section 9.3.3.

Applying switching lemmas several times allows us to collapse an alternating circuit to a low-depth decision tree. The collapse applies even to circuits of exponential size. We state and prove this consequence trading simplicity of exposition for parameter optimization.

Corollary 9.26. Let $C : [2]^n \to [2]$ be an alternating circuit of depth d size $s \leq 2^{n^{c_d}}$. Let ρ be a random restriction with $2 \log s$ stars. The probability that C_ρ is not a decision tree of depth $\log s$ is $\leq 1/s$.

In particular, there is a restriction ρ with $\geq \log s$ stars s.t. C_ρ is constant.

Proof. Set $w := \log s$. We view ρ as successive applications of restrictions whose number of stars is square root of the number of variables. View the circuit as having depth $d + 1$ and the input gates have fan-in $1 \leq w$. The first application of Lemma 9.25 gives error $\leq (cw/\sqrt{n})^w$. In the good case, the input gates now are decision trees of depth $\leq w$. We can write this as a CNF or DNF with terms of size $\leq w$, and merge the output gate with the gates in the next level in the circuit, which are now computing Ors (or Ands) of functions on $\leq w$ bits. The next application of Lemma 9.25 gives error $\leq (cw/n^{1/4})^w$, and so on. In general, application i of the lemma gives an error of

$$(cw/n^{1/2^i})^w \leq c^{-w} = 1/s^2.$$

Taking a union bound over all gates in the circuit, the error probability is as desired.

The number of stars in the final restriction is $\geq 2 \log s$ by our assumption on s. **QED**

Exercise 9.27. Prove the "in particular."

One can use the switching lemma to get an alternative proof that parity has exponentially small correlation with AC^0 (Corollary 9.20).

Proof of Corollary 9.20.. The correlation between parity on m bits and decision trees of depth $< m$ is zero. View a uniform input as first picking a restriction, and then filling the stars. By Corollary 9.26, after picking the restriction the circuit is a decision tree of depth $\log s$, which is strictly less than the number of remaining stars, except with probability $1/s$. **QED**

As for impossibility results, switching lemmas optimize parameters somewhat. One can prove that depth-d alternating circuits for parity have size $\geq 2^{n^{c/(d-1)}}$ which is tight up to the

constants. This bound holds even if the circuits have any correlation $\geq 1/2^{n^{c/(d-1)}}$. Using yet another switching lemma, one can prove a better tradeoff between size and correlation, see the notes.

We illustrate the parameter optimization in the simple and significant case of lower bounds for depth 3. We leave the extension to arbitrary depth and correlation bounds as an exercise.

Corollary 9.28. Parity requires depth-3 alternating circuits of size $\geq 2^{c\sqrt{n}}$.

Proof. Pick a random restriction ρ which is the composition of two restrictions, the first picking cn stars, and the second picking $c\sqrt{n}$. For concreteness and w.l.o.g. let C be an And of DNFs. View the depth-1 gates as 1-CNFs. By Lemma 9.25, the first restriction collapses a depth-1 gate to a $c\sqrt{n}$-depth decision tree except with prob. $\leq 2^{-c\sqrt{n}}$. This is small enough that we can do a union bound over all gates at depth 1. When all the gates simplify to $c\sqrt{n}$-depth decision trees, we can rewrite the restricted circuit as a And of $c\sqrt{n}$-DNFs. Now, over the second restriction, the probability that one such DNF does not become a $c\sqrt{n}$-depth decision tree is again $2^{-c\sqrt{n}}$, and we can again do a union bound over all the DNFs, of which there are still $\leq 2^{c\sqrt{n}}$. We can write each such decision tree as a $c\sqrt{n}$-CNF. By merging the And gates we obtain that C_ρ is a t-CNF, where $t \leq c\sqrt{n}$. At the same time, by the choice of parameters we have $> t$ stars. To conclude, we argue that such a CNF cannot compute parity. To prove this, note that it suffices to fix a clause of the CNF to 0 to fix the entire CNF to 0. Since this is a t-CNF, we only need to fix $\leq t$ variables. However, there is still at least one star, so parity is not fixed. **QED**

9.3.3 Proof of switching II

In this section we prove Lemma 9.25. We illustrate the proof in stages.

The simplest case: Or of n bits. Here f is simply the Or of n bits $x_1, x_2, \ldots, x_n$. In the restriction some of the bits may become 0, others 1, and others yet may remain unfixed, i.e., assigned to stars. Those that become 0 you can ignore, while if some become 1 then the whole circuit C becomes 1.

We will show that the number of *bad* restrictions, those for which the restricted circuit $C|_\rho$ requires decision trees of depth $\geq d$, is small.

For this simple case, a straightforward proof of a stronger bound exists.

Exercise 9.29. Give it.

We give an alternative argument which we can then extend to the general case. We are going to encode such restrictions *using another restriction*. In this simple case, the restriction will have no stars; that is, it is just a 0/1 assignment to the variables). The gain is clear: just think of a restriction with zero stars versus a restriction with one star. Let us quantify this gain.

Definition 9.30. Denote by N_s the number of restrictions with exactly s stars.

Exercise 9.31. $N_s = \binom{n}{s}2^{n-s}$.

Going back to the gain, we have $N_0 = 2^n$, while $N_1 = 2^{n-1} \cdot n$, so $N_0/N_1 \leq c/n$. Note that this an upper bound on the error probability.

We repeat that *we only want to encode the bad restrictions for which $C|_\rho$ requires large depth*. So ρ does not map any variable to 1, for else the Or is 1 which has decision trees of depth 0. The way we are going to encode ρ is this: *Simply replace the stars with ones*. To go back, replace the ones with stars. We are using the ones in the encoding to "signal" where the stars are. Hence, the number of bad restrictions is at most $N_0 = 2^n$, which is tiny compared to the number $N_s = \binom{n}{s}2^{n-s}$ of restrictions with s stars (see Exercise 9.31). The error probability is then (using Fact A.7)

$$\frac{2^n}{\binom{n}{s}2^{n-s}} = \frac{2^s}{\binom{n}{s}} \leq \left(\frac{2s}{n}\right)^s.$$

This is stronger than Lemma 9.25. (We can assume $d \leq s$, since every function on s bits has decision trees of depth s, and so for $d \geq s$ the error probability is 0.)

The medium case: Or of functions on disjoint inputs. So, again, let's take a random restriction ρ with exactly s stars. Some of the functions may become 0, others 1, and others yet may remain unfixed. Those that become 0 you can ignore, while if some become 1 then the whole circuit becomes 1.

As before, we will show that the number of restrictions for which the restricted circuit $C|_\rho$ requires decision trees of depth $\geq d$ is small. To accomplish this, we are going to encode/map such restrictions using/to a restriction with just $s - d$ stars, plus a little more information. As we saw already, the gain in reducing the number of stars is clear. In particular, saving d stars reduces the number of restrictions by a factor $(cs/n)^d$:

$$\begin{aligned}\frac{N_{s-d}}{N_s} &= \frac{\binom{n}{s-d}2^{n-(s-d)}}{\binom{n}{s}2^{n-s}} = \frac{\frac{n!}{(s-d)!(n-s+d)!}}{\frac{n!}{s!(n-s)!}} \cdot 2^d = \frac{s(s-1)\cdots(s-d+1)}{(n-s+1)(n-s+2)\cdots(n-s+d)} \cdot 2^d \\ &\leq \frac{s^d}{(n-s+1)^d} \cdot 2^d \leq \left(\frac{2s}{n-s}\right)^d \leq \left(\frac{cs}{n}\right)^d.\end{aligned}$$

Here we first use Exercise 9.31. The first inequality amounts to picking the larger term in the numerator and the smallest in the denominator. The last inequality holds because we can assume $s \leq cn$, for else the probability bound in Lemma 9.25 is larger than one.

The auxiliary information will give us a factor of w^d, leading to the claimed bound. Specifically, as before, recall that we only want to encode restrictions for which $C|_\rho$ requires large depth. So no function in $C|_\rho$ is 1, for else the circuit is 1 and has decision trees of depth 0. Also, you have d stars among inputs to functions that are unfixed (i.e., not even

fixed to 0), for else again you can compute the function reading less than d bits. Because the functions are unfixed, there is a setting for those d stars (and possibly a few more stars – that would only help the argument) that make the corresponding functions 1. We are going to pick precisely that setting in our restriction ρ' with $s-d$ stars. This allows us to "signal" which functions had inputs with the stars we are saving (namely, those that are the constant 1). To completely recover ρ, we add extra information to indicate where the stars were. The saving here is that we only have to say where the stars are among w symbols, not n.

Specifically, we can encode the positions of the stars with an element $a \in [cw]^d$, indicating which of the w symbols is a star, and also whether the star is the last in this function. To recover the restriction, we look for the first function that's set to 1. We then read a to find out how may stars were there and their positions. Then we move to the second function that's fixed to 1. Again we look at a to know how many stars were there and what their positions were, and so on.

The general case. The idea is the same, except we have to be slightly more careful because when we set values for the stars in one function we may also affect other functions. The idea is to fix one function at the time. Specifically, starting with ρ, consider the first function f that's not made constant by ρ. So the inputs to f have some stars. As before, let us replace the stars with constants that make the function f equal to the constant 1, and append the extra information that allows us to recover where these stars were in ρ.

We'd like to repeat the argument. Note however we only have guarantees about $C|_\rho$, not $C|_\rho$ with some stars replaced with constants that make f equal to 1. We also can't just jump to the 2nd function that's not constant in $C|_\rho$, since the "signal" fixing for that might clash with the fixing for the first – this is where the overlap in inputs makes things slightly more involved. Instead, because $C|_\rho$ required decision tree depth at least d, we note there have to be some assignments to the m stars in the input to f so that the resulting, further restricted circuit still requires decision tree depth $\geq d-m$ (else $C|_\rho$ has decision trees of depth $< d$). We append this assignment to the auxiliary information and we continue the argument using the further restricted circuit.

9.4 $\mathbf{AC^0}$ vs L, $\mathbf{NC^1}$, and $\mathbf{TC^0}$

In this section we prove that constant-depth alternating circuits can simulate with sub-exponential size several classes of interest.

9.4.1 L

We first utilize the checkpoint technique (section §6.3) to simulate L.

Theorem 9.32. Let $f \in$ BrL and $d \in \mathbb{N}$. Then f_n has alternating circuits of depth d and size $2^{n^{c_f/d}}$.

Another way of saying this is that the function is in AC^0 on inputs of power-log length (padded to length n, cf remarks in section §1.2). This is quite useful.

Exercise 9.33. Prove that for any d, computing the majority of x given as input 1^k with $|x| = \log^d k$ is in AC^0.

Proof. It suffices to prove the result for boolean f. Consider a branching program of size $S = n^a$ for f. We apply the checkpoint technique to this branching program recursively, with parameter $b := n^{ca/d}$. Each application of the technique reduces the path length by a factor b. Hence with cd applications we can reduce the path length to 1. The corresponding gate is connected to the input bit which the branching program reads.

The resulting depth of the circuit is d. Next we bound the size. In one application, we have an $\exists$ quantifier over $b-1$ nodes, corresponding to an Or gate with fan-in S^{b-1}, and then a $\forall$ quantifier over b smaller paths, corresponding to an And gate with fan-in b. This gives a tree with $S^{b-1} \cdot b \leq S^b = n^{ab}$ leaves. Iterating, the number of leaves will be n^{abd}, and the total size of the circuit $\leq cn^{abd}$. This is $\leq 2^{n^{ca/d}}$ for large enough n. **QED**

We can refine the connection between branching programs and small-depth circuits in Theorem 9.32 to take into account width.

Theorem 9.34. Let $f : [2]^n \to [2]$ be computable by a branching program with width W and time t. Then f is computable by an alternating circuit of depth-3 of size $\leq 2^{c\sqrt{t \log W}}$.

As we have seen in Corollary 9.28, parity requires depth-3 circuits of size $2^{c\sqrt{n}}$. Theorem 9.34 shows that improving this would also improve the lower bounds for small-width branching programs (cf section §6.9).

Exercise 9.35. Prove Theorem 9.34.

A more general version of Theorem 9.34. states that for any parameter b one can have a depth-3 circuit with size

$$2^{b \log W + t/b + \log t},$$

output fan-in W^b, and input fan-in t/b. Interestingly, again this trade-off essentially matches known impossibility results for depth-3 circuits! This can be shown with the same reasoning as in the proof of Corollary 9.28.

9.4.2 Linear-size log-depth

There is a non-trivial simulation of linear-size log-depth circuits by alternating circuits of depth 3.

Theorem 9.36. Any circuit $C : [2]^n \to [2]$ of size an and depth $a \log n$ has an equivalent alternating circuit of depth 3 and size $2^{c_a n/\log\log n}$.

The idea of the simulation is to identify a set of $o(n)$ wires to remove from C so that the resulting circuit becomes very disconnected: each connected component has depth $\leq 0.1 \log n$. Since the circuit has fan-in 2, the output of each component can depend on at most $n^{0.1}$ input bits, and so, given the assignment to the removed edges, the output can be computed in brute-force by a depth-2 circuit of sub-exponential size. Trying all $2^{o(n)}$ assignments to the removed edges and collapsing some gates completes the simulation. We now proceed with a formal proof.

The graph corresponding to C in Theorem 9.36 is connected, but we will also work with disconnected graphs.

For the depth reduction in the proof, it is convenient to think of depth as a function from nodes to integers. The next definition and simple claim formalize this.

Definition 9.37. Let $G = (V, E)$ be a directed acyclic graph. The *depth* of a node in G is the number of nodes in a longest directed path terminating at that node. The depth of G is the depth of a deepest node in G. A *depth function D for G* is a map $D : V \to \{1, 2, \ldots, 2^k\}$ such that if $(a, b) \in E$ then $D(a) < D(b)$.

Exercise 9.38. Prove that a directed acyclic graph $G = (V, E)$ has depth at most 2^k if and only if there is a depth function $D : V \to \{1, 2, \ldots, 2^k\}$ for G.

The following is the key lemma which allows us to reduce the depth of a graph by removing few edges.

Lemma 9.39. Let $G = (V, E)$ be a directed acyclic graph with w edges and depth 2^k. It is possible to remove $\leq w/k$ edges so that the depth of the resulting graph is $\leq 2^{k-1}$.

Proof. Let $D : V \to \{1, 2, \ldots, 2^k\}$ be a depth function for G. Consider the set of edges E_i for $1 \leq i \leq k$:

$$E_i := \{(a, b) \in E | \text{ the most significant bit position where } D(a) \text{ and } D(b) \text{ differ is the } i\text{-th}\}.$$

Note that $E_1, E_2, \ldots, E_k$ is a partition of E. And since $|E| = w$, there exists an index $i, 1 \leq i \leq k$, such that $|E_i| \leq w/k$. Fix this i and remove E_i. We need to show that the depth of the resulting graph is at most 2^{k-1}. To do so we exhibit a depth function $D' : V \to [2]^{k-1}$. Specifically, let D' be D without the i-th output bit. We claim that D' is a valid depth function for the graph $G' := (V, E \setminus E_i)$. For this, we need to show that if $(a, b) \in E \setminus E_i$ then $D'(a) < D'(b)$. Indeed, let $(a, b) \in E \setminus E_i$. Since $(a, b) \in E$, we have $D(a) < D(b)$. Now, consider the most significant bit position j where $D(a)$ and $D(b)$ differ. There are three cases to consider:

j is more significant than i: In this case, since the j-th bit is retained, the relationship is also maintained, i.e., $D'(a) < D'(b)$;

$j = i$: This case cannot occur because it would mean that the edge $(a, b) \in E_i$;

j is less significant than i: In this case, the i-th bit of $D(a)$ and $D(b)$ is the same and so removing it maintains the relationship, i.e., $D'(a) < D'(b)$. **QED**

Now we prove the main theorem.

Proof of Theorem 9.36. For simplicity, we assume that both a and $\log n$ are powers of two. Let $2^\ell := a \cdot \log n$.

Applying the above lemma we can reduce the depth by a factor 1/2, i.e. from 2^ℓ to $2^{\ell-1}$, by removing $\leq a \cdot n/\ell$ edges. Applying the lemma again we reduce the depth to $2^{\ell-2}$ by removing $\leq a \cdot n/(\ell-1)$ edges. If we repeatedly apply the lemma $\log(2a)$ times the depth reduces to

$$\frac{a \log n}{2^{\log(2a)}} = \frac{\log n}{2},$$

and the total number of edges removed is at most

$$a \cdot n \left(\frac{1}{\ell} + \frac{1}{\ell-1} + \ldots + \frac{1}{\ell - \log(2a) + 1} \right) \leq a \cdot n \cdot \frac{\log(2a)}{\ell - \log(2a) + 1} = a \cdot n \cdot \frac{\log(2a)}{\log \log n}.$$

For slight convenience we also think that the circuit has an output edge e_{output}, and remove it; this way we can represent the output of the circuit in terms of the value of e_{output}. We remove at most

$$r := c_a \cdot n / \log \log n$$

edges.

We define the depth of an edge $e = g \to g'$ as the depth of g, and the value of e on an input x as the value of the gate g on x.

For every input $x \in [2]^n$ there exists a unique assignment h to the removed edges that corresponds to the computation of $C(x)$. Given an arbitrary assignment h and an input x we can check if h is the correct assignment by verifying if the value of every removed edge $e = g \to g'$ is correctly computed from (1) the values of the removed edges whose depth is less than that of e, and (2) the values of the input bits g is connected to. Since the depth of the component is $\leq (\log n)/2$ and the circuit has fan-in 2, at most $\sqrt{n}$ input bits are connected to g; we denote them by $x|_e$. Thus, for a fixed assignment h to the removed edges, the check for e can be implemented by a function $f_h^e : [2]^{\sqrt{n}} \to [2]$ (when fed the $\leq \sqrt{n}$ values of the input bits connected to g, i.e. $x|_e$).

Induction on depth shows:

$$C(x) = 1 \Leftrightarrow \exists \text{ assignment } h \text{ to removed edges such that } h(e_{\text{output}}) = 1$$
$$\text{and } \forall \text{ removed edge } e \text{ we have } f_h^e(x|_e) = 1.$$

We now claim that the above expression for the computation $C(x)$ can be implemented with the desired resources. Since we removed $r = c_a \cdot n/\log \log n$ edges, the existential quantification over all assignments to these edges can be implemented with an $\vee$ (OR) gate with fan-in 2^r. Each function $f_h^e(x|_e)$ can be implemented via brute-force by a CNF, i.e. a depth-2 $\wedge\vee$ circuit, of size $\sqrt{n} \cdot 2^{\sqrt{n}}$. For any fixed assignment h, we can combine the output $\wedge$ gates of these CNF to implement the check

$$\forall \text{ removed edge } e : f_h^e(x|_e) = 1$$

by a CNF of size at most

$$r \cdot \sqrt{n} \cdot 2^{\sqrt{n}}.$$

Finally, accounting for the existential quantification over the values of the r removed edges, we get a circuit of depth 3 and size

$$2^r \cdot r \cdot \sqrt{n} \cdot 2^{\sqrt{n}} = 2^{c_a n/\log\log n}.$$

QED

9.4.3 TC^0

The same simulation in Theorem 9.32 applies to functions in NC^1 and TC^0, just because BrL contains these classes. In the other direction, we can use Theorem 9.8 to show that AC^0 can be simulated by threshold circuits of depth 3, albeit not quite of power size.

Theorem 9.40. Let $f \in \mathrm{AC}^0\text{-}\oplus$. Then f_n has threshold circuits of depth 3 and size $2^{\log^{c_f} n}$.

Exercise 9.41. Prove Theorem 9.40 but for depth 4 instead of 3.

9.5 The power of AC^0: Gap majority

We proved in Theorem 9.5 that Majority is not in AC^0. In fact it requires exponential size, even if parity gates are allowed. Yet, AC^0 can approximate majority in a certain sense. This approximation is very useful and is the basis for Theorem 5.36 that BPP $\subseteq$ PH.

Definition 9.42. Gap-$\mathrm{Maj}_{\alpha,\beta}$ is the problem of deciding if an input $x \in [2]^n$ has weight $|x| \leq \alpha n$ or $|x| \geq \beta n$.

We have the following somewhat surprising result:

Lemma 9.43. Gap-$\mathrm{Maj}_{1/3,2/3} \in \mathrm{AC}^0$.

Proof. This is a striking application of the probabilistic method. For a fixed pair of inputs (x, y) we say that a distribution C on circuits *gives* $(\leq p, \geq q)$ if $\mathbb{P}_C[C(x) = 1] \leq p$ and $\mathbb{P}_C[C(y) = 1] \geq q$; and we similarly define *gives* with reverse inequalities. Our goal is to have a distribution that gives

$$(\leq 2^{-n}, \geq 1 - 2^{-n}) \tag{9.1}$$

for every pair $(x, y) \in [2]^n \times [2]^n$ where $|x| \leq n/3$ and $|y| \geq 2n/3$. Indeed, if we have that we can apply a union bound over the $< 2^n$ inputs to obtain a fixed circuit that solves Gap-Maj.

We construct the distribution C incrementally. Fix any pair (x, y) as above. Begin with the distribution $C_\wedge$ obtained by picking $2 \log n$ bits uniformly from the input and computing their And. This gives

$$\left((1/3)^{2\log n}, (2/3)^{2\log n}\right).$$

Let $p := (1/3)^{2\log n}$ and note $(2/3)^{2\log n} = p \cdot n^2$. So we can say that $C_\wedge$ gives

$$\left(\leq p, \geq p \cdot n^2\right).$$

Now consider the distribution $C_\vee$ obtained by complementing the circuits in $C_\wedge$. This gives

$$\left(\geq 1-p, \leq 1 - p \cdot n^2\right).$$

Next consider the distribution $C_{\wedge\vee}$ obtained by taking the And of $m := p^{-1}/n$ independent samples of $C_\vee$. This gives

$$\left(\geq (1-p)^m, \leq (1 - p \cdot n^2)^m\right).$$

Approximations for the exponential function, Fact A.5, yield $(1-p)^m \geq e^{-2pm} = e^{-2/n} \geq 0.9$ and $(1 - p \cdot n^2)^m \leq e^{-n}$:

$$(\geq 0.9, \leq e^{-n}).$$

Next consider the distribution $C_{\vee\wedge}$ obtained by complementing the circuits in $C_{\wedge\vee}$. This gives

$$(\leq 0.1, \geq 1 - e^{-n}).$$

Finally, consider the distribution $C_{\wedge\vee\wedge}$ obtained by taking the And of n independent samples of $C_{\vee\wedge}$. This gives

$$\left(\leq 0.1^n, \geq \left(1 - e^{-n}\right)^n\right).$$

For the rightmost quantity we can use Fact A.8; this gives

$$\left(\leq 0.1^n, \geq 1 - ne^{-n}\right).$$

We have $ne^{-n} < 2^{-n}$. Thus this distribution in particular gives equation (9.1). The bounds on the number of gates and the fan-in holds by inspection. **QED**

By inspection, this circuit has depth 3 and the fan-in of the gates at level 1 is $c \log n$. We shall use these facts next.

9.5.1 Back to the PH

We now return to the result that BPP $\subseteq$ PH, Theorem 5.36. We provide a proof of the first part of Theorem 5.36 (which recall suffices for BPP $\subseteq$ PH). A good way to think of these simulations is as follows. Fix a BPTime(t) machine M and an input $x \in [2]^n$, and write y for its random bits. In time t the machine uses $t' \leq ct \log t$ random bit, so $|y| \leq t'$. The simulating alternating machine is trying to decide if for most choices of the random bits y we have $M(x,y) = 1$, or if for most choices we have $M(x,y) = 0$. This is an instance of Gap-Maj on the exponentially long input

$$(M(x,0), M(x,1), M(x,2), \ldots, M(x, 2^{t'} - 1)) \in [2^{t'}].$$

To prove Theorem 5.36 we "only" need the circuits for Gap-Maj in Lemma 9.43 to be sufficiently explicit, or uniform. Let us denote these circuits on T bits by GMC_T. A simple

notion of uniformity is that GMC_T is constructible in FP. This does not work here because T is exponential in the input length $|x|$ of the BPTime computation. Instead, we need a refined notion of uniformity, arguably even more natural.

Definition 9.44. A family of alternating circuits $C_n : [2]^n \to [2]$ is *gate-uniform* if given n, an index to a gate $\bigwedge_{j\in[r]} t_j$ or $\bigvee_{j\in[r]} t_j$, and j, we can compute t_j (which is either an index to another gate or a literal) in Quasi-Linear-Time.

Lemma 9.45. Suppose GMC_T is gate uniform. Then Theorem 5.36 follows.

Proof. Let M be a machine corresponding to some function in BPTime(t) that uses $t' \leq ct \log t$ random bits. Consider GMC_T for $T := 2^{t'}$. As remarked earlier, this is a depth-3 circuit, and the gates at depth 1 have fan-in $c \log T = ct'$. Let us first prove (1) in Theorem 5.36. Use two quantifiers and the fact that the circuits are gate-uniform to index an And gate at depth 1. This takes quasi-linear time in t' and hence quasi-linear in t. Then for each $i \leq ct'$ compute input literal i of that gate, which corresponds to a choice for the random bits for the machine, and evaluate the machine on that choice. Each evaluation takes time ct, for a total of time ctt'. **QED**

Exercise 9.46. Show that (2) in Theorem 5.36 also follows.

There remains to construct explicit circuits for Gap-Maj. We give a construction which has worse parameters than Lemma 9.43 but is simple and suffices for (1) in Theorem 5.36. The idea is that if the weight of x is large (where recall weight is the number of bits set to 1 in x), then we can find a few *shifts* of the ones in x that cover each of the n bits. But if the weight of x is small we can't. By "shift" by s we mean the string $x_{i\oplus s}$, obtained from x by permuting the indices by xoring them with s. (Other permutations would work just as well.)

Lemma 9.47. Let $r := \log n$. The following circuit solves $\text{GapMaj}_{1/r^2,1-1/r^2}$ on every $x \in [2]^n$:

$$\bigvee s_1, s_2, \ldots, s_r \in [2]^r : \bigwedge i \in [2]^r : \bigvee j \in \{1, 2, \ldots, r\} : x_{i\oplus s_j}.$$

Note that the subformula rooted at $\bigwedge$ means that every bit i in $[n] = [2]^r$ is covered by some shift s_j of the input x.

Proof. If weight$(x) \leq n/r^2$. Each shift s_i contributes at most n/r^2 ones. Hence all the r shifts contribute $\leq n/r$ ones, and we do not cover every bit i.

Now assume weight$(x) \geq n(1-1/r^2)$. We show the existence of shifts s_i that cover every bit by the probabilistic method. Specifically, for a fixed x we pick the shifts uniformly at

random and aim to show that the probability that we do not cover every bit is < 1. Indeed:

$$\begin{aligned}
&\mathbb{P}_{s_1,s_2,\ldots,s_r}\left[\exists i \in [2]^r : \forall j \in \{1,2,\ldots,r\} : x_{i\oplus s_j} = 0\right] \\
\leq &\sum_{i\in[2]^r} \mathbb{P}_{s_1,s_2,\ldots,s_r}\left[\forall j \in \{1,2,\ldots,r\} : x_{i\oplus s_j} = 0\right] && \text{(union bound)} \\
= &\sum_{i\in[2]^r} \mathbb{P}_s[x_{i\oplus s} = 0]^r && \text{(independence of the } s_i\text{)} \\
\leq &\sum_{i\in[2]^r} (1/r^2)^r && \text{(by assumption on weight}(x)\text{)} \\
\leq &(2/r^2)^r \\
< &1.
\end{aligned}$$

QED

Exercise 9.48. Prove (1) in Theorem 5.36.

Lemma 9.47 is not sufficient for (2) in Theorem 5.36. One can prove (2) by *derandomizing* the shifts in Lemma 9.47. This means generating their r^2 bits using a seed of only $r \log^c r$ bits (instead of the trivial r^2 in Lemma 9.47), see Problem 11.5 and Problem 12.3.

9.6 Mod 6

The polynomial method (section 9.1) is effective to prove impossibility results against AC^0-Mod-m if m is a prime power. These techniques are illustrated in section 9.1 in the fundamental case $m = 2$. They can be extended to any prime power m (see Problem 9.2), but they break down when m is composite. It is consistent with our knowledge that AC^0-Mod-6 equals CktP. Even the status of simple functions is unkonwn:

Open question 9.49. Is Majority in AC^0-Mod-6?

Nevertheless AC^0-Mod-m can be simulated by polynomials even for composite m. The simulation is incomparable to the one for $m = 2$ that we saw in section 9.1 (see Theorem 9.8). In the new simulation we work with polynomials over a larger domain which we then map to a boolean value. Equivalently, we can think of this as a depth-2 circuit whose output gate is a symmetric function. On the other hand, this new simulation works for every input as opposed to most inputs.

Lemma 9.50. Let $f \in \text{ACC}^0$. Then for every n and $d := \log^{c_f} n$

$$f_n(x_1, x_2, \ldots, x_n) = b(p(x_1, x_2, \ldots, x_n))$$

for some polynomial p of degree d over $\mathbb{Z}_{2^d}$, and some function $b : [2^d] \to [2]$.

Note this Lemma 9.50 generalizes Theorem 9.40.

The proof uses *modulus-amplifying polynomials*, which allow us to treat a Mod m gate as an integer sum and move it towards the output of the circuit.

Lemma 9.51. [Modulus-amplifying polynomials] For every integer ℓ there is a univariate polynomial F_ℓ of degree $2\ell - 1$ over $\mathbb{Z}$ such that for every $y \in \mathbb{Z}$:

$$y \mod 2 = F_\ell(y) \mod 2^\ell.$$

Exercise 9.52. Prove Lemma 9.51 with the weaker degree bound ℓ^c, which suffices for all applications in this book. Guideline:

(1) Let $a(m) := m^2(3-2m)$. Prove that for both $b \in [2]$, if $m = b \mod 2^j$ then $a(m) = b \mod 2^{2j}$, for any j.

(2) Conclude the proof.

One can now prove Lemma 9.50. We present the proof of a representative special case.

Proof of special case of Lemma 9.50.. Consider a depth-3 $\mathrm{Sym}_{n^a} \oplus \wedge_{\log^a n}$ circuit C: The output is a symmetric function of n^a polynomials p_i over $\mathbb{F}_2$ of degree $\log^a n$. (The symmetric function could be Mod-3, but the argument is more general.) View each p_i as a polynomial over $\mathbb{Z}$ and consider a modulus amplifying polynomial F of degree $2\ell - 1$ such that $2^\ell > n^a$, for which it suffices $\ell \leq c_a \log n$.

The value of C is determined by

$$\sum_{i \in [n^a]} (p_i \mod 2) = \sum_{i \in [n^a]} \left(F(p_i) \mod 2^\ell\right) = \left(\sum_{i \in [n^a]} F(p_i)\right) \mod 2^\ell,$$

where the last equality holds because $2^\ell > n^a$. Each $F(p_i)$ is a polynomial of degree $\leq \log^{c_a} n$. Hence so is the sum, and the result follows. **QED**

9.6.1 The power of ACC^0

We give an example of the power of ACC^0 which complements Problem 7.2.

Theorem 9.53. Let G be a finite solvable group. Iterated product of elements in G is in ACC^0.

Proof. Let us first illustrate the main idea with a jargon-free example. Let G be the (dihedral) group whose elements are (t, b) where $t \in \mathbb{Z}_3$ and $b \in \mathbb{Z}_2$ and $(t, b)(t', b')$ equals

$$(t + t', b') \text{ if } b = 0,$$
$$(t - t', b' + 1) \text{ if } b = 1.$$

We have to show that given

$$(t_1, b_1), (t_2, b_2), \ldots, (t_m, b_m)$$

we can compute their product in ACC^0. To do this, first "take all the b to the right," i.e., compute a tuple

$$(t'_1, 0), (t'_2, 0), \ldots, (t'_m, 0), (0, b')$$

with the same product.

Here each t'_i depends on t_i and $\prod_{j<i} b_j$. This latter product can be computed with a Mod 2 gate.

Then we can compute the product of the t' with a Mod 3 gate. Finally, we can simulate both a Mod 2 and a Mod 3 gate using a Mod 6 gate and repeating bits. This concludes the proof for this case.

To prove the result for any solvable group we proceed by induction on the length of the series in the definition of solvable group, see section §A.8. The base case corresponds to the group $\{1\}$ and is therefore trivial. For the inductive step, we are given

$$g_1, g_2, \ldots, g_m \in G$$

and we want to compute their product. Write each g_i as a coset representative h_i times an element n_i of the normal subgroup N. So now we want to compute

$$h_1 n_1 h_2 n_2 \cdots h_m n_m.$$

Since the quotient is cyclic we can write $h_i = a^{\epsilon_i}$ (all these conversions can be done in brute force since the group is finite). Let

$$b_i := a^{\epsilon_1} a^{\epsilon_2} \cdots a^{\epsilon_i}$$

and note we want to compute

$$(b_1 n_1 b_1^{-1})(b_2 n_2 b_2^{-1}) \cdots (b_m n_m b_m^{-1}) b_m.$$

Each b_i can be computed by summing the exponent, which can be done with a Mod c_G gate.

Also, the product in each bracketed expression belongs to N because N is normal. By induction iterated product in N is in ACC^0, and so is iterated product in G. **QED**

9.7 Impossibility results for ACC^0

Essentially the best negative result we know for ACC^0 is the following.

Theorem 9.54. $\bigcup_k \mathrm{NTime}\left(n^{\log^k n}\right) \not\subseteq \mathrm{ACC}^0$.

In particular, NExp $\not\subseteq$ ACC^0. It remains open if NP $\subseteq$ AC^0-Mod-6.

The only proof we know of this Theorem 9.54 is a brilliant combination of the simulation by polynomials (Lemma 9.50), nondeterministic diagonalization (Problem 5.2), and quasilinear completeness (Theorem 5.16). In this section we sketch it

The high-level idea in the proof is that the satisfiability of alternating circuits with modular gates can be decided faster than brute-force, and if NExp were in ACC^0 we can use this to derive a contradiction with the non-deterministic time hierarchy (Problem 5.2). First, let us present the satisfiability algorithm.

Lemma 9.55. [Satisfiability for alternating circuits with modular gates] Given an alternating circuit C on m bits of size m^d, depth d, with Mod d gates, we can decide if there is $x \in [2]^m : C(x) = 1$ in time $2^{m-m^{c_d}}$.

Note this is noticeably faster than trying all 2^m assignments.

Proof sketch. Let $\delta := c_d$ and write an m-bit input x to C as $x = yz$ where $|y| = m^\delta$ and $|z| = m' := m - m^\delta$. Consider the circuit C' on m' input bits defined as

$$C'(z) := \bigvee_{y \in [2]^{m^\delta}} C(y, z).$$

It suffices to determine the satisfiability of C'. We shall show how to do that in time close to $2^{m'}$.

Apply the transformation in Lemma 9.50 and let p and b be the corresponding functions. The lemma was only stated for power-size circuits, but it applies to C' as well, since the $\bigvee$ gate has low-degree probabilistic polynomials. Moreover, (a representation of) the functions p and b can be computed in the desired time; for the special case of Lemma 9.50 we proved, this can be verified by inspection.

Next, compute the entire truth-table of p, i.e., the evaluations of p on every possible input. This is a classic transformation that can be performed by a divide-and-conquer approach in time quasi-linear in $2^{m'}$. Once we have the truth table we apply b to every output and determine the satisfiability. **QED**

To prove impossibility results for ACC^0, let f be a function in NTime(2^n) that is not computable in, say, NTime($2^n/n$). The existence of such f follows from the hierarchy for non-deterministic time, Problem 5.2. Consider the function h that on input $x \in [2]^n$ and $i \leq 2^n \cdot n^c$ computes the 3CNF from Theorem 5.16 on $2^n \cdot n^c$ variables, computes its first satisfying assignment if one exists, and outputs its bit i. We shall show that $h \notin$ ACC^0.

An important feature of Theorem 5.16 that we shall use is that the 3CNF is explicit: In FP we can construct an alternating circuit I (for indexing) of constant-depth that given an index to a clause outputs the corresponding literals.

Suppose towards a contradiction that $h \in$ ACC^0. By hardwiring, for every $x \in [2]^n$ there is a corresponding circuit C_x that on input i computes that bit i. We contradict the assumption on f by showing how to compute it in NTime($2^n/n$).

Consider the algorithm that on input $x \in [2]^n$ guesses the above circuit C_x. Then it constructs the following circuit D:

$$D(j) = C_x(i_0) \oplus b_0 \bigvee C_x(i_1) \oplus b_1 \bigvee C_x(i_2) \oplus b_2$$

where $I(j) = i_0 b_0 i_1 b_1 i_2 b_2$. In words, D takes as input an index j to a clause in ϕ. It computes the corresponding indexes i_0, i_1, i_2 of the variables and corresponding bits b_0, b_1, b_2 using I. Then it runs C_x on each of i_0, i_1, i_2, complements the output accordingly, and takes an Or.

Running the satisfiability algorithm in Lemma 9.55 on D determines if ϕ is satisfiable and hence if $f(x) = 1$.

9.8 The power of AC^0: sampling

Recall that even though NC^0 cannot compute parity, we showed in Exercise 7.38 that they can sample (or generate) input-output pairs of the parity function. It turns out that AC^0 has even greater sampling capabilities: We can sample $(X, f(X))$ for uniform X for any symmetric function f. This is more involved and beautiful, and is only known to be possible up to a small statistical distance (see Definition A.10 for a definition of this distance). Try for a few minutes to sample $(X, \text{majority}(X))$ efficiently using alternating circuits before reading on!

The first step is sampling a uniform permutation.

Lemma 9.56. There are alternating circuits $C : [2]^{n^c} \to [n]^n$ of depth c and size n^c whose output distribution is 2^{-n^c} close (in statistical distance) to a uniform permutation π over $[n]$.

Exercise 9.57. Assume Lemma 9.56. Prove that there are alternating circuits $C : [2]^{n^c} \to [2]^n$ of depth c and size n^c whose output distribution is 2^{-n^c} close to:
(1) a uniform string of weight i, for any $i \leq n$,
(2) $(X, \text{majority}(X))$ for uniform $X \in [2]^n$.

Proof of Lemma 9.56. The high-level idea is "dart throwing:" we view the input random bits as random pointers $p_0, p_1, \ldots, p_{n-1} \in [m]$ into $m \gg n$ words. We then write $i \in [n]$ in the p_i-th word (unhit words get "$*$"). If there are no collisions, the ordering of $[n]$ in the words gives a random permutation of $[n]$. However, it is not clear how to explicitly write out this permutation using small depth, because to determine the image of i one needs to count how many words before p_i are occupied, which cannot be done in AC^0.

The key insight is to view the words as representing the permutation in a different format, known as *cycle format*, from which we can easily write out the permutation. Just like the standard format, the cycle format represents a permutation via an array $A[0..n-1]$ whose entries contain all the elements $[n]$. However, rather than thinking of $A[i]$ as the image of i, we think of the entries of A as listing the cycles of the permutation in order. Each cycle is listed starting with its smallest element, and cycles are listed in decreasing order of the

first element in the cycle. This format allows for computing the permutation efficiently: the image of i is the element to the right of i in A, unless the latter element is the beginning of a new cycle, in which case the image of i is the first element in the cycle containing i. Identifying the first element of a cycle is easy, because it is smaller than any element preceding it in A. This format works even if the array A has $m \gg n$ words, of which $m - n$ are unhit and marked by "$*$."

One can now verify that computing the image of i is in AC^0. Here in particular we use the fact that in AC^0 we can, given an array A and an index i, compute the least $j > i$ such that $A[j]$ is not "$*$". This can be accomplished by trying all j, noting that one can determine if a fixed j is the least $j > i$ such that $A[j]$ is not "$*$" using one unbounded fan-in And.

This gives a uniform permutation, unless there is a collision in the pointers, which happens with probability $\leq 1/n^c$ for $m \geq n^c$. Because we can detect if there is a collision in AC^0, the error probability can be shrunk to exponentially small as in section §3.1. Specifically, pick ℓ uniform and independent sets of pointers $p_0^i, \ldots, p_{n-1}^i$, $i \in [\ell]$, where each pointer has range $[m]$ for m the smallest power of 2 larger than $2n^2$ (thus each pointer can be specified by $\log m$ bits). If there exists i such that the pointers $p_0^i, \ldots, p_{n-1}^i$ are all distinct (i.e., there are no collisions), then run the above algorithm on the output corresponding to the first such i. This results in a random permutation.

Since the pointers are chosen independently, the probability that there is no such i is

$$\mathbb{P}\left[\forall i \in [\ell], \exists j, k \in [n] : p_j^i = p_k^i\right] = \mathbb{P}\left[\exists j, k \in [n] : p_j^0 = p_k^0\right]^{\ell} \leq (1/2)^{\ell}.$$

Choosing say $\ell := n$ proves the lemma. **QED**

9.9 Problems

Problem 9.1. Prove that integer multiplication is not in AC^0.

Problem 9.2. Prove that Parity $\notin \mathrm{AC}^0$-Mod-3 by developing the polynomial method over $\mathbb{F}_3$.

Guideline: Follow the argument in section 9.1 and prove:

(1) AC^0-Mod-3 has low-degree probabilistic polynomials over $\mathbb{F}_3$.

(2) Any polynomial of degree d over $\mathbb{F}_3$ fails to compute parity on at least a $1/2 - cd/\sqrt{n}$ fraction of the n-bit inputs.

Exercise 9.58. Prove that degree-d $\mathbb{F}_2$ polynomials have correlation $\leq cd/\sqrt{n}$ with Majority. Hint: Follow closely the proof of Theorem 9.8.

Problem 9.3. Let $a(n) : \mathbb{N} \to \mathbb{N}$ be a function. Show that $\text{Gap-Maj}_{1/2-1/\log^{a(n)} n, 1/2+1/\log^{a(n)} n}$ is in AC^0 iff $a(n)$ is bounded by a constant.

9.10 Notes

Impossibility results for AC are among the most famous results in complexity and were obtained in the 80's in [121, 11, 389, 289, 320, 163] via various techniques. The first two works obtained super-power results, the others exponential. Each work gives a negative result for a symmetric function and hence applies to majority as well.

The polynomial method over finite fields such as $\mathbb{F}_2$ Theorem 9.8 is from [289] (an earlier work developed a somewhat related method for *monotone* circuits). The precise degree bound in Lemma 3.6 in [219]. Theorem 9.5 is from [321]. Stronger bounds for alternating circuits with parity gates matching switching-lemma parameters are obtained in [359] but for less explicit functions (think NExp), building on the proof of Theorem 9.54. For a survey on correlation bounds for AC^0-$\oplus$ and more on Question 9.13 see [374].

The polynomial method over the reals, and the lower bounds for Maj-AC^0 Theorem 9.17 are from [31]. The equivalence between 1 majority gates and more is from [55]. Exponential bounds on the correlation between parity and AC^0 were first proved using switching lemmas (discussed below). The proof by reduction to Maj-AC^0 lower bounds presented in section 9.2.4 is from [213]. The proof in [213] used the XOR and hard-core set lemmas section 11.3.2. The proof we presented is a later simplification which was presented first in [365], Chapter 2.

As we have seen switching lemmas are an instantiation of the restrict-and-simplify method. The first such lemma is Lemma 9.24, from [121, 11]. Other such lemmas were obtained in [389, 168, 165], the last one giving a refined tradeoff between size and correlation for parity. See also the book [163]. For a switching lemma primer see [49]. Lemma 9.25 is essentially in [168]. For a perspective on these lemmas see [373]. Various proofs exist; I have tried to give a slightly different exposition of a proof based on an encoding argument.

Simulations of various classes by alternating circuits go back to [264]. The main ideas behind Theorem 9.36 and its proof are from [108]. The stated version is from [351]. Our exposition is based on [365], apparently the first exposition after [351].

Turning to ACC^0, Lemma 9.50 (and Theorem 9.40) is from [17, 392, 57]. The result that NEexp is not a subset of ACC^0 is from [386]. One step of the original proof was somewhat indirect and was streamlined in [198], this is reflected in the exposition here. The general approach goes back to [385]. The class NExp was later improved to quasi-power non-deterministic time, yielding Theorem 9.54, in [260]. Theorem 9.53 is from [255].

The sampling capabilities of AC^0, Lemma 9.56, follow from [244, 155], though they don't mention AC^0. Our presentation follows [369].

Two lines of research appear intertwined around few main ideas. The first line is the study of complexity classes defined in terms of various quantifiers (or operators). The second is that of circuits with various gates. One basic idea is that And can be approximated by low-degree polynomials. This idea appears in [353] and [289]. Another basic idea is that of modulus amplifying polynomials. They originate from [338] and were studied further in [392, 57], the latter proving Lemma 9.51. A third basic idea is that gap majority is in AC^0.

This is from [11] (where Lemma 9.43 is proved) and [318]. For more constructions, see [364] and [12].

Chapter 10

Proofs

The notion of proof is pervasive. We have seen many proofs in this book until now. But the notion extends to others realms of knowledge, including empirical science, law, and more. Complexity theory has contributed a great deal to the notion of proof, with important applications in several areas such as cryptography.

10.1 Static proofs

As remarked in Chapter 5, we can think of problems in NP as those admitting a solution that can be verified efficiently, namely in P. Let us repeat the definition of NP using the suggestive letter V for verifier.

Definition 10.1. A function $f : X \subseteq [2]^* \to [2]$ is in NP iff there is $V \in \mathrm{P}$ (called "verifier") and $d \in \mathbb{N}$ s.t.:

$$f(x) = 1 \Leftrightarrow \exists y \in [2]^{|x|^d} : V(x, y) = 1.$$

We are naturally interested in fast proof verification, and especially the complexity of V. It turns out that proofs can be encoded in a format that allows for very efficient verification. This message is already in the following.

Theorem 10.2. Definition 10.1 does not change if we insist that V_n is a 3CNF.

That is, whereas when defining NP as a proof system we considered arbitrary verifiers V in P, in fact the definition is unchanged if one selects a very restricted class of verifiers: small 3CNFs.

Proof. This is just a restatement of Theorem 5.1. **QED**

This extreme reduction in the verifier's complexity is possible because we are allowing proofs to be long, longer than the original verifier's running time. If we don't allow for that, such a reduction is not known. Such "bounded proofs" are very interesting to study, but we shall not do so now.

Instead, we pursue a different direction. The 3CNF in the above theorem still depends on the entire proof. We can ask for a verifier that only depends on few bits of the proof. Taking this to the extreme, we can ask whether V can only read a constant number of bits from y. Without randomness, this is impossible.

Exercise 10.3. Suppose V in Definition 10.1 only reads $\leq d$ bits of y, for a constant d. Show that the corresponding class would be the same as P.

Surprisingly, if we allow randomness this is possible. Moreover, the use of randomness is fairly limited – only logarithmically many bits – yielding the following central characterization, a.k.a. the PCP theorem.

Theorem 10.4. A function $f : X \subseteq [2]^* \to [2]$ is in NP iff there is $V \in \mathrm{P}$ and $d \in \mathbb{N}$ s.t.:
$f(x) = 1 \Rightarrow \exists y \in [2]^{|x|^d} : \mathbb{P}_{r \in [2]^{d \log |x|}}[V(x, y, r) = 1] = 1$,
$f(x) = 0 \Rightarrow \forall y \in [2]^{|x|^d} : \mathbb{P}_{r \in [2]^{d \log |x|}}[V(x, y, r) = 1] < 0.01$,
and moreover V reads $\leq d$ bits of y.

We give a proof of this theorem assuming Theorem 4.33.

Proof. For the "if" direction, note we can enumerate over all choices for r in power time. So we can compute in P the probability that V accepts, regardless of how many bits of the proof it reads.

For the "only if" direction, it suffices to give the proof system for 3Sat. In turn, by Theorem 4.33 it suffices to give it for $(1 - c)$-Gap-3Sat. Let ϕ be an instance of $(1 - c)$-Gap-3Sat. The random choice r is used to select a uniform clause in ϕ, the verifier then reads the 3 bits from that clause, and accepts accordingly. We can repeat this a constant number of times to reduce the error probability. **QED**

10.2 Zero-knowledge proofs

In Theorem 10.4 the verifier gains statistical confidence about the validity of the proof, just be inspecting a constant number of bits. Hence the verifier "learns" at most a constant number of bits of the proof. This is remarkable, but we can further ask if we can modify the proof so that the verifier learns nothing about the proof. Such proofs are called *zero knowledge* and are extensively studied and applied.

We sketch how this is done for Gap-3Color, which is also NP-complete. (This is the problem of deciding if a graph has a 3-coloring or every coloring of the nodes with 3 colors will result in a constant fraction of edges with the same color at the nodes.) Rather than a single proof y, now the verifier will receive a random proof Y. This Y is obtained from a 3 coloring y by randomly permuting colors (so for any y the corresponding Y is uniform over 6 colorings). The verifier will pick a random edge and inspect the corresponding endpoints, and accept if they are different.

The verifier "learns nothing" because all that they see is two random different color. One can formalize "learning nothing" by noting that the verifier can produce this distribution

by themselves, without looking at the proof. (So why does the verifier gain anything from y? The fact that a proof y has been written down means that colors have been picked so that every two endpoints are uniform colors, something that the verifier is not easily able to reproduce.)

This gives a zero-knowledge proof for verifiers that follow the protocol of just inspecting an edge. In a cryptographic setting one has to worry about verifiers which don't follow the protocol. Using cryptographic assumptions, one can force the verifier to follow the protocol by considering an *interactive* proof: First a proof y is committed to but not revealed, then the verifier selects an edge to inspect, and only then the corresponding colors are revealed, and only those. This protocol lends itself to an entertaining physical implementation, for example by placing on a table a coloring with all colors covered (a commitment), and then only opening one color selected by the audience.

10.3 Interactive proofs

We now consider interactive proofs where the verifier V engages in a protocol with a prover P. Given an input x to both V and P, the verifier asks questions, the prover replies, the verifier asks more questions, and so on. The case of NP corresponds to a passive verifier which does not ask questions, and a prover that simply sends the proof y to the verifier. It turns out that for the general case of active verifiers, it suffices for the verifier to send uniformly random questions $Q_1, Q_2, \ldots$ to the prover. This leads to a simple definition.

Definition 10.5. We denote by IP (for *interactive proof systems)* the class of functions $f : X \subseteq [2]^* \to [2]$ for which there is $V \in \mathrm{P}$ and $d \in \mathbb{N}$ such that for every $x \in [2]^n$, letting $b := n^d$:

- If $f(x) = 1$ then $\exists P : [2]^* \to [2]^b$ such that

$$V\left(x, Q_1, P(Q_1), Q_2, P(Q_1, Q_2), \ldots, Q_b, P(Q_1, Q_2, \ldots, Q_b)\right) = 1$$

  for every $Q_1, Q_2, \ldots, Q_b \in [2]^b$.

- If $f(x) = 0$ then $\forall P : [2]^* \to [2]^b$ we have

$$\mathbb{P}_{Q_1, Q_2, \ldots, Q_b \in [2]^b}\left[V\left(x, Q_1, P(Q_1), Q_2, P(Q_1, Q_2), \ldots, Q_b, P(Q_1, Q_2, \ldots, Q_b)\right) = 1\right] \leq 1/3.$$

In other texts interactive proofs where the verifier only asks random questions are denoted AM, whereas IP is reserved for general interactive protocols where the verifier can ask arbitrary questions. But this distinction is not too important here because of the following amazing result which shows the power of interactive proofs, compared to non-interactive proofs.

Theorem 10.6. IP = PSpace.

Note that like for NP, it is not clear if IP is closed under complement from its definition. Yet it obviously follows from Theorem 10.6.

As a first step towards the proof of Theorem 10.6 we show that IP contains problems not known to be in NP. We introduce a problem that is an extension of ZIC (Definition 3.8). First we define arithmetic circuits over the integers (later we will consider finite fields, see below and Definition 14.8).

Definition 10.7. An *arithmetic circuit* of size s in n variables is a sequence of assignments to gates g_i where assignment i is of the following types:

- $g_i := a$, with $a \in \mathbb{N}$
- $g_i := t \circ t'$, where $\circ$ is either $+$ or $\times$, and each of the terms t, t' is either a gate g_j with $j < i$, or a variable x_k.

The circuit represents the natural computed by the last instruction, with all operations over $\mathbb{N}$.

We are interested in computing the sum of the outputs of the circuit over every $x \in [2]^n$, modulo p, i.e., over the field $\mathbb{F}_p$. The same theory works over any field; we focus on the prime case for simplicity. We shall be able to check this sum in IP, assuming the p is large enough and the circuit computes a polynomial of small enough degree d. The latter assumption is formally enforced by including 1^d in the input (cf remark in section §1.2).

Definition 10.8. The SAC (sum arithmetic circuit) problem: Given an arithmetic circuit $C(x_1, x_2, \ldots, x_v)$ computing a polynomial of degree d, and given 1^d, a prime p satisfying $(1 - d/p)^v \geq 2/3$, and $s \in [p]$, decide if

$$\sum_{x_1, x_2, \ldots, x_v \in [2]} C(x_1, x_2, \ldots, x_v) = s \mod p. \tag{10.1}$$

Theorem 10.9. SAC is in IP.

The protocol in this result is known as the *sum-check protocol*.

Proof. If $v = 1$ then V can decide this question by itself, by evaluating the circuit. For larger v we give a way to reduce v by 1.

As the first prover answer, V expects a univariate polynomial a of degree d modulo p in the variable x, which is meant to be

$$s'(x) := \sum_{x_2, x_3, \ldots, x_v \in [2]} C(x, x_2, x_3 \ldots, x_v).$$

Note that it is feasible for the prover to send a since its degree is no more than d which is no more than the length of the SAC instance, and its coefficients only need to be sent modulo p.

V checks if $a(0)+a(1)=s$, and if not rejects. Otherwise, it recursively runs the protocol to verify that

$$\sum_{x_2,x_3,\ldots,x_v\in[2]} C(Q_1,x_2,x_3,\ldots,x_v) = a(Q_1), \tag{10.2}$$

where Q_1 is uniform in $[p]$.

This concludes the description of the protocol. We now verify its correctness.

In case equation (10.1) is true, P can send polynomials that cause V to accept.

Otherwise, suppose equation (10.1) is false. Consider the first iteration of the protocol. We have $s'(0)+s'(1)\neq s$. Hence, unless V rejects right away because $a(0)+a(1)\neq s$, we also have $a(x)\neq s'(x)$. The polynomials a and s' have degree $\leq d$. Hence $a-s'$ is a non-zero polynomial of degree $\leq d$, and therefore it has $\leq d$ roots (Fact A.50). So we get:

$$\mathbb{P}_{Q_1}[a(Q_1)\neq s'(Q_1)]\geq 1-d/p.$$

When this event occurs, equation (10.2) is again false, and we can repeat the argument. Overall, the probability that we maintain a false statement throughout the protocol is $\geq (1-d/p)^v \geq 2/3$. **QED**

Exercise 10.10. The protocol exchanges cv messages (equivalently, has v alternations between prover and verifier, or cv rounds). Modify it so that it exchanges only $v/100$ messages. Hint: Use the Polynomial Identity Fact A.50.

To apply the sum-check protocol to *boolean* rather than *algebraic* circuits we use a far-reaching technique: *arithmetization*. We construct an arithmetic circuit C_ϕ over a field $\mathbb{F}$ which agrees with ϕ on *boolean* inputs, but that can then be evaluated over other elements of the field. This is done in the following way:

$$\begin{aligned} x &\to x \\ f\wedge g &\to f\cdot g \\ f\vee g &\to f+g-f\cdot g \\ \neg f &\to 1-f. \end{aligned}$$

Theorem 10.11. Given a 3CNF formula ϕ and $k\in\mathbb{N}$, deciding if ϕ has exactly k satisfying assignments is in IP.

Proof. Let C_ϕ be the arithmetization of ϕ, which note has degree $\leq cn$. Let p be a prime of size $\leq 2^{cn}$. It suffices to solve the SAC instance (Definition 10.8) $\sum_x C_\phi(x) = k \mod p$, since $k\leq 2^n$. The prime specifying the field can be sent from the prover (the verifier can check it using that Primes is in P). **QED**

To show PSpace $\subseteq$ IP or in fact even weaker statements like Π_2P $\subseteq$ IP one needs to consider formulas with both $\exists$ and $\forall$ quantifier. To determine the validity of such a formula

it is natural to proceed as in Theorem 10.11 and make the following substitutions:

$$\exists x : f(x) \to \sum_{x \in [2]} f(x)$$
$$\forall x : f(x) \to \prod_{x \in [2]} f(x).$$

This is a valid transformation: the formula is true iff the corresponding expression is > 0.

We would then be running the sum-check protocol, with the difference that when the polynomial p corresponds to a Π gate we check that $p(0) \cdot p(1) = s$ (instead of $p(0) + p(1) = s$, note here the variable s does not just correspond to a sum). We call this the *sum-prod-check* protocol.

The main problem with this approach is that the degree of the polynomials to be sent in the protocol can explode, making it unfeasible for the verifier to even receive such polynomials.

Example 10.12. Let

$$f(x, y_1, \ldots, x_k) := \exists x \forall y_1 y_2 \cdots y_k : x.$$

The arithmetization of this would be

$$f(x, y_1, \ldots, x_k) := \sum_{x \in [2]} \prod_{y_1, \ldots, y_k \in [2]} x = \sum_{x \in [2]} x^{2^k}.$$

The polynomial in x corresponding to the $\sum$ gate has too large a degree.

The solution is similar in spirit to the reduction of circuits to Sat, Theorem 5.1: we are going to add new variables.

Lemma 10.13. Let $f = \exists x_1 \forall x_2 \exists x_3 \cdots \forall x_k g(x_1, x_2, \ldots, x_k)$ be a formula where g is quantifier-free. All quantifiers are over [2]. Proceeding from left to right, replace an occurrence of

$$\forall x_i \exists x_{i+1} \cdots Q_k x_k : g(x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \cdots, x_k)$$

with the formula

$$\forall x_i \exists x'_1, x'_2, \ldots, x'_{i-1} : \left(\left(\wedge_{j=1}^{i-1} x_j \iff x'_j \right) \wedge \exists x_{i+1} \cdots Q_k x_k : g(x'_1, x'_2, \ldots, x'_{i-1}, x_i, x_{i+1}, \cdots, x_k) \right).$$

At the end of the process we have obtained f' s.t.:
(1) $f \iff f'$,
(2) $|f'| \leq |f|^c$,
(3) If f' is true then there exists a prover that in the sum-prod-check protocol on the arithmetization of f' sends polynomials of degree at most the degree of the arithmetization of g plus a constant.

Note that $x_j \iff x'_j$ means that the variables are equal; it can be written as $(x_j \wedge x'_j) \vee (\neg x_j \wedge \neg x'_j)$. We picked f to start with an $\exists$ and and end with a $\forall$ quantifier. This is w.l.o.g., since we can always ignore some variables.

Proof. (1) and (2) follow by inspection.

(3): Note that we are applying the sum-prod-check protocol to the arithmetization of a formula that is not in prenex form (a formula is in prenex form if all quantifiers are at the beginning). However, it is almost in prenex form, the only difference are the $\iff$ equality checks. In a generic step of the execution, we have fixed some variables to field elements, and we are considering a univariate polynomial in a variable x corresponding to a quantifier Qx. If a $\forall$ quantifier appears to the right of x, then the degree of x is $\leq$ the degree of the equality check; the rest of the formula does not involve the variable x and thus does not affect its degree. If no $\forall$ quantifier appears then the degree is at most that of the equality check plus the degree in the arithmetization of g. **QED**

Example 10.14. Let us return to Example 10.12 and set $k = 2$ for simplicity. Thus we consider the formula

$$\exists x \forall y_1 y_2 : x.$$

First note that the arithmetization is

$$\sum_{x \in [2]} x^4,$$

which is a polynomial of degree 4. Let us now see how the transformation in Lemma 10.13 reduces the degree. Applying it to the leftmost $\forall y_1$ quantifier in f we get

$$\exists x \left[\forall y_1 \exists x' : (x' \iff x) \wedge \forall y_2 : x'\right].$$

Next the transformation would be applied again to the $\forall y_2$ quantifier. We don't write this down because the above formula only depends on x', but this wouldn't change the degree of x in the protocol. The important point is that in the arithmetization the degree in x is only the degree of the equality check, which is 2 (whereas recall previously it was 4).

Proof of IP=PSpace Theorem 10.6.. To show IP $\subseteq$ PSpace one can give a recursive algorithm that, given a verifier and an input, computes the highest probability that the verifier can be made to accept by some prover. The details of this are similar to the proof that QBF $\in$ PSpace, see Theorem 6.32, and are omitted.

For the other direction it suffices to show that QBF $\in$ IP by Theorem 6.32. Given a QBF f, the verifier applies the transformation in Lemma 10.13 to obtain f'. The verifier then computes the arithmetization h of f' where $\exists x \in [2]$ in f' becomes $\Sigma_{x \in [2]}$ in h and $\forall x \in [2]$ becomes $\prod_{x \in [2]}$. Hence h denotes a number and we note

$$f \text{ is true} \iff f' \text{ is true} \iff h > 0.$$

The prover will show that $h > 0$ by first sending a prime p and then proving $h = K$ over $\mathbb{F}_p$ using the sum-prod-check protocol. Note that g is at most doubly exponential in n. By Lemma 3.11 a prime of n^c bits can be found so that $h \neq 0$ over $\mathbb{F}_p$. By Lemma 10.13 during the execution of the sum-prod-check protocol the degree remains $\leq cn$. **QED**

10.4 Delegating computation: Interactive proofs for muggles

The study of interactive proofs is rich. Many interrelated aspects are of interest, including the efficiency of the verifier, the number of rounds of the protocol, the communication complexity, and the error parameter. The efficiency of the prover is also of interest. By this we mean the efficiency of the prover in the case $f(x) = 1$. The verifier should reject with high probability in case $f(x) = 0$ even when interacting with a computationally unbounded prover. (Again, variants in which the protocol only withstands computationally bounded provers are of interest too).

In this section we discuss interactive proofs for problems in P. The goal is having the verifier in Quasi-Linear-Time and the prover in FP. Surprisingly, this can be accomplished for any function in NL or any boolean function in NC that satisfies a certain uniformity condition.

Theorem 10.15. Any function in NL has interactive proofs where the verifier runs in quasi-linear time, and moreover when a proof exists it can be computed in FP.

The appeal of this theorem is clear: One can delegate expensive computation of functions in L (for example, functions which naively take time n^{10}) and verify it in quasi-linear time; and the delegated computation is still feasible.

The proof of Theorem 10.15 displays a beautiful interplay between algebra and computation, and in fact, we will establish stronger results (applying to other classes, such as a uniform version of NC). Before this, however we give a simple example where an efficient prover exists. This serves as a warm-up for the proof of Theorem 10.15.

10.4.1 Warm-up: Counting triangles

We consider the problem of counting the number of triangles in a graph. Such a problem can be trivially solved in power time. Whether faster run time such as quasi-linear is unknown. We show that such time bounds can be achieved via interaction, and moreover the proofs are still feasible. We can think of the graph as being given as a list of edges.

Theorem 10.16. Given an undirected graph G with n nodes and $m \geq n$ edges, and an integer s, there is an interactive proof for deciding if the number of triangles in G is s such that the verifier is in Quasi-Linear-Time and the prover is in FP.

Proof. We construct a suitable arithmetic circuit and then apply the sum-check protocol from Theorem 10.9. The circuit has $v := 3 \log n$ variables x_i organized in 3 blocks y_0, y_1, y_2 where each y_i consists of $\log n$ variable. Each y_i corresponds to a node in the graph. Thus, the number of triangles can be written as

$$\sum_{x_1, x_2, \ldots, x_v \in [2]} E(y_0, y_1) \cdot E(y_0, y_2) \cdot E(y_1, y_2)$$

where E is 1 if y_i and y_j are an edge in G.

To run the sum-check protocol we need an arithmetic circuit, that is, we need to be able to make sense of evaluating the circuit over large fields. The function E is only defined over bits, so we need to view it as a polynomial that can be evaluated over larger fields. At the same time, computing this polynomial should be easy for the verifier (so we can't just say it has some polynomial like any other function, since the polynomial could have degree $2 \log n$ and require quadratic time, which isn't in the verifier budget). The standard expansion will do. Define

$$\widehat{E}(z, z') := \sum_{\alpha, \alpha'} [z = \alpha] \cdot [z' = \alpha'] \tag{10.3}$$

where the sum is over all edges $\{\alpha, \alpha'\}$ and $z = z_0 z_1 \cdots z_{\log n - 1}$. (We should assume that the graph has no self loops.) In turn, we can write

$$[z = \alpha] \iff \prod_{i \in [\log n] : \alpha_i = 1} z_i \prod_{i \in [\log n] : \alpha_i = 0} (1 - z_i)$$

and the same for z'. Note $\widehat{E}$ is a polynomial of degree $2 \log n$.

With this notation, the verifier needs to verify that

$$\sum_{x_1, x_2, \ldots, x_v \in [2]} \hat{E}(y_0, y_1) \cdot \hat{E}(y_0, y_2) \cdot \hat{E}(y_1, y_2) = s.$$

The whole expression is a polynomial of degree 3 times the degree of $\hat{E}$, that is $6 \log n$.

Because the sum is $\leq n^c$, by the remaindering Theorem 6.14 it suffices to verify the expression modulo a prime p uniformly chosen from a set of $\geq c \log n$ primes. We shall run the sum-check protocol over such a field $\mathbb{F}_p$. For the correctness of the sum-check protocol (Theorem 10.9) it suffices that $p \geq \log^c n$. Selecting a uniform prime $\leq \log^c n$ suffices for the overall correctness (most primes will be large enough for Theorem 10.9, using Lemma 3.11).

Let us now verify the running times. The prover at each round sends a univariate polynomial which is obtained by summing over n^c values. By inspection, this polynomial has degree 2 (degree 1 for each of the factors $\hat{E}$ where the variable occurs). Hence the prover is in FP. The verifier at each round except the last one needs to evaluate this polynomial, and perform a constant number of field operations. This takes time $\log^c n$. At the last round, the verifier needs to evaluate

$$\hat{E}(y'_0, y'_1) \cdot \hat{E}(y'_0, y'_2) \cdot \hat{E}(y'_1, y'_2)$$

for some $y'_i \in \mathbb{F}_p$. Each factor $\widehat{E}(y'_i, y'_j)$ can be computed by the verifier using equation (10.3). This requires going through the m edges of the graph, and for each edge perform $c \log n$ field operations. This concludes the proof that the verifier is in Quasi-Linear-Time. **QED**

Exercise 10.17. Similarly to Exercise 10.10, show how to reduce the number of rounds to constant while maintaining the complexity of the verifier and the prover.

10.4.2 Delegating NC

In this section we prove a stronger theorem than Theorem 10.15, for circuits. Naturally, the circuits need to satisfy a certain uniformity condition. This condition (stated in the theorem) will be algebraic, as one can expect from arithmetization.

Before this we should discuss some conventions on circuits. We shall consider functions computable by circuits of power-size $k(n) := n^e$ and depth $d(n) := \log^e n$. (The results generalize to larger depth, with the depth factoring in the verifier runtime.) It will be convenient to arrange the circuit into a matrix of $d + 1$ rows indexed with $[d + 1]$ and k columns, where row 0 corresponds to the input literals, here also called gates and padded with zero gates to length k, while row $i > 0$ contains the gates at depth i. Row 0 also contains the constant gates 0 and 1, and further we assume that gates in rows > 0 are all NAnd (any other gate can be simulated with these gates).

The circuit is leveled, so gates on row i take as input gates on row $i - 1$ only, at the cost of possibly duplicating gates. Row d contains the output gate.

We can index the gates in one row using $\log k = e \log n$ bits. Naturally, we will need to work over larger fields; at the same time, this switch to larger fields should not cause the number of descriptions of gates to become infeasible. So, we shall pick a field $\mathbb{F}$ and $\mathbb{H} \subseteq \mathbb{F}$ and index gates by strings of

$$m := \frac{\log k}{\log |\mathbb{H}|}$$

elements from $\mathbb{H}$. When working over larger fields, we will use the same number of elements, but this time from $\mathbb{F}$. In terms of bits this will be

$$\log |\mathbb{F}| \cdot m$$

which is within a constant factor of $\log k$ if $|\mathbb{F}|$ is a power of $|\mathbb{H}|$. Indeed, we set

$$|\mathbb{H}| := \log n,$$
$$|\mathbb{F}| := \log^{c_e} n.$$

As anticipated, the results apply to NC provided that a suitable uniformity condition is met.

Definition 10.18. We denote by algebraic-uniform (boolean) NC the class of functions $f : [2]^* \to [2]$ for which there is e such that:

(1) f_n is computable by a circuit C_n of size n^e and depth $\log^e n$, for $n \geq c$,

(2) given n (in binary) we can compute in FP polynomials $\phi_i : (u, v, w) \in \mathbb{H}^{3m} \to [2]$ for $i = 1, 2, \ldots, d$ of degree $\log^e n$ where ϕ_i computes if gate u in row i takes as input gates v and w in row $i - 1$ in C_n.

We used the same parameter e for both the complexity of f and the degree of ϕ for simplicity and w.l.o.g., for increasing e makes the definition easier and easier to satisfy, as we can always ignore some gates. Satisfying this Definition 10.18 is typically not an issue. In particular, we have:

Lemma 10.19. NL $\subseteq$ algebraic-uniform NC.

The proof of this is somewhat tangential to proof systems. Yet let us give it.

Proof. Let $f \in$ NL. The circuit for f_n can have the following structure (cf Theorem 7.5) for $a = c_f$. There is a layer A of n^a gates at depth 1, each depending on a single input bit. The rest of the circuit consists of $a \log n$ copies of a circuit $M : [2]^{n^a} \to [2]^{n^a}$ stacked on top of each other, with one copy taking A as input.

Here A computes the $n^{a/2} \times n^{a/2}$ adjacency matrix of the configuration graph of f on input x. Each gate corresponds to an edge $u \to v$ where $u \to v$ are configurations.

M is matrix squaring but with $+$ replaced by $\vee$. Each output bit is an $\vee$ over $n^{a/2}$ products of 2 gates.

We now describe the ϕ_i. First we give a AC^0 circuits for the $\phi + i$ (over the $3m \log |\mathbb{H}|$ bits corresponding to the field elements). Then we use a generic transformation to obtain polynomials.

For the function ϕ_1, corresponding to A, there are some low-level details. Recall ϕ_1 is given as input a gate representing two configurations u, v, and a gate representing an input bit literal ℓ_i, or a constant b, and we need to decide if that's the right connection. Configuration u may not read any input. In this case the gate should be the constant 1 if v is the next configuration and the constant 0 otherwise, that is, we have to make sure that it's connected to the right constant b. Looking back at Definition 1.1 and using Exercise 6.3, we can compute v from u in AC^0. This concludes the case when u does not read any input. Otherwise, configuration u reads an input bit x_i. In this case, the value of the bit is in some register in v, and ϕ_1 checks that the gate in A is connected to the corresponding literal ℓ_i.

We not turn to the ϕ_i for $i > 1$. These correspond to M. We can arrange the $\vee$ in the description of M above in a binary tree, and index by $i \to 2i, i \to 2i + 1$, as in the heap data structure. Then ϕ_i can be computed just using bit-shift and addition by 1, which is in AC^0 (Example 9.2).

This concludes the description of the circuit for the ϕ_i. Now we transform these circuits into polynomials. We can view each circuit as using $\wedge$ and $\neg$ gates, and we arithmetize it (replace $\wedge$ by multiplication and $\neg x$ by $1 - x$). The degree is as desired since the circuit has power size and constant depth. We could just use this polynomial if our inputs were bits, but they are in fact field elements in $\mathbb{H}$. However, we can extract each of the $\log |\mathbb{H}|$ bits using a polynomial of degree $\leq |\mathbb{H}|$ computable in FP. There are several ways to do

this; since $\mathbb{H}$ has size only $\log n$, a brute-force approach (which works for every function) suffices. Namely, for any function $f : [2]^{\log |\mathbb{H}|} \to [2]$ in FP we can write down the polynomial $p(x) = \sum_{a:f(a)=1} [x = a]$ (see Exercise 10.20 for $[x = a]$). This has degree $\leq |\mathbb{H}|$ and can be computed in time $|\mathbb{H}|^c \leq \log^c n$. Composing these polynomials with the ones coming from the alternating circuit concludes the proof. **QED**

Exercise 10.20. Give a polynomial for $[x = a]$ of degree $\mathbb{H} - 1$. Hint: Use Fact A.29.

Thanks to Lemma 10.19, Theorem 10.15 follows from the following main statement about circuits.

Theorem 10.21. Algebraic-explicit NC has interactive proofs where the verifier is in Quasi-Linear-Time and the prover is in FP.

In the remainder of this section we present the proof of this Theorem 10.21. Let f be in algebraic-uniform NC and e as in Definition 10.18. Given input x, let

$$\alpha_i : \mathbb{H}^m \to [2]$$

denote the value of the gates in row i. So α_0 contains the input and α_d the output. The protocol proceeds in stages. In stage $i = d, d-1, \ldots, 1$ a claim of the form

$$\widehat{\alpha}_i(z_i) = b_i$$

is reduced to a claim of the form

$$\widehat{\alpha}_i(z_i) = b_i.$$

Here $z_i \in \mathbb{F}^m, b_i \in \mathbb{F}$, and $\widehat{\alpha_i} : \mathbb{F}^m \to \mathbb{F}$ is the arithmetization of α_i which agrees with it over $\mathbb{H}^m$ and is defined as

$$\widehat{\alpha_i}(x) := \sum_{y \in \mathbb{H}^m} [x = y] \cdot \alpha_i(y).$$

Here $[x = y]$ is a polynomial that, when evaluated over $\mathbb{H}^m$, computes equality.

Exercise 10.22. Give such a polynomial for $[x = y]$ that has degree $m(\mathbb{H} - 1)$ and can be evaluated in time $\log^{c_e} n$.

Note these polynomials are never sent to the verifier, as they are not within their budget. Various *univariate* restrictions of these polynomials (of the same degree) will sent to the verifier. On the other hand, the polynomials are computable by the prover in power time. What constitutes a "claim" are the values i, z_i, b_i.

At Stage $i = d$ we have that z_d is the output gate and b_d is the output of the circuit.

Exercise 10.23. Explain how a claim $\widehat{\alpha}_0(z_0) = b_0$, corresponding to the input level, is verified in Quasi-Linear-Time without further interaction.

We now discuss the reduction. Recall we assume that the circuit consists of NAnd gates only). Hence we can write

$$\widehat{\alpha}_i(z) = \sum_{u,v,w \in \mathbb{H}^m} [z = u] \cdot \phi_i(u,v,w) \cdot (1 - \widehat{\alpha}_{i-1}(v) \cdot \widehat{\alpha}_{i-1}(w)).$$

However, we need to apply the sum-check protocol so we need algebraic computation. The current claim is that the rhs equals b_i. The term inside the sum on the rhs is an algebraic circuit computing a polynomial of degree

$$\leq m \cdot |\mathbb{H}| + \log^e n + 2m|\mathbb{H}| \leq \log^{c_e} n.$$

The sum-check protocol reduces the current claim to

$$[z = \widehat{u}] \cdot \phi_i(\widehat{u},\widehat{v},\widehat{w}) \cdot (1 - \widehat{\alpha}_{i-1}(\widehat{v}) \cdot \widehat{\alpha}_{i-1}(\widehat{w})) = b,$$

for some $\widehat{u},\widehat{v},\widehat{w} \in \mathbb{F}^m$ and $b \in \mathbb{F}$. (We only stated the sum-check protocol for sums over $[2]$, but one can readily extend it to sums over larger sets such as $\mathbb{H}$: In each iteration the prover sends a univariate polynomial of degree $\log^{c_e} n$ and the verifier evaluates it at $|\mathbb{H}| = \log n$ points). This takes time $\log^{c_e} n$. The error will be $\leq \log^{c_e} n/|\mathbb{F}| \leq 1/\log^{c_e} n$ in each iteration, by our choice for $|\mathbb{F}|$. Overall, the error is $\leq 1/\log^{c_e} n$.

However, the new claim appears to require two evaluations of $\widehat{\alpha_{i-1}}$, which would yield an exponential increase in complexity. To avoid it, we use another idea to reduce the evaluation at two points to a single point. The prover sends a univariate polynomial $p(t)$ of the same degree as $\widehat{\alpha_{i-1}}$, which is meant to be $p(t) := \widehat{\alpha_{i-1}}(\widehat{v} + t(\widehat{w} - \widehat{v}))$. The verifier checks that

$$[z = \widehat{u}] \cdot \phi_i(\widehat{u},\widehat{v},\widehat{w}) \cdot (1 - p(0) \cdot p(1)) = b$$

and it rejects if it does not hold. If it does hold, it picks a uniform value $t \in \mathbb{F}$ and the new claim is now

$$\widehat{\alpha_{i-1}}(\widehat{v} + t(\widehat{w} - \widehat{v})) = p(t).$$

Note that if $p(t) \neq \widehat{\alpha}_i(\widehat{v} + t(\widehat{w} - \widehat{v}))$ then the claim is still false, except with probability, again, $\log^{c_e} n/|\mathbb{F}| \leq 1/\log^{c_e} n$ over t. The complexity and error probability at this step are no more than those incurred in the sum-check protocol.

By a union bound over the $\log^e n$ stages, the probability of error is $\leq 1/\log^{c_e} n$.

10.5 Problems

Problem 10.1. For $k \in \mathbb{N}$ define IP$[k]$ as IP except that the verifier V only asks k questions, i.e., b is replaced by k in the acceptance condition.

(1) Prove that for any $k, a \in \mathbb{N}$ the class IP$[k]$ remains the same if the constant $1/3$ is replaced with 2^{-n^a}.

(2) Consider the variant IP'$[1]$ of IP$[1]$ where the prover's message does not depend on Q_1, i.e., the acceptance condition is $V(x, P(0), Q_1) = 1$ where $P(0)$ is a string. Prove that

IP'[1] $\subseteq$ IP[1]. Hint: Decrease the error probability enough to afford a union bound over $P(0)$.

(3) Prove that IP[1] = IP[k] for any $k \in \mathbb{N}$, $k \geq 1$. Hint: Extend (2).

(4) Prove that IP[1] $\subseteq$ PH.

Exercise 10.24. Give an alternative proof, not using Theorem 10.15 but following instead the proof of Theorem 10.6 and using Problem 6.2, that L has interactive proofs where the verifier is in Quasi-Linear-Time (but the prover may not be in P).

10.6 Notes

Interactive proofs were put forth simultaneously in [38, 142]. The latter paper also introduced zero-knowledge. The zero-knowledge protocol for 3Color is from [137].

Theorem 10.6 is from [238, 310], with the last paper proving it as stated. For the history of this famous result see [32].

Interactive proofs for functions in P with efficient verifier were first studied in [141], where essentially Theorem 10.21 appears. Our presentation of this result is based on [133], which differs from [141] in the way the uniformity of circuits is handled. In fact our treatment of uniformity is different still, and is inspired by [197] where stronger conditions are achieved. Theorem 10.16 is from [139]. [293] give protocols that are even constant-round for TiSP. [337] gives alternative arguments (not achieving constant-round) based on matrix powering. Simpler constant-round protocols for smaller classes are in [140]. For a survey of some of these works, see [133].

For more on interactive proofs and zero knowledge see the book [337].

Chapter 11

Pseudorandomness

Pseudorandomness is the far-reaching theory of mathematical objects that "look random" besides being computed using little or no randomness. This fascinating theory has deep ramifications throughout complexity theory. In fact, the Grand Challenge can be seen as a question in pseudorandomness: We know that a random function is hard to compute (Theorem 2.9), and we aim to construct one deterministically. Additional applications are for example in section 5.4.2, Chapter 12, and Chapter 17.

As mentioned in section 0.1.2, we will mostly work with the behavioristic definition of randomness, which we now make precise.

Definition 11.1. Let D and E be distributions over $[2]^n$. A function $f : [2]^n \to [2]$ *ϵ-distinguishes* D and E if

$$|\mathbb{E}[f(D)] - \mathbb{E}[f(E)]| \geq \epsilon.$$

If E is omitted it is assumed to be the uniform distribution, and we say that f *ϵ-breaks* D.

We say that D is *ϵ-pseudorandom* for (or *ϵ-fools*) a set of functions F if no $f \in F$ ϵ-distinguishes D.

We are interested in distributions that are pseudorandom yet are supported on few strings. As in section §1.8, counting arguments show that very little support size is needed, if we only care about *non-explicit* distributions. Specifically, the support size can be about logarithmic in the number of tests to be fooled. We are mostly interested in *explicit* distributions. One motivation is that such distributions imply P = BPP (see Claim 11.4). But more generally constructing such distributions is closely related to the grand challenge. Indeed, we know that a random function is hard to compute by circuits (Theorem 2.9). Finding explicit hard functions can be seen as reducing the support size. We now make this connection precise.

Claim 11.2. Let R be a distribution over $[2]^n$ with support of size $< 2^n/2$. Suppose that R $1/2$-fools a set F of functions. Then the indicator function $g : [2]^n \to [2]$ of the support of R is not in F.

Proof. We have $\mathbb{E}[g(U)] < 1/2$ while $\mathbb{E}[g(R)] = 1$, so g is not 1/2-fooled and cannot be in F. **QED**

For example, if $F = \text{CktP}$ and R can be sampled in P then $g \in \text{NP}$, and so $\text{NP} \not\subseteq \text{CktP}$. The above claim can be strengthened. In general, constructing such distributions can be thought of as a refined impossibility result that is closely related to correlation, recall Definition 8.11.

To simplify the following discussion, we introduce the notion of a pseudorandom generator which makes it easier to talk about the support size and its explicitness.

Definition 11.3. A function $f \in \text{FP}$ is a *pseudorandom generator*, abbreviated PRG, with error $\epsilon(n)$ and seed length $s(n, \epsilon)$ for sets T_n of functions if on input 1^n, ϵ, and a uniform $x \in [2]^{s(n,\epsilon)}$ f outputs a distribution D on n bits that ϵ-fools any function $t : [2]^n \to [2]$ in T_n. The *stretch* of f is $n - s(n, \epsilon)$.

Note that we use n to denote the *output length* of the PRG, because it is the *input length* for a test that's trying to distinguish the output of the PRG from uniform.

Claim 11.4. Suppose there is $a \in \mathbb{N}$ and a PRG with seed length $s(n) = a \log n$ and error 0.1 for circuits of size n. Prove that P = BPP.

Proof. Let $f \in \text{BPP}$. On input $x \in [2]^n$, consider the circuit C_x of size n^{c_f} s.t. $f(x) = 1$ if $\mathbb{P}_{r \in [2]^{n^{c_f}}}[C_x(r) = 1] \geq 2/3$ and $f(x) = 0$ if $\mathbb{P}_{r \in [2]^{n^{c_f}}}[C_x(r) = 1] \leq 1/3$. Use the generator with output length n^{c_f} to approximate to within 0.1 the acceptance probability of C_x, which suffices to compute $f(x)$. To compute the approximation, run C_x on every possible output of the generator, and take the average. (Alternatively, computing majority suffices.) The seed length of the generator is

$$a \log n^{c_f} \leq c_f \log n.$$

Hence we can go through all the seeds in power time n^{c_f}. Because the PRG is computable in power time in n, we have $f \in \text{P}$. **QED**

Note that we set the size of the circuit test equal to the output length n of the PRG. Recall from Definition 2.1 that input gates are not counted towards size, so the circuit may simply ignore most of its input bits, which makes sense since the information in very few input bits suffices (non-explicitly) to distinguish the output of the PRG from uniform.

Given Claim 11.2, there are two main avenues for research, closely paralleling the development of earlier chapters. The first is proving unconditional results for restricted models, like polynomials, NC^0 circuits, AC^0circuits, etc. Here even very simple models like NC^0 circuits give rise to a rich and useful theory. The second avenue is proving reductions, that is linking the existence of PRGs to other conjectures. Some basic techniques apply to both settings.

11.1 Basic PRGs

In this section we present PRGs for several basic classes of tests. These PRGs have a surprising range of applications, of which we give a few examples.

11.1.1 Local tests

The simplest model to consider is perhaps that of local functions (Definition 9.22). In this case we can obtain even error zero with a short seed length. Equivalently, any k output bits of the generator are uniform.

Theorem 11.5. There is a PRG for k-local functions with error zero and seed length $s = ck \log n$, for any function $k = k(n)$.

Exercise 11.6. Show that for $k = 1$ we can in fact have seed length $s = 1$.

Even for $k = 2$ one needs larger seed and a more complicated argument. In fact, we have already seen this in Problem 5.1. The case of larger k is a natural generalization.

Proof. W.l.o.g. assume that n is a power of 2 and let $\mathbb{F}$ be the field of size n. The range of the PRG will in fact be a tuple of n field elements, and the distribution of any k coordinates will be uniform over $\mathbb{F}^k$. To construct the PRG, view the input as coefficients $a = (a_0, a_1, \ldots, a_{k-1})$ of a polynomial p_a of degree $k - 1$. Then, for $i \in \mathbb{F}$ define the i output element of the PRG to be

$$\mathrm{PRG}(1^n, a)_i := p_a(i) = \sum_{j \in [k]} a_j i^j.$$

Now consider any k output coordinates. We claim that the distribution on these coordinates is uniform. This is because if two different polynomials p_a and $p_{a'}$ take the same values on these coordinates, then their difference $p_a - p_{a'}$ is a non-zero polynomial of degree $k - 1$ with $\geq k$ roots, violating Fact A.50. (We can assume $k \leq n$ for else the theorem is trivial.) Since the number $|\mathbb{F}|^k$ of polynomials equals the number of possible values for the k output coordinates, those coordinates are uniform. **QED**

The output distribution of a PRG as in Theorem 11.5 is known as *k-wise uniform.*

Definition 11.7. A distribution over $[q]^n$ is *k-wise uniform* if every k coordinates are uniform in $[q]^k$.

Such distributions are well-studied. It is known that powerlog-wise uniformity suffices to fool AC^0 circuits.

Theorem 11.8. A $\log(m/\epsilon)^{cd}$-wise uniform distribution over $[2]^n$ is ϵ-pseudorandom for AC^0 circuits of size m and depth d.

Combining Theorem 11.8 with Theorem 11.5 we obtain PRGs for AC^0 with power-log seed length. We give an alternative construction of such generators below (Corollary 11.35).

The seed length in Theorem 11.5 can be reduced dramatically if we tolerate some error, see Theorem 11.16 below. But for error zero the bound is tight up to constants. We prove this for even k. If k is odd then obviously the bound for $k-1$ applies.

Theorem 11.9. For even k, the seed length of a PRG for k-local functions is $\geq \log_2 \binom{n}{k/2} \geq \frac{1}{2}k\log(2n/k)$.

Proof. Linear algebra magic. Think of the output of the PRG as a string in $\{-1,1\}^n$, and write down the $2^s \times n$ matrix where row x is the output of the PRG on seed x. For any $T \subseteq [n]$ of size $k/2$, consider the vector $v_T \in \{-1,1\}^{2^s}$ obtained by multiplying together the columns indexed in T, coordinate-wise. The v_T are orthogonal. Hence they are independent (Fact A.38), and so $2^s \geq \binom{n}{k/2}$, whence $s \geq (k/2)\log(2n/k)$ (Fact A.7). **QED**

Exercise 11.10. Prove that the v_T are orthogonal.

11.1.2 Low-degree polynomials

Another natural model is that of low-degree polynomials.

Theorem 11.11. There is a PRG for degree-1 polynomials in n variables over $\mathbb{F}_2$ with error ϵ and seed length $s = c\log(n/\epsilon)$.

The seed length is known to be optimal up to constant factors. For a constant-factor improvement see Problem 11.2.

The output distribution of a PRG for degree-1 polynomials is also called ϵ-*bias*, or *small-bias*.

Definition 11.12. A distribution over $[2]^n$ is ϵ-biased if it fools linear polynomials with error ϵ.

Such distributions are in fact closely related to error-correcting codes Problem 3.6.

Exercise 11.13. Let S be a subset of $[2]^k$ s.t. the uniform distribution over S is ϵ-biased. Let M_S be the $|S| \times k$ matrix M_S where the rows are the elements of S. Prove that $M_S x$ and $M_S y$ differ in $\geq (1/2-\epsilon)|S|$ coordinates if $x \neq y$. (In other words, $\{M_S x : x \in [2]^k\}$ is a $(|S|, k, (1/2-\epsilon)|S|)$-code, cf Problem 3.6.)

Proof of Theorem 11.11.. Let $q := 2^{\log n/\epsilon}$ and identify the field $\mathbb{F}_q$ with bit strings of length $\log q$. We view a seed as a pair $(s,t) \in {\mathbb{F}_q}^2$. Then for $i \in [n]$ define the i output bit of the PRG to be

$$\mathrm{PRG}(1^n, s, t)_i := \langle s^i, t\rangle$$

where s^i is exponentiation in $\mathbb{F}_q$ and $\langle .,. \rangle$ is inner product: $\langle u, v\rangle := \sum u_i \cdot v_i$ over $\mathbb{F}_2$.

Consider any non-zero $x \in [2]^n$. We have to show that the inner product between x and the output of the PRG is nearly unbiased. The critical step is to note that that for every s, t, x:

$$\sum_i \langle s^i, t\rangle x_i = \sum_i \langle x_i \cdot s^i, t\rangle = \langle \sum_i x_i \cdot s^i, t\rangle.$$

Now, because $x \neq 0$, the probability over s that $\sum_i x_i \cdot s^i = 0$ is $\leq n/q \leq \epsilon$. This is because any s that gives a zero is a root of the non-zero, univariate polynomial $q(y) := \sum_i x_i \cdot y^i$ of degree $\leq n$ over $\mathbb{F}_q$, and by Fact A.50.

Whenever $\sum_i x_i \cdot s^i \neq 0$, the probability over t that $\langle \sum_i x_i \cdot s^i, t\rangle = 0$ is 1/2. **QED**

To fool polynomials of degree $d > 1$, we can take the xor of d independent copies of generators for degree 1.

Theorem 11.14. The bit-wise xor of d independent ϵ-biased distributions fools degree-d polynomials with error $\leq c\epsilon^{1/2^{d-1}}$.

This gives non-trivial PRGs up to degree $d < c \log n$. It is unknown what happens for large degree.

Open question 11.15. Does the bit-wise xor of d independent small-bias distributions fool degree-d polynomials for $d > \log n$?

Because of Theorem 9.8, a positive answer for power-log degrees would imply a PRG for AC^0-Mod-2.

11.1.3 Local tests, II

In section 11.1.1 we gave PRGs for k-local functions (with error zero). The seed length was always $\geq \log n$, as soon as $k \geq 2$. Small-bias distributions can also be used (see Problem 11.3) but again give seed length $\geq \log n$. The next result gives PRGs which much shorter seed length: the dependency on n becomes doubly-logarithmic.

Theorem 11.16. There is a PRG for k-local functions with error ϵ and seed length $\leq c(k + \log 1/\epsilon + \log\log n)$.

The PRG is an instructive combination of the PRGs for k-local functions and degree-1 polynomials, and the hypercube analysis Fact A.39.

Proof. For slight convenience we think of the output of the PRG in $\{-1, 1\}^n$. Consider a k-local function f on n bits. W.l.o.g. suppose f depends on bits $x_0, x_1, \ldots, x_{k-1}$. According to Fact A.39, we can write f as multi-linear real polynomial

$$f(x_0, x_1, \ldots, x_{k-1}) = \sum_{S \subseteq [k]} a_S x^S$$

where $a_S \in \mathbb{R}$, and x^S denotes $\prod_{i \in S} x_i$.

We can first apply a PRG for k-local functions from Theorem 11.5, which has seed length $s \leq ck \log n$. (The parameter k can be computed from the seed length, ϵ, and n, all of which are given.) Selecting a field of size a power of 2 in the definition of that PRG, a critical observation is that each output bit of this PRG is a linear map over $\mathbb{F}_2$ in the seed bits. Viewing the element of $\mathbb{F}_2$ as $\{-1, 1\}$, this means that each output value x_i is the product of some subset of the seed bits y_j. In turn, x^S corresponds to $y^{g(S)}$ for some remapping g of the sets. Because the generator fools with error 0, the expectation of f over uniform equals the expectation over the output of the PRG, which is

$$\sum_{S \subseteq [k]} a_S y^{g(S)}.$$

Note that the right-hand side is a polynomial that may not have small degree, yet is *sparse*: The number of terms is $\leq 2^k$. Further note that if $S \neq \emptyset$ then also $g(S) \neq \emptyset$ (for else the parity of the bits in S would always be zero, invalidating the PRG).

Now let us further replace the y with a PRG for degree-1 polynomials with error $\epsilon/2^{k/2}$. The gain is that this PRG only needs to output $ck \log n$ bits instead of n. By Theorem 11.11 the final seed length is then

$$c \log \left((ck \log n) \cdot \frac{2^k}{\epsilon} \right) = c(k + \log 1/\epsilon + \log \log n)$$

as desired.

For the analysis, note that

$$\left| \mathbb{E}[f(U)] - \mathbb{E}\left[\sum_{S \subseteq [k]} a_S y^{g(S)} \right] \right| = \left| \sum_{S \subseteq [k], S \neq \emptyset} a_S \mathbb{E}[y^{g(S)}] \right| \leq \sum_{S \neq \emptyset} |a_S| \left| \mathbb{E}[y^{g(S)}] \right|.$$

As we remarked, $g(S) \neq \emptyset$. So by the property of the PRG for degree-1 polynomials, each expectation is $\leq c\epsilon/2^{k/2}$ in absolute value, so the final error is $\leq (c\epsilon/2^{k/2}) \cdot \sum_{S \subseteq [k]} |a_S| \leq c\epsilon$ (Fact A.39). **QED**

11.1.4 Local small bias

A distribution over $[2]^n$ is (ϵ, k)-biased if it ϵ-biased (Definition 11.12) on any k coordinates. This captures all the distributions constructed in the previous section. Obviously, ϵ-bias is the same as (ϵ, n)-bias. Theorem 11.16 really constructs an $(\epsilon/2^k, k)$-biased distribution, and then infers almost local uniformity. We can get all the way to k-wise uniformity using the following fact:

Fact 11.17. An (ϵ, k)-biased distribution over $[2]^n$ has distance $\leq (cn/k)^{k/2}\epsilon$ from a k-wise uniform distribution.

Distributions which are (ϵ, k)-biased are extensively studied, especially w.r.t. their ability to fool various classes of tests, such as tail bounds, see the notes.

11.2 PH is a random low-degree polynomial

In this section we use small-bias generators (Theorem 11.11) to prove a uniform, "scaled up version" of Theorem 9.8. First, let us define the relevant class. We will only be concerned with power times so we can afford a definition that is a bit simpler than those we saw in Chapter 5.

Definition 11.18. We denote by BP $\oplus$ P ("BP parity P") the class of functions $f : X \subseteq [2]^* \to [2]$ for which there is $g \in$ P a constant d such that for every $x \in X$:

- $f(x) = 1 \Rightarrow \mathbb{P}_{y_1 \in [2]^{n^d}} \left[\oplus y_2 \in [2]^{n^d} : g(x, y_1, y_2) = 1 \right] \geq 2/3$, and
- $f(x) = 0 \Rightarrow \mathbb{P}_{y_1 \in [2]^{n^d}} \left[\oplus y_2 \in [2]^{n^d} : g(x, y_1, y_2) = 1 \right] \leq 1/3$.

Theorem 11.19. PH $\subseteq$ BP $\oplus$ P.

This is saying that any constant number of $\exists$ and $\forall$ quantifier can be replaced by a BP quantifier followed by a $\oplus$ quantifier. What does this have to do with the simulation in Theorem 9.8 of AC^0 by polynomials? The BP quantifier samples a polynomials (with an exponential number of monomials), and the $\oplus$ quantifier picks a monomial in the polynomial, which can be computed in P since it has low degree.

Proof. Let $f \in \Sigma_d$P. Consider the AC^0 circuit C on 2^{dn^d} bits with depth d, where each gate has fan-in 2^{n^d}. We need to evaluate this circuit on the input consisting of the bits

$$M(x, y_1, y_2, \ldots, y_d)$$

for all values of the quantified variables y_i.

We claim that using a seed of n^{cd} bits r we can sample a polynomial P_r of degree n^{cd} which computes the circuit in the sense of Definition 9.7 with error $1/3$, and moreover given r and an index i of n^{cd} bits we can compute monomial i of P_r in time n^{cd}. Given this claim, the proof is completed as follows. We use the BP quantifier to select r, and then the $\oplus$ quantifier to pick an index to a monomial. We can compute this monomial in power time by evaluating $M(x, y_1, y_2, \ldots, y_d)$ at the corresponding inputs.

There remains to prove the claim. We follow the construction in Theorem 9.8. Recall that the basic building block is to compute Or on an input z via the construction

$$p_{A_1, A_2, \ldots, A_t}(z) := p_{A_1}(z) \vee p_{A_2}(z) \vee \cdots \vee p_{A_t}(z)$$

where recall $p_A(z_1, z_2, \ldots, z_n) := \sum_i A_i \cdot z_i$. Here each A_i is generated via the small-bias PRG (Theorem 11.11). Using the polynomial for Or from Example 9.6 we can write

$$p_{A_1, A_2, \ldots, A_t}(z) = \sum_{a \in [2]^t : a \neq 0} \prod_{i \leq t} (p_{A_i}(z) + a_i + 1).$$

From this we can observe that to index a monomial in p we can select a choice of a and then a choice for a monomial in each of the t linear factors $p_{A_i}(z) + a_i + 1$. For each factor, inspection of the small-bias generator (Theorem 11.11) reveals that we can index its monomials.

A similar construction exists for And.

Note each of these polynomials is on $2^{n^{c_d}}$ variables and we can set $t := n^{c_d}$ so that it has degree n^{c_d} and error $2^{-n^{c_d}}$, less than c times the total number of gates in the circuit. The seed length necessary to sample it will also be $\leq n^{c_d}$.

As in Theorem 9.8, the final polynomial is the composition of these polynomials at each gate. Critically, the seed used to sample the polynomials is re-used for each gate. We can afford this because we use a union bound in the analysis. Hence the seed length for the final composed polynomial is again just n^{c_d}.

Finally, from an index we can compute the corresponding monomial of the composed polynomial in the natural way. Some details: Start at the output gate. We use n^{c_d} bits in the index to choose a monomial in the corresponding polynomial. We write down this monomial, as discussed above. This monomial is over $2^{n^{c_d}}$ variables $z_1, z_2, \ldots$ corresponding to the children of the output gate, and as remarked has degree $\leq n^{c_d}$. To each z_i there corresponds another polynomial p_i. Choose a monomial from p_i and replace z_i with that monomial; do this for every i. The choice of the monomials is done again using bits from the index. Because each monomial is over n^{c_d} variables, and the depth is constant, the total number of bits which are needed to choose monomials is n^{c_d}. **QED**

In turn, we now present a uniform, scaled-up analogue of Lemma 9.50.

Definition 11.20. We denote by SymP the class of functions $f : X \subseteq [2]^* \to [2]$ for which there are $h, g \in \mathrm{P}$ and a constant d such that for every $x \in X$ we have $f(x) = h(z)$ where z is the number of $y \in [2]^{n^d}$ s.t. $g(x, y) = 1$.

In other words, f is efficiently computable from the number of y s.t. $g(x, y) = 1$. Note that SymP includes $\Sigma_1\mathrm{P}$, $\Pi_1\mathrm{P}$, $\oplus\mathrm{P}$, MajP. On the other hand, we have SymP $\subseteq$ MajMajP. This latter inclusion can be proved along the lines of Exercise 8.6. A relatively simple proof gives three Maj quantifiers; we won't need the fine result that you can use 2.

Theorem 11.21. $\mathrm{BP} \oplus \mathrm{P} \subseteq$ SymP.

By Theorem 11.19 we get PH $\subseteq$ SymP.

Proof. Let $f \in \mathrm{BP} \oplus \mathrm{P}$, and let g and d correspond go f as in Definition 11.18. We set $\ell := n^{c_d}$. For an input x, $f(x)$ is determined by the sum

$$\sum_{y_1 \in [2]^{n^d}} \left(\sum_{y_2 \in [2]^{n^d}} g(x, y_1, y_2) \mod 2 \right) \in \mathbb{N}. \tag{11.1}$$

Using Lemma 9.51 we rewrite this as

$$\sum_{y_1 \in [2]^{n^d}} \left(F_\ell \left(\sum_{y_2 \in [2]^{n^d}} g(x, y_1, y_2) \right) \mod 2^\ell \right) = \left(\sum_{y_1 \in [2]^{n^d}} F_\ell \left(\sum_{y_2 \in [2]^{n^d}} g(x, y_1, y_2) \right) \right) \mod 2^\ell.$$

Now let $F_\ell(z) = \sum_{i \le \ell^c} a_i z^i$ (we used the weaker degree bound from Exercise 9.52). We can then rewrite the sum as

$$\sum_{y_1 \in [2]^{n^d}} \sum_{i \le \ell^c} a_i \left(\sum_{y_2 \in [2]^{n^d}} g(x, y_1, y_2) \right)^i = \sum_{y_1 \in [2]^{n^d}} \sum_{i \le \ell^c} \sum_{y_2^1, y_2^2, \ldots, y_2^i} a_i \prod_{j=1}^{i} g(x, y_1, y_2^j).$$

The inner product can be computed in P, using that the a_i can be computed in P. The latter can be verified by inspection for the construction in Exercise 9.52, which suffices for this application. Letting the y in Definition 11.20 be $y = (y_1, i, y_2^1, y_2^2, \ldots, y_2^i)$ the result follows. **QED**

11.3 Pseudorandom generators from hard functions

In this section we present a fascinating paradigm to construct PRGs from functions that are hard to compute. We begin with a general claim showing that PRGs with non-trivial seed length $s(n) = n - 1$ are in fact equivalent to correlation bounds, recall Definition 8.11.

Claim 11.22. For a function $f : [2]^\ell \to [2]$ define the distribution $D := (x, f(x))$ for uniform x.

(1) If $C : [2]^{\ell+1} \to [2]$ ϵ-breaks D then there is $b \in [2]$ s.t. $C'_b : [2]^\ell \to [2]$ defined as $C'_b(x) := C(xb) \oplus b$ has correlation $\ge \epsilon$ with f.

(2) If $C : [2]^\ell \to [2]$ has ϵ-correlation with f then $C' : [2]^{\ell+1} \to [2]$ defined as $C'(x, b) := C(x) \oplus b$ $\epsilon/2$-breaks D.

Proof of (1). Pick b uniformly and write

$$\begin{aligned}
\mathbb{E}_{x,b} e[C'_b(x) \oplus f(x)] &= \mathbb{E}_{x, b \oplus f(x)} e\left[C'_{b \oplus f(x)}(x) \oplus f(x)\right] \\
&= \mathbb{E} e\left[C_{b \oplus f(x)}(x(b \oplus f(x))) \oplus b\right] \\
&= \frac{1}{2} \left| \mathbb{E}_x C(xf(x)) - \mathbb{E}_x C(x\overline{f}(x)) \right| \\
&= \left| \mathbb{E}_x C(xf(x)) - \mathbb{E} C(U) \right| \\
&\ge \epsilon
\end{aligned}$$

Here $\bar{f}$ is the complement of f, and U is uniform. So there is b s.t. the lhs is at least ϵ, completing the proof of the first claim. **QED**

Exercise 11.23. Prove (2) in Claim 11.22.

The contrapositive of (1) is that a function $h : [2]^\ell \to [2]$ that is hard to correlate with immediately gives a PRG $G(x) := (x, h(x)) \in [2]^{\ell+1}$ with seed length $s(n) = n - 1$.

Equivalently, G has 1 bit of stretch. Naturally, we'd like to increase the stretch. A natural idea is *repetition*:

$$R(x_1, x_2, \ldots, x_k) := x_1 h(x_1) x_2 h(x_2) \cdots x_k h(x_k) \tag{11.2}$$

which has seed length $s(n) = n - k$. This repetition won't even give seed length $s(n) < n/2$, but is nevertheless a fundamental construction which arises in many settings, and serves as a warm-up for better ways to increase the stretch which are presented below in Claim 11.27 and section 11.3.1. To study repetition in a general setting, for a distribution D over $[2]^n$ denote by D^k the concatenation of k independent copies D, a distribution over over $[2]^{k \cdot n}$.

Claim 11.24. If f ϵ-distinguishes D^k and E^k then a restriction of f ϵ/k-distinguishes D and E.

The repetition PRG in equation (11.2) corresponds to $D = (x, h(x))$ and E uniform. In this case the claim is saying that if we can break D^k then we can break D, which by Claim 11.22 means that we can correlate with h. To spell out the contrapositive, if it is hard to correlate with h then D^k is a PRG with seed length $s(n) = n - k$.

Proof. Via the "hybrid method," a.k.a. the triangle inequality. Define

$$H_i := D_0 D_1 \cdots D_{i-1} E_i E_{i+1} \cdots E_{k-1}$$

over nk bits for $i \in [k]$, where each factor in the rhs is over n bits. Note that H_0 is E^k and H_k is D^k. Write

$$\begin{aligned}
\epsilon &\leq |\mathbb{E}[f(H_0)] - \mathbb{E}[f(H_{k-1})]| \\
&= \left| \sum_{i \in [k]} \mathbb{E}[f(H_i)] - \mathbb{E}[f(H_{i+1})] \right| \\
&\leq \sum_{i \in [k]} |\mathbb{E}[f(H_i)] - \mathbb{E}[f(H_{i+1})]| \,.
\end{aligned}$$

So one of the terms on the rhs is $\geq \epsilon/k$. The corresponding distributions H_i and H_{i+1} differ in only one factor. We can fix all others and the claim follows. **QED**

Note we went from ϵ-distinguishing to ϵ/k. This means the claim is only applicable when ϵ is fairly small. In general, this loss cannot be avoided:

Exercise 11.25. Exhibit a distribution D that is 0.1-pseudorandom for all functions yet D^k is not even 0.9 pseudorandom for circuits of size k, for all large enough k.

One can also obtain D of the form $xf(x)$ for some boolean function f, see Problem 11.1.

However, this loss in repetition can be avoided for *resampleable* functions. These are functions h s.t. given any "correct" pair $(x, h(x))$ we can generate uniform correct pairs $(y, h(y))$, and similarly for "incorrect" pairs $(x, h(x) \oplus 1)$, both using the same distribution.

Definition 11.26. A function $h : [2]^n \to [2]$ is *resampled* by a distribution F on functions from $[2]^{n+1}$ to $[2]^{n+1}$ if for every $x \in [2]^n$ and $b \in [2]$, $F(x, h(x) \oplus b)$ outputs $(y, h(y) \oplus b)$ for uniform $y \in [2]^n$.

Claim 11.27. Let $h : [2]^n \to [2]$ be a function that is resampled by F. Define the distribution $D = (x, h(x))$ for uniform x. Suppose f ϵ-breaks D^k. Then $f(G, G, \ldots, G)$ $\epsilon/2$-breaks D, where each occurrence of G is either an occurrence of F or a fixed value.

Proof. As in the proof of Claim 11.22 we can write the uniform distribution on $n+1$ bits as $(X, h(X) \oplus b)$ for uniform X and $b \in [2]$. Because f ϵ-breaks D, we can fix coins $b_1, \ldots, b_k$ s.t. the quantities

$$\mathbb{E}\left[f\left((X_1, h(X_1)), (X_2, h(X_2)), \ldots, (X_k, h(X_k))\right)\right]$$
$$\mathbb{E}\left[f\left((X_1, h(X_1) \oplus b_1), (X_2, h(X_2) \oplus b_2), \ldots, (X_k, h(X_k) \oplus b_k)\right)\right]$$

have distance $\geq \epsilon$.

The coordinates where $b_i = 0$ are the same. We can fix the corresponding inputs and obtain a restriction f' of f s.t. for some $j \leq k$ the quantities

$$\mathbb{E}\left[f'\left((X_1, h(X_1)), (X_2, h(X_2)), \ldots, (X_j, h(X_j))\right)\right] =: (I)$$
$$\mathbb{E}\left[f'\left((X_1, \overline{h}(X_1)), (X_2, \overline{h}(X_2)), \ldots, (X_j, \overline{h}(X_j))\right)\right] =: (II)$$

have distance $\geq \epsilon$.

Now we use this to break D. Again as in the proof of Claim 11.22 it suffices to distinguish D from $\overline{D} := (X, \overline{h}(X))$. On input $z \in [2]^{n+1}$, we compute

$$f'(F(z), F(z), \ldots, F(z))$$

where the occurrences of the resampler F are independent. If $z = D$ then the expectation of this is the same as (I), and if $z = \overline{D}$ it is the same as (II). **QED**

Claim 11.28. Parity is resampleable by a distribution F of NC^0 circuits.

Exercise 11.29. Prove this. Hint: This is similar to the random self-reducibility of parity established in section 9.2.4.

Resampleability gives PRGs with the shortest known seed length for AC^0-Mod-p circuits. For example, for AC^0-Mod-3 circuits Problem 9.2 in fact establishes a non-trivial correlation bound. By computing parities on power-log bits and analyzing repetition via resampleability we obtain a generator stretching $n - n/\log^{c_d} n$ bits to n that fools AC^0-Mod-3 circuits of depth d and sizer n. For AC^0-Mod-2 one can follow the same approach but starting from a different function. In both cases, no better stretch is not known.

Open question 11.30. Give a PRG with seed length $0.99n$ for AC^0-Mod-2 circuits of depth 3 and size n.

One candidate PRG was presented in section 11.1.2, see Question 11.15.

11.3.1 Stretching correlation bounds: The bounded-intersection generator

The repetition PRG (equation (11.2)) outputs values of a hard function on independent inputs. We now study a powerful technique which instead outputs values from *dependent* inputs, selected from a collection of subsets of a *universe* $[u]$, where u is the seed length.

Definition 11.31. Let $S = T_1, T_2, \ldots, T_s$ be a list of s subsets of $[u]$, each of size ℓ. Then the *bounded-intersection generator*

$$\mathrm{BIG}_S : [2]^u \to \left([2]^\ell\right)^s$$

is defined as

$$\mathrm{BIG}_S(x) := x_{T_1}, x_{T_2}, \ldots, x_{T_s},$$

where x_{T_j} denotes the ℓ bits of x indexed by T_j.

For a distribution H on functions from $[2]^\ell \to [2]$ we denote

$$H \circ \mathrm{BIG}_S(x) := H(x_{T_1}), H(x_{T_2}), \ldots, H(x_{T_s})$$

where the occurrences of H denote independent samples.

To illustrate, note that if H is a uniform function then $H \circ \mathrm{BIG}_S$ is uniform over $[2]^s$ regardless of the sets S.

The next key result shows that BIG *preserves indistinguishability*, similar to the repetition generator, as long as the sets in S have small intersections. This is true even if the seed is revealed. The intersection size governs the locality (recall Definition 9.22), and hence the complexity, of the reduction.

Theorem 11.32 (BIG PI)**.** Let BIG and $S = T_1, T_2, \ldots$ be as in Definition 11.31. Further suppose that $|T_i \cap T_j| \leq d$ for any $i \neq j$. Let V and W be two distributions on functions from $[2]^\ell$ to $[2]$.

Suppose f ϵ-distinguishes the distributions $(X, V \circ \mathrm{BIG}_S(X))$ and $(X, W \circ \mathrm{BIG}_S(X))$, over $u + s$ bits, where X is uniform in $[2]^u$.

Then there are d-local functions g_i s.t. $f(g_1, g_2, \ldots, g_{u+s})$ distinguishes $(X, V(X))$ from $(X, W(X))$ with advantage $\geq \epsilon/s$, where X is uniform in $[2]^\ell$.

As we discussed for repetition (Claim 11.24) a basic setting is where V is a fixed hard function h and W is a uniform function. In this case, Theorem 11.32 in conjunction with Claim 11.22 says that if it is hard to correlate with h then $(X, h \circ \mathrm{BIG}_S(X))$ is a PRG. The seed length now is u which can be much smaller than, even logarithmic in, the output length $u + s$. One can think of the output length as being essentially s, but in fact we achieve a little more, $n = u + s$.

Exercise 11.33. Derive Claim 11.24 from Theorem 11.32 for the special case $D = (X, V(X))$ and $E = (X, W(X))$.

Proof. Similarly to the proof of Claim 11.24, define hybrids

$$H_i := X, V(x_{T_1}), V(x_{T_2}), \ldots, V(x_{T_{i-1}}), W(x_{T_i}), W(x_{T_i}), \ldots, W(x_{T_s})$$

over $u + s$ bits, for $i \in [1..s+1]$ indicating the index of first set where W is evaluated. Note that H_1 is $(X, W \circ \mathrm{BIG}_S(X))$ and H_{s+1} is $(X, V \circ \mathrm{BIG}_S(X))$. So there is $i \in [1..s]$ s.t. f distinguishes two adjacent hybrids H_i and H_{i+1} with advantage $\geq \epsilon/s$. These hybrid differ only in the evaluation on the set T_i. We can then fix the $u - \ell$ bits in the seed X that are not in set T_i while maintaining the distinguishing advantage. Now every bit in the hybrid, except the one corresponding to set T_i, depends on $\leq d$ of the bits in T_i (this includes the bits in the seed X, which are either fixed or the bits in T_i). Hence this bit can be computed by a distribution G_j on d-local functions (corresponding to a restriction of either V or W).

We can then distinguish $(X, V(X))$ from $(X, W(X))$ with advantage $\geq \epsilon/s$ using the following distribution on functions: On input $(x, b) \in [2]^{\ell+1}$ output

$$f\left(G_1(x), G_2(x), \ldots, G_u(x), G_{u+1}(x), \ldots, G_{u+i-1}(x), b, G_{u+i+1}(x), G_{u+i+2}(x), \ldots, G_{u+s}(x)\right).$$

Finally, we can fix the distributions G_i to fixed functions g_i while maintaining the advantage, concluding the proof. **QED**

To apply Theorem 11.32 we'd like to have as many sets T_i as possible (that's the output length of the generator) which are as large as possible (that's the input length to the hard function) while being subsets of as small a universe set $[u]$ as possible (that's the seed length) and such that any two have as small intersection as possible (that's the overhead in the reduction). The probabilistic method shows that we can achieve great parameters, in particular sufficient for applications such as P = BPP. For starters, the following is a simple explicit construction.

Lemma 11.34. [Sets with small intersections] There are collections of q^d subsets of $[q^2]$ of size q such that any two sets intersect in $\leq d$ elements, for any q that is a power of 2 and $> d$. Given 1^q and d we can compute set i in FP.

Proof. View the universe $[u] = [q^2]$ as $\mathbb{F}_q^2$. For a parameter d, the sets correspond to the graphs of polynomials p of degree $< d$. (I.e., the set $\{(x, p(x)) : x \in \mathbb{F}_q\}$.) The number of sets is q^d. The size of each set is $q = \sqrt{u}$. To bound the intersection of two sets, consider the corresponding polynomials and let p be their difference, which is not zero. Any element in the intersection of the sets corresponds to a zero of p. By Fact A.50, the intersection has size $\leq d$.

To compute a set we need to construct a representation of the field $\mathbb{F}_q$ and perform field operations. This can be done within the stated resources (Fact A.31). In fact, we can even pick very explicit field representations, see Example A.32, dispensing with the construction of $\mathbb{F}_q$ altogether under minor conditions on q. **QED**

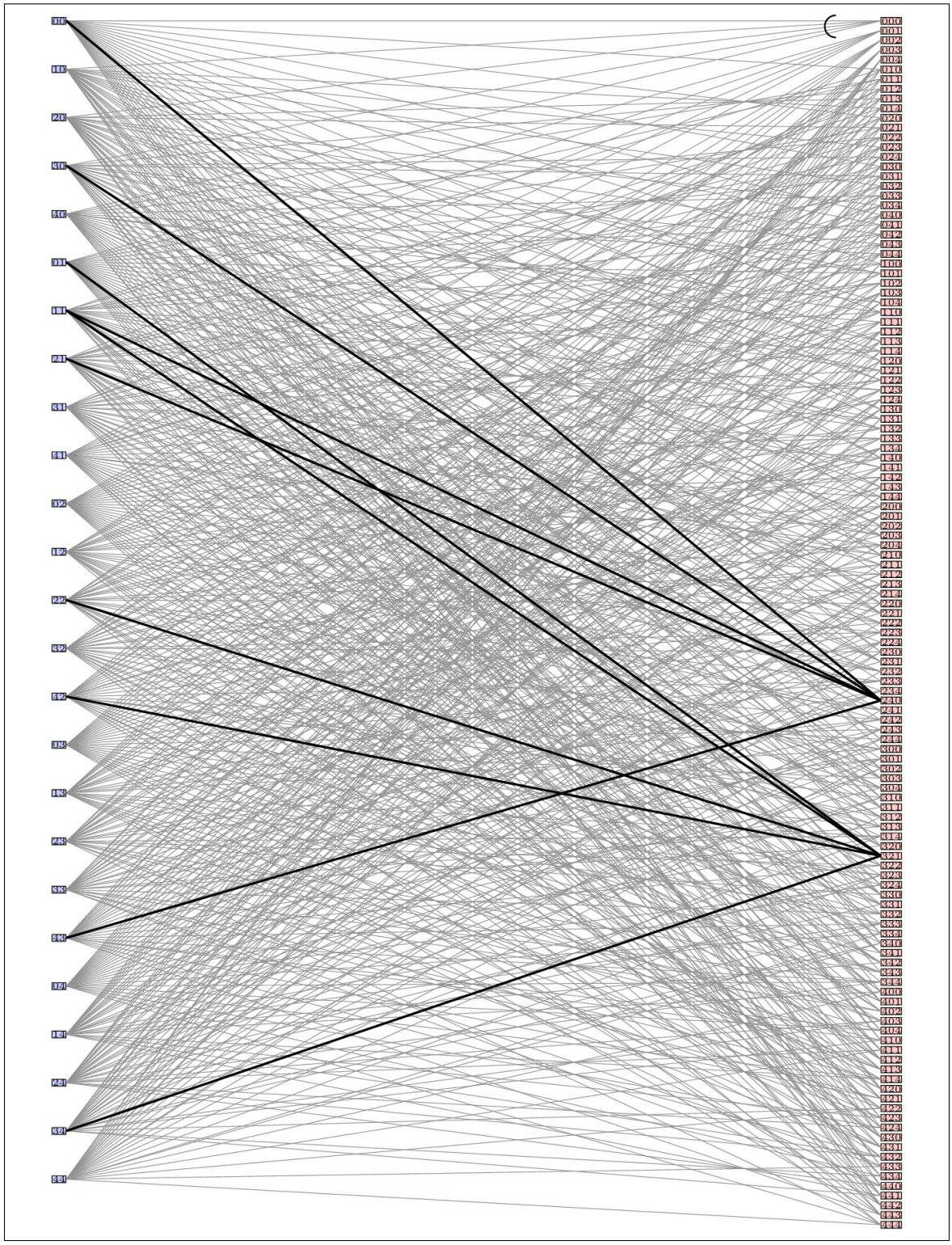


Figure 11.1: BIG Definition 11.31 using Lemma 11.34 for $q := 5$, $d := 3$. We have $s = 5^3$ choices for the polynomial coefficients in the rhs, and a universe of size $u = 5^2$ on the lhs. The degree of each node on the right, marked with a curved line, is $\ell = 5$. The two highlighted sets intersect in 1 element.

figure 11.1 shows BIG (Definition 11.31) using Lemma 11.34. To further illustrate parameters, Lemma 11.34 allows for $s = q^d$ subsets of size $\ell = q$ from a universe of size $u = \ell^2 = q$ with intersections at most d. For example, given a desired output length n we can set $d = \log n$ and $q = \log^a n$. This way the intersection size is $\ell^{1/a}$ while the universe size is $u = \ell^2$.

Corollary 11.35. There is a pseudorandom generator with seed length $\log^{c_d} n$ and error $1/n$ for AC^0 circuits of size n and depth d.

Proof. This follows by combining Corollary 9.20, Theorem 11.32, and Lemma 11.34. Specifically, we begin with the parity function on $\ell := \log^{c_d} n$ which by Corollary 9.20 has correlation $\leq 1/n^2$ with AC^0 circuits of size n^c and depth $d + c$. We then apply the BIG Theorem 11.32 using the sets from Lemma 11.34, setting the intersection size to be $\log n = \ell^{1/c_f}$. If a circuit C of size n $1/n$-breaks the PRG, then Theorem 11.32 gives a new circuit C' with correlation $> 1/n^2$ with parity on ℓ bits. The new circuit computes the functions on the intersections in brute-force. This takes depth c and size n^c. So C' has size n^c and depth $d + c$, contradicting Corollary 9.20 for our parameter choice. **QED**

The seed length in Corollary 11.35 is about the best we can do given current impossibility results, and recall once again from section §6.3 that stronger impossibility would imply major separations.

Still, one can ask if PRGs *could* be built *if* we had stronger correlation bounds. In particular, one would like to have seed length say $c \log n$ instead of $\log^c n$. This is the setting that allows for conclusions such as P = BPP, see Claim 11.4. The construction in Lemma 11.34 doesn't give this, since the universe is always at least quadratic in the set size, but the following construction does.

Lemma 11.36. [Sets with small intersections, II] For any a and $n \geq c_a$ there is a collection of n subsets of $[c_a \log n]$ of size $a \log n$ such that any two sets intersect in $\leq \log n$ elements, and the collection can be computed in time n^{c_a}.

The proof is a prototypical example of how PRGs (specifically pairwise uniformity) can be used to derandomize probabilistic arguments and turn them into explicit constructions.

Proof. Let $\ell := \log n$. First we show existence via a probabilistic argument. Then we derandomize it. View the universe as $a\ell$ blocks of $b := \lceil 4ea \rceil$ elements each. Let us choose the sets $T_1, T_2, \ldots, T_n$ uniformly at random with exactly one element in each block. Notice the size of each set is $a\ell$, as required.

For every $i \neq j$, by a union bound and Fact A.7,

$$\mathbb{P}\left[|T_i \cap T_j| \geq \ell\right] \leq \binom{a\ell}{\ell} \left(\frac{1}{b}\right)^\ell \leq \left(\frac{ea}{b}\right)^\ell \leq \left(\frac{1}{4}\right)^\ell = \frac{1}{n^2}.$$

So, by another union bound, the probability that there are $i \neq j$ such that $|T_i \cap T_j| \geq \ell$ is

$$\leq \frac{\binom{n}{2}}{n^2} < 1.$$

Therefore such sets exist.

This proof goes through even if the sets T_i are just pairwise uniform. Such a distribution can be generated using a seed of $c_a \log n$ bits, see Problem 5.1 or the proof of Theorem 11.5. Hence we can try every possible seed. For each seed we compute the sets and check in time n^c if they satisfy the intersection property. We output the first collection that does. **QED**

Using this we can prove the following weaker version of Theorem 3.15.

Corollary 11.37. Suppose there is $\epsilon > 0$ and $f \in \mathrm{E}$ such that f_n has correlation $\leq 2^{-\epsilon n}$ with circuits of size $2^{\epsilon n}$. Then $\mathrm{P} = \mathrm{BPP}$.

Exercise 11.38. Prove Corollary 11.37 assuming Lemma 11.36.

11.3.2 Turning hardness into correlation bounds: XOR lemmas and hard-core sets

> *We remark that none of the known hardness amplification results can be applied to the computational models for which we actually can establish the existence of hard functions (i.e. prove lower bounds).*

In section §8.4 we proved that impossibility results are equivalent to correlation bounds *under some distribution.* For PRGs, we need correlation under the uniform distribution (Claim 11.22). We can't expect to prove a similar equivalence in this case, as one can construct functions which are easy to compute on 99% of the inputs, but are nevertheless hard to compute on 100% of the inputs. So instead in this section we show how starting with a function $f : [2]^n \to [2]$ that has correlation $\leq \epsilon$ with small circuits we can construct a related function $f' : [2]^{n'} \to [2]$ which has much smaller correlation $\epsilon' \ll \epsilon$ with slightly smaller circuits. A natural candidate is

$$f'(x_1, \ldots, x_k) := \oplus_{i=1}^{k} f(x_i).$$

An *XOR Lemma* establishes that the correlation of f' decays *exponentially fast* with the number k of copies, i.e., it is at most about ϵ^k.

There is a strong *informational* (as opposed to computational) intuition why the XOR Lemma should work. To illustrate, consider the "computational model" of constant functions 0 or 1. The claim that f has correlation at most ϵ with these functions then simply means that for every constant function $g(x) = 0$ or $g(x) = 1$ we have

$$|\mathbb{E}ef(x) + g(x)| = |\mathbb{E}ef(x)| \leq \epsilon.$$

And indeed in this case the correlation decays exponentially fast:

$$|\mathbb{E}ef'(x') + g(x')| = |\mathbb{E}ef'(x')| = |\mathbb{E}ef(x)|^k \leq \epsilon^k.$$

More generally, we can claim that if f has correlation $\leq \epsilon$ with small circuits C, then f' indeed has correlation $\leq \epsilon^k$ with small circuits *of the special product form* $C'(x_1, \ldots, x_k) := \oplus_{i=1}^k C_i(x_i)$. This is again because

$$|\mathbb{E} e f'(x') + C'(x_1, \ldots, x_k)| = \left|\mathbb{E} e \oplus_{i=1}^k (f(x_i) \oplus C_i(x_i))\right| = \prod_i |\mathbb{E} e f(x) \oplus C_i(x)| \leq \epsilon^k.$$

This generalizes the example of constant functions since they are trivially in the special product form. Intuitively, no circuit can do better than a circuit in the special form, and the XOR Lemma is true. *But is the intuition true?*

Remark 11.39. The XOR lemma is false for Maj-AC^0 (Definition 9.15). Consider the parity function, the result in Theorem 9.17 actually bounds the correlation between this class and parity, say by 1/2. However, the XOR lemma applied to parity simply yields parity again, and the correlation between parity and Maj-AC^0, or even just a single majority gate, does *not* decay exponentially: It remains $\geq 1/n^c$.

Exercise 11.40. Prove the last claim. Hint: Use Fact A.6.

Open question 11.41. Does the XOR lemma hold for AC^0-Mod-2, or even constant-degree polynomials over $\mathbb{F}_2$?

If it does hold, we could construct functions with small correlation and (using BIG) obtain PRGs for AC^0-Mod-2, answering Question 9.13 and Question 11.30.

But for (general) circuits, we can indeed prove the xor lemma. Several proofs exist (see the notes). We present in the next section a proof that closely matches the information-theoretic intuition above, and which is amenable to useful extensions. Before that, however, we emphasize an important point:

The XOR lemma is only known to hold for computational models which can compute majority. Specifically, to prove correlation $\leq \epsilon$ the reduction proof needs the circuit to compute majorities on $\geq \log 1/\epsilon$ bits. This is apparent in section §11.4 and Problem 11.6. So to apply the XOR lemma, we have to start from an impossibility result for circuits that can compute majority. As discussed in Chapter 7, we essentially have no such result. So the results in the next section are mostly conditional. Still, they allow us to spin a fascinating web of reductions between correlation and randomness, pointing to several challenges.

11.3.2.1 Simple XOR lemma

I present here a simple proof of an XOR lemma which may provide a simple explanation of why xor-ing decreases correlation. We show that if a function f has correlation $\leq \alpha$ with small circuits than the XOR of two copies has correlation $\leq \epsilon \ll \alpha$ with the XOR of two

copies of f. We state and prove the contrapositive, that a circuit achieving correlation $\geq \epsilon$ with two copies of f can be transformed into a circuit achieving larger correalation α with f. We use range $\{-1,1\}$ and multiplicative notation (cf Definition 8.11).

Theorem 11.42. Let $f : [2]^n \to \{-1,1\}$ be a function. Suppose there is a circuit $C : [2]^n \times [2]^n \to \{-1,1\}$ of size s s.t. $\mathbb{E}_{x,y\in[2]^n} C(x,y)f(x)f(y) \geq \epsilon$. Then there is a circuit C' of size $\leq 3s+5$ s.t. $\mathbb{E}_{x\in[2]^n} C'(x)f(x) \geq \alpha$ where α is the solution of $3\epsilon - \alpha^3 = 2\alpha$.

One can verify that $\alpha \geq 1 - 0.8(1-\epsilon)$ for $\epsilon \geq 1/2$, and $\alpha \geq 1.2\epsilon$ for $\epsilon \leq 1/2$.

Proof. Let $C'(x) := \text{Maj}\left(C(x,y_1)f(y_1), C(x,y_2)f(y_2), C(x,y_3)f(y_3)\right)$ for uniform $y_1, y_2, y_3 \in [2]^n$. Because $\text{Maj}_3(x_1,x_2,x_3) = \frac{1}{2}(\sum_i x_i - \prod_i x_i)$ as a real polynomial from $\{-1,1\}^3$ to $\{-1,1\}$, we have

$$\begin{aligned}
\mathbb{E}_{x,y_1,y_2,y_3} C'(x)f(x) &= \mathbb{E}_{x,y_1,y_2,y_3} \frac{1}{2}\Big(\sum_i C(x,y_i)f(y_i)f(x) - f(x)\prod_i C(x,y_i)f(y_i)\Big) \\
&= \frac{3}{2}\mathbb{E}_{x,y\in[2]^n} C(x,y)f(x)f(y) - \frac{1}{2}\mathbb{E}_x\left[f(x)\mathbb{E}_y^3 C(x,y)f(y)\right]. \\
&\geq \frac{3\epsilon}{2} - \frac{1}{2}\max_x \left|\mathbb{E}_y^3 C(x,y)f(y)\right|.
\end{aligned}$$

If there is x s.t. $|\mathbb{E}_y C(x,y)f(y)| \geq \alpha$ the conclusion holds with $C'(y) := C(x,y)$. Otherwise, the last displayed expression is $\geq 3\epsilon/2 - \alpha^3/2 = \alpha$. We can fix the y_i to maintain the inequality. **QED**

By taking larger majorities, one can extend this proof to show that $\epsilon \leq \alpha^{1+c}$, and repeating this argument one can indeed prove that the correlation shrinks exponentially fast with the number of copies.

Next we develop the theory of hard-core sets and use it to give another proof of the XOR lemma which is more flexible for applications, such as obtaining P = BPP under assumptions, and hardness amplification within NP.

11.3.2.2 Hard-core sets

Suppose a function f has correlation $\leq 1-\delta$ with small circuits. One way in which this can happen is that there is a set H of size δ on which every circuit has very small correlation with f. The hard-core set lemma says that this is the *only way* that small correlation may arise:

Lemma 11.43. [Hard-core set] Let $\epsilon, \delta \in (0, 1/2]$. Suppose a function $f : [2]^n \to [2]$ has correlation $\leq 1-\delta$ with circuits of size s.

Then there is a set H of size $\geq c\delta 2^n$ such that every circuit C of size $(\epsilon\delta)^c s$ has correlation with f on H $\mathbb{E}_{x\in H} e[C(x) + f(x)] \leq \epsilon$.

The parameters of the hard-core lemma can be improved, see the notes. However we won't need the improvement and the proof of the version we stated is simpler. The hard-core set lemma is very useful, we shall see several applications of it. As in Claim 11.22 it is

convenient to think of functions with very small correlation as being random. In our case, the function is only random on the hard-core set.

Definition 11.44. We say that a distribution on functions $F : [2]^n \to [2]$ is δ-*random* if there exists a subset $H \subseteq [2]^n$ with $|H| = \delta 2^n$ such that $F(x) = U_1$ (i.e. a coin flip) for $x \in H$ and $F(x)$ is deterministic (i.e., a fixed value) for $x \notin H$.

Thus, a δ-random function has a set of relative size δ on which it is information-theoretically unpredictable, and so it has correlation $\leq 1-\delta$ with any sets of functions. The combination of Claim 11.22 and the hard-core set 11.43 implies that functions that don't correlate with small circuits cannot be distinguished from δ-random functions by small circuits. By Claim 11.24 this extends to product distributions, which is how we will apply this result.

Corollary 11.45. Let $\epsilon, \delta \in (0, 1/2]$. Let $f : [2]^n \to [2]$ have correlation $\leq 1-\delta$ with circuits of size s.

Then there exists a $c\delta$-random function $g : [2]^n \to [2]$ such that the distributions

$$X_1 \cdots X_k \cdot f(X_1) \cdots f(X_k) \text{ and}$$
$$X_1 \cdots X_k \cdot g(X_1) \cdots g(X_k)$$

are $k\epsilon$-indistinguishable by circuits of size $(\epsilon\delta)^c s$, where the X_i's are uniform and independent in $[2]^n$.

We can now easily formalize the XOR lemma from section 11.3.2. Suppose that f has correlation $\leq 1-\delta$ with small circuits. Then f looks like a $c\delta$-random function to small circuits. Then $f'(x_1, \ldots, x_k) := \oplus_{i=1}^k f(x_i)$ will be almost a coin flip: the probability that the output is not a coin flip is $(1-c\delta)^k$, the probability that no input falls into the hard-core set.

Lemma 11.46. Suppose $f : [2]^n \to [2]$ has correlation $\leq 1-\delta$ with circuits of size s. Then $f' : [2]^{nk} \to [2]$ defined as $f'(x_1, \ldots, x_k) := \oplus_{i=1}^k f(x_i)$ has correlation $\leq (1-c\delta)^k + k/s^c$ with circuits of size $\delta^c s^c$.

For example, if $s = 2^{n^c}$ and $\delta = 0.1$, we can take $k = cn$ and have hardness 2^{-cn}. However the function is on cn^c bits, so in terms of the input length n', f' has hardness $2^{-n'^c}$.

Proof. We use Corollary 11.45 with $\epsilon := 1/s^c$. From its conclusion it follows that

$$X_1 \cdots X_k \cdot \oplus_i f(X_i) \text{ and } X_1 \cdots X_k \cdot \oplus_i g(X_i)$$

are k/s^c-indistinguishable for size $\delta^c s^c$. Following the intuition above, the right-hand distribution is $(1-c\delta)^k$ close to $X_1 \cdots X_k \cdot U_1$. Hence the left-hand distribution is $((1-c\delta)^k + k/s^c)$-indistinguishable from $X_1 \cdots X_k \cdot U_1$ and the result follows from Claim 11.22. **QED**

11.3.3 Derandomizing the XOR lemma

A drawback of the XOR Lemma 11.46 is that the input length of the new function f' grows with the number k of copies of the original function f. This only allows us to take a modest number of copies, greatly limiting applications. We would like to decouple the input length of f' from k. We shall achieve this using... pseudorandomness! Rather than independently, we will pick the k inputs to f from a small seed. We need two properties of the k inputs. First, they should behave preserve the indistinguishability of f from the random function given by the hard-core set 11.43. For this we can use BIG (Theorem 11.32). Also, the k inputs should "hit" the hard-core set as if they were random, for which we shall use HIT, defined below. We can get both properties with BIG-HIT which is the XOR of BIG and HIT:

$$\text{BIG-HIT}(\sigma_1, \sigma_2) := \text{BIG}_S(\sigma_1) \oplus \text{HIT}(\sigma_2).$$

The map HIT is defined as follow. On input $\sigma_2 = (a, b) \in \mathbb{F}_2^n$, $\text{HIT}(a, b)$ outputs the evaluations of the line $ax + b$ on $s \leq 2^n$ distinct elements $g_0, g_1, \ldots$ of $\mathbb{F}_2^n$; i.e.,

$$\text{HIT}(a, b) := (g_0 a + b, g_1 a + b, g_2 a + b, , \ldots).$$

We set s equals to the number $|S|$ of sets for BIG, so HIT and BIG have the same output length. HIT satisfies the following "hitting" property.

Lemma 11.47. For $s \leq 2^n$ the map $\text{HIT} : [2]^{2n} \to ([2]^n)^s$ satisfies the following: for every set $H \subseteq [2]^n$ of size ϵ, $\mathbb{P}_\sigma [\text{HIT}(\sigma)_i \notin H \text{ for every } i \in [s]] \leq \epsilon^{-1}/s$.

Proof. Let $\delta := \epsilon^{-1}/s$. Let X_i be the indicator variable of $\text{HIT}(\sigma)_i \in H$, for uniform σ. The X_i are pairwise independent. Let $X := \sum_{i \in [s]} X_i$. We have $\mathbb{E}[X] = \epsilon s = \delta^{-1}$. Hence the probability to bound is $\mathbb{P}[|X - \delta^{-1}| \geq \delta^{-1}]$. Squaring both sides and using Fact A.13, this prob. is $\leq \delta^2 \mathbb{E} \left[X - \delta^{-1}\right]^2$. Thus it suffices to show that the expectation is $\leq \delta^{-1}$. **QED**

Exercise 11.48. Finish the proof.

Using BIG-HIT we can boost a modest correlation bound to an exponentially small correlation bound.

Lemma 11.49. Suppose E has a function $f : [2]^* \to [2]$ such that f_n has correlation $\leq 1 - 1/n^{1/\alpha}$ with circuits of size $2^{\alpha n}$ for a constant $\alpha \in (0, 1/2]$, $n \geq c$. Then E has a function $f' : [2]^* \to [2]$ such that f_n has correlation $\leq 2^{-c_a n}$ with circuits of size $2^{c_a n}$ for $n \geq c$.

Note the conclusion implies P = BPP by Corollary 11.37. The hypothesis can be weakened but we will not need that.

Proof. Define $f' : [2]^{c_a n} \to [2]$ as $f'(\sigma) := \oplus_{i=1}^s f(x_i)$ where $\text{BIG-HIT}(\sigma) = (x_1, x_2, \ldots, x_s)$ for $s = 2^{\gamma \alpha n}$ for a small constant $\gamma > 0$. The set system for BIG is from Lemma 11.36: a collection of s sets of size n s.t. any two sets intersect in $\leq \gamma \alpha n$ elements. To illustrate, the

parameter s is chosen large enough to give an exponentially small correlation bound by the HIT Lemma 11.47, but at the same time small enough to withstand the circuit loss in the reductions, which will be a power s and should not exceed the given bound of $2^{\alpha n}$.

We use the hard-core set Corollary 11.45 with $\epsilon := 1/s^3$. Let g the corresponding $c\delta$-random function where $\delta = 1/n^{1/\alpha}$. From Corollary 11.45 we have that the distributions

$$X_1 \cdots X_s \cdot f(X_1) \cdots f(X_s) \text{ and}$$
$$X_1 \cdots X_s \cdot g(X_1) \cdots g(X_s)$$

are $1/s^2$-indistinguishable by circuits of size $(\epsilon\delta)^c 2^{\alpha n} \geq 2^{-c3\gamma\alpha n + \alpha n}\delta^c \geq 2^{\alpha n/2}$, where the X_i's are uniform and independent in $[2]^n$.

From Theorem 11.32 it follows (cf Exercise 11.50) that

$$\sigma, f \circ \text{BIG-HIT} \text{ and } \sigma, g \circ \text{BIG-HIT}$$

are $1/s$-indistinguishable by circuits of size $2^{\alpha n/2}$ divided by the circuit size required to compute s $\gamma\alpha n$-local functions. Each $\gamma\alpha n$ local function is computable by circuits of size $\leq 2^{\gamma\alpha n}$ by Theorem 2.5. So the distributions are $1/s$-indistinguishable by circuits of size $2^{\alpha n/2}/s \cdot 2^{\gamma\alpha n} \geq 2^{\alpha n/3}$. In particular this holds if we take parities, so even

$$\sigma, \oplus_{i=1}^s f(X_i) \text{ and } \sigma, \oplus_{i=1}^s g(X_i)$$

are $1/s$-indistinguishable by circuits of size $2^{\alpha n/3}$, where $(X_1, \ldots, X_k) = G(\sigma)$. By the hitting property of HIT, Lemma 11.47 (cf Exercise 11.50), the chance that no X_i hits the hardcore set is $\leq cn^{1/\alpha}/s$, which is exponentially small. We conclude as in the proof of Lemma 11.46. **QED**

Exercise 11.50. Theorem 11.32 and Lemma 11.47 are stated for BIG and HIT respectively, but in the proof we used them for BIG-HIT, the xor of BIG and HIT. Modify the statements and proofs of Theorem 11.32 and Lemma 11.47 so that they apply to BIG-HIT as well.

11.3.4 Encoding the whole truth table

The results in the previous section give us exponentially small correlation bounds starting from modest correlation bounds. However, they can't be used when starting from a worst-case, impossibility result (i.e., a correlation bound $1 - 2/2^{-n}$ for a function on n bits. To start from worst-case hardness we need to encode the entire truth table of the function. We give a simple code that suffices for our results.

Theorem 11.51. Suppose there is $f \in \text{E}$ such that f_n cannot be computed by circuits of size $s(n)$. Then there is $f' \in \text{E}$ such that f_n has correlation $(1 - 1/n^c)$ with circuits of size $s(cn)/n^c$.

Proof. Think of an n-bit input x to f as ℓ variables $x_1, x_2, \ldots, x_\ell$ of n/ℓ bits; so each variable is over a set H of size $2^{n/\ell}$. We can write down f as a polynomial p_f of degree $(|H|-1)\ell$ over any field that includes H. This is done via a low-degree extension as in section 10.4.2, using Exercise 10.22:

$$p_f(x) = \sum_{a \in H^\ell} f(a) \cdot [x = a].$$

The gain is that we can think of evaluating p_f over a larger fields. Set $q := n^{10}$ and $d := n^5$ and $\ell := n/\log d$; note that $|H| = d = n^5$. The new function f' is constructed in two steps. First, we consider inputs over $\mathbb{F}_q^\ell \supseteq H^\ell$ (thinking of H as any fixed set of d elements of $\mathbb{F}_q$). Note the length of such inputs is $\leq c\ell \log q \leq cn$ bits, as desired. This gives a non-boolean function. To make the function boolean, we output bit i of p_f, where i is part of the new input. That is,

$$f'(x_1, \ldots, x_\ell, i) := p_f(x_1, \ldots, x_\ell)_i$$

where $x_i \in \mathbb{F}_q$ and $i \in [\log q]$.

We'd like to show that if there's a small circuit C computing f' on a $1 - 1/n^c$ fraction of inputs then there's another small circuit computing f everywhere. Let $C(x) := C(x,1) \cdots C(x, \log q)$. First note that the fraction α of $x \in \mathbb{F}_q^\ell$ such that $C(x) \neq p_f(x)$ is $\leq c/d\ell$. Because if it's larger, every such x contributes at least one input (x,i) where C disagrees with f', contradicting the assumption by the choice of constants.

Using C we give a distribution C' on circuits which computes p_f w.h.p. on every given input y. Pick a uniform line going through y, and run C on this line for $d\ell$ points. That is, pick uniform $s \in \mathbb{F}_q^\ell$ and run $C(y+ts)$ for (some fixed set of) $d\ell$ points $t \in \mathbb{F}_q$. Because each evaluation point is uniform, and $d\ell\alpha \leq c$, with prob. $> 1/2$ all the evaluations of C will be correct, and equal $p_f(y+ts)$.

Note that for fixed y and s, $p_f(y+ts)$ is a univariate polynomial p' in t of degree $\leq d\ell$. We can compute the coefficients of p' from its evaluations at $d\ell$ points. This amounts to solving a linear system which has a unique solution by Fact A.50. We can then output $q(0) = p_f(y)$.

Finally, we can repeat this cn times and output the most likely value. On every input x this errs w.p. $< 2^{-n}$. Hence we can fix the random choices and obtain a fixed circuit that succeeds on every x. This new circuit runs C on n^c evaluations and performs other power-time computation; the result follows. **QED**

11.3.5 Getting P = BPP

We can now "put it all together" and prove Theorem 3.15.

Proof of Theorem 3.15.. We apply Theorem 11.51, then Lemma 11.49, and finally Corollary 11.37. **QED**

The construction in this proof of Theorem 3.15 uses BIG twice. In fact, a single application suffices. We can use BIG-HIT and Lemma 11.49 to generate an $m \times n$ matrix of inputs, evaluate the hard function on every input, and then XOR the rows. As long as each column

has an input in the hard-core sets, the output will be close to uniform. For other alternative constructions see the notes.

11.3.6 Monotone amplification within NP

In this section we consider hardness amplification within NP. For this task we cannot use XOR since NP is not known to be closed under complement. However the machinery we developed is flexible and we can use it to establish the following hardness amplification result for NP.

Theorem 11.52. If there $f \in$ NP s.t. f_n has correlation $\leq 1 - \alpha$ with circuits of size 2^{n^α} for a constant α, and $\mathbb{P}[f_n(U) = 1] = 1/2$, then there is $f' \in$ NP with correlation $\leq 2^{-n^{c_\alpha}}$ with circuits of size $2^{n^{c_\alpha}}$.

The assumption that $\mathbb{P}[f_n(U) = 1] = 1/2$ is convenient but can be dropped in some settings. Several other extensions and optimizations have been devised, see the notes.

Rather than XOR, to amplify we shall use the *Tribes* function, a monotone DNF.

Definition 11.53. The Tribes function on s bits is:

$$\text{Tribes}(x_1, \ldots, x_s) := (x_1 \wedge \ldots \wedge x_b) \vee (x_{b+1} \wedge \ldots \wedge x_{2b}) \vee \ldots \vee (x_{k-b+1} \wedge \ldots \wedge x_s)$$

where there are s/b terms each of size b, and b is the largest integer such that $(1 - 2^{-b})^{k/b} \geq 1/2$.

It can be verified that $b \leq c \log s$.

The property of xor that we used is that if one bit is uniform, then the output is uniform. We use an analogous property for Tribes: If several bits are uniform, then the output is close to uniform. This is captured by the following.

Lemma 11.54. Let N_p be a noise vector where each is 1 independently with probability p. Then $\mathbb{E}_{x \in [2]^s, N_p} e[\text{Tribes}(x) \oplus \text{Tribes}(x \oplus N_p)] \leq 1/s^{c_p}$.

The proof is elementary but technical. We refer to the notes for it.

Proof of 11.52.. We set $s := 2^{n^{\gamma\alpha}}$ for a small constant γ. We use the map BIG-AC$^0(\sigma) = (x_1, x_2, \ldots, x_s)$, which is like BIG-HIT except that HIT is replaced with the generator in Corollary 11.35, for alternating circuits of depth c and size $(s2^n)^c$ with error, say, 2^{-n}. For BIG we use the sets from Lemma 11.34, which have size n, pairwise intersection $\leq \log s = n^{\gamma\alpha}$ and are taken from a universe of size n^2.

The seed length of BIG-AC^0 is $n' \leq n^{c_\alpha}$.

Define $f' : [2]^{n'} \to [2]$ as $f'(\sigma) := \text{Tribes}(f(x_1), \ldots, f(x_s))$ where BIG-AC$^0(\sigma) = (x_1, x_2, \ldots, x_s)$.

Following the same steps as in the proof of Lemma 11.49, one obtains that it suffices to show that the distribution

$$\sigma, \text{Tribes} \circ g \circ \text{BIG-AC}^0$$

is $1/2^{n^{c_\alpha}}$ close to uniform, where g is a c_α-random function. That is, we have to show that with high probability over σ, the distribution of Tribes$\circ g \circ$ BIG-AC^0 is close to a uniform bit just over the randomness in g. In turn, for this it suffices to bound (exercise)

$$\mathbb{E}_\sigma |\mathbb{E}_g e[\text{Tribes}\circ g \circ \text{BIG-AC}^0(\sigma)]|.$$

The square of this expectation is (see section A.6) at most

$$\leq \mathbb{E}_\sigma \mathbb{E}_g^2 e[\text{Tribes}\circ g \circ \text{BIG-AC}(\sigma)] = \mathbb{E}_{\sigma,g,g'} e[\text{Tribes}\circ g \circ \text{BIG-AC}(\sigma) \oplus \text{Tribes}\circ g' \circ \text{BIG-AC}(\sigma)].$$

Now the critical step is that Tribes$\circ g$ is computable by a distribution on alternating circuits of size $(s2^n)^c$ and depth c. These circuits compute g in brute-force, then compute Tribes. The dependence on n is terrible, but that on s is good! Because BIG-AC^0 fools such circuits with error 2^{-n}, the latter expectation is within $1/2^{n^{c_\alpha}}$ of

$$\mathbb{E}_{x_1,\ldots,x_s,g,g'} e[\text{Tribes}\circ g\circ(x_1,\ldots,x_s)\oplus\text{Tribes}\circ g'\circ(x_1,\ldots,x_s)] = \mathbb{E}_{x,N_p} e[\text{Tribes}(x)\oplus\text{Tribes}(x\oplus N_p)],$$

for $p = c_\alpha$. For this step we use that the random function can be taken to be balanced, which follows from the fact that in 11.43 the functions is nearly balanced on the hard-core set H, for else the constant function would have too large correlation. We can now conclude by Lemma 11.54. **QED**

11.4 Finding the hard core

In this section we prove 11.43. At the high-level, this is just min-max and concentration of measure, just like the equivalence between computation and correlation in Theorem 8.13. However, the proof is slightly more involved than one might anticipate. We break it up in two claims. First we obtain correlation w.r.t. a *δ-smooth distribution* D: $D(x) \leq 1/(\delta N)$ for every x. For example, D could be "flat," i.e. uniform over a set of size δN (then $D(x)$ is either 0 or $1/(N\delta)$). In the proof, D will be a combination of such flat distributions. Second we obtain a set from a smooth distribution. The straightforward combination of the claims, presented below, yields the lemma.

Claim 11.55. Let $f : [2]^n \to [2]$ be a function, G be a set of functions from $[2]^n \to [2]$, and $\epsilon, \delta \in (0, 1/2]$. Suppose that for every δ-smooth distribution D on $[2]^n$ there is $g \in G$ s.t. $\mathbb{E}_{x\leftarrow D} e[g(x) + f(x)] > \epsilon$.

Then there is a function h which has correlation $\geq 1 - c\delta$ with f and h equals the majority of $\leq c\log(1/\delta)/\epsilon^2$ functions from G.

Proof. We use the duality Theorem A.23 where one set consists of sets S of δN inputs, and the other is G, and $p(S, g) := \mathbb{E}_{x\in S} e[g(x) + f(x)]$. (In the proof of Theorem 8.13 p was just a "single point," here it's an average.) By the theorem and our assumption there is a distribution C over G for s.t. for any set S of size δN we have

$$\mathbb{E}_C p(S, C) = \mathbb{E}_C \mathbb{E}_{x\in S} e[C(x) + f(x)] = \mathbb{E}_{x\in S} \mathbb{E}_C e[C(x) + f(x)] \geq \epsilon. \tag{11.3}$$

Call an input x *hard* if C does not do well on it: $\mathbb{E}_C e[C(x) + f(x)] \leq \epsilon/2$. Now, there can't be δN hard inputs, for else we contradict equation (11.3) for the corresponding set T.

We conclude by observing that for every easy x, picking $c \log(1/\delta)/\epsilon^2$ samples from C and taking majority gives error prob. $\leq \delta$, using Lemma A.15 as in the proof of Theorem 3.3. The prob. of not computing correctly a uniform $x \in [N]$ is then at most the prob. that x is hard plus the prob. that the samples of C give the wrong value: $\delta + \delta \leq 2\delta$. **QED**

When using the following claim for a hard function h, we can let F be the set of functions of the type $e(h(x)+C(x))$ where C is a small circuit. In this way $|\mathbb{E}|_D[f(D)]|$ is the correlation of h and C w.r.t. D.

Claim 11.56. Let $\epsilon, \delta \in (0, 1/2]$. Let D be a δ-smooth distribution over $[N]$. Let F be a set of $\leq c e^{c\epsilon^2\delta^2 N}$ functions $f : [N] \to \{-1, 1\}$. Suppose for every $f \in F$ we have $|\mathbb{E}_D[f(D)]| \leq \epsilon$.

Then there is a set $S \subseteq [N]$ of size $\geq c\delta N$ s.t. for every $f \in F$ we have $|\mathbb{E}_{x \in S}[f(x)]| \leq c\epsilon$.

Even this second step is not immediate, due to the fact that the set S is constructed probabilistically and so its size – which is the normalization in the correlation – is not fixed. So we'll first prove concentration around a quantity related to D only, then connect it to $|S|$.

Proof. Construct S by placing each $x \in [N]$ in S independently with prob. $D(x)\delta N \in [0, 1]$. Let $f \in F$ and consider $X := \sum_{x \in [N]} S(x) f(x)$, where S is the indicator of set S. The variables $S(x)f(x)$ are independent and have range $[-1, 1]$. Also,

$$|\mathbb{E}[X]| = \left| \sum_{x \in [N]} [\mathbb{E} S(x)]\, f(x) \right| = (\delta N)\, |\mathbb{E}_D[f(x)]| \leq c\epsilon\delta N.$$

By tail bounds, Exercise A.18:

$$\mathbb{P}_S\left[|X| \geq c\epsilon\delta N\right] \leq 2e^{-c\epsilon^2\delta^2 N}.$$

Also, $\mathbb{E} \sum_{x \in [N]} S(x) = \delta N$. And so again by tail bounds the probability that $|S| \leq c\delta N$ is $\leq e^{-c\delta^2 N}$.

By a union bound, there exists S of size $\geq c\delta N$ s.t. for every $f \in F$ we have

$$|X| \leq c\epsilon\delta N.$$

Dividing this equation by $|S|$ we obtain

$$\frac{|X|}{|S|} = \frac{|\sum_{x \in S} f(x)|}{|S|} = |\mathbb{E}_{x \in S}[f(x)]| \leq c\epsilon\delta N/|S| \leq c\epsilon,$$

as desired. **QED**

Proof of 11.43. If the conclusion is not true then by Claim 11.56 for every δ-smooth distribution D there is circuit of size $(\epsilon\delta)^c s$ that has correlation $\geq c\epsilon$ with f. Here we use

that, as in the proof of Theorem 2.9, circuits of size s' can be described with $cs' \log s'$ bits. To apply Claim 11.56 we need $(\epsilon\delta)^c s \leq c\epsilon^2\delta^2 2^n$ for which it suffices that $s \leq 2^n/n$, which in turn can be assumed since otherwise the hypothesis if false by Theorem 2.5.

By Claim 11.55 the majority of $c\log(1/\delta)/\epsilon^2$ circuits has correlation $> \epsilon$ with f. This majority is a circuit of size $< s$, contradicting the hypothesis. **QED**

11.5 Cryptographic pseudorandom generators

The PRGs constructed from a hard function in section §11.3 are only proved to fool circuits that do not have enough resources to compute the PRG or the hard function. Indeed, the bounded-intersection generator in Theorem 11.32 outputs its seed as well as the evaluation of the hard functions on some subsets of the seed, and so if we can compute the hard function we can easily break the PRG.

By contrast, in this section we disuss PRGs that can fool even circuits that can compute the PRG itself. Such PRGs are called *cryptographic* and employ radically different techniques. As the name suggests, such PRGs are useful in a cryptographic context where a system should be secure even against attackers that have much more comptuational power than the user of the system. They also have an application to the Grand Challenge, discussed in section §17.2.

To set the stage we first discuss cryptographic PRGs against AC^0. Then we study how to construct such PRGs against general circuits.

11.5.1 AC^0

The PRG against AC^0 in Corollary 11.35 is built from the hard function parity. It is computable by AC^0 circuits that are larger than those it fools, and so it is not cryptographic. However, one can have non-trivial cryptographic generators against AC^0. In fact, the PRG is even local.

Theorem 11.57. There is a PRG with seed length $n-1$ that fools AC^0 circuits of depth d and size $2^{n^{c_d}}$ with error $1/2$. Moreover, (each output bit of) the PRG is 2-local.

Proof. The *output distribution* of the PRG is $(x, \mathrm{parity}(x))$ where x is uniform in $[2]^{n-1}$; its fooling properties follow by Claim 11.22 Corollary 9.20). While the straightforward way to compute this PRG (pick x and compute its parity) cannot be implemented in AC^0 (Corollary 9.20), as we saw in Exercise 7.38 its output distribution can be *sampled* by a 2-local map. **QED**

11.5.2 Circuits

We now discuss how to construct cryptographic PRGs against (general) circuits under assumptions. These constructions and the accompanying reductions can be carried out in the

uniform setting (i.e., we can implement the reductions by randomized word programs) but for simplicity we focus on the non-uniform, circuit setting.

Cryptographic generators with 1 bit of stretch are in fact equivalent to one-way functions (section §7.7). We refer to the notes for this equivalence. Instead, we focus on increasing the stretch. There are two constructions for this, one which takes 1 bit of stretch to linear stretch, and another which takes linear stretch to exponential (in a suitable sense).

To increase one bit of stretch, from a map $G : [2]^n \to [2]^{n+1}$ define $L : [2]^n \to [2]^{2n}$ as follows:

$$\begin{array}{ccccccccccccc} S_1 & \to & \boxed{G} & \to & S_2 & \to & \boxed{G} & \to & \cdots & \to & S_n & \to & \boxed{G} & \to & S_{n+1} \\ & & \downarrow & & & & \downarrow & & & & & & \downarrow & & \\ & & r_1 & & & & r_2 & & & & & & r_n & & \end{array} \tag{11.4}$$

where the S_i have length n, the r_i have length 1, and L maps $S_1 \in [2]^n$ to $(r_1, r_2, \ldots, r_n, S_{n+1}) \in [2]^{2n}$. In words, compute G, output one bit and take the remaining n bits as input, and repeat.

Theorem 11.58. Suppose a circuit of size s ϵ-breaks L, and G is computable by circuits of size g. Then a circuit of size $s + ng$ breaks G with advantage ϵ/n.

Note that this result is only non-trivial in the cryptographic regime where G fools circuits of size $> g$.

Proof. As in the proof of Claim 11.24, consider the hybrid distributions H_i over $2n$ bits where the bits $r_1, r_2, \ldots, r_{i-1}$ are uniform, S_i is uniform, and the bits $r_i, \ldots, r_n, S_{n+1}$ are obtained following the same process in the definition of L (i.e., we apply G to S_i to get bit r_i, then use the other n bits as seed and repeat). Note that H_1 is the same as L, whereas H_{n+1} is uniform.

By assumption and the triangle inequality there is a circuit C of size s which distinguishes two adjacent hybrids H_i, H_{i+1} with advantage $\geq \epsilon/n$. We can fix all the bits up to r_{i-1} and preserve this advantage. We can now break G as follows. Given a challenge $(x, b) \in [2]^n \times [2]$, output b for r_i and get the next bits by the process starting with seed x, then run C. The corresponding circuit runs G for $\leq n$ times and so has size $s + ng$. It ϵ/n-breaks G because:

If (x, b) is uniformly distributed then the circuit C is run on H_{i+1}.

If $(x, b) = G(U)$ then C is run on H_i. **QED**

The construction of L (equation (11.4)) and the corresponding Theorem 11.58 stand in stark contrast with BIG (Definition 11.31 and Theorem 11.32). The former increases stretch by a corresponding increase in depth, whereas the latter does not.

Open question 11.59. Can we increase the stretch of a cryptographic PRG without a corresponding increase in circuit depth?

Once we have a cryptographic PRG G whose output length is twice the input length, i.e., $G : [2]^n \to [2]^{2n}$, we can use a more efficient construction to increase the stretch. Define the maps

$$G_i : [2]^n \to [2]^{2^i \cdot n}$$

recursively as follows

$$G_0(x) := x$$
$$G_{i+1}(x) := G_i(y)G_i(z), \text{ where } G(x) = yz.$$

We can visualize G_i as a binary tree and think of it as a *pseudorandom function* from $[2]^i$ to $[2]^n$. The output of the function at $y \in [2]^i$ is the leaf reached following the path in the tree corresponding to y. Importantly, these functions can be computed efficiently even when 2^i is prohibitive (since an evaluation only requires following a path in the tree). One can show that such functions cannot be distinguished by a uniform function. We state and prove this in a simple setting where the distinguisher has enough resources to look at the entire function (one can also consider distinguishers which have fewer resources but have oracle access to the function as in section §17.1); for example, we can think of $i = n^{0.1}$. We will see an application in section §17.2.

Theorem 11.60. Suppose a circuit of size s ϵ-breaks G_i and G is computable by circuits of size g. Then a circuit of size $s + 2 \cdot g \cdot 2^i$ breaks G with advantage $\epsilon/(i \cdot 2^i)$.

Proof. Consider the hybrid distributions

$$H_j := \left(G_{i-j}(U)\right)^{2^j},$$

i.e., the concatenation of 2^j copies of G_{i-j} on independent seeds. Note $H_0 = G_i$ and $H_i = (U)^{2^i}$ is uniform. By the triangle inequality there is $j \in [i]$ and a circuit of size s that ϵ/i-distinguishes two adjacent hybrids

$$H_{j+1} = \left(G_{i-j-1}(U)\right)^{2^{j+1}} = \left(G_{i-j-1}(U)G_{i-j-1}(U)\right)^{2^j} \text{ and}$$
$$H_j = \left(G_{i-j}(U)\right)^{2^j} = \left(G_{i-j-1}(Y)G_{i-j-1}(Z)\right)^{2^j},$$

where $(Y, Z) := G(U)$.

By Claim 11.24 there is a circuit C of size $\leq s$ that distinguishes

$$G_{i-j-1}(U)G_{i-j-1}(U)$$

from

$$G_{i-j-1}(Y)G_{i-j-1}(Z)$$

with advantage $\geq \epsilon/(i \cdot 2^i)$.

Computing G_k amounts to $2^k - 1$ evaluations of G. Hence G_{i-j-1} can be computed by circuits of size $\leq g \cdot 2^i$. Hence we can break G with advantage $\epsilon/(i \cdot 2^i)$ by computing G_{i-j-1} on the two halves and then running C. The circuit has size $\leq s + 2 \cdot g \cdot 2^i$. **QED**

11.6 Problems

Problem 11.1. Prove that there is a function $f : [2]^n \to [2]$ such that the following holds for the distribution $D := Uf(U)$ on $n+1$ bits.

(1) D is 0.1-pseudorandom for circuits of size n^2.

(2) circuits of size a 0.9-break D^a, for a constant a.

Problem 11.2. Give a PRG for degree-1 polynomials in n variables over $\mathbb{F}_2$ with error ϵ and seed length $\log n + c \log\log n + c\log(1/\epsilon)$. Hint: What property do you need from t in the construction in Theorem 11.11?

Problem 11.3. Prove that any ϵ-biased distribution section 11.1.2 on n bits also $\left(2^{d/2}\epsilon\right)$-fools d-local function. Hint: Use Fact A.39.

Problem 11.4. Let D be a *zero*-bias distribution over $[2]^n$. Prove that D is uniform. Hint: Use Fact A.39.

Problem 11.5. Give a PRG G that ϵ-fools the product of (indicator functions of) t subsets of $[2]^m$ on disjoint bitsm with seed length $m + c(\log t)\log(mt/\epsilon)$ (note the output length of the PRG is $n = mt$; you can assume that t and m are given as part of the input, or even that $t = m$). In other words, for t functions $f_i : [2]^m \to [2]$ we have $|\mathbb{E}[\prod_i f_i(U_i)] - \mathbb{E}[\prod_i f_i(X_i)]| \leq \epsilon$ where $(X_1, X_2, \ldots, X_t) = G(U)$.

Guideline: First prove it for $t = 2$. The generator for this base case outputs $(U, U + D)$ where U is uniform and D is an ϵ-biased distribution. To analyze, write f_1 and f_2 as polynomials and use Fact A.39. Second, prove it for any t using recursion.

Let $t = m$ and suppose that given a seed and $i \in [t]$ we can compute the m-bit output of the PRG corresponding to set i in quasi-linear time. Prove Item (2) in Theorem 5.36.

Problem 11.6. [Alternative xor lemma without the hard-core set 11.43] Let $f : [2]^n \to \{-1, 1\}$ have correlation $\leq \epsilon$ with any circuit $C : [2]^n \to \{-1, 1\}$ of size s, i.e.,

$$|\mathbb{E}_x[f(x) \cdot C(x)]| \leq \epsilon.$$

(Note we use range $\{-1, 1\}$ to simplify exposition.) Define $f' : [2]^n \times [2]^n \to \{-1, 1\}$ as $f'(x, y) := f(x) \cdot f(y)$. Prove that f' has correlation $\leq c\epsilon^2$ with any circuit $C' : [2]^n \times [2]^n \to \{-1, 1\}$ of size $(\epsilon/n)^c s$.

Guideline: Suppose the conclusion does not hold and write

$$\epsilon^2 \leq \mathbb{E}_{x,y}[f'(x, y) \cdot C'(x, y)] = \mathbb{E}_x \left[f(x)\mathbb{E}_y[f(y)C'(x, y)]\right].$$

Let $h(x) := \mathbb{E}_y[f(y)C'(x, y)]$. Prove that $|h(x)| \leq \epsilon$ for every x.

Let $C'' : [2]^n \to \{-1, 1\}$ be a distribution on circuits s.t. for every x we have $\mathbb{E}[C''(x)] = h(x)/\epsilon$. Prove $\mathbb{E}_{x,C''}[f(x) \cdot C''(x)] \geq \epsilon$.

Conclude the proof by constructing a distribution C'' on circuits of the desired size s.t. $\mathbb{E}[C''(x)]$ is close to $h(x)/\epsilon$. Use Lemma A.15.

11.7 Notes

For more on unconditional pseurandom generators see [169]. For a broader view of pseudorandomness, with an emphasis on connections between various objects, see [347].

k-wise uniform distributions were studied before complexity theory, see [286]. The complexity viewpoint is from [88, 23].

(ϵ, k)-biased distributions originate in [261], with alternative constructions in [24]. For more on these distributions, see [102, 103]. A proof of Theorem 11.17 can be found in [25, 22, 275], with the last work achieving the stated parameters.

The idea of xoring small-bias generators to fool higher-degree polynomials is from [64]. It was studied further in [236, 366, 104]. [366] proves Theorem 11.14.

The idea of using "hard" functions to construct PRGs originates in [309]. This idea was then greatly extended in several directions. Historically, the cryptographic setting cam first. Works such as [63, 388, 135, 136, 167, 156] showed how to construct cryptographic pseudorandom generators from one-way functions. In particular, the construction of linear-stretch generators, Theorem 11.58, is from [63]; the construction of pseudorandom functions, Theorem 11.60, is from [135].

The construction of PRGs from functions that are hard to compute, section §11.3 shares some underlying ideas with the cryptographic setting. But the key difference is how generators with large stretch are obtained, cf Question 11.59. For more on this question see [361, 249]. The BIG (Definition 11.31) and accompanying Theorem 11.32 for the case of uniform distributions are from [267, 271]. The general statement in terms of preserving indistinguishability is from [172]. The relationship between [267, 271] is somewhat unclear. The PRG for AC^0 circuits, Corollary 11.35, and the construction Lemma 11.34 are from [267]. More general statements are made in [271], including Claim 11.4, Corollary 11.37. Lemma 11.36 also appears there. Several other proofs of this Lemma 11.36 appear in [214], [360], [154] (building on [160]). The proof we presented is from [360]. The construction of functions with

The XOR lemma was reportedly announced in talks associated with the work [388], see [138] where three proofs appear. For another proof see [376]. Hardness amplification within NP was first studied in [273] achieving correlation about $1/\sqrt{n}$. Exponentially small correlation was achieved in [172], with optimizations in [237, 144]. Our exposition follows [172]. The key result for Tribes, Lemma 11.54, is proved in [273], cf [172].

The had-core set 11.43 is from [187]. After that a number of alternative proofs and optimizations have been obtained, including [212, 179, 46, 79]. Theorem 11.51 is from [35], building on results in the same spirit from [53, 233, 34, 127].

The construction of PRGs sufficient to obtain P = BPP from hard functions in E, Theorem 3.15, is from [192]. The proof presented here is somewhat different. The original proof in [192] uses another result from [187], one proof in [332] uses list-decoding algorithms, another proof in [332] uses extractors, see Lemma 15 in [332]. The alternative construction I sketched in section 11.3.5 replaces extractors with the simpler XOR. Alternative constructions of such PRGs from hard functions are in [306, 346], the latter giving a comprehensive tradeoff between the hardness of the function and the stretch of the PRG.

For more on hitters, see Appendix C in [131].

The quote at the beginning of section 11.3.2 is from my PhD thesis [362].

Theorem 11.19 and Theorem 11.21 are from [338]. Several proofs of Theorem 11.19 have appeared [203, 114]. Our presentation, based on Theorem 9.8 seems a little different. $\oplus$P was defined in [278]. A class similar to SymP (but with majority instead of an arbitrary symmetric function) is defined in [184].

Chapter 12

Expansion

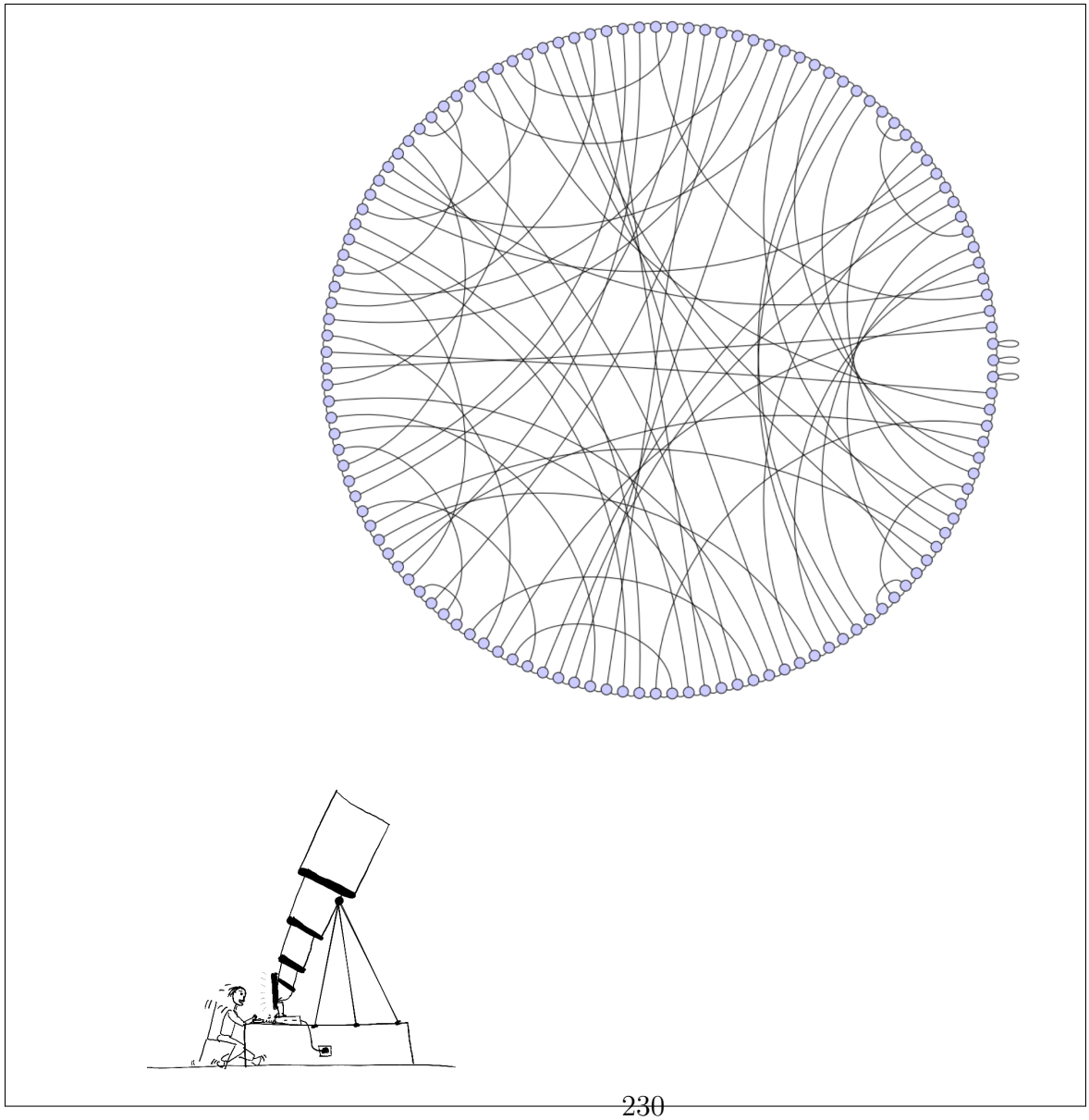


Figure 12.1: You may have seen a black hole, but have you ever seen an expander? The expander graph in Example 12.4.

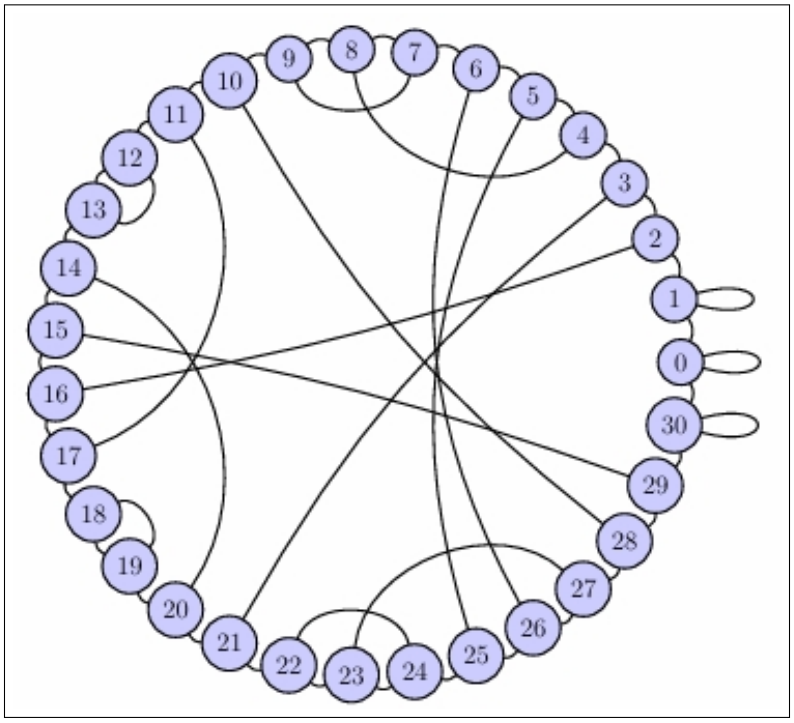


Figure 12.2: A 3-regular expander.

An *expander graph* is a graph that is sparse yet "highly connected." These graphs have many applications ranging from circuit lower bounds (see Chapter 18), to log-space computation (Theorem 6.22), communication complexity (Problem 13.2) and PCPs (see Theorem 4.33 and section §12.5). Various related notions of expansion exist. We begin in section §12.1 with a construction of expanders with respect to the "combinatorial" notion of *edge expansion.* The results in this section are not needed for the rest of this chapter, but can provide helpful intuition and motivation. In section §12.2 we discuss spectral expansion. In section §12.3 we use spectral expansion to give a log-space algorithm for undirected reachability (Theorem 6.22). As a byproduct, we will also get a different construction of spectral expanders. In section §12.5 we sketch a proof of the PCP Theorem 4.33 which uses expander graphs and in fact has the same high-level structure as the construction of spectral expanders in section §12.3.

We first discuss how we represent graphs. Our graphs are regular, undirected, and may have self loops and repeated edges. We can represent such a graph in several ways. One way is via the *normalized adjacency matrix* A, where "normalized" means each entry is divided by the degree d. This is also called the *random-walk matrix*. For example, the following is

the random-walk matrix of the graph in figure 12.3.

$$A = \begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 2/3 & 0 \\ 1/3 & 0 & 2/3 \end{bmatrix}.$$

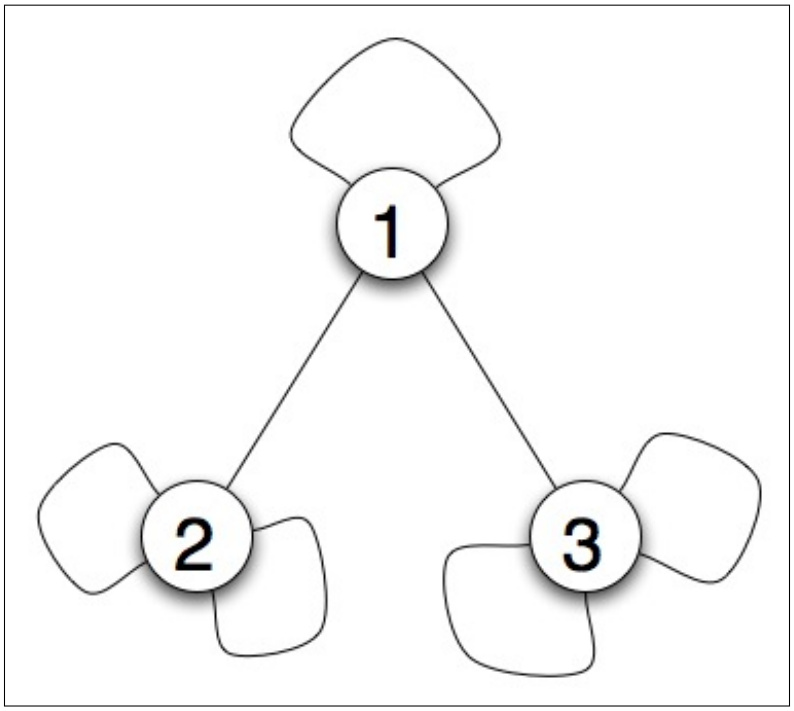


Figure 12.3: Graph example.

For a probability distribution p on the nodes, Ap is then the probability obtained by starting from a random node selected according to p, and going to a uniformly selected neighbor. For example, for $v = (1, 0, 0)$ (the "distribution" which always picks node 1) we have $Av = (1/3, 1/3, 1/3)$ in the graph above. Matrices do not play a significant role until section §12.2.

Another way to describe graphs, which is particularly suitable when the graph is large, is to identify a d-regular graph on vertex set V with its *neighbor function* $f : V \times [d] \to V$ which given a node and an edge outputs the corresponding neighbor. We use the notation $v[i] := f(v, i)$ when the graph is clear from the context. For the neighbor function to correspond to a graph it must be symmetric, so the number of i s.t. $f(x, i) = y$ must be equal to those s.t. $f(y, i) = x$, for every $x, y \in V$. A *rotation map* can give the corresponding permutation on $V \times [d]$, i.e., it maps (x, i) to (y, j) where edge i leaving x equals edge j entering y. A natural, convenient, and stronger condition is *consistency,* which asks that an edge name is the same from either endpoint: $f(f(x, i), i) = x$. This is equivalent to an edge coloring of the graph with d colors, something not every d-regular graph enjoys.

Exercise 12.1. Show that the natural neighbor function for the circle graph from Example 12.4, $f(x,i) := x + i$, for $i \in \{-1, 1\}$, is not consistent. Give a consistent neighbor function for the same graph.

In section §12.1 we shall work with consistent neighbor functions only, but later we will have to drop consistency.

12.1 Edge expansion

In this section we give a simple construction of expanders with respect to a combinatorial notion of expansion defined next.

Definition 12.2. Let $S \subseteq V$ be a subset of the nodes V of a d-regular graph $f : V \times [d] \to V$. The *crossing probability* $\nmid(S)$ of S is the probability that starting from a uniform $s \in S$ and moving to a uniform neighbor we end up outside of S. The graph is an *edge δ-expander* if $\nmid(S) \geq \delta$ whenever $|S| \leq |V|/2$.

Note that

$$\nmid(S) = \mathbb{P}_{x \in S, i \in [d]}[f(x,i) \notin S] = \mathbb{P}_{x \in S, i \in [d]}[x[i] \notin S] = \frac{E\left(S, \overline{S}\right)}{d|S|},$$

where $E(S,T)$ is the multiset of edges with an endpoint in S and the other in T.

Whereas Definition 12.2 is stated for sets of density $\leq 1/2$, it implies expansion for larger sets as well.

Exercise 12.3. Let $f : V \times [d] \to V$ be a δ-expander and let $S \subseteq V$ be of density $\leq (1-\epsilon)$. Prove $\nmid(S) \geq \delta\epsilon/(1-\epsilon)$.

Any connected d-regular graph on vertex set V is an edge δ-expander for $\delta \geq 1/(|V|d)$, since at least one edge leaves S. *Expander graphs* have much larger edge expansion, ideally a constant independent of the size of the graph.

Example 12.4. The *circle graph* over nodes $\mathbb{Z}_N$ where the neighbors of x are $x \pm 1$, depicted in figure 12.4, is not an expander. For example, $\nmid([N/2]) \to 0$ with $N \to \infty$.

On the other hand, the 3-regular graph in figure 12.2, and figure 12.1, which can be obtained from the circle graph by adding "chords," is an edge c-expander. This expander has $\mathbb{F}_p$ as nodes, and the neighbors of x are $x \pm 1$ and $1/x$. For $x = 0$ we replace the undefined $1/x$ with a self-loop, making the graph 3-regular.

For another example, we define an 8-regular graph on nodes $\mathbb{Z}_M \times \mathbb{Z}_M$. The neighbors of the vertex (x,y) are $(x+y, y), (x-y, y), (x, y+x), (x, y-x), (x+y+1, y), (x-y+1, y), (x, y+x+1), (x, y-x+1)$.

For proofs of expansion see the notes.

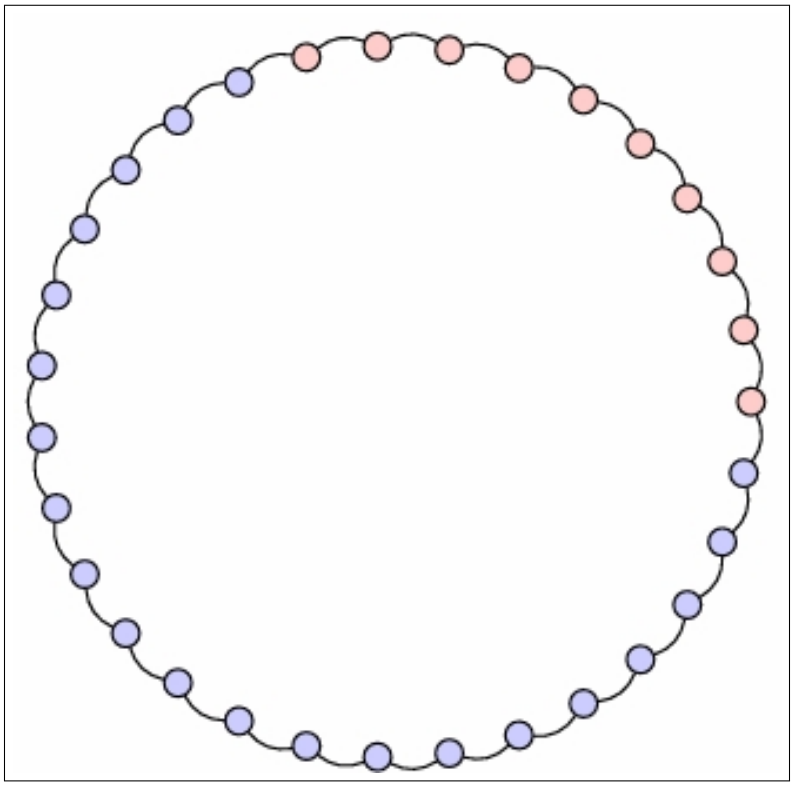


Figure 12.4: The circle graph, on 31 nodes. A set of 10 consecutive nodes only has 2 edges leaving it.

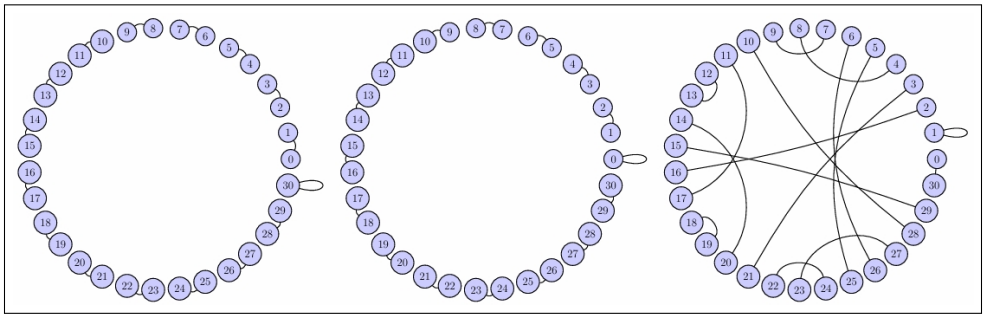


Figure 12.5: The expander in figure 12.2 is the union of these three matchings.

The main result in this section is that there are expander graphs that are explicit: They have a consistent neighbor function computable in FP.

Theorem 12.5. There are edge c-expanders on 2^n nodes with degree c and a consistent neighbor function in FP.

The expander is obtained by starting with expanders with logarithmic degree, and combining them using an operation on graphs called *replacement product* which reduces the degree without sacrificing the expansion. It suffices to apply this product three times to reduce the problem to constructing expander graphs on very few nodes – which we can just brute force.

We now give each of these component. First, we give a non-explicit construction of expanders.

Theorem 12.6. There are c-regular c-expanders on N nodes with a consistent neighbor function computable in time 2^{N^c}.

Note that the time is doubly-exponential in the length of a node name.

Proof. We use the probabilistic method. We pick a random d-regular graph by picking d independent, uniform *matchings*. A matching is a maximal collection of disjoint edges. figure 12.5 illustrates a decomposition in matchings for the expander in figure 12.2. Because the number of nodes is odd there, each matching also includes a self-loop. Here we assume that N is even, and we exclude self-loops from matchings, so a matching has size $N/2$.

We will soon need to analyze how the distribution of a uniform matching looks locally, i.e., on a set of S nodes. For this purpose, we note that for any node v we can sample a uniform matching by first sampling a uniform edge $\{v, w\}$ for $w \neq v$ and then sampling a uniform matching on the remaining $N - 2$.

Now fix a set S of size $\leq N/2$. We pick one matching iteratively, at every iteration matching the first unmatched node (in an arbitrary ordering of S). This is possible because of the way we can sample a uniform matching we just described. The iterative process is repeated over $\geq \lceil S/2 \rceil =: T$ elements of S (accounting for the fact that one edge matches 2

or 1 nodes). For each of the first T such iterations, we have matched $\leq S/2 \leq N/4$ elements outside of S. Hence the probability of matching within S at that iteration is

$$\leq p := \frac{S}{N - N/4} = \frac{S}{0.75N} \leq 2/3.$$

These events are not independent, but we can still use the deviation bound Lemma A.15 via Exercise A.18. So the prob. that this matching is *bad* for S in the sense that more than q fraction of these T iterations is matched within S satisfies

$$\mathbb{P}[\text{bad}]^{1/T} \leq \left(\frac{p}{q}\right)^q \left(\frac{1-p}{1-q}\right)^{1-q} \leq \left(\frac{p}{q}\right)^q \left(\frac{1}{1-q}\right)^{1-q} \leq p^{q/2} \leq p^c.$$

The latter inequality holds for a suitable choice of $q < 1$ because using $p \leq 2/3$ it is implied by

$$(2/3)^{q/2} \leq q^q (1-q)^{1-q}$$

which holds for $q \to 1$ as the lhs goes to $\sqrt{2/3} < 1$, and the rhs goes to 1.

If we pick d matchings the probability that they are all bad is $\leq p^{cdT}$. (This loose bound suffices for our claim.) When that does not happen, the crossing prob. $\nmid(S)$ is at least the prob. of selecting a good matching ($\geq 1/d$), times the probability of picking one of the $\geq (1-q)T \geq (1-q)\lceil S/2 \rceil \geq cS$ nodes matched outside of S. Overall, $\nmid(S) \geq c/d$, which is as desired for constant d.

The prob. that there exists a bad set S of size k for which all matchings are bad is by a union bound and Fact A.7

$$\leq \binom{N}{k} p^{cdT} \leq \left(\frac{eN}{k}\right)^k \left(\frac{k}{0.75N}\right)^{cd\lceil k/2 \rceil} \leq \frac{1}{2^k}$$

for $d \geq c$.

Hence the prob. there is any bad set of size $\leq N/2$ is $\leq \sum_{k=1}^{N/2} 2^{-k} < 1$.

For the explicitness, we enumerate over all graphs. Each graph can be described using $Nc \log N$ bits. Its expansion can again be checked by enumerating over all $\leq 2^N$ subsets of nodes. Because our graph is obtained as the union of matchings, it has a consistent neighbor function (each matching corresponds to an edge index, or equivalently a color). **QED**

Next we give explicit expanders with logarithmic degree. They come from small-bias generators (Definition 11.12).

Theorem 12.7. There are c-expanders on 2^n nodes with degree n^c with a consistent neighbor function computable in FP.

Proof. Let Y be an ϵ-biased distribution on $\mathbb{F}_2^n$ (see Theorem 11.11). For minor simplification we think of the domain as $\{-1,1\}^n$. The neighbors of $x \in \{-1,1\}^n$ are then $x \cdot Y$, where multiplication is component-wise and $y \in \{-1,1\}^n$. Theorem 11.11 implies that the graph is explicit. The neighbor function is consistent because $(x \cdot Y) \cdot Y = x$.

To analyze, let S be a set of density $p := |S|/2^n$ with characteristic function $f : [2]^n \to [2]$. We have

$$\nmid (S) = p^{-1}\mathbb{E}[f(X)(1 - f(X \cdot Y))] = 1 - p^{-1}\mathbb{E}[f(X)f(X \cdot Y)].$$

We write $f = \sum_{T \subseteq [n]} \widehat{f_T} x^T$ in the basis of parity functions (Fact A.39) and get, using linearity of expectation:

$$\begin{aligned}
\mathbb{E}[f(X)f(X \cdot Y)] &= \mathbb{E}\left[\left(\sum_{T \subseteq [n]} \hat{f}_T X^T\right)\left(\sum_{U \subseteq [n]} \hat{f}_U (X \cdot Y)^U\right)\right] \\
&= \mathbb{E}\left[\sum_{T,U \subseteq [n]} \hat{f}_T \hat{f}_U X^T \cdot X^U \cdot Y^U\right] \\
&= \sum_{T,U \subseteq [n]} \hat{f}_T \hat{f}_U \mathbb{E}\left[X^T \cdot X^U \cdot Y^U\right].
\end{aligned}$$

When $T \neq U$ the expectation is 0 over the choice of X, which is uniform. So we can rewrite the expression as

$$\sum_{T \subseteq [n]} \hat{f}_T^2 \mathbb{E}\left[Y^T\right] = p^2 + \sum_{T \neq \emptyset} \hat{f}_T^2 \mathbb{E}\left[Y^T\right] \leq p^2 + 2\epsilon \sum_{T \neq \emptyset} \widehat{f}_T^2,$$

using that Y is ϵ-biased. The factor 2 arises as in Definition 8.11 because the assumption on Y is $|\mathbb{P}[Y^T = 1] - 1/2| \leq \epsilon$ but we analyze $\mathbb{E}[Y^T]$.

Using that f has values in [2] we also have:

$$\widehat{f_0} = \mathbb{E}[f] = \mathbb{E}[f^2] = \sum_T \widehat{f}_T^2 = p.$$

Putting these facts together we get $\mathbb{E}[f(X)f(X \cdot Y)] \leq p^2 + \epsilon(p - p^2)$ and so

$$\nmid (S) \geq 1 - p - 2\epsilon(1 - p) = (1 - p)(1 - 2\epsilon).$$

The latter is $\geq c$ for p and ϵ both $\leq c$. **QED**

We now seek to reduce the degree to constant by means of the following operation.

Definition 12.8. [Replacement product] Let G be a D-regular graph on N vertices and H a d-regular graph on D vertices. The replacement product $G \boxtimes H$ is the following $2d$-regular graph on $N \cdot D$ vertices written $(x, i) \in [N] \times [D]$. For every vertex $x \in G$ there is a copy H_x of H, i.e., we connect the nodes (x, i) for $i \in [D]$ according to H. In addition, we connect (x, i) to (y, i) if $x[i] = y$, with d parallel edges.

Equivalently, we can write edges as $[bj]$ where $b \in [2]$ and $j \in [d]$; then $(v, i)[0j]$ is connected to $(v, i[j])$ and $(v, i)[1j]$ is connected to $(v[i], i)$. (The square brackets correspond to neighbor functions of 3 different graphs.) Note that if G and H have consistent neighbor function, then so does $G \boxtimes H$. Repeating edges makes it equally likely that a random neighbor makes a step inside a copy of H or outside, corresponding to an edge in G.

Theorem 12.9. Suppose E_1 is a D-regular δ_1-expander on N nodes, and E_2 is a d-regular δ_2-expander on D nodes. Then $E_3 := E_1 \boxtimes E_2$ is $2d$-regular $c\delta_1^2\delta_2$-expander on ND nodes.

Proof. Let X be a set of nodes in E_2 of size $\leq ND/2$. We view the vertex set of E_3 as composed of N clusters of vertices C_i, each of size $D = |C_i|$. Let $X_i := X \cap C_i$ and consider two cases. Either many X_i are underfull (within C_i), in which case many edges are leaving X within the clusters due to the expansion of E_2; or many X_i are almost full, in which case there are many edges leaving X between the clusters, due to the expansion of E_1. Details follow.

Let I' be the indices of the X_i which we call *underfull* and have size $\leq (1-\delta_1/4)C_i$, and let I'' be the others; let $X' := \bigcup_{i\in I'} X_i$ and $X'' := \bigcup_{i\in I''} X_i$.

If $|X'|/|X| \geq \delta_1/10$ (X consists of a noticeable amount of underfull clusters) then we can think of the experiment in $\nmid(X)$ as sampling a uniform node from X' with prob. $\geq \delta_1/10$. For any $X_i \subseteq X'$ we have $\nmid(X_i) \geq 0.5 \cdot \delta_2(\delta_1/4)$ by Exercise 12.3, where the 0.5 is for taking a step within C_i. Hence $\nmid(X) \geq (\delta_1/10) \cdot 0.5 \cdot \delta_2\delta_1/4$ and we are done in this case.

Otherwise $|X''|/|X| \geq (1-\delta_1/10)$ (X is almost all made of almost full clusters). Let F (for "full") be the union $\bigcup_{i\in I''} C_i$ of the almost full clusters. We have

$$E(X,\overline{X}) \geq E(F,\overline{F}) - E(\cup_{i\in I''}\overline{X_i},\overline{F}) - E(X',\overline{X'}). \tag{12.1}$$

To bound $E(F,\overline{F})$, let us first note that F is fairly small. Recall $X_i \geq C_i(1-\delta_1/4)$ for $i \in I''$. Summing over such i we get

$$|F| \leq \frac{|X''|}{1-\delta_1/4} \leq \frac{4|X''|}{3} \leq \frac{4|X|}{3} \leq \frac{2ND}{3}.$$

Hence by Exercise 12.3, $E(F,\bar{F}) \geq 0.5\delta_1 d|F|$.

To bound the next term $E(\bigcup_{i\in I''}\overline{X_i},\overline{F})$ we simply use that $\bigcup_{i\in I''}\overline{X_i}$ has size $\leq |I''| \cdot D \cdot \delta_1/4 = |F|\delta_1/4$, since each X_i is not underfull. Since each node in any $\bar{X}_i$has $\leq d$ neighbors outside of F, we have that $E(\bigcup_{i\in I''}\overline{X_i},\overline{F}) \leq |F|d\delta_1/4$.

So the first difference in the rhs in equation (12.1) above is $\geq (|F|\delta_1/4)\cdot d$. Because

$$|F| \geq (1-\delta_1/4)|X''| \geq (1-\delta_1/4)(1-\delta_1/10)|X| \geq (3/4)(9/10)|X| \geq 0.5|X|,$$

this difference is $\geq |X|d\delta_1/8$.

Finally, $E(X',\overline{X'})$ is trivially $\leq d|X'| \leq d|X|\delta_1/10$.

Putting these bounds together we obtain $E(X,\overline{X}) \geq |X|dc\delta_1$. **QED**

We can now mix these three ingredients to construct explicit expanders.

Proof of Theorem 12.5. Start with E_1 the expander from Theorem 12.7 with 2^n nodes and degree n^c, and E_2 the same expander but on n^c nodes and degree $\log^c n$.

Then $E_1 \boxtimes E_2$ is an expander with degree $2\log^c n$. That is, one replacement product allows us to reduce the degree logarithmically. Doing this again, we can reduce the degree to $\leq \log^c \log n$. Finally, we take a replacement product with the non-explicit expander from

Theorem 12.6 to obtain constant degree. The neighbor function in the latter is computable in time $2^{\log^c \log n} = n^{o(1)}$. Because each graph is an expander, and we only apply replacement product three times, the final graph is an expander by Theorem 12.9. **QED**

Exercise 12.10. Prove that the neighbor function is computable in FL.

12.2 Spectral expansion

In this section we discuss a different notion of expansion which says that taking a random step on the graph gets us closer to the uniform distribution. It is called *spectral* expansion because it is the same as the second largest eigenvalue in absolute value, and eigenvalues are called the spectra of the matrix. But we will not use eigenvalue theory.

Definition 12.11. [Spectral expander] A regular graph with random-walk matrix A is a spectral λ-expander if for any probability distribution p we have

$$|Ap - u| \leq \lambda |p - u|,$$

where u is the uniform distribution and $|\cdot|$ is the geometric norm $|v| := \sqrt{\sum_i v_i^2}$.

Example 12.12. The $n \times n$ matrix J_n which has $1/n$ in every entry (corresponding to a complete graph with a self-loop on each node) has spectral expansion 0. This is because multiplying any probability distribution v by J we obtain the uniform distribution u, so the lhs is 0. By contrast, any permutation matrix has spectral expansion 1.

An equivalent formulation of Definition 12.11 states that for every vector v whose sum of coordinates is 0 we have

$$|Av| \leq \lambda |v|.$$

We write $v \perp u$ for v perpendicular to the uniform distribution, which is an equivalent way of saying that the sum of the coordinates of v is 0.

Exercise 12.13. Prove the equivalence.

Edge expansion increases with the crossing probability $\nmid$ that a random edge leaving a set crosses its boundary, whereas spectral expansion improves as the parameter λ becomes smaller.

Note that the edge expansion of a graph grows with the crossing probability, but the spectral expansion is better the smaller λ.

Edge and spectral expansion are equivalent, as long as each node has enough self loops. Self-loops are essential because without them we could take a bipartite graph with good edge expansion. But bipartite graphs are not spectral expanders.

Exercise 12.14. Show that bipartite graphs are not good spectral expanders.

Having few self loops does not give a good equivalence, as their effect can be negligible if the degree is large. If half the edges are self loops, the equivalence is good. This is similar to what we did in the replacement product Definition 12.8 where the edges across clusters were duplicated to balance their effect against that of the edges within clusters.

Theorem 12.15. [Equivalence of edge and spectral expansion] Consider a regular undirected graph.

(1) If it is a spectral λ-expander then it is also an edge $(1-\lambda)/2$-expander.

(2) If the graph is connected and has at least one self-loop at each node then it is a spectral $(1-1/n^c)$-expander, where n is the number of nodes and an upper bound on the degree.

(3) If it is an edge δ-expander, and at least half the edges leaving any node are self-loops, then it is also a spectral $(1-c\delta^2)$-expander.

Proof. (1) Let G be d regular on N nodes. For any vector v note

$$|v|^2 - \langle Av, v\rangle = \sum_{i,j} A_{i,j}v_i^2 - \sum_{i,j} A_{i,j}v_iv_j = \frac{1}{2}\sum_{i,j} A_{i,j}(v_i - v_j)^2 = \frac{1}{d}\sum_{\{i,j\}\in E}(v_i - v_j)^2. \quad (12.2)$$

Let's now pick a suitable vector. Let $v_i := |\overline{S}|$ if $i \in S$ and $-|S|$ if $i \in \overline{S}$. Note that $\sum v_i = 0$.

The rhs in equation (12.2) is $(1/d)E(S,\bar{S})N^2$ since each edge leaving S contributes N^2, and edges within S or within $\bar{S}$ contribute 0. Also $|v|^2 = |S||\overline{S}|^2 + |\overline{S}||S|^2 = |S||\overline{S}|N$, and $\langle Av, v\rangle \le |Av|\cdot|v| \le \lambda\cdot|v|^2$ by Fact A.22 and the spectral expansion hypothesis.

Putting things together and dividing everything by $|v|^2$ we get

$$\frac{(1/d)E(S,\bar{S})N^2}{|S||\overline{S}|N} \ge 1-\lambda.$$

The lhs is

$$\frac{E(S,\bar{S})}{d|S|}\cdot\frac{N}{|\bar{S}|} \le \upharpoonright(S)\cdot 2$$

concluding the proof.

(2) Let v be a vector with $v \perp u$. We need to bound $|Av|/|v|$. By scaling, we can assume that $|v| = 1$. Let $w := Av$. It suffices to prove that

$$|v|^2 - |w|^2 \ge 1/n^c.$$

Using a derivation similar to Part (1), but for two vectors rather than one, we can rewrite the lhs as

$$|v|^2 - 2\langle Av, w\rangle + |w|^2 = \sum_{i,j} A_{i,j}(v_i - w_j)^2.$$

Since $v \perp u$, some coordinates in v are > 0 and some others are < 0. Moreover, since $|v| = 1$ some coordinates will be $\ge 1/\sqrt{n}$ in absolute value. W.l.o.g. we can assume $v_i - v_j \ge 1/\sqrt{n}$

for some i, j. Because the graph is connected, there is a path from i to j of length $\leq t$. By renaming the coordinates we can assume $i = 1$ and $j = t + 1$ and the path is $1, 2, \ldots, t + 1$. Telescoping we can now write

$$\begin{aligned} v_1 - v_{t+1} &= \sum_{k=1}^{t} (v_k - v_{k+1}) \\ &= \sum_{k=1}^{t} ((v_k - w_k) + (w_k - v_{k+1})) \\ &\leq \sqrt{2t} \sqrt{\sum_{k=1}^{t} \left((v_k - w_k)^2 + (w_k - v_{k+1})^2\right)}. \\ &\leq n^c \sqrt{\sum_{i,j} A_{i,j} (v_i - w_j)^2}. \end{aligned}$$

Here the first inequality applies Fact A.20 to the $2t$ terms. The second inequality uses that $A_{k,k} \geq 1/n$ because each node has at least one self-loop and the degree is $\leq n$, $A_{k,k+1} \geq 1/n$ similarly because $1, 2, \ldots, t + 1$ is a path, and $t \leq n$.

(3) See the notes.

QED

We further illustrate the equivalence between edge and spectral expansion via the following "gem" that directly relates the bias of the construction in Theorem 12.7 to the spectral expansion.

Theorem 12.16. Let D be an ϵ-biased distribution on $[2]^n$, and let A be the $2^n \times 2^n$ adjacency matrix of the graph on $[2]^n$ where $A_{a,b} := \mathbb{P}[D = a + b]$ (i.e., the weight of edge $a \to b$ is the prob. of going from a to b when xor-ing a with a sample from X. Then the graph is a spectral ϵ-expander.

Proof. Given a vector v let us write $v(x) = \sum_{S \subseteq [n]} a_S x^S$ according to the hypercube-analysis Fact A.39. . If $\sum_i v_i = 0$ then we have $a_\emptyset = 0$. For minor convenience we think of the input space as $\{-1, 1\}^n$. We have

$$\begin{aligned} (Av)(x) &= \sum_{S \neq \emptyset} a_S \sum_{y \in \{-1,1\}^n} A_{x,y} y^S = \sum_{S \neq \emptyset} a_S \sum_{y \in \{-1,1\}^n} \mathbb{P}[D = x \cdot y] y^S \\ &= \sum_{S \neq \emptyset} a_S \sum_{y \in \{-1,1\}^n} \mathbb{P}[D = y] y^S \cdot x^S = \sum_{S \neq \emptyset} a_S x^S \mathbb{E}[D^S]. \end{aligned}$$

Hence

$$|Av|^2 = \sum_x \sum_{S \neq \emptyset, T \neq \emptyset} a_S a_T x^S x^T \mathbb{E}[D^S] \mathbb{E}[D^T] = 2^n \sum_{S \neq \emptyset} a_S^2 \mathbb{E}[D^S]^2 \leq \epsilon^2 2^n \sum_S a_S^2 = \epsilon^2 |v|^2.$$

QED

Spectral expansion is convenient to analyze a simple *randomized* log-space algorithm for UConn (see Definition 6.21). This *random walk algorithm* decides if s and t are connected as follows. Starting with $v := s$, it moves to a uniformly selected neighbor of v for n^c times. If it ever encounters t, it reports *connected*, otherwise it reports *not-connected*.

Theorem 12.17. The random walk algorithm has error prob. $\leq 1/2$.

Proof. We assume that the graph is n-regular (if it isn't, you can add self loops). Further assume that the graph is connected (since if it isn't the algorithm will never accept by its definition). The graph has edge expansion $\geq 1/n^c$, since at least one edge is leaving any strict subset of nodes. By Theorem 12.15, the graph is a spectral λ-expander for $\lambda \leq 1 - 1/n^c$.

Let A be the random-walk matrix of the graph. Taking a random walk of ℓ steps is the same as taking a step in the graph A^ℓ. Iterating Definition 12.11 we see that starting at a node s the random walk takes us to a distribution which is

$$\leq (1 - 1/n^c)^\ell$$

close to uniform. This distribution puts mass $\geq 1/n^c$ on t, for else the distance to uniform would be too large for $\ell = n^c$.

Hence the random-walk algorithm has a noticeable probability of reporting connected. We can repeat this n^c times to boost the probability to constant, as in section §3.1. This however wouldn't quite be precisely the random-walk algorithm. Instead, we can break a walk of length ℓ into subwalks of length $\sqrt{\ell}$ and argue that each subwalk has a noticeable probability of ending at t from any starting point, so the probability that the entire walk never goes through t is less than, say, $1/2$. **QED**

The random walk algorithm can be implemented in logarithmic space using Rand instructions since we only need to keep track of one node.

12.3 Undirected reachability in L

In this section we give a (deterministic) log-space algorithm to decide connectivity in undirected graphs, thus proving Theorem 6.22. The starting point is that this problem is easy on expanders.

Lemma 12.18. Undirected reachability is in L when the input graph is a constant-degree expander.

Proof. The analysis is similar to that of the random-walk algorithm (Theorem 12.17). The only difference is that the closeness to uniform is now

$$\lambda^\ell$$

for a constant $\lambda < 1$. Hence, it suffices to take $\ell = c_\lambda \log n$ and one of the paths will go through t. Since the graph has constant degree we can enumerate over all paths in logarithmic space. **QED**

How are we going to solve reachability on general graphs? The brilliant idea is

Let's turn the graph into an expander.

One way to boost the spectral expansion is to take walks on the graph, as we have done in Theorem 12.17 and Lemma 12.18. The squared graph is the graph in which edges correspond to paths of length 2 in the original graph; Figure 12.6 shows an example.

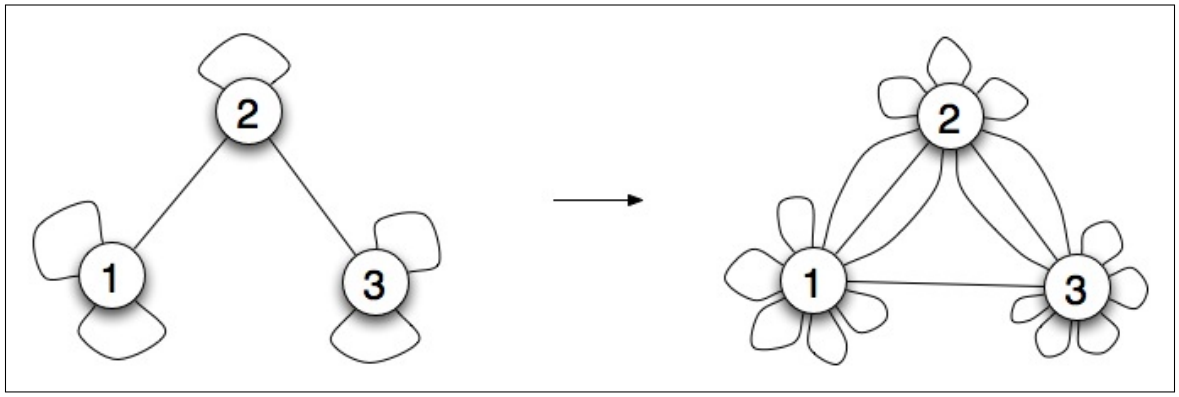


Figure 12.6: Squaring a graph.

If we start with a connected graph with a self-loop at each node and we square it $\ell = c \log n$ we obtain spectral expansion

$$\left(1 - \frac{1}{n^c}\right)^{2^\ell} \leq 1 - c.$$

Although we obtained the desired spectral expansion, if we started with a graph of constant degree d then the degree of the new graph is now $d^{2^\ell} \geq d^n$. This degree is too large: we cannot proceed as in Lemma 12.18 to determine connectivity in logarithmic space, since we cannot even write down an edge in memory.

The solution to this is to alternate taking powers of the graph with replacement product to keep the degree small. We need a new, spectral analysis of this product, because combining the edge analysis we saw in Definition 12.8 with the equivalence in Theorem 12.15 does not seem enough. We show that if H is an expander (constant spectral expansion) then $G \boxtimes H$ has roughly the same expansion of G (while having smaller degree).

Whereas previously (Theorem 12.9) we worked with consistent graphs only, as we said we now have to drop consistency. Indeed, our input graph may not even admit a consistent neighbor function. Instead, we can think of it as being described as a rotation map. We have to modify the definition of the replacement product accordingly: Whereas previously

(Definition 12.8) we connected (x, i) to (y, i) if $x[i] = y$, with d repeated edges, now we connect (x, i) to the output (y, j) of the rotation map. With this change, the replacement product defines a (symmetric) graph. For our application, we only need this change for the outer graph G, the inner graph H will be an expander with a consistent neighbor function.

Theorem 12.19. If G is a spectral $(1 - \epsilon)$-expander and H is a spectral 0.5-expander then $G \boxtimes H$ is a spectral $(1 - c\epsilon)$-expander.

For the proof we start with a useful lemma that shows that a random step in a spectral λ-expander can be seen as going to the uniform distribution with probability $(1 - \lambda)$, and not increasing norm otherwise. The latter is formalized using the *operator norm* $|C|_o$ of a matrix C, which is defined similarly to spectral expansion (cf Definition 12.11 and Exercise 12.13) except that the vector doesn't have to be orthogonal to uniform, see Definition A.45. Recall from Example 12.12 the $n \times n$ matrix J_n with $1/n$ everywhere.

Lemma 12.20. Let A be a regular spectral λ-expander. Then $A = (1 - \lambda)J_n + \lambda C$ where $|C|_o \leq 1$.

Proof. Let $C := (A - (1 - \lambda)J_n)/\lambda$. Let v be a vector and write $v = a \cdot u + w$ where a is a constant, u represents uniform distribution and $u \perp w$. By definition

$$\lambda Cv = Av - (1 - \lambda)Jv = au + Aw - (1 - \lambda)Jau = \lambda au + Aw.$$

Hence using that $Aw \perp u$ since $w \perp u$ and Fact A.37 we have

$$|Cv|^2 = |au + Aw/\lambda|^2 = |au|^2 + |Aw/\lambda|^2 \leq |au|^2 + |w|^2 = |v|^2.$$

QED

Proof of Theorem 12.19. We prove that $(G \boxtimes H)^3$ is a spectral $(1 - c\epsilon\delta^2)$-expander, which suffices for our application. (Eigenvalue theory shows that the spectral expansion of A^t is that of A to the power t, which readily implies the claimed result for $G \boxtimes H$, but we are not discussing eigenvalues.)

We can write the random walk matrix of $G \boxtimes H$ as

$$G \boxtimes H = \frac{1}{2}\hat{A} + \frac{1}{2}I_n \otimes B$$

where $\hat{A}$is the $nD \times nD$ permutation matrix corresponding to G, and B is the $D \times D$ random-walk matrix of H.

Hence we can write the random walk matrix of $(G \boxtimes H)^3$ as

$$(G \boxtimes H)^3 = \left(\frac{1}{2}\hat{A} + \frac{1}{2}I_n \otimes B\right)^3.$$

By Lemma 12.20 we have

$$B = 0.5B' + 0.5J_D$$

where B' is a contraction. Plugging this above we get

$$(G \boxtimes H)^3 = \left(\frac{1}{2}\hat{A} + \frac{1}{2}0.5I_n \otimes (B' + J_D)\right)^3 = \frac{1}{8}\left(\hat{A} + 0.5I_n \otimes B' + 0.5I_n \otimes J_D\right)^3.$$

The matrices $\hat{A}$, I_n, B', J_D are all contractions. By Lemma A.46, multiplying or tensoring two such matrices results again in a contraction. We are going to bound separately only one term in the cube: $(I_n \otimes J_D)\,\hat{A}\,(I_n \otimes J_D)$. This term is a more cumbersome expression for $A \otimes J_D$, which is a spectral $(1-\epsilon)$-expander, because A is.

Putting this together we have for $p \perp u$

$$|\,(G \boxtimes H)^3\, v| \leq (1 - 0.5^2/8)|v| + (0.5^2/8)(1-\epsilon)|v| \leq (1 - c\epsilon)|v|.$$

QED

We can now give the log-space algorithm for undirected reachability.

Proof of Theorem 6.22. We can assume that the input graph G is connected and 3-regular. A node v with degree $D > 3$ can be replaced with a cycle on D nodes; each node in the cycle has one edge leaving the cycle and going to a neighbor of v. We can add self-loops to nodes with degree < 3. Because the graph is connected, by Theorem 12.15 the graph is a spectral $(1 - 1/n^c)$-expander.

For constants b and d to be determined later, we do as follows.

By adding self-loops, we can assume the input graph has degree d^b. Call this G_0.

Let H be a spectral 0.5-expander on d^b nodes with degree $d/2$. For example we can take the expander construction based on small-bias distributions from Theorem 12.16.

We define the sequence of graphs

$$G_k := (G_{k-1} \boxtimes H)^b.$$

Note this is well defined because each G_k has degree d^b for every k. Indeed, this is true for G_0. Then, $G_{k-1} \boxtimes H$ has degree d (twice the degree $d/2$ of H), and so G_k has degree d^b. Note that if G_0 is on m nodes, G_k is on $d^{bk}m$.

The expansion improves up to constant, because if G_{k-1} is a spectral $(1-\epsilon)$-expander, then by Theorem 12.19 $G_{k-1} \boxtimes H$ is a $(1 - c\epsilon)$-expander. Then G_k is a $(1 - c\epsilon)^b$-expander. For $b \geq c$ and $\epsilon \leq c$ the expansion is $(1 - 2\epsilon)$.

Hence if we repeat this $k := c \log n$ times, we obtain constant spectral expansion. On such graphs undirected reachability is solvable in L by Lemma 12.18, as long as we can compute the neighbor function of $G_{c \log n}$ in logarithmic space. The recursive definition of G_k suggests a natural recursive algorithm to compute the rotation map. Each step of the recursion only requires a constant extra amount of memory, for a total space of $c \log n$. Implementing this recursion presents no particular difficulty. The algorithm takes as input k, a node v in G_k, an edge label e in G_k, and computes the rotation map in G_k by overriding (v, e) with its image (v', e'). This is accomplished as follows. By definition of G_k, it suffices to compute the rotation map of $G'_k := G_{k-1} \boxtimes H$; we can then repeat this b times. To compute the

rotation map of G'_k we can write $v = (v', h')$ and examine the edge label. If it corresponds to an edge in H we can compute the corresponding neighbor in constant time, since H has constant size. This only requires changing h'. Otherwise, we run the procedure for G_{k-1}. The algorithm keeps track of k to know which level of the recursion is at. For the base case G_0 the algorithm uses space $c \log n$ to compute the rotation map of G_0 by inspecting the graph. **QED**

As a byproduct, we obtain a construction of spectral expanders. Indeed, we can obtain arbitrarily large graphs with constant degree and constant spectral expansion starting the construction in the proof of Theorem 6.22 from any large enough connected graph.

12.4 What do expanders fool?

Various ways exist to use expanders to construct PRGs. In fact, already the definition of edge expansion Definition 12.2 can be viewed as asserting a PRG-like property for membership in any set. The *expander mixing lemma* is the stronger assertion that the number of edges between two sets is close to what you'd expect in a random graph. To illustrate, consider choosing a graph with N nodes and no self-loops at random by picking each of the $\binom{N}{2}$ edges independently with prob. p. The expected number $p\binom{N}{2}$ equals equals the number $dN/2$ of edges in a d-regular graph without self loops for $p = d/(N-1)$. For any two disjoint sets S and T, the expected number of edges between them would then be $p|S||T|$. The lemma says that in any expander graph the number $|E(S,T)|$ of edges between S and T is close to this expectation:

Lemma 12.21. [Expander mixing lemma] Let G be a d-regular spectral λ-expander on vertex set $[N]$, and let $S, T \subseteq [N]$. Then

$$\left| |E(S,T)| - \frac{d}{N}|S| \cdot |T| \right| \leq \lambda d \sqrt{|S| \cdot |T|}.$$

For the proof see Problem 12.1. The expander mixing Lemma 12.21 is a strengthening of Theorem 12.15, (1), and is in fact a characterization of expanders: Any graph that satisfies the expander mixing lemma is a spectral expander.

Exercise 12.22. In the setting of Lemma 12.21, show that any two sets of size $> \lambda N$ are connected.

More generally, *random walks* on expanders fool various classes of functions applied to indicator functions of sets. A random walk of length k is a sequence $X_1, X_2, \ldots, X_k$ where X_1 is a uniform node, and X_{i+1} is a uniform neighbor of X_i. The expander mixing lemma can be seen as the special case for walks consisting of a single edge and the function checking if the edge crosses S and T. The classes of functions include symmetric functions and AC^0. We give a basic example showing that random walks on expanders fool the And of sets as in Problem 11.5.

Theorem 12.23. Let G be a spectral λ-expander on n nodes. Let B be a subset of $\leq \beta n$ nodes. Let $X_1, X_2, \ldots, X_k$ be a random walk over G. The prob. that $X_i \in B$ for every i is $\leq (\beta + \lambda(1-\beta))^k$.

Proof. Let A be the random-walk matrix of G and also denote by B the matrix that zeroes out the components outside of B, i.e., the diagonal matrix with 1 corresponding to B and 0 elsewhere. We can write and bound the target probability as follows:

$$\begin{aligned}
\left|(BA)^{k-1} Bu\right|_1 &\leq \sqrt{\beta n}\left|(BA)^{k-1} Bu\right| \text{ by } Fact\text{??} \\
&= \sqrt{\beta n}\left|(BAB)^{k-1} Bu\right| \text{ because } BB = B \\
&\leq \sqrt{\beta n}|\,(BAB)^{k-1}\,|_o |Bu| \text{ by } Definition A.45 \\
&\leq \sqrt{\beta n}|BAB|_o^{k-1}\sqrt{\frac{\beta}{n}} \text{ by definition of } B \text{ and } Lemma A.46 \\
&= \beta |BAB|_o^{k-1} \\
&\leq \beta(\beta + \lambda(1-\beta))^{k-1} \\
&\leq (\beta + \lambda(1-\beta))^k.
\end{aligned}$$

The bound on the operator norm $|BAB|_o$ in the second to last inequality is shown as follows. Write $A = (1-\lambda)J_n + \lambda C$ according to Lemma 12.20 (where recall $|C|_o \leq 1$ and J_n has $1/n$ everywhere. So by Lemma A.46 we have

$$|BAB|_o \leq (1-\lambda)|BJB|_o + \lambda.$$

The proof is complete by noting that $|BJB|_o \leq \beta$. To show this, note that the top left square of dimension βn of BJB is the matrix M which has $1/n$ everywhere, and BJB has 0 everywhere else. Hence it suffices to bound $|Mx|$ for a vector x with βn coordinates. We have $Mx = u|x|_1$, where u has βn coordinates. Hence $|Mx| = |u| \cdot |x|_1 \leq |u| \cdot \sqrt{\beta n}|x| = \beta|x|$, using Fact A.20. **QED**

For more on the properties of expander walks see the notes.

12.5 On the proof of the PCP theorem

In this section we give an overview of the proof of the PCP Theorem 4.33. It is a convenient to work with a natural generalization of the gap-3Sat Definition 4.32 where the variables take values in larger alphabets and the clauses become arbitrary functions.

Definition 12.24. A *constraint satisfaction problem* (CSP) instance with n variables, alphabet size W, m constraints, and arity q is a q-local map

$$\phi : [W]^n \to [2]^m.$$

Each output bit is called a *constraint*. The *value* of the instance, denoted $\mathrm{val}(\phi)$, is the maximum over $x \in [W]^n$ of the relative weight $\sum_i \phi(x)/m$ of $\phi(x)$. The instance is *satisfiable* if $\mathrm{val}(\phi) = 1$.

3Sat is the special case where $q = 3$, $W = 2$, and each constraint is a clause on 3 variables. The main claim is for $W = 2$, but the proof requires an excursion to $W > 2$.

Our goal is to efficiently reduce a CSP instance ϕ with m constraints to a CSP instance ϕ' such that:

$$\mathrm{val}(\phi) = 1 \Rightarrow \mathrm{val}(\phi') = 1,$$
$$\mathrm{val}(\phi) \leq 1 - 1/m \Rightarrow \mathrm{val}(\phi') \leq 1 - c.$$

This challenge, and the way we are going to tackle it, are similar to the way in which we turned an abritrary graph into an expander in section §12.3 (to prove Theorem 6.22). There we started with a graph with expansion $1 - 1/n^c$ and "moderately, bravely" obtained a graph with expansion $1 - c$ by iterating a basic operation which improves the expansion. In the context of CSPs we shall iterate the following lemma.

Lemma 12.25. There is a constant $q \geq 3$ such that we can map in FP a CSP $\phi : [2]^n \to [2]^m$ with arity q to a CSP $\phi' : [2]^{n'} \to [2]^{m'}$ with arity q where
(1) $\mathrm{val}(\phi) = 1 \Rightarrow \mathrm{val}(\phi') = 1$,
(2) $\mathrm{val}(\phi) \leq 1 - \epsilon \Rightarrow \mathrm{val}(\phi') \leq 1 - 2\epsilon$,whenever $\epsilon \leq c$, and
(3) $m' \leq cm$ and $n' \leq cn$,

Proof of the PCP Theorem 4.33 from 12.25.. We view a 3Sat instance with m clauses as a CSP ϕ with arity 3 and alphabet size 2. We repeatedly apply 12.25 $\log m - c$ times. Denote by ϕ' the resulting CSP. If ϕ is satisfiable then so is ϕ'. If ϕ is unsatisfiable then its value is $\leq 1 - 1/m$, and so the value of ϕ' is $\leq 1 - c$. Because each application of the lemma only increases the number of constraints (and variables) by a constant factor, and we apply the lemma $\leq \log m$ times, the reduction runs in power time.

There remains to reduce ϕ' to a 3Sat instance. Each constraint can be computed by a circuit of size c_q (by Theorem 2.5). By Theorem 5.1 we can replace each circuit with a 3CNF using c_q additional variables. Call ϕ'' the resulting 3CNF. If ϕ' is satisfiable then so is ϕ''. On the other hand, for any assignment to the variables in ϕ', any constraint of ϕ' that is not satisfied gives at least one constraint in ϕ'' that is not satisfied, for any choice of the additional variables. Since the number of constraints in ϕ'' is at most c_q times that of ϕ', the result follows. **QED**

Continuing the parallel with expander graphs, the basic operation in section §12.3 was in turn made of two steps. The first, powering, improves the expansion at the cost of degree, the second, replacement product, reduces the degree without sacrificing the expansion too much. Before we could start iterating the basic operation, there was also an initialization step to make the graph constant degree.

We will have three corresponding steps in the basic operation for CSPs. We only give the high-level ideas of these steps and refer to the notes for full proofs.

In the first step, we turn the CSP into an expander graph. More specifically, we reduce ϕ to a certain CSP ϕ' with arity 2 over a larger alphabet of size 2^q. Because ϕ' has arity 2, we can think of its *constraint graph* where edges correspond to constraints. We require that this graph is a d-regular 0.9-spectral expander, for a constant d. (Moreover, as in Theorem 12.15, we require that each node has $d/2$ self-loops.) The larger alphabet size will allow us to accommodate in a single variable the q input bits to a constraint in ϕ. The consistency can be verified with additional constraints. In section §12.3 we turned a graph with arbitrary degree into a 3-regular graph by replacing vertices of larger degree by cycles. Here we also need to guarantee that the new graph is an expander, so we replace vertices by corresponding expanders. In addition to this "local" change, we also make a global change by adding a number of "dummy" constraints that make the entire graph an expander. This step makes the value of the CSP worse, but this will be recouped in the next step.

The second step is a kind of "powering" for CSPs on expander graphs. We turn ϕ' into another CSP ϕ'' over the same variables but over a larger alphabet. The value of a variable y in ϕ'' corresponds to the values of all the variables in ϕ' which are in a small neighborhood of y in ϕ'. Relying on the expansion of the graph, one proves the key fact that this step improves the value of the CSP. This improvement is large enough to withstand the losses in the first and third steps.

The third and final step reduces the alphabet of the CSP to binary, while increasing the arity back to the arity q of ϕ and making the value a little worse.

12.6 Problems

Problem 12.1. Prove Lemma 12.21. Hint: Use Lemma 12.20.

Problem 12.2. Let G be a d-regular spectral λ-expander on nodes $[n]$. Let S, T be two disjoint subsets of the nodes of size $\geq n/3$. Prove that there is an *induced matching* M between S and T of size $\geq c_{d,\lambda} n$. An induced matching is a subset of $E(S,T)$ whose endpoints are not connected by any other edge in G. Hint: Exercise 12.22.

Problem 12.3. Assuming that multiplication is in quasi-linear time, prove that the endpoint of a random walk on the 8-regular graph G on nodes $\mathbb{Z}_M \times \mathbb{Z}_M$ in Example 12.4 can be computed in quasi-linear time. Use this, and Theorem 12.23, and the fact that the graph is a spectral expander (for which we refer to the notes) to give another proof of Item (2) in Theorem 5.36.

Hint: Write the neighbors of $(x, y) \in \mathbb{Z}_M \times \mathbb{Z}_M$ as $M(x, y, 1)$ for 3×3 matrices M.

12.7 Notes

Expanders can be traced back to [218] (cf [150]). The existence of edge expanders (Theorem 12.6) is proved in [283]. For the analysis of the expanders in Example 12.4 see Section 11.1.2 in [180]. Early uses of expanders in pseudorandomness include [205, 14]. The replacement

product is a natural operation that is often performed on graphs. The expansion of this product (for specific graphs) was already studied in [149]. Our construction of edge expanders is a variant of [26], which proves Theorem 12.9. The random walk algorithm and its analysis are from [16]. In [294] a new construction of expander graphs was presented using the *zig-zag product*, which is closely related to the replacement product. This construction is iterative and similar to the one we obtain from the proof of Theorem 6.22, which was inspired by it. The analysis of expansion in [294] avoids some mathematical result used in previous constructions such as those in Example 12.4 . The expansion of the replacement product was simultaneously and independently obtained in [243] for different purposes.

The proof of Theorem 6.22 is from [292]. An alternative proof is in [298]. Each proof has its advantages. [298] uses a single operation (*derandomized squaring,* similar to the combination of powering and replacement product) and keeps the vertex set the same as the original graph. The cons is that it needs expanders of any size (as opposed to constant size) and *directed* graphs (derandomized squaring turns undirected graphs into directed graphs). Because of this, [298] is perhaps more similar in spirit to a PRG (specifically, the PRG in [188]).

Around the same time as this chapter was written, and for different reasons, the works [76, 75] gave constructions of expanders and a proof that undirected reachability is in L avoiding the use of eigenvalues. The constructions are different. [76, 75] rely on the results by [248] (which they re-render as Lemma 12 in [76]) and which is here Theorem 12.15.(3). This result is somewhat technical and we avoid it. Their construction of expanders in [76] uses iterated recursion and several graph operations, whereas we apply replacement product a constant number of times (in combination with suitable expander constructions). For the undirected reachability result they follow the proof in [298] while we followed [292] (these two proofs are compared above).

That the expander mixing Lemma 12.21 characterizes expansion was proved in [61].

For the fooling properties of random walks on expanders see [143]. The fact that expander walks fool the And of sets, Theorem 12.23, is from [201].

For more on expanders see the monograph [180].

Problem 12.3 is from [105], cf [364].

Chapter 13

Communication

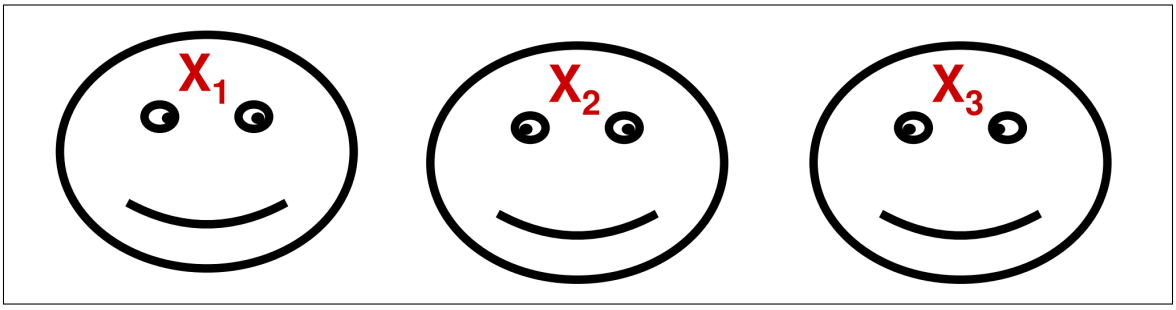


Figure 13.1: Three parties with inputs on their foreheads. Each party can see the other two parties' foreheads, but not their own.

Communication complexity is the study of... *rectangles.* Alright, let me try to make the subject sound more appealing to the uninitiated; though the ultimate goal is to convince them that the study of rectangles is fascinating. Communication complexity is the study of the amount of information that needs to be exchanged among two or more parties (a.k.a. players) which are interested in reaching a computational goal. This goal is a function of several inputs which are *distributed among the parties* in various ways. Another critical difference with interactive proofs (section §10.3) is that in communication complexity the parties *cooperate.* By contrast, the setting of proofs is *adversarial:* The verifier may be interacting with a malicious party and their main goal is to verify the prover's claims. Also, in communication complexity we can be more basic and disregard the computational model: we bestow unlimited computational power on the parties, viewing them simply as functions. The measure of complexity will only be the amount of communication bits exchanged.

We begin in section §13.1 with protocols among two parties. Then in section §13.2 we move to more parties.

13.1 Two parties

We start with the model in which there are only 2 parties, A and B. Their task is to compute a function $f : X \times Y \to Z$ on an input (x, y) where A only knows x and B only knows. The parties A and B engage in a communication protocol and exchange bits, until some party computes the output $z = f(x, y)$. More formally, we can define a communication protocol as a tree. Each internal node v in the tree is labeled with a function f_v mapping $X \to [2]^{b_v}$ or $Y \to [2]^{b_v}$. The first case corresponds to A speaking, the second to B. The message length is b_v which is the number of children of v. Each leal ℓ is labeled with an output $z \in Z$. Party A speaks first, so the domain of the function at the root is X. The protocol computes f if for every (x, y) following the path according to the functions leads to a leaf labeled with $f(x, y)$.

Definition 13.1. The *communication complexity* of a protocol is the maximum over all leaves of the sum of the b_v over the nodes v in the path. The communication complexity of a function is the minimum communication complexity of a protocol computing the function.

Note that the protocol specifies in advance the exact number of bits exchanged b_v at every node v. We can transform any protocol into one in which the parties alternate sending one bit, at a modest cost. However, this transformation does not preserve the *rounds* of the protocol, an important complexity measure which is discussed below and is the analogue of alternation.

13.1.1 The rectangle method and the Equality function

Consider the function Equality : $[2]^n \times [2]^n \to [2]$, define as Equality$(x, y) = 1 \Leftrightarrow x = y$. Trivially, Equality can be computed with communication $n + 1$: A sends her input to B, who then outputs the answer. The same trivial upper bound holds for any function $f : [2]^n \times [2]^n \to [2]$. For Equality it is tight:

Theorem 13.2. The communication complexity of equality is $\geq n$.

Before proving this theorem, we cover some properties of protocols.

Definition 13.3. A *rectangle* in $X \times Y$ is a subset $R \subseteq X \times Y$ such that $R = S \times T$ for some $S \subseteq X$ and $T \subseteq Y$. Equivalently, $R \subseteq X \times Y$ is a rectangle if whenever $\{(x, y), (x', y')\} \subseteq R$ then we also have $\{(x, y'), (x', y)\} \subseteq R$.

Exercise 13.4. Prove the equivalence.

The connection between rectangles and protocols is the following.

Lemma 13.5. Let v be a node in a communication protocol. The set of inputs that lead to v is a rectangle.

Proof. Suppose this is true for a node v labeled w.l.o.g. with $f : X \to [2]^b$, and let $R = S \times T$ be the corresponding rectangle. The inputs leading to a child w of v form a rectangle $S_w \times T$, where S_w are the elements in S on which f points to w. **QED**

Applying this the leaves we can readily partition the input space in *f-monochromatic rectangles*, i.e., rectangles where the value of f is constant.

Corollary 13.6. Suppose $f : X \times Y \to Z$ has communication complexity d. Then there is a partition of $X \times Y$ in 2^d f-monochromatic rectangles.

Equality, seen as a matrix, is the identity. figure 13.2 shows two ways to partition it in monochromatic rectangles. Intuitively, because the ones are only on the diagonal, we need many rectangles in any monochromatic partition. The lower bound for equality formalizes this.

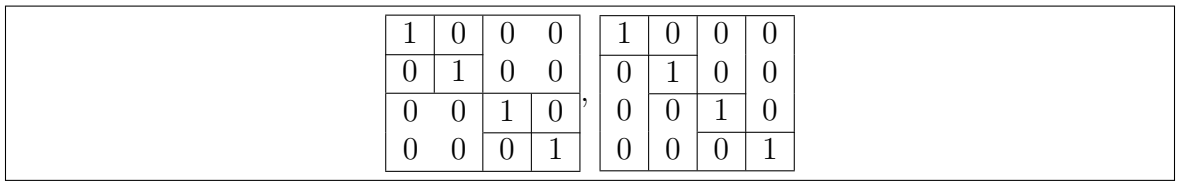


Figure 13.2: Two ways to partition Equality : $[4] \times [4] \to [2]$ in monochromatic rectangles.

Proof of Theorem 13.2. Suppose P is a protocol with communication d for Equality. By Corollary 13.6 P induces a partition of $X \times Y$ in 2^d Equality-monochromatic rectangles. Consider the 2^n inputs (e, e) where $e \in \{0,1\}^n$. Observe that no equality-monochromatic rectangle can contain both (e, e) and (b, b) if $e \neq b$, for else (e, b) is in the rectangle, but since $e \neq b$ this cannot be equality-monochromatic. Since the rectangles must cover all of the 2^n inputs (e, e), we need $\geq 2^n$ rectangles. Hence $2^d \geq 2^n$. **QED**

13.1.2 Rounds: Pointer chasing

Here we present a new proof technique and use it to prove an impossibility result for the *pointer chasing* function. The input to the k-pointer-chasing function consists of two arrays $x, y \in [n]^n$ of pointers. The output is the pointer reached after following k pointers, starting at $y[0]$. For example, for $i = 0, 1, 2, 3, \ldots$ the outputs of i-pointer chasing are $y[0]$,$x[y[0]]$, $y[x[y[0]]]$, $x[y[x[y[0]]]]$, ... The impossibility results in this section apply even if we just need to compute one bit of the pointer, say the first; for clarity of exposition we consider computing the entire pointer.

Theorem 13.7. The communication complexity of $n/8$-pointer-chasing is $\geq cn$.

In fact, we prove a refined result which goes hand-in-hand with the proof technique. We show that k-pointer-chasing requires large communication for *k-round protocols*. The number of rounds in a protocol is the number of *alternations* between parties. So in a 0

round protocol, only party A communicates (and computes the output). In a 1 round, party A sends a message to party B who then computes the output, and so on.

Let us slowly build intuition for the proof technique.

The case $k = 0$ rounds. Here Party A is supposed to compute $y[0]$ from their input x, with no communication. This is clearly impossible, since A does not know $y[0]$. In particular, we can pick any $y[0]$ that's different from A's output on their input x. This is impossible regardless of the communication.

The case $k = 1$ rounds. Here Party A sends a message of t bits to B, who is then supposed to output $x[y[0]]$. This $k = 1$ case is already interesting. If A sends the entire x to B, then B can compute the answer. So the impossibility result depends on t. The idea is to reduce to the case $k = 0$ by fixing the most likely message of A and considering the corresponding set $X_0 \subseteq X$ of inputs. Because A sends t bits, the most likely message is sent by at least $|X_0| \geq |X|/2^t$ messages, hence if t is small then X_0 is fairly large. We'd like to say that most pointers $x[i]$ in X_0 are not fixed in X_0. Suppose we do have such a pointer i. Then we can just set $y[0] := i$ and we have reduced to the $k = 0$ case, except the roles of A and B are swapped, and the pointers are permuted. Specifically, Party B needs to compute $x[i]$ from their input y and A's message. But since given all this information $x[i]$ is not fixed, they again cannot do that.

Finding alive pointers is where the bound on t kicks in. If t is very large, the set X_0 could consist of a single array, and all the pointers would be fixed. But if A sends few bits, many pointers are unfixed, for else the size of X_0 would be too small.

To generalize this idea to $k = 2, 3, \ldots$ rounds it is not sufficient to distinguish between fixed and unfixed pointers. This is because we need to iteratively be able set a pointer to point to another "good" pointer, which requires more than just being unfixed (since the number of good pointers may be small). Instead, we consider *alive* pointers that have somewhat noticeable entropy, and we keep track of how many pointers are fixed. This is formalized in the following theorem, from which Theorem 13.7 follows for $X = Y = [n]^n$ and $F_X = F_Y = \emptyset$.

Theorem 13.8. There is no k-round protocol with communication s and sets $X, Y \subseteq [n]^n$ and $F_X, F_Y \subseteq [n]$ such that:

(0) The protocol computes k-pointer-chasing on every input in $X \times Y$,

(1) The F_X pointers in X are fixed, i.e., $\forall i \in F_X \exists v \forall x \in X, x[i] = v$, and the same for Y,

(2) The *unfixed density* of X, defined as $|X|/n^{n-|F_X|}$, is $\geq 2^{2s-n/4+|F_X|}$, and the same for Y,

(3) $y[0]$ is *alive*, defined as $\mathbb{P}_{y \in Y}[y[0] = v] < 2/n$ for every $v \in [n]$.

Proof. Proceed by induction on k, a.k.a. round elimination. For the base case $k = 0$, we can fix A's input x. But since $y[0]$ is alive, the probability that the output is correct is

$< (2/n) \cdot n/2 < 1$. (This bound is strong enough to rule out even boolean pointer chasing, where the output is say the first bit of the pointer.)

The induction step is by contrapositive. Assuming there is such a protocol and there are such sets, we construct a $(k-1)$-round protocol and sets violating the inductive assumption. Suppose A sends t bits as their first message. Fix the most likely message, and let $X_0 \subseteq X$ be the set of $\geq 2^{-t}|X|$ corresponding strings. Next is the key idea. If there is a pointer $i \in [n] - F_X$ which is not alive in X_0, fix it to its most likely value, call $X_1 \subseteq X_0$ the corresponding subset, and let $F_{X_1} := F_X \cup \{i\}$. Note that the unfixed density increases by a factor 2 since

$$\frac{|X_1|}{n^{n-|F_{X_1}|}} \geq \frac{2}{n}\frac{|X_0|}{n^{n-|F_X|-1}} = 2\frac{|X_0|}{n^{n-|F_X|}}.$$

Continue fixing until every unfixed pointer is alive, and call X', $F_{X'}$ the resulting sets. The unfixed density of X' is then

$$\frac{|X'|}{n^{n-|F_{X'}|}} \geq \frac{2^{-t}|X|}{n^{n-|F_X|}} 2^{|F_{X'}|-|F_X|} \geq 2^{2s-n/4+|F_X|-t+|F_{X'}|-|F_X|} = 2^{2(s-t)-n/4+|F_{X'}|}.$$

Now note that $|F_{X'}| \leq n/4$ because the density cannot be larger than 1. We use this to analyze the other side. Because $y[0]$ is alive,

$$\mathbb{P}_{y \in Y}[y[0] \in F_{X'}] \leq \frac{2|F_{X'}|}{n} \leq \frac{1}{2}.$$

So there is an alive pointer $x'[i]$ such that $\mathbb{P}[y[0] = i] \geq 1/(2n)$. Let $Y' \subseteq Y$ be the corresponding subset with $y[0] = i$, and let $F_{Y'} := F_Y \cup \{0\}$. The unfixed density of Y' is

$$\frac{|Y'|}{n^{n-F_{|Y'|}}} \geq \frac{|Y|/2n}{n^{n-|F_Y|-1}} = \frac{|Y|/2}{n^{n-|F_Y|}} \geq 2^{2s-n/4+|F_Y|-1} = 2^{2s-n/4+|F_{Y'}|-2} \geq 2^{2(s-t)-n/4+|F_{Y'}|}$$

since $t \geq 1$. This gives a $(k-1)$-round protocol where B goes first that computes $(k-1)$-pointer-chasing where the first pointer is $x[i]$. We can swap the parties A and B and permute pointers so that $y[0]$ is the first pointer. **QED**

13.1.3 Randomness

We define randomized protocols as a distribution on protocols, similarly to the randomized polynomials in Definition 9.7.

Definition 13.9. A *randomized protocol* P with communication d is a distribution on protocols with communication d. The randomized communication complexity of a function $f : X \times Y \to Z$ with error ϵ is the minimum communication of a randomized protocol that computes f on every input (x, y) with error $\leq \epsilon$. If the error is not specified it is taken to be 1/3.

We give a couple of examples of the power of randomness in communication. The first and most dramatic example is Equality, for which the communication complexity drops from largest (linear) to smallest (constant).

Theorem 13.10. Equality has randomized communication complexity $\leq c \log 1/\epsilon$ with error ϵ, for any $\epsilon \leq 1/2$.

Proof. We use the random parity principle Fact 3.6. To compute Equality : $[2]^n \times [2]^n \to [2]$ on input (x, y) we pick a uniform string $s \in [2]^n$. If $x = y$ then also their inner products with s will be equal. Otherwise they will be different with prob. $\geq 1/2$. We can amplify this one-sided error probability as in section §3.1. **QED**

As another example consider the Greater-Than function, where the parties wish to determine if $x > y$ as n-bit integers.

Theorem 13.11. The randomized communication complexity of Greater-Than is $\leq c \log n$.

Proof. We perform binary search to find the most significant bit where x and y differ. Each comparison during this binary search corresponds to an instance of Equality (Theorem 13.10). The naive way to implement this search is to set the error to $\leq c/\log n$ in Theorem 13.10. This allows us to take a union bound over all $\leq c \log n$ comparisons performed during the binary search.

However, each Equality would require communication $\geq \log \log n$, resulting in a total communication of $c(\log n) \log \log n$.

To remove the $\log \log n$ term we set the error to constant and perform *binary search with noisy comparisons*. A random-walk-with-backtrack algorithm shows that again $c \log n$ comparisons suffice, leading to the result. The idea is to start each recursive call with a check that the target element is contained in the current interval, and if not backtrack. We leave the details as Problem 13.1. **QED**

The main technique for proving negative results for randomized communication is to prove a correlation bound for *deterministic* protocols for a suitable distribution, which recall is the "easy" direction of the equivalence in Corollary 8.15. One can apply this technique to prove that the upper bound for Greater-Than in Theorem 13.11 is tight.

Theorem 13.12. The randomized communication complexity of greater-than is $\geq c \log n$.

For the proof see the notes.

Another important example is the *disjointness function* Disj : $[2]^n \times [2]^n \to [2]$ defined as $\text{Disj}(x, y) = \vee_{i \in [n]} x_i \wedge y_i$. It asks to determine if x and y, viewed as subsets of $[n]$, (do not) intersect. This function is of central importance pretty much for the same reason that 3Sat is: Its simple structure makes it excellent for reductions.

Theorem 13.13. The randomized communication complexity of Disj is $\geq cn$.

While we refer to the notes for the proof, the hard distribution is defined as follows for $n = 4m-1$. First pick a uniform partition of $[n]$ into $(P, Q, \{i\})$ where P (and Q) is a uniform set of size $2m - 1$. Now let x (resp., y) be a uniform subset of $P \bigcup \{i\}$ (resp. $Q \bigcup \{i\}$) of size m. In particular, the intersection is either empty or a singleton. Note that the distribution is not product; it is known that the communication is $\leq c\sqrt{n}$ on product distributions.

Yet another important example is the *inner product* function $\mathrm{IP}(x, y) := \sum x_i y_i \mod 2$. The randomized communication complexity of IP is $\geq cn$. This is a special case of Theorem 13.15, proved below. Here the hard distribution is simply uniform.

13.1.4 Arbitrary partition

In our discussion so far, the partition of the input among the parties was fixed in advance. It is natural instead to let the parties pick a partition, i.e., for a function $f : [2]^{2n} \to [2]$ ask what is the minimum communication complexity over some partition of the $2n$ input bits into two sets of size n. In all the impossibility results we have seen so far, the specific choice of the partition was essential, because w.r.t. another choice the communication complexity becomes trivial:

Exercise 13.14. Show that for each of the functions Equality, Greater-Than, Disj, and IP, viewed as functions on $2n$ bits, there is a partition of the input bits in two sets of size n such that the function can be computed with constant (deterministic) communication.

By contrast, there are functions which require linear communication for every partition, see Problem 13.2.

13.2 Number-on-forehead

There are various ways in which we can generalize the 2-party model of communication complexity to $k > 2$ parties whose goal is computing a k-argument function $f(x_1, \ldots, x_k)$. The obvious generalization is to let x_i be the input to Party i. This model, known as "number-in-hand," is useful in some scenarios. But we focus on a different, fascinating model which has an unexpected variety of applications: the "number-on-forehead" model. Here, the twist is that Party i knows *all inputs except* x_i, which we can imagine being placed on their forehead. This overlap of the information makes the model extremely versatile. See figure 13.1 for an illustration for $k = 3$ parties.

Formally, we define a k-party protocol to be a tree similarly to the 2-party case (section §13.1). The difference is that each node is now labeled with a party $i \in [k]$ as well as a function which depends on all the k arguments except x_i.

In section 13.2.1 we prove impossibility results for up to $k \leq (1 - c) \log n$ parties. The grand challenge for this number-on-forehead model is thus to give an explicit function $f : ([2]^n)^k \to [2]$ that requires large communication even for more parties, say $k = 2 \log n$, or k power-logarithmic in n. Such a result would have many applications. In section §13.2.2 we present one such application as well as several other examples of the power of protocols with many parties.

13.2.1 Generalized inner product

In this section we prove an impossibility result for computing the generalized inner product function GIP : $([2]^n)^k \to [2]$:

$$\mathrm{GIP}(x_1, \ldots, x_k) := \sum_{i=1}^{n} \bigwedge_{j=1}^{k} (x_j)_i \mod 2.$$

As discussed in section 13.1.3, our strategy is to bound even the correlation between GIP and k-party protocols exchanging d bits, denoted here $\mathrm{Cor}(\mathrm{GIP}, d\text{-bit } k\text{-party})$. Recall (Definition 8.11) that correlation is defined as the maximum of $|\mathbb{E}_x e[\mathrm{GIP}(x) + f(x)]|$ over any k-party protocol f with communication d, where x is uniform and $e(z) := (-1)^z$.

Theorem 13.15. $\mathrm{Cor}(\mathrm{GIP}, d\text{-bit } k\text{-party}) \leq 2^d \cdot 2^{-cn/4^k}$.

By (the easy direction of) Corollary 8.15, this Theorem 13.15 implies that the randomized communication complexity of GIP is large. Let us spell out again this implication: Having randomized communication complexity $\leq d$ with small error means that there is a distribution of protocols with communication d s.t. on every input x, a randomly selected protocol achieves error $\leq \epsilon$. From this, we can average over x, and then fix a protocol to obtain small correlation. Because we prove next that small correlation is impossible, it follows that the randomized communication complexity is large too.

To prove Theorem 13.15 we associate to any function a quantity $R(f) \in \mathbb{R}$ enjoying the following two lemmas:

Lemma 13.16. $\mathrm{Cor}(f, d\text{-bit } k\text{-party}) \leq 2^d \cdot R(f)^{1/2^k}$, for any $f : X_1 \times ... \times X_k \to [2]$.

Lemma 13.17. $R(\mathrm{GIP}) \leq 2^{-cn/2^k}$.

The combination of these two facts proves Theorem 13.15.

Intuition for $R(f)$: Think of $k = 2$; we saw that any 2-party d-bit protocol partitions the inputs in 2^d f-monochromatic rectangles. How about we check how well f can be so partitioned? Instead of picking an arbitrary rectangle, let us pick one in which each side has length 2, and see how balanced the function is there. If a "good" partition exists, with somewhat high probability our little rectangle should fall in a monochromatic rectangle, and we should always get the same values of f. Otherwise, we should get mixed values of f.

Specifically, for $k = 2$,

$$R(f) := \mathbb{E}_{\substack{x_1^0, x_2^0 \\ x_1^1, x_2^1}} e\left[f(x_1^0, x_2^0) + f(x_1^0, x_2^1) + f(x_1^1, x_2^0) + f(x_1^1, x_2^1)\right] \in \mathbb{R}.$$

In general, for any k:

$$R(f) := \mathop{\mathbb{E}e}_{\substack{x_1^0, \ldots, x_k^0 \\ x_1^1, \ldots, x_k^1}} \left[\sum_{\varepsilon_1, \ldots, \varepsilon_k \in [2]} f(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k})\right] \in \mathbb{R}.$$

Exercise 13.18. Prove:

$R(f) \geq 0$ for every f.

$R(f) = 1$ for constant f.

$\mathbb{E}_F R(F) = 1 - (1 - 2^{-n})^k \leq k/2^n$ for uniform $F : ([2]^n)^k \to [2]$. Hint: The inequality is Fact A.8.

The proof of Lemma 13.16 is via a sequence of claims which put together enable the following chain of inequalities (Cor* and $*$-protocols will be defined below).

$$\begin{aligned} \mathrm{Cor}(f, d-\text{bit}) &\leq 2^d \cdot \mathrm{Cor}^*(f) && \text{Claim 13.21} \\ &\leq 2^d R(f+p)^{1/2^k} && \text{Claim 13.22, for some } *\text{-protocol } p \\ &= 2^d R(f)^{1/2^k}. && \text{Claim 13.23} \end{aligned}$$

First we need suitable extensions of the theory of rectangles that we saw for 2 parties. Recall we saw that a 2-party protocol partitions the input in monochromatic rectangles (section §13.1). The extension of this fact to k parties is via *cylinder intersections*.

Definition 13.19. A function $g_i : X_1 \times \ldots \times X_k \to [2]$ is a *cylinder* in the i-th dimension if it does not depend on x_i. A set $S \subseteq X_1 \times \ldots \times X_k$ is a *cylinder intersection* if $\exists$ cylinders $g_1, \ldots, g_k$ such that $S = \{x : \prod g_i(x) = 1\}$.

Note a rectangle is a cylinder intersection with $k = 2$.

Claim 13.20. Any d-bit k-party protocol for $f : ([2]^n)^k \to [2]$ partitions the inputs in 2^d f-monochromatic cylinder intersections.

The proof is very similar to the case of rectangles and is left as exercise.

Using the notion of cylinder intersections we can now relate an arbitrary protocol to a special class of protocols called $*$-protocols. Each $*$-protocol protocol p^* can be written as $p^*(x) = \sum_i g_i(x) \mod 2$, where g_i is a cylinder in i-th dimension. This corresponds to each party sending just one bit independently of the others, and the output of the protocol being the XOR of the bits. Note the communication parameter is not present anymore. We write Cor^* for the corresponding correlation, where k is given by the context.

Claim 13.21. $\mathrm{Cor}(f, d-\text{bit}) \leq 2^d \cdot \mathrm{Cor}^*(f)$.

Proof. We use a general trick to turn an And into an Xor, here turning a cylinder intersection $\prod_i g_i(x) = 1$ into a $*$-protocol $\sum g_i(x) \mod 2$. Fix any d-bit protocol, let $\{x : \prod_i g_i^1(x) = 1\}, \ldots, \{x : \prod_i g_i^D(x) = 1\}$ be the corresponding $D := 2^d$ f-monochromatic cylinder intersections (by the previous claim). Observe that for a fixed x,

$$\mathop{\mathbb{E}}_{y_1,\ldots,y_k \in \{-1,1\}} \left[(y_1)^{1+g_1(x)} \cdot (y_2)^{1+g_2(x)} \cdot \ldots \cdot (y_k)^{1+g_k(x)} \right] = \begin{cases} 1 & \text{if } \exists i : g_i(x) = 0 \\ 0 & \text{if } \forall i : g_i(x) = 1. \end{cases}$$

Therefore,

$$e(p(x)) = \sum_{i=1}^{D} r(i) \mathop{\mathbb{E}}_{y_1,\ldots,y_k \in \{-1,1\}} \left[(y_1)^{1+g_1^i(x)} \cdot (y_2)^{1+g_2^i(x)} \cdot \ldots \cdot (y_k)^{1+g_k^i(x)} \right]$$

where $r(i) \in \{-1, 1\}$ is the value of the protocol on the i-th cylinder intersection. Note that for any x exactly one expectation will be 1, the one corresponding to the cylinder intersection where x lands. So we have:

$$\begin{aligned}
&\mathbb{E} e[f(x) + p(x)] \\
&= \mathbb{E}_x[e(f(x)) \cdot e(p(x))] \\
&= \mathbb{E}_x \left[e(f(x)) \cdot \textstyle\sum_{i=1}^{D} r(i) \mathop{\mathbb{E}}_{y_1,\ldots,y_k \in \{-1,1\}} \left[(y_1)^{1+g_1^i(x)} \cdot (y_2)^{1+g_2^i(x)} \cdot \ldots \cdot (y_k)^{1+g_k^i(x)} \right] \right] \\
&= \textstyle\sum_{i=1}^{D} \mathbb{E}_{x,y_1,\ldots,y_k \in \{-1,1\}} \left[e(f(x)) \cdot r(i) \cdot (y_1)^{1+g_1^i(x)} \cdot (y_2)^{1+g_2^i(x)} \cdot \ldots \cdot (y_k)^{1+g_k^i(x)} \right] \\
&\leq D \cdot \mathbb{E}_{x,y_1,\ldots,y_k \in \{-1,1\}} \left[e(f(x)) \cdot r(i) \cdot (y_1)^{1+g_1^{i*}(x)} \cdot (y_2)^{1+g_2^{i*}(x)} \cdot \ldots \cdot (y_k)^{1+g_k^{i*}(x)} \right],
\end{aligned}$$

where $i*$ is the value of i that gives the largest summand. Now fix $y_1, \ldots, y_k$ to maximize the expectation, and let $J \subseteq \{1, \ldots, k\}$ be the indices corresponding to $y_j = -1$, i.e., $j \in J \Rightarrow y_j = -1$. The last expression above is

$$\begin{aligned}
&D \cdot \mathbb{E}_x \left[e(f(x)) \cdot \prod_{j \in J} (-1)^{1+g_j^{i*}(x)} \right] \\
&= D \cdot \mathbb{E}_x e \left[f(x) + \sum_{j \in J} \left(1 + g_j^{i*}(x)\right) \right] \\
&\leq D \cdot \mathrm{Cor}^*(f).
\end{aligned}$$

QED

Claim 13.22. $\mathbb{E} e_x[g(x)] \leq R(g)^{1/2^k}$ for every function $g := X_1 \times \ldots \times X_k \to [2]$.

Proof. Recall that for every random variable X: $\mathbb{E}[X^2] \geq \mathbb{E}[X]^2$, Fact A.20. Also recall that if X, X' are independent then $\mathbb{E}[X \cdot X'] = \mathbb{E}[X] \cdot \mathbb{E}[X']$. Applying the "squaring trick," see section A.6, we get:

$$\mathbb{E}_{x_1,\ldots,x_k} e[g(x_1, \ldots, x_k)]^2 = \mathbb{E}_{x_1,\ldots,x_{k-1}} [\mathbb{E}_{x_k} e[g(x_1, \ldots, x_k)]]^2 \leq \mathbb{E}_{x_1,\ldots,x_{k-1}} [\mathbb{E}_{x_k} e[g(x_1, \ldots, x_k)]^2]$$

$$= \mathbb{E}_{x_1,\ldots,x_{k-1}} [\mathbb{E}_{x_k^0, x_k^1} e[g(x_1, \ldots, x_{k-1}, x_k^0) + g(x_1, \ldots, x_{k-1}, x_k^1)]].$$

The lemma follows by repeating this k times. **QED**

Claim 13.23. For every function $f : X_1 \times \ldots \times X_k \to [2]$, and every $*$-protocol p^*,

$$R(f + p^*) = R(f).$$

Proof. Suppose $p^*(x) = g_1(x) + ... + g_k(x)$, where g_i is a cylinder in the i-th dimension. We show $\forall f, R(f \oplus g_k) = R(f)$; the same reasoning works for the other coordinates. Note for every x,

$$\begin{aligned}
&\sum_{\varepsilon_1,\ldots,\varepsilon_k \in [2]} (f(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k}) + g_k(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k})) \\
&= \sum_{\varepsilon_1,\ldots,\varepsilon_k} f(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k}) + 2 \sum_{\varepsilon_1,\ldots,\varepsilon_{k-1}} g_k(x_1^{\varepsilon_1}, \ldots, x_k^{0}) \\
&= \sum_{\varepsilon_1,\ldots,\varepsilon_k} f(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k}) \mod 2,
\end{aligned}$$

using that g_k does not depend on x_k. **QED**

This completes the sequence of claim that establish the chain of inequalities above, thus proving Lemma 13.16.

Finally, there remains to prove Lemma 13.17.

Proof of Lemma 13.17. We have:

$$\begin{aligned}
R(\text{GIP}) = \mathop{\mathbb{E}e}_{\substack{x_1^0,\ldots,x_k^0 \\ x_1^1,\ldots,x_k^1}} \left[\sum_{\varepsilon_1,\ldots,\varepsilon_k \in \{0,1\}} \sum_i \prod_j (x_j^{\varepsilon_j})_i \right] &= \mathbb{E} \prod_i e \left[\sum_{\varepsilon_1,\ldots,\varepsilon_k} \prod_j (x_j^{\varepsilon_j})_i \right] \\
&= \mathbb{E}e \left[\sum_{\varepsilon_1,\ldots,\varepsilon_k} \prod_j (x_j^{\varepsilon_j})_1 \right]^n = R\left(\bigwedge_k \right)^n,
\end{aligned}$$

using in the last equality the fact that any two independent random variables X, Y satisfy $\mathbb{E}[X \cdot Y] = \mathbb{E}[X] \cdot \mathbb{E}[Y]$, and where $\bigwedge_k$ is the AND function on k bits.

To save in notation let us replace $(x_1^0)_1, \ldots, (x_k^0)_1$ with $(y_1^0, \ldots, y_k^0)$, where $(y_i^0) \in [2]$; and similarly for $(x_1^1)_1, \ldots, (x_k^1)_1$. So we have:

$$R(\text{GIP}) = \mathop{\mathbb{E}e}_{\substack{y_1^0,\ldots,y_k^0 \\ y_1^1,\ldots,y_k^1}} \left[\sum_{\varepsilon_1,\ldots,\varepsilon_k \in \{0,1\}} \prod_j y_j^{\varepsilon_j} \right]^n.$$

Suppose that $y_1^0 \neq y_1^1, \ldots, y_k^0 \neq y_k^1$; then there exists exactly one choice of $\varepsilon_1, \ldots, \varepsilon_k$ making $\prod_j y_j^{\varepsilon_j} = 1$ (recall that $y_j^{\epsilon} \in [2]$; if any one of them is 0 the whole product is zero), and consequently

$$e\left(\sum_{\varepsilon_1,\ldots,\varepsilon_k} \prod_j y_j^{\varepsilon_j} \right) = e(1) = -1.$$

We have $y_1^0 \neq y_1^1, \ldots, y_k^0 \neq y_k^1$ with probability 2^{-k}. Therefore:

$$R(\text{GIP}) = \mathbb{E}e \left[\sum \prod_j y_j^{\varepsilon_j} \right]^n \leq (-1 \cdot 2^{-k} + 1 \cdot (1 - 2^{-k}))^n = (1 - 2^{-k+1})^n \leq e^{-cn/2^k}.$$

QED

13.2.2 The power of logarithmic parties

In this section we give two examples of the surprising power of protocols with many parties. First, note that the techniques beyond the impossibility result in the previous section 13.2.1 are effective when the number of players is $k \leq c \log n$, but useless when $k \geq \log n$. For GIP this is unsurprising, since it is almost always 0 for large k. But this is not clear if we replace, say, And with Majority in the definition of GIP. In fact, surprisingly there are general protocols that allow us to compute efficiently many such composed functions. For functions $f : [2]^n \to [2]$ and $g : [2]^k \to [2]$ we consider computing $f \circ g^{(k)}$ whose input is a $k \times n$ matrix M and the output is obtained by computing g on each of the n columns, resulting in an n-bit vector v, and then evaluating f on v. Here player j has row j of M on the forehead.

Theorem 13.24. Let $k \geq \log n + 2$. For any symmetric $f : [2]^n \to [2]$ and $g : [2]^k \to [2]$ there is a k-party protocol with communication k^c for $f \circ g^{(k)}$.

Proof. Let M be the $k \times n$ input matrix. For $v \in [2]^k$ denote by n_v the number of columns equals to v. It suffices to compute

$$\sum_{v:g(v)=1} n_v \tag{13.1}$$

because f is symmetric.

Let us assume that the players know a "base" vector $u \in [2]^k$ with $n_u = 0$. From this, they can compute any n_v as follows. Consider a path from v to the base vector u: $w_0 := v \to w_1 \to w_2 \to \ldots \to w_s := u$ where w_i and w_{i+1} differ in exactly one bit and $1 \leq s \leq k$. Note these paths only depend on u.

Then, telescopically:

$$n_v = \sum_{i \in [s]} (-1)^i \left(n_{w_i} + n_{w_{i+1}} \right), \tag{13.2}$$

using that $n_{w_s} = n_u = 0$. Note there is player that can compute $n_{w_i} + n_{w_{i+1}}$: since w_i and w_{i+1} differ in exactly one position h, player h can communicate the number of columns which agree in all other positions.

Then to compute (13.1) each player will communicate the sum over $v : g(v) = 1$ of their terms in equation (13.2). The total sum is $\leq 2^k \cdot k \cdot 2n$, so $k + \log kn + c$ bits suffice. The total amount of communication, over all parties, is then $\leq k^c$.

There remains to compute u. The first party can simply communicate a string $u' \in [2]^{k-1}$ that does not occur as a column in matrix M with the first row removed. This takes $\leq k$ bits. Such a u' exists because $2^{k-1} > n$. In particular, $u := 0u' \in [2]^k$ does not occur and so $n_u = 0$. **QED**

This protocol can be strengthened in various ways. We can handle different functions g. The communication can be improved to $\log^c n$, for any k. Possibly, some parties are not communicating anything (they will be absent from the protocol tree). For these extensions see Problem 13.4. The protocol can be also be made non-interactive, a.k.a. *simultaneous*.

This means that the communication of a party does not depend on the messages of the previous party. So all parties can speak simultaneously, and their collective messages determine the output. This is significant because the next key example gives simultaneous protocols.

As a further example of the power of many parties, we show that functions computable in ACC^0 (section §9.6) have low communication complexity. Hence communication lower bounds for power-log parties imply impossibility results for ACC^0

Lemma 13.25. Any $f \in \mathrm{ACC}^0$ has k-party protocols with communication k, for $k = \log^{c_f} n$ and any partition of the input bits into k sets.

Proof. By Lemma 9.50 it suffices to prove it for depth-2 circuits consisting of a symmetric gate on s And gates of fan-in t, where $\log s$ and t are $\leq \log^{c_f} n$. Fix an arbitrary partition of the input in $t+1$ sets $x_1, \ldots, x_{t+1}$. All that the players need to compute is the number of And gates that evaluate to 1. Consider any And gate. Since it depends on at most t variables, it does not depend on the bits in one of the sets, say x_j. Then the j-th party can compute this And without communication. So let us partition the And gates among the parties so that each party can compute the gates assigned to them without communication. Each party evaluates all the And gates assigned to them privately and broadcasts the number $\leq s$ of these gates that evaluate to 1. This takes a total of $ct \log s$ bits. **QED**

Note that the protocol is again simultaneous.

13.2.3 Pointer chasing

We consider the pointer chasing problem (section 13.1.2) in the multiparty model. The input consists of k layers of pointers, with edges going from nodes in layer i to nodes in layer $i+1$ only. Party i has the pointers on Layer i on their forehead. The last layer does not have edges but labels in [2] for each node. The goal is to compute the label at the node reached from a start node in Layer 1. (Equivalently, instead of labels we can allow for $k+1$ layers with the last layer consisting of two nodes only, and the task is outputting the node reached.) We consider *one-way* protocols where the parties speak onlhy once in the natural order that appears to make the problem hard: $1, 2, \ldots, k$.

It is convenient to work with a version of this problem where we output m labels, and where the size of each layer grows by a factor b. Note for $m = 1$ this is a boolean function. Formally, the input to the pointer-chasing function $G_k^{m,b}$ is a layered graph as above, where Layer $i = 1, 2, \ldots, k$ has mb^i nodes, each pointing to $[mb^{i+1}]$. (Nodes on the last layer k can point to [2].)

Exercise 13.26. Let $m = 1$.

Give a simultaneous protocol with communication cb.

Show that if the parties speak in any other order then the communication is $\leq \log b + c$.

We shall show that the communication of $G_k^{m,b}$ is $\geq c_k b$. The next theorem works in tandem with the proof technique and rules out even protocols that compute $G_k^{m,b}$ on an exponentially small fraction of inputs.

Theorem 13.27. There is no k-party one-way protocol P with communication $< \epsilon_k mb$ s.t. $\mathbb{P}_x[P(x) = G_k^{m,b}(x)] > 2^{-\epsilon_k m}$, for $\epsilon_k = k^{-ck}$.

Proof. We proceed by induction on k. For every k we prove the statement for any setting of m, b. The base case $k = 1$ is clear: $P(x)$ is a fixed string, while G is a uniform string in $[2]^m$. The probability that $P = G$ is $\leq 2^{-m} \leq 2^{-\epsilon_1 m}$, since $\epsilon_1 = 1$.

For the induction step we prove the contrapositive. We assume there is a k-party protocol P with communication $\leq \epsilon_k mb$ s.t. $\mathbb{P}[P(x) = G_k^{m,b}(x)] > 2^{-\epsilon_k m}$ and obtain a $(k-1)$-party protocol P' with communication $\leq \epsilon_{k-1} m'b$ s.t. $\mathbb{P}[P(x) = G_{k-1}^{m',b}(x)] > 2^{-\epsilon_{k-1} m'}$ for $m' := mb$.

Write $x = (x_1, y)$ where x_1 is on the forehead of the first party. Write $\gamma := 2^{-\epsilon_k m}$. We claim that

$$\mathbb{P}_y \left[\mathbb{P}_{x_1}[P(x_1, y) \geq G_k^{m,b}(x_1, y)] \geq \gamma^2 \right] \geq \gamma/2.$$

To get a sense of the parameters, note that γ can be close to 1. In that setting, having γ^2 instead of $\gamma/2$ in the inner bound makes a difference. To verify the claim, let us call y *heavy* if the $\mathbb{P}_{x_1}[P(x_1, y) \geq G_k^{m,b}(x_1, y)] \geq \gamma^2$, and let p be the probability that y is heavy. Then by the assumption on P we have

$$\gamma \leq p + (1-p)\gamma^2 \Rightarrow p(1-\gamma^2) \geq \gamma - \gamma^2 \Rightarrow p \geq \gamma \frac{1-\gamma}{1-\gamma^2} = \gamma \frac{1}{1+\gamma} \geq \gamma \frac{1}{2}.$$

The last inequality uses that $\gamma \leq 1$, for else the hypothesis would be false (the probability would be > 1).

Now let P_a be the protocol P where the first party's message is fixed to a, regardless of y. Since the length of the message is $\leq \epsilon_k mb$ there exists a s.t.

$$\mathbb{P}_y \left[\mathbb{P}_{x_1}[P_a(x_1, y) = G_k^{m,b}(x_1, y)] \geq \gamma^2 \right] \geq \gamma^b \cdot \gamma/2 = 2^{-\epsilon_k m' - \epsilon_k m - 1} \geq 2^{-3\epsilon_k m'}.$$

Here we use that $\epsilon_k m' \geq \epsilon_k b \geq 1$, for else the communication is ≤ 1 and the conclusion is trivial since the protocol cannot even output a bit. Critically, the probability over x_1 is not reduced because the first party's message does not depend on x_1.

On input y, the $(k-1)$-party protocol P' runs P_a for

$$r := c \log(1/\epsilon_k) b$$

times on inputs (x^i, y) for $i \in [r]$, where the x^i are independent choices for the first party's input. For any of the m' bits that are pointed to by some x^i, P' outputs the corresponding answer from P_a. In case different runs of P_a give different answers, the output of P' can be arbitrary. For any bit that is not pointed by any x^i, P' guesses at random. This gives a randomized protocol; one can fix the randomness and preserve the success probability.

The communication of P' is $\leq \epsilon_k mb \cdot r \leq c\epsilon_k \log(1/\epsilon_k) m'b$.

To analyze the success probability, fix any heavy y for which $\mathbb{P}_{x_1}[P_a(x_1, y) = G_k^{m,b}(x_1, y)] \geq \gamma^2$. The probability that all the r runs are correct is then

$$\geq \gamma^{2r} = 2^{-2\epsilon_k mr}.$$

The probability that there are $\geq (1 - 8\epsilon_k)m'$ bits that are not pointed to by some x^i is at most the probability that there is a set of size $8\epsilon_k m'$ s.t. mr pointers all fall there, which is at most (by Fact A.7, Fact A.5)

$$\leq \binom{m'}{8\epsilon_k m'} (1 - 8\epsilon_k)^{mr} \leq e^{c \log(1/\epsilon_k)\epsilon_k m'} e^{-8\epsilon_k mr} \leq 2^{-4\epsilon_k mr}$$

by the definition of r. (We assume for simplicity that $8\epsilon_k m'$ is an integer.) The constants are chosen to compare favorably with the prob. above that all the r runs are correct.

Putting altogether, the success probability over uniform y is at least the prob. of getting a heavy y as above, times the prob. that all the r runs are correct and cover at least a $(1 - 8\epsilon_k)$ fraction of m', times the prob. that the uncovered $\leq 8\epsilon_k m'$ bits are guessed correct. This is

$$\geq 2^{-c\epsilon_k m'} \cdot (2^{-2\epsilon_k mr} - 2^{-4\epsilon_k mr}) \cdot 2^{-c\epsilon_k m'} \geq 2^{-c\epsilon_k \log(1/\epsilon_k) m'}.$$

This shows that we can take $\epsilon_{k-1} \leq c\epsilon_k \log(1/\epsilon_k)$, which is consistent with our definition of $\epsilon_k := k^{-ck}$. **QED**

The total input length of $G_k^{1,b}$ is $n \leq kb^{k-1}$. The communication lower bound is $c\epsilon_k b \geq b/k^{ck} \geq n^{1/(k-1)}/k^{ck}$. The bound remains non-trivial for $k = \log^c n$. In the case $k = 3$ the bound is $\geq c\sqrt{n}$.

These bounds are tight for $G_k^{1,b}$, because of its definition with increasing layer size, see Exercise 13.26. But one can consider the natural *pointer chasing* where each layer has n nodes, of which $G_k^{1,b}$ is a special case.

Open question 13.28. Does multiparty pointer chasing require much communication even for $k \geq \log n$ parties? Does it require communication $\geq \sqrt{n}$ for $k = 3$?

It is tantalizing to conjecture that the answer to both is affirmative, cf Chapter 18. For $k = 3$ the trivial protocol takes communication n, and one might again be tempted to conjecture that that is tight. In fact, a clever protocol achieves sublinear communication. For concreteness, for $k = 3$ we consider pointer chasing on layers of sizes $1, n, n$, and define G as

$$G(i, g, h) := h(g(i))$$

where $i \in [n]$, $g : [n] \to [n]$, and $h : [n] \to [2]$.

Theorem 13.29. G has communication $o(n)$.

Proof. We sketch the proof in case g is a permutation π, see the notes for details. The protocol is based on a bipartite graph H between the n nodes in the middle layer and the n nodes in the last. For any permutation π, let $G_{H,\pi}$ denote the graph on the n last nodes where $\{x, y\}$ is an edge iff $\pi^{-1}(x)$ has an edge to y in H. The main claim is that there is H of degree $d = o(n)$ s.t. for any π $G_{H,\pi}$ has a partition in $r = o(n)$ sets s.t. any two nodes in the same set are connected in $G_{H,\pi}$. We call the sets *cliques.* Note any graph has a trivial partition consisting of n singletons. Given this, the protocol is as follows.

Party 1, for each of the r cliques, announces the parity of the bits $h(x)$ for x in the clique.

Party 2 announces $h(x)$ for all the d neighbors x of i in H. This is d bits.

Party 3 knows $k := \pi(i)$. It considers the clique of $G_{H,\pi}$ containing k. It knows the parity of the $h(x)$ for x in this clique. Also, for any x in the clique, x and k are connected, hence $\pi^{-1}(k) = i$ is adjacent to x. So from the message of Party 2 we know $h(x)$. We can subtract off all these bits to get $h(k)$. (As stated, this protocol requires interaction. To remove interaction, let Party 3 announce which of the cliques k is in, and also which of the d neighbors of i are connected via H to nodes in that clique that are not k. With this information we have one bit per clique, know which bit to look at, and know which bits of Party 2 to consider. Party 3 message would takes $\log r + d$ bits.)

The existence of H with suitable parameters can be established by the probabilistic method. Specifically, let H be distributed as $G(n, p)$, a random graph where each edge is present independently with prob. p. We observe that for any permutation $G_{H,\pi}$ is random from $G(n, p^2)$. Thus its complement $\overline{H}$ is random from $G(n, 1 - p^2)$. For a suitable choice of p one can show that w.h.p. $\overline{H}$ has chromatic number at $\leq r = o(n)$. This implies that H has a covering is equal to the minimum number of independent sets that you need to cover the nodes of the graph. From this the result follows. **QED**

This protocol can be generalized to more parties.

For the basic case of $k = 3$ we are left with a tantalizing gap between the lower bound of $c\sqrt{n}$ and the upper bound of $o(n)$.

13.3 Problems

Problem 13.1. Conclude the proof of the protocol for Greater-Than (Theorem 13.11).

Problem 13.2. [Arbitrary-partition communication complexity]

Give an explicit function (say in P) on $2n$ bits that requires (2-party) randomized communication $\geq cn$ for *every partition* of the $2n$ input bits in two sets of n bits. Hint: Use the result for IP (Theorem 13.15 for $k = 2$) in combination with Problem 12.2. Show that in every partition we can find a large instance of IP.

Problem 13.3. A randomized *private-coin* protocol is like a deterministic protocol except that each node is now labeled with a distribution on functions rather than a single function, and these distributions are independent. Suppose that $f : [2]^n \times [2]^n \to [2]$ has a randomized protocol with communication d and error ϵ. Show that f has a private-coin protocol with

communication $d + c\log(n/\epsilon)$ and error 2ϵ. Guideline: Use tail bounds and the union bound to show that the randomized protocol needs only small support.

Problem 13.4. Extend the proof of Theorem 13.24 to obtain:

(1) The same bound for the more general case of $f(g_1, g_2, \ldots, g_k)$ where the g_i may be different.

(2) An improved bound of $\log^c n$ for any k.

Problem 13.5. Let $f : \{0,1\}^{n^2 \cdot (0.3 \log n)} \to \{0,1\}$ be defined by

$$f(x) := \bigoplus_{i=1}^{n} \bigwedge_{j=1}^{0.3 \log n} \bigoplus_{k=1}^{n} x_{i,j,k}.$$

Prove that for every $d \in \mathbb{N}$ there is $\epsilon > 0$ such that f_n has correlation $\leq n^{-\epsilon \log n}$ with any Maj-AC^0 circuit (Definition 9.15) of depth d and size $n^{\epsilon \log n}$.

Use this to obtain a PRG (Definition 11.3) against Maj-AC^0 circuits. Specifically, show that for every d there is $a > 0$ and a PRG with seed length $2^{a\sqrt{\log n}}$ that fools Maj-AC^0 circuits of size n with error $1/n$.

13.4 Notes

> *Because it is* basic.

Communication complexity was initiated in [391] (to whose author the quote is credited, according to [380]). Pointer chasing was first studied in [276]. Theorem 13.7 was claimed in [270] where a proof is sketched. [287] provides a proof. We presented the proof from [357] which is simpler. The key idea in the proof is taken from the proof of the fixed-set Lemma 3.14 from [148]. The randomized communication complexity of pointer chasing is studied in [270, 393, 242].

The arbitrary partition model was introduced in [276].

Problem 13.3 is from [266].

Several proofs of the lower bound for disjointness (Theorem 13.13) exist [202, 291, 44], see the books [222, 287] or the survey [82] for two different expositions. For more on disjointness see also the survey [313].

The protocol for Greater-Than, Theorem 13.11, is from [269]. For the random-walk with backtrack, see [111]. The matching lower bound, Theorem 13.12, is from [370].

The linear lower bound for IP (Theorem 13.15 for $k = 2$) is from [87] who improved previous weaker bounds.

The number-on-forehead model is from [80]. Computing degree-k polynomials with $k+1$ parties, used in the proof of Lemma 13.25, is from [166].

The lower bound for GIP, Theorem 13.15, is from [39]. Several presentations of the proof exist: [89, 288, 377]. We followed the latter. While for GIP the lower bound is limited to

$k \leq 0.5 \log n$ parties, for other functions [39] proves lower bounds for as many as $\delta \log n$ parties for any constant $\delta < 1$. For an alternative proof of such lower bounds see [100].

The extension to arbitrary partition, Problem 13.2, is from [170].

The protocol for composed functions – Theorem 13.24 – is from [6]. Our presentation has some minor differences. Besides Problem 13.4 the result can be extended in various ways, see [6] and [159]. The first amazing protocol in this line of works is from [36] which applies when all composed functions are symmetric, cf Chapter 18.

A proof of the multiparty pointer chasing lower bounds – Theorem 13.27 – in the case $k = 3$ appeared in [37] but did not readily extend to larger k. The proof we presented is a streamlined version of the argument in [378] (the latter paper works with trees instead of graphs). The upper bound in Theorem 13.29 is from [285]. The extension to general functions is from [68]. These works don't quite give simultaneous protocols though.

Regarding randomized vs deterministic multiparty protocols, a non-explicit linear separation is in [51]. A logarithmic separation is also in [51]. A stronger explcit separation is [209]. A candidate for an explicit linear separation is the problem of deciding if $xyz = 1_G$ for a group G. With randomness, this can be solved with constant communication, for any group. (Show this!) Without randomness, this could require large communication for suitable groups, see [372, 196].

Chapter 14

Arithmetic

Previous chapters have mostly investigated the complexity of computing boolean functions starting from basic functions and combining them via simple operations. For example for boolean circuits the basic functions are the x_i and we combine them via various instructions such as And. It is natural to consider computing other objects using different basic functions and compositions. In fact, we have already encountered such variants, like polynomials (section §9.2) and integer circuits (Definition 3.7). In this chapter we (re)explore a number of arithmetic models, focusing on non-uniform models. Perhaps unexpectedly, many of the main themes and results seen so far will recur. These include the grand challenge, reductions, completeness, depth reduction, the surprising power of restricted models, and impossibility results that are "just short" of proving major separations.

14.1 Linear transformations

One of the simplest models is that of *xor circuits*, circuits whose gates only compute Xor on 2 bits. Xor circuits can't compute arbitrary functions: They always compute a *linear function*. Any linear function $M : \mathbb{F}_2^n \to \mathbb{F}_2^m$ can be computed by an xor circuit of size $\leq mn$. As in Theorem 2.9, one can show that most linear functions require about that size. The grand challenge here is to come up with an explicit linear function requiring xor circuits of large size. Again similarly to the boolean setting (see section §9.4.2) we do not know of explicit linear functions requiring circuits of super-linear size, even if the depth is restricted to logarithmic. The techniques in Theorem 9.36 show that the (matrix of the) linear transformation computed by an xor circuit of linear size and logarithmic depth is *not rigid*, i.e., we can change a small number of entries to obtain a low-rank matrix. Hence, constructing rigid matrices is a basic question which would imply new circuit lower bounds.

Theorem 14.1. Let $C : \mathbb{F}_2^n \to \mathbb{F}_2^m$ be a circuit of size $an \geq m$ and depth $a \log n$. Then the $m \times n$ matrix M_C of the linear transformation computed by C equals

$$L + E$$

where L and E as $m \times n$ matrices over $\mathbb{F}_2$, L has rank $\leq c_a n / \log\log n$, and E has $\leq n^{1.01}$ non-zero entries.

In other words, C computes a linear function that is close to a low-rank function L. The matrix E gives the entries to be changed.

Exercise 14.2. Prove Theorem 14.1 by modifying the proof of Theorem 9.36.

As we did in the boolean setting (Chapter 8, Chapter 9) we can consider circuits of constant depth with *unbounded fan-in*. In this setting, however, the size of a circuit should not be taken as the number of gates, because any linear function $M : \mathbb{F}_2^n \to \mathbb{F}_2^m$ is computable with just m gates. Instead, we take the number of *wires* as the complexity measure. The trivial upper bound of nm wires can be implemented in depth 1. We seek explicit linear functions requiring a large number of wires for constant depth.

14.1.1 The power of depth-2 xor circuits

A natural candidate for a hard linear transformation is the encoding map of a *good linear code*. Recall from Problem 3.6 that a binary error-correcting code with block length n, message length k, and minimum distance d, called an $(n, k, d)_2$ code for short, is a subset $C \subseteq [2]^n$ of size 2^k s.t. for any distinct $x, y \in C$, x and y differ in $\geq d$ coordinates. Such (families of) codes are good if $k \geq cn$ and $d \geq cn$.

In this section we are specifically interested in *linear codes*. An $(n, k, d)_2$ code C is *linear* if

$$C = \{Mx : x \in \mathbb{F}_2^k\}$$

for an $n \times k$ matrix M over $\mathbb{F}_2$.

Exercise 14.3. Prove C is a linear $(n, k, d)_2$ code iff any non-zero $x \in C$ has weight $\geq d$.

Good linear codes can be shown to exist via the probabilistic method. We will give a stronger result in Theorem 14.4.

To compute a good linear code, xor circuits of depth 1 require a nearly quadratic number of wires (exercise). One might have suspected that a quadratic or almost quadratic number of wires is required even for constant-depth circuits. In fact, even circuits of depth 2 can have a quasi-linear number of wires.

Theorem 14.4. There are binary linear codes $(n, cn, cn)_2$ whose encoding map has depth-2 xor circuits with $cn \log^2 n$ wires.

Proof. Denote by $k = cn$ the input length to the circuit, which we can assume to be a power of 2. By Exercise 14.3 it suffices to exhibit an xor circuit that for any non-zero input outputs a string with $\geq cn$ non-zero bits. Divide the gates in the middle layer into $\log k$ *blocks.* We will show that for every input there is a block that is nearly balanced, that is, the fraction of gates in the block that evaluates to 1 is in, say, $[0.1, 0.9]$. From this, we construct the

output layer via the probabilistic method. For each output bit select uniformly at random a gate from each block and output their sum. Do this independently for each output bit. The probability that the output has $\leq cn$ non-zero bits is $\leq 2^{-k}$. Hence by a union bound over the $2^k - 1$ possible non-zero inputs we can fix the output layer and obtain the desired circuit. Note that the output layer consists of $cn \log n$ wires.

There remains to construct the middle layer. Block i, for $i \in [\log k]$, will be nearly balanced on the subset $X_i \subseteq [2]^k$ of inputs of weight $\in [2^i, 2^{i+1}]$. To construct this block, for each gate in the block do the following. First select $k/2^i$ uniformly chosen input bits, then pick a random subset of them, and output the sum. For every fixed input $x \in X_i$, each gate has a constant probability of selecting a 1 bit in the first step, in which case the sum is 1 with probability 1/2 in the second step. We want to fix the block so that it works for every input in X_i. Again by the probabilistic method, it suffices to pick $c \log |X_i|$ gates in that block.

Now we have $\log |X_i| \leq c2^i \log k$. (For example one can note $|X_i| \leq c2^i \binom{k}{2^i} \binom{k}{2^{i+1}}$ for every i and then use Fact A.7.) Hence overall the number of wires in the block is

$$c\frac{k}{2^i} \cdot 2^i \log k \leq ck \log k.$$

Summing over all $\log k$ blocks, the total number of wires in the middle layer is $ck \log^2 k$. **QED**

By a slightly more careful balancing of parameters one can improve the number of wires to $cn(\log n / \log\log n)^2$, which is tight (see the notes).

14.2 Integers

We consider computing n-bit integers using integer circuits (Definition 3.7). Again, counting arguments as in the proof of Theorem 2.9 show that most n-bit integers require integer circuits of size $\geq n/\log^c n$, and this is nearly tight:

Exercise 14.5. Show that any n-bit integer has an integer circuit of size $\leq cn$.

The grand challenge here is to exhibit explicit integers that are hard to compute. In particular, we seek n-bit integers that cannot be computed with $\log^a n$ operations for a constant a. A prominent integer in this context is the *factorial*:

$$k! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot k \in \left[(k/2)^{k/2}, k^k\right].$$

Thus the bit length of $k!$ equals k up to a $c \log k$ factor (which won't play a role in these connections).

Similarly to Theorem 1.22, we can show that if computing factorials is easy then factoring is also easy. Specifically, one can factor the product of two random primes with high probability, refuting a popular conjecture in cryptography that is the basis of several cryptosystems.

Theorem 14.6. Suppose $k!$ has integer circuits of size $\log^a k$ for a constant a. Let P and Q be i.i.d. n-bit primes, arbitrarily distributed. There is a boolean circuit of size n^c that given $P \cdot Q$ computes P with prob. $\geq c$.

Proof. First we show that for any k there are boolean circuits of size n^{c_a} that factor the product x of any two n-bit primes p and q s.t. $p \leq k < q$. By assumption, there are integer circuits for $k!$ of size $\log^{c_a} k \leq n^{c_a}$. We use this integer circuit to compute

$$k! \bmod x$$

by a boolean circuit. The boolean circuit first proceeds like the integer circuit but takes the remainder mod x at each gate to keep the bit length feasible. After that, it outputs (see Fact A.2)

$$\gcd(k! \bmod x, x) = \gcd(k!, x) = p.$$

QED

Exercise 14.7. Conclude the proof of Theorem 14.6.

Thus we have shown that if factorial is easy, then factoring is also easy. And we will see below in section §14.4 that if factorial is hard then another long-sought separation follows. Hence the complexity of factorials in the integer-circuit model appears pivotal.

14.3 Univariate polynomials

A next natural question is computing *univariate* polynomials by *arithmetic circuits*. In this section we briefly discuss this setting for context and to set up the model. We have already encountered arithmetic circuits in Definition 10.7, where we considered them over $\mathbb{N}$ or the prime finite fields. Here we consider them over an arbitrary field $\mathbb{F}$:

Definition 14.8. An *arithmetic circuit* of size s in n variables over a field $\mathbb{F}$ is a sequence of assignments to gates g_i where assignment i is of the following types:

- $g_i := at + a't'$, where $a \in \mathbb{F}$ and t is either 1, or a variable x, or a gate g_j with $j < i$, and similarly for a', t',
- $g_i := at \cdot t'$, where $a \in \mathbb{F}$ and t and t' are as above.

We make several remarks on the model.

First, note the goal is to understand *monomials*, not coefficients, so we allow gates that compute any field element (unlike in section §14.2).

Second, a variant of this model has been considered where we also allow *division* is an allowed instruction. However, circuits with division gates computing low-degree polynomials can be simulated by arithmetic circuits without division gates, see the notes.

Finally, a distinction must be made. We can consider computing polynomials *formally*, which we can think of as a sequence of coefficients, or *informally*, as functions. This distinction disappears when the field is larger than the degree by Fact A.50, but otherwise leads to different theory. For example over $\mathbb{F}_2$ we have $x^2 = x$ informally (i.e. the identity holds for every field element) but obviously not formally. Formal identities are also informal, so informal impossibility results are harder to establish than formal.

When computing univariate polynomials, circuits only have $n = 1$ variable. Regarding impossibility results, the situation is similar to the previous section §14.2. A specific polynomial of interest is the approximation to the exponential function: $\sum_{i=0}^{n} X^i/i!$.

14.4 Multivariate polynomials

Arguably the most studied setting is the one of polynomials in n variables, because it is closely related to other classes (as we shall see).

Exercise 14.9. Let $B : [2]^n \to [2]$ be a (boolean) circuit of size s. Show that over any field there is an arithmetic circuit A of size $\leq cs$ s.t. $A(x) = B(x)$ for all $x \in [2]^n$.

The same remarks in section §14.3 apply to this section.

Again, the challenge is to exhibit "explicit" polynomials that are hard to compute. For several explicit degree-d polynomials in n variables we can prove bounds of the form $cn \log d$. We now give examples. First, we can prove the following result for computing polynomial maps.

Theorem 14.10. A circuit computing the n polynomials $x_1^d, x_2^d, \ldots, x_n^d$ requires size $\geq cn \log d$, over any field.

For the proof see the notes.

We can obtain as a corollary impossibility results for computing a single polynomial.

Theorem 14.11. Computing $\sum_{i \in [n]} x_i^d$ requires arithmetic circuits of size $cn \log d$, over any field.

Theorem 14.11 follows from 14.10 via the somewhat surprising fact that computing a polynomial in n variables is not much harder than computing all its *partial derivatives* with respect to the n variables. The partial derivative $\partial_x p$ w.r.t. a variable x of a polynomial p can be defined recursively as follows:

$$\begin{aligned} \partial_x(x) &= 1, \\ \partial_x(a) &= 0 \text{ for } a \in \mathbb{F}, \\ \partial_x(p+q) &= \partial_x(p) + \partial_x(q), \\ \partial_x(p \cdot q) &= \partial_x(p) \cdot q + p \cdot \partial_x(q). \end{aligned}$$

Lemma 14.12. Suppose a polynomial $p(x_1, x_2, \ldots, x_n)$ over a field $\mathbb{F}$ is computable by an arithmetic circuit of size s. Then there is an arithmetic circuit of size cs computing

$$\partial_{x_1} p, \partial_{x_2} p, \ldots, \partial_{x_n} p.$$

Proof. The proof is by induction on s using the *chain rule* of partial derivatives. To state this rule let $f = f(x_1, x_2, \ldots, x_n, y)$ and $q = q(x_1, x_2, \ldots, x_n)$ and introduce the notation

$$f[y \leftarrow q]$$

which denotes the polynomial obtained by replacing y with q in f. The chain rule is then

$$\partial_{x_i} f(x_1, x_2, \ldots, x_n, q(x_1, x_2, \ldots, x_n)) = (\partial_{x_i} f)[y \leftarrow q] + (\partial_y f)[y \leftarrow q] \cdot \partial_{x_i} q.$$

We only need this when q is a sum or product of two variables. The proof follows by writing the polynomials as a sum of monomials and using the definition of ∂ and we omit it.

For example, if $f = y^2$ and $q = x_1 + x_2$ then

$$\partial_{x_1}(x_1 + x_2)^2 = 0 + 2y[y \leftarrow x_1 + x_2] \cdot \partial_{x_1}(x_1 + x_2) = 2(x_1 + x_2) \cdot 1.$$

We prove the lemma by induction on s. The base case $s = 1$ is left as exercise. For the inductive step, consider a circuit of size s for p and select an instruction that does not depend on other gates. Suppose that instruction computes polynomial q (in ≤ 2 variables). Replace that instruction with variable y (that is, remove the instruction and replace each occurrence of the gate with y in other instructions) and denote $f(x_1, x_2, \ldots, x_n, y)$ the resulting polynomial. Because we can compute $f(x_1, x_2, \ldots, x_n, y)$ with $s - 1$ instructions, by induction we can compute its partial derivatives with size cs.

To compute the partial derivatives of

$$p(x_1, x_2, \ldots, x_n) = f(x_1, x_2, \ldots, x_n, q)$$

use the chain rule. The partial derivatives $\partial_{x_i} f$ and $\partial_y f$ are available by induction. Replacing y with q only takes an extra instruction. For the variables x_i that do not occur in q we have $\partial_{x_i} q = 0$. For those that occur we can compute $\partial_{x_i} q$ with a constant number of gates. In all, we add a constant number of gates for each instruction. **QED**

Similarly to NP, as we saw already in Definition 10.8, an important class of polynomials can be defined by summing over all boolean values of a set of variables.

Definition 14.13. A Σ-*arithmetic circuit* of size s on n variables is an arithmetic circuit $C(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_s)$ of size s with s extra variables which computes the polynomial

$$p(x_1, x_2, \ldots, x_n) := \sum_{y_1, \ldots, y_s \in [2]} C(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_s).$$

Several polynomials of interest that are not known to have small arithmetic circuits can be shown to have small Σ-arithmetic circuits. A notable example is the *permanent* polynomial in n^2 variables $x_{i,j}$, defined as

$$\mathrm{perm}(x) := \sum_{\pi} \prod_{i \in [n]} x_{i,\pi(i)}$$

where π ranges over all permutations of $[n]$. To show that perm has power-size Σ-arithmetic circuits, encode π using n^2 bits M specifying an $n \times n$ permutation matrix, also written M. Suppose we have a polynomial g s.t. $g(M) = 1$ if M is a permutation and 0 otherwise. Then we have:

$$\mathrm{perm} = \sum_{M \in [2]^{n^2}} g(M) \prod_{i \in [n], j \in [n]} x_{i,j} \cdot M_{i,j}.$$

Thus it only remains to show that g has small arithmetic circuits.

Exercise 14.14. Show this.

Similar to the P vs. NP question, the prominent question here is whether Σ-arithmetic and arithmetic circuits have similar power. This is known as the VP vs. VNP question. Arithmetic and boolean questions are interrelated. As an example, we connect the power of Σ-arithmetic circuits to the complexity of computing factorials by integer circuits (section §14.2), and hence by Theorem 14.6 factoring by boolean circuits. To state this connection we slightly change the model of arithmetic circuits. We work over the integers, and only use constants $+1$ and -1.

Theorem 14.15. [If Σ-arithmetic circuits are easy then so is factorial] Suppose there is a constant a s.t. every Σ-arithmetic circuit of size s over $\mathbb{Z}$ using constants $-1, 1$ only has an equivalent arithmetic circuit of size s^a over $\mathbb{Z}$ using constants $-1, 1$ only.

Then $n!$ has integer circuits of size $\log^{c_a} n$.

The proof is an excellent display of "scaling up and down" (cf section §4.5) and connecting disparate complexity results.

Proof. Rather than giving circuits for $n!$, a number of $\leq cn \log n$ bits, we will give circuits for $2^n/n^c!$, a number of 2^n bits. This makes it slightly easier to connect to other results, and to index bits.

First we claim bit i of this 2^n-bit factorial given $i \in [2]^n$ is computable by constant-depth majority circuits of size 2^{n^c}. This follows from the fact that iterated multiplication is in TC^0 (Theorem 8.7). The hypothesis now allows us to iteratively turn each majority gate into a power-size boolean circuit. We omit the details of this step. In the end, we obtain a boolean circuit C of size n^c s.t. $C(i)$ is bit i of the factorial.

We can view C as an arithmetic circuit over the integers and consider

$$S(x_{n-1}, \ldots, x_0) := \sum_{j_0, j_1, \ldots, j_{n-1} \in [2]} C(j) x_0^{j_0} x_1^{j_1} \cdot x_{n-1}^{j_{n-1}}.$$

Note that $S(2^{n-1}, \ldots, 8, 4, 2, 1)$ equals the desired $n!$. We can't immediately apply the hypothesis to S, until we note

$$X_0^{j_0} = (X_0 j_0 + 1 - j_0)$$

which allows to write S as a Σ-arithmetic circuit. Applying the hypothesis again yields an equivalent n^c-size arithmetic circuit C', and then again the desired factorial is $C'(2^{n-1}, \ldots, 8, 4, 2, 1)$. The powers of 2 take cn operations. **QED**

14.5 Depth reduction and completeness

To set the stage, we note that reducing the depth of arithmetic circuits is impossible:

Exercise 14.16. Give an arithmetic circuit of size s that does not have an equivalent arithmetic circuit of depth $< s$, for all s.

However, interestingly it is possible to reduce the depth under the additional assumption that the circuit computes a polynomial of low degree:

Theorem 14.17. Any arithmetic circuit of size s computing a polynomial degree d has an equivalent arithmetic circuit of size s^c and depth $c(\log s)(\log d)$.

In the unbounded fan-in setting, the following is known and reminiscent of Theorem 9.36.

Theorem 14.18. Any n-variate polynomial over $\mathbb{Q}$ of degree d computable by an arithmetic circuit of size $s \geq n$ can also be computed by a depth-3 arithmetic circuit of size $s^{c\sqrt{d \log d}}$.

We shall prove in section 14.6.2 impossibility results for circuits of size $n^{c\sqrt{d}}$ computing degree-d polynomials. Thus, as mentioned at the beginning of the chapter, the situation in the arithmetic world is strikingly analogous to that in the boolean world discussed in section §6.3. We have impossibility results for small-depth circuits that are "just short" of having major consequences (i.e., super-power size lower bounds for circuits of arbitrary depth).

Turning to completeness, the results in section §7.5, extended to other fields, show that iterated product of 3×3 matrices is complete for arithmetic circuits of small depth. As in that section, the reduction is as simple as it gets: For any power-size circuit one can write down a power-size product where the matrix entries are either constants or variables that computes the same polynomial. Using the depth reduction Theorem 14.17, one can extend this completeness to circuits of any depth *computing low-degree polynomials.* However, the size overhead is more than a power (due to the fact that the depth is not order of $\log s$ but $\geq (\log s) \log d$). One can show a similar completeness result for the determinant. In a similar spirit, one can show that the permanent is complete for Σ-arithmetic circuits. We refer to the notes for pointers to these results.

14.6 Alternation

We now turn to constant-depth circuits. As before (Chapter 9), the circuits now have unbounded fan-in; moreover, for a fine analysis of depth, we allow multiplication by arbitrary constants along wires. In other words we allow instructions

- $g_i := \sum_k a_k t_k$, where the $a_k \in \mathbb{F}$ and each t_k is either $t_k = x_j$ for a variable x_j, or $t_k = 1$, or $t_k = g_j$ for a gate g_j with $j < i$,
- $g_i := a \prod_k t_k$, where $a \in \mathbb{F}$ and each t_k is as above.

We use strings of Σ and Π to denote the alternation in a circuit. For example, a $\Pi\Sigma$ circuit is a depth-2 circuit whose output is a Π instruction taking as input Σ instructions, while a $\Sigma\Pi\Sigma$ circuit is a depth-3 circuit whose output is a Σ instruction taking as input $\Pi\Sigma$ circuits.

First we give an example of the power constant-depth circuits. Then we discuss several impossibility results.

14.6.1 The power of depth 3

Consider the elementary symmetric polynomial of degree d in n variables:

$$e_{n,d}(x_1, x_2, \ldots, x_n) := \sum_{S \subseteq [n], |S|=d} \prod_{i \in S} x_i. \tag{14.1}$$

These polynomials, and efficient ways for computing them, are of central importance, as we also see below in section 14.6.2. The rhs is an expression for a circuit of size $\geq n^d$. Surprisingly, much smaller circuits exist.

Theorem 14.19. For any d, $e_{n,d}$ has depth-3 $\Sigma\Pi\Sigma$ arithmetic circuits of size cn^2, over any field of size $\geq n + 1$.

Proof. *Linear algebra magic.* Note that

$$p(t, x) := \prod_{i=1}^{n} (1 + t x_i) = \sum_{i=0}^{n} t^i e_{n,i},$$

that is, $e_{n,i}$ is the "coefficient" of t^d in $p(t, x)$, where $x = (x_1, x_2, \ldots, x_n)$. We can compute $p(t, x)$ efficiently, and so we should be able to get its coefficients via interpolation. Specifically, for a field element α denote by v the "moment" vector

$$v := (\alpha^0, \alpha^1, \ldots, \alpha^n).$$

Also denote by e the vector of polynomials in x

$$e := (e_0, e_1, \ldots, e_n).$$

Note that $p(\alpha, x) = \langle v, e \rangle$. Suppose we have field elements α_i for $i \in [n+1]$ whose corresponding vectors v_i are linearly independent. Then we can find a linear combination

$$\sum_i a_i v_i$$

that equals the vector w_d with 1 in the coordinate with index d, and zero elsewhere. We could then compute e_d as

$$\sum_i a_i p(\alpha_i, x) = \sum_i a_i \langle v_i, e \rangle = \langle \sum_i a_i v_i, e \rangle = \langle w_d, e \rangle = e_d.$$

The lhs is an arithmetic circuit of size cn^2.

There remains to exhibit such field elements. In fact, the vectors v_i are linearly independent for any choice of distinct field elements α_i (Fact A.41). Over the complex numbers we can also let $\alpha_i := \omega^i$ where ω is the $(n+1)$-th primitive root of unity $e^{\sqrt{-1}2\pi/(n+1)}$. The vectors v_i are then orthogonal because

$$\langle v_i, v_j \rangle = \sum_{k \in [n+1]} \omega^{k(i-j)}$$

which is 0 if $i \neq j$ and otherwise is $n+1$. Hence they are independent (Fact A.38). **QED**

We shall see in Claim 14.23 a different approach where the circuit is somewhat larger but has additional structure.

14.6.2 Impossibility results

Over the field $\mathbb{F}_2$, impossibility results for constant-depth arithmetic circuits follow from the impossibility results for AC^0 with parity gates we have obtained in section 9.1, Theorem 9.5. This is because in $\mathbb{F}_2$ Xor is the same as addition, while And is the same as multiplication. These results are even *informal* (cf section §14.3) and nothing better is known even if one is formal.

These techniques in section 9.1 can be extended to slightly larger fields $\mathbb{F}_q$. The idea is similar to that in section 9.1: we show that such circuits are approximated by low-degree polynomials. We sketch this idea in the case of depth-3 $\Sigma\Pi\Sigma$ circuits, highlighting where the field size q plays a role. It suffices to approximate $\Pi\Sigma$ circuits well. Consider one such circuits, and let r be the rank of the linear forms input to the Π gate (excluding their constants, if any). If the rank is large, then if q is small over a uniform input it is likely that at least one linear form will be zero and so the whole circuit is zero. If the rank r is small, then we can write each linear form as a linear combination of $\leq r$ linear forms. If we expand the Π gate we get a sum of products of these r linear forms. Now we can use the fact that $X^q = X$ in $\mathbb{F}_q$, so we reduce the degree of each form in any product to at most $q-1$. Overall, the degree will be $\leq (q-1)r$. Using these ideas one can obtain arithmetic impossibility results over small fields.

But for larger fields a different set of techniques appears necessary. Indeed, the above techniques from section 9.1 give impossibility results for symmetric polynomials, but over larger fields symmetric polynomials are easy by Theorem 14.19. A natural candidate hard polynomials is the *iterated matrix multiplication* polynomial $\mathrm{IMM}_{\ell,d}$ on $n = d\ell^2$ variables, defined as follows. The variables are partitioned into d sets X_i, each viewed as an $\ell \times \ell$ matrix X_i. Then $\mathrm{IMM}_{\ell,d}$ is defined to be the polynomial that is the entry $1,1$ of the product matrix $\prod_i X_i$. Equivalently,

$$\mathrm{IMM}_{\ell,d} = \sum_{i_0,i_1,\ldots,i_{d-2} \in [\ell]} (X_0)_{1,i_0} (X_1)_{i_0,i_1} (X_2)_{i_1,i_2} \cdots (X_{d-1})_{i_{d-2},1}.$$

We have the following result:

Theorem 14.20. The $\mathrm{IMM}_{\ell,d}$ polynomial in $n = d\ell^2$ variables where $d \leq c \log n$ requires depth-3 arithmetic circuits of size $\geq n^{c\sqrt{d}}$.

An extension of this result gives superpower impossibility results for any constant depth, and applies to other fields as well. The proof, presented next, contains most of the ideas that go into extending the result to higher depth and other fields.

Proof of Theorem 14.20

We prove the result for a different explicit polynomial. The result for IMM follows via a reduction without new ideas, the details can be found in the paper referenced in the notes. The proof has the following steps.

1. We define set-multilinear circuits and prove that any circuit can be converted into a set-multilinear circuit efficiently.
2. We introduce a complexity measure, and give a specific setting of parameters and an explicit set-multilinear polynomial for which it is noticeably large.
3. We show that the measure is much smaller for the set-multilinear circuits obtained in 1.

We now develop each step in turn.

Step 1

We partition the n variables into d sets X_i, $i \in [d]$. This partition is fixed throughout the argument. Jumping ahead, the sizes of the X_i will not be equal, but we will worry about this later.

Definition 14.21. A polynomial p is *set-multilinear*, abbreviated sm, if there is $D \subseteq [d]$ s.t. every monomial in p has exactly one variable in X_i for every $i \in D$, and no variable in X_i for $i \notin D$. A circuit is sm if every gate computes an sm polynomial

In particular, p has degree D and is multilinear. Note that D need not equal $[d]$; in particular, p does not have to have degree d, and this will be useful later. On the other hand, if p does have degree d then necessarily $D = [d]$.

Lemma 14.22. Any $\Sigma\Pi\Sigma$ circuit of size s computing a degree-d sm polynomial has an equivalent sm $\Sigma\Pi\Sigma\Pi\Sigma$ circuit of size $d^{cd}s^c$.

In turn we break the proof of this lemma in three claims. The second and third claim are technically easy, but provide useful breaking points for the proof. Some of the "magic" happens right in the first claim. A simple consequence of it, stated in the second claim, is making the fan-in of multiplication gates $\leq d$. This then makes the size blow-up in the third claim tolerable. The lemma and the first claim generalize without new ideas to circuits of any depth; we focus on $\Sigma\Pi\Sigma$ for simplicity. The other claims we state in full generality.

The first claim makes polynomials *homogeneous*, i.e., each gate computes a polynomial where each term has the same degree (but the polynomial is not necessarily sm or even multilinear). For a polynomial p we define $p^{(k)}$ as the degree-k homogeneous part, consisting of the monomials of degree exactly k. We say p is homogeneous if $p = p^{(k)}$ for some k

Claim 14.23. Suppose a $\Sigma\Pi\Sigma$ circuit of size s computes a polynomial p. Then there is a $\Sigma\Pi\Sigma\Pi\Sigma$ homogeneous circuit of size $d^{cd}s^c$ which computes the homogeneous parts $(p^{(0)}, p^{(1)}, \ldots, p^{(d)})$ of p.

The size bound can be improved to $c^{\sqrt{d}}s^c$ by a less crude analysis of the repeated applications of Fact A.49 below.

Proof. Consider one product gate q. We show how to compute the k-homogeneous part $q^{(k)}$ of q with the desired resources. From this we compute $p^{(k)}$ as follows. Either $p^{(k)} = 0$, in which case there is nothing to do, or else it is the sum of $q_i^{(k)}$ where $p = \sum q_i$.

Write the input Σ gates of q as $\ell_i + b_j$ where each ℓ_i is a sum $\sum_j a_{i,j}x_j$ and b_j is a constant term. We can assume that $b_j = 1$. Indeed, the case $b_j = 0$ can be treated separately, and otherwise we can divide by b_j and multiply back in the multiplication gate. We then have

$$q = \prod_{i \leq s}(\ell_i + 1) = \sum_{S \subseteq [s]} \prod_{i \in S} \ell_i.$$

And so

$$q^{(k)} := \sum_{S \subseteq [s], |S| = k} \prod_{i \in S} \ell_i.$$

This is the elementary symmetric polynomial $e_{s,k}$, equation (14.1), evaluated at the ℓ_i. It is a homogeneous polynomial, but as mentioned in section §14.6.1, computing it directly as in the rhs above would give size $\geq n^d$, which we cannot afford. Theorem 14.19 gives smaller circuits, but not homogeneous.

We seek an alternative efficient, homogeneous, $\Sigma\Pi\Sigma\Pi\Sigma$ circuit. Towards this, consider the power sum polynomials

$$p_{s,k} := \sum_{i \leq s} x_i^k.$$

We have by Fact A.49

$$k \cdot e_k = \sum_{i=1}^{k} (-1)^{i-1} e_{k-i} \cdot p_i,$$

for all k. Applying this repeatedly, we write e_k as a sum of $\leq k^k$ products of $\leq k$ polynomials p_i. Since each p_i is naturally a homogeneous $\sum\prod$ circuit, we obtain a homogeneous $\sum\prod\sum\prod$ circuit for e_k. Instantiating the variables with the ℓ_i we obtain a $\sum\prod\sum\prod\sum$ circuit for $q^{(k)}$. The size of this circuit is $\leq k^k \cdot k \cdot s \cdot k \cdot n$, where each factor is the fan-in at the corresponding depth.

We only need to consider $k \leq d$. Hence the total size is $\leq d^{cd} s^c$. **QED**

Claim 14.24. A homogeneous circuit computing a polynomial of degree d has an equivalent homogeneous circuit of no larger size in which the fan-in of each Π gate is $\leq d$.

Proof. Replace all instructions computing polynomials of degree $> d$ with the constant 0. The correctness of this step can be argued inductively. For a Σ instruction computing a polynomial of degree d, it is safe to set to 0 any summand with degree $> d$. This is because by homogeneity these summands do not have monomials of degree $\leq d$, so all their monomials must cancel in the output. For a Π gate the same holds, because multiplication increases degree (Fact A.47), so if it is computing a polynomial p of degree d and a term has degree $> d$, then in fact $p = 0$.

After this, a non-zero product gate can have $\leq d$ non-constant terms. The constant terms need not count towards fan-in since they can be absorbed elsewhere. **QED**

Finally, we get sm.

Claim 14.25. Suppose sm degree-d polynomial p is computable by a circuit of size s and depth t where the fan-in of each multiplication gate is $\leq d$. Then p is also computable by a sm circuit of depth t and size $d^{cd}s$.

Proof. We inductively replace each gate g in the circuit with 2^d gates g_D where for $D \subseteq [d]$ gate g_D computes the monomials in g that are sm w.r.t. D; and add circuitry accordingly.

If $t = 1$ we have $t_\emptyset = 1$ and $t_D = 0$ otherwise.

If $t = x$ where $x \in X_i$ we have $t_D = x$ if $D = \{i\}$ and $t_D = 0$ otherwise.

If

$$g = \sum_i a_i t_i$$

then simply

$$g_D = \sum_i a_i t_{i,D}$$

for any D.

Finally, if

$$g = a \prod_{i \le \ell} t_i$$

then

$$g_D = a \sum t_{1,D_1} t_{2,D_2} \cdots t_{\ell,D_\ell}$$

where the sum is over all tuples $(D_1, \ldots, D_\ell)$ that partition D: the D_i are disjoint, and their union is D. Now we use the assumption that the fan-in ℓ of this multiplication is $\le d$. Hence the number of such partitions is $\le d^d$.

Applying these transformations, the depth of the circuit does not increase as we can merge adjacent Σ and Π gates. The size multiplies by d^{cd}. **QED**

Example 14.26. Let $n = d = 2$, $X_0 = \{0\}$ and $X_1 = \{1\}$ and $x_i \in X_i$. Consider the circuit

$$C := (x_0 + x_1)^2 - x_0^2 - x_1^2.$$

This circuit is homogeneous but not sm or even multilinear, yet it computes the sm polynomial $2x_0x_1$.

Let us illustrate how we transform C into an sm circuit.

Consider the Π gate g computing x_0^2. Then $g_{\{0\}} = x_{0,\{0\}} \cdot x_{0,\emptyset} + x_{0,\emptyset} \cdot x_{0,\{0\}} = x_0 \cdot 0 + 0 \cdot x_0 = 0$.

Consider now $g = (x_0 + x_1)^2$. Then $g_{\{0,1\}} = x_0x_1 + x_0x_1 = 2x_0x_1$. And the other g_D are 0.

Note how non-multilinear terms such as x_0^2 are removed during the process.

Step 2

For this step we further partition the index set $[d]$ as $[d] = T_1 \cup T_2$ where $T_1 = [t], T_2 = [t..d-1]$.

Definition 14.27. Let p be an sm polynomial w.r.t. $D \subseteq [d]$. We define the matrix M_p where the rows are indexed by the $R := \prod_{i \in D \cap T_1} |X_i|$ monomials with variables X_i for $i \in D \cap T_1$ and the columns by the $C := \prod_{i \in D \cap T_2} |X_i|$ monomials with variables X_i for $i \in D \cap T_2$. The m_1, m_2 entry of the matrix is the coefficient of the monomial m_1m_2 in p.

The complexity measure $\mu(p)$ is the rank of M_p normalized by the geometric mean of the two sides:

$$\mu(p) := \frac{\text{rank}(M_p)}{\sqrt{R \cdot C}}.$$

If either $D \cap T_1$ or $D \cap T_2$ is empty the matrix is a row or column matrix, and we can think of either m_1 or m_2 as being the constant 1, and the corresponding product in the denominator to be 1.

Note that $\mu(p) \le 1$ always, since the rank is $\le \min\{R, C\}$ (Fact A.35). The denominator can be written simply as $\sqrt{\prod_{i \in D} |X_i|}$. However thinking of it as a mean of the two sides may help following the argument.

Example 14.28. Suppose there are only two sets of variables, $X = \{x_0, x_1, \ldots\}$ and $Y = \{y_0, y_1, \ldots\}$ ($d = 2, t = 1$) with $|X| = |Y|$. The inner product polynomial

$$p := \sum_i x_i y_i$$

is sm and has M_p equal to the identity matrix: $(M_p)_{i,j} = 1$ if $i = j$ and 0 otherwise. Hence $\mu(p) = |X|/\sqrt{|X| \cdot |Y|} = |X|/|X| = 1$.

Consider a sm polynomial ℓ of degree 1. All the variables belong to one set X_i, and $D = \{i\}$. Then M_ℓ is a vector, of rank 1. So $\mu(\ell) = 1/\sqrt{|X_i|}$.

The hard polynomials p will have large μ. For this, ideally we would like to have the dimensions R and C equal. We then could pick M_p to have full rank, for example a diagonal or permutation matrix. As in Example 14.28, this would give

$$\mu(p) = \frac{R}{\sqrt{\prod_{i \in T_1} |X_i| \cdot \prod_{i \in T_2} |X_i|}} = \frac{\prod_{i \in T_1} |X_i|}{\prod_{i \in T_1} |X_i|} = 1.$$

As we remarked this value of μ is maximum.

However, we will not be able to set exactly $R = C$ due to rounding issues. We now explain how the parameters are set instead. We let X_i with $i \in T_1$ have size m and the others have size $m^{1-\delta}$ where

$$\delta := 0.5/\sqrt{d}.$$

This is possible if m is a power of 2 and $d = \log m$, which we can assume. Note this step limits the degree to be $\leq c \log n$. The total number of variables is $tm + (d-t)m^{1-\delta}$, and so $m \geq n^c$.

There remains to pick t. In the ideal setting $R = C$ we would have

$$m^t = m^{(1-\delta)(d-t)} \iff t = d - d/(2-\delta).$$

However t also has to be an integer, so we set $t := \lceil d - d/(2-\delta) \rceil$.

Again, for the hard polynomial p we pick a matrix of full rank. Because the two sides R and C are only off by a factor m^c, by our choice of t, we have

$$\mu(p) \geq \frac{\min\{R, C\}}{\sqrt{R \cdot C}} \geq \frac{1}{m^c}. \tag{14.2}$$

The polynomial p has Σ-arithmetic circuits of power size, see Problem 14.4.

Step 3

We now proceed to show that for the circuit μ is much smaller: It will be $\leq 1/m^{c\sqrt{d}}$, compared to the $\geq 1/m^c$ of p.

First we establish some useful properties of μ:

Lemma 14.29. The measure μ enjoys:

[Sub-additivity] If p and q are sm w.r.t. the same D, $\mu(p+q) \leq \mu(p) + \mu(q)$; and

[Multiplicativity] If p_1 and p_2 are sm w.r.t. D_1 and D_2, where $D_1 \cap D_2 = \emptyset$, then $p_1 p_2$ is sm w.r.t. $D_1 \cup D_2$ and $\mu(p_1 p_2) = \mu(p_1) \cdot \mu(p_2)$.

Exercise 14.30. Prove this. For multiplicativity, use Fact A.43.

Consider a $\Sigma\Pi\Sigma\Pi\Sigma$ circuit. Let q be a Π gate closest to the output and write

$$q := f_1 \cdot f_2 \cdot \cdots \cdot f_k$$

where the sum of the degrees of the f_i is $\leq d$ and each f_i is a $\Sigma\Pi\Sigma$ circuit.

We will show that

$$\mu(q) \leq 1/m^{c\sqrt{d}} \tag{14.3}$$

from which we conclude the proof of Theorem 14.20 as follows. By Lemma 14.22, equation (14.3), and the sub-additivity property in Lemma 14.29, the measure of the circuit is

$$\leq d^{cd} s^c / m^{c\sqrt{d}}.$$

As remarked earlier, $m \geq n^c$ and so the ratio is$< 1/m^{c\sqrt{d}}$, smaller than the bound on $\mu(p)$ from 14.2.

There remains to prove equation (14.3) we consider two cases.

Some f_i has degree $\geq \sqrt{d}$. Let $C_{i,j}$ be a $\prod\sum$ sub-circuit of f_i. W.l.o.g. $C_{i,j}$ has the same degree as f_i (which is $\geq \sqrt{d}$). Since f_i is homogeneous, subcircuits of smaller degree would cancel and so can be set to 0.

We then have $\mu(C_{i,j}) \leq 1/m^{c\sqrt{d}}$. To verify this recall from Example 14.28 that an sm Σ circuit w.r.t. $\{i\} \subseteq [d]$ has measure $1/\sqrt{|X_i|}$. The bound then follows by the multiplicativity property from Lemma 14.29, and because we are multiplying $\geq \sqrt{d}$ Σ circuits, and the minimum size of an X_i is $m^{1-\delta}$.

By the sub-additivity property we then get $\mu(f_i) \leq s/m^{c\sqrt{d}} \leq 1/m^{c\sqrt{d}}$, using that $s \leq m^{c\sqrt{d}}$, else there is nothing to prove.

Finally, using multiplicativity again we obtain $\mu(q) \leq 1/m^{c\sqrt{d}}$, as desired.

Every f_i has degree $\leq \sqrt{d}$. This is where we use the unbalance of the sets X_i with $i \in T_1$ and T_2. At the high-level, the idea is simple. Recall the target matrix is square, and has measure 1, which is the maximum. For the circuits we are considering, the matrix is "made up" of many small pieces. The size of the pieces are chosen so that few pieces don't make a square. Specifically, the difference in the side lengths translates into a bound on μ for each f_i, showing that μ is significantly smaller than 1. By multiplicativity of μ, the bound for q will be the product of these bounds, giving something substantially smaller than 1.

Let f_i be sm w.r.t. D_i and let $a_i := |D_i \cap T_1|$ and $b_i := |D_i \cap T_2|$. The matrix $M(f_i)$ has sides m^{a_i} and $m^{(1-\delta)b_i}$. Because the rank of $M(f_i)$ is at most the minimum of the sides, we have that $\mu(f_i)$ is at most the minimum of $\sqrt{\frac{m^{a_i}}{m^{(1-\delta)b_i}}}$ and $\sqrt{\frac{m^{(1-\delta)b_i}}{m^{a_i}}}$, which is

$$\leq \frac{1}{m^{0.5|a_i - b_i(1-\delta)|}}.$$

We now argue that this is small, using that a_i and b_i are naturals summing to $\leq \sqrt{d}$. Specifically, we claim

$$|a_i - (1-\delta)b_i| \geq c(a_i + b_i)/\sqrt{d}.$$

Using this claim we get

$$\mu(f_i) \leq \frac{1}{m^{c(a_i+b_i)/\sqrt{d}}}.$$

By the multiplicativity property from Lemma 14.29 we then have

$$\mu(q) \leq \prod_i \frac{1}{m^{c(a_i+b_i)/\sqrt{d}}} \leq \frac{1}{m^{cd/\sqrt{d}}} \leq \frac{1}{m^{c\sqrt{d}}},$$

as desired.

There remains to prove the claim. Writing $a = a_i$ and $b = b_i$ for simplicity, we proceed by case analysis.

If $a \geq b$ then the lhs $|a - (1-\delta)b| = a - b + \delta b \geq 0.5\delta(a-b) + \delta b \geq 0.5\delta(a+b)$.

If $a < b$ then we also have $a < (1-\delta)b = b - \delta b$ because $\delta b < 1$ and a, b are naturals. Hence the lhs is $(1-\delta)b - a = b - a - \delta b \geq 1 - \delta b \geq \delta b \geq 0.5\delta(b+a)$, using that $\delta b \leq 1/2$.

14.7 Problems

Problem 14.1. Prove that Theorem 14.19 is false over $\mathbb{F}_2$.

Problem 14.2. Improve the number of wires in Theorem 14.4 to $cn(\log n/\log\log n)^2$ (which is tight). Hint: Follow the proof of Theorem 14.4 and balance the number of wires in the two layers.

Problem 14.3. Suppose $f : [2]^n \to [2]$ has circuits of size s. Show that the polynomial $\sum_{a\in[2]^n} f(a) \prod_{i:a_i=1} x_i$ has Σ-arithmetic circuits of size csn.

Problem 14.4. Prove that the polynomial in Theorem 14.20 has Σ-algebraic circuits of power size. Hint: Use Problem 14.3.

14.8 Notes

The complexity of linear transformations was studied independently in [146, 351]. Theorem 14.4 is from [123]. More generally they prove tight bounds for any constant depth; for depth 2 see Problem 14.2. They also show that depth $\log^* n$ suffices for a linear number of wires.

For the connection between computing integers and factoring see [308, 234].

Regarding arithmetic circuits, the removal of division instructions is from [329]. Impossibility results for univariate polynomials were first studied in [330]. For more on this see the book [73]. 14.12 is from [48]; we presented the simpler proof in [258]. 14.10 is from [328].

Σ-arithmetic circuits and their relations to the permanent are from [350]. For the completeness of the determinant see [349]. The connection with computing factorials, Theorem 14.15, is from [217, 72].

A general depth reduction theorem was first established in [183], but the size bounds were not controlled. Theorem 14.17, where the size is simultaneously controlled, is from [352] (see discussion in [352] for more on the history). The study of depth reduction in arithmetic complexity was rekindled by [10] which was followed by several other works including [151, 336], whose combination proves Theorem 14.18. A formulation for any depth appears in [232].

The construction of small depth-3 circuits for the elementary symmetric polynomials, Theorem 14.19 appears in [315], where an unpublished work is credited. Several related constructions, also involving some of the ideas that go into Claim 14.23, appear in [315]. For more on the power of constant-depth arithmetic circuits see [27].

The lower bounds over small fields in section 14.6.2 are from [147].

The lower bound over large fields, Theorem 14.20, are from [231], which builds on a long line of works, see [232] for discussion. The complexity measure originates in [272]. The transformation to sm in the proof of Lemma 14.22 only works for large characteristic, due to the factors arising in Fact A.49 and related steps. [112] proves the transformation over any field, although whether circuits can be made homogeneous over any field remains open, see [112] for more discussion.

For more on arithmetic complexity see [73, 71, 316, 301].

Chapter 15

Structures

Data structures aim to maintain data in memory so as to be able to support various operations, such as answering queries about the data, and updating the data. The study of data structures is fundamental and extensive. We distinguish and study in turn two types of problems: *static* and *dynamic*. In the former the input is given once and is not modified by the queries. In the latter queries can modify the input; this includes classical problems such as supporting insert, search, and delete of keys.

15.1 Static

In a static data-structure problem, refer to figure 15.1, we have an input of n bits about which we would like to answer m queries. The data structure aims to accomplish this by storing the input into s words of memory, where each word is w bits. This is accomplished via an arbitrary map g, with no bound on resources. But after that, the queries can be answered by a very efficient map h: each query only depends on t words (this is like a t-local function Definition 9.22 but over word instead of bits). In general, these words can be read adaptively, as in a decision tree (Definition 9.22). But for simplicity we focus on the local setting in which they are fixed by the data structure and the same for every input $x \in [2]^n$. This restriction suffices for the data structures we exhibit, and no stronger negative results are currently known for non-adaptive then for adaptive data structures.

Definition 15.1. A *static data-structure problem* is a function $f : [2]^n \to [q]^m$. A data structure for f with word-space s, word size w and time t is a decomposition

$$f(x) = h(g(x))$$

where $g : [2]^n \to [2^w]^s$, $h : [2^w]^s \to [q]^m$, and each of the m output coordinates of h depends on $\leq t$ input words.

We say *word-space* to emphasize s refers to the number of words, not bits as in Chapter 6. The *bit-space* of the structure is sw.

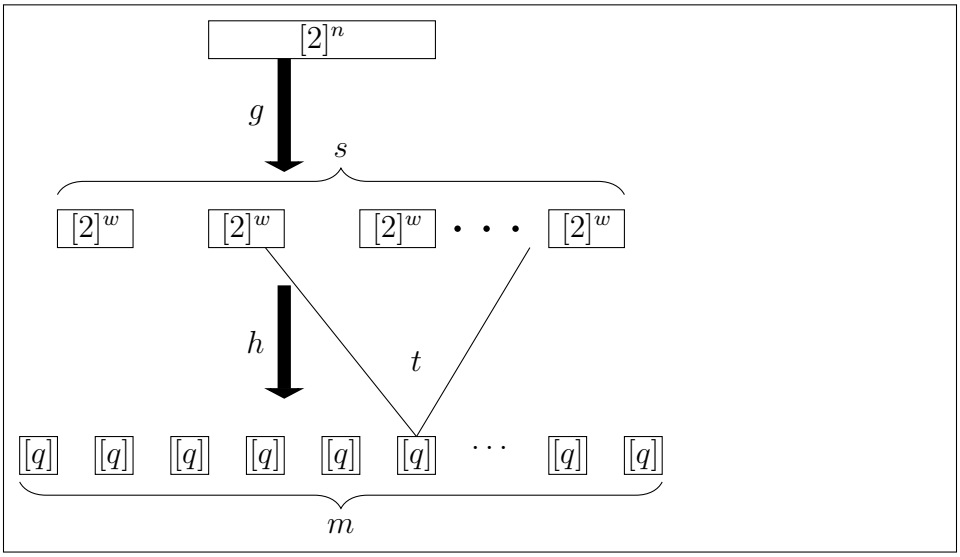


Figure 15.1: The static data-structure problem.

Exercise 15.2. Consider the static data-structure problem $f : [2]^n \to [2]^m$ where $q := 2$, $m := n^2$ and query $(i, j) \in \{1, 2, \ldots, n\}^2$ is the parity of the input bits from i to j. Give a data structure for this problem with word size $w = 1$, space $s = n$, and time $t = 2$. Hint: Exercise 7.38

Just like any function can be computed in brute force by a circuit (Theorem 2.5), any static data-structure problem admits "brute-force" data structures which use very large space or time:

Exercise 15.3. Let $f : [2]^n \to [2]^m$ be a static data-structure problem. Show that f has data structures with word size $w = 1$ and either of the following setting:
(1) Space $s = m$, and time $t = 1$;
(2) Space $s = n$ and $t = n$.

Again similarly to the case of circuits Theorem 2.9, we can prove the existence of static data-structure problems that require either very large space or time.

Exercise 15.4. Prove that for every n and $m \leq 2^{n/2}$ there exists a static data-structure problem $f : [2]^n \to [2]^m$ s.t. any data structure for f using space $s = m/2$ and $w = 1$ requires time $t \geq n - c$.

As usual, the grand challenge is to exhibit explicit (say in FP) static data-structure problems for which these lower bounds hold. We present next the best-known impossibility results. To set the stage, we first establish an impossibility result via a simple reduction to communication protocols (see Chapter 13).

Lemma 15.5. Suppose a static data-structure problem $f : [2]^n \to [q]^m$ has a data structure with word size w, space s, and time t. Then the function $f' : [2]^n \times [m] \to [q]$ defined as $f'(x, y) := f(x)_y$ has communication complexity $\leq ct(\log s + w)$.

Proof. The party who knows $y \in [m]$ communicates the indices of the t locations $f(x)_y$ depends on. This takes $ct \log s$ bits. Then the other party who knows x communicates the w-bit values of those t locations. **QED**

Hence, if f' requires communication $c \log m$ we obtain

$$t \geq c \frac{\log m}{\log s + w}.$$

This bound is meaningful when m is large; but in typical settings where $n \leq s \leq m \leq n^c$ it gives nothing.

The next result refines this bound by replacing the term $\log s$ in the denominator with $\log(s/n)$. This makes for a meaningful bound as long as s is close to n. However, remarkably settings such as $s = n^2$ or even $s = n^{1.01}$ remain *terra incognita*. The lower bound is for the *polynomial evaluation* static data-structure problem (cf Theorem 11.5). Here, the n input bits are viewed as a list of n/w w-bit coefficients of a univariate polynomial over the field $\mathbb{F}_{2^w}$. The queries are then the evaluation of the polynomial over the field elements. For simplicity, we assume that n is a power of 2 and set $w := 10 \log n$, in which case polynomial evaluation is a static data-structure problem $f : [2]^n \to [n^{10}]^{n^{10}}$.

Theorem 15.6. Any data structure for the polynomial evaluation static data-structure problem $f : [2]^n \to [n^{10}]^{n^{10}}$ with word size $w = 10 \log n$ using word-space s and time t has:

$$t \geq c \frac{\log n}{\log(ws/n)}.$$

Proof. We show that an efficient data structure for f implies the existence of a set S of $< u := n/w$ memory words that still allows us to answer $\geq u$ queries. This is impossible, because any u queries can take any tuple of values in $(\mathbb{F}_{2^w})^u$, but we cannot get all those combinations from less than u memory words (of w bits). The subset S is constructed via the probabilistic method. Let $p := 1/n^{2t}$ and include each memory word in S with probability p, independently. We have

$$\mathbb{P}[|S| \geq u] \leq \binom{s}{u} p^u \leq \left(\frac{esp}{u}\right)^u$$

by Fact A.7.

We can use the data structure to answer a query if all the memory words it depends on are in S. For a fixed query, the probability that this happens is $\epsilon := p^t = 1/n^2$. We now claim that with prob. $\geq 0.5\epsilon$ we can answer $0.5\epsilon n^{10} \geq n^5 \geq u$ queries. Indeed, let B be the number of queries we cannot answer. We have $\mathbb{E}[|B|] \leq n^{10}(1-\epsilon)$. And so by Fact A.13 we have

$$\mathbb{P}\left[|B| \geq n^{10}(1 - 0.5\epsilon)\right] \leq \frac{1-\epsilon}{1-0.5\epsilon} \leq 1 - 0.5\epsilon.$$

(To verify the last inequality note that $(1-0.5\epsilon)^2 = 1 + 0.5^2\epsilon^2 - \epsilon \geq 1 - \epsilon$.)

Hence, we can find the set S and reach a contradiction if

$$\left(\frac{esp}{u}\right)^u \leq 0.5/n^2.$$

For large enough n, this inequality is implied by $sp \leq cu$, because $u = n/w = n/(10 \log n)$. Hence

$$sp \geq cu \Rightarrow \left(\frac{ws}{cn}\right)^{2t} \geq n,$$

and the result follows by taking logarithms. **QED**

It is natural to wonder whether a stronger bound holds, and in fact related conjectures have been made, see Chapter 18. Surprisingly, the tradeoff in Theorem 15.6 is not far from tight: There are data structures whose space approaches the optimal n while enjoying moderate time.

Theorem 15.7. The polynomial evaluation static data-structure problem $f : [2]^n \to [n^{10}]^{n^{10}}$ has data structures with word size $w = 10 \log n$, space $n^{1+\delta}$, and time $\log^{c\delta} n$.

See the notes for a proof, and the notes and Chapter 18 for further discussion.

15.1.1 The power of randomness

We illustrate the power of randomness in data structures via a beautiful example that ties together expander-like graphs and data structures. We consider the *membership* problem

$$f : [2]^n \to [2]^m$$

of encoding subsets $X \subseteq [m]$ of size $|X| = b$ while supporting membership queries, i.e., $f(X)_i = 1 \iff i \in X$. Here $n = \lceil \log \binom{m}{b} \rceil$, but it is more natural to parameterize this problem by m and b. We shall consider $b \leq m^{1-c}$, for example $b = \sqrt{m}$,and give a *randomized data structure* that answers query in time *one* and using space within a constant factor of n, i.e., $s \leq cb \log m$.

A randomized data structure can be defined as in Definition 15.1 except that we have a distribution on functions h (but a single encoding map g). However, we in fact prove a stronger result. We give a single data structure which has larger time D, however the computation done by h is gap majority (cf Definition 9.42). Hence, we can answer f_i by outputting a uniform bit queried by g_i. Throughout, the word size is $w = 1$.

To specify the function g in the data structure it suffices to specify the underlying graph of probes (since as we said the function computed is gap majority). For this, we take a bipartite graph with s nodes on the left, m nodes on the right, right degree D, with the following expansion-like property: any set of size $k \leq 2b$ has $\geq 0.9Dk$ distinct neighbors.

Here's how a set X of size b is encoded in memory (this defines g in Definition 15.1). First, we set all the memory to 0. Then we perform the following for stage $i = 0, 1, \ldots$. Let B_i be the set of elements in $[m]$ that are not answered correctly. If $B_i \neq \emptyset$, set all their

neighbors to the correct value, and repeat. This process defines a sequence of bad sets as follows:

$$X = B_0 \supseteq B_2 \supseteq B_4 \supseteq \dots$$
$$[m] - X \supseteq B_1 \supseteq B_3 \supseteq B_5 \supseteq \dots.$$

Exercise 15.8. Show that the containments are correct.

The main claim is that this process terminates, so that at the end there is no mistake.

Claim 15.9. $|B_{i+1}| \leq |B_i|/2$.

Proof. Every element in B_{i+1} has $\geq D/3$ neighbors in B_i (This is because $B_{i+1} \subseteq B_{i-1}$ as noted above, and all the neighbors of B_{i-1} were set to the right value.) Suppose that $|B_{i+1}|$ were larger than $|B_i|/2$. We would have that the set Y consisting of B_i and $|B_i|/2$ elements from B_{i+1} has $\leq D|B_i| + (2/3)D|B_i|/2 = D(4/3)|B_i|$ neighbors. But Y has size $|B_i|3/2$, so by the property of the graph (note $|B_i|3/2 \leq 2b$) it should have $\geq 0.9D|B_i|3/2$ neighbors, which is a contradiction because $0.9 \cdot 3/2 > 4/3$. **QED**

Finally, we use the probabilistic method to show the existence of suitable graphs. The graph is chosen at random. It suffices to bound the prob. that there is some $k \leq 2b$ and a subset $X \subseteq [m]$ of size k and a subset $T \subseteq [s]$ of size $\leq 0.9Dk$ such that all the neighbors of X fall in T. By a union bound this probability is at most

$$\begin{aligned}
\sum_{k=1}^{2b} \binom{m}{k} \binom{s}{0.9Dk} \left(\frac{0.9Dk}{s}\right)^{Dk} &\leq \sum_{k=1}^{2b} \left(\frac{em}{k}\right)^k \left(\frac{es}{0.9Dk}\right)^{0.9Dk} \left(\frac{0.9Dk}{s}\right)^{Dk} \quad \text{(by A.7)} \\
&\leq \sum_{k=1}^{2b} \left(\frac{em}{k}\right)^k e^{0.9Dk} \left(\frac{0.9Dk}{s}\right)^{0.1Dk} \\
&\leq \sum_{k=1}^{2b} \left(\frac{em}{k}\right)^k \left(\frac{1}{2em}\right)^k \quad \text{for } s = cb\log m, D = c\log m \\
&\leq \sum_{k=1}^{2b} \left(\frac{1}{2}\right)^k \\
&< 1.
\end{aligned}$$

Note for the randomized data structure it is not important if D is small, but the construction of the graph needs $s \geq Dk$ and $D \geq \log m$.

15.1.2 Succinct

Succinct data structures are those where the space is close to the input length n (which is the minimum, for any one-to-one problem). When the word size is $w = 1$ this can be

expressed as $s = n + r$ for some small r, called *redundancy*. In the succinct setting we can show a lower bound that is stronger than Theorem 15.6. The next theorem states such a lower bound for the static data-structure problem corresponding to a good error-correcting code, see Problem 3.6 for the definition. Here we identify a code with its encoding map.

Theorem 15.10. Let $f : [2]^n \to [2]^m$ be an $(m, \epsilon m, \epsilon m)_2$-code. Any data structure for f with $w = 1$ using space $n + r$ requires time $\geq c_\epsilon n/r$.

For example, if the data structure uses space $n + \sqrt{n}$ (i.e., the redundancy r is $\sqrt{n}$) then the time is $\geq n^c$. By contrast, if the space is $2n$ the best time lower bound we can prove for explicit data-structure problems is logarithmic Theorem 15.6.

Surprisingly, there are good error-correcting codes which admit data structures nearly matching Theorem 15.10.

Theorem 15.11. There are (m, cm, cm) codes $f : [2]^n \to [2]^m$ which have data structures with $w = 1$, space $n + r$, and time $c(n/r) \log^3 n$, for any m, r.

Such codes can be constructed from the code in Theorem 14.4.

Exercise 15.12. Do this.

This construction of a data structure from a circuit holds generically. It implies that proving a data-structure time lower bound of $(n/r) \log^c n$ (where r is the redundancy) for an explicit linear static data-structure problem gives new circuit lower bounds for constant-depth xor-circuits. Similarly, such a lower bound for any (not necessarily linear) explicit data-structure problem gives new circuit lower bounds for constant-depth circuits with gates computing arbitary functions. For more on these connections see the notes.

In the same spirit, but with somewhat looser parameters, there is also a simulation of linear-size log-depth circuits (studied in section §9.4.2).

Theorem 15.13. Let $f : [2]^n \to [2]^m$ be a function computable by circuits of size an and depth $a \log n$ for a constant a. Then f has a data structure with word size $w = 1$, space $n + c_a n/\log n$, and time $n^{0.01}$.

In particular, proving $n^{0.1}$ time lower bounds for data structures using space $1.01n$ would give functions that cannot be computed by linear-size log-depth circuits.

15.1.3 Succincter

In this section we present the solution to the trits problem from section 0.1.4. It can be viewed as a static data-structure problem $f : [2]^n \to [3]^m$ where $n = \lceil \log_2 3^m \rceil$.

Theorem 15.14. The trits problem (see section 0.1.4) has a data structure with word size $w \leq \log m$, bit-space $\lceil m \log_2 3 \rceil$, and constant time

Next we present the proof.

Definition 15.15. [Encoding and redundancy] An *encoding* of a set A into a set B is a one-to-one (a.k.a. injective) map $f : A \to B$. The *redundancy* of the encoding f is $\log_2 |B| - \log_2 |A|$.

We shall use routinely that if an encoding $A \to B$ can be written as the composition of two encodings $A \to C \to B$ then the redundancy of f is at most the sum of the redundancies of the two encodings.

Exercise 15.16. Prove this.

The following lemma gives the building-block encoding we will use.

Lemma 15.17. For all sets X and Y, there is an integer b, a set K and an encoding

$$f : X \times Y \to [2]^b \times K$$

with redundancy $\leq c/\sqrt{|Y|}$ s.t. $x \in X$ can be recovered just by reading the b bits in $f(x, y)$, for any y.

Exercise 15.18. Prove $|K| \leq c|Y|$ in any such encoding.

The basic idea for proving the lemma is to break Y into $C \times K$ and then encode $X \times C$ by using b bits:

$$X \times Y \to X \times C \times K \to [2]^b \times K.$$

We will first pick b and then try to stuff as much as possible inside $[2]^b$. Our choice of b will make $|C|$ about $\sqrt{|Y|}$.

Proof. Assume $|Y| > 1$ without loss of generality. Define $b := \left\lceil \log_2 \left(|X| \cdot \sqrt{|Y|} \right) \right\rceil$, and let $B := [2]^b$. To simplify notation, define $d := 2^b/|X|$. Note $c\sqrt{|Y|} \leq d \leq c\sqrt{|Y|}$.

How much can we stuff into B? For a set C of size $\lfloor |B|/|X| \rfloor$, we can encode elements from $X \times C$ in B. The redundancy of such an encoding can be bounded as follows:

$$\begin{aligned}
&\log |B| - \log |X| - \log |C| = \\
=&\log \frac{2^b}{|X|} - \log \left\lfloor \frac{2^b}{|X|} \right\rfloor \\
=&\log d - \log \lfloor d \rfloor \\
\leq&\log d - \log(d-1) \\
=&\log \left(1 + \frac{1}{d-1} \right) \\
\leq&\frac{c}{d-1} \\
\leq&\frac{c}{\sqrt{|Y|} - 1} \\
\leq&\frac{c}{\sqrt{|Y|}},
\end{aligned}$$

where the second inequality is Fact A.5.

To calculate the total redundancy, we still need to examine the encoding from Y to $C \times K$. Choose K of size $\lceil |Y|/|C| \rceil$, so that this encoding is possible. With a calculation similar to the previous one, we see that the redundancy is:

$$\begin{aligned}
&\log |C| + \log |K| - \log |Y| \\
=&\log \left\lceil \frac{|Y|}{|C|} \right\rceil - \log \frac{|Y|}{|C|} \\
\leq&\log \left(1 + \frac{|C|}{|Y|}\right) \\
\leq&c \frac{|C|}{|Y|} \\
\leq&c \frac{\left\lfloor \frac{2^b}{|X|} \right\rfloor}{|Y|} \\
\leq&c \frac{2^b}{|X| \cdot |Y|} \\
\leq&c \frac{2^{\log\left(|X| \cdot \sqrt{|Y|}\right)}}{|X| \cdot |Y|} \\
=&c \frac{1}{\sqrt{|Y|}}.
\end{aligned}$$

The total redundancy is then $c/\sqrt{|Y|}$. By construction, $x \in X$ can be recovered from the element in B only. **QED**

We can now prove Theorem 15.14 by repeated applications of Lemma 15.17.

Proof of Theorem 15.14. Break the m input trits x into blocks of size $t := a \log m$ for a constant a to be set later: $x = (x_1, x_2, \ldots, x_{m/t}) \in T_1 \times T_2 \times \ldots \times T_{m/t}$, where $T_i = [3]^t$ for all i. We assume for simplicity that m is a power of 2 and t divides m; otherwise we can treat the last block differently.

The encoding can be drawn as

$$\begin{array}{ccccccccccccccccc}
 & & T_2 & & & & T_3 & & & & & & T_{m/t} & & & & \\
 & & \downarrow & & & & \downarrow & & & & & & \downarrow & & & & \\
T_1 & \to & \square & \to & K_1 & \to & \square & \to K_2 & \to & \cdots & \to & K_{m/t-2} & \to & \square & \to & K_{m/t-1} \\
 & & \downarrow & & & & \downarrow & & & & & & \downarrow & & & \downarrow & \\
 & & B_1 & & & & B_2 & & & & & & B_{m/t-1} & & & B_{m/t} &
\end{array}$$

where each square is the encoding in Lemma 15.17, and the last encoding is the arithmetic one, i.e., encode $K_{m/t-1}$ in $2^{\lceil \log |K_{m/t-1}| \rceil}$.

To retrieve a trit, we retrieve the entire block $x_i \in T_i$ of t trits. For this, it suffices to read $y_{i-1} \in B_{i-1}$ and compute $k_{i-1} \in K_{i-1}$. By Lemma 15.17 $k_{i-1} \in K_{i-1}$ can be computed by

reading only $y_i \in B_i$. Thus to complete the proof it suffices to show that each y_i has length $\leq ct$. This is not completely obvious because $|K_i|$ may vary with i. However, Exercise 15.18 guarantees that $|K_i| \leq c|T_i| = c3^t$. Hence, each application of Lemma 15.17 is to sets of size $\leq c3^t$. Since the encoding in Lemma 15.17 has redundancy ≤ 1, we have $|B_i| \leq 2^{ct}$.

To bound the redundancy, note that from (1) in Lemma 15.17, the composition of the encodings except the last one has redundancy $\leq \left(\frac{m}{t} - 1\right) \cdot \frac{1}{2^{ct}} \leq \frac{m}{2^{ct}} \leq \frac{1}{m^{c_a}}$. Together with the final arithmetic encoding we use bit space

$$\left\lceil m \log_2 3 + \frac{1}{m^{c_a}} \right\rceil \leq \lceil m \log_2 3 \rceil + 1.$$

This proves the theorem except for the "+1."

In fact, the "+1" is not needed, because the distance of $m \log_2 3$ to an integer is $\geq 1/m^c$, which is more than $1/m^{c_a}$ for large enough a. Proving this bound on the distance however is apparently complicated, and we refer to the notes for it. **QED**

The proof gives an *exponential* tradeoff between time and redundancy. For a simpler *power* tradeoff see Problem 15.2.

15.1.4 Impossibility results by ruling out samplers

One can show impossibility results for succinct data structures by ruling out *samplers*. One shows that efficient (for example local) samplers fail to sample the distribution over the queries *even with very large statistical distance*. If the statistical distance is larger than $1 - 2^{-r}$ then redundancy $\geq r$ is required. This connection is formalized in the following exercise.

Exercise 15.19. Suppose $f : [2]^n \to [2]^m$ has a data structure with word size $w = 1$, time t, and redundancy r. Show that there is a t-local map whose output distribution has statistical distance $\leq 1 - 2^{-r}$ from $f(U)$.

To illustrate, we now state and prove a negative result for sampling m trits.

Theorem 15.20. Let T be the uniform distribution over m trits (where e.g. each trit is represented as 2 bits). Let f be a t-local map. The statistical distance between T and the output distribution of f is $\geq 1 - 2^{-c_t m}$.

By Exercise 15.19 this implies that data structure with word size $w = 1$ and time t for the trits problem requires linear redundancy $\geq c_t m$. This stands in sharp contrast with Theorem 15.14 which gives constant time and redundancy with logarithmic word size.

Proof. First observe that just looking at one trit one has statistical distance $\geq c_t$. This is because the probability that a trit is 0 in T is $1/3$, but for a local map this probability is a power of 2. The idea is to boost this probability by looking at many independent output bits of f. We proceed by induction on t. The base case $t = 1$ is left as exercise. For the

inductive step, suppose there are $s := c_t m$ output bits of f that depend on disjoint input bits, and so are in particular independent. Then the result follows by the observation and tail bounds Lemma A.15 (exercise). Otherwise, as in the proof of Lemma 9.24, there is a set of $\leq st$ input bits s.t. any output bit depends on one of them. By fixing these bits, we can write the output distribution of f as a convex combination of $\leq 2^{st}$ distributions, each samplable with smaller locality $t-1$. By Problem 15.3 and induction the statistical distance of f from T is $\geq 1 - 2^{st} \cdot 2^{-c_t m}$. **QED**

15.2 Dynamic

We now study dynamic data structures, where the input is not fixed but can be modified by the queries. Unlike the static setting, here we directly consider adaptive data structures because adaptivity is more critical in the dynamic setting. We first define a simple and powerful non-uniform extension of word programs (Definition 1.1), called *tree programs*, which allows us to focus on read and write operations, ignoring all other details of the model. A tree program is like a decision tree Definition 9.22 except that the alphabet is $[2^w]$ instead of $[2]$, where w is the word size, and moreover it can also write in memory. To model writing in a simple way, we let the tree output a tuple of words which are the contents written at the locations read along a path.

Definition 15.21. [Tree program] A *tree program* with word size w and depth t is a complete tree of degree 2^w and depth t where each internal node is labeled with an index $i \in [2^w]$ and each leaf is labeled with an output tuple $y \in [2^w]^t$. The tree computes on a memory array $M \in [2^w]^{2^w}$ consisting of 2^w words of w bits as follows. Starting at the root, we read the memory word $M[i]$ labeled at the node, and move to the corresponding child. We repeat this until we arrive at a leaf with label $y = (y_0, y_1, \ldots, y_{t-1})$. Finally, the t memory words $i_0, i_1, \ldots, i_{t-1}$ read along the the path are overwritten with y: $M[i_0] \leftarrow y_0, M[i_1] \leftarrow y_1$.

One can consider larger memory arrays, which can be useful since in time t we can use 2^{wt} words. However this can be simulated by increasing w, which is supported in the proofs.

We now define dynamic data structures. For simplicity, we consider boolean functions, since this will suffice for our examples, but one can easily extend the definitions to non-boolean functions.

Definition 15.22. [Dynamic data structure] A *dynamic data-structure* for $f : [2]^n \to [2]^m$ with word size w and time t is a collection of $2n + m$ tree programs with word size w and depth t of the following types:

Update(i, b) for $i \in \{1, 2, \ldots, n\}$ and $b \in [2]$, which sets bit i of the input to b, and

Query(i) for $i \in \{1, 2, \ldots, m\}$ which computes bit i of f evaluated at the current input (say in memory word 0).

Starting with all-zero memory M and all-zero input $x \in 0^n$, any sequence of Update programs followed by a Query(i) program should result in $M[0] = f(x)_i$, where x is the input corresponding to the updates.

For dynamic data structures we can prove impossibility results for time t about logarithmic. We illustrate this in a simple example: Encoding a good error correcting code (see Problem 3.6 and section 14.1.1).

Theorem 15.23. Let $f : [2]^n \to [2]^m$ be (the encoding function of) a binary $(m, n, d)_2$ code with n and d both $\geq \epsilon m$. Any dynamic data-structure for f with word length $w \leq a \log m$ requires time

$$\geq c_{\epsilon,a} \frac{\log m}{\log \log m}.$$

Proof. We write $C(i)$ for Query(i). Pick $x \in [2]^n$ uniformly and $i \in \{1, 2, \ldots, m\}$ uniformly, and consider the sequence of operations

$$\text{Update}(1, x_1), \text{Update}(2, x_2), \ldots, \text{Update}(n, x_n), C(i).$$

That is, we set the message to a uniform x one bit at a time, and then ask for a uniformly selected bit of the associated codeword $f(x)$ which we denote by $C_x := (C_x(1), C_x(2), \ldots, C_x(m)) \in [2]^m$.

We use the so-called *chronogram* technique whose intuition is as follows. We divide the n updates into consecutive blocks, called *epochs,* whose sizes decreases rapidly, and consider the set of memory cells written in each epoch. There will be about $\log n$ epochs. Essentially we are going to show that to answer $C(i)$ we must make read at least one cell from those written at each epoch, which gives the time lower bound bound. However, the memory locations written in different epochs may not be disjoint, so instead we are going to consider the set L_e of memory locations that were written *last* at epoch e. Note that the L_e are disjoint. So if we show that $C(i)$ must read locations from L_e for every e, we obtain the lower bound. To prove the latter, we fix a particular epoch e, and all the updates that were made before that. If $C(i)$ has a small probability of reading locations in L_e then C_x is close to a string that does not depend on the updates in epoch e, which can be shown to contradict the properties of the code. And this remains true even taking into account updates in future epochs, since there are so few of them compared to those in epoch e. Details follow.

Epoch e consists of n/w^{3e} operations. Hence there are $\geq c \log_w n$ epochs, and we can assume that we have exactly this many epochs (by discarding some bits of n if necessary). The geometrically-decaying size of epochs is chosen so that the number of message bits set during an epoch e is much more than all the memory words written by the data structure in future epochs. A key idea of the proof is to see what happens when the memory words written during a certain epoch are reverted to their contents right before the epoch. Specifically, after the execution of the first n operations of type M, we can associate to each memory word the last epoch during which this word was written. Let L_e be the memory words associated with epoch e. Let $D^e(x)$ denote the memory after the n Update operations, but with the change that the words L_e are replaced with their contents right before epoch e. Define $C^e_x(i)$ to be the program $C(i)$ run on memory $D^e(x)$, and $C^e_x = C^e_x(1), C^e_x(2), \ldots, C^e_x(n)$.

Let $t(x,i,e)$ equal 1 if $C(i)$ reads a memory word in L_e, 0 otherwise. The time of the data structure is at least

$$\max_{x,i} \sum_e t(x,i,e) \geq \mathbb{E}_{x,i} \sum_e t(x,i,e) = \sum_e \mathbb{E}_{x,i} t(x,i,e) \geq \sum_e \mathbb{E}_x \Delta(C_x, C_x^e), \tag{15.1}$$

where $\Delta(C_x, C_x^e)$ denotes the relative distance $\mathbb{P}_i[C_x^e(i) \neq C_x(i)]$, and the last inequality holds because $C_x^e(i) \neq C_x(i)$ implies $t(x,i,e) \geq 1$.

We claim that $\mathbb{E}_x \Delta(C_x, C_x^e) \geq c$ for every e. This concludes the proof by summing over all $\geq c \log_w n$ epochs.

To verify the claim, fix arbitrarily the bits of x set before epoch e. For a uniform setting of the remaining bits of x, note that the message ranges over

$$\geq 2^{n/w^{3e}}$$

values. On the other hand, we claim that C_x^e ranges over much fewer values. Indeed, if t is the time of the data structure, the total number of memory words written in all epochs after e is

$$\leq t \sum_{i \geq e+1} n/w^{3i} \leq ctn/w^{3(e+1)}.$$

We can describe all these memory words by writing down their indices and contents, using a total of $B := ctn/w^{3e+2}$ bits. Note that this information can depend on the programs run during epoch e, but the point is that it takes few possible values overall. Since the memory words last changed during epoch e are reverted to their contents before epoch e, this information suffices to describe $D^e(x)$, and hence C_x^e. Therefore, C_x^e ranges over $\leq 2^B$ strings.

Finally, here's where we use the distance of the code. For each string in the range of C_x^e at most two codewords can be at distance $< 0.5am$, for else you'd have two codewords at distance $< am$, violating the distance of the code.

Hence except with probability $2 \cdot 2^B / 2^{n/w^{3e}}$ over x, we have $\Delta(C_x, C_x^e) \geq 0.5a$. We can assume $t \leq w$, else the theorem is proved, in which case the first probability is ≤ 0.1, and so $\mathbb{E}_x \Delta(C_x, C_x^e) \geq c$, proving the claim. **QED**

This bound is not far from the best known bound of $\log^2 n$ for dynamic data-structure problems, see the notes. One might wonder if stronger bounds hold for error-correcting codes. But in fact there exist good codes for which the bounds are nearly tight.

Theorem 15.24. There are binary linear $(m, \epsilon m, \epsilon m)_2$ codes with dynamic data structures with word length $\leq c \log m$ and time $\leq c \log^2 m$.

For a proof see Problem 15.1.

The chronogram technique can prove lower bounds similar to Theorem 15.23 for simpler functions. The *prefix sum* problem asks to maintain an array of n bits while supporting querying the parity of the bits up to i. There are data structures for this problem with logarithmic time. The next theorem shows that this is tight.

Theorem 15.25. Let $f : [2]^n \to [2]^n$ be defined as $f(x)_i := \sum_{j \le i} x_j \bmod 2$. Any dynamic data-structure for f with word length $a \log n$ requires time

$$\geq c_a \frac{\log n}{\log \log n}.$$

For the proof see Problem 15.4.

One can prove lower bounds of about $\log^2 n$ for the polynomial evaluation problem by a combination of the chronogram technique and the sampling technique in Theorem 15.6 Let us sketch it. We proceed as in the chronogram method but rather than showing that on average $C(i)$ reads at least one location in L_e for every e, we will show it reads more: at least about $\log n$. To show this, assume towards a contradiction that on average $C(i)$ reads less. As in the proof of Theorem 15.6, by the probabilistic method we can find a subset $S \subseteq L_e$ whose size is about a $\log n$ fraction of $|L_e|$ and it still allows us to answer n^c queries. For boolean codes, this wouldn't be enough to reach a contradiction, but over larger alphabets and for the specific code given by polynomial evaluation one can still show that for each C^e_x there can only be few close codewords, and then conclude as before.

Open question 15.26. Are there explicit (say in FP) boolean functions $f : [2]^n \to [2]^m$ with m a power of n that require $\log^2 n$ time for dynamic data structures?

15.2.1 Reductions

Naturally, even for data structures there exists a vast web of reductions. We give an example. By a reduction from the prefix sums Theorem 15.25, we can prove a lower bound for maintaining a graph while supporting connectivity queries. Specifically, the *graph connectivity* problem is $f : [2]^m \to [2]^m$ where $m = n^2$ and the input is the $n \times n$ adjacency matrix of a directed graph, and Update adds or remove an edge, and Query(i, j) for $i, j \in [n]$ is 1 iff node i is connected to j.

Theorem 15.27. Any dynamic data-structure for graph connectivity $f : [2]^m \to [2]^m$ with word length $a \log m$ requires time

$$\geq c_a \frac{\log m}{\log \log m}.$$

Proof. We reduce from prefix sums, Theorem 15.25. An array of length n corresponds to a graph with $2n+2$ nodes organized in a $2 \times (n+1)$ matrix; for each 0 in the array we include a $=$ gadget which connects two nodes in a column to the corresponding nodes in the next column, and for each 1 we include a $\times$ gadget which is like $=$ except the rows get swapped.

For example, here's an array and its corresponding graph:

$$\begin{array}{ccccc} 0 & 1 & 1 & 0 & 1 \\ = & \times & \times & = & \times \end{array}.$$

An update in prefix sums corresponds to c updates in graph connectivity. The sum of the bits $< i$ in the array is 1 iff node 0 in the first row is connected to node i in the second row. **QED**

One can also relate the complexity of data structures to problem in communication complexity or various Time complexity classes. For example, one can prove strong dynamic data-structure lower bounds from the 3Sum Conjecture 4.1, see the notes.

15.3 Problems

Problem 15.1. Prove Theorem 15.24. Guideline: Use the code in Theorem 14.4. The data structure is the middle layer. Bound the fan-out of the input bits.

Problem 15.2. [Block-wise arithmetic coding] Give a simple data structure for the trits problem on m trits which retrieves a bit by reading ct bits and using space

$$\frac{m}{t}\lceil t \log_2 3 \rceil \leq m \log_2 3 + \frac{m}{t},$$

assuming t divides n. This provides a *power* trade-off between time and redundancy, but the inequality can be shown to be almost tight (see the notes).

Problem 15.3. Let r and t be distributions over the same domain. Suppose that $r = 2^{-s}\sum_{i=1}^{2^s} p_i$ where each p_i has statistical distance $\geq 1-\epsilon$ from t. The r has statistical distance $\geq 1 - 2^s\epsilon$ from t.

Problem 15.4. Prove Theorem 15.25. Guideline: Follow the proof of Theorem 15.23. Organize the updates as follows. The n/w^{3e} updates in epoch e are performed at locations $i \cdot w^{3e}$ for $i = 1, 2, \ldots, n/w^{3e}$. As in equation (15.1), prove that $\mathbb{E}_{x,i} t(x, i, e) \geq c$ for every e.

15.4 Notes

The connection between data structures and communication complexity is from [251] and was studied more in [253].

The lower bound for polynomial evaluation, Theorem 15.6, is from [317]. It was rediscovered in [226]. The efficient data structures for polynomial evaluation, Theorem 15.7, are from [208]. We note that while we have focused on polynomial evaluation for concreteness, results in the same spirit (i.e., the lower bound in Theorem 15.6 and a nearly matching construction as in Theorem 15.7) were known earlier for hash functions [317], of which polynomial evaluation is a special case.

The randomized data structure for membership in section 15.1.1 is from [70].

The lower bound for succinct data structures (Theorem 15.10) for error-correcting codes is from [124]. The nearly matching data structure is from [371]. The latter also proves generic simulations of small-depth circuits by efficient data structures, including Theorem 15.13.

The breakthrough result on the trits problem is from [280]. After [280] a negative result was proved in [368]. The parameters in [368] were then matched [280, 106] where Theorem 15.14 appears. Our exposition follows [363]. The fact that the "+1" is not necessary in the proof of Theorem 15.14 follows from Theorem 3.1 in [42], cf [368]. For more on Problem 15.2 and the limitations of the approach see [368].

For dynamic data structures, the technique in the proof of Theorem 15.23 is from [117] and has been applied to many other natural problems. It is not far from the state-of-the-art lower bounds, which are of the form $\log^a n$ for a constant $a \leq 2$ [225, 227]. The about $\log^2 n$ bound for polynomial evaluation is from [226]. The dynamic data structure for error-correcting codes, Theorem 15.24, is from [371]. For conditional lower bounds based on the 3Sum Conjecture 4.1 see [281].

Chapter 16

Tapes

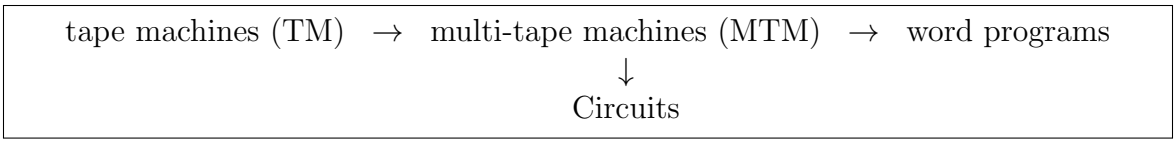


Figure 16.1: Relations between various computational models. An arrow from A to B means that B can simulate A efficiently (from time t to $t \log^c t$). The models in the first row are uniform, circuits are non-uniform.

In this chapter we discuss *tape machines*, a uniform model of computation. They are simpler to define than word programs (Chapter 1) yet give an equivalent definition of P. However, tape machines are unsuitable for capturing running times less than quadratic, and do have as tight a time hierarchy as word programs. Tape machines and word programs both shape our understanding of computation, but in different ways. For a quick summary of some of the relationships we will prove, see figure 16.1.

Tape machines are equipped with an infinite tape (or memory) of symbols from the *tape alphabet* A, and a *tape head* pointing to one symbol. The tape alphabet A consists of the symbols $0, 1$ which are used to encode input and output, and the blank symbol $\_$ which is used to delimit them. Later we shall also consider larger alphabets. If this sounds boring, check out Question 16.9: We still do not understand if larger alphabets give more power!

The machine can read and write the symbol under the head. We think of the machine as starting with an input x on the tape, and the head on the first symbol of x. After some computation, the output would be y if the machine stops with the head on the first symbol of y on the tape (and blank tape elsewhere).

The main differences with the word programs from Chapter 1 is that a tape machine keeps one pointer to the memory, and can only increase or decrease this pointer by 1 in one step. This is a severe type of *locality of computation*, cf Chapter 1, exploited in many proofs. Also, the memory symbols are from a constant-size alphabet, as opposed to larger words. A beautiful aspect of tape machines is that we can give a shorter definition of the model, ignoring the more complicated semantics of word programs and other programming

languages:

Definition 16.1. *A tape machine* (TM) is a map

$$t : S \times A \to S \times A \times \{\text{Left}, \text{Right}, \text{Still}\},$$

called the *transition* map, where S is a finite set of *states,* including two special states *start* and *stop*, and $A := \{0, 1, \_\}$ is the *tape alphabet*. The symbol $\_ \in A$ is called *blank*.

A *configuration* of a TM encodes its state, the tape content, and the position of the head on the tape. It can be written as a triple (s, m, h) where $s \in S$ is the state, $m : \mathbb{Z} \to A$ specifies the tape contents, and $h \in \mathbb{Z}$ indicates the position of the head on the tape.

We write $m[h \leftarrow x]$ for the map m' obtained by m by setting the value at h equal to x, i.e., $m'[h] = x$ and $m'[i] = m[i]$ for $i \neq h$. We adopt the convention that the machine first writes, then moves the head. So, a configuration

$$\begin{aligned}(s, m, h) \text{ with } s \neq \text{stop } \textit{yields} &(s', m[h \leftarrow y], h + 1) \text{ if } t(s, m[h]) = (s', y, \text{Right}),\\ &(s', m[h \leftarrow y], h - 1) \text{ if } t(s, m[h]) = (s', y, \text{Left}),\\ &(s', m[h \leftarrow y], h) \text{ if } t(s, m[h]) = (s', y, \text{Still}).\end{aligned}$$

We say that a TM computes $y \in [2]^*$ on input $x \in [2]^*$ in *time* t (or in t steps) if, starting in configuration $(\text{start}, m, 0)$ where $x = m[0]m[1] \cdots m[|x| - 1]$ and m is blank elsewhere, it yields a sequence of $t + 1$ configurations where the last one is (stop, m', h) and has $y = m[h]m[h + 1] \cdots m[h + |y| - 1]$ and m is blank elsewhere.

Many details are somewhat arbitrary. For example, we allowed the head to remain still. This is redundant, as one can simulate a still operation by a sequence of Right and Left, but makes some programs such as the one in the next example more intuitive (as the machine can write the output and then stop while keeping the head on the output).

Remark 16.2. We defined TMs with a fixed-size alphabet (as opposed any constant size). This choice slightly simplifies the exposition (one less parameter), while being more in line with common experience (it is more natural to increase the length of a program rather than its alphabet). This choice affects the proof of the time hierachy; but the details don't seem any worse.

Example 16.3. We show that parity can be computed by a TM in time $n + 1$ on inputs of length n. This is n steps to read the input, and one to write the output and stop. A machine with 3 states which scans the input once suffices. The start state also corresponds to a parity of 0 in the input so far. Another state, called *one* corresponds to a parity 1 so far. This highlights an important fact: We can use states to store small amount of information. The machine overwrites the input with the blank symbol _ while doing a pass, and flips between states start and one whenever a 1 is read. When the machine reads a blank, it writes the output and stops. We can represent the TM via a diagram as in figure 16.2. However such diagrams quickly become uninformative. Instead, it suffices to give a high-level description of the behavior of the machine.

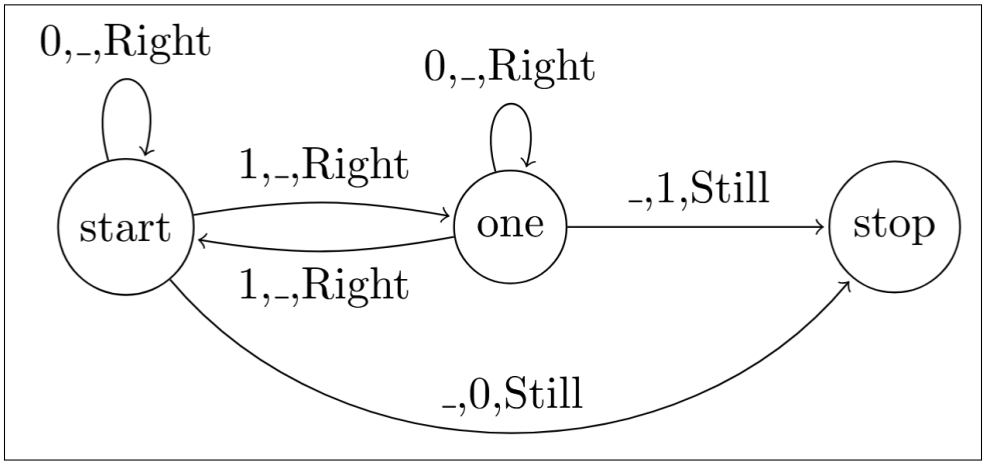


Figure 16.2: Diagram of a TM computing parity. An arrow with label x, y, z from a state s to a state s' indicates that if the machine is in state s and read symbol x from the tape then it writes y on the tape, moves the head according to z, and changes the state to s'. For example, the label from start to one with label $1, \_, \text{Right}$ means that if the machine is in state start and reads a 1 then it overwrites it with blank, moves the head right, and changes state to 1.

Example 16.4. On input an integer $x \in [2]^*$ given in binary, we wish to compute $x + 1$ (i.e., we think of x as an integer in binary, and increment by one). If x is the empty string, we can define $x + 1$ as 0.

This can be accomplished by a TM as follows. Move the head to the least significant bit of x (note the TM starts on the most significant digit of x). If you read a 0, write a 1, move the head to the beginning, and stop. If instead you read a 1, write a 0, move the head left, and repeat.

On an input x of length n, the TM does a constant number of passes over the input, so the running time (the number of steps) is $c|x| + c$. The constant summand is to take into account the empty string. (Without it, the running time would be 0 on the empty string, but that is not correct, as the machine needs at least one transition to stop.)

Example 16.5. We now describe a TM that on input $x \in [2]^*$ computes the length $n := |x|$ in binary, in time $cn \log n + c$. This illustrates the important technique of keeping a counter next to the head. The TM operates by repeating a basic routine. When starting the routine, the memory is organized as $\ldots \_\_s\_x\_\_ \ldots$ where s and x are bit strings, and the head is on the blank symbol between s and x. String s is interpreted as an integer. The routine consists of incrementing s by 1, and then moving the symbols to the left of x right by 1 position (thus overwriting the first symbol in x). To increment s we can use the TM in Example 16.4. The routine is repeated until x is empty, in which case the counter contains the output. To get things started, on input x the machine moves the head left by two positions, writes 0, and then moves the head right.

For example, starting the routine in

__11_100010__

would result in

__100_00010__

note how the counter increased, and the string on the right got shorter.

How long does the routine take? If the counter has ℓ bits, it takes $c\ell + c$ steps to increment it by 1, as in Example 16.4. The same number of steps suffice to move it to the right. Since we count up to n, we know $\ell \leq \log n + c$. So each routine takes $\leq c \log n + c$ steps. This routine is repeated n times, so the total time is $\leq cn \log n + cn + c$, where constant summand takes into account the initialization phase that gets things started.

Exercise 16.6. Describe a TM that decides if an n-bit string $x \in [2]^*$ is palindrome (cf section §0.1.1) in time $cn^2 + c$.

How powerful are tape machines? Perhaps surprisingly, tape machines are robust and all-powerful. *One tape is all you need.* Indeed, we prove below (section 16.2.1) that TMs are power-equivalent to word programs. Thus the power-time computability thesis (section 1.3) can be equivalently stated in terms of TM. However, the quasi-linear computability thesis (section 1.3) for word programs does *not* hold for TMs.

Next we define complexity classes similarly to Definition 16.7. Some of the results in this chapter involve "fine" distinctions among time and space bounds for which it makes a difference if we work with total functions or partial. Hence we specifically define total classes. Also, we not consider relations.

Definition 16.7. [Time complexity classes] Let $t : \mathbb{N} \to \mathbb{N}$ be a function. TM-FTime(t) is the set of functions $f : X \subseteq [2]^* \to [2]^*$ for which there exists a TM M that computes $f(x)$ within $t(|x|)$ steps on any input $x \in X$ of length $\geq |M|$.

The restriction to boolean functions $f : X \subseteq [2]^* \to [2]$ is denoted TM-Time(t). The restriction to total boolean functions $f : [2]^* \to [2]$ is denoted TM-TotalTime(t).

16.1 On the alphabet

As our first example of robustness, we show that TMs using arbitrarily large tape alphabet can be simulated by TMs with the fixed tape alphabet $A = \{0, 1, \_\}$ in Definition 16.1. This might seem like a detail, but in fact it leads to a cute problem in the area which will come up again and is, as far as I know, open. We define TMs *with alphabet* B as in Definition 16.1 but using $B \supseteq A$ instead of A. We define TM-Time($t(n)$) similarly.

Theorem 16.8. TM-Time($t(n)$) with alphabet $B \subseteq$ TM-Time($c_B t(n) + c_B n^2$).

Proof. Given a TM N with alphabet B we construct TM M as follows. We think of the tape of M as divided in *blocks* of $b := \lceil \log_A B \rceil$ symbols in A, where one block encodes one tape symbol of N.

To simulate one step of N, the machine reads the corresponding block and stores the symbol in B in its state. Then, based on the transition function of N, it determines which state to go to, which symbol in B to write on the block, and where to move the head. All these operations can be performed in c_B steps.

This would conclude the argument if the input were given in encoded format, giving a running time of $c_B t(n)$. However, the input is instead given as a string of n symbols in A. So we re-encode x by repeating its bits. We replace 0 with b 0s, and and 1 with b 1s (which we can take to be the encodings of 0 and 1. This is accomplished as follows. Read and erase an input bit. Move the head to the second blank to the right, write two copies of this bit, move the head back to the second non-blank symbol to the left. Repeat until the input is over. This accomplishes the task for $b = 2$. For larger b we can just repeat.

It takes $c_B n$ steps to duplicate one bit, and we repeat this n times. Overall, the number of steps is $\leq c_B n^2$ steps.

To illustrate, these are the memory snapshots starting with an input of 5 bits and after each symbol is duplicated:

$$
\begin{aligned}
&x_1x_2x_3x_4x_5\_\\
&x_2x_3x_4x_5\_x_1x_1\_\\
&x_3x_4x_5\_x_1x_1x_2x_2\_\\
&x_4x_5\_x_1x_1x_2x_2x_3x_3\_\\
&x_5\_x_1x_1x_2x_2x_3x_3x_4x_4\_\\
&\_x_1x_1x_2x_2x_3x_3x_4x_4x_5x_5\_.
\end{aligned}
$$

QED

Yet the simulation is unsatisfactory due to the need to re-encode the input which gives a quadratic time blow-up.

Open question 16.9. Is the n^2 term in Theorem 16.8 necessary?

16.1.1 The universal TM

We give a universal TM, similar to Lemma 1.18. Recall that to establish the time hiearchy Theorem 1.30 we needed a "clocked" version to ensure that we do not simulate for too many steps. Implementing such clocks is more subtle for TMs than it is for word programs, so we state and prove this clocked version. Whereas the clock in Theorem 1.30 only gives a constant factor slow down, here it will give a logarithmic slow down.

Lemma 16.10. There is a TM U that on input (M, t, x) where M is a TM, t is a natural, and x is a string:

-Stops in time $|M|^c \cdot t \cdot |t|$,
-Outputs $M(x)$ if the latter stops within t steps on input x.

Proof. To achieve this running time we extend Example 16.5 where we computed the length of the input by keeping a counter close to the head. Here, we maintain the invariant that after the simulation of one step of M, the tape of U contains

$$(x, s, M, t', y)$$

which means that M is in state s, the tape of M contains xy and the head of M is on the left-most symbol of y, and t' is the time count which is decreased at every simulated step. Moreover, the head of U will also be on the left-most symbol of y.

To start off, from input (M, t, x) we simply write the start state to the left of M, and move the head to the first symbol of x. This takes time $c|M| \cdot |t|$. After that, computing the transition of M takes time $|M|^c$. For example, we can write the transition function of M as a table, and computing this transition then amounts to finding a match in this table with the state with s and the left-most symbol of y. Decreasing the time counter and checking if it reaches 0 takes time $c|t|$, similarly to Example 16.4. To move M and t next to the tape head takes $c|M||t|$ time. **QED**

16.2 Multi-tape machines

A natural and significant extension of TMs is obtained by allowing for more tapes. Each tape has its own head. In one step, the machine can read the symbols under all the heads, and based on that and the state the machine updates the symbols and the head positions.

Definition 16.11. *A k-TM* with s states is a map

$$t : S \times A^k \to S \times A^k \times \{\text{Left}, \text{Right}, \text{Still}\}^k,$$

called the *transition* map, where S and A are as in Definition 16.1.

A *configuration* of a TM encodes the state, the contents of the k tapes, and the position of the k heads on the tape. It can be written as a tuple $(s, m_1, m_2, \ldots, m_k, h_1, h_2, \ldots, h_k)$ where $s \in S$ is the state, $m_i : \mathbb{Z} \to A$ specifies the contents of Tape i, and $h_i \in \mathbb{Z}$ indicates the position of the head on Tape i.

The yield configuration can be defined similarly to Definition 16.1.

We say that a k-TM (with $k \geq 2$) computes $y \in [2]^*$ on input $x \in [2]^*$ in *time* t (or in t steps) if, starting in configuration

$$(m_1, m_2, \ldots, m_k, 0, 0, \ldots, 0, \text{start})$$

where $x = m_1[0]m_1[1] \cdots m_1[|x| - 1]$ and m_1 is blank elsewhere, and all the other m_i are blank, it yields a sequence of t configurations where the last one is

$$(m'_1, m'_2, \ldots, m'_k, h_1, h_2, \ldots, h_k, \text{stop})$$

and the output y is written on the second tape m_2: $y = m_2[h_2]m_2[h_2+1]\cdots m_2[h_2+|y|-1]$ and m_2 is blank elsewhere. We call Tape 1 the *input tape* and Tape 2 the *output tape*.

We define k-TM-Time similarly to TM-Time (Definition 16.7). We write MTM for k-TM for some k.

At the end of the computation we do not require tapes different from m_2 to be blank; this allows us to have a read-only input tape in the Definition 16.32 of space-bounded TMs.

Exercise 16.12. Define *yield* for 2-TMs by extending the corresponding Definition 16.1 for TMs.

The convention that the input and output tape are distinct will be useful (see Definition 6.2).

Having multiple tapes allows us to solve some problems faster. Consider deciding if an input string is a palindrome (cf section §0.1.1). We will prove in Theorem 16.21 that it requires quadratic time on TMs, cf Exercise 16.6. But it can be solved in linear time on a 2-TM.

Exercise 16.13. Prove it.

However, TMs and MTMs are power-time equivalent:

Theorem 16.14. k-TM-Time$(t(n)) \subseteq$ TM-Time$(c_k t^2(n))$ for any $t(n) \geq n$ and k.

Proof. Let M be a k-TM. We give a simulation by a TM using a large alphabet B, which we then simulate by a TM using Theorem 16.8. The alphabet B is set to $A^k \times \{\hat{}, \square\}$, and thus consists of k symbols from A plus k bits. A symbol in B represents the contents of the corresponding symbols on each of the k tapes of M, and moreover it indicates which of the k heads is at that position: Symbol $\hat{}$ indicates the head is there, $\square$ indicates it is not.

For example, suppose $k = 2$. In the following matrix, the first two rows give the tape contents and head positions (indicated with the symbol $\hat{}$) of the two tapes, and the last row gives their encoding as a single tape with symbols in B:

$$\begin{array}{cccccc} \_ & 0 & \hat{1} & 0 & 1 & \_ \\ \_ & 1 & 1 & 0 & \hat{0} & 1 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ (\_,\_,\square,\square) & (0,1,\square,\square) & (1,1,\hat{},\square) & (0,0,\square,\square) & (1,0,\square,\hat{}) & (\_,1,\square,\square). \end{array}$$

To simulate 1 step of the k-TM, the TM does a one-way pass on the tape, storing in its state the symbols read under the k tapes. After that, it does one more pass on the tape and updates it based on the transition map of the k-TM. For each of the k tapes, this pass involves finding the symbol with the corresponding $\hat{}$,and updating that symbol and its neighbors. (So this pass is not one-way, as the TM may go back to a neighboring symbol.) Since the k-TM runs in time t, it does not use more than t symbols. So each pass takes time $c_k t$ for the TM, and we can simulate one step in time $c_k t$.

Hence to simulate t steps the time would be $c_k t^2$. By Theorem 16.8 we can simulate this TM over alphabet B using the fixed alphabet A in time $c_B t^2 + c_B n^2$. Because we assumed $t \geq n$, the simulation runs in time $c_B t^2$ as desired. **QED**

Working with MTMs makes it slightly easier to prove that problems are in power time on a TM.

Example 16.15. Let us show that computing the addition $x + y$ and multiplication $x \cdot y$ of two input integers x and y is in TM-Time(n^c). By Theorem 16.14 it suffices to give an MTM.

For addition, the standard column addition with carry over can be implemented. Specifically, we can first copy y on a separate tape (neither input or output). Then we can move the head of the input tape to the least significant bit of x, and another head to the least significant bit of y. Starting with a carry of 0 (stored in the state), the TMs writes the lower-order bit of the sum of these two bits of x and y and the carry in the output and stores any carry in the state. It then moves all the heads (on x, y, and on the output tape) to the left, and continues until the end of the input. (If x and y do not have the same length, the TM can fill missing positions with 0, and continue until both x and y are over.)

To compute $x \cdot y$ we can write $y = \sum_{i=0}^{|y|-1} y_i \cdot 2^i$. Hence

$$x \cdot y = \sum_{i=0}^{|y|-1} x \cdot y_i \cdot 2^i.$$

Now, each term $x \cdot y_i \cdot 2^i$ can be computed by first reading the bit y_i. If $y_i = 0$ the term is 0. If $y_i = 1$ we shift x to the left by i positions. To sum these terms, we can use the MTM for addition (on separate tapes).

Finally, we have the following fundamental result about MTMs. It shows how to reduce the number of tapes to *two*, at little cost in time. Moreover, the head movements of the simulator are restricted in a sense that at first sight appears too strong.

Theorem 16.16. k-TM-Time($t(n)$) $\subseteq$ 2-TM-Time($c_k t(n) \log t(n)$), for every function $t(n) \geq n$. Moreover, the 2-TM is *oblivious:* the movement of each tape head depends only on the length of the input.

Using this results one can prove the existence of universal MTMs similar to the universal TMs in Lemma 16.10. However, we won't need this result so we omit the proof. (See the notes for the proof.)

16.2.1 Time vs. TM-Time

We showed earlier than TMs can simulate MTMs with a power slow down. We now show that TMs can simulate word programs too, with a comparable slow-down.

Theorem 16.17. Time($t(n)$) $\subseteq$ TM-Time($t^c(n)$), for any $t(n) \geq n$.

Proof. We give a simulation by MTMs, from which one can obtain TMs by Theorem 16.14. Given a word program P, we construct an MTM which dedicates one tape to each register $R[i]$, one tape for the word size, one tape for the memory bound, and one more tape for the memory of the program (represented in a certain format explained below). The tape corresponding to register $R[0]$ is initialized to the input length n using Example 16.5.

The program and program counter (indicating which instruction is to be read next) can be stored in the transition function and state of the TM, as they have constant size. Instructions involving assignments, arithmetic operations, and conditional jumps, as well as MALLOC, can be computed in power-time. For addition and multiplication we can use Example 16.15. But in fact for this simulation something weaker suffices, as these operations are on registers on w bits, and w is logarithmic in our desired time bound. All these operations are done on registers whose bit length is the current word length. After each operation, the program counter is increased by 1, except when a conditional jump is executed, in which case it is set accordingly.

The part that is less obvious is how to handle memory. The memory M of the word program is represented on a tape as pairs (i, v) meaning $M[i] = v$. In an initial phase, the MTM inserts pairs corresponding to the input. To simulate one Write instruction $M[R[i]] := R[j]$ the MTM appends the pair $(R[i], R[j])$, where each register has a number of bits corresponding to the current word length. To simulated a Read instruction $R[i] := M[R[j]]$, the MTM goes through all these pairs in order, and for any pair of the type $(R[j], v)$ it finds, it copies v on $R[i]$. This way of doing things allows us to correctly simulate Reads and Writes interspersed with MALLOC instructions (recall the memory words of the program can hold larger and larger integers, if MALLOC instructions are executed). If no pair is found, $R[i]$ is set to 0.

Since the program runs in time $t(n)$ it can only access $t(n)$ memory locations, and so the number of pairs is $t(n)$. Moreover, as noted in section §1.7, the word length is always $\leq \log t(n)$. The time bound follows. **QED**

In the other direction, we remark that word programs can simulate TMs as well. This is less surprising, but not completely obvious as the TM handles the memory in a different way (as a tape unbounded on both sides).

Theorem 16.18. TM-Time$(t(n)) \subseteq$ Time$(t^c(n))$, for any $t(n) \geq n$.

Proof. Given a TM, we construct a word program as follows. The program contains the transition function of the TM and its state. For any input (s, a) to the transition function t of the TM we have a conditional jump taking the program to a line corresponding to the state in $t(s, a)$ and performing the corresponding write and head movement.

Memory symbols $0..n-1$ contain the input. Memory symbols $n, n+2, n+4, \ldots$ are used for the tape symbols to the right of the input, and memory symbols $n+1, n+3, n+5, \ldots$ for the symbols to the left. We dedicate a register for the position of the tape head of the TM. If the tape head moves past the symbols that we can index, we execute two MALLOC instructions (one for each side). **QED**

16.3 TMs vs circuits

By Theorem 16.18 word programs can simulate TMs, and hence by Theorem 2.12 circuits can simulate TMs. Let us give a direct proof of this simulation which also sets the stage for a stronger result proved below.

Theorem 16.19. TM-Time$(t(n)) \subseteq \bigcup_a$ CktSize$(at^2(n))$ for any $t(n) \geq n$.

For the proofs in this section it is convenient to represent a configuration of a TM in a specific way. W.l.o.g. we let the states of the machines be $[s]$, and we consider an extra symbol $\square$ indicating the absence of the head. We represent a configuration as a string of symbols over the alphabet $A \times ([s] \cup \{\square\})$. String

$$(a_1, \square)(a_2, \square) \dots (a_{i-1}, \square)(a_i, j)(a_{i+1}, \square) \dots (a_m, \square)$$

with $j \in [s]$ indicates that (1) the tape content is $a_1 a_2 \dots a_m$ with blanks on either side, (2) the machine is in state $j \in [s]$, and (3) the head of the machine is on the i tape symbol a_i in the string. Note one symbol in a string only depends on the corresponding symbol in the previous step and its two neighbors – three symbols total – because the head only moves by at most one symbol.

Proof of Theorem 16.19. Given a TM M with s states consider a $(t+1) \times (2t+1)$ matrix T, a.k.a. the *computation table*, where row i is the configuration at time i. The starting configuration (corresponding to time 0) is in the first row and has the head on the middle symbol. Note we don't need more than t symbols to the right or left because the head moves only by one symbol in one step. By locality of computation, each symbol in Row $i+1$ can be computed from 3 symbols in Row i. By Theorem 2.5, this symbol can be computed by a circuit of size c_s. Repeating this for each symbol, the entire Row $i+1$ can be computed from Row i by a circuit of size $c_s t$.

Stacking t such circuits we obtain a circuit of size $c_s t^2$ which computes the end configuration of the TM.

There remains to output the value of the function. Had we assumed that the TM writes the output in a specific symbol, we could just read it off by a circuit of size c. Without the assumption, we can have a circuit $C : A \times ([s] \cup \{\square\}) \to [2]$ which outputs 1 on (x, y) iff $x = 1$ and $y \neq \square$ (i.e., if x is a 1 that is under the TM's head). The circuit C has size c. Taking the Or of ct copies of such circuits applied to every entry in the last row of T we compute the output. This final step takes size ct by Example 2.4. **QED**

However, this simulation incurs a quadratic loss, as also noted before in Chapter 5. Interestingly, for MTMs we can give a quasi-linear simulation.

Theorem 16.20. k-TM-Time$(t(n)) \subseteq \bigcup_a$ CktSize$(a \cdot t(n) \log t(n))$, for any $t(n) \geq n$.

One can prove this from Theorem 16.16 (see Problem 16.6). However, next we give a direct proof that doesn't need Theorem 16.16.

Proof. We prove this for $k = 1$, the extension to larger k does not need new ideas and is omitted. Given a TM M, we construct a circuit S_m that on input a configuration of M with m tape symbols where the head position is within $m/4$ symbols from the center, it computes the configuration reached by M after $m/4$ steps of the computation.

We shall give an inductive construction of S_m satisfying

$$\text{Size}(S_m) \leq 2 \cdot \text{Size}(S_{m/2}) + cm$$

with base case $\text{Size}(S_c) \leq c_M$. This implies $\text{Size}(S_t) \leq c_M t \log t$. From the output of S_t we can compute the output of the TM via simple circuitry (left as exercise). This concludes the proof.

To construct S_m we think of the m symbols as divided into c blocks, and we rely on a couple of auxiliary circuits. Circuit H_m, given an m-symbol configuration, computes in which block the head is; circuit R_m given an m-symbol configuration and $i \leq c$, rotates the blocks by i positions.

To illustrate, consider a configuration of 15 symbols from tape symbol -7 to tape symbol 7, where we put $\hat{}$ on top of the symbol where the head lies, and do not show the state of the TM. Divide the configuration in 5 blocks, of size 3, numbered $-2, -1, \ldots, 2$:

$01\hat{0}$	000	001	110	1__
-2	-1	0	1	2

The value of H_{15} would be -2, as the head is on position -5 in block -2.

After rotation via R_{15} by 2 blocks we would obtain:

110	1__	$01\hat{0}$	000	001
1	2	-2	-1	0

.

Note how the head is now in position 1, much closer to the center.

We can now program S_m as follows. First run H_m and let the head be in block i. Use R_m to rotate the blocks by i positions so that the head is in a block closest to the middle. By our choice for the number of blocks, the head is guaranteed to lie within $m/8$ of the middle. Hence we can run $S_{m/2}$ to compute the configurations obtained after $m/4$ steps. Now repeat the trick. Run again H_m to get j, and then R_m to move block j closest to the middle, and then $S_{m/2}$. Finally, use R_m to restore the blocks by rotating them back by $i + j$ positions.

This circuit simulates $(m/2)/4 + (m/2)/4 = m/4$ steps, as desired. The circuits R and H can be implemented using cm gates. **QED**

16.4 The grand challenge for TMs

Tape machines pinpoint the frontier of the grand challenge. It is consistent with our knowledge that all natural computational problems (as defined in section §1.8) can be solved

- in time cn^2 on a TM, and
- in time cn on a 2-TM.

Recall that we can simulate 2-TMs by a TM with a quadratic loss by Theorem 16.14, so proving a super-quadratic lower bound for TMs is no easier than proving a super-linear lower bound for 2TMs.

In the remainder of this section we prove a quadratic lower bound for TMs, then discuss the time hierarchy for TMs.

16.4.1 Communication bottleneck: crossing sequences

Intuitively, the weakness of TMs is the bottleneck of passing information from one end of the tape to the other. We now show how to formalize this and use it show that deciding if a string is a palindrome requires *quadratic* time on TMs, which is tight (Exercise 16.6). The same bound can be shown for other functions; palindromes just happen to be convenient to obtain matching bounds.

Theorem 16.21. A TM with s states that decides if an n-bit input is a palindrome requires time $\geq cn^2/\log s$.

The main concept that allows us to formalize the information bottleneck is the following.

Definition 16.22. A *crossing sequence* of a TM M on input x and tape boundary i, abbreviated i-CS, is the sequence of states that M is transitioning to when going from symbol i to $i+1$ or vice versa during the computation on x.

The idea in the proof of Theorem 16.21 is very interesting. If M accepts inputs x and y and those two inputs have the same i-CS for some i, then we can "stitch together" the computation of M on x and y at boundary i to create a new input z that is still accepted by M. The input z is formed by picking bits from x to the left of boundary i and bits from y to the right of i:

$$z := x_0 x_2 \cdots x_i y_{i+1} y_{i+2} \cdots y_{n-1}.$$

The proof that z is still accepted is left as an exercise.

Now, for many problems including palindromes, stitched input z should *not* be accepted by M, and this gives a contradiction.

Example 16.23. We think of a step of a TM as first changing state and then moving the head. We write $u\overset{i}{\hat{v}}w$ if the tape content is uvw (with infinite blanks on each side) and the TM is in state i with the head on v, where $u, w \in A^*$ and $v \in A$, see Definition 16.1. The computation

_	Start $\hat{0}$	0	0	_
_	_	$\overset{5}{\hat{0}}$	0	_
_	_	0	$\overset{2}{\hat{0}}$	_
_	_	0	1	$\overset{2}{\hat{}}$ _
_	_	0	$\overset{3}{\hat{1}}$	_
_	_	$\overset{1}{\hat{0}}$	1	_
_	_	_	Stop $\hat{1}$	_

on palindrome 000 has the 1-cs (corresponding to double vertical line; first input symbol is at position 0) equal to $2, 1, \text{Stop}$.

The following computation on palindrome 101 has the same 2-cs as the previous example

_	Start $\hat{1}$	0	1	_
_	_	$\overset{4}{\hat{0}}$	1	_
_	_	1	$\overset{2}{\hat{1}}$	_
_	_	$\overset{1}{\hat{1}}$	1	_
_	_	_	Stop $\hat{1}$	_

.

Hence, this TM would also accept the "stitched" input

001

which is *not* a palindrome. The TM accepts 001 because it has the following "stitched" computation:

_	Start $\hat{0}$	0	1	_
_	_	$\overset{5}{\hat{0}}$	1	_
_	_	0	$\overset{2}{\hat{1}}$	_
_	_	$\overset{1}{\hat{0}}$	1	_
_	_	_	Stop $\hat{1}$	_

.

Note that the number of steps of the stitched computations needs not be the same.

To execute this plan, we are going to find two palindromes x and y that have the same i-CS for some i, but the corresponding z is not a palindrome, yet it is still accepted by M. We can find these two palindromes if M takes too little time. The basic idea is that if M runs in time t, because i-CSs for different i correspond to different steps of the computation, for every input there is a value of i such that the i-CS is short, namely has length at most $t(|x|)/n$. If $t(n)$ is much less than n^2, the length of this CS is much less than n, from which we can conclude that the number of CSs is much less than the number of inputs, and so we can find two inputs with the same CS.

Proof of Theorem 16.21. Let n be divisible by four, without loss of generality, and consider palindromes of the form

$$p(x) := x0^{n/2}x^R$$

where $x \in [2]^{n/4}$ and x^R is the reverse of x.

Assume we can find $x \neq y$ in $[2]^{n/4}$ and i in the middle part, i.e., $n/4 \leq i \leq 3n/4 - 1$, so that the i-CS of M on $p(x)$ and $p(y)$ is the same. Then we can define the stitched input

$$z := x0^{n/2}y^R$$

which is not a palindrome but is still accepted by M, concluding the proof.

There remains to prove that the assumption allows to find x and y. Suppose M runs in time t. Since crossing sequences at different boundaries correspond to different steps of the computation, for every $x \in [2]^{n/4}$ there is a value of i in the middle part such that the i-CS of M on $p(x)$ has length $\leq ct/n$. This implies that there is an i in the middle s.t. there are $\geq c2^{n/4}/n$ values x for which the i-CS of M on $p(x)$ has length $\leq ct/n$.

For fixed i, the number of i-CS of length $\leq \ell$ is $\leq (s+1)^\ell$.

Hence there are $x \neq y$ for which $p(x)$ and $p(y)$ have the same i-CS whenever $c2^{n/4}/n \geq (s+1)^{ct/n}$. Taking logs one gets $cn \geq ct\log(s)/n$ as claimed. **QED**

Exercise 16.24. For every s and n describe an s-state TM deciding palindromes in time $cn^2/\log s$ (matching Theorem 16.21).

Exercise 16.25. Let $L := \{xx : x \in [2]^*\}$. Show $L \in$ TM-Time(cn^2), and prove this is tight up to constants.

The crossing-sequence argument essentially shows that TMs have efficient randomized communication protocols (cfChapter 13). This is formalized in the following.

Theorem 16.26. For a function $f : [2]^n \times [2]^n \to [2]$ consider the padded function $p_f : [2]^{3n} \to [2]$ defined as $p_f(x0^n y) = f(x, y)$. If p_f is computable by an s-state TM in time t then f has randomized protocols with communication $c(\log s)t/n$ and error $\leq 1/2$.

Proof. For $i \in [n]$, define the protocol P_i as follows: A is in charge of the first $n+i$ symbols (which include x); B is in charge of last $n+(n-i)$ symbols (which include y). They simulate the TM in turn, communicating $\log s + c$ bits whenever the TM crosses the boundary of the

$(n+i)$-th symbol. These bits represent the state of the machine or a special symbol denoting that the computation is over with final state s, from which the value of the function can be determined. The parties carry this simulation for up to $(t/n)/(c \log s)$ crossings. If the TM hasn't stopped they stop and output, say, 0.

The distribution on protocols is P_I where I is uniform in $[n]$. As in the proof of Theorem 16.21, for every x and y the probability over I that the protocol cannot complete the simulation of the TM on input $x0^n y$ is $\leq 1/2$. **QED**

Using Theorem 16.26 we can obtain stronger quadratic impossibility results for TMs, ruling out even randomized TMs and establishing correlation bounds, as a consequence of suitable communication lower bounds (such as the correlation bound for inner product, Theorem 13.15). By contrast, palindromes can be solved in quasi-linear time by randomized TMs, see Problem 16.8 and section 16.7.1 for a definition of randomized TMs.

16.5 TM-Time hierarchy

Using the universal TM (Lemma 16.10) and reasoning as in the time hierarchy Theorem 1.30 one can prove that

$$\text{TM-Time}(t(n) \log t(n)) \subsetneq \text{TM-Time}(o(t(n))$$

for all functions t which are "TM-time constructible," a notion similar to time-constructibility (Definition 1.27) but for TMs. Again, such functions t include most functions of interest. We do not state this time hierarchy result precisely. Instead, let us not that there is a gap of about $\log t$ in the bounds, compared to the constant gap in Definition 1.27. This gap arises from the corresponding loss in the simulation Lemma 16.10.

Is there a tighter hiearchy result for TM? For time bounds like $t = n^2$ this is unknown. However, surprisingly, all running times below $n \log n$ compute the same *total* functions. (The result is not true for partial functions, see Problem 16.9.)

Theorem 16.27. $\text{TM-TotalTime}(t(n)) = \text{TM-TotalTime}(n+1)$ for any $t(n) = o(n \log n)$.

Note that time $n+1$ is barely enough to scan the input. Indeed, the corresponding machines in Theorem 16.27 will only move the head in one direction. The "+1" only reflects that we charge one time step to write the output and stop in Definition 16.1. Moreover, such machines have no use of writing to the tape (except to write down the output). These constrained machines are well-studied and are known as *finite-state-automata*. The functions they compute are called *regular* .

Definition 16.28. A function $f : [2]^* \to [2]$ is *regular* if it can be computed in time $n+1$ by a TM that does not move the head left.

The proof of Theorem 16.27 is the combination of the next two lemmas.

Lemma 16.29. Let M be an s-state TM running in time $t(n) := (n \log n)/(cs)$. Then on every input $x \in [2]^*$ every i-CS with $i \leq |x|$ has length $\leq c_s$.

Proof. We proceed by contradiction. Let x be a shortest input with a crossing sequence of length $> \ell$, for a parameter ℓ to be set, and let $m := |x|$. By picking ℓ sufficiently larger than s, we can assume that $m \geq |M| + c$ and so the machine runs in time $\leq t(m)$ on x. (This is because there are only a finite number of inputs of length $\leq |M| + c$. Each of these inputs has crossing sequences of constant length since we can assume that the machine stops on every input, possibly by adding $\leq n$ to the time, which does not change the assumption on $t(n)$, as discussed in 1.10.)

There are $m + 1 \geq m$ tape boundaries within or bordering x. If we pick a boundary uniformly at random, the average length of a CS on x is $\leq t(m)/m$. By Fact A.13 there are $\geq m/2$ choices for i s.t. the i-CS on x has length $\leq 2t(m)/m$.

The number of such crossing sequences is

$$\leq (s+1)^{2t(m)/m} = (s+1)^{m \log(m)/(mcs)} \leq m^{0.1}.$$

Hence, the same crossing sequence occurs at $\geq (m/2)/m^{0.1} \geq 4$ indices i, using that $m \geq c$.

Let j be the boundary of the crossing sequence of length $> \ell$ from the definition of x. Of the four indices i, at least two are different from j and on the same side of j. We can remove the corresponding interval of the input without removing the CS of length $> \ell$. Hence we obtain a shorter input with a CS of length $> \ell$, contradiction. **QED**

This theorem already implies that time $o(n \log n)$ equals linear time. But as we mentioned we can prove the stronger result that the time is $n + 1$.

Lemma 16.30. Suppose $f : [2]^* \to [2]$ is computable by a TM such that on every input x, every i-CS with $i \leq |x|$ has length $\leq b$ for a constant b. Then f is regular.

Proof. The states of the desired TM are subsets of crossing sequences of the given TM. We maintain the invariant that after reading the first i bits $x_{\leq i}$ of the input x the state of the desired TM is the set of i-CSs consistent with the first i bits of the input, that is, the set of i-CSs that can arise in the computation of some input $x_{\leq i}y$. At the end, only one CS is left. We can assume w.l.o.g. that this CS will encode whether the computation is accepting (e.g. the TM can enter an accept state and move the head to the rightmost symbol, this does not change the assumption).

We leave it as an exercise to verify that the invariant can be maintained. **QED**

16.6 Randomness

As in Chapter 1 and Chapter 6 we augment the model with randomness. A *randomized* TMs has two transition functions σ_0 and σ_1, where each is as in Definition 16.1. At each step, the TM uses σ_0 or σ_1 with probability $1/2$ each, corresponding to tossing a coin. We can define the corresponding class of functions TM-BPTime(t). Whereas for BPTime(t) we do not know of a deterministic simulation that runs faster than time 2^t, for TM-BPTime we have a simulation that runs in time about $2^{\sqrt{t}}$.

Theorem 16.31. TM-BPTime$(t) \subseteq$ Time$(2^{\sqrt{t}\log^c t})$, for any $t = t(n) \geq n$.

See the notes for the proof.

16.7 Sub-logarithmic space

TMs are convenient for defining space. Indeed, for space bounds $\geq \log n$, space complexity on TMs and on word programs is the same, up to constant factors. More importantly, TMs allow us to investigate sub-logarithmic space bounds that are hard to make sense of with word programs. We define a *space-bounded TM* as a 2-TM whose input tape is read-only. Recall this is consistent with Definition 16.11 of 2-TM which stipulates that the output is written on another tape.

Definition 16.32. A function $f : X \subseteq [2]^* \to [2]$ is computable in TM-Space$(s(n))$ if there is a 2-TM M which on input x on the first tape, where $x \in X$, $|x| \geq |M|$ computes $f(x)$ and M never writes on the first tape; and moreover the heads are limited as follows:

(1) the head on the second tape is always in $[s]$,

(2) the head on the first (input) tape is always in $[-1..n]$ on inputs of length n (corresponding to the input bits plus a blank symbol on each side as delimiter).

The restriction to total functions $f : [2]^* \to [2]$ is denoted by TM-TotalSpace$(s(n))$.

Condition (2) makes sense since the tape is read-only; it also allows us to bound configurations precisely.

Let $f \in$ TM-Space$(s(n))$ and M be the corresponding machine from Definition 16.32. A *configuration* of M contains c_M bits for the state, cs bits for the contents of the 2nd tape and the position of its head, and $\log n + c$ bits for the position of the head on the first tape. The latter relies on the convention in Definition 16.32 that the input head does not wander more than one blank symbol past the input. In total, configurations take $cs + \log n + c_M$ bits. In particular, their number is

$$\leq c_M \cdot c^s \cdot n. \tag{16.1}$$

First we prove a result analogue to Theorem 16.27 that space $\leq c \log\log n$ equals regular. Then we prove two results showcasing the surprising power of small-space algorithms.

Theorem 16.33. TM-TotalSpace$(c \log\log n)$ = Regular.

Proof. Let $f : [2]^* \to [2]^*$ be in TM-TotalSpace$(c \log\log n)$ and let M be a corresponding machine as in Definition 16.32. By equation (16.1) the number of configurations of M is

$$\leq c_M \cdot c^{c \log\log n} \cdot n \leq c_M \cdot 2^{0.5 \log\log n} \cdot n = c_M \cdot n \cdot \sqrt{\log n}.$$

Reasoning as in Theorem 6.4, the machine M runs in time $\leq c_M \cdot n \cdot \sqrt{\log n}$.

The result now follows from Theorem 16.27. **QED**

16.7.1 The power of sub-logarithmic space

Now is a good time to pause and ask yourself if you think TM-TotalSpace($o(\log n)$) = Regular as well. After all, it is hard to imagine what you can do in space $o(\log n)$, since one can't even compute the length of the input. In fact, we have the following surprising result showing that increasing space $c \log\log n$ by a constant factor more gives more power.

Theorem 16.34. TM-TotalSpace($c \log\log n$) $\neq$ Regular.

Proof. Consider the problem of deciding if a given input string is of the form

$$0\#1\#10\#11\#100\#101\#...$$

where # is a separator, and adjacent numbers are consecutive (we can encode this in binary as discussed in section §1.2).

This problem is not regular, see Problem 16.7.

To solved the problem in space $c \log\log n$ start by checking if the first two symbols are 0#. Then check that for every pair of consecutive numbers $x\#y$ in the sequence, $y = x + 1$. The critical point is that this check can be done in space $c \log |x|$, regardless of the length of y, and if the check passes then the length of y is obviously bounded as well, so the numbers stay $\leq \log n$. **QED**

Finally, consider computing Majority. It is not regular (see Problem 16.7). In particular by Theorem 16.33 it cannot be solved in constant space. Yet, surprisingly, it can be solved in constant space using randomness (see section §16.6 for the definition of randomized TM). The corresponding time bound will be exponential.

Theorem 16.35. Majority can be solved in constant space by a randomized TM.

Proof. On input $x \in [2]^n$, the algorithm performs many one-way scans of the input. In each scan the machine checks if $x = N_{3/4}$ and if $x = N_{1/4}$. Here, N_β is a string of n i.i.d. bits with prob. p of being 1. Let E_β be the event that $x = N_\beta$ and p_β be $\mathbb{P}[E_\beta]$, all depending on x.

The algorithm keeps performing scans until exactly one of $E_{3/4}$ or $E_{1/4}$ happens, in which case it outputs 1 if it was $E_{3/4}$ and 0 otherwise.

Let us analyze this algorithm. The prob. of outputting 1 is

$$\frac{p_{3/4}(1-p_{1/4})}{p_{3/4}(1-p_{1/4}) + p_{1/4}(1-p_{3/4})}.$$

We claim that if x has weight more than $n/2$ this probability is $> 2/3$. This claim is equivalent to

$$p_{3/4}(1-p_{1/4}) \geq 2p_{1/4}(1-p_{3/4})$$

which is implied by

$$0.99p_{3/4} \geq 2p_{1/4}.$$

In turn, this is true because every pair of 1 and 0 in x gives the same factor $3/4 \cdot 1/4$ in $p_{3/4}$ and $p_{1/4}$. Since x has more zeroes than ones, we are left with at least one factor $3/4$ in $p_{3/4}$ with a corresponding factor $1/4$ in $p_{1/4}$, and

$$0.99 \cdot \frac{3}{4} \geq 2 \cdot \frac{1}{4}.$$

A similar analysis holds if x has weight less than $n/2$.

This concludes the proof – unless n is even and the weight of x is exactly $n/2$. This case can be reduced to the case of odd n, depending on how majority is defined. **QED**

16.8 Problems

Problem 16.1. Without using the generic simulation Theorem 16.17, describe a TM that computes in power time the division of two input naturals x and y, defined as the largest z s.t. $yz \leq x$.

Problem 16.2. Prove that the following problems are in TM-Time($an \log n$) for a constant a:

- Indexing (defined in Exercise 2.6).
- Majority.

Problem 16.3. Show Time(t) is in non-deterministic time $t \log^c t$ on a multi-tape machine. Hint: Follow the proof structure in section §5.3. Use the MergeSort algorithm.

This result can be used to obtain an alternative proof of the results in section §5.3.

Problem 16.4. Prove that 3Sat is not TM-Time($n^{1.99}$).

Guideline: Consider a variant of the palindromes problem where the input bits are suitably spaced out with zeroes. Prove a time lower bound for this variant by explaining what modifications are needed to the proof of Theorem 16.21. Conclude by giving a suitable reduction.

Problem 16.5. A 1.5-TM is like a 2-TM except that the input tape is read-only. Prove that 3Sat requires time n^{1+c} on a 1.5-TM.

Guideline: Combine the ideas in Theorem 6.53 and Theorem 16.21.

1. Let M be a 1.5-TM running in time $t(n)$. Divide the read-write tape of M into consecutive blocks of b symbols, shifted by an offset $i < b$. (So the first symbols of the blocks are $i, i+b, i+2b, \ldots$.) Prove that for every input $x \in [2]^n$ there is i such that the sum of the lengths of the crossing sequences between any adjacent blocks of the computation M on x is at most $t(n)/b$. Here a crossing sequence also encodes the position of the head on the input tape, and the time at which each crossing occurs.

2. Prove that 1.5-TM-Time($n^{1.1}$) $\subseteq \exists y \in [2]^{n^{1-c}} \text{TiSp}(n^c, n^{1-c})$. (Define the right-hand side.)

3. Conclude the proof.

Problem 16.6. Give an alternative proof of the simulation of MTMs by circuits, Theorem 16.20, using the oblivious simulation in Theorem 16.16.

Problem 16.7. Prove that the following problems are not regular:
- Majority.
- Deciding if the input is in the form 0#1#10#11#100#101#... (cf Theorem 16.34).

Problem 16.8. Solve palindromes in time $n \log^c n$ on a randomized TM (note the TM has only one tape, and see section §16.6 for the definition of randomized TMs). Hint: View the input as a polynomial, use Fact A.50.

Problem 16.9. Prove that Theorem 16.27 is false for partial functions $f : X \subseteq [2]^* \to [2]$.

16.9 Notes

These notes augment the notes to Chapter 1.

The oblivious simulation of MTMs by 2TMs, Theorem 16.16, is from [177, 284]. The simulation of MTMs by circuits, Theorem 16.20, is towards the end of [284].

The proof of Problem 16.5 but for computing a function in $\Sigma_1\text{Time}(cn)$ is from [240]. The proof was apparently rediscovered in [356] and extended to 3Sat.

For multi-tape machines, a separation between deterministic and non-deterministic linear time is in [282, 300].

Crossing sequences and the quadratic bound for palindromes are from [174].

The time hierarchy for tape machines originates in [176] and was later optimized in [177].

The collapse of time bounds $o(n \log n)$, Theorem 16.27, follows by combining results in [175, 216].

For the deterministic simulation of TM-BPTime, Theorem 16.31, see [375]. The proof there assumes that the time bound is known, but in general one can try all time bounds, as in, say, the proof of Theorem 6.45.

The separation between space $c \log \log n$ and regular, Theorem 16.34, is from [324].

The constant-space randomized TMs for majority, Theorem 16.35, are from [120]. The result, "a bit unexpected," is proved there for strings 0^n1^n, but a similar proof applies to majority.

The quasi-linear time randomized TMs for palindromes, Problem 16.8, are from [119].

Chapter 17

Barriers

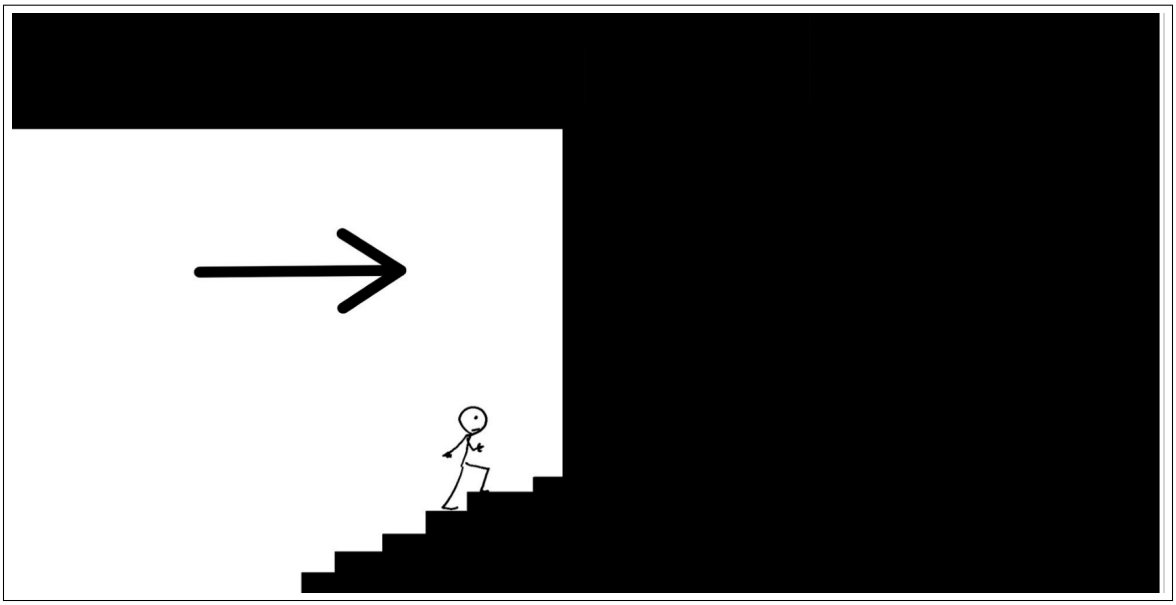


Figure 17.1: Barriers appear to block progress on basic questions.

In an attempt to understand the Grand Challenge (section §1.8), one can identify several proof techniques and show or speculate that they cannot solve it. Such arguments are known as "barriers." Barriers may appear to block natural research directions, see figure 17.1. Several barriers have been put forth. The two main ones are the *black-box* (a.k.a. *oracle* or *relativization* barrier) and the *natural proofs* barrier, discussed next. Amusingly, there are even barriers to barriers, i.e., arguments indicating that proving a barrier is difficult, making complexity theory a rather philosophical and introspective field.

17.1 Black-box

As hinted, many of the results we have shown don't really exploit the specifics of the model we are working with, but work in greater generality. To make this precise we augment our model with access to an auxiliary function $f : [2]^* \to [2]$ called *oracle*. If a proof also applies when the model is augmented with *any* oracle we say it is black-box, or that it *relativizes*. The black-box barrier helps us understand the limits of basic simulation arguments, including diagonalization, which tend to relativize.

More precisely, we augment word programs with an instruction

- $R[i] :=$ Query

which sets $R[i]$ to the output of the oracle f on the input stored in memory words $R[0]$ to $R[1]$, one bit per memory word. For an oracle f, we define the corresponding complexity classes, denoted P^f, Pspace^f, and so on.

As a first example, consider the separation $\mathrm{P} \neq \mathrm{Exp}$ (1.31). This separation relativizes:

Theorem 17.1. For every oracle f, $\mathrm{P}^f \neq \mathrm{Exp}^f$.

Proof. The result that $\mathrm{P} \neq \mathrm{Exp}$ is a consequence of the Time Hierarchy Theorem 1.30, which in turn is based on the simulation Lemma 1.18. This simulation works the same for oracle machines: When the simulated machine makes an oracle query, the simulating machine performs the same query. **QED**

Next we argue that relativizing techniques cannot resolve other major questions. Perhaps the simplest example is for the "P = Pspace?" question. We show that there is an oracle w.r.t. which the answer is yes, and another one for which it is no. This means the question cannot be resolved via a black-box proof.

Theorem 17.2. There are oracles f, g s.t.:
$\mathrm{P}^f = \mathrm{PSpace}^f$, and
$\mathrm{P}^g \neq \mathrm{PSpace}^g$.

Proof. The oracle f can be any PSpace-complete function, such as QBF (see Theorem 6.32). First we claim that $\mathrm{PSpace}^f = \mathrm{PSpace}$. This is just because the oracle queries can be answered in PSpace since QBF is in PSpace. Second, we claim that $\mathrm{PSpace} \subseteq \mathrm{P}^f$. This is because any function in PSpace map-reduces to QBF in FP, and then we can query the oracle. Combining these two claims we obtain $\mathrm{PSpace}^f \subseteq \mathrm{P}^f$, and the reverse inclusion is as simple as it is without oracles (Theorem 6.4).

The construction of g is more involved. To illustrate the main idea, let us first assume that oracle programs, on inputs of length n, only query the oracle at inputs of length n as well. Then we can define the oracle g as follows. Let $M_1, M_2, \ldots$ be an enumeration of all oracle programs. On an input of length n, run M_n for $2^n - 1$ steps on input 1^n, returning zero for all oracle queries. If M outputs 1 then set g to be zero on all $[2]^n$. Otherwise, set

g to be zero on all $[2]^n$ except for one input y that M didn't query, where $g(y) = 1$. This concludes the definition of the oracle.

Now consider the problem H^g of determining, on input 1^n, if there exists $y \in [2]^n : g(y) = 1$. This problem is in PSpace^g, by going through all y. But by construction it isn't in P^g. To show the latter, assume there is a s.t. M_a solves the problem in time n^a. Pick b large enough so that M_b is equivalent to M_a and $b^a < 2^b$, and consider $M_b(1^b) = M_a(1^b)$. By construction, g returns 0 on all oracle queries, and queries $< 2^b$ of the inputs of length b. By construction, the machine returns 0 iff there is $y \in [2]^b : g(y) = 1$. Contradiction. This concludes the proof under the assumption that on inputs of length n only oracle queries of length n are made.

We now explain how to drop the assumption. The idea is to pick sufficiently spaced-out inputs so that the above strategy can be executed. We construct the oracle g in stages, considering one oracle program M_i at a time (whereas previously one could have processed all oracle programs simultaneously). At the end of a stage, the oracle is defined only for a constant number of input lengths. For each oracle program M_i we set the values of the oracle at an additional constant number of input lengths, in a way that M_i makes a mistake. More precisely, stage i starts with an oracle g s.t. H^g cannot be solved by any program M_j running in time n^j for $j < i$, and only the first c_i input lengths of the oracle are set. Again, our goal is to extend the oracle so that H^g cannot be solved even by M_i running in time n^i. To do this, pick the first input length $m > c_i$ s.t.

1. $M_i = M_m$,
2. $m^i < 2^m$, and
3. $g(y)$ was not set for any $y \in [2]^m$.

We define g on input length m as before. That is, run $M_i = M_m$ on input 1^m for $2^m - 1$ steps, answering all oracle queries of length m or bigger with zero, and all oracle queries of length $< m$ following the definition of g (which we can assume to be set on all lengths $< m$, and note may involve both 0 and 1 outputs). Then we define g as before: If M_m outputs 1 we set g to be zero on all inputs of length m, otherwise we set it to 1 on one of the queries of length m that wasn't made by M_m on input 1^m. **QED**

Exercise 17.3. Show that the question "P = NP?" cannot be resolved via black-box techniques. (To define NP^f, augment the program P in Definition 5.3 with an oracle.)

17.2 Natural proofs

The natural proofs barrier is more tailored to understanding impossibility results against non-uniform models of computation that do not rely on diagonalization. The insight is twofold:

(1) Many proof techniques to establish an impossibility result for computing a function h (e.g., $h =$ parity) by a class of functions F (e.g., $g = \mathrm{AC}^0$) also yield an *efficient* algorithm that distinguished truth tables of F from uniform:

Definition 17.4. A *natural proof* against a set of functions F is a function $t \in \mathrm{P}$ such that for all large enough n:

For any $f : [2]^n \to [2]$ in F, $t(f) = 1$, where the input to t is the truth table of f of length 2^n.

$\mathbb{P}[t(U) = 1] \leq 1/2$, for a uniform bit string U of length 2^n.

Note that for a natural proof to establish an impossibility result for a specific function h one would also need $t(g) = 0$. But we don't need this condition now.

(2) For many classes F for which we would like to prove impossibility results (e.g., $F = \mathrm{TC}^0$) natural proofs do not exist *under popular conjectures*. In other words, these classes are believed to be powerful enough to compute pseudorandom generators (PRGs, see Definition 11.3) with output length 2^n that fool all efficient distinguishers (the seed of the generators is hard-wired in the class F, so is typically of length n^c). We stress that in most cases the construction of such PRGs is not known unconditionally. This is where complexity theory gets quite philosophical. We can't really *prove* the existence of such PRGs without solving a version of the Grand Challenge, in which case these barriers are not actual barriers.

17.2.1 Examples of proofs that are natural

In this section we illustrate via examples the fact that many impossibility results yield natural proofs. For our first example we consider impossibility results for AC^0 circuits established via the restrict-and-simplify method (Chapter 9).

Theorem 17.5. There are natural proofs against functions on n bits computable by alternating circuits of depth d and size $2^{n^{c_d}}$.

Proof. The distinguisher algorithm, given the truth table of length 2^n of a function f, checks if there is a restriction with n^{c_d} stars that makes f constant. This distinguisher is efficient because the number of restrictions, with any number of stars, is $\leq 3^n$; and for each restriction we can check if the restricted function is constant in power time in 2^n.

If f is computable by an alternating circuit $C : [2]^n \to [2]$ of depth d size $2^{n^{c_d}}$ such a restriction exists by Corollary 9.26.

On the other hand, consider a uniform function $U : [2]^n \to [2]$. The restriction U_ρ to a restriction ρ with n^{c_d} stars is a function on n^{c_d} bits. The prob. that this function is constant is

$$\leq 2 \cdot 2^{-2^{n^{c_d}}}.$$

By a union bound, the prob. that there is a restriction ρ with n^{c_d} stars s.t. U_ρ is constant is

$$\leq 3^n \cdot 2 \cdot 2^{-2^{n^{c_d}}} \leq 0.5,$$

using again that the number of restrictions with any number of stars is $\leq 3^n$. **QED**

As a second example we consider TMs (Chapter 16).

Theorem 17.6. There are natural proofs against functions on n bits computable by TMs with s states running in time $\leq cn^2/\log s$.

Proof. The distinguisher algorithm, given the truth table of length 2^n of a function f, considers the function $f_0 : [2]^{n/3} \times [2]^{n/3} \to [2]$ defined as $f_0(x, y) := f(x0^{n/3}y)$ and checks if $R(f_0) \geq 2^{-0.1n}$ where R is the quantity defined in section 13.2.1 for $k = 2$. This distinguisher is efficient because we can compute R in time 2^{cn} by inspection of its definition.

If $f : [2]^n \to [2]$ is computable by an s-state TM running in time $cn^2/\log s$ then by Theorem 16.26 f_0 has 2-party protocols with communication cn and error $\leq 1/2$, and by Lemma 13.16 we have

$$\frac{1}{2} \leq 2^{cn} \cdot R(f_0)^{1/4}$$

and so

$$R(f_0) \geq \frac{1}{2^{0.1n}}.$$

On the other hand, consider a random function $U : [2]^n \to [2]$. By Exercise 13.18 we have

$$\mathbb{E}_U[R(U)] \leq \frac{c}{2^{n/3}}$$

and that R is never negative. So by Fact A.13 we have

$$\mathbb{P}_U[R(U) \geq 2^{-0.1n}] \leq c\frac{2^{0.1n}}{2^{n/3}} \leq 1/2.$$

QED

Along the same lines one has natural proofs against multi-party protocols. We leave the formulation as an exercise.

17.2.2 Ruling out natural proofs under popular conjectures

Under various assumptions we can construct pseudorandom generators that rule out natural proofs for classes of functions at the frontier of the grand challenge. In this section we discuss several examples. First, from a PRG with 1 bit of stretch that fools even circuits of size 2^{n^ϵ} we can rule out natural proofs against CktP.

Theorem 17.7. Suppose there is a PRG G with seed length $n - 1$ with error $1/2^{n^\epsilon}$ against circuits of size 2^{n^ϵ}. Then there is no natural proof against circuits of size $n^{c_{G,\epsilon}}$.

As remarked in section 11.5.2 such PRGs can be constructed from one-way functions (with suitable parameters) which in turn can be based on popular conjectures such as the hardness of factoring (cf section §7.7).

Proof. Assume towards a contradiction there is a natural proof, and let the distinguisher t in Definition 17.4 run in time 2^{an} on input a truth table of length 2^n. Consider G on inputs of length $n^{2/\epsilon}$. Using the constructions in Theorem 11.58 and Theorem 11.60 we can construct a PRG stretching $n^{2/\epsilon}$ bits into 2^n bits that fools circuits of size $2^{n^2} \geq 2^{an}$. As remarked in section 11.5.2, each output bit of the PRG is computable by circuits of size $n^{c_G/\epsilon}$. By Definition 17.4 of natural proof t accepts a uniform string U of length 2^n with prob. $\leq 1/2$. Since the PRG fools t, there is a seed s.t. t also does not accept the corresponding output. We can hardwire this seed in the circuit to obtain a truth-table computable by a circuit of size $n^{c/\epsilon}$ that is not accepted by t, contradicting Definition 17.4. **QED**

For more constrained circuit classes like TC^0 we can have a similar result based on more specific assumptions, for example relying on the hardness of factoring, see the notes. Thus, natural proofs appear to explain why lower bounds are available for AC^0 but not TC^0. The essential feature of TC^0 that enables these results is the ability to perform iterated multiplication Theorem 8.7.

Next we discuss our final example which addresses our inability to prove super-quadratic lower bounds against TMs. By the simulations in Theorem 16.19 and Theorem 16.20 this example also applies to power-size circuits thus providing an alternative assumption under which there are no natural proofs against circuits of size n^a for a constant a (cf Theorem 17.7). A variant of this candidate can also be implemented in TC^0 (see the notes).

As in the proof of Theorem 17.7, we give a candidate PRG stretching n^c bits into 2^n such that given $i \in [2]^n$ output bit i of the PRG computable in quadratic time on a TM with a quadratic number of states. The candidate is an asymptotic generalization of a well-documented and widely used cryptographic "*block cipher*": the *Advanced Encryption Standard*, *AES*. AES is based on the substitution-permutation network (SPN) structure, and will actually compute a function from $[2]^n \to [2]^n$ (whereas range [2] would suffice for our goals). On input $x \in [2]^n$, an SPN is computed over a number r of rounds, where each round "confuses" the input by dividing it into m/b bundles of b bits and applying a substitution function (S-box) to each bundle, and then "diffuses" the bundles by applying a matrix M with certain "branching" properties. At the end of each round i, the n bits are xor-ed with an n-bit round seed k_i, refer to figure 17.2.

The candidate follows the design considerations behind the AES block cipher, and particularly its S-box. For any n that is a multiple of 32, we break the input into $m := n/8$ bundles of $b = 8$ bits each, viewed as elements in the field $\mathbb{F}_{2^8}$, and perform $r = n$ rounds. We use the S-box $S(x) := x^{2^b-2}$. M is computed in two (linear) steps. In the first step, a permutation $\pi : [m] \to [m]$ is used to shuffle the b-bit bundles of the state; namely, bundle i moves to position $\pi(i)$. The permutation π is computed as follows. The m bundles are arranged into a $4 \times m/4$ matrix. Then row i of the matrix ($0 \leq i < 4$) is shifted circularly to the left by i places. In the second step, a specific matrix (with so-called maximal branch number) is applied to each column of 4 bundles.

Let us now illustrate how one round can be computed in time cn with cn states. The bundles are written on the tape in column-major order: First the 4 bundles of the 1st column, then the 4 bundles of the 2nd column, and so on. The cn instances of S and φ can

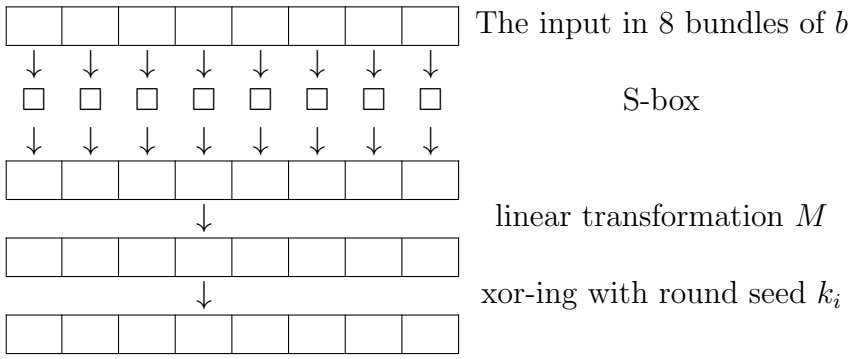


Figure 17.2: One round of an SPN.

be computed in time cn. To see that π can also be computed in time cn, note that due to the representation, we can compute π with one pass, using that all but c bundles need to move $\leq c$ positions away. Finally, encoding the n-bit seed in the TM's state transitions, the addition of each round key also takes time cn.

Therefore, the $r = n$ rounds can be computed in time cn^2 with cn^2 states.

17.3 Problems

Problem 17.1. An *oracle circuit* has special Query$(t_1, t_2, \ldots, t_s)$ instructions where the t_i are as in Definition 2.1 which return the value of the oracle on $(t_1, t_2, \ldots, t_s)$. Such a gate counts s towards the size of the circuit. Prove that:

(1) For every oracle f, for every k, $\text{Exp}^f \not\subseteq \text{CktSize}^f(n^k)$, i.e., Theorem 2.13 relativizes.

(2) There is an oracle g and $k \in \mathbb{N}$ s.t. $\text{E}^g \subseteq \text{CktSize}^g(n^k)$.

Problem 17.2. For $\epsilon > 0$ let F be the class of functions f s.t. f_n is computable by depth-2 TC^0 circuits of size $2^{\epsilon n}$ with weights of $\leq \epsilon n$ bits. Give a natural proof against F, for some constant ϵ. Hint: Impossibility results for such circuits are sketched in section §8.4.

17.4 Notes

Relativization originated in the seminal work [43] and led to a myriad of oracle constructions. Often, these constructions are related to, and have provided some motivation for the study of, basic complexity classes. Indeed, a major motivation for the impossibility results for AC^0 in section §9.3 was showing that the PH does not collapse w.r.t. some oracles [389]. After this first prolific phase of oracle constructions, starting about half a century ago, a second phase has followed during which the emergence of non-relativizing techniques, such as the arithmetization technique used in the proof that IP = PSpace (Theorem 10.6), cast doubt on the significance of "oracle separations," a term that carried a mildly negative or dismissive connotation. However, a variant of the black-box barrier where the oracles are augmented with additional algebraic structure is sharper and captures many such techniques as well,

see [4]. In a more recent third phase, oracle results have made a comeback in cryptography and quantum computing, often under the terminological disguise of *black-box*.

Natural proofs are from [290] where Theorem 17.7 is proved. The pseudorandom function in TC^0 is from [262]. AES is described in [98]. The SPN structure of alternating "confusion" and "diffusion" steps was put forth already in [311]. The candidate pseudorandom function computable in quadratic time on a TM is from [250]. The work [20] considers threshold circuits of size $n^{1+\epsilon}$ and notes that they evade the natural proof barrier from [290, 262]. A candidate pseudorandom function computable by such circuits, also based on the SPN structure, is presented in [250].

Chapter 18

Speculation

> *[...] Now it seems to me, however, to be completely within the realm of possibility that $\phi(n)$ grows that slowly. Since it seems that $\phi(n) \geq k \cdot n$ is the only estimation which one can obtain by a generalization of the proof of the undecidability of the Entscheidungsproblem and after all $\phi(n) \sim k \cdot n$ (or $\sim k \cdot n^2$) only means that the number of steps as opposed to trial and error can be reduced from N to $\log N$ (or $(\log N)^2$). However, such strong reductions appear in other finite problems [...]." [129]*

The only things that matter in a theoretical study are those that you can prove, but it's always fun to speculate. After worrying about P vs. NP for half my life, and having carefully reviewed the available "evidence" I have decided I do not believe that P $\neq$ NP. *Tertium non datur*, so I should believe that P = NP.

A main justification for my belief is history:

1. In the 1950's Kolmogorov conjectured that multiplication of n-bit integers requires time $\geq cn^2$. That's the time it takes to multiply using the method that mankind has used for at least six millennia. Presumably, if a better method existed it would have been found already. Kolmogorov subsequently started a seminar where he presented again this conjecture. Within one week of the start of the seminar, Karatsuba discovered his famous algorithm running in time $cn^{\log_2 3} \approx n^{1.58}$. He told Kolmogorov about it, who became agitated and terminated the seminar. Karatsuba's algorithm unleashed a new age of fast algorithms, including the next one. I recommend Karatsuba's own account [204] of this compelling story.

2. In 1968 Strassen started working on proving that the standard cn^3 algorithm for multiplying two $n \times n$ matrices is optimal. Next year his landmark $cn^{\log_2 7} \approx n^{2.81}$ algorithm appeared in his paper "Gaussian elimination is not optimal" [327].

3. In the 1970s Valiant showed that the graphs of circuits computing certain linear transformations must be a *super-concentrator*, a graph which certain strong connectivity properties. He apparently conjectured that super-concentrators have a super-linear

number of wires, from which super-linear circuit lower bounds would then follow. However, he later disproved the conjectured [348, 351]: using results of Pinsker [283] on expander graphs (Theorem 12.6) and related objects he constructed super-concentrators using a linear number of edges.

4. At the same time Valiant also defined *rigid* matrices and showed that an explicit construction of such matrices yields new circuit lower bounds. A specific matrix that was conjectured to be sufficiently rigid is the Hadamard matrix. Alman and Williams showed that, in fact, the Hadamard matrix is not rigid [21].

5. Constructing rigid matrices is one of *three* ways to get circuit lower bounds from a graph decomposition in [351]. Another way is via communication lower bounds. Here a specific candidate was the *sum-index* function, but then Sun [333] gave an efficient protocol for sum-index.

6. The LBA problems (Is L = NL? Is NL closed under complement?) were stated in the 60's. A solution to the second problem was found after more than 20 years [185, 334, 335]. The general belief was that NL is not closed under complement, just like today the general belief seems to be that NP is not. But in fact, as we saw in Theorem 6.47, NL *is* closed under complement! Note a negative solution to the second problem would have implied a negative solution to the first, which remains open.

7. After finite automata, a natural step in lower bounds was to study sightly more general programs with constant memory. Consider a program that only maintains c bits of memory, and reads the input bits in a fixed order, where bits may be read several times. It seems quite obvious that such a program could not compute the majority function in polynomial time (see Chapter 0). This was explicitly conjectured by several people, including [66]. Barrington [255] famously disproved the conjecture by showing that in fact those seemingly very restricted constant-memory programs are in fact equivalent to log-depth circuits, which can compute majority (and many other things) (see Theorem 7.21).

8. Mansour, Nisan, and Tiwari conjectured [241] in 1990 that computing hash functions on n bits requires circuit size $\geq cn \log n$. Their conjecture was disproved in 2008 [193] where a circuit of size cn was given.

9. The PCP Theorem 4.33 was considered unlikely by many, including Papadimitriou and Yannakakis who wrote in 1991: *"Furthermore, most problem reductions do not create or preserve such gaps. There would appear to be a last resort, namely to create such a gap in the generic reduction [...]. Unfortunately, this also seems doubtful. The intuitive reason is that computation is an inherently unstable, non-robust mathematical object, in the sense that it can be turned from non-accepting by changes that would be insignificant in any reasonable metric – say, by flipping a single state to accepting."*

10. In 1995 Miltersen [252] considered arithmetic data structures for polynomial evaluation (cf section §15.1). (An arithmetic data structure is one where the query algorithm is given by an arithmetic circuit.) Over large enough fields he proved that the trivial data structure is optimal. He conjectured that this lower bound holds for "smaller" fields as well. This was disproved in [60], building on the data structure in [208] (Theorem 15.7).

11. In number-on-forehead communication complexity, the function Majority-of-Majorities was raised as a candidate for being hard for $k \geq \log^{1+c} n$ players. This was disproved in [36] and subsequent works, where many other counter-intuitive protocols are presented, see section §13.2.2. For pointer chasing, similar bounds were claimed and then retracted multiple times (personal communications). The approach was either shown to require additional assumptions [99], or found in contradiction with the protocol in [285] (Theorem 13.29).

12. In [1] Aaronson made a conjecture which he called "generalized Linial-Nisan." The conjecture is that AC^0 is fooled not only by powerlog-wise uniform distributions (Theorem 11.8) but even by a broader class of distributions which are only close to powerlog-wise uniform in a suitable sense. Aaronson himself later disproved the conjecture for depth-3 circuits [2]. For depth-2 circuits see [15].

13. In [281] Patrascu made a conjecture in communication complexity, and showed that if true then many data-structure lower bounds would follow. He stated a general conjecture, which is overkill for the data-structure application, but has an appealing intuition. This strong form was refuted in [81].

14. For 30+ years the fastest run-time for graph isomorphism was exponential. A great deal was written on efficient proof systems for graph non-isomorphism. In 2015 Babai shocked the world with an almost power-time algorithm for graph isomorphism [33, 173].

15. In 2017 and 2018 a number of preprints considered cryptographic pseudorandom generators (section 11.5.2) that are 2-local or computable by quadratic polynomials. (It was shown that these generators, together with other techniques, can be used to construct other interesting objects.) Many of the generators proposed in these papers were quickly broken in [235, 45], where the reader can find more discussion and links to the preprints.

16. Maxflow is a central problem studied since the dawn of computer science. All solutions had running time $\geq n^{1+c}$, until a stunning quasi-linear algorithm obtained in 2022 [84].

17. Chattopadhyay, Hatami, Hosseini, Lovett, and Zuckerman [83] introduced a novel technique with which they established an xor lemma for low-degree polynomials and consequently new correlation bounds. In particular, they proved that the correlation of the xor of two majority functions with *constant-degree* polynomials is $\log^c n/n$, a result

for which previous xor lemmas do not seem sufficient. Note that the bound is indeed about the square of the bound in Exercise 9.58. The key ingredient in the approach in [83] is a restriction result about low-degree polynomials. They prove it for degree up to $c \log n$ and they conjecture it holds even for much larger degrees. Their conjecture was disproved in [194]. In fact, [194] rules out even weaker parameters and shows that what is proved in [83] is essentially the best possible.

The list goes on and on, and I haven't even touched on the many broken conjectures in cryptography. Many other surprising results permeate this book, even if they weren't specifically conjectured to be false.

On the other hand, the lower bounds we have are hardly surprising. (Of course, the issue may be that we can prove so few lower bounds that we shouldn't expect surprises.) Some of the undecidability results I do consider surprising, for example the solution to Hilbert's 10th problem. But what is actually surprising in those results are the *algorithms*, showing that even very restricted models can simulate more complicated ones. Ditto for NP completeness. In terms of lower bounds they all build on diagonalization, that is, go through every program and flip the answer, which is boring.

Here are two more, related points for my belief:

Stop right before major results.

Why do available techniques for impossibility results stop "right before" proving major results? This phenomenon appears to permeate complexity theory: see Chapter 9 (section §9.4), Chapter 13, and Chapter 14. For example, as we saw in Chapter 9, we have lower bounds of $2^{n^{c/d}}$ on the size of constant-depth depth-d alternating circuits, and stronger bounds would also apply to NC^1 and L. This was known since the 80's. It took almost 40 years to arrive at *exactly* the same scenario for arithmetic circuits: We have exponential lower bounds for constant-depth arithmetic circuits (Theorem 14.20) which stop just short of impacting general arithmetic circuits (Theorem 14.18). The depth reductions in the two lines of works are different enough, though related. The lower-bound proofs appear very different. Why are we witnessing two instances of this phenomenon? It seems to me hard to maintain that this is just coincidence. The most reasonable conclusion is that this happens because the major results are actually false.

Get stuck at the same point.

An issue related to the previous one is why the same impossibility results that we have are sometimes obtained via seemingly very different proofs. One of several examples: the polynomial method and the switching lemma give two different proofs that AC^0 can't compute parity, see Chapter 9. The proofs appear genuinely different. Why should different approaches stop at the same point, except because there is nothing else to prove?

The evidence is clear: we have grossly underestimated the reach of efficient computation, in a variety of contexts. All signs indicate that we will continue to see bigger and bigger surprises in upper bounds. The idea that lower bounds are obviously true and we just can't prove them is not only baseless but even clashes with historical evidence. Throughout history, science has often proved wrong those who refused to take things at face value.

18.1 Critique of common arguments for P $\neq$ NP

In this section I review and critique some common arguments for P $\neq$ NP.

Thousands of problems.

Instances of this argument include:

- "Among the NP-complete problems are many [...] for which serious effort has been expended on finding polynomial-time algorithms. Since either all or none of the NP-complete problems are in P, and so far none have been found to be in P, it is natural to conjecture that none are in P." [181], Page 341.
- "The class NP [...] contains thousands of different problems for which no efficient solving procedure is known."[132]
- "To my mind, however, the strongest argument for P $\neq$ NP involves the thousands of problems that have been shown to be NP-complete, and the thousands of other problems that have been shown to be in P. If just one of these problems had turned out to be both NP-complete and in P, that would've immediately implied P = NP. Thus, we could argue, the P $\neq$ NP hypothesis has had thousands of chances to be 'falsified by observation.'" [3]

I find these claims strange. In fact, the theory of NP completeness leads me to an opposite conclusion. As we saw in Chapters 4 and 5, see especially Problem 4.4, the problems can all be translated one into the other by extremely simple procedures, essentially doing *nothing*. In what sense are the problems different? I think a good definition of different is "not reducible to each other in a simple manner." Resolving the grand challenge probably requires fantastic insight. I doubt that the standard NP-completeness reductions (section §4.4) can be regarded as providing genuinely new perspectives.

Lots of people tried.

Here's an instance of this argument.

- "Many of these problems have arisen in vastly different disciplines, and were the subject of extensive research by numerous different communities of scientists and engineers. These essentially independent studies have all failed to provide efficient algorithms for

> solving these problems, a failure that is extremely hard to attribute to sheer coincidence or a stroke of bad luck."[132]

The fact that different communities of scientists have attacked these problems can indeed be a strength. Unshackled by the trends of the community, and without much interaction, the scientists may have been free to explore radically new ideas.

However, it can also be a weakness. Unaware of the well-studied pitfalls, and with little communication, these scientists are likely to all have followed the same route. Indeed, most of the countless bogus proofs claiming to resolve major open problem in complexity fail in one of only a handful of different ways; and a number of these proofs have been produced by scholars belonging to different communities (mathematics, engineering, etc.).

Catastrophe.

A common argument for P $\neq$ NP is that P = NP would cause a catastrophe. A vast literature would suddenly become worthless, most cryptography would be broken, etc. However, it's easy (and possibly reassuring to some) to consider scenarios in which P = NP would not cause a catastrophe. A trivial one is if the algorithms take time n^d for exceedingly large constant d. A less obvious scenario is that the algorithms use complicated or impractical component X (think the classification of finite simple groups, or the 4-color theorem, etc.). And then we would enter a phase in which for a problem you ask if it can be solved without using X. People continue to publish papers and use the cryptographic protocols regardless. The P = NP result just gives a more nuanced view. This would not be too different, perhaps, from the fact that the simplex algorithm is commonly used, even though there's a proof that it takes exponential time in some cases.

Appendix A

Miscellanea

Of all escapes from reality, mathematics is the most successful ever.

Here we present mathematical definitions and results that are used in the main text.

A.1 Logic

Fact A.1. For any logical statements P and Q, $\neg(P \vee Q) = \neg P \wedge \neg Q$, and $\neg(P \wedge Q) = \neg P \vee \neg Q$.

A.2 Integers

The *greatest common divisor* is denoted gcd.

Fact A.2. $\gcd(x, y) = \gcd(x \bmod y, y)$.

Next we prove the bound on the number of primes in Lemma 3.11. As we saw in section §1.7, factorials and binomials tell us a great deal about primes. So let us follow the lead.

Proof of Lemma 3.11. Consider

$$T := \binom{2t}{t}.$$

On the one hand

$$T \geq 2^{2t}/(2t+1) \tag{A.1}$$

because this is the largest binomial coefficient, and there are $2t+1$ such coefficients. A stronger bound holds (see Fact A.6) but we don't need it.

On the other hand we can prove an upper bound in terms of π. The main claim for this is that *any prime power dividing T is $\leq 2t$*. Let us first assume this and conclude the proof

of Lemma 3.11. Denote by $e_p(x)$ the exponent of p in the prime decomposition of x, i.e., the largest i s.t. p^i divides x. Then

$$T = \prod_{p \le 2t} p^{e_p(T)} \le \prod_{p \le 2t} 2t = (2t)^{\pi(2t)}.$$

Taking logarithms and using equation (A.1) we obtain

$$\pi(2t)(\log 2t) \ge 2t - \log(2t+1)$$

from which Lemma 3.11 follows.

There remains to prove the claim. For this we note that for $k \in \mathbb{N}$ and a prime p:

$$e_p(k!) = \sum_{i \ge 1} \left\lfloor \frac{k}{p^i} \right\rfloor. \tag{A.2}$$

The proof of this is left as exercise. Applying this to $T = \frac{(2t)!}{(t!)^2}$ we get

$$e_p(T) = \sum_{i \ge 1} \left\lfloor \frac{2t}{p^i} \right\rfloor - 2 \left\lfloor \frac{t}{p^i} \right\rfloor.$$

Now, if $x := t/p^i$ is of the form $j + \epsilon$ where $j \in \mathbb{N}$ and $\epsilon \in [0, 0.5)$ then $2\lfloor x \rfloor = \lfloor 2x \rfloor = 2j$ and each difference in the sum is 0; while if $\epsilon \in [0.5, 1)$ then $2\lfloor x \rfloor = 2j$, and $\lfloor 2x \rfloor = 2j+1$ and each difference is 1. In particular,

$$e_p(T) \le \sum_{i \ge 1 : p^i \le 2t} 1 \le \log_p 2t.$$

And so $p^{e_p(T)} \le p^{\log_p 2t} = 2t$. **QED**

Exercise A.3. Prove Equation (A.2).

A.3 Sums

Fact A.4. For any $\alpha \ne 1$, $\ell \in \mathbb{N}$ we have

$$1 + \alpha + \alpha^2 + \cdots + \alpha^\ell = \frac{1 - \alpha^{\ell+1}}{1 - \alpha}.$$

In particular, if $\alpha \in [0, 1/2]$ the lhs is ≤ 2.

Proof. The trick is to multiply the lhs by $1 - \alpha$ and note that every term cancels except 1 and $-\alpha^{\ell+1}$:

$$\left(1 + \alpha + \alpha^2 + \cdots + \alpha^\ell\right)(1 - \alpha) = 1 - \alpha^{\ell+1}.$$

Now if $\alpha \ne 1$ we can divide by $1 - \alpha$, concluding the proof. **QED**

A.4 Inequalities

Fact A.5. $1 + \alpha \leq e^{\alpha} \leq 1 + \alpha + \alpha^2$, for all $\alpha \leq 1$.

The rhs is $\leq 1 + 2\alpha$ for $\alpha \in [0, 1]$, and $\leq 1 + \alpha/2$ for $\alpha \in [-1/2, 0]$ (because $\alpha(1 + \alpha) \leq \alpha/2 \iff (1 + \alpha) \geq 1/2$).

Fact A.6. The largest binomial coefficient $\binom{n}{k}$ is attained at $k = n/2$ if n is even, and $k = n/2 \pm 1/2$ if n is odd. Its value is within a constant factor of $2^n/\sqrt{n}$.

Proof. We prove $\binom{2m}{m} \leq c2^{2m}/\sqrt{m}$ and leave the rest as exercise. By the binomial theorem

$$\sum_{k=0}^{2m} \binom{2m}{k} = 2^{2m}.$$

Since the maximum term is for $k = m$ we have

$$(2m + 1)\binom{2m}{m}^2 \leq \sum_{k=0}^{2m} \binom{2m}{k}\binom{2m}{2m-k} = \binom{4m}{2m}.$$

Hence

$$\binom{2m}{m}^2 \leq \frac{\binom{4m}{2m}}{2m+1} \leq \frac{2^{4m}}{2m+1}.$$

The result follows by taking square roots. **QED**

Fact A.7. $(n/k)^k \leq \binom{n}{k} \leq (en/k)^k$.

The proof uses the definition and Fact A.5 and is left as exercise.

Fact A.8. $(1 + \alpha)^r \geq 1 + r\alpha$ for all $\alpha \geq -1$ and $r \geq 1$.

Fact A.9. We have

$$1 + \alpha \leq \frac{1}{1 - \alpha} \leq 1 + (1 + \epsilon)\alpha$$

where the first inequality holds for any $\alpha \in \mathbb{R}$, the second for $\alpha \in [0, \epsilon/(1 + \epsilon)]$.

For example, $1/(1 - \alpha) \leq 1 + 2\alpha$ ($\epsilon = 1$) for $\alpha \leq 1/2$.

A.5 Probability theory

Developing intuition about random variables is one of the hardest skills to master, or even define. I believe the following joke well illustrates how elusive the concept of probability can be.

A certain professor of probability is well known for refusing to fly. He often tells his students:
"The probability that there's a bomb on a plane is not small enough for my taste."
One day, a student runs into him at the airport. The professor is checking in, carrying a large, heavy bag.
"Professor!" the student says. "I thought you never fly. Didn't you say the risk of a bomb was too high?"
"Yes – the professor replies smiling, patting his bag – but the probability that there are *two* bombs on the same plane is negligible. So now I just carry my own."

To anyone struggling I'd like to mention that my background was null, and I even felt that the emphasis on randomization was exaggerated (and sometimes I feel the same). Naturally, with effort I grew to like probability theory. I think of it simply as *normalized counting*, and I do find the normalization useful. Many times when reading a new result I find myself translating the statements in the language of probability to make them more "physical." Still, I find refreshing the rare occasions when there is a new cool result that does *not* involve randomness.

Definition A.10. Let P and Q be two distributions P and Q over a set X. The *statistical distance* between P and Q is

$$\frac{1}{2}\sum_{x\in X}|P(x)-Q(x)| = \max_{f:X\to[2]}|\mathbb{P}_P[f(P)=1]-\mathbb{P}_Q[(Q)=1]|.$$

The equivalence is left as exercise.

If two distributions have statistical distance ϵ then they can be typically used interchangeably up to an error of ϵ.

The *expectation* of a real-valued random variable X is

$$\mathbb{E}[X] := \sum_x x\cdot\mathbb{P}[X=x].$$

Fact A.11. [Linearity of expectation] $\mathbb{E}[aX+bY] = a\mathbb{E}[X]+b\mathbb{E}[Y]$, for any random variables X, Y and $a, b \in \mathbb{R}$.

Fact A.12. If X and Y are independent, real-valued random variables then $\mathbb{E}[X\cdot Y] = \mathbb{E}[X]\cdot\mathbb{E}[Y]$.

Fact A.13. Let X be a real-valued r.v. s.t. $X \geq 0$ always. Then $\mathbb{P}[X \geq t] \leq \mathbb{E}[X]/t$ for every $t > 0$.

Exercise A.14. Prove this.

A.5.1 Deviation bounds for the sum of random variables

Here we discuss probability bounds for the deviation of the sum of random variables from the mean. Such deviation bounds permeate theoretical computer science, and many other fields as well, see section §3.2. Fact A.13 gives a generic deviation bound showing that a deviation by ℓ times the expectation has prob. $\leq 1/\ell$. The next result gives a much sharper result for the sum of independent random variables.

Lemma A.15. Let $X_0, X_2, \ldots, X_{t-1}$ be i.i.d. boolean random variables with $p := \mathbb{P}[X_i = 1]$. Then for any $0 < p \leq q < 1$ we have $\mathbb{P}[\sum_{i\in[t]} X_i \geq qt] \leq 2^{-D(q|p)t}$, where

$$D(q|p) := q \log\left(\frac{q}{p}\right) + (1-q)\log\left(\frac{1-q}{1-p}\right)$$

is the *divergence.*

While the statement involves a complicated-looking quantity – divergence – it is the best bound on this pervasive quantity, and it is flexible: From it one can get a variety of inequalities by bounding divergence for different settings of parameter. We give an example and then state a general bound for D which we use shortly.

Example A.16. Let us toss t fair coins X_i, with $\mathbb{P}[X_i = \text{heads}] = 1/2 = p$. To bound the probability that we get $\geq 0.75 = q$ heads we compute the divergence $D(q|p) = 0.189\ldots$. Thus the probability is $\leq 2^{-D(q|p)t} \leq 2^{-0.189t}$. It decays exponentially fast and will be astronomically small even for moderate values of t.

Fact A.17. $D(q|p) \geq c(p-q)^2$, for any $p, q \in [0,1]$.

Proof of Lemma A.15. For $z \geq 1$, to be picked later, the function $x \to z^x$ is increasing. Using this and then Fact A.13 and finally the independence of the X_i, the lhs equals

$$\mathbb{P}[z^{\sum_{i=1}^t X_i} \geq z^{qt}] \leq \frac{\mathbb{E}[z^{\sum_{i=1}^t X_i}]}{z^{qt}} = \frac{\prod_{i=1}^t \mathbb{E}[z^{X_i}]}{z^{qt}} = \left(\frac{pz + 1 - p}{z^q}\right)^t =: b^t.$$

To minimize b we set

$$z := \frac{q(1-p)}{(1-q)p}.$$

This value can be derived using calculus, see Exercise A.19, or one can just remember it. Note $z \geq 1$ because $q \geq p$, and obtain

$$b = \frac{\frac{1-p}{1-q}}{z^q} = \left(\frac{p}{q}\right)^q \left(\frac{1-p}{1-q}\right)^{1-q}.$$

QED

The proof of the tail-bound Lemma A.15 is flexible and applies to a variety of useful settings. The most interesting extensions concern *dependent* random variables, where in general the bounds are weaker. In the next exercise we instead explore settings where the bounds in Lemma A.15 continue to hold; note independence is dropped in the last.

Exercise A.18. Prove that the tail bound in Lemma A.15 holds as stated more generally for any independent random variables $X_1, X_2, \ldots, X_t$ distributed in $[0,1]$ with $p := \sum_i \mathbb{E}[X_i]/t$. Guideline: Repeat the same proof as before. Use that $z^x \leq 1 + x(z-1)$ and the arithmetic-mean geometric mean inequality (AM-GM) inequality: for all $a_i \geq 0$: $(\sum_{i\in[t]} a_i)/t \geq (\prod_{i\in[t]} a_i)^{1/t}$.

Now suppose the X_i are more generally distributed in $[a,b]$. For $q = \epsilon + p$ prove a deviation bound of $2^{-c\epsilon^2 t/(b-a)^2}$.

Go back to the the tail bound in Lemma A.15. Prove it holds as stated even if the X_i are not independent, but conditioned on any $X_1, X_2, \ldots, X_{i-1}$, we have $\mathbb{E}[X_i] \leq p$.

Exercise A.19. Derive the minimizing value of z in the proof of Lemma A.15.

A.6 Squaring tricks

Fact A.20. For every real random variable X, $\mathbb{E}^2[X] \leq \mathbb{E}[X^2]$.

Equivalently, for any real vector v:$\sum_{i\in[n]} v_i \leq \sqrt{n}\sqrt{\sum_i v_i^2}$.

Proof. $\mathbb{E}[(X - \mathbb{E}[X])^2]$ is ≥ 0. Expand the square. **QED**

Exercise A.21. Prove the equivalence.

This is a special case of:

Fact A.22. For real random variables X, Y, jointly distributed: $\mathbb{E}^2[XY] \leq \mathbb{E}[X^2]\mathbb{E}[Y^2]$.

Proof. Let

$$Z := X - \frac{\mathbb{E}[XY]}{\mathbb{E}[Y^2]} Y.$$

Since $\mathbb{E}[Z^2] \geq 0$, expanding Z^2 concludes the proof. **QED**

Equivalently, for reals a_i, b_i one has $\left(\sum_i a_i b_i\right)^2 \leq \left(\sum_i a_i^2\right)\left(\sum_i b_i^2\right)$; and for vectors v, w one has $\langle v, w\rangle \leq |v| \cdot |w|$.

A.7 Duality

As mentioned in section §8.4, the following powerful result is known under a variety of names such as *the min-max theorem from game theory, linear-programming duality, the theorem of the alternatives for linear systems.*

Theorem A.23 (Linear duality)**.** Let X and Y be sets, $p : X \times Y \to \mathbb{R}$, and $\alpha \in \mathbb{R}$. Then either

(1) there is a distribution D on X s.t. $\mathbb{E}_D p(D, y) < \alpha$ for every $y \in Y$, or

(2) there is a distribution E on Y s.t. $\mathbb{E}_E p(x, E) \geq \alpha$ for every $x \in X$.

Equivalently, $\neg(1) \Rightarrow (2)$: If for every distribution D on X there is $y \in Y : \mathbb{E}_D p(D, y) \geq \alpha$ then (2). In words, if for every distribution on X there is y that gets $\geq \alpha$ in expectation, then in fact there is a single distribution on Y that for every x get $\geq \alpha$ in expectation.

Exercise A.24. Prove the "if" direction of Corollary 8.15 using Theorem A.23.

The proof of the duality Theorem A.23 relies on the following.

Lemma A.25. [Alternatives] Let A_i be affine functions from $\mathbb{R}^d$ to $\mathbb{R}$ (i.e., $A_i = \sum_{j \in [d]} a_{i,j} x_j + a_i$). Exactly one of the two holds:
(1) $\exists x : A_i(x) \geq 0$ for all i.
(2) There are $\lambda_i \geq 0 : \sum \lambda_i A_i(x) = -1$.

Proof. Induction on d. If $d = 0$ then the A_i are constants. If they are not all positive then scaling we can satisfy (2).

For the inductive step write $x \in \mathbb{R}^d$ as (x', t) where $t \in \mathbb{R}$. Up to a positive scaling each $A_i(x)$ can be written as one of the following three types: $A_i^+(x') + t$, $A_i^-(x') - t$, or $A_i^0(x')$.

Note that $A_i^+(x') + t \geq 0 \iff A_i^+(x') \geq -t \iff -A_i^+(x') \leq t$ and $A_i^-(x') - t \geq 0 \iff A_i^-(x') \geq t$.

Suppose there is $x' : A_i^0(x') \geq 0$ for all i and $A_j^-(x') \geq -A_k^+(x')$ for all j, k. Then we can find t to satisfy (1).

Otherwise, by induction there is a positive combination of A_i^0 and $A_i^+ + A_i^-$ yielding -1. But $A_i^+ + A_i^-$ is a positive combination of the A_i. **QED**

Proof of Theorem A.23.. Write d_x for the weight of D on x. For two distributions D and E write $P(D, E)$ for the *expected payoff* $\sum_{x,y} p(x, y) d_x e_y$. Suppose (1) is not true. Then the $|X| + 2 + |Y|$ inequalities

$$\begin{aligned} d_x &\geq 0 \quad \text{for } x \in X \\ 1 - \sum d_x &\geq 0 \\ \sum d_x - 1 &\geq 0 \\ \ell(y) := \alpha - \sum_x p(x, y) d_x &\geq 0 \quad \text{for } y \in Y \end{aligned}$$

have no solution. By Lemma A.25 there is a non-negative combination of the left-hand sides of these inequalities that gives -1. Let w_y be the weight for $\ell(y)$, and $W := \sum_y w_y$, $e_y := w_y / W$. Note

$$\sum_y w_y \ell(y) = W \sum_y e_y \left(\alpha - \sum_x p(x, y) d_x \right) = W(\alpha - \sum_{y,x} p(x, y) d_x e_y) = W(\alpha - P(D, E)).$$

Hence a non-negative combination of d_x for $x \in X$, $1 - \sum d_x$, $\sum d_x - 1$, and $W(\alpha - P(D, E))$ gives -1. This implies that for every distribution D on X we have $P(D, E) \geq \alpha$. **QED**

A.8 Groups

The theory of groups is *rich and pervasive.* A group is a set G equipped with an operation $\cdot$ called *multiplication* mapping $G^2 \to G$ and written xy for $x \cdot y$ which enjoys:

1. *Associativity*: $x(yz) = x(yz)$.
2. *Identity*: There exists an element $1_G \in G$ (also written 1) s.t. $1x = x1 = x$ for every $x \in G$.
3. *Inverse*: For every $x \in G$ there is x^{-1} (also written $1/x$) s.t. $xx^{-1} = x^{-1}x = 1$. We can think of multiplication by $1/x$ as *division* by x.

If in addition the operation $\cdot$ satisfies *commutativity*: $xy = yx$ for every $x, y \in G$ then the group is called *commutative.* In this case we sometimes use 0 for the identity element and $+$ for the operation, and we call $+$ *addition*.

Example A.26. $\mathbb{Z}_m$ is the group given by the set $[m]$ with addition modulo m. $\mathbb{Z}_m^t$ is the group given by the set $[m]^t$ with component-wise addition modulo m. These groups are commutative. Non-commutative groups include S_5, the group of permutations of 5 elements, and the group of invertible 3×3 matrices over the field $\mathbb{F}_2$ (fields are defined in section §A.9) , cf Chapter 7.

A group G is *cyclic* if it is generated by a single elements: $G = \{1, g, g^2, g^3, \ldots\}$.

Theorem A.27. [Fundamental theorem of commutative finite groups] A finite group is commutative iff it is a direct product of finite cyclic groups.

A subgroup N of G is *normal*, written $N \lhd G$ if it is invariant under conjugation: $g^{-1}Ng = N$ for every $g \in G$. Normal subgroups allow us to define the *quotient* group G/N, which is the group of cosets of N, with operation $(gN)(hN) := ghN$. A finite group G is *solvable* if there is a series

$$\{1\} = G_1 \lhd \cdots \lhd G_t = G$$

s.t. the quotient groups G_{i+1}/G_i are cyclic.

Fact A.28. Any finite, not solvable group has a non-trivial subgroup whose commutator subgroup (i.e., the subgroup generated by commutators) is itself.

Fact A.29. $g^G = 1$ for every group G and $g \in G$.

A.9 Fields

A field is a set F with two operations, $+$ called *addition* and $\cdot$ called *multiplication* such that:

1. F with $+$ is a commutative group with identity element 0_F (written 0).

2. $F - \{0\}$ with $\cdot$ is a commutative group with identity element 1_F (written 1).

3. *Distributivity* of $\cdot$ over $+$: $x(y+z) = xy + xz$.

By convention, $\cdot$ has precedence over $+$.

Example A.30. The reals $\mathbb{R}$ or the rationals $\mathbb{Q}$ are *infinite* fields. The integers modulo a prime p form a finite field. For $p = 2$ this gives the field with two elements where $+$ is Xor and $\cdot$ is And. For larger p you add and multiply as over the integers but then you take the result modulo p.

Fact A.31. [Finite fields] A finite field of size q exists iff $q = p^t$ where p is a prime and $t \in \mathbb{N} - \{0\}$. This field is unique and denoted $\mathbb{F}_q$.

Elements in the field can be identified with $\{0, 1, \ldots, p-1\}^t$.

Given q, one can compute a *representation* of a finite field of size q in time $(tp)^c$. This representation can be identified with p plus an element of $\{0, 1, \ldots, p-1\}^t$.

Given a representation r and field elements x, y computing $x+y$ and $x \cdot y$ is in $\text{Time}(n \log^c n)$.

Fields of size 2^t are of natural interest in computer science. It is often desirable to have very explicit representations for such and other fields. Such representations are known and are given by simple formulas, and are in particular computable in linear time.

Example A.32. We can represent the elements of $\mathbb{F}_{p^t}$ as (the coefficients of) polynomials of degree $< t$ over $\mathbb{F}_p$. Addition is done component-wise, and multiplication occurs modulo an irreducible polynomial of degree t over the base field $\mathbb{F}_p$, i.e., a polynomial that cannot be factored as the product of two non-constant polynomials.

While in general computing irreducible polynomials is non-trivial, there are some very explicit families. For example, it is known that the polynomial $z^t + z^{t/2} + 1$ is irreducible over $\mathbb{F}_2$ for $t = 2 \cdot 3^\ell$ for any ℓ, giving very explicit representations. To illustrate field operations, consider the field elements $z^2 + 1$ and $z^{t-1} + 1$ over such a representation of $\mathbb{F}_{2^t}$. Their sum equals $z^{t-1} + z^2$, and their product equals $z^{t+1} + z^2 + z^{t-1} + 1 = z^{t-1} + z^{t/2+1} + z^2 + z + 1$.

Fact A.33. The multiplicative group of a field $\mathbb{F}$ is *cyclic*, meaning there exists a generator $g \in \mathbb{F} - \{0\} : \forall x \in \mathbb{F} - \{0\}, x = g^i$, for some $i \in \mathbb{N}$.

We can think of i as the *logarithm* of x (with respect to the generator g).

Example A.34. For $\mathbb{F}_5$ we can take $g = 2$: $2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8 = 3$.

A.10 Linear algebra

The only game in town.

First we list some basic definitions. An n-dimensional vector over a field $\mathbb{F}$ is a tuple

$$v = (v_1, v_2, \ldots, v_n) \in \mathbb{F}^n.$$

Vectors v_i are *linearly independent* if there is no linear combination $\sum_i a_i v_i$ that gives 0, except if all $a_i = 0$. Equivalently, there are no two different linear combinations $\sum_i a_i v_i$ and $\sum_i a'_i v_i$ that give the same value. The *rank* of a set of vectors is the maximum number of linearly independent vectors in the set.

Fact A.35. In any matrix, the rank of the column vectors equals the rank of the row vectors.

The *inner product* between vectors v and w is $\langle v, w\rangle := \sum_i v_i \cdot w_i$.
Vectors v and w are *orthogonal*, denoted $v \perp w$, if $\langle v, w\rangle = 0$.
The length of a real vector v is $|v| = |v|_2 := \sqrt{\sum_i v_i^2} = \sqrt{\langle v, v\rangle}$.

Fact A.36. [Triangle inequality for vectors] $|v + w| \leq |v| + |w|$ for any real vectors v, w.

Fact A.37. If $v \perp w$, then $|v + w|^2 = |v|^2 + |w|^2$.

Proof. The lhs is $\sum_i (v(i) + w(i))^2$. Expand the square and use orthogonality. **QED**

Fact A.38. Orthogonal vectors are linearly independent.

Proof. Suppose that $\sum_i a_i v_i = 0$ for some coefficients a_i. Then for any j we have, using orthogonality,

$$0 = \langle \sum_i a_i v_i, v_j\rangle = \sum_i a_i \langle v_i, v_j\rangle = a_j \langle v_j, v_j\rangle,$$

and so $a_j = 0$. **QED**

We now illustrate a useful change of basis. We can view a function $f : \{-1, 1\}^n \to \mathbb{R}$ as a vector v of length 2^n whose x entry $v_x = f(x)$. The difference between functions and vectors is just notational. The function notation emphasizes that the indexing map from x to $f(x)$ is important. We can write f as a linear combination of the point functions $p_a(x)$ which are 1 if $a = x$ and 0 otherwise:

$$f = \sum_{a \in \{-1,1\}^n} f(a) \cdot p_a.$$

The $f(a) \in \mathbb{R}$ are the coefficients of the linear combination.
Equivalently, one can write an equation involving a generic point x:

$$f(x) = \sum_{a \in \{-1,1\}^n} f(a) \cdot p_a(x). \tag{A.3}$$

It is sometimes very convenient to write f as a linear combination of a different set of functions. Rather than the point functions p_a we shall consider the parity functions

$$x^S := \prod_{i \in S} x_i$$

for $S \subseteq [n]$. It turns out that every function can be written as a linear combination of the x^S; we indicate by $\hat{f}_S$ the corresponding coefficient.

Fact A.39 (Hypercube analysis)**.** Any $f : \{-1,1\}^n \to \mathbb{R}$ can be written as

$$f(x) = \sum_{S \subseteq [n]} \hat{f}_S \cdot x^S,$$

where
(1) $\hat{f}_S = \mathbb{E}[f(x)x^S] \in \mathbb{R}$.
(2) $\hat{f}_\emptyset = \mathbb{E}[f(x)]$.
(3) $\mathbb{E}[f^2(x)] = \sum_S \hat{f}_S^2$.
(4) $\sum_S \left|\hat{f}_S\right| \leq 2^{n/2}$.

Exercise A.40. Prove this. Start with equation (A.3) and suitably rewrite the point functions p_a. For (3) use Fact A.20.

Fact A.41. Let $\alpha_i, i \in [n+1]$ be different elements from a field $\mathbb{F}$. Then the vectors $(\alpha_i^0, \alpha_i^1, \ldots, \alpha_i^n)$ are linearly independent.

Proof. We show it instead for the vectors $(\alpha_0^j, \alpha_1^j, \ldots, \alpha_n^j)$, then appeal to the fact that row rank equals column rank (for the $n+1 \times n+1$ matrix α_i^j). (The proof of this is left as exercise, or see the notes.) Suppose there are a_j, not all zero, s.t.:

$$\sum_{j \in [n+1]} a_j(\alpha_0^j, \alpha_1^j, \ldots, \alpha_n^j) = (0, 0, \ldots, 0).$$

Then the α_i are roots of the non-zero polynomial $\sum_{j \in [n+1]} a_j x^j$ of degree n. This contradicts Fact A.50. **QED**

Definition A.42. Let A be a $n \times m$ matrix, and B be a $n' \times m'$ matrix. The *tensor product* of A and B is is an $n \cdot n' \times m \cdot m'$ matrix defined as $(A \otimes B)_{(iA,iB),(jA,jB)} = A_{iA,jA} \cdot B_{iB,jB}$.

Diagrammatically,

$$A \otimes B = \begin{bmatrix} a_{11}B & \ldots & a_{1n}B \\ \vdots & \ldots & \vdots \\ a_{n1}B & \ldots & a_{nn}B \end{bmatrix}.$$

But the algebraic Definition A.42 makes most sense and is almost always more convenient.

Fact A.43. $\text{rank}(A \otimes B) = \text{rank}(A) \cdot \text{rank}(B)$.

Fact A.44. Every matrix is a root of its characteristic polynomial: If $p_A(x) := \det(xI - A)$ then $p_A(A) = 0$.

Definition A.45. The *operator norm* of an $n \times n$ matrix A is

$$|A|_o := \sup_{v \in \mathbb{R}^n, v \neq 0} \frac{|Av|}{|v|} = \sup_{v \in \mathbb{R}^n, |v|=1} |Av|.$$

Lemma A.46. [Properties of operator norm] Both $|A \cdot B|_o$ and $|A \otimes B|_o$ are $\leq |A|_o \cdot |B|_o$.

Proof. To bound $|A \cdot B|_o$ note $|ABv| \leq |Bv| \leq |v|$, with the first inequality holding even if $Bv = 0$.

For $|A \otimes B|_o$ write

$$\begin{aligned}
|(A \otimes B)v|^2 &= \sum_{iA,iB} \left(\sum_{jA,jB} A_{iA,jA} B_{iB,jB} v_{jA,jB} \right)^2 \\
&= \sum_{iB} \sum_{iA} \left(\sum_{jA} A_{iA,jA} \left(\sum_{jB} B_{iB,jB} v_{jA,jB} \right) \right)^2 \\
&\leq |A|_o^2 \cdot \sum_{iB} \sum_{jA} \left(\sum_{jB} B_{iB,jB} v_{jA,jB} \right)^2 \\
&\leq |A|_o^2 \cdot |B|_o^2 \cdot \sum_{jA} \sum_{jB} v_{jA,jB}^2 \\
&= |A|_o^2 \cdot |B|_o^2 \cdot |v|.
\end{aligned}$$

QED

A.11 Polynomials

Fact A.47. Let p and q be multi-variate polynomials over a field. Define the *degree* deg of a polynomial as the maximum sum of exponents of any monomial. Then $\deg(p \cdot q) = \deg(p) + \deg(q)$.

Proof. $\leq$ is obvious. $\geq$ is not because some terms may cancel. Define an ordering on monomials where larger degree comes first, and for equal degree we use lexicographic order. (That is, first compare the exponent of x_1, if equal compare the exponents of x_2, and so on.) We claim that the product of the first (in this order) monomial in p times the first monomial in q occurs in no other way, because if $m_1 > m_2$ and $m_3 > m_4$ then $m_1 \cdot m_3 > m_2 \cdot m_4$, where the m_i are any monomials. The result follows. **QED**

Definition A.48. The *elementary symmetric polynomial* of degree d (in n variables):

$$e_d(x_1, x_2, \ldots, x_n) := \sum_{S \subseteq [1..n], |S| = d} \prod_{i \in S} x_i.$$

The *power sum polynomial* of degree d in (n variables):

$$p_d(x_1, x_2, \ldots, x_n) := \sum_{i=1}^{n} x_i^d.$$

We note

$$\begin{aligned} e_1 &= p_1 \\ e_2 &= p_1^2/2 - p_2/2 \\ e_3 &= p_1^3 - 3p_1p_2/2 + p_3 \\ &\dots \end{aligned}$$

and so on, which convinces us that we can express the e_k via the p_k. Indeed, we have:

Fact A.49. $k \cdot e_k = \sum_{i=1}^{k} (-1)^{i-1} e_{k-i} \cdot p_i$, for all k.

Proof. Let $f(x) := (x-x_1)(x-x_2)\cdots(x-x_n)$. The coefficient of x^i in $f(x)$ is $e_{n,n-i}(x_1, x_2, \dots, x_n) \cdot (-1)^{n-i}$. We also have $f(x_j) = 0$ for any j. Summing over j proves the claim for $k = n$. To prove it for $n > k$, consider any monomial on the lhs. Set to 0 all the other variables. Now the claim reduces to the case $n = k$. This shows that the coefficient of any monomial on the lhs is the same asthat on the rhs, finishing the proof. **QED**

Fact A.50. [Polynomial identity] Let p be a polynomial over a field $\mathbb{F}$ with n variables and degree $\leq d$. Let S be a finite subset of $\mathbb{F}$, and suppose $d < |S|$. The following are equivalent:

p is the zero polynomial.

$p(x) = 0$ for every $x \in \mathbb{F}^n$.

$\mathbb{P}_{x_1,x_2,\dots,x_n \in S}[p(x) = 0] > d/|S|$.

Proof of Fact A.50.. The implications 1. $\Rightarrow$ 2. $\Rightarrow$ 3. are trivial, but note that for the latter we need $d < |S|$. The implication 3. $\Rightarrow$ 1. is not trivial. We proceed by induction on n.

The base case $n = 1$ is the fact that if p has more than d roots then it is the zero polynomial. This fact in turn can be proved by induction on the degree. The base case $d = 0$ is obvious. For larger d, suppose a is a root of p and use division for polynomials to write $p = (x - a)q + r$ where q has degree $\leq d - 1$ and $r \in \mathbb{F}$. Because a is a root we have $r = 0$, and so $p = (x - a)q$ and q has $d - 1$ roots, and by induction $q = 0$ and so $p = 0$.

For larger n write

$$p(x_1, x_2, \dots, x_n) = \sum_{i=0}^{d} x_1^i p_i(x_2, x_3, \dots, x_n).$$

If p is not the zero polynomial then there is at least one i such that p_i is not the zero polynomial. Let j be the largest such i. Note that p_j has degree at most $d-j$. By induction hypothesis

$$\mathbb{P}_{x_2,\dots,x_n \in S}[p_j(x) = 0] \leq (d - j)/|S|.$$

For every choice of $x_2, x_3, \dots, x_n$ s.t. $p_j(x) \neq 0$, the polynomial p is a non-zero polynomial $q_{x_2,x_3,\dots,x_n}(x_1)$ only in the variable x_1. Moreover, its degree is at most j by our choice of j. Hence by the $n = 1$ case the probability that q is 0 over the choice of x_1 is $\leq j$.

Overall,

$$\mathbb{P}_{x_1,x_2,\ldots,x_n \in S}[p(x) = 0] \leq (d-j)/|S| + j/|S| = d/|S|.$$

QED

Exercise A.51. Show that the equivalence between 1. and 2. does not hold over small fields such as $\mathbb{F}_2$ and large d.

A.12 Notes

Many of the results covered in this chapter are standard and can be found in typical textbooks. A tail bound similar to Lemma A.15 was first proved in 1938, see [95] and [96] for a translation. Since then, there has been an explosion of such bounds and proofs in the literature. Lemma A.15 as stated first appeared in [86]. For a computer-science friendly introduction to this theory see the book [107]. For a number of different proofs of Lemma A.15 see [259].

For more on groups see e.g. [74] ("solvable" is written "soluble" in the text).

For more on finite fields see e.g. [230]. The reference for Fact A.31 is [314]. For the fields $\mathbb{F}_{2^t}$ in Example A.32 see Theorem 1.1.28 in [354].

For a history of the Polynomial Identity Fact A.50 and related results, see [62]. One can get a sharper bound taking into account the individual degrees of the variables, in addition to the total degree.

For more on hypercube analysis (Fact A.39) see [274].

For more on linear algebra, including that row rank equals column rank, see for example [341].

The duality Theorem A.23 is from [379]. There are closely related formulations of Lemma A.25, which is originally from [110].

The introductory quote is from [297].

Appendix A

Landscape

The next diagram contains some of the main complexity classes in this book. Containments are expressed as follows. We have $X \subseteq Y$ if Y is to the right of X or below X. The two classes RP and NP "wrap around" to the next line: by definition RP $\subseteq$ BPP and NP $\subseteq \Sigma_2$P.

	NC^0	
FO	AC^0	
	AC^0-Mod-2	
	AC^0-Mod-6	
FOM	TC^0	
	NC^1	
L	BrL	
NL	AC^1	
	NC^2	
P		
ZPP		RP
BPP	CktP	
		NP
Σ_2P		
PH		
BP $\oplus$ P		
SymP		
PSpace = IP		
Exp		NExp

References

[1] Scott Aaronson. BQP and the polynomial hierarchy. In *42nd ACM Symp. on the Theory of Computing (STOC)*, pages 141–150. ACM, 2010.

[2] Scott Aaronson. A counterexample to the generalized linial-nisan conjecture. *Electron. Colloquium Comput. Complex.*, TR10-109, 2010.

[3] Scott Aaronson. P = np? In John Forbes Nash Jr. and Michael Th. Rassias, editors, *Open Problems in Mathematics*, pages 1–122. Springer, 2016.

[4] Scott Aaronson and Avi Wigderson. Algebrization: a new barrier in complexity theory. In *40th ACM Symp. on the Theory of Computing (STOC)*, pages 731–740, 2008.

[5] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 59–78. IEEE Computer Society, 2015.

[6] Anil Ada, Arkadev Chattopadhyay, Omar Fawzi, and Phuong Nguyen. The NOF multiparty communication complexity of composed functions. *Comput. Complex.*, 24(3):645–694, 2015.

[7] Leonard Adleman. Two theorems on random polynomial time. In *19th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 75–83. 1978.

[8] Peyman Afshani, Casper Benjamin Freksen, Lior Kamma, and Kasper Green Larsen. Lower bounds for multiplication via network coding. In *46th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:12. Schloss Dagstuhl – Leibniz-Zentrum fÃŒr Informatik, 2019.

[9] M. Agrawal, N. Kayal, and N. Saxena. Primes in p. *Annals of Pure Mathematics*, 160(2):781–793, 2004.

[10] Manindra Agrawal and V. Vinay. Arithmetic circuits: A chasm at depth four. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 67–75. IEEE Computer Society, 2008.

[11] Miklós Ajtai. Σ^1_1-formulae on finite structures. *Annals of Pure and Applied Logic*, 24(1):1–48, 1983.

[12] Miklós Ajtai. Approximate counting with uniform constant-depth circuits. In *Advances in computational complexity theory*, pages 1–20. Amer. Math. Soc., Providence, RI, 1993.

[13] Miklós Ajtai. A non-linear time lower bound for boolean branching programs. *Theory of Computing*, 1(1):149–176, 2005.

[14] Miklós Ajtai, János Komlós, and Endre Szemerédi. Deterministic simulation in logspace. In *19th ACM Symp. on the Theory of Computing (STOC)*, pages 132–140, 1987.

[15] Yaroslav Alekseev, Mika Göös, Ziyi Guan, Gilbert Maystre, Artur Riazanov, Dmitry Sokolov, and Weiqiang Yuan. Generalised Linial-Nisan conjecture is False for DNFs. Technical Report TR25-058, Electronic Colloquium on Computational Complexity

(ECCC), May 2025.

[16] Romas Aleliunas, Richard M. Karp, Richard J. Lipton, László Lovász, and Charles Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979*, pages 218–223. IEEE Computer Society, 1979.

[17] Eric Allender. A note on the power of threshold circuits. In *30th Symposium on Foundations of Computer Science*, pages 580–584, Research Triangle Park, North Carolina, 30 October–1 November 1989. IEEE.

[18] Eric Allender. The division breakthroughs. *Bulletin of the EATCS*, 74:61–77, 2001.

[19] Eric Allender. Arithmetic circuits and counting complexity classes. *Complexity of Computations and Proofs, Quaderni di Matematica Vol. 13, Seconda Universita di Napoli*, 2004.

[20] Eric Allender and Michal Koucký. Amplifying lower bounds by means of self-reducibility. *J. of the ACM*, 57(3), 2010.

[21] Josh Alman and R. Ryan Williams. Probabilistic rank and matrix rigidity. In *ACM Symp. on the Theory of Computing (STOC)*, pages 641–652, 2017.

[22] Noga Alon, Alexandr Andoni, Tali Kaufman, Kevin Matulef, Ronitt Rubinfeld, and Ning Xie. Testing k-wise and almost k-wise independence. In *ACM Symp. on the Theory of Computing (STOC)*, pages 496–505, 2007.

[23] Noga Alon, László Babai, and Alon Itai. A fast and simple randomized algorithm for the maximal independent set problem. *Journal of Algorithms*, 7:567–583, 1986.

[24] Noga Alon, Oded Goldreich, Johan Håstad, and René Peralta. Simple constructions of almost k-wise independent random variables. *Random Structures & Algorithms*, 3(3):289–304, 1992.

[25] Noga Alon, Oded Goldreich, and Yishay Mansour. Almost k-wise independence versus k-wise independence. *Inf. Process. Lett.*, 88(3):107–110, 2003.

[26] Noga Alon, Oded Schwartz, and Asaf Shapira. An elementary construction of constant-degree expanders. *Comb. Probab. Comput.*, 17(3):319–327, 2008.

[27] Robert Andrews and Avi Wigderson. Constant-depth arithmetic circuits for linear algebra problems. *Electronic Colloquium on Computational Complexity (ECCC)*, 80, 2024.

[28] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in NC^0. *SIAM J. on Computing*, 36(4):845–888, 2006.

[29] Sanjeev Arora and Boaz Barak. *Computational Complexity*. Cambridge University Press, 2009. A modern approach.

[30] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *J. of the ACM*, 45(3):501–555, May 1998.

[31] James Aspnes, Richard Beigel, Merrick Furst, and Steven Rudich. The expressive power of voting polynomials. *Combinatorica. An Journal on Combinatorics and the Theory of Computing*, 14(2):135–148, 1994.

[32] László Babai. E-mail and the unexpected power of interaction. In *SCT*, pages 30–44.

IEEE Computer Society, 1990.

[33] László Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 684–697. ACM, 2016.

[34] László Babai, Lance Fortnow, and Carsten Lund. Nondeterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1(1):3–40, 1991.

[35] 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.

[36] László Babai, Anna Gál, Peter G. Kimmel, and Satyanarayana V. Lokam. Communication complexity of simultaneous messages. *SIAM J. on Computing*, 33(1):137–166, 2003.

[37] László Babai, Thomas P. Hayes, and Peter G. Kimmel. The cost of the missing bit: communication complexity with help. *Combinatorica. An Journal on Combinatorics and the Theory of Computing*, 21(4):455–488, 2001.

[38] László Babai and Shlomo Moran. Arthur-merlin games: A randomized proof system, and a hierarchy of complexity classes. *J. Comput. Syst. Sci.*, 36(2):254–276, 1988.

[39] László Babai, Noam Nisan, and Márió Szegedy. Multiparty protocols, pseudorandom generators for logspace, and time-space trade-offs. *J. of Computer and System Sciences*, 45(2):204–232, 1992.

[40] Arturs Backurs. *Below P vs NP: fine-grained hardness for big data problems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, USA, 2018.

[41] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM J. Comput.*, 47(3):1087–1097, 2018.

[42] Alan Baker. *Transcendental number theory*. Cambridge Mathematical Library. Cambridge University Press, second edition, 1990.

[43] Theodore Baker, John Gill, and Robert Solovay. Relativizations of the P=?NP question. *SIAM J. on Computing*, 4(4):431–442, 1975.

[44] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *J. of Computer and System Sciences*, 68(4):702–732, 2004.

[45] Boaz Barak, Zvika Brakerski, Ilan Komargodski, and Pravesh K. Kothari. Limits on low-degree pseudorandom generators (or: Sum-of-squares meets program obfuscation). In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 649–679. Springer, 2018.

[46] Boaz Barak, Moritz Hardt, and Satyen Kale. The uniform hardcore lemma via approximate bregman projections. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 1193–1200, USA, 2009. Society for Industrial and Applied Mathematics.

[47] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[48] Walter Baur and Volker Strassen. The complexity of partial derivatives. *Theoret. Comput. Sci.*, 22(3):317–330, 1983.

[49] Paul Beame. A switching lemma primer. Technical Report UW-CSE-95-07-01, Department of Computer Science and Engineering, University of Washington, November 1994. Available from http://www.cs.washington.edu/homes/beame/.

[50] Paul Beame, Stephen A. Cook, and H. James Hoover. Log depth circuits for division and related problems. *SIAM J. Comput.*, 15(4):994–1003, 1986.

[51] Paul Beame, Matei David, Toniann Pitassi, and Philipp Woelfel. Separating deterministic from nondeterministic nof multiparty communication complexity. In *34th Coll. on Automata, Languages and Programming (ICALP)*, pages 134–145. Springer, 2007.

[52] Paul Beame, Michael Saks, Xiaodong Sun, and Erik Vee. Time-space trade-off lower bounds for randomized computation of decision problems. *J. of the ACM*, 50(2):154–195, 2003.

[53] Donald Beaver and Joan Feigenbaum. Hiding instances in multioracle queries. In *Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 37–48, 1990.

[54] Undefined Behavior. What makes mario NP-hard? https://www.youtube.com/watch?v=oS8m9fSk-Wk, 2019. YouTube video, uploaded Jan 28, 2019.

[55] Richard Beigel. When do extra majority gates help? polylog(N) majority gates are equivalent to one. *Computational Complexity*, 4(4):314–324, 1994.

[56] Richard Beigel, Nick Reingold, and Daniel Spielman. The perceptron strikes back. In *Structure in Complexity Theory Conference*, pages 286–291, 1991.

[57] Richard Beigel and Jun Tarui. On ACC. *Computational Complexity*, 4(4):350–366, 1994.

[58] Michael Ben-Or and Richard Cleve. Computing algebraic formulas using a constant number of registers. *SIAM J. on Computing*, 21(1):54–58, 1992.

[59] Stuart J. Berkowitz. On computing the determinant in small parallel time using a small number of processors. *Inform. Process. Lett.*, 18(3):147–150, 1984.

[60] Vishwas Bhargava, Sumanta Ghosh, Mrinal Kumar, and Chandra Kanta Mohapatra. Fast, algebraic multivariate multipoint evaluation in small characteristic and applications. *J. ACM*, 70(6):42:1–42:46, 2023.

[61] Yonatan Bilu and Nathan Linial. Lifts, discrepancy and nearly optimal spectral gaps. *Combinatorica*, 26(5):495–519, 2006.

[62] Anurag Bishnoi, Pete L. Clark, Aditya Potukuchi, and John R. Schmitt. On zeros of a polynomial in a finite grid. *Comb. Probab. Comput.*, 27(3):310–333, 2018.

[63] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. on Computing*, 13(4):850–864, November 1984.

[64] Andrej Bogdanov and Emanuele Viola. Pseudorandom bits for polynomials. *SIAM J. on Computing*, 39(6):2464–2486, 2010.

[65] Allan Borodin. Computational complexity and the existence of complexity gaps. *Jour-*

nal of the ACM, 19(1):158–174, 1972.

[66] Allan Borodin, Danny Dolev, Faith E. Fich, and Wolfgang J. Paul. Bounds for width two branching programs. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 87–93, 1983.

[67] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

[68] Joshua Brody and Amit Chakrabarti. Sublinear communication protocols for multi-party pointer jumping and a related lower bound. In *25th Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 145–156, 2008.

[69] Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In *ACM Symp. on the Theory of Computing (STOC)*, pages 857–866, 2014.

[70] Harry Buhrman, Peter Bro Miltersen, Jaikumar Radhakrishnan, and Venkatesh Srinivasan. Are bitvectors optimal? *SIAM J. on Computing*, 31(6):1723–1744, 2002.

[71] Peter Bürgisser. *Completeness and Reduction in Algebraic Complexity Theory*, volume 7 of *Algorithms and computation in mathematics*. Springer, 2000.

[72] Peter Bürgisser. On defining integers and proving arithmetic circuit lower bounds. *Comput. Complex.*, 18(1):81–103, 2009.

[73] Peter Bürgisser, Michael Clausen, and Mohammad Amin Shokrollahi. *Algebraic complexity theory*, volume 315 of *Grundlehren der mathematischen Wissenschaften*. Springer, 1997.

[74] William Burnside. *Theory of Groups of Finite Order*. Cambridge University Press, Cambridge, 2nd edition, 1911.

[75] Sam Buss, Anant Dhayal, Valentine Kabanets, Antonina Kolokolova, and Sasank Mouli. A logspace constructive proof of $\mathsf{SL} = \mathsf{L}$. *arXiv*, abs/2511.12011, 2025. 39 pages.

[76] Sam Buss, Valentine Kabanets, Antonina Kolokolova, and Michal Koucký. Expander construction in vnc^1. *Ann. Pure Appl. Log.*, 171(7):102796, 2020.

[77] Samuel R. Buss and Ryan Williams. Limits on alternation trading proofs for time-space lower bounds. *Comput. Complex.*, 24(3):533–600, 2015.

[78] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *J. of Computer and System Sciences*, 18(2):143–154, 1979.

[79] Sílvia Casacuberta, Cynthia Dwork, and Salil Vadhan. Complexity-theoretic implications of multicalibration. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, STOC 2024, pages 1071–1082, New York, NY, USA, 2024. Association for Computing Machinery.

[80] Ashok K. Chandra, Merrick L. Furst, and Richard J. Lipton. Multi-party protocols. In *15th ACM Symp. on the Theory of Computing (STOC)*, pages 94–99, 1983.

[81] Arkadev Chattopadhyay, Jeff Edmonds, Faith Ellen, and Toniann Pitassi. A little advice can be very helpful. In *SODA*, pages 615–625. SIAM, 2012.

[82] Arkadev Chattopadhyay and Toniann Pitassi. The story of set disjointness. *SIGACT*

News, 41(3):59–85, 2010.

[83] Eshan Chattopadhyay, Pooya Hatami, Kaave Hosseini, Shachar Lovett, and David Zuckerman. XOR lemmas for resilient functions against polynomials. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *ACM Symp. on the Theory of Computing (STOC)*, pages 234–246. ACM, 2020.

[84] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Almost-linear-time algorithms for maximum flow and minimum-cost flow. *Communications of the ACM*, 66(12):85–92, 2023.

[85] Lijie Chen and Roei Tell. Bootstrapping results for threshold circuits "just beyond" known lower bounds. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 34–41. ACM, 2019.

[86] Herman Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Ann. Math. Statistics*, 23:493–507, 1952.

[87] Benny Chor and Oded Goldreich. Unbiased bits from sources of weak randomness and probabilistic communication complexity. *SIAM J. on Computing*, 17(2):230–261, 1988.

[88] Benny Chor, Oded Goldreich, Johan Håstad, Joel Friedman, Steven Rudich, and Roman Smolensky. The bit extraction problem or t-resilient functions (preliminary version). In *26th Symposium on Foundations of Computer Science*, pages 396–407, Portland, Oregon, 21–23 October 1985. IEEE.

[89] Fan R. K. Chung and Prasad Tetali. Communication complexity and quasi randomness. *SIAM Journal on Discrete Mathematics*, 6(1):110–123, 1993.

[90] Richard Cleve. Towards optimal simulations of formulas by bounded-width programs. *Computational Complexity*, 1:91–105, 1991.

[91] James Cook and Ian Mertz. Tree evaluation is in space o(log n · log log n). In *STOC*, pages 1268–1278. ACM, 2024.

[92] James Cook and Edward Pyne. Efficient catalytic graph algorithms. *CoRR*, abs/2509.06209, 2025.

[93] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[94] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing (STOC)*, pages 73–80. ACM, 1972.

[95] Harald Cramér. Sur un nouveau théorème-limite de la théorie des probabilités. In *Actualités Scientifiques et Industrielles*, volume 736, pages 5–23, Paris, 1938. Hermann et Cie. Colloque consacré à la théorie des probabilités.

[96] Harald Cramér and Hugo Touchette. On a new limit theorem in probability theory (translation of 'sur un nouveau théorème-limite de la théorie des probabilités'), 2018. Translation by Hugo Touchette of the original 1938 French paper.

[97] L. Csanky. Fast parallel matrix inversion algorithms. *SIAM J. Comput.*, 5(4):618–623, 1976.

[98] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Verlag, 2002.

[99] Carsten Damm, Stasys Jukna, and Jiří Sgall. Some bounds on multiparty communication complexity of pointer jumping. *Computational Complexity*, 7(2):109–127, 1998.

[100] Matei David, Toniann Pitassi, and Emanuele Viola. Improved separations between nondeterministic and randomized multiparty communication. *ACM Trans. Computation Theory*, 1(2):1–20, 2009.

[101] K. De Leeuw, Edward F. Moore, Claude E. Shannon, and Norman Shapiro. Computability by probabilistic machines. In *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 183–198. Princeton University Press, Princeton, N.J., 1956.

[102] Harm Derksen, Peter Ivanov, Chin Ho Lee, and Emanuele Viola. Pseudorandomness, symmetry, smoothing: I. In *Conf. on Computational Complexity (CCC)*, 2024.

[103] Harm Derksen, Peter Ivanov, Chin Ho Lee, and Emanuele Viola. Pseudorandomness, symmetry, smoothing: II. 2024.

[104] Harm Derksen and Emanuele Viola. Fooling polynomials using invariant theory. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, 2022.

[105] Scott Diehl and Dieter van Melkebeek. Time-space lower bounds for the polynomial-time hierarchy on randomized machines. *SIAM J. on Computing*, 36(3):563–594, 2006.

[106] Yevgeniy Dodis, Mihai Pătraşcu, and Mikkel Thorup. Changing base without losing space. In *42nd ACM Symp. on the Theory of Computing (STOC)*, pages 593–602. ACM, 2010.

[107] Devdatt Dubhashi and Alessandro Panconesi. *Concentration of measure for the analysis of randomized algorithms*. Cambridge University Press, 2009.

[108] Pál Erdős, Ronald L. Graham, and Endre Szemerédi. On sparse graphs with dense long paths. *Comp. and Maths. with Appls.*, 1:365–369, 1975.

[109] Euclid. *The Thirteen Books of Euclid's Elements, Vol. 2*. Dover Publications, New York, 1956. Originally published in 300 B.C.

[110] Julius Farkas. Theorie der einfachen ungleichungen. *Journal für die reine und angewandte Mathematik*, 124:1–27, 1902.

[111] Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. Computing with noisy information. *SIAM J. Comput.*, 23(5):1001–1018, 1994.

[112] Michael A. Forbes. Low-depth algebraic circuit lower bounds over any field. In *CCC*, volume 300 of *LIPIcs*, pages 31:1–31:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[113] Lance Fortnow. Time-space tradeoffs for satisfiability. *J. Comput. Syst. Sci.*, 60(2):337–353, 2000.

[114] Lance Fortnow. A simple proof of toda's theorem. *Theory Comput.*, 5(1):135–140, 2009.

[115] Lance Fortnow, Richard Lipton, Dieter van Melkebeek, and Anastasios Viglas. Time-space lower bounds for satisfiability. *J. of the ACM*, 52(6):835–865, 2005.

[116] Caxton C. Foster and Fred D. Stockton. Counting responders in an associative memory. *IEEE Trans. Computers*, 20(12):1580–1583, 1971.

[117] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *ACM Symp. on the Theory of Computing (STOC)*, pages 345–354, 1989.

[118] Michael L. Fredman and Dan Dominic Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[119] R. Freivalds. Probabilistic machines can use less running time. In *IFIP Congress*, pages 839–842, 1977.

[120] Rusins Freivalds. Probabilistic two-way machines. In *MFCS*, volume 118 of *Lecture Notes in Computer Science*, pages 33–45. Springer, 1981.

[121] Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984.

[122] Anka Gajentaan and Mark H. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Comput. Geom.*, 5:165–185, 1995.

[123] Anna Gál, Kristoffer Arnsfelt Hansen, Michal Koucký, Pavel Pudlák, and Emanuele Viola. Tight bounds on computing error-correcting codes by bounded-depth circuits with arbitrary gates. *IEEE Transactions on Information Theory*, 59(10):6611–6627, 2013.

[124] Anna Gál and Peter Bro Miltersen. The cell probe complexity of succinct data structures. *Theoretical Computer Science*, 379(3):405–417, 2007.

[125] Howard Gardner. *The Mind's New Science: A History of the Cognitive Revolution*. Basic Books, New York, 1985. According to this book, John von Neumann remarked "It's all over" upon learning of Gödel's incompleteness theorem. This phrasing appears, without citation, in Gardner's boo.

[126] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[127] Peter Gemmell, Richard Lipton, Ronitt Rubinfeld, Madhu Sudan, and Avi Wigderson. Self-testing/correcting for polynomials and for approximate functions. In *Twenty Third ACM Symposium on Theory of Computing*, pages 32–42, New Orleans, Louisiana, 6–8 May 1991.

[128] Kurt Godel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme, i. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[129] Kurt Godel, 1956. Letter to John von Neumann. https://www.anilada.com/notes/godel-letter.pdf.

[130] Mikael Goldmann, Johan Håstad, and Alexander A. Razborov. Majority gates vs. general weighted threshold gates. *Computational Complexity*, 2:277–300, 1992.

[131] Oded Goldreich. A sample of samplers - a computational perspective on sampling (survey). *Electronic Coll. on Computational Complexity (ECCC)*, 4(020), 1997.

[132] Oded Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008.

[133] Oded Goldreich. On doubly-efficient interactive proof systems. *Found. Trends Theor.*

Comput. Sci., 13(3):158–246, 2018.

[134] Oded Goldreich. On the cook-mertz tree evaluation procedure. In Oded Goldreich, editor, *Computational Complexity and Local Algorithms - On the Interplay Between Randomness and Computation*, volume 15700 of *Lecture Notes in Computer Science*, pages 102–112. Springer, 2025.

[135] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. of the ACM*, 33(4):792–807, October 1986.

[136] Oded Goldreich and Leonid Levin. A hard-core predicate for all one-way functions. In *21st ACM Symp. on the Theory of Computing (STOC)*, pages 25–32, 1989.

[137] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity for all languages in NP have zero-knowledge proof systems. *J. ACM*, 38(3):691–729, 1991.

[138] Oded Goldreich, Noam Nisan, and Avi Wigderson. On Yao's XOR lemma. Technical Report TR95–050, *Electronic Colloquium on Computational Complexity*, March 1995. www.eccc.uni-trier.de/.

[139] Oded Goldreich and Guy N. Rothblum. Simple doubly-efficient interactive proof systems for locally-characterizable sets. In *ITCS*, volume 94 of *LIPIcs*, pages 18:1–18:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[140] Oded Goldreich and Guy N. Rothblum. Constant-round interactive proof systems for AC0[2] and NC1. In *Computational Complexity and Property Testing*, volume 12050 of *Lecture Notes in Computer Science*, pages 326–351. Springer, 2020.

[141] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *40th ACM Symp. on the Theory of Computing (STOC)*, pages 113–122, 2008.

[142] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[143] Louis Golowich and Salil P. Vadhan. Pseudorandomness of expander random walks for symmetric functions and permutation branching programs. In Shachar Lovett, editor, *37th Computational Complexity Conference, CCC 2022, Philadelphia, PA, USA, July 20-23, 2022*, volume 234 of *LIPIcs*, pages 27:1–27:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[144] Parikshit Gopalan, Raghu Meka, Omer Reingold, Luca Trevisan, and Salil Vadhan. Better pseudorandom generators from milder pseudorandom restrictions. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, 2012.

[145] Raymond Greenlaw, H. James Hoover, and Walter Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. 02 2001.

[146] D. Yu Grigoriev. Using the notions of seperability and independence for proving the lower bounds on the circuit complexity. *Notes of the Leningrad branch of the Steklov Mathematical Institute, Nauka*, 1976.

[147] Dima Grigoriev and Alexander A. Razborov. Exponential lower bounds for depth 3 arithmetic circuits in algebras of functions over finite fields. *Appl. Algebra Eng. Commun. Comput.*, 10(6):465–487, 2000.

[148] Aryeh Grinberg, Ronen Shaltiel, and Emanuele Viola. Indistinguishability by adaptive procedures with advice, and lower bounds on hardness amplification proofs. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, 2018.

[149] Mikhail Gromov. Filling Riemannian manifolds. *Journal of Differential Geometry*, 18(1):1–147, 1983.

[150] Misha Gromov and Larry Guth. Generalizations of the kolmogorov–barzdin embedding estimates. *Duke Mathematical Journal*, 161(13):2549–2603, 2012.

[151] Ankit Gupta, Pritish Kamath, Neeraj Kayal, and Ramprasad Saptharishi. Arithmetic circuits: A chasm at depth 3. *SIAM J. Comput.*, 45(3):1064–1079, 2016.

[152] Yuri Gurevich. Unconstrained church-turing thesis cannot possibly be true. *Bull. EATCS*, 127, 2019.

[153] Yuri Gurevich and Saharon Shelah. Nearly linear time. In *Logic at Botik, Symposium on Logical Foundations of Computer Science*, pages 108–118, 1989.

[154] Dan Gutfreund and Emanuele Viola. Fooling parity tests with parity gates. In *8thWorkshop on Randomization and Computation (RANDOM)*, pages 381–392. Springer, 2004.

[155] Torben Hagerup. Fast parallel generation of random permutations. In *18th Coll. on Automata, Languages and Programming (ICALP)*, pages 405–416. Springer, 1991.

[156] Iftach Haitner, Omer Reingold, and Salil P. Vadhan. Efficiency improvements in constructing pseudorandom generators from one-way functions. In *42nd ACM ACM Symp. on the Theory of Computing (STOC)*, pages 437–446, 2010.

[157] András Hajnal, Wolfgang Maass, Pavel Pudlák, Mário Szegedy, and György Turán. Threshold circuits of bounded depth. *J. of Computer and System Sciences*, 46(2):129–154, 1993.

[158] Joseph Y. Halpern, Michael C. Loui, Albert R. Meyer, and Daniel Weise. On time versus space III. *Math. Syst. Theory*, 19(1):13–28, 1986.

[159] Yassine Hamoudi. Simultaneous multiparty communication protocols for composed functions. In *MFCS*, volume 117 of *LIPIcs*, pages 14:1–14:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[160] T. Hartman and R. Raz. On the distribution of the number of roots of polynomials and explicit weak designs. *Random Structures & Algorithms*, 23(3):235–263, 2003.

[161] Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[162] David Harvey and Joris van der Hoeven. Integer multiplication in time $O(n\log n)$. *Annals of Mathematics*, 193(2):563 – 617, 2021.

[163] Johan Håstad. *Computational limitations of small-depth circuits*. MIT Press, 1987.

[164] Johan Håstad. The shrinkage exponent of de morgan formulas is 2. *SIAM J. Comput.*, 27(1):48–64, 1998.

[165] Johan Håstad. On the correlation of parity and small-depth circuits. *SIAM J. on Computing*, 43(5):1699–1708, 2014.

[166] Johan Håstad and Mikael Goldmann. On the power of small-depth threshold circuits. *Computational Complexity*, 1(2):113–129, 1991.

[167] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM J. on Computing*, 28(4):1364–1396, 1999.

[168] John Hastad. Almost optimal lower bounds for small depth circuits. *Adv. Comput. Res.*, 5:143–170, 1989.

[169] Pooya Hatami and William Hoza. Theory of unconditional pseudorandom generators. *Electron. Colloquium Comput. Complex.*, TR23-019, 2023.

[170] Thomas P. Hayes. Separating the k-party communication complexity hierarchy: an application of the zarankiewicz problem. *Discret. Math. Theor. Comput. Sci.*, 13(4):15–22, 2011.

[171] Songhua He. A note on a hierarchy theorem for promise-bptime. *Electron. Colloquium Comput. Complex.*, TR25-004, 2025.

[172] Alexander Healy, Salil P. Vadhan, and Emanuele Viola. Using nondeterminism to amplify hardness. *SIAM J. on Computing*, 35(4):903–931, 2006.

[173] Harald Andrés Helfgott. Graph isomorphism, quasipolynomial time. *Astérisque*, 407:135–182, 2019.

[174] F. C. Hennie. Crossing sequences and off-line turing machine computations. In *Symposium on Switching Circuit Theory and Logical Design (SWCT) (FOCS)*, pages 168–172, 1965.

[175] F. C. Hennie. One-tape, off-line turing machine computations. *Information and Control*, 8(6):553–578, 1965.

[176] Fred Hennie and Richard Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[177] Fred Hennie and Richard Stearns. Two-tape simulation of multitape turing machines. *J. of the ACM*, 13:533–546, October 1966.

[178] William Hesse, Eric Allender, and David A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *J. of Computer and System Sciences*, 65(4):695–716, 2002. Special issue on complexity, 2001 (Chicago, IL).

[179] Thomas Holenstein. Key agreement from weak bit agreement. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 664–673, New York, NY, USA, 2005. Association for Computing Machinery.

[180] Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bull. Amer. Math. Soc. (N.S.)*, 43(4):439–561 (electronic), 2006.

[181] John E. Hopcroft and Jeffrey D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc., 1969.

[182] William M. Hoza. Better pseudodistributions and derandomization for space-bounded computation. In Mary Wootters and Laura Sanità, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2021, University of Washington, Seattle, Washington, USA (Virtual Conference), August 16-18, 2021*, volume 207 of *LIPIcs*, pages 28:1–28:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[183] Laurent Hyafil. On the parallel evaluation of multivariate polynomials. *SIAM J.*

Comput., 8(2):120–123, 1979.

[184] John T. Gill III. Computational complexity of probabilistic turing machines. In *STOC*, pages 91–95. ACM, 1974.

[185] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, 1988.

[186] Neil Immerman. *Descriptive Complexity*. Graduate Texts in Computer Science. Springer, 1999.

[187] Russell Impagliazzo. Hard-core distributions for somewhat hard problems. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 538–545, 1995.

[188] Russell Impagliazzo, Noam Nisan, and Avi Wigderson. Pseudorandomness for network algorithms. In *26th ACM Symp. on the Theory of Computing (STOC)*, pages 356–364, 1994.

[189] Russell Impagliazzo and Ramamohan Paturi. The complexity of k-sat. In *IEEE Conf. on Computational Complexity (CCC)*, pages 237–, 1999.

[190] Russell Impagliazzo, Ramamohan Paturi, and Michael E. Saks. Size-depth tradeoffs for threshold circuits. *SIAM J. Comput.*, 26(3):693–707, 1997.

[191] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Computer & Systems Sciences*, 63(4):512–530, Dec 2001.

[192] Russell Impagliazzo and Avi Wigderson. $P = BPP$ if E requires exponential circuits: Derandomizing the XOR lemma. In *29th ACM Symp. on the Theory of Computing (STOC)*, pages 220–229. ACM, 1997.

[193] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography with constant computational overhead. In *40th ACM Symp. on the Theory of Computing (STOC)*, pages 433–442, 2008.

[194] Peter Ivanov, Liam Pavlovic, and Emanuele Viola. On correlation bounds against polynomials. In *Conf. on Computational Complexity (CCC)*, 2023.

[195] Kazuo Iwama and Hiroki Morizumi. An explicit lower bound of $5n - o(n)$ for boolean circuits. In *Symp. on Math. Foundations of Computer Science (MFCS)*, pages 353–364, 2002.

[196] Michael Jaber, Yang P. Liu, Shachar Lovett, Anthony Ostuni, and Mehtaab Sawhney. Quasipolynomial bounds for the corners theorem. *CoRR*, abs/2504.07006, 2025.

[197] Hamid Jahanjou, Eric Miles, and Emanuele Viola. Succinct and explicit circuits for sorting and connectivity. Available at http://www.ccs.neu.edu/home/viola/, 2014.

[198] Hamid Jahanjou, Eric Miles, and Emanuele Viola. Local reductions. *Information and Computation*, 261(2), 2018. Available at http://www.ccs.neu.edu/home/viola/.

[199] Hermann Jung. Depth efficient transformations of arithmetic into boolean circuits. In Lothar Budach, editor, *Fundamentals of Computation Theory, FCT '85, Cottbus, GDR, September 9-13, 1985*, volume 199 of *Lecture Notes in Computer Science*, pages 167–174. Springer, 1985.

[200] Franz Kafka. Aphorisms.

[201] Nabil Kahale. Eigenvalues and expansion of regular graphs. *J. of the ACM*, 42(5):1091–

1106, 1995.

[202] Bala Kalyanasundaram and Georg Schnitger. The probabilistic communication complexity of set intersection. *SIAM J. Discrete Math.*, 5(4):545–557, 1992.

[203] Ravi Kannan, H. Venkateswaran, V. Vinay, and Andrew Chi-Chih Yao. A circuit-based proof of toda's theorem. *Inf. Comput.*, 104(2):271–276, 1993.

[204] A. A. Karatsuba. The complexity of computations. *Trudy Mat. Inst. Steklov.*, 211(Optim. Upr. i Differ. Uravn.):186–202, 1995.

[205] Richard Karp, Nicholas Pippenger, and Michael Sipser. A time-randomness tradeoff. In *AMS Conference on Probabilistic Computational Complexity*, 1985.

[206] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[207] Richard M. Karp and Richard J. Lipton. Turing machines that take advice. *L'Enseignement Mathématique. Revue Internationale. IIe Série*, 28(3-4):191–209, 1982.

[208] Kiran S. Kedlaya and Christopher Umans. Fast polynomial factorization and modular composition. *SIAM J. on Computing*, 40(6):1767–1802, 2011.

[209] Zander Kelley, Shachar Lovett, and Raghu Meka. Explicit separations between randomized and deterministic number-on-forehead communication. *CoRR*, abs/2308.12451, 2023.

[210] Leonid G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20(1):191–194, 1979.

[211] V. M. Khrapchenko. A method of obtaining lower bounds for the complexity of π-schemes. *Mathematical Notes of the Academy of Sciences of the USSR*, 10:474–479, 1972.

[212] Adam Klivans and Rocco A. Servedio. Boosting and hard-core sets. *Machine Learning*, 53(3):217–238, 2003.

[213] Adam R. Klivans. On the derandomization of constant depth circuits. In *Workshop on Randomization and Computation (RANDOM)*. Springer, 2001.

[214] Adam R. Klivans and Dieter van Melkebeek. Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses. In *ACM Symposium on Theory of Computing (Atlanta, GA, 1999)*, pages 659–667. ACM, New York, 1999.

[215] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, 3rd edition, 1998.

[216] Kojiro Kobayashi. On the structure of one-tape nondeterministic turing machine time hierarchy. *Theor. Comput. Sci.*, 40:175–193, 1985.

[217] Pascal Koiran. Valiant's model and the cost of computing integers. *Comput. Complex.*, 13(3-4):131–146, 2005.

[218] A. N. Kolmogorov and Ya. M. Barzdin. On the realization of nets in 3-dimensional space. *Probl. Cybernet.*, 8:261–268, 1967. See also Selected Works of A.N. Kolmogorov, Vol. 3, pp. 194–202 (remark on p. 245), Kluwer Academic Publishers, 1993.

[219] Swastik Kopparty and Srikanth Srinivasan. Certifying polynomials for $\mathrm{AC}^0[\oplus]$ circuits,

with applications to lower bounds and circuit compression. *Theory of Computing*, 14(1):1–24, 2018.

[220] Kenneth Krohn, W. D. Maurer, and John Rhodes. Realizing complex Boolean functions with simple groups. *Information and Control*, 9:190–195, 1966.

[221] S.-Y. Kuroda. Classes of languages and linear-bounded automata. *Inf. Control.*, 7(2):207–223, 1964.

[222] Eyal Kushilevitz and Noam Nisan. *Communication complexity*. Cambridge University Press, 1997.

[223] Gabriel Lam. Note sur la limite du nombre des divisions dans la détermination d'un plus grand commun diviseur. *Comptes Rendus de l'Académie des Sciences de Paris*, 19:867–870, 1844.

[224] Klaus-Jörn Lange, Birgit Jenner, and Bernd Kirsig. The logarithmic alternation hierarchiy collapses: A sigmaˆc_2 = A piˆc_2. In *ICALP*, volume 267 of *Lecture Notes in Computer Science*, pages 531–541. Springer, 1987.

[225] Kasper Green Larsen. The cell probe complexity of dynamic range counting. In *ACM Symp. on the Theory of Computing (STOC)*, pages 85–94, 2012.

[226] Kasper Green Larsen. Higher cell probe lower bounds for evaluating polynomials. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 293–301, 2012.

[227] Kasper Green Larsen, Omri Weinstein, and Huacheng Yu. Crossing the logarithmic barrier for dynamic boolean data structure lower bounds. *SIAM J. Comput.*, 49(5), 2020.

[228] Leonid A. Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

[229] Jiatu Li and Tianqi Yang. $3.1n$ - $o(n)$ circuit lower bounds for explicit functions. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 1180–1193. ACM, 2022.

[230] Rudolf Lidl and Harald Niederreiter. *Finite fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, second edition, 1997.

[231] Nutan Limaye, Srikanth Srinivasan, and Sébastien Tavenas. Superpolynomial lower bounds against low-depth algebraic circuits. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 804–814. IEEE, 2021.

[232] Nutan Limaye, Srikanth Srinivasan, and Sébastien Tavenas. Guest column: Lower bounds against constant-depth algebraic circuits. *SIGACT News*, 53(2):40–62, 2022.

[233] Richard Lipton. New directions in testing. In *Proceedings of DIMACS Workshop on Distributed Computing and Cryptography*, volume 2, pages 191–202. ACM/AMS, 1991.

[234] Richard J. Lipton. Straight-line complexity and integer factorization. In *International Algorithmic Number Theory Symposium*, pages 71–79, 1994. Cited at 124.

[235] Alex Lombardi and Vinod Vaikuntanathan. Limits on the locality of pseudorandom generators and applications to indistinguishability obfuscation. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography - 15th International Conference, TCC*

2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I, volume 10677 of *Lecture Notes in Computer Science*, pages 119–137. Springer, 2017.

[236] Shachar Lovett. Unconditional pseudorandom generators for low degree polynomials. In *40th ACM Symp. on the Theory of Computing (STOC)*, pages 557–562, 2008.

[237] Chi-Jen Lu, Shi-Chun Tsai, and Hsin-Lung Wu. Improved hardness amplification in NP. *Theor. Comput. Sci.*, 370(1-3):293–298, 2007.

[238] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *J. of the ACM*, 39(4):859–868, October 1992.

[239] O. B. Lupanov. A method of circuit synthesis. *Izv. VUZ Radiofiz.*, 1:120–140, 1958.

[240] Wolfgang Maass and Amir Schorr. Speed-up of Turing machines with one work tape and a two-way input tape. *SIAM J. on Computing*, 16(1):195–202, 1987.

[241] Yishay Mansour, Noam Nisan, and Prasoon Tiwari. The computational complexity of universal hashing. *Theoretical Computer Science*, 107:121–133, 1993.

[242] Xinyu Mao, Guangxu Yang, and Jiapeng Zhang. Gadgetless lifting beats round elimination: Improved lower bounds for pointer chasing. In *ITCS*, volume 325 of *LIPIcs*, pages 75:1–75:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[243] Russell A. Martin and Dana Randall. Sampling adsorbing staircase walks using a new markov chain decomposition method. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, Redondo Beach, California, USA, November 12-14, 2000*, pages 492–502. IEEE Computer Society, 2000.

[244] Yossi Matias and Uzi Vishkin. Converting high probability into nearly-constant time-with applications to parallel hashing. In *23rd ACM Symp. on the Theory of Computing (STOC)*, pages 307–316, 1991.

[245] W. D. Maurer and John L. Rhodes. A property of finite simple non-abelian groups. *Pacific Journal of Mathematics*, 16(2):491–495, 1965.

[246] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[247] Pierre McKenzie and Stephen A. Cook. The parallel complexity of abelian permutation group problems. *SIAM J. Comput.*, 16(5):880–909, 1987.

[248] Milena Mihail. Conductance and convergence of markov chains-a combinatorial treatment of expanders. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 526–531. IEEE Computer Society, 1989.

[249] Eric Miles and Emanuele Viola. On the complexity of constructing pseudorandom functions (especially when they don't exist). *J. of Cryptology*, pages 1–24, 2013.

[250] Eric Miles and Emanuele Viola. Substitution-permutation networks, pseudorandom functions, and natural proofs. *J. of the ACM*, 62(6), 2015.

[251] Peter Bro Miltersen. Lower bounds for union-split-find related problems on random access machines. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing (STOC)*, pages 625–634. ACM, 1994.

[252] Peter Bro Miltersen. On the cell probe complexity of polynomial evaluation. *Theor. Comput. Sci.*, 143(1):167–174, 1995.

[253] Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. *J. of Computer and System Sciences*, 57(1):37 – 49, 1998.

[254] Marvin Minsky and Seymour Papert. *Perceptrons*. MIT Press, Cambridge, MA, 1969.

[255] David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *J. of Computer and System Sciences*, 38(1):150–164, 1989.

[256] David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1. *J. of Computer and System Sciences*, 41(3):274–306, 1990.

[257] Cristopher Moore and Stephan Mertens. *The Nature of Computation*. Oxford University Press, 2011.

[258] Jacques Morgenstern. How to compute fast a function and all its derivatives: a variation on the theorem of baur-strassen. *SIGACT News*, 16(4):60–62, 1985.

[259] Wolfgang Mulzer. Five proofs of chernoff's bound with applications. *Bull. EATCS*, 124, 2018.

[260] Cody Murray and R. Ryan Williams. Circuit lower bounds for nondeterministic quasi-polytime: an easy witness lemma for NP and NQP. In *STOC*, pages 890–901. ACM, 2018.

[261] J. Naor and M. Naor. Small-bias probability spaces: efficient constructions and applications. In *22nd ACM Symp. on the Theory of Computing (STOC)*, pages 213–223. ACM, 1990.

[262] Moni Naor and Omer Reingold. Synthesizers and their application to the parallel construction of pseudo-random functions. *J. Comput. Syst. Sci.*, 58(2):336–375, 1999.

[263] E. I. Nechiporuk. A boolean function. *Soviet Mathematics-Doklady*, 169(4):765–766, 1966.

[264] Valery A. Nepomnjaščiĭ. Rudimentary predicates and Turing calculations. *Soviet Mathematics-Doklady*, 11(6):1462–1465, 1970.

[265] NEU. From RAM to SAT. Available at http://www.ccs.neu.edu/home/viola/, 2012.

[266] Ilan Newman. Private vs. common random bits in communication complexity. *Information Processing Letters*, 39(2):67–71, 1991.

[267] Noam Nisan. Pseudorandom bits for constant depth circuits. *Combinatorica. An Journal on Combinatorics and the Theory of Computing*, 11(1):63–70, 1991.

[268] Noam Nisan. RL $\subseteq$ SC. In S. Rao Kosaraju, Mike Fellows, Avi Wigderson, and John A. Ellis, editors, *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 619–623. ACM, 1992.

[269] Noam Nisan. The communication complexity of threshold gates. In *Combinatorics, Paul Erdős is Eighty, number 1 in Bolyai Society Mathematical Studies*, pages 301–315, 1993.

[270] Noam Nisan and Avi Wigderson. Rounds in communication complexity revisited. *SIAM J. on Computing*, 22(1):211–219, 1993.

[271] Noam Nisan and Avi Wigderson. Hardness vs randomness. *J. of Computer and System*

Sciences, 49(2):149–167, 1994.

[272] Noam Nisan and Avi Wigderson. Lower bounds on arithmetic circuits via partial derivatives. *Comput. Complexity*, 6(3):217–234, 1996/97.

[273] Ryan O'Donnell. Hardness amplification within NP. *J. of Computer and System Sciences*, 69(1):68–94, August 2004.

[274] Ryan O'Donnell. *Analysis of Boolean Functions*. Cambridge University Press, 2014.

[275] Ryan O'Donnell and Yu Zhao. On closeness to k-wise uniformity. In Eric Blais, Klaus Jansen, José D. P. Rolim, and David Steurer, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2018, August 20-22, 2018 - Princeton, NJ, USA*, volume 116 of *LIPIcs*, pages 54:1–54:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[276] Christos H. Papadimitriou and Michael Sipser. Communication complexity. *J. of Computer and System Sciences*, 28(2):260–269, 1984.

[277] Christos H. Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. *J. Comput. Syst. Sci.*, 43(3):425–440, 1991.

[278] Christos H. Papadimitriou and Stathis Zachos. Two remarks on the power of counting. In *Theoretical Computer Science*, volume 145 of *Lecture Notes in Computer Science*, pages 269–276. Springer, 1983.

[279] Seymour Papert. One AI or Many? *Daedalus*, 117, 1988.

[280] Mihai Pătraşcu. Succincter. In *49th IEEE Symp. on Foundations of Computer Science (FOCS)*. IEEE, 2008.

[281] Mihai Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *ACM Symp. on the Theory of Computing (STOC)*, pages 603–610, 2010.

[282] Wolfgang J. Paul, Nicholas Pippenger, Endre Szemerédi, and William T. Trotter. On determinism versus non-determinism and related problems (preliminary version). In *IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 429–438, 1983.

[283] M. S. Pinsker. On the complexity of a concentrator. *Problems of Information Transmission*, 9(4):292–297, 1973.

[284] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. of the ACM*, 26(2):361–381, 1979.

[285] Pavel Pudlák, Vojtěch Rödl, and Jiří Sgall. Boolean circuits, tensor ranks, and communication complexity. *SIAM J. on Computing*, 26(3):605–633, 1997.

[286] C. Radhakrishna Rao. Factorial experiments derivable from combinatorial arrangements of arrays. *Suppl. J. Roy. Statist. Soc.*, 9:128–139, 1947.

[287] Anup Rao and Amir Yehudayoff. *Communication complexity*. 2019.

[288] Ran Raz. The BNS-Chung criterion for multi-party communication complexity. *Computational Complexity*, 9(2):113–122, 2000.

[289] Alexander Razborov. Lower bounds on the dimension of schemes of bounded depth in a complete basis containing the logical addition function. *Akademiya Nauk SSSR. Matematicheskie Zametki*, 41(4):598–607, 1987. English translation in Mathematical Notes of the Academy of Sci. of the USSR, 41(4):333-338, 1987.

[290] Alexander Razborov and Steven Rudich. Natural proofs. *J. of Computer and System*

Sciences, 55(1):24–35, August 1997.

[291] Alexander A. Razborov. On the distributional complexity of disjointness. *Theor. Comput. Sci.*, 106(2):385–390, 1992.

[292] Omer Reingold. Undirected connectivity in log-space. *J. of the ACM*, 55(4), 2008.

[293] Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. *SIAM J. Comput.*, 50(3), 2021.

[294] Omer Reingold, Salil Vadhan, and Avi Wigderson. Entropy waves, the zig-zag graph product, and new constant-degree expanders. *Ann. of Math. (2)*, 155(1):157–187, 2002.

[295] J. M. Robson. An O(T log T) reduction from RAM computations to satisfiability. *Theoretical Computer Science*, 82(1):141–149, 1991.

[296] Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 1962.

[297] Gian-Carlo Rota. From cardinals to chaos. In Nigel G. Cooper, editor, *From Cardinals to Chaos*, page 26. Cambridge University Press, Cambridge, 1989.

[298] Eyal Rozenman and Salil P. Vadhan. Derandomized squaring of graphs. In *Workshop on Randomization and Computation (RANDOM)*, pages 436–447, 2005.

[299] Michael E. Saks and Shiyu Zhou. BP $_{\mathrm{h}}$space(s) subseteq dspace($s^{3/2}$). *J. Comput. Syst. Sci.*, 58(2):376–403, 1999.

[300] Rahul Santhanam. On separators, segregators and time versus space. In *IEEE Conf. on Computational Complexity (CCC)*, pages 286–294, 2001.

[301] Ramprasad Saptharishi. A survey of lower bounds in arithmetic circuit complexity. GitHub repository, 2025. Commit `7dc9b26bf88d2a63a7d5e9bf7243eb00284a76a8`.

[302] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[303] M. Schaefer and C. Umans. Completeness in the polynomial-time hierarchy: a compendium. *SIGACT News, Complexity Theory Column*, 2002.

[304] Arnold Schönhage. On the power of random access machines. In *ICALP*, volume 71 of *Lecture Notes in Computer Science*, pages 520–529. Springer, 1979.

[305] Arnold Schönhage. Storage modification machines. *SIAM J. Comput.*, 9(3):490–508, 1980.

[306] Ronen Shaltiel and Christopher Umans. Simple extractors for all min-entropies and a new pseudorandom generator. *J. of the ACM*, 52(2):172–216, 2005.

[307] Yakov Shalunov. Improved bounds on the space complexity of circuit evaluation. *arXiv preprint*, 2025.

[308] Adi Shamir. Factoring numbers in o(log n) arithmetic steps. *Information Processing Letters*, 8(1):28–31, 1979.

[309] Adi Shamir. On the generation of cryptographically strong pseudo-random sequences. In *Proceedings of the 8th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 62 of *Lecture Notes in Computer Science*, pages 544–550. Springer-Verlag, 1981.

[310] Adi Shamir. IP = PSPACE. *J. of the ACM*, 39(4):869–877, October 1992.

[311] Claude Shannon. Communication theory of secrecy systems. *Bell Systems Technical*

Journal, 28(4):656–715, 1949.

[312] Claude E. Shannon. The synthesis of two-terminal switching circuits. *Bell System Tech. J.*, 28:59–98, 1949.

[313] Alexander A. Sherstov. Communication complexity theory: Thirty-five years of set disjointness. In *Symp. on Math. Foundations of Computer Science (MFCS)*, pages 24–43, 2014.

[314] Victor Shoup. New algorithms for finding irreducible polynomials over finite fields. *Mathematics of Computation*, 54(189):435–447, 1990.

[315] Amir Shpilka and Avi Wigderson. Depth-3 arithmetic circuits over fields of characteristic zero. *Comput. Complex.*, 10(1):1–27, 2001.

[316] Amir Shpilka and Amir Yehudayoff. Arithmetic circuits: A survey of recent results and open questions. *Foundations and Trends in Theoretical Computer Science*, 5(3-4):207–388, 2010.

[317] Alan Siegel. On universal classes of extremely random constant-time hash functions. *SIAM J. on Computing*, 33(3):505–543, 2004.

[318] Michael Sipser. A complexity theoretic approach to randomness. In *ACM Symp. on the Theory of Computing (STOC)*, pages 330–335, 1983.

[319] Michael Sipser. *Introduction to the theory of computation, 3rd ed.* PWS Publishing Company, 1997.

[320] Roman Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In *19th ACM Symp. on the Theory of Computing (STOC)*, pages 77–82. ACM, 1987.

[321] Roman Smolensky. On representations by low-degree polynomials. In *34th IEEE IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 130–138, 1993.

[322] P. M. Spira. On time hardware complexity tradeofis for boolean functions. In *Proceedings of the Fourth Hawaii International Symposium on System Sciences*, pages 525–527, 1971.

[323] A. Spivak. Brainteasers b 201: Strange painting. *Quantum*, page 13, 1997.

[324] Richard Edwin Stearns, Juris Hartmanis, and Philip M. Lewis II. Hierarchies of memory limited computations. In *SWCT*, pages 179–190. IEEE Computer Society, 1965.

[325] Larry Stockmeyer and Albert R. Meyer. Cosmological lower bound on the circuit complexity of a small problem in logic. *J. ACM*, 49(6):753–784, 2002.

[326] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976.

[327] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[328] Volker Strassen. Die rcrechnungskomplcxittit von elementarsvmmetrischen funktionen und von interpolationskoetfiziearten. *Numer. Math.*, 20:238–251, 1973.

[329] Volker Strassen. Vermeidung von divisionen. *Journal für die Reine und Angewandte Mathematik*, 264:182–202, 1973.

[330] Volker Strassen. Polynomials with rational coefficients which are hard to compute. *SIAM J. Comput.*, 3:128–149, 1974.

[331] B. A. Subbotovskaya. Realizations of linear functions by formulas using +, *, -. *Soviet*

Mathematics-Doklady, 2:110–112, 1961.

[332] Madhu Sudan, Luca Trevisan, and Salil Vadhan. Pseudorandom generators without the XOR lemma. *J. of Computer and System Sciences*, 62(2):236–266, 2001.

[333] Xiaoming Sun. A 3-party simultaneous protocol for SUM-INDEX. *Algorithmica*, 36(1):89–111, 2003.

[334] Róbert Szelepcsényi. The moethod of focing for nondeterministic automata. *Bull. EATCS*, 33:96–99, 1987.

[335] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[336] Sébastien Tavenas. Improved bounds for reduction to depth 4 and depth 3. *Inf. Comput.*, 240:2–11, 2015.

[337] Justin Thaler. Proofs, arguments, and zero-knowledge. *Found. Trends Priv. Secur.*, 4(2-4):117–660, 2022.

[338] Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM J. on Computing*, 20(5):865–877, 1991.

[339] B. A. Trakhtenbrot. Turing computations with logarithmic delay. *Algebra i Logika*, 3(4):33–48, 1964. In Russian; original paper on complexity-gap style results.

[340] Boris A. Trakhtenbrot. A survey of russian approaches to perebor (brute-force searches) algorithms. *IEEE Ann. Hist. Comput.*, 6(4):384–400, 1984.

[341] Sergei Treil. Linear algebra done wrong, 2016. Lecture notes.

[342] Luca Trevisan. Personal communication via salil vadhan. Email correspondence, 2006. Communicated to the author through Salil Vadhan.

[343] Luca Trevisan. The program-enumeration bottleneck in average-case complexity theory. In *CCC*, pages 88–95. IEEE Computer Society, 2010.

[344] Vladimir Trifonov. An o(logn loglogn) space algorithm for undirected st-connectivity. *SIAM J. Comput.*, 38(2):449–483, 2008.

[345] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, 1937.

[346] Christopher Umans. Pseudo-random generators for all hardnesses. *J. of Computer and System Sciences*, 67(2):419–440, 2003. Special issue on STOC2002 (Montreal, QC).

[347] Salil P. Vadhan. Pseudorandomness. *Foundations and Trends in Theoretical Computer Science*, 7(1-3):1–336, 2012.

[348] Valiant. On non-linear lower bounds in computational complexity. In *ACM Symp. on the Theory of Computing (STOC)*, pages 45–53, 1975.

[349] L. G. Valiant. Completeness classes in algebra. In *Conference Record of the Eleventh Annual ACM Symposium on Theory of Computing (Atlanta, Ga., 1979)*, pages 249–261. ACM, New York, 1979.

[350] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

[351] Leslie G. Valiant. Graph-theoretic arguments in low-level complexity. In *6th Symposium on Mathematical Foundations of Computer Science*, volume 53 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1977.

[352] Leslie G. Valiant, Sven Skyum, S. Berkowitz, and Charles Rackoff. Fast parallel computation of polynomials using few processors. *SIAM J. Comput.*, 12(4):641–644, 1983.

[353] Leslie G. Valiant and Vijay V. Vazirani. NP is as easy as detecting unique solutions. *Theor. Comput. Sci.*, 47(3):85–93, 1986.

[354] J. H. van Lint. *Introduction to coding theory*. Springer-Verlag, Berlin, third edition, 1999.

[355] Dieter van Melkebeek. A survey of lower bounds for satisfiability and related problems. *Foundations and Trends in Theoretical Computer Science*, 2(3):197–303, 2006.

[356] Dieter van Melkebeek and Ran Raz. A time lower bound for satisfiability. *Theor. Comput. Sci.*, 348(2-3):311–320, 2005.

[357] Emanuele Viola. Communication complexity of pointer chasing via the fixed-set lemma. *Theory of Computing*.

[358] Emanuele Viola. Material for the algorithms class. Course page for an algorithms class taught by Emanuele Viola, includes description and lecture links.

[359] Emanuele Viola. New lower bounds for probabilistic degree and AC0 with parity gates. *Theory of Computing*. Available at http://www.ccs.neu.edu/home/viola/.

[360] Emanuele Viola. The complexity of constructing pseudorandom generators from hard functions. *Computational Complexity*, 13(3-4):147–188, 2004.

[361] Emanuele Viola. On constructing parallel pseudorandom generators from one-way functions. In *20th IEEE Conf. on Computational Complexity (CCC)*, pages 183–197, 2005.

[362] Emanuele Viola. The complexity of hardness amplification and derandomization. *Ph.D. thesis, Harvard University*, 2006.

[363] Emanuele Viola. Gems of theoretical computer science. Lecture notes of the class taught at Northeastern University. Available at http://www.ccs.neu.edu/home/viola/classes/gems-08/index.html, 2009.

[364] Emanuele Viola. On approximate majority and probabilistic time. *Computational Complexity*, 18(3):337–375, 2009.

[365] Emanuele Viola. On the power of small-depth computation. *Foundations and Trends in Theoretical Computer Science*, 5(1):1–72, 2009.

[366] Emanuele Viola. The sum of d small-bias generators fools polynomials of degree d. *Computational Complexity*, 18(2):209–217, 2009.

[367] Emanuele Viola. Reducing 3XOR to listing triangles, an exposition. Available at http://www.ccs.neu.edu/home/viola/, 2011.

[368] Emanuele Viola. Bit-probe lower bounds for succinct data structures. *SIAM J. on Computing*, 41(6):1593–1604, 2012.

[369] Emanuele Viola. The complexity of distributions. *SIAM J. on Computing*, 41(1):191–218, 2012.

[370] Emanuele Viola. The communication complexity of addition. *Combinatorica*, pages 1–45, 2014.

[371] Emanuele Viola. Lower bounds for data structures with space close to maximum imply circuit lower bounds. *Theory of Computing*, 15:1–9, 2019. Available at

http://www.ccs.neu.edu/home/viola/.

[372] Emanuele Viola. Non-abelian combinatorics and communication complexity. *SIGACT News, Complexity Theory Column*, 50(3), 2019.

[373] Emanuele Viola, 2022. https://emanueleviola.wordpress.com/2022/09/14/myth-creation-the-switching-lemma/.

[374] Emanuele Viola. Correlation bounds against polynomials, a survey. 2022.

[375] Emanuele Viola. Pseudorandom bits and lower bounds for randomized turing machines. *Theory of Computing*, 18(10):1–12, 2022.

[376] Emanuele Viola. Simple xor lemma. 2026.

[377] Emanuele Viola and Avi Wigderson. Norms, XOR lemmas, and lower bounds for polynomials and protocols. *Theory of Computing*, 4:137–168, 2008.

[378] Emanuele Viola and Avi Wigderson. One-way multiparty communication lower bound for pointer jumping with applications. *Combinatorica*, 29(6):719–743, 2009.

[379] John von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[380] Avi Wigderson. *Mathematics and Computation: A Theory Revolutionizing Technology and Science*. Princeton University Press, 2019.

[381] Wikipedia contributors. List of PSPACE-complete problems, 2025. Accessed: 2025-10-10.

[382] Wikipedia contributors. List of np-complete problems, 2026. Accessed: 2026-02-13.

[383] Ryan Williams. *Algorithms and Resource Requirements for Fundamental Problems*. PhD thesis, Carnegie Mellon University, 2007.

[384] Ryan Williams. Non-uniform ACC circuit lower bounds. In *IEEE Conf. on Computational Complexity (CCC)*, pages 115–125, 2011.

[385] Ryan Williams. Improving exhaustive search implies superpolynomial lower bounds. *SIAM J. on Computing*, 42(3):1218–1244, 2013.

[386] Ryan Williams. Nonuniform ACC circuit lower bounds. *J. of the ACM*, 61(1):2:1–2:32, 2014.

[387] Ryan Williams. Simulating time in square-root space. *Electron. Colloquium Comput. Complex.*, TR25-017, 2025.

[388] Andrew Yao. Theory and applications of trapdoor functions. In *23rd IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 80–91. IEEE, 1982.

[389] Andrew Yao. Separating the polynomial-time hierarchy by oracles. In *26th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 1–10, 1985.

[390] Andrew Chi-Chih Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 222–227. IEEE Computer Society, 1977.

[391] Andrew Chi-Chih Yao. Some complexity questions related to distributive computing. In *11th ACM Symp. on the Theory of Computing (STOC)*, pages 209–213, 1979.

[392] Andrew Chi-Chih Yao. On ACC and threshold circuits. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 619–627, 1990.

[393] Amir Yehudayoff. Pointer chasing via triangular discrimination. *Comb. Probab. Com-*

put., 29(4):485–494, 2020.

"All that he does seems to him, it is true, extraordinarily new, but also, because of the incredible spate of new things, extraordinarily amateurish, indeed scarcely tolerable, incapable of becoming history, breaking short the chain of the generations, cutting off for the first time at its most profound source the music of the world, which before him could at least be divined. Sometimes in his arrogance he has more anxiety for the world than for himself." [200]

Index