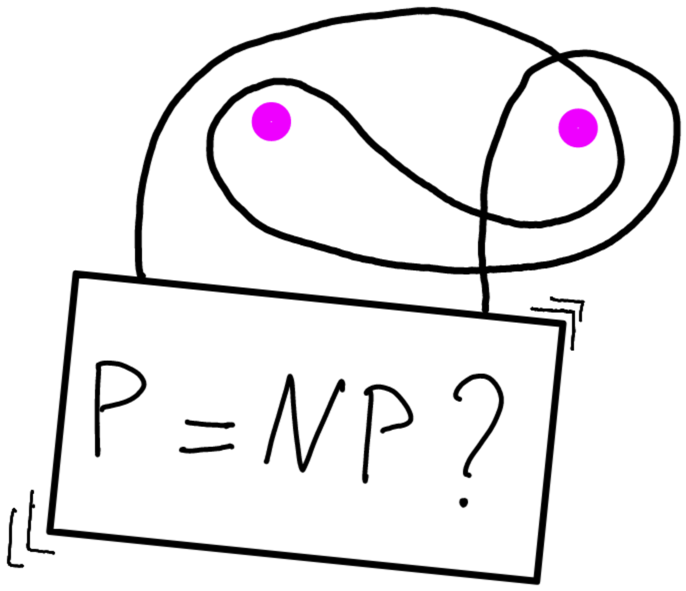
MATHEMATICS OF THE IMPOSSIBLE

THE UNCHARTED COMPLEXITY OF COMPUTATION

Compiled on April 28, 2025

Emanuele “Manu” Viola



Contents

About this book

This is a book about *computational complexity theory*. However, it is perhaps *sui generis* for various reasons:

1. The presentation is also *geared towards an algorithmic audience.* Our default model is the RAM (Chapter 1), the standard model for algorithmic research. This is in contrast with other texts which focus on tape machines. I reduce RAM 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, boolean and algebraic circuits, and communication protocols.

2. The book contains a number of recent, exciting results which are not covered in available texts, including: space-efficient simulations (Chapter 7), connections between various small-depth circuit classes (section §8.2.3), catalytic computation (section §9.6), cryptography in NC^0 (section §9.5), doubly-efficient proof systems (which have formed the basis of some deployed cryptographic systems) (section §10.5), simple constructions of expanders avoiding iterated recursion (Chapter 12), recent number-on-forehead communication protocols (section §13.4), succinct data structures (section 15.1.2), impossibility results for constant-depth algebraic circuits (Chapter 14), and natural-proof barriers that are informed by deployed cryptosystems (Chapter 16).

3. I also present several little-known but important results. This includes several simulations between models, the fact that RAMs with large registers can factor efficiently (Theorem 1.7), the result that time $o(n \log n)$ equals linear time on 1-tape machine (section 3.3.1), cosmological bounds (Theorem 3.8), the complexity of computing integers and its connections to factoring (section §14.2), and several results on pointer chasing section 13.4.1).

4. A number of well-known results are presented in a different way. Why? To demystify them and expose illuminating connections. Some of this was discussed above in 1. In addition, unlike other texts where they appear later, here I present *circuits* and *randomness* right away, and weave them through the narrative henceforth. For example, the exposition of alternation (Chapter 6) is through the lens of the circuit model. That exposition also emphasizes pseudorandom generators and thus ties with the later Chapter 11 on pseudorandomness. And in that chapter, the BIG-HIT generator is used to give a streamlined construction of pseudorandom generators from hard functions, avoiding some of the steps of previous constructions. Finally, 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.

5. This book *challenges traditional assumptions and viewpoints*. For example, I discuss my reasons for not believing P $\neq$ NP. In particular, I catalog and contextualize for the first time conjectures in complexity lower bounds which were later disproved (Chapter 17). 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 §7.3), and I expose the limitations of standard tools such as the hard-core set lemma (section 11.2.2). Finally, I include a series of historical vignettes which put key results in perspective.

I made several other choices to focus the exposition on the important points. For example I work with partial (as opposed to total) functions by default, which streamlines the presentation of several results, such as the time hierarchy (section §3.3), and eliminates NP-intermediate problems (Exercise 5.3). 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.

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. 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 theory of computation, suitable for a beginner undergraduate student, 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 6 to 10. Such a course can be supplemented with isolated topics from Chapters 11 to 16. 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 16. The pace is about one chapter a week, and I ask the students to attempt all exercises.

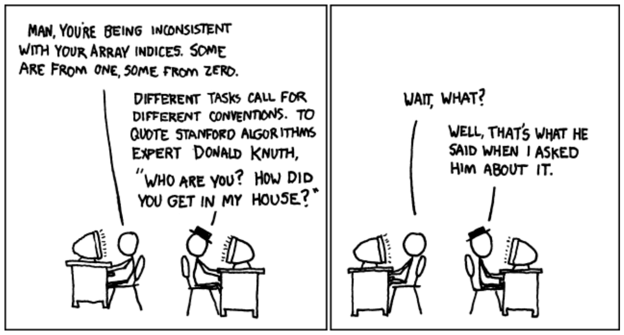
- For a special-topics course or seminar one can use Chapters 11 to 16. One possibility, which I tested, is covering all these chapters.

Chapters 1 to 10 are best read in order. Chapters 11 to 16 have fewer dependencies and can be read more or less in any order.

I hope this text will keep the reader engaged and serve as an invitation and guide to the mysterious land of complexity, until the reader stands at its frontier, gazing into the vast unknown.

Chapter -1

Conventions, choices, and caveats


MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.
DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"
WAIT, WHAT?
WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.

https://xkcd.com/163/

I write this section before the work is complete, so some of it may change.

To test your understanding of the material... this book is interspersed with mistakes, some subtle, some blatant, some not even mistakes but worrying glimpses into the author's mind. Please send all bug reports and comments to *MathematicsOfTheImpossible@gmail.com* to pin your name to this book; but hurry! The next version will be out soon.

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. This replaces, is more compact than, and is less prone to abuse than the big-Oh notation (sloppiness hides inside brackets). Let us illustrate via few examples:

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

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

The following are correct statements:

– "It is an open problem to show that some function in NP requires circuits of size cn." At the moment of this writing, one can replace this occurrence with 5. Note such a claim will remain true if someone proves a $6n$ lower bounds. One just needs to “recompile” the constants in this book.

– “$c > 1 + c$”, e.g. assign 2 to the first occurrence, 1 to the second.

– “$100n^{15} < n^c$”, for all large enough n. Assign $c = 16$.

The following are not true:

– “$c < 1/n$ for every n”. 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. There exist choices for these functions such that the claims in this book are (or are meant to be) correct. Again, each occurrence can indicate a different function. For the reader who prefers the big-Oh notation a quick an dirty fix is to replace every occurrence of c in this book with $O(1)$.

Cardinality For a set A I also write A for its cardinality $|A|$.

The alphabet of TMs. I define TMs with a fixed alphabet. This choice slightly simplifies the exposition (one parameter vs. two), while being more in line with common experience (it is more common experience to increase the length of a program than its alphabet). This choice affects the proof of Theorem 3.4; but the details don’t seem any worse.

Partial vs. total functions (a.k.a. on promise problems).

> Recall that *promise problems offer the most direct way of formulating natural computational problems. [...]* In spite of the foregoing opinions, we adopt the convention of focusing on standard decision and search problems. *[97]*

I define complexity w.r.t. *partial* functions whereas most texts consider *total* functions, i.e. we consider computing functions with arbitrary domains rather than any possible string. This is sometimes called “promise problems.” This affects many things, for example the hierarchy for BPTime (Exercise 3.4).

References and names. I have also decided to not spell out names of authors, which is increasingly awkward. Central results, such as the PCP theorem, are co-authored by five or more people. But I don’t mean to deprive the reader entirely of the thrill of name-splashing. So 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. For who got what award for what see [190].

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.

One issue is that "power" is not an adjective. However, terminology such as "power law" is commonplace, and quite apt.

Random-access machines. "Random access" also leads to strange expressions like "randomized random-access" [18]. I use "rapid access."

Reductions. Are presented as an implication. Clashing with most texts, this affects several things, for example the definition of NP-intermediate problems, see Exercise 5.3.

Exercises, problems, and questions. Exercises are interspersed within the narrative and serve as "concept check." They are not meant to be difficult or new, though some are. Problems are collected at the end and tend to be harder and more original, though some are not. Questions are meant as *research* questions, or open problems, or challenges.

Summary of some terminological and not choices

Some other sources	this book	acronym
$O(1)$, $\Omega(1)$	c	
$\|A\|$ for the size of a set A	A	
Turing machine	tape machine	TM
random-access machine	rapid-access machine	RAM
polynomial time	power time	P
superpolynomial	superpower	
mapping reduction (sometimes)	A reduces to B in P means $B \in \mathrm{P} \Rightarrow A \in \mathrm{P}$	
Extended Church-Turing thesis	Power-time computability thesis	
pairwise independent	pairwise uniform	
FP, promise-P	P	
TM with any alphabet	TM with fixed alphabet	
classes have total functions	classes have partial functions	
AC^0	AC	
TC^0	TC	
P/poly	PCkt	
$\{0,1\}$	[2]	
tree evaluation	recursive function evaluation	

The $\{0,1\}$ notation is cumbersome for people and compilers. What I really would like is use $2 = \{0,1\}$ as in $f : 2^n \to 2$, but I fear it's pushing it a little

Acknowledgments

Obviously, this work would not have been possible had I not had the privilege of interacting with many (most?) of the leading complexity theorists over the years. I am indebted to the students in my Spring 2023 Complexity Theory class, during which this material was first flushed out, for much useful feedback, and similarly to the students in my Spring 2024, Fall 2024, and Spring 2025 offerings. I am similarly indebted to the readers of my blog, where some of this material appeared first.

Many people helped in various ways, including:

Noga Alon, Ravi Boppana, Peter Bürgisser, Michal Koucký, Nutan Limaye, Oded Schwartz, Srikanth Srinivasan, Justin Thaler, Salil Vadhan

Thanks to Marco Genovesi for his rendering of the Maj image.

Unindexed mathematical notation and symbols

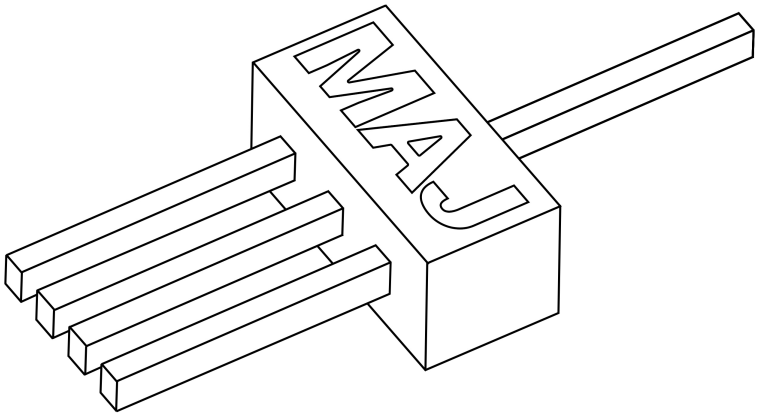
$[i..j]$	$\{i, i+1, i+2, \ldots, j\}$
$[i]$	$[0..i-1] = \{0, 1, 2, \ldots, i-1\}$
$[2]^n$	binary strings of length n
$[2]^*$	binary strings of any length
$i \mid j$	i divides j
$\mathbb{C}$	complex numbers
$\mathbb{E}$	expectation
$\mathbb{N}$	natural numbers $\{0, 1, 2, \ldots\}$
$\mathbb{P}$	probability
$\mathbb{Q}$	rational numbers (from *quotient*)
$\mathbb{R}$	real numbers
$\mathbb{Z}$	integer numbers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ (from *Zahlen*)
§	section

Abbreviations

a.k.a.	also known as
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

Chapter 0

A teaser



Consider a computer with *three* bits of memory. There's also a clock, beating $1, 2, 3, \ldots$ 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 computer can do.

First, it can compute the And function on n bits:

```
Computing And of ($x_1$,$x_2$,...,$x_n$)
For $i = 1, 2, \ldots$ 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 *palindrome* if it reads the same both ways, like *racecar*, *non*, *anna,* and so on. Similarly, example 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 $(x_1, x_2, \dots, x_n)$ is palindrome:

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

That was easy. 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.

Exercise 0.1. Prove that any function $f : [2]^n \to [2]$ can be computed on such a computer in time 2^{cn}.

So this 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?

> Convince yourself that this is impossible. Hint: If you start counting bits, you'll soon run out of memory.

If you managed to convince yourself, you are not alone.

And yet, we will see the following shocking result:

Theorem 0.1. Majority can be computed on such a computer in time n^c.

And this is not a trick tailored to majority. Many other problems, apparently much more complicated, can also be solved in the same time.

But, there's something possibly even more shocking.

Shocking situation:
It is consistent with our state of knowledge that every "textbook algorithm" can be solved in time n^c on such a computer! Nobody can disprove that. (Textbook algorithms include sorting, maxflow, dynamic programming algorithms like longest common subsequence etc., graph problems, numerical problems, etc.)

The **Shocking theorem** gives some explanation for the **Shocking situation**. It will be hard to rule out efficient programs on this model, since they are so powerful and counterintuitive. In fact, we will see later that this can be formalized. Basically, we will show that the model is so strong that it can compute functions that provably escape the reach of current mathematical proofs... if you believe certain things, like that it's hard to factor numbers. This now 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.

Proofs Above is one way in which complexity theory has contributed to the ancient concept of "proof." But the impact is much more widespread, and also affects many computer systems currently deployed.

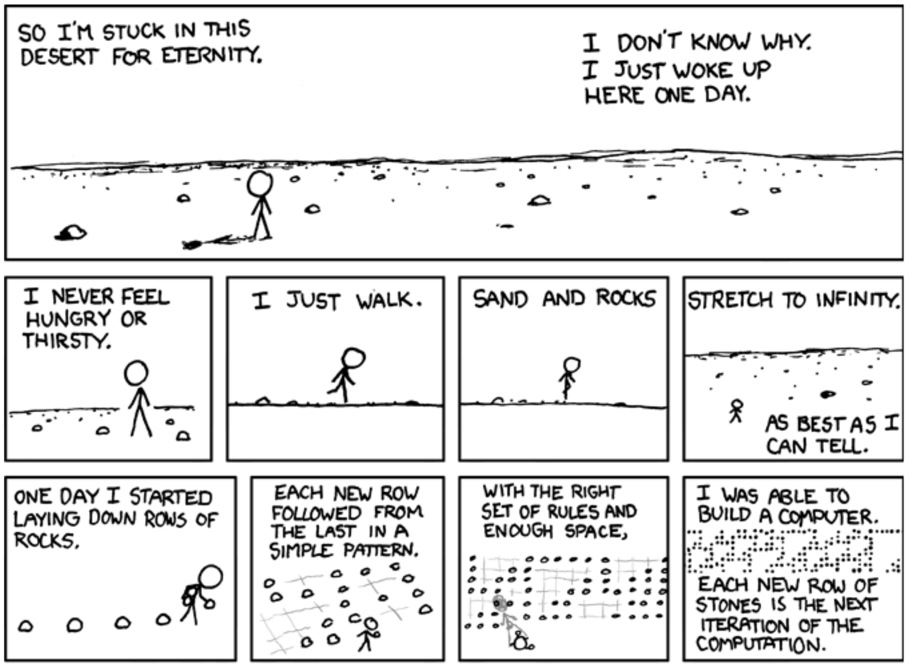
Suppose I claim I do have a program as above that computes majority. The program is fairly short, but it runs in time n^{10}. Naturally, you want to check it on an input x. But even if the input has length $|x|$ =5 bits, this would take forever. Can I convince you that the program does compute majority correctly quickly, much faster than it would take you to run it? This sounds impossible, I could be cheating in any way, you could only be sure if you checked each step of the computation. But in fact, it is possible, and your computation time would be essentially linear in $|x|$. The proof will be *interactive*, you will ask me a question, I will give a reply, and so on a few times, and all your computation time (but not mine) will be very small.

OK, it works on one input, but how can you check if it works on *every* input of length say $n = 100$? Now, this gotta be really impossible to do efficiently, after all there are 2^{100} possible inputs. Turns out we will again see how you can verify this in time power, not exponential, in n. And in fact this works not just for these simple programs, but for *any* program.

This is just a glimpse of the fascinating world of complexity we are about to enter.

Chapter 1

The alphabet of Time


SO I'M STUCK IN THIS DESERT FOR ETERNITY.
I DON'T KNOW WHY. I JUST WOKE UP HERE ONE DAY.
I NEVER FEEL HUNGRY OR THIRSTY.
I JUST WALK.
SAND AND ROCKS
STRETCH TO INFINITY.
AS BEST AS I CAN TELL.
ONE DAY I STARTED LAYING DOWN ROWS OF ROCKS.
EACH NEW ROW FOLLOWED FROM THE LAST IN A SIMPLE PATTERN.
WITH THE RIGHT SET OF RULES AND ENOUGH SPACE,
I WAS ABLE TO BUILD A COMPUTER.
EACH NEW ROW OF STONES IS THE NEXT ITERATION OF THE COMPUTATION.

https://xkcd.com/505/ (selection)

The details of the model of computation are not too important if you don't care about power differences in running times, such as the difference between solving a problem on an input of length n in time cn or cn^2. But they matter if you do.

The fundamentals features of computation are two:

- **Locality.** Computation proceeds in small, 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 problems. At one extreme, we can think of a single algorithm which applies to an

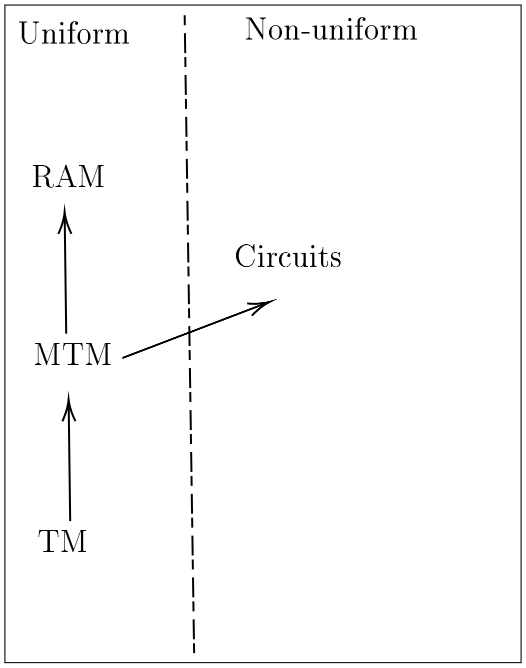


Figure 1.1: Computational models for Time. An arrow from A to B means that B can simulate A efficiently (from time t to $t \log^c t$).

infinite number of inputs. This is called *uniform* computation. 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.

Keep in mind these two principles when reading the next models.

1.1 Tape machines (TMs)

Tape machines are equipped with an infinite tape of cells with symbols from the *tape alphabet* A, and a *tape head* lying on exactly one cell. The machine is in one of several states, which you can think of as lines in a programming language. In one step the machine writes a symbol where the head points, changes state, and moves the head one cell to the right or left. Alternatively, it can stop. Such action depends only on the state of the machine and the tape symbol under the head.

We are interested in studying the *resources* required for computing. Several resources are of interest, like time and space. In this chapter we begin with time.

Definition 1.1. *A tape machine* (TM) with s states is a map (known as *transition* or *step*)

$$\sigma : \{\underline{1}, \underline{2}, \ldots, \underline{s}\} \times A \to A \times \{\text{Left}, \text{Right}, \text{Stop}\} \times \{\underline{1}, \underline{2}, \ldots, \underline{s}\},$$

where $A := \{0, 1, \#, -, \_\}$ is the *tape alphabet.* The alphabet symbol $\_$ is known as *blank*.

A *configuration* of a TM encodes its tape content, the position of the head on the tape, and the current state. It can be written as a triple $(M, i, \underline{j})$ where M maps the integers to A and specifies the tape contents, i is an integer indicating the position of the head on the tape, and $\underline{j}$ is the state of the machine.

A configuration $(\mu, i, \underline{j})$ *yields* $(\mu', i+1, \underline{j'})$ if $\sigma(\underline{j}, \mu[i]) = (a, \text{Right}, \underline{j'})$ and $\mu'[i] = a$ and $\mu' = \mu$ elsewhere, and similarly it yields $(\mu', i-1, \underline{j'})$ if $\sigma(\underline{j}, \mu[i]) = (a, \text{Left}, \underline{j'})$ and $\mu'[i] = a$ and $\mu' = \mu$ elsewhere, and finally it yields itself if $\sigma(\underline{j}, \mu[i]) = (a, \text{Stop}, \underline{j'})$.

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 $(\mu, 0, \underline{1})$ where $x = \mu[0]\mu[1] \cdots \mu[|x| - 1]$ and μ is blank elsewhere, it yields a sequence of t configurations where the last one is $(\mu, i, \underline{j})$ where $\sigma(\mu[i], \underline{j})$ has a Stop instruction, and $y = \mu[i]\mu[i+1] \cdots \mu[i + |y| - 1]$ and μ is blank elsewhere.

Describing TMs by giving the transition function quickly becomes complicated and uninformative. Instead, we give a high-level description of how the TM works. The important points to address are how the head moves, and how information is moved across the tape.

Example 1.1. On input $x \in [2]^*$ we wish to compute $x + 1$ (i.e., we think of x as an integer in binary, and increment by one). This can be accomplished by a TM with c states as follows. Move the head to the least significant bit 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 by one symbol, and repeat. If you reach the beginning of the input, shift the input by one symbol, append a 1, move the head to the beginning and stop.

The TM only does a constant number of passes over the input, so the running time is $c|x|$.

Example 1.2. On an input $x \in [2]^*$ we wish to decide if it has the the same number of zeros and ones. This can be done as follows. Do a pass on the input, and cross off one 0 and one 1 (by replacing them with tape symbol $\#$). If you didn't find any 0 or or 1, accept (that is, write 1 on the tape and stop). If only find a 0 but not a 1, or vice versa, reject.

Since every time we do a pass we cross at least two symbols, the running time is cn^2.

Exercise 1.1. Describe a TM that decides if a string $x \in [2]^*$ is palindrome, and bound its running time.

Exercise 1.2. Describe a TM that on input $x \in [2]^*$ computes $n := |x|$ in binary in time $cn \log n$.

TMs can compute any function if they have sufficiently many states:

Exercise 1.3. Prove that every function $f : [2]^n \to [2]$ can be computed by a TM in time n using 2^{n+1} states.

1.1.1 You don't need much to have it all

How powerful are tape machines? Perhaps surprisingly, they are all-powerful.

Power-time computability thesis. For any "realistic" computational model C there is $d > 0$ such that: Anything that can be computed on C in time t can also be computed on a TM in time t^d.

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 2.

However, the thesis can be proved for many standard computational models, which include all modern programming languages. The proofs aren't hard. One just tediously goes through each instruction in the target model and gives a TM implementation. We prove a representative case below (Theorem 1.8) for *rapid-access machines* (RAMs), which are close to how computers operate, and from which the jump to a programming language is short.

Given the thesis, why bother with TMs? Why not just use RAMs or a programming language as our model? In fact, we will basically do that. Our default for complexity will be RAMs. However, some of the benefits of TMs remain

- TMs are easier to define – just imagine how more complicated Definition 1.1 would be were we to use a different model. Whereas for TMs we can give a short self-contained definition, for other models we have to resort to skipping details. There is also some arbitrariness in the definition of other models. What operations exactly are allowed?
- TMs make it easier to establish certain reductions among problems. For example Theorem 5.2 is easier to prove for TMs, and the same is true for analogous statements for many other problems. The proofs for RAM would tend to proceed by first simulating a RAM by a TM or a similar device.
- Finally, TMs allow us to better pinpoint the limits of our knowledge about computation; we will see several examples of this.

In short, RAMs and programming languages are useful to carry computation, TMs to analyze it.

1.1.2 Time complexity, P, and EXP

We now define our first complexity classes. We are interested in solving a variety of computational tasks on TMs. So we make some remarks before the definition.

- We often need to compute *structured* objects, like tuples, graphs, matrices, etc. One can encode such objects in binary by using multiple bits. We will assume that such encodings are fixed and allow ourselves to speak of such structures objects. For example, we can encode a tuple $(x_1, x_2, \ldots, x_t)$ where $x_i \in [2]^*$ by repeating each bit in each x_i twice, and separate elements with 01.

- We can view machines as *computing functions*, or *solving problems,* or *deciding sets*, or *deciding languages*. These are all equivalent notions. For example, for a set A, the problem of deciding if an input x belongs to A, written $x \in A$, is equivalent to computing the boolean characteristic function f_A which outputs 1 if the input belongs to A, and 0 otherwise. We will use this terminology interchangeably. In general, "computing a function" is more appropriate terminology when the function is not boolean.

- 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 $[2]^*$ or $[2]^n$. Partial functions are a natural choice for many problems, cf. discussion in Chapter -1.

- We measure the running time of the machine in terms of the *input length,* usually denoted n. Input length can be a coarse measure: it is often natural to express the running time in terms of other parameters (for example, the time to factor a number could be better expressed in terms of the number of factors of the input, rather than its bit length). However for most of the discussion this coarse measure suffices, and we will discuss explicitly when it does not.

- We allow non-boolean outputs. However the running time is still only measured in terms of the input. (Another option which sometimes makes sense, it to bound the time in terms of the output length as well, which allows us to speak meaningfully of computing functions with very large outputs, such as exponentiation.)

- More generally, we are interested in computing not just functions but *relations*. That is, given an input x we wish to compute some y that belongs to a *set* $f(x)$. For example, the problem at hand might have more than one solution, and we just want to compute any of them.

- We are only interested in sufficiently large n, because one can always hard-wire solutions for inputs of fixed size, see Exercise 1.3. This allows us to speak of running times like $t(n) = n^2/1000$ without worrying that it is not suitable when n is small (for example, $t(10) = 100/1000 < 1$, so the TM could not even get started). This is reflected in the $n \geq c_M$ in the definition.

With this in mind, we now give the definition.

Definition 1.2. [Time complexity classes – boolean] Let $t : \mathbb{N} \to \mathbb{N}$ be a function. TM-Time(t) denotes the functions f that map bit strings x from a subset $X \subseteq [2]^*$ to a set $f(x)$ for which there exists a TM M such that, on any input $x \in X$ of length $\geq c_M$, M computes y within $t(|x|)$ steps and $y \in f(x)$.

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

We will not need to deal with relations and partial functions until later in this text.

Also, working with boolean functions, i.e., functions f with range $[2]$ slightly simplifies the exposition of a number of results we will see later. To avoid an explosion of complexity classes, we adopt the following convention.

Convention about complexity classes:
Unless specified otherwise, inclusions and separations among complexity classes refer to boolean functions. For example, an expression like $\mathrm{P} \subseteq \mathrm{NP}$ means that every boolean function in P is in NP.

As hinted before, the definition of P is robust. In the next few sections we discuss this robustness in more detail, and also introduce a number of other central computational models.

1.2 TMs with large alphabet

As our first example of robustness, we discuss TMs with arbitrary alphabet. This might seem like a detail, but in fact we are going to shortly present a cute problem in the area which will come up again and is, as far as I know, open. To set the stage, we first a relatively straightforward power-time simulation.

We define TMs *with alphabet size* a as in Definition 1.1 but with $|A|$ of size a; we will only be interested in $a \geq |A|$ so we can think of adding symbols to A in Definition 1.1. We define similarly TM-Time$(t(n))$ with alphabet size a.

Theorem 1.1. TM-Time$(t(n))$ with alphabet size $a \subseteq$ TM-Time$(c_a t(n) + c_a n^2)$.

Proof. Given a machine M_a as in the LHS, we construct machine M as in the RHS as follows. We use $c \log a \leq c_a$ tape symbols of M to encode one tape symbol of M_a. First we need to re-encode the input x. This takes time $c_a n^2$ as follows. First we move the head to the rightmost symbol of x in position $|x|$, and we shift it right of c_a positions. Then we go to the adjacent symbol in position $|x| - 1$, and shift all the contents to the right of this by c_a positions, to the right. We continue in this way.

Once this re-encoding is done, M can simulate M_a step-by-step, spending time c_a for each step of M_a. **QED**

This shows that the definition of P is robust w.r.t. different alphabet sizes. Yet the simulation is unsatisfactory due to the need of re-encode the input which gives a quadratic time blow-up.

Question 1.1. *Is the n^2 term in Theorem 1.1 necessary?*

1.2.1 The universal TM

Universal machines can simulate any other machine on any input. These machines play a critical role in some results we will see later. They also have historical significance: before them machines were tailored to specific tasks. One can think of such machines as epitomizing the victory of *software* over *hardware:* A single machine (hardware) can be programmed (software) to simulate any other machine.

Lemma 1.1. There is a TM U that on input (M, t, x) where M is a TM, t is an integer, 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. We maintain the invariant that M and t are always next to the tape head of U. After the simulation of each step of M the tape of U will contain

$$(x, M, \underline{i}, t', y)$$

where M is in state $\underline{i}$, the tape of M contains xy and the head is on the left-most symbol of y. The integer t' is the counter decreased at every step. Computing the transition of M takes time $|M|^c$. Decreasing the counter takes time $c|t|$. To move M and t next to the tape head takes $c|M||t|$ time. **QED**

1.3 Multi-tape machines (MTMs)

Definition 1.3. A k-TM is like a TM but with k tapes, where the heads on the tapes move independently. The input is placed on the first tape, and all other tapes are initialized to _. The output is on the first tape. k-TM-Time is defined analogously to TM-Time. We write MTM for multi-tape machine for some number k of tapes.

Exercise 1.4. Prove that Palindromes is in 2-TM-Time(cn). Compare this to the run-time from the the TM in Exercise 1.1.

The following result implies in particular that P is unchanged if we define it in terms of TMs or k-TMs.

Theorem 1.2. k-TM-Time$(t(n)) \subseteq$ TM-Time$(c_k t^2(n))$ for any $t(n) \geq n$ and k.

Exercise 1.5. Prove this. Recall that we defined TMs with fixed alphabet, and cf. section §1.2.

A much less obvious simulation is given by 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 1.3. [131, 211]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.

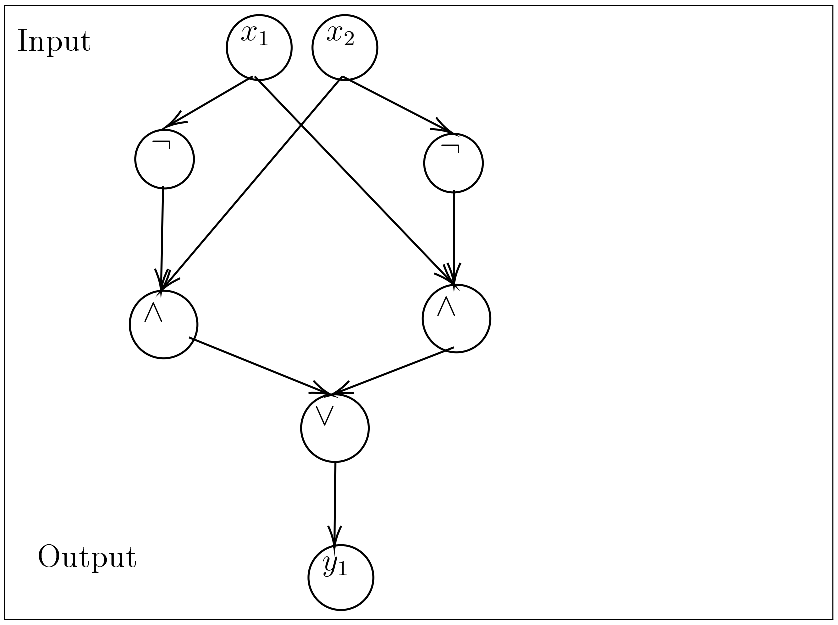


Figure 1.2: A circuit computing the Xor of two bits.

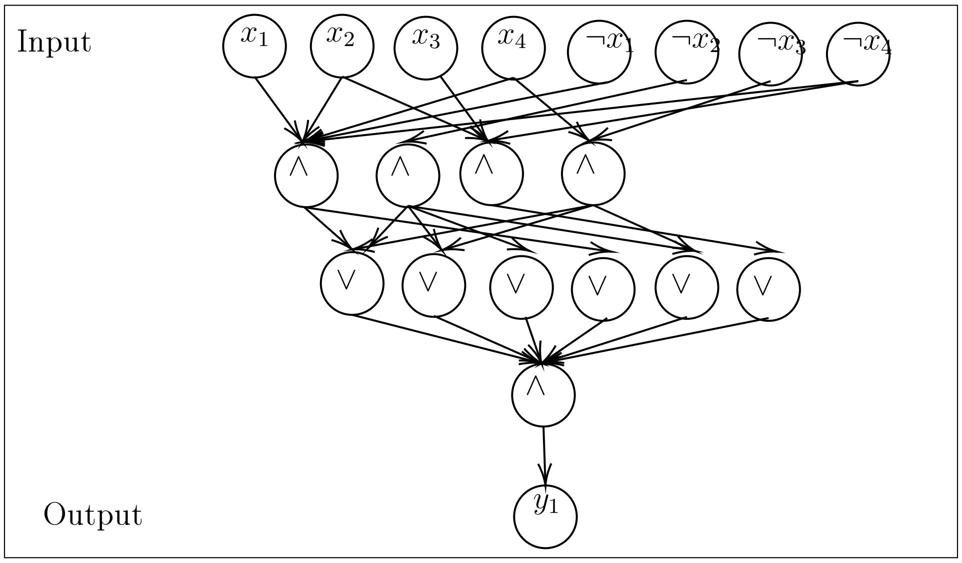


Figure 1.3: An alternating circuit.

Using this results one can prove the existence of universal MTMs similar to the universal TMs in Lemma 1.1. However, we won't need this result so we omit the proof.

1.4 Circuits

We now define circuits. It may be helpful to refer to figure 1.2 and figure 1.3.

Definition 1.4. A *circuit,* abbreviated Ckt, is a directed acyclic graph where each node is one of the following types: an input variable (fan-in 0), an output variable (fan-in 1), a negation gate $\neg$ (fan-in 1), an And gate $\wedge$ (fan-in 2), or an Or gate $\vee$ (fan-in 2). The *fan-in* of a gate is the number of edges pointing to the gate, the *fan-out* is the number of edges pointing away from the gate.

An *alternating circuit*, abbreviated AC, is a circuit with unbounded fan-in Or and And gates arranged in alternating layers (that is, the gates at a fixed distance from the input all have the same type). For each input variable x_i the circuit has both x_i and $\neg\, x_i$ as input.

A DNF (resp. CNF) is an AC whose output is Or (resp. And). The non-ouput gates are called *terms* (resp. *clauses*) .

CktGates$(g(n))$ denotes the set of function $f : [2]^* \to [2]^*$ that, for all sufficiently large n, on inputs of length n have circuits with $g(n)$ gates; input and output gates are not counted. The *Size* of a circuit is the number of gates. We also define

$$\text{PCkt} := \bigcup_d \text{CktGates}(n^d).$$

We also denote by AC the class of functions computable by AC circuits of size n^d and depth d for a constant d.

Exercise 1.6. [Pushing negation gates at the input] Show that for any circuit $C : [2]^n \to [2]$ with g gates and depth d there is a monotone circuit C' (that is, a circuit without Not gates) with $2g$ gates and depth d such that for any $x \in [2]^n : C(x_1, x_2, \dots, x_n) = C'(x_1, \neg x_1, x_2, \neg x_2 \dots, x_n, \neg x_n)$.

Often we will consider computing functions on *small inputs.* In such cases, we can often forget about details and simply appeal to the following result, which gives exponential-size circuits which are however good enough if the input is really small. In a way, the usefulness of the result goes back to the locality of computation. The result, which is a circuit analogue of Exercise 1.3, will be extensively used in this book.

Theorem 1.4. Every function $f : [2]^n \to [2]$ can be computed by
(1) circuits of size $\leq c2^n/n$, and
(2) A DNF or CNF with $\leq 2^n + 1$ gates (in particular, circuits of size $\leq n2^n$).

Often, and soon, we have to deal with functions which output many bits, but each output bit is a function on small inputs.

Definition 1.5. A function $f : [2]^n \to [2]^m$ is d-local if each output bit depends on $\le d$ input bits.

Exercise 1.7. Prove a d-local function $f : [2]^n \to [2]^m$ has circuits of size $cm2^d$.

Exercise 1.8. Prove that the Or function on n bits has circuits of size cn. Prove Item (2) in Theorem 1.4. Prove a weaker version of Item (1) in Theorem 1.4 with bound $cn2^n$.

Exercise 1.9. Prove that the sum of two n-bit integers can be computed by circuits with cn gates, and by ACs of depth c and size n^c. (Hint: For ACs, you might want to first warm up by solving the problem when one of the integers is a power of two, so that its binary representation has weight 1.)

We now show that circuits can simulates MTMs. We begin with a simple but instructive simulation of TMs which incurs a quadratic loss, then present a more interesting quasilinear simulation.

Theorem 1.5. Suppose an s-state TM computes $f : [2]^* \to [2]$ in time $t \ge n$. Then $f \in \text{CktGates}(c_s t^2(n))$. In particular

$$\text{P} \subseteq \text{PCkt}.$$

For this proof and the next it is convenient to represent a configuration of a TM in a slightly different way, as a string of symbols over the alphabet $A \times \{0, 1, \ldots, s\}$. String

$$(a_1, 0)(a_2, 0) \ldots (a_{i-1}, 0)(a_i, j)(a_{i+1}, 0) \ldots (a_m, 0)$$

with $j > 0$ indicates that (1) the tape content is $a_1 a_2 \ldots a_m$ with blanks on either side, (2) the machine is in state $\underline{j}$, and (3) the head of the machine is on the i tape symbol a_i in the string.

Locality of computation here means that one symbol in a string only depends on the symbols corresponding to the same tape cell i in the previous step and its two neighbors – three symbols total – because the head only moves in one step.

Proof of Theorem 1.5. 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 in the middle cell. Note we don't need more than t cells to the right or left because the head moves only by one cell in one step. By locality of computation, each symbol in Row $i + 1$ can be computed from 3 symbols in Row i, and hence by a c_s-local function. Repeating this for every symbol

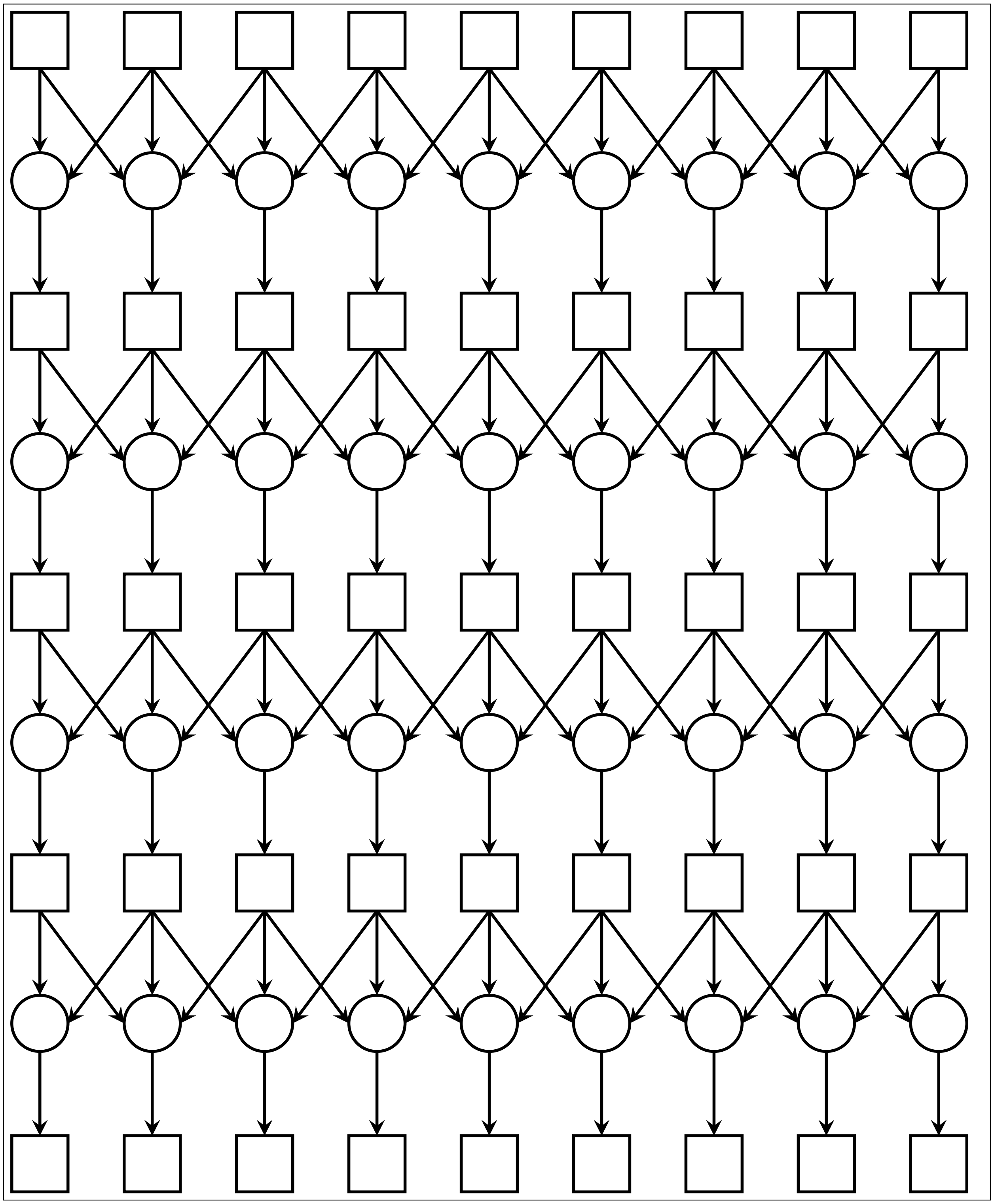


Figure 1.4: Main circuit in the proof of Theorem 1.5, for $t = 4$. Squares indicate symbols, circles indicate circuits.

we obtain that Row $i + 1$ can be computed from Row i by a c_s-local function. Hence by Exercise 1.7 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. This circuit is illustrated in figure 1.4.

There remains to output the value of the function. Had we assumed that the TM writes the output in a specific cell, we could just read it off by a circuit of size c. Without the assumption, we can have a circuit $C : A \times \{0, 1, \ldots, s\} \to [2]$ which outputs 1 on (x, y) iff $y \neq 0$ and $x = 1$ (i.e., if x is a 1 that is under the TM's head). Taking an Or such circuits applied to every entry in the last row of T concludes the proof. **QED**

The simulation in Theorem 1.5. incurs a quadratic loss. However, a better simulation exists. In fact, this applies even to k-TMs.

Theorem 1.6. Suppose an s-state k-TM computes $f : [2]^* \to [2]$ in time $t(n) \geq n$. Then $f \in \text{CktGates}(c_{s,k} t(n) \log t(n))$.

Exercise 1.10. Prove Theorem 1.6 assuming Theorem 1.3.

Next we give a direct proof that doesn't need Theorem 1.3.

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 $\text{Size}(S_t) \leq c_M t \log t$, as desired.

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. We can now program S_m as follows. First run H_m to get in which block i the head is. Use R_m to rotate the blocks by i positions so that the head is in a block closest to the middle. Run $S_{m/2}$. Now again run H_m to get j, and then R_m to move block j closest to the middle. Run $S_{m/2}$. 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**

In the other direction, TMs can simulate circuits if they have enough states. In general, allowing for the number of states to grow with the input length gives models with "hybrid uniformity."

Exercise 1.11. Suppose that $f : [2]^n \to [2]$ has circuits with s gates. Show that f can be computed by a TM with s^c states in time s^c.

1.5 Rapid-access machines (RAMs)

The main feature that's missing in all models considered so far is the ability to read and write a memory cell in one time step given the address. This feature is called *direct addressing*, and is common place in programming languages (where for example we define an array A and then we can access cell i in the array via $A[i]$). One can augment MTMs with this capability by equipping each tape with a companion *addressing tape* and a special "jump" state which causes the head on a tape to move in one step to the address on the address tape. This model is known as RAMTM (rapid-access multi-tape machines).

A RAMTM running in time ct can write down a t-bit address and jump to that location, i.e., use an exponential amount of tape. This allows us to solve in linear time problems that we don't know how to solve in linear time without this feature.

Exercise 1.12. The Element-Distinctness problem: Given m vectors of w bits, decide if two vectors are equal.

Solve Element-Distinctness in linear time $cn = ctw$ on a RAMTM.

Your solution likely needs the memory to be initialized to blank. What happens if the memory is in an unknown state when computation starts?

One can use a data structure to simulate a RAMTM running in time t by a bounded-address RAMTM that only uses addresses of $c \log t$ bits and runs in time $t \log^c t$ (see the notes). We will not be too concerned with logarithmic factors, so we could just use RAMTMs as our model, at the price of making some later statements more complicated.

But the main issue with RAMTMs is that they do not operate like common hardware and software. Processors have built-in capabilities to perform arithmetic and logical operations on small registers, of say 64 or 128 bits, and to use them to address memory. Programming languages provide similar abstractions. This allows us to separate, in algorithm design, the size of the registers from their number, yielding more modular and useful algorithm specifications. We want to think of the memory as an array μ of s *cells* of w bits and allow for typical operations of them, including addressing arithmetic and *direct addressing*: reading and writing the cell indexed by another cell. There is a variety of such models, some arbitrariness in their definition, and several issues.

How much memory. One issue that arises is how much memory the machine should have and consequently how big w should be. There are two main options here. For "typical programming," we have a fixed memory size s and time bound t in mind, for example $s = n^3$ and $t = n^2$. A good choice then is to set $w := \lceil \log_2 s \rceil$ bits. This however makes it harder to compare machines with different memory bounds. Also in some scenarios the memory size and the time bound are not fixed. This occurs for example when simulating another machine. To handle such scenarios, unless specified otherwise, we start with a memory of $s = n + c$ cells, and a cell size of $w = \lceil \log_2 s \rceil$ bits, enough to access the input. We then equip machines with the operation MAlloc which increases the memory (i.e., s) by one, and always sets $w := \lceil \log_2 s \rceil$. Note the latter operation may increase w by 1. The MAlloc operation is akin to the TM's tape head wandering into unknown cells.

How the input is given. There are also two options for how the input is given to the machine. The difference doesn't matter if you don't care about w factors in time, but it matters if you do. For many problems, like sorting, 3Sum (cf. Definition 4.5), etc., we think of the input and the output as coming in n cells of w bits. (Typically, $w = c \log n$, and one can simulate such cells with c cells with $\log n$ bits.) In this case, the RAM is computing a function $f : ([2]^w)^n \to ([2]^w)^m$ and the input to the RAM is given accordingly. This is what one often has in mind when writing programs that involve numbers. For other problems, it is natural to just give one bit of the input in each cell. That is, the RAM is computing $f : [2]^n \to [2]^m$ and bit i of the input is placed in the i input cells. We will not be concerned too much with small factors and so we pick the second choice as default for simplicity. This choice will also make it easier later to write computation in certain useful formats (cf. Exercise 6.2). But we will also explore the first choice, see 2.3.3.

Definition 1.6. A *w-bit ℓ-line rapid-access machine* (RAM) *with s cells* consists of a memory array $\mu[1..s]$ of s cells of w bits, c *registers* $r_1, r_2, \ldots$ of w bits, and a program of ℓ lines.

Each line of the program contains an instruction among the following:

- Standard arithmetical, logical, and control-flow operations, such as $r_1 = r_2 + r_3$, if $r_1 = 0$ then goto line 17, etc.
- $r_i := \mu[r_j]$, called a Read operation, which reads the r_j memory cell and copies its content into r_i,
- $\mu[r_i] := r_j$, called a Write operation, which writes r_j into memory cells r_i, memory cell and copies its content into r_i,
- MAlloc which increases s by 1 and, if $s \geq 2^w$ also increases w by 1,
- Stop.

Read and write operations out of boundary indices have no effect.

On an input $x \in [2]^n$, the RAM starts the computation with $s := n + 1$ cells of memory and $w := \lceil \log_2 s \rceil$. The input is written in cells $1..n$, while $\mu[0]$ contains the length n of the input.

The output is written starting in cell 1.

Why bounded registers? Having defined the model, we now turn back to another issue related to its definition, perhaps the most interesting mathematically. One is tempted to brush aside details and consider instead a cleaner model where we simply have *unbounded registers* with unit-cost operations. Indeed, this abstraction is useful and commonplace when writing algorithms. Such a machine can use an exponential amount of memory, but this is not problematic for, as we remarked earlier, one can use a data structure to simulate such a machine by one that uses memory close to the running time. Instead, the ability to perform arithmetic over unbounded integers raises some difficulty. As we now show, this allows us to factor integers efficiently. However factoring is not known to be in P, and many deployed cryptographic systems, in fact, rely on it not being efficiently solvable.

Definition 1.7. The *unbounded RAM*, denoted uRAM, is a RAM with $w = \infty$. (Or $w = c_n$ suffices.)

Theorem 1.7. *A uRAM can factor n-bit integers in time $\log^c n$.*

The main technical step is to compute factorials efficiently.

Lemma 1.2. *A uRAM can compute the factorial $x!$ of an n-bit integer x in time $\log^c n$.*

Note that $x!$ can take $\geq 2^n$ bits to write down, and that's precisely where unbounded registers are useful.

Armed with this lemma, the proof of Theorem 1.7 is simple enough.

Exercise 1.13. Prove Theorem 1.7 assuming Lemma 1.2. Hint: The main idea is that computing the greatest common divisor (gcd) of a and $b!$ tells us whether a has a factor $\leq b$ or not. Also, critically computing the gcd of a and b takes time linear in the bit length of $\min\{a, b\}$

There remains to compute the factorials.

Proof of Lemma 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

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

To compute the binomial we use the binomial theorem to write, for any ℓ:

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

Hence by choosing ℓ suitably the binary representation of the lhs contains the binary representation of the rhs in an interval of the bits. These bits can be extracted using division and remainder. Also the lhs can be computed efficiently by repeated squaring. **QED**

What is computation *really* about? The cell-probe RAM We didn't specify which operations are allowed on registers, and indeed there is no consensus, and processors have different capabilities. The *cell-probe RAM,* is a useful, simple generalization of the RAM model which abstracts away such details focusing solely on locality of computation and information transfer.

Definition 1.8. A *cell-probe RAM* running in time t with m-bit state and a memory S of s cells of w bits operates in t time steps. At each time step $i \le t$ the machine:

(1) Selects an index $j \le s$ to a memory cell, as a function of i and the state of the machine.

(2) Updates its state, as a function of i, the current state, and $S[j]$.

(3) Writes a value in $S[j]$, as a function of i and the current state.

All functions involved are arbitrary.

A cell RAM can simulate any of the previous models with no overhead.

1.5.1 Time

We use RAMs as our main model for time inside P.

Definition 1.9. Time$(t(n))$ is defined as TM-Time$(t(n))$ but for RAMs instead of TMs.

Theorem 1.8. Time$(t(n)) \subseteq$ TM-Time$(t^c(n))$, for any $t(n) \ge n$.

Exercise 1.14. Prove it.

What is the relationship between circuits and RAMs? If a "description" of the circuit is given, then a RAM can simulate the circuit efficiently. The other way around is not clear. It appears that circuits need a quadratic blow-up to simulate RAMs.

Exercise 1.15. Give a function $f : [2]^* \to [2]$ in Time$(c \log n)$ but which requires circuits of size $\ge cn$.

There are universal RAMs that can simulate any other RAM with only a constant-factor overhead, unlike the logarithmic-factor overhead for tape machines.

Lemma 1.3. There is a RAM U that on input (P, t, x) where P is a RAM, t is an integer, and x is an input

-Stops in time ct,

-Outputs $P(x)$ if the latter stops within t steps on input x.

Proof. Throughout the computation, U will keep track of the memory size s_P and cell-size w_P of P. These are initialized as in the initial configuration of P on input x, whereas U starts with bigger values, since its input also contains P and t. Let h be the first cell where the input x starts. Memory location i of P is mapped to $i+h$ during the simulation. When P performs an operations among registers, U simulates that with its own registers, but discards the data that does not fit into w_P bits.

After each step, U decreases the counter. The counter can be stored in t cells, one bit per cell. The total number of operations to decrease such a counter from t to 0 is $\le ct$. Alternatively, we can think of the counter as being stored in a single register at the beginning of the simulation. Then decreasing the counter is a single operation. **QED**

1.6 Padding

To develop intuition about complexity, we now discuss a general technique known as *padding.* In short, the technique shows that if you can trade resource X for Y, then you can also trade *a lot of X* for *a lot of Y*. For a metaphor, if you have a magical device that can turn one pound of sill into gold, you can also use it to turn *two* pounds of sill into gold. The contrapositive is that if you *can't* trade a lot of X for a lot of Y, then you also can't trade a little of X for a little of Y. Hence inclusions among complexity classes imply inclusions with more resources, separations imply separations with less resources.

We give an example using the classes that we have encountered so far.

Claim 1.1. Suppose that $\text{Time}(cn) \subseteq \text{TM-Time}(n^{1.5})$. Then $\text{Time}(cn^2) \subseteq \text{TM-Time}(cn^3)$.

Proof. Let $f : [2]^* \to [2]$ be a function in $\text{Time}(cn^2)$. Consider the function f' that on input x of length n equals f computed on the first $\sqrt{n}$ bits of x. Thus, inputs to f' are padded with $n - \sqrt{n}$ useless symbols.

Note that $f' \in \text{Time}(cn)$, since in linear time we can erase the last $n - \sqrt{n}$ symbols and then just run the algorithm for f which takes time quadratic in $\sqrt{n}$ which is linear in n. (If computing square roots is not an available instruction, one can show that computing $\sqrt{n}$ can be done in linear time, for example using binary search.)

By assumption, $f' \in \text{TM-Time}(n^{1.5})$.

To compute f in TM-Time cn^3 we can then do the following. Given input x of length n, pad x to an input of length n^2 in time cn^2. Then run the algorithm for f'. This will take time $\le (cn^2)^{1.5} = cn^4$. Note this requires computing the length n of the input, squaring it, and padding x accordingly. These operations are not trivial on a TM, but will take less time $o(n^3)$, so do not affect the final bound. **QED**

To further illustrate padding, consider a simulation like the one in Theorem 1.2. If we have such a simulation for say $t(n) = cn$, we can then infer via the generic padding technique a simulation for other t. A caveat is that padding requires us to compute t. For some pathological functions, this may not be possible, whereas Theorem 1.2 works for any t.

1.7 Problems

Problem 1.1. [Indexing] Describe a TM that on input $(x, i) \in [2]^n \times \{1, 2, \ldots, n\}$ outputs bit i of x in time $cn \log n$.

Problem 1.2. [Indexing] Describe a circuit with cn gates that on input $(x, i) \in [2]^n \times \{1, 2, \ldots, n\}$ outputs bit i of x.

Problem 1.3. A TM is b-block-respecting if on any input x it crosses cells boundaries that are multiples of b only during computation steps that are multiple of b. In this problem you will show that any MTM can be transformed into a b-block-respecting MTM, with only a constant overhead in time. To isolate the essence of this problem, we shall consider TMs with an extra feature. A TM with a b-clock is a TM where in addition the transition function can depend on whether or not the current computation step is divisible by b.

Let M be a k-TM running in time $t(n)$, and let $b(n)$ be a function. Give an equivalent c_k-TM with a $b(n)$-clock that is $b(n)$-block-respecting and runs in time $ct(n)$.

Hint: Triplicate each tape, and for each block have both the preceding and following block, *reversed.* Explain how the simulation is carried out and where the clock is used.

Problem 1.4. Let $f : [2]^n \to [2]$ be computable by an s-state k-TM in time t. Think of the input as m cells of w bits, so that $n = mw$. Consider circuits made or arbitrary functions which take as input $c_{s,k}$ cells and output one cell. (Each wire in this circuit carries one cell – the bits cannot be "broken up," but the result would be non-trivial even if they could.) Show that f can be computed by such circuits of size $(t/w)^c$. For example, if $t = 100n$ and $w = 0.01n$ we have circuits of a constant number of gates.

1.8 Notes

"It's all over."

The fundamental work on complexity is [93]. That work formalized computation for the first time, and discovered its self-referential ability, essentially inventing universal machines and the diagonalization technique (cf. section §3.3). Of course, [93] 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 [190] for an account of this compelling history. Still, if a fundamental work must be picked, [93] seems appropriate, for it can be considered the first work on impossibility results about general computation.

The formalization of computation in [93] is in terms of recursive functions, not unlike modern functional programming languages. As we saw, many other equivalent formalizations came about later. Tape machines were introduced in [261] and are closer to computer hardware or imperative programming languages. They also make it a little more intuitive to measure time and space in computation.

The power-time computability thesis is an efficient version of the computability thesis. For a proof of a formalization of the latter, and related discussion see [112].

The brute-force computation of functions via circuits, Theorem 1.4, goes back to [233], see also [178].

For more on the circuit model in Problem 1.4 see [118].

RAMs appear in [18] with the comment: "In some sense we are therefore merely making concrete intuitions that already pervade the literature. A related model has, indeed, been treated explicitly [...]." For simulating RATMs by bounded-address RATMs, see [197].

Theorem 1.7 is from [230]. The algorithm for the greatest common divisor is very old, see [74], but there was no bound on its efficiency until [164], which provides the bound we need.

Theorem 1.6 is towards the end of [211].

Chapter 2

Randomness

Oh how strange to see you here! Didn't you use to say that you don't fly because the probability of a bomb on a plane is too high?
Absolutely! But then I computed the probability of TWO bombs on a plane, and that's really low. So now I just carry my own.

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. So let us augment our model with randomness.

Definition 2.1. A *randomized* (or *probabilistic*) RAM, written RRAM, is a RAM equipped with the extra instruction

- $r_i :=$ Rand, which sets r_i to a uniform value, independent of all previous random choices.

For a RRAM and a sequence $R = R_1, R_2, \ldots$ we write $M(x, R)$ for the execution of M on input x where the j-th instruction $r_i :=$ Rand is replaced with $r_i := R_j$.

We refer to BPTime($t(n)$) with error $\epsilon(n)$ as the set of functions f that map bit strings x from a subset $X \subseteq [2]^*$ to a set $f(x)$ for which there exists a RRAM M such that, on any input $x \in X$ of length $\geq c_M$, M stops within $t(|x|)$ steps and $\mathbb{P}_R[M(x,R) \in f(x)] \geq 1-\epsilon(|x|)$.

If the error ϵ is not specified then it is assumed to be 1/3. Finally, we define

$$\text{BPP} := \bigcup_a \text{BPTime}(n^a).$$

Exercise 2.1. 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 return NOT PRIME, else return PRIME."

Today, one usually takes BPP, not P, for "feasible computation." The introduction of randomness in our model raises several fascinating questions. First, does randomness exists "in nature?" Second, do we need "perfect" randomness for computation? And finally, do we need randomness at all? Is P = BPP? We will explore the latter two in this chapter.

We begin our journey by investigating how robust BPP is.

2.1 Error reduction for one-sided algorithm: Repeat and you'll get luckier

TBD

2.2 Error reduction and deviation bounds for the sum of random variables

The error in the definition of BPTime is somewhat arbitrary because it can be reduced. The way you do this is natural. For boolean functions, you repeat the algorithm many times, and take a majority vote. To analyze this you need probability bounds for the deviation of the sum of random variables (corresponding to the outcomes of the algorithm) from the mean. Such deviation bounds permeate theoretical computer science, and many other fields as well.

Theorem 2.1. Let $X_1, X_2, \ldots, X_t$ be i.i.d. boolean random variables with $p := \mathbb{P}[X_i = 1]$. Then for any $0 < p \leq q < 1$ we have $\mathbb{P}[\sum_{i=1}^t X_i \geq qt] \leq 2^{-D(q|p)t}$, where

$$D(q|p) := q \log\left(\frac{q}{p}\right) + (1-q)\log\left(\frac{1-q}{1-p}\right)$$

is the *divergence.*

The proof uses the following basic facts.

Claim 2.1. 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 2.2. Prove this. Hint: Use that for any event E, $\mathbb{E}[X] = \mathbb{E}[X|E]\mathbb{P}[E] + \mathbb{E}[X|\text{not } E]\mathbb{P}[\text{not } E]$.

Claim 2.2. If X and Y are independent, real-valued random variables then $\mathbb{E}[X \cdot Y] = \mathbb{E}[X] \cdot \mathbb{E}[Y]$.

Proof of Theorem 2.1. For $z \geq 1$, to be picked later, the function $x \to z^x$ is increasing. Using this and then Claim 2.1 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 Problem 2.3, 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

Now one can get a variety of bounds by bounding divergence for different settings of parameter. We state one such bound which we use shortly.

Fact 2.1. $D(q|p) \geq c(p-q)^2$, for any $p, q \in [0,1]$.

Exercise 2.3. For $q = 1/2$ and $p = 1/2 - \epsilon$ plot both sides of Fact 2.1 as a function of ϵ. (Hint: I used https://www.desmos.com/calculator)

The proof of the tail-bound Theorem 2.1 is flexible and applies to a variety of useful settings. The most interesting extensions concern *dependent* random variables, where in general the bounds aren weaker. In the next exercise we instead several settings where the bounds in Theorem 2.1 continue to hold; note independence is dropped in the last.

Exercise 2.4. Prove that the tail bound in Theorem 2.1 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 $z^x \le 1 + x(z-1)$ and the arithmetic-mean geometric mean inequality (AM-GM) inequality: for all $a_i \ge 0$: $(\sum_{i\in[t]} a_i)/t \ge (\prod_{i\in[t]} a_i)^{1/t}$.

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 Theorem 2.1. 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 $\mathbb{E}[X_i] \le p$.

Using the tail-bound Theorem 2.1 we can prove the error reduction stated earlier.

Theorem 2.2. [Error reduction for BPP] For boolean functions, the definition of BPP (Definition 2.1) remains the same if 1/3 is replaced with $1/2 - 1/n^a$ or $1/2^{n^a}$, for any constant a.

Proof. Suppose that f is in BPP with error $p := 1/2 - 1/n^a$ and let M be the corresponding RRAM. On an input x, let us run $t := n^{2a} \cdot n^b$ times M, each time with fresh randomness, and take a majority vote. The new algorithm is thus

$$\text{Maj}(M(x,R_1), M(x,R_2), \ldots, M(x,R_t)).$$

This new algorithm makes a mistake iff at least $t/2$ runs of M make a mistake. To analyze this error probability we invoke Theorem 2.1 where $X_i := 1$ iff run i of the algorithm makes a mistake, i.e., $M(x,R_i) \ne f(x)$, and $\epsilon := 1/n^a$. By Fact 2.1 we obtain an error bound of

$$2^{-D(1/2|1/2-\epsilon)t} \le 2^{-\epsilon^2 t} \le 2^{-n^b},$$

as desired. The new algorithm still runs in power time, for fixed a and b. **QED**

Exercise 2.5. Consider an alternative definition of BPTime, denoted BPTime', which is analogous to BPTime except that the requirement that the machine always stops within $t(|x|)$ steps is relaxed to "the *expected* running time of the machine is $t(|x|)$."

Show that defining BPP with respect to BPTime or BPTime' is equivalent.

Exercise 2.6. Consider *biased* RRAMs which are like RRAMs except that 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 if we use biased RRAMs.

A large body of research has been devoted to greatly generalize Exercise 2.6 to pinpoint the imperfect sources of randomness that suffice for simulating BPP.

2.3 The power of randomness

In this section we explore various computing paradigms that are enabled by randomness. Along the way we will see extremely important concepts that will be used many times later.

2.3.1 Parity of random subset, checking $AB = C$

A fact which is as simple as it is useful is this:

Fact 2.2. [*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, if $A \in [2]^n$ is uniform, $\sum_i A_i x_i \mod 2$ is uniform in [2].

To illustrate, consider the problem of checking products of matrices over bits with addition mod 2. This is the finite field $\mathbb{F}_2$; finite fields are discussed below.

Definition 2.2. The $AB = C$ problem: Given 3 matrices A, B, C over $\mathbb{F}_2$, is $AB = C$?

Theorem 2.3. The $AB = C$ problem is in BPTime(cn).

Proof. Let $d = c\sqrt{n}$ be the dimension of the matrices. Pick U uniformly in $[2]^d$. Compute $A(BU)$ and CU in time cn, and check if they are equal. If $AB = C$ the check always passes. In case $AB \neq C$, one of the rows is different, and by Fact 2.2 the check passes with probability $\leq 1/2$. We can reduce the error as in section §2.1. **QED**

By contrast, the fastest deterministic algorithm takes time n^{1+c}.

2.3.2 Polynomial identity testing

We now discuss an important problem which is in BPP but not known to be in P. In fact, in a sense to be made precise later, this is *the* problem in BPP which is not known to be in P. To present this problem we introduce two key concepts which will be used many times: *finite fields*, and *arithmetic circuits*.

Finite fields A finite field $\mathbb{F}$ is a finite set with elements 0 and 1 that is equipped with operations $+$ and $\cdot$ that behave "in the same way" as the corresponding operations over the reals $\mathbb{R}$ or the rationals $\mathbb{Q}$, which are *infinite* fields. One example are the integers modulo a prime p. 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.

The following summarizes key facts about finite fields. The case of prime fields suffices for the main points of this section, but stating things for general finite fields actually simplifies the exposition overall (since otherwise we need to add qualifiers to the size of the field).

Fact 2.3. [Finite fields] A unique finite field of size q exists iff $q = p^t$ where p is a prime and $t \in \mathbb{N}$. This field is 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 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 2.1. 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. It is known $z^t+z^{t/2}+1$ is irreducible over $\mathbb{F}_2$ for any $t = 2\cdot 3^\ell$ for any ℓ, giving very explicit representations. For example, 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 $z^{t+1}+z^2+z^{t-1}+1 = z^{t-1}+z^{t/2+1}+z^2+z+1$.

Arithmetic circuits We now move to defining arithmetic circuits, which are a natural generalization of the circuits we encountered in section §1.4.

Definition 2.3. An *arithmetic circuit* over a field $\mathbb{F}$ is a circuit where the gates compute the operations $+$ and $\cdot$ over $\mathbb{F}$, or are constants, or are input variables. Such a circuit computes a polynomial mapping $\mathbb{F}^n \to \mathbb{F}$.

The PIT (polynomial identity testing) problem over $\mathbb{F}$: Given an arithmetic circuit C over $\mathbb{F}$ with n input variables, does $C(x) = 0$ for every $x \in \mathbb{F}^n$?

The PIT problem *over large fields* is in BPP but it is not known to be in P. The requirement that the field be large is critical, see Problem 4.2.

Theorem 2.4. [PIT over large fields in BPP] Given an arithmetic circuit C and the representation of a finite field of size $\geq c2^{|C|}$ we can solve PIT in BPP.

To prove this theorem we need the following fundamental fact.

Lemma 2.1. [Polynomial identity lemma] 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:

1. p is the zero polynomial.
2. $p(x) = 0$ for every $x \in \mathbb{F}^n$.
3. $\mathbb{P}_{x_1,x_2,\ldots,x_n \in S}[p(x) = 0] > d/|S|$.

Proof of Lemma 2.1.. 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 $\le d-1$ 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

$$p(x_1, x_2, \ldots, x_n) = \sum_{i=0}^{d} x_1^i p_i(x_2, x_3, \ldots, x_n).$$

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

$$\mathbb{P}_{x_2,\ldots,x_n \in S}[p_j(x) = 0] \le (d-j)/|S|.$$

For every choice of $x_2, x_3, \ldots, x_n$ s.t. $p_j(x) \neq 0$, the polynomial p is a non-zero polynomial $q_{x_2,x_3,\ldots,x_n}(x_1)$ 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 $\le j$.

Overall,

$$\mathbb{P}_{x_1,x_2,\ldots,x_n \in S}[p(x) = 0] \le (d-j)/|S| + j/|S| = d/|S|.$$

QED

Exercise 2.7. Show that the equivalence between 1. and 2. does not hold over small fields such as $\mathbb{F}_2$ and large d.

Proof of Theorem 2.4. A circuit C contains at most $|C|$ multiplication gates. Each multiplication gate at most squares the degree of its inputs. Hence C computes a polynomial of degree $\le 2^{|C|}$. Let S be a subset of size $c \cdot 2^{|C|}$ of $\mathbb{F}$. Assign uniform values from S independently to each variables, and evaluate the circuit. If C evaluates to 0 everywhere then obviously the output will be 0. Otherwise, by Lemma 2.1, the probability we get a 0 is $\le 2^{|C|}/c2^{|C|} \le 1/3$. **QED**

To show that the PIT problem over the *integers* is in BPP the following result is useful.

Theorem 2.5. [Prime number theorem] $\lim_{n\to\infty}(\text{Number of primes} \le n)/(n/\log_e n) = 1$.

As is often the case in computer science, we don't need the full strength of Theorem 2.5. An approximate version with $\log_e n$ replaced by $\log^c n$ suffices, and it has a considerably easier proof. (Moreover sometimes easier proofs are easier to adapt to other settings of interest in computer science.) This weak version is stated next.

Theorem 2.6. [Weak prime number theorem] The number of primes in $[t]$ is $\ge t/\log^c t$, for every $t \ge c$.

Exercise 2.8. Show that the PIT problem over the *integers* is in BPP. (Hint: Use Theorem 2.5 and that checking if a number is prime is in P.)

2.3.3 Element distinctness, and hashing

Another fundamental paradigm that is allowed by randomization is *hashing*. One of the most basic instantiations is given by *pairwise uniformity*.

Definition 2.4. A distribution H on functions mapping $S \to T$ is called *pairwise uniform* if for every $x, x' \in S$ and $y, y' \in T$ one has

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

This is saying that on every pair of inputs H is behaving as a completely 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.

Exercise 2.9. 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 $t \leq n$.

Exercise 2.10. Define the random function $H_1 : [2]^n \to [2]$ as $H_1(x) := \sum_{i \leq n} A_i x_i + B \mod 2$ where A is uniform in $[2]^n$ and B is uniform in $[2]$.

Prove that H_1 is pairwise uniform.

Explain how to use H_1 to obtain a pairwise uniform function from $[2]^n$ to $[2]^t$ for any given $t \leq n$.

To illustrate hashing in action, we return to the element distinctness problem, cf. 1.12.

Exercise 2.11. Consider RAMs with registers of w bits and s memory cells, and the problem of solving Element-Distinctness (ED) on m vectors of w bits, one vector per input cell.

1. Solve ED in time cm for any $s \geq 2^w + cm$, deterministically. (Assume memory is initialized to blank.)
2. Solve ED in time cm for any $s \geq cm^2$, using randomness. (Hint: Hash to cm^2 slots. What is the chance of a collision.)
3. Solve ED in time cm for any $s \geq cm$, using randomness. Guideline: Hash to cm slots. Count-sort the input elements using their hash value as key. Now brute-force element distinctness for elements with the same key. What is the expected running time of the last step?

The last solution is the most satisfying (and non-trivial), as we only need a linear amount of memory! It is not known how to achieve the same parameters without randomness, so this is a very basic problem for which randomness seems to buy time.

2.4 Does randomness really buy time? Is P=BPP?

We can always brute-force the random choices in exponential time. If a randomized machine uses r random bits then we can simulate it deterministically by running it on each of the 2^r choices for the bits. A RRAM machine running in time $t \geq n$ has registers of $\leq c \log t$ bits. Each Rand operation gives a uniform register, so the machine uses $\leq ct \log t$ bits. This gives the following inclusions.

Theorem 2.7. Time(t) $\subseteq$ BPTime(t) $\subseteq$ Time($c^{t \log t}$), for any function $t = t(n)$. In particular, P $\subseteq$ BPP $\subseteq$ EXP.

Proof. The first inclusion is by definition. The idea for the second was discussed before, but we need to address the detail that we don't know what t is. One way to carry through the simulation is as follows. The deterministic machine initializes a counter r to 0. For each value of r it enumerates over the 2^r choices R for the random bits, and runs the RRAM on each choice of R, keeping track of its output on each choice, and outputting the majority vote. If it ever runs out of random bits, it increases r by 1 and restarts the process.

To analyze the running time, recall we only need $r \leq ct \log t$. So the simulation runs the RRAM at most $ct \log t \cdot 2^{ct \log t} \leq 2^{ct \log t}$ times, and each run takes time ct, where this last bound takes into account the overhead for incrementing the choice of r, and redirecting the calls to Rand to R. **QED**

Now, two surprises. First, BPP $\subseteq$ EXP is the fastest deterministic simulation we can *prove* for RAMs, or even 2-TMs. On the other hand, what may come as a bigger surprise, despite the examples in section §2.3 it appears that many people *believe* that in fact P = BPP! Moreover, it appears commonly believed that the overhead to simulate randomized computation deterministically is very small. Here the mismatch between our ability and common belief is abysmal.

However, we can do better for TMs. A *randomized* TMs has two transition functions σ_0 and σ_1, where each is as in Definition 1.1. At each step, the TM uses σ_0 or σ_1 with probability 1/2 each, corresponding to tossing a coin. We can define TM-BPTime as BPTime but with randomized TMs instead of RRAMS.

Theorem 2.8. TM-BPTime(t) $\subseteq$ Time($2^{\sqrt{t} \log^c t}$), for any $t = t(n) \geq n$.

One of the exciting developments of complexity theory has been the connection between the P $=^?$ BPP question and the "shocking situation" from Chapter 0 and the "grand challenge" from Chapter 3. 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 true randomness. Quantitatively, the harder the function the less randomness we need. At one extreme, we have the following striking connection:

Theorem 2.9. 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, Exercise 11.17.

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

Theorem 2.10. BPP$\subseteq$ PCkt.

Proof. Let $f : X \subseteq [2]^* \to [2]$ be in BPP. By Theorem 2.2 we can assume that the error is $\epsilon < 2^{-n}$, and let M be the corresponding RRAM. Note

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

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 1.5. **QED**

Exercise 2.12. In this exercise you will practice the powerful technique of combining tail bounds with union bounds, which was used in the proof of Theorem 2.10, and also see a related application of Lemma 2.1 .

An *error-correcting code* with block length n, message length k, minimum distance d, over the alphabet q, written $(n, k, d)_q$ is a subset $C \subseteq [q]^n$ of size q^k s.t. for any distinct $x, y \in C$, x and y differ in at least d coordinates.

(1) Prove the existence of $(n, an, an)_2$ codes, for a constant $a > 0$ and every n. Such codes are called *good* or *a-good.*

(2) Given a prime power q, and $k \leq q$ construct an explicit $(q, k, q-k)_q$ code using Lemma 2.1. For explicitness, show that given q and $x \in [q]^k$ computing the corresponding codeword is in P.

2.5 Problems

Problem 2.1. Show that Palindromes can be solved in time $n \log^c n$ on a randomized TM. (Yes, only one tape.)

Hint: View the input as coefficients of polynomials.

Problem 2.2. Give a function $f : X \subseteq [2]^* \to [2]$ that is in BPTime(c) but not in Time($n/100$).

Problem 2.3. Derive the minimizing value of z in the proof of Theorem 2.1.

Problem 2.4. For a circuit C on n bits denote by p_C the probability $\mathbb{P}_x[C(x) = 1]$.

(1) Show how to efficiently approximate p_C. Specifically: Give a power-time randomized algorithm that on input a circuit C and $\epsilon > 0$ written in unary (for example, as a string of $1/\epsilon$ ones) outputs p s.t. $|p - p_C| \leq \epsilon$ w.p. ≥ 0.9.

(2) Show that the following decision version of (1) 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 $p_C \geq p$.

(3) What happens if in (2) you replace the assumption that $|p_C - p| \geq \epsilon$ with $|p_C - p| > 0$?

(4) Assume P = BPP (recall this only refers to boolean functions, see the convention on page 26). Show how the approximation in (1) can be computed in P.

Problem 2.5. Assume Theorem 14.8 and its notation. Assume P = BPP. Show that given a circuit C and ϵ written in unary s.t. $p_C \geq \epsilon$ we can compute $x : C(x) = 1$ in P. In particular, given a non-zero arithmetic circuit we can find a non-zero assignment.

2.6 Notes

For a computer-science friendly exposition of deviation bounds see the book [72].

Theorem 2.10 is from [5].

Theorem 2.9 is from [142].

Theorem 2.8 is from [284].

Theorem 2.3 is from [85].

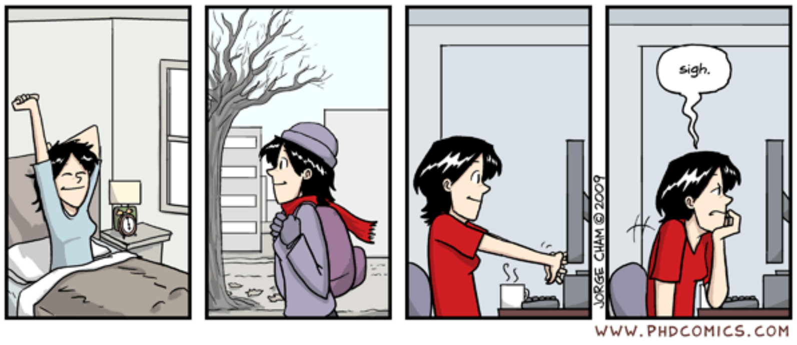
The reference for Fact 2.3 is [235]. For more on finite fields see [169], for the fields $\mathbb{F}_{2^t}$ in Example 2.1 see Theorem 1.1.28 in [267].

For a history of the Polynomial Identity Lemma 2.1 and related results, see [44]. One can get a sharper bound taking into account the individual degrees of the variables, in addition to the total degree.

Hashing originates from [55] and pervades the computer science literature. The analysis of the best solution in Exercise 2.11 relies on Lemma 1 in [83].

Chapter 3

The grand challenge



As mentioned in Chapter 0, our ability to prove impossibility results related to efficient computation appears very limited. We can now express this situation more precisely with the models we've introduced since then.

It is consistent with our knowledge that any problem in a standard algorithm textbook can be solved

1. in Time cn^2 on a TM, and
2. in Time cn on a 2-TM, and
3. by circuits of size cn.

Note that 2. implies 1. by Theorem 1.2, and many other relationships have been explored in Chapter 1.

In this chapter we begin to present several impossibility results, covering a variety of techniques which will be used later as well. As hinted above, they appear somewhat weak. However, jumping ahead, there is a flip side to all of this:

1. At times, contrary to our intuition, stronger impossibility results are actually *false*. One example appears in Chapter 0. A list will be given later.

2. Many times, the impossibility results that we can prove turn out to be, surprisingly, just "short" of proving major results. Here by "major result" I mean a result that would be phenomenal and that was in focus long before the connection was established. We will see several examples of this (section §7.3, section §8.2.3).

3. Yet other times, one can identify broad classes of proof techniques, and argue that impossibility results can't be proved with them (Chapter 16).

Given this situation, I don't subscribe to the general belief that stronger impossibility results are true and we just can't prove them.

3.1 Information bottleneck: Palindromes requires quadratic time on TMs

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 and likely matches the time in Exercise 1.1. The same bound can be shown for other functions; palindromes just happen to be convenient to obtain matching bounds.

Theorem 3.1. Palindromes $\notin$ TM-Time$(t(n))$ for any $t(n) = o(n^2)$.

More precisely, for every n and s, an s-state TM 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 mentioned above is the following.

Definition 3.1. A *crossing sequence* of a TM M on input x and boundary i, abbreviated i-CS, is the sequence of states that M is transitioning to when crossing cell boundary i (i.e., going from Cell i to $i+1$ or vice versa) during the computation on x.

Example 3.1. We think of a step of a TM as first changing state and then moving the head. We write $u\overset{i}{v}w$ if the tape content is uvw and the TM is in state i with the head on v, where $u, w \in A^*$ and $v \in A$, cf. Definition 1.1. The computation

$\overset{0}{0}$	0	0	0
#	$\overset{5}{0}$	0	0
#	0	$\overset{2}{0}$	0
#	0	x	$\overset{2}{0}$
#	0	$\overset{2}{\text{x}}$	0
#	$\overset{1}{0}$	x	0
#	0	$\overset{7}{\text{x}}$	0

has the 2-cs (marked with double vertical line; first column is cell 1) equal to $2, 1, 7$.

The idea in the proof 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 cell boundary i and bits from y to the right of i:

$$z := x_1 x_2 \cdots x_i y_{i+1} y_{i+2} \cdots y_n.$$

The proof that z is still accepted is left as an exercise.

Example 3.2. The following computation has the same 2-cs as the previous example

$\overset{0}{0}$	1	1	0
0	$\overset{4}{1}$	1	0
0	0	$\overset{2}{1}$	0
0	$\overset{1}{0}$	0	0
0	0	$\overset{7}{0}$	0

.

If $\underline{7}$ is the accept state, then the TM would also accept the "stitched" input

0010

because on that input the TM has the following "stitched" computation:

$\overset{0}{0}$	0	1	0
#	$\overset{5}{0}$	1	0
#	0	$\overset{2}{1}$	0
#	$\overset{1}{0}$	0	0
#	0	$\overset{7}{0}$	0

.

Note that the number of steps of the stitched computations needs not be the same.

Now, for many problems, stitched input z should *not* be accepted by M, and this gives a contradiction. In particular this will be be the case for palindromes. 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 3.1. 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 there are $x \neq y$ 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 $z := x0^{n/2}y^R$ which is not a palindrome but is still accepted by M, concluding the proof.

There remains to prove that the assumption of Theorem 3.1 implies the assumption in the previous paragraph. 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 $\geq c2^{n/4}/n$ inputs x for which the i-CS of M on 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 $ct\log(s)/n \leq cn$. **QED**

Exercise 3.1. For every s and n describe an s-state TM deciding palindromes in time $cn^2/\log s$ (matching Theorem 3.1).

Exercise 3.2. Let $L := \{xx : x \in [2]^*\}$. Show $L \in \text{TM-Time}(cn^2)$, and prove this is tight up to constants.

One may be tempted to think that it is not hard to prove stronger bounds for similar functions. In fact as mentioned above this has resisted all attempts!

3.2 Counting: impossibility results for non-explicit functions

Proving the *existence* of hard functions is simple: Just count. If there are more functions than efficient machines, some function is not efficiently computable. This is applicable to any model; next we state it for TMs for concreteness. Later we will state it for circuits.

Theorem 3.2. There exists a function $f : [2]^n \to [2]$ that cannot be computed by a TM with s states unless $cs\log s \geq 2^n$, regardless of time.

Proof. The number of TMs with s states is $\leq s^{cs}$, and each TM computes at most one function (it may compute none, if it does not stop). The number of functions on n bits is 2^{2^n}. Hence if $2^n > cs\log s$ some function cannot be computed. **QED**

Note this bound is not far from that in Exercise 1.3.

It is instructive to present this basic result as an application of the probabilistic method:

Proof. Let us pick f uniformly at random. We want to show that

$$\mathbb{P}_f[\exists \text{ an } s\text{-state TM } M \text{ such that } M(x) = f(x) \text{ for every } x \in [2]^n] < 1.$$

Indeed, if the probability is less than 1 than some function exists that cannot be computed. By a union bound we can say that this probability is

$$\leq \sum_M \mathbb{P}_f[M(x) = f(x) \text{ for every } x \in [2]^n],$$

where the sum is over all s-state machines. Each probability in the sum is $(1/2)^{2^n}$, since M is fixed. The number of s-state machines is $\leq s^{cs}$. So the sum is $\leq s^{cs}(1/2)^{2^n}$, and we can conclude as before taking logs. **QED**

3.3 Diagonalization and time hierarchy

Can you compute more if you have more time? For example, can you write a program that runs in time n^2 and computes something that cannot be computed in time $n^{1.5}$? The answer is yes for trivial reasons if we allow for non-boolean functions.

Exercise 3.3. Give a function $f : [2]^* \to [2]^*$ in $\text{Time}(n^2) \setminus \text{Time}(n^{1.5})$.

The answer is more interesting if the functions are boolean. Such results are known as *time hierarchies*, and a generic technique for proving them is *diagonalization,* applicable to any model.

We first illustrate the result in the simpler case of partial functions, which contains the main ideas. Later we discuss total functions.

Theorem 3.3. There is a partial function in TM-Time$(t(n))$ such that any TM M computing it runs in time $\geq c_M t(n)$, for any $t(n) = \omega(1)$.

In other words, $\text{Time}(t(n)) \supsetneq \text{Time}(o(t(n))$.

Proof. Consider a TM H that on input $x = (M, 1^{n-|M|})$ of length n runs M on x until it stops and then complements the answer. (We can use a simple encoding of these pairs, for example every even-position bit of the description of M is a 0.) The TM is specifically implemented as follows: H begins by making a copy of M in time $|M|^c \leq t(n)/2$. Then every step of the computation of M can be simulated by H with $|M|^c$ steps, always keeping the description of M to the left of the head.

Now define X to be the subset of pairs s.t. M runs in time $\leq t(n)/|M|^c$ on inputs of length n, and $|M|^c \leq t(n)/2$. On these inputs, H runs in time $|M|^c + |M|^c \cdot t(n)/|M|^c \leq t(n)$, as desired.

Now suppose N computes the same function as H in time $t(n)/|N|^c$. Note that $x := (N, 1^{n-|N|})$ falls in the domain X of the function, for n sufficiently large, using that $t(n) = \omega(1)$. Now consider running N on x. We have $N(x) = H(x)$ by supposition, but $H(x)$ is the complement of $N(x)$, contradiction. **QED**

This proof is somewhat unsatisfactory; in particular we have no control on the running time of H on inputs not in X. It is desirable to have a version of this fundamental result for total functions. Such a version is stated next. It requires additional assumptions on t and a larger gap between the running times. Perhaps surprisingly, as we shall discuss, both requirements are essential.

Theorem 3.4. Let $t(n) \geq n$ be a function s.t. $f(x) := t(|x|)$ is in TM-Time$(t(n)/\log^c n)$.

There is a total function in TM-Time$(ct(n)\log t(n))$ that cannot be computed by any TM M in time $c_M t(n)$.

The assumption about t is satisfied by all standard functions, including all those in this book. (For example, take $t(n) := n^2$. The corresponding f is then $|x|^2$. To compute f on input x of n bits we can first compute $|x| = n$ in time $cn\log n$ (Exercise 1.2). This is a number of $b := \log n$ bits. We can then square this number in time b^c. Note that the time to compute $f(x)$ is dominated by the $cn\log n$ term coming from computing $|x|$, which does not depend on t and is much less than the $n^2/\log^c n$ in the assumption.) The assumption cannot be removed altogether because there exist pathological functions t for which the result is false.

The proof is similar to that of Theorem 3.3. However, to make the function total we need to deal with arbitrary machines, which may not run in the desired time or even stop at all. The solution is to clock H in a manner similar to the proof of the universal machine, Lemma 1.1.

Also, we define a slightly different language to give a stronger result – a unary language – and to avoid some minor technical details (the possibility that the computation of f erases the input).

We define a TM H that on input 1^n obtains a description of a TM M, computes $t(n)$, and then simulates M on input 1^n for $t(n)$ steps in a way similar to Lemma 1.1, and if M stops then H outputs the complement of the output of M; and if M does not stop then H stops and outputs anything. Now H computes a function in time about $t(n)$. We argue that this function cannot be computed in much less time as follows. Suppose some TM M computes the function in time somewhat less than $t(n)$. Then we can pick an 1^n for which H obtains the description of this M, and the simulation always stops. Hence, for that 1^n we would obtain $M(1^n) = H(1^n) = 1 - M(1^n)$, which is a contradiction.

However, there are interesting differences with the simulation in Lemma 1.1. In that lemma the universal machine U was clocking the steps of the machine M being simulated. Now instead we need to clock the steps of U itself, even while U is parsing the description of M to compute its transition function. This is necessary to guarantee that H does not waste time on big TM descriptions.

Whereas in Lemma 1.1 the tape was arranged as

$$(x, M, \underline{i}, t', y),$$

it will now be arranged as

$$(x, M', \underline{i}, t', M'', y)$$

which is parsed as follows. The description of M is $M'M''$, M is in state $\underline{i}$, the tape of M contains xy and the head is on the left-most symbol of y. The integer t' is the counter decreased at every step

Proof. Define TM H that on input 1^n:

1. Compute $(n, t(n), 1^n)$.
2. Compute $(M_n, t(n), 1^n)$. Here M_n is obtained from n by removing all left-most zeroes until the first 1. I.e., if $n = 0^j 1x$ then $M_n = x$. This is similar to the fact that a program does not change if you add, say, empty lines at the bottom.
3. Simulate M_n on 1^n, reducing the counter $t(n)$ at every step, including those parsing M_n, as explained before.
4. If M_n stops before the counter reaches 0, output the complement of its output. If the counter reaches 0 stop and output anything.

Running time of H.

1. Computing n is similar to Exercise 1.2. By assumption $t(n)$ is computable in time $t(n)/\log^c n$. Our definition of computation allows for erasing the input, but we can keep n around spending at most another $\log^c n$ factor. Thus we can compute $(n, t(n))$ in time $t(n)$. Finally, in case it was erased, we can re-compute 1^n in time $cn \log n$ by keeping a counter (initialized to a copy of n).
2. This takes time $c(n + t(n))$: simply scan the input and remove zeroes.
3. Decreasing the counter takes $c|t(n)|$ steps. Hence this simulation will take $ct(n) \log t(n)$ time.

Overall the running time is $ct(n) \log t(n)$.

Proof that the function computed by H requires much time. Suppose some TM M computes the same function as H. Consider inputs 1^n where $n = 0^j 1M$. Parsing the description of M to compute its transition function takes time c_M, a value that depends on M only and not on j. Hence H will simulate $\lfloor t(n)/c_M \rfloor$ steps of M. If M stops within that time (which requires $t(n)$ to be larger than c_M, and so n and j sufficiently large compared to M) then the simulation terminates and we reach a contradiction as explained before. **QED**

The extra $\log t(n)$ factor cannot be reduced because of the surprising result presented in Theorem 3.5 showing that, on TMs, time $o(n \log n)$ equals time n for computing total functions.

However, tighter time hierarchies hold for more powerful models, like RAMs. Also, a time hierarchy for total functions for BPTime is... an open problem! But a hierarchy is known for partial functions.

Exercise 3.4. (1) State and prove a tighter time hierarchy for Time (which recall corresponds to RAMs) for total functions. You don't need to address simulation details, but you need to explain why a sharper separation is possible.

(2) Explain the difficulty in extending (1) to BPTime. You don't need to provide a counterexample (unknown, in fact) just explain where your argument fails if you replace Time with BPTime.

(3) State and prove a time hierarchy for BPTime for partial functions.

3.3.1 TM-Time($o(n \log n)$) = TM-Time($n + 1$)

In this subsection we prove the result in the title, which we also mentioned earlier. First let us state the result formally.

Theorem 3.5. Let $f : [2]^* \to [2]$ be in TM-Time($t(n)$) for a $t(n) = o(n \log n)$. Then $f \in$ TM-Time($n + 1$).

Note that time $n + 1$ is barely enough to scan the input. Indeed, the corresponding machines in Theorem 3.5 will only move the head in one direction. The "+1" only reflects that we charge one time step to stop in 1.1. 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 *regular* or *finite-state-automata*.

The rest of this section is devoted to proving the above theorem. Let M be a machine for f witnessing the assumption of the theorem. We can assume that M stops on every input (even though our definition of time only applies to large enough inputs), possibly by adding $\leq n$ to the time, which does not change the assumption on $t(n)$. The theorem now follows from the combination of the next two lemmas.

Lemma 3.1. Let M be a TM running in time $t(n) \leq o(n \log n)$. Then on every input $x \in [2]^*$ every i-CS with $i \leq |x|$ has length $\leq c_M$.

Proof. Assume towards a contradiction that for every $b \in \mathbb{N}$ there are inputs which have crossing sequences of length $\geq b$. Specifically let $x(b)$ be a shortest input of length $n(b) := |x(b)|$ such that there exists $j \in \{0, 1, \ldots, n(b)\}$ for which the j-CS in the computation of M on $x(b)$ has length $\geq b$.

We have that $n(b) \to \infty$ for $b \to \infty$ (see exercise below).

There are $n(b)+1 \geq n(b)$ tape boundaries within or bordering $x(b)$. If we pick a boundary uniformly at random, the average length of a CS on $x(b)$ is $\leq t(n(b))/n(b)$. Hence there are $\geq n(b)/2$ choices for i s.t. the i-CS on $x(b)$ has length $\leq 2t(n(b))/n(b)$.

The number of such crossing sequences is

$$\leq (s+1)^{2t(n(b))/n(b)} = (s+1)^{o(n(b)\log(n(b))/n(b)} = n(b)^{o(\log s)}.$$

Hence, the same crossing sequence occurs at $\geq (n(b)/2)/n(b)^{o(\log s)} \geq 4$ positions i, using that $n(b)$ is large enough.

Of these four, one could be the CS of length $\geq b$ from the definition of $x(b)$. Of the other three, two are on the same side of j. We can remove the corresponding interval of the input without removing the CS of length $\geq b$. Hence we obtained a shorter input with a CS of length $\geq b$, contradicting our definition of $x(b)$ and so our initial assumption. **QED**

Exercise 3.5. Prove $n(b) \to \infty$ for $b \to \infty$.

Lemma 3.2. 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 $\leq b$. Then f is computable in time n by a TM with c_b states that only moves the head in one direction.

Exercise 3.6. Prove this.

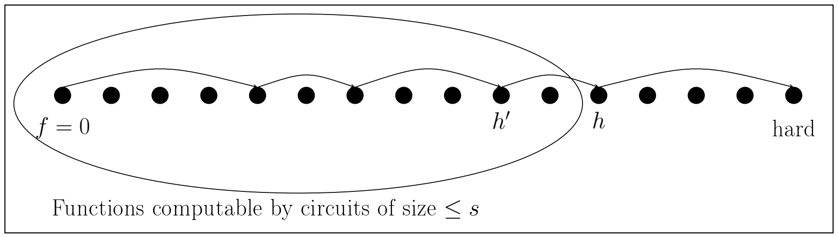


Figure 3.1: Illustration of the proof of Theorem 3.7.

3.4 Circuits

The situation for circuits is similar to that of 2-TMs, and by Theorem 1.6 we know that proving $\omega(n \log n)$ bounds on circuits is harder than for 2-TMs. Even a bound of cn is unknown. The following is a circuit version of Theorem 3.2. Again, the bound is close to what we saw in Theorem 1.4.

Theorem 3.6. There are functions $f : [2]^n \to [2]$ that require circuits of size $\geq c2^n/n$, for every n.

Proof. A circuit of size s can be described by $cs \log s$ bits, while a function can be described by 2^n bits. Since each circuit computes ≤ 1 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 3.7. Show that there are functions $f : [2]^n \to [2]$ that require *alternating* circuits of size $\geq 2^{cn}$, for every n. Give two proofs, one based on the statement of Theorem 3.6, another one modifying the proof of Theorem 3.6. Explain why the bound is weaker than in Theorem 3.6.

One can prove a hierarchy for circuit size, by combining Theorem 3.6 and Theorem 1.4. As stated, the results give that circuits of size cs compute more functions than those of size s. In fact, the "$o(1)$" in the theorems is small, so one can prove a sharper result. But a stronger and more enjoyable argument exists.

Theorem 3.7. [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 $s + cn$ but not by circuits of size s.

Proof. Refer to figure 3.1. 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 3.6, 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 that 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 cn. **QED**

Exercise 3.8. Prove a stronger hierarchy result for alternating circuits, where the "cn" in Theorem 3.7 is replaced with "c." (But only prove this for $s \leq 2^{cn}$, using Theorem 3.7.)

In fact, this improvement is possible even for (non alternating) circuits, see Problem 3.2.

3.4.1 The circuit won't fit in the universe: Non-asymptotic impossibility

Most of the results in this book are *asymptotic*, i.e., we do not explicitly work out the constants because they become irrelevant for larger and larger input lengths. As stated, these results don't say anything for inputs of a fixed length. For example, any function on 10^{100} bits is in Time(c).

However, it is important to note that all the proofs are *constructive* and one can work out the constants, and produce non-asymptotic results. We state next one representative example when this was done. It is about a problem in logic, an area which often produces very hard problems.

On an alphabet of size 63, the language used to write formulas includes first-order variables that range over $\mathbb{N}$, second-order variables that range over finite subsets of $\mathbb{N}$, the predicates "$y = x + 1$" and "$x \in S$" where x and y denote first-order variables and S denotes a set variable, and standard quantifiers, connectives, constants, binary relation symbols on integers, and set equality. For example one can write things like "every finite set has a maximum:" $\forall S \exists x \in S \forall y \in S, y \leq x$.

Theorem 3.8. To decide the truth of logical formulas of length at most 610 in this language requires a circuit containing at least 10^{125} gates. So even if each gate were the size of a proton, the circuit would not fit in the known universe.

Their result applies even to randomized circuits with error 1/3, if 610 is replaced with 614. (We can define randomized circuits analogously to BPTime.)

3.5 Average-case hardness, and correlation

An impossibility result is just one of the negative things we can ask about a model. We can ask stronger things. Perhaps the most natural strengthening is proving an *average-case* impossibility result. One justification for this quest is that an impossibility result may be irrelevant to instances that occur "in nature," if the latter follow a specific distribution.

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.

Definition 3.2. Let X be an input distributed according to a distribution D. Let h and f be boolean functions, and let F be a set of boolean functions.

We say that h is *δ-hard* for f w.r.t. D (or over D) if $\mathbb{P}[f(X) \neq h(X)] \geq \delta$, 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

$$|\mathbb{E}[(-1)^{f(X)+h(X)}]| = |\mathbb{P}[f(X) = h(X)] - \mathbb{P}[f(X) \neq h(X)]| = |1 - 2\mathbb{P}[f(X) \neq h(X)]|.$$

We also introduce the notation $\mathbb{E}e[y]$ for $\mathbb{E}[(-1)^y]$ which allows us to write correlation as

$$|\mathbb{E}e[f(X) + g(X)]| .$$

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 assume to be the uniform distribution.

Thus the correlation (or hardness) between two functions is a measure of how often the functions agree on a uniform input. 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 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. (As can be proved via counting arguments as in Chapter 3.)

Exercise 3.9. [Average-case/correlation version of Theorem 3.6] Prove that there are functions $h : [2]^n \to [2]$ that have correlation 2^{-an} with circuits of size 2^{an}, for some constant $a > 0$.

At first sight, average-case hardness and correlation bounds seem stronger than impossibility. In fact, *impossibility* results are *equivalent* to strong correlation bounds!

Theorem 3.9. [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. $\mathbb{E}_{x \leftarrow D} e[f(x) + h(x)] \geq \epsilon$. Then there exist cn/ϵ^2 functions $f_i \in F$ s.t.

$$h = \text{Majority}(f_1, f_2, \ldots, f_{cn/\epsilon^2}).$$

[$\Rightarrow$] Suppose $h = \text{Majority}(f_1, f_2, \ldots, f_t)$ for some t. Then for any distribution D there is i s.t. $\mathbb{E}e[(f_i + h)(D)] \geq 1/t$.

Exercise 3.10. Prove the [$\Rightarrow$]. Feel free to assume t is odd for simplicity.

In particular, for models that are able to compute majority, such as PCkt, we get that $h \notin$ PCkt iff there is a distribution D for which any $f \in$ PCkt has correlation $\le 1/n^a$ 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. By contrast, in section 11.2.2 we will study a similar connection but under the *uniform* distribution. The proofs are closely related.

We now develop machinery to understand and prove Theorem 3.9. A useful viewpoint, here and elsewhere, is the *equivalence between randomness in the input and having it in the model*:

Corollary 3.1. Let h be a function, and F a set of functions. There is a distribution over F s.t. $\mathbb{E}e[F(x) + h(x)] \ge \alpha$ for every input x iff for every distribution X over inputs there is $f \in F$ s.t. $\mathbb{E}e[f(X) + h(X)] \ge \alpha$.

Exercise 3.11. Prove the "only if" direction of Corollary 3.1.

The "if" direction of Corollary 3.1 is a special case of the *min-max theorem from game theory, a.k.a. linear-programming duality, theorem of the alternatives for linear systems, etc*, stated next.

Theorem 3.10 (Linear duality)**.** Let X and Y be sets, $p : X \times Y \to \mathbb{R}$, and $\alpha \in \mathbb{R}$. Then either

there is a distribution D_X on X s.t. $\mathbb{E}_{D_X} p(D_X, y) \le \alpha$ for every $y \in Y$, or
there is a distribution D_Y on Y s.t. $\mathbb{E}_{D_Y} p(x, D_Y) \ge \alpha$ for every $x \in X$.

Exercise 3.12. Prove the "if" direction of Corollary 3.1. Hint: Don't worry about $\le$ vs. $<$; I don't.

Theorem 3.9 follows from Corollary 3.1 and tail bounds (Theorem 2.1), similarly to the proof of the error-reduction Theorem 2.2 for BPTime.

Proof of Theorem 3.9, $\Leftarrow$. Use Corollary 3.1 to get a distribution on F. The majority of cn/ϵ^2 samples from F has error $< 2^{-n}$ by Theorem 2.1. By a union bound we can fix the samples to the f_i. **QED**

3.6 Problems

Problem 3.1. Hierarchy Theorem 3.4 only gives a function f that cannot be computed fast on *all* large enough input lengths: it is consistent with the theorem that f can be computed fast on infinitely many input lengths.

Give a function $f : [2]^* \to [2]^*$ mapping x to $[2]^{\log\log\log|x|}$ that is computable in time n^c but such that for any RAM M running in time n^2 the following holds. For every $n \ge c_M$ and every $x \in [2]^n$ one has $M(x) \ne f(x)$.

Hint: Note the range of f. Can this result hold as stated with range [2]?

Problem 3.2. Replace "cn" in Theorem 3.7 with "c."

Problem 3.3. Prove that $\{0^i1^i : i \geq 0\} \in$ TM-Time$(cn \log n) \setminus$ TM-Time$(t(n))$, for any $t(n) = o(n \log n)$.

For the negative result, don't use pumping lemmas or other characterization results not covered in this book.

Problem 3.4. The following argument contradicts Theorem 3.4; what is wrong with it?

"By Theorem 3.5, TM-Time$(n \log^{0.9} n)$ = TM-Time(cn). By padding (section §1.6), TM-Time$(n \log^{1.1} n)$ = TM-Time$(n \log^{0.9} n)$. Hence TM-Time$(n \log^{1.1} n)$ = TM-Time(n)."

3.7 Notes

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. [287]

Crossing sequences and the tight quadratic bound for palindromes are from [128].

The time hierarchy originates in [130] and was later optimized [131].

The existence of hard functions via counting arguments, Theorem 3.6, goes back to [233], Theorem 7, "*Are most functions simple or complex?*"

Theorem 3.5 follows by combining results in [129, 159].

Theorem 3.8 is from [246], see the reference for the history of the result.

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, by the Pythagoreans about 2500 years ago. This can be seen as a statement about computation, where the computational model are rational numbers, and 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 began to appear. Complexity theory takes a more quantitative angle on impossibility. It focuses on a resource such as time and 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 then. As we have seen, the first results on tape machines are from the 60s, and so are the first results on circuits such as [195, 188], though circuit complexity started already from [233]. [258] provides a survey of research on lower bounds in the Soviet Union. A fresh wave of results in circuit complexity came in the 80's and 90's, including a proof that an explicit function requires circuits of size cn [45] – a bound that has stood for a long time and has only seen minor improvements [76] – and several other results on small-depth circuits discussed later in section §8.5, Chapter **??**, Chapter 13. This wave soon "hit the wall," for example the "wall" of constant-depth circuits with mod 6 gates, see section §8.3, and stalled. Two lines of works 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 [210, 78, 80, 290] (cf. section §7.9 and Theorem 8.11) that do not have a non-uniform counterpart. Since the beginning of complexity a number of "barrier" results have been proposed to explain the lack of progress, see Chapter 16. And that's pretty much where we stand now.

Chapter 4

The art of reductions

One can relate the complexity of functions via *reductions*. This concept is so ingrained in common reasoning that giving it a name may feel, 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.

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 $X = \text{P}$, or $X = \text{BPP}$; 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 is *subroutine* or *oracle* or *black-box* reduction. In this type, we augment our computational model with the ability to make queries to some function f. For example, for TMs we can reserve one tape for the input y to f, and augment the states with a special *query* state. Upon entering that state, in one time step the input y to f is replaced with the output $f(y)$. For some complexity classes, making this definition useful is not immediate. For example, should the input y be erased? This matters if precise time bounds are sought. But for a robust class like P the definition is clear:

Definition 4.2. We denote by P^f the functions computable in P with oracle access to f.

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

Definition 4.3. 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 4.1. Suppose that f map reduces to g in X.
(1) Suppose $X = \mathrm{P}$. Show f reduces to g in X.
(2) Suppose $X = \bigcup_d \mathrm{Time}(d \cdot n^2)$. Can you still show that f reduces to g in X?

In general, the harder the problems we are reducing the more constrained the class X can be. In several interesting cases, extremely constrained classes X suffice for the reductions, such as functions with constant locality, or even locality one, cf. Problem 4.7. But for many other problems, the reductions we shall see are not even mapping reductions. In fact, our first example is not a mapping reduction.

4.2 Reductions

4.2.1 Multiplication

Summing two n-bit integers is in $\mathrm{CktGates}(cn)$ (Exercise 1.9). But the smallest circuit known for multiplication has $\geq cn \log n$ gates. (The same situation holds for MTMs; over RAMs and related models multiplication can be done in time cn.) 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. Multiplication is the problem of computing the product of two n-bit integers. Squaring is the problem of computing the square of an n-bit integer.

Theorem 4.1. If Squaring has linear-size circuits then Multiplication has linear-size circuits.

Proof. Suppose C computes Squaring. Then we can multiply using equation (4.1). Specifically, given a and b we use Exercise 1.9 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 1.9 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**

4.2.2 3Sum

Definition 4.5. The 3Sum problem: Given a set S of integers, are there three integers $x, y, z \in S$ that sum to 0?

In Definition 4.5 we allow for repeated integers (e.g., $x = y$). Problem 4.1 considers the variant where repetitions are not allowed. It is good to think of integers with $c \log n$ bits, and some works show that these are the hardest instances.

It is easy to solve 3Sum in time $n^2 \log^c n$ on a RAM, at least if the numbers have $\log^c n$ bits: We can first sort the integers 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 4.2. Solve 3Sum in time $n^2 \log^c n$ when each input integer can have a different length. Hint: Recall that n refers to the total input length. As a warm-up, begin with the case where each integer takes w bits.

3Sum has been conjectured to require quadratic time.

Definition 4.6. SubquadraticTime $:= \bigcup_{\epsilon>0} \text{Time}(n^{2-\epsilon})$.

Exercise 4.3. Show that 3Sum is subquadratic equivalent to the Tripartite-3Sum problem: Given sets A_1, A_2, and A_3 of numbers, are there $a_i \in A_i$ s.t. $a_1 + a_2 + a_3 = 0$? Hint: Recall in Definition 4.5 we allow for duplicates.

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

Theorem 4.2. Collinearity $\in$ SubquadraticTime$\Rightarrow$ 3Sum $\in$ SubquadraticTime (i.e., Conjecture 4.1 is false).

Proof. We map instance $a_1, a_2, \ldots, a_t$ of 3Sum to the points

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

and solve Collinearity on those points.

To verify correctness, notice that points $(x, x^3), (y, y^3)$, and (z, z^3) 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

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

Assuming $y \neq z$, i.e., that the 3Sum instance consists of distinct numbers, this is equivalent to $x + y + z = 0$, as desired. (The case where there can be duplicates is left as an exercise.)

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

This problem can be solved in time $n^{2\omega/(\omega+1)+o(1)}$ where $\omega < 2.373$ is the exponent of matrix multiplication. If $\omega = 2$ then the bound is $n^{1.3\bar{3}+o(1)}$. It is not known if any subquadratic algorithm for 3Sum would improve these bounds. However, we can show that an improvement follows if 3Sum $\in$ Time$(n^{1+\epsilon})$ for a small enough ϵ.

Theorem 4.3. 3Sum $\in$ Time$(t(n)) \Rightarrow$ 3Cycle-Detection $\in$ BPTime$(ct(n))$, for any $t(n) \geq n$.

The reduction can be derandomized (that is, one can replace BPTime with Time in the conclusion) but the randomized case contains the main ideas.

Proof. Given a list of t edges with $c \log t$ bits per node, we assign random numbers r_x with $4 \log t$ 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 $n = ct \log t$.

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**

Exercise 4.4. Prove analogous results for:

3Sum vs. 3 Cycles on undirected graphs.

4Sum vs. 4 Cycles on undirected graphs. Hint: This might not be as easy as the first part.

To be clear, in the input to the undirected graph problems we do not allow repeated edges among nodes, and a cycle cannot use an edge more than once.

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

4.3 Reductions from 3Sat

In this section we begin to explore an important cluster of problems not known to be in BPP. 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.

Perhaps the most basic problem in the cluster is the following.

Definition 4.9. A 3CNF is a CNF where every clause has at most three literals. The 3Sat problem: Given a 3CNF ϕ, is there an assignment x s.t. $\phi(x) = 1$?

Conjecture 4.2. 3Sat$\not\in$ P.

Stronger conjectures have been made.

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.

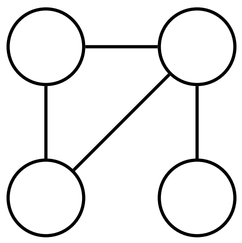
We now give reductions from 3Sat to several other problems. The reductions are in fact mapping reductions. Moreover, the reduction map can be extremely restricted, see Problem 4.7. In this sense, therefore, these reductions can be viewed as direct translations of the problems, and maybe we shouldn't really be thinking of the problems as different, even if they at first sight refer to different objects (formulas, graphs, numbers, etc.).

For videos covering these reductions you can watch videos 29, 30, 31, and 32 covering reductions: 3SAT to CLIQUE, CLIQUE to VERTEX-COVER, 3SAT to SUBSET-SUM, 3SAT to 3COLOR from `https://www.ccs.neu.edu/home/viola/classes/algm-generic.html` Note: The videos use the terminology "polynomial time" instead of "power time" here.

4.3.1 3Sat to Clique

Definition 4.10. 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 4.1. The following graph has a clique of size 3 but not of size 4:



Theorem 4.4. Clique $\in$ P $\Rightarrow$ 3Sat $\in$ 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 P.

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

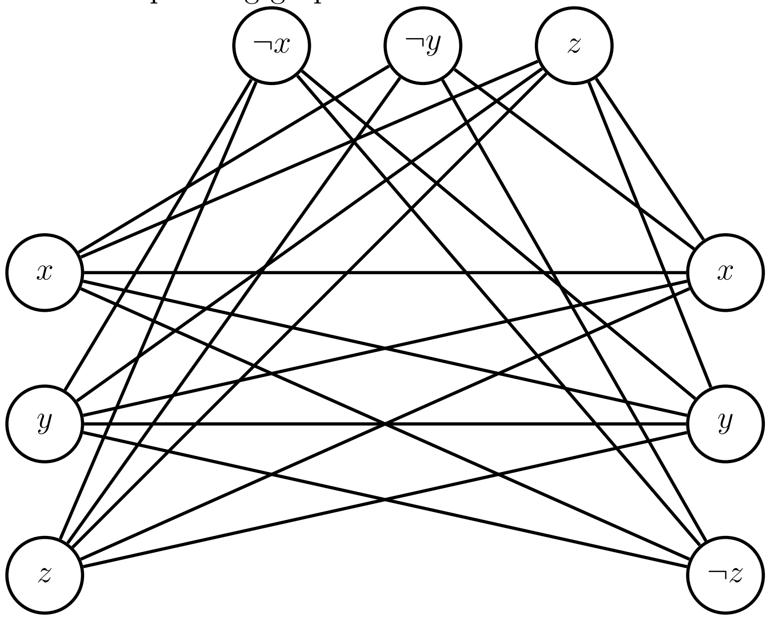
Only if: Given a satisfying assignment, collect exactly one node which is satisfied in each clause. This makes $t = k$ nodes. For any pair of such nodes, (A) does not hold by construction, and (B) because they correspond to an assignment.

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**

Example 4.2. Consider

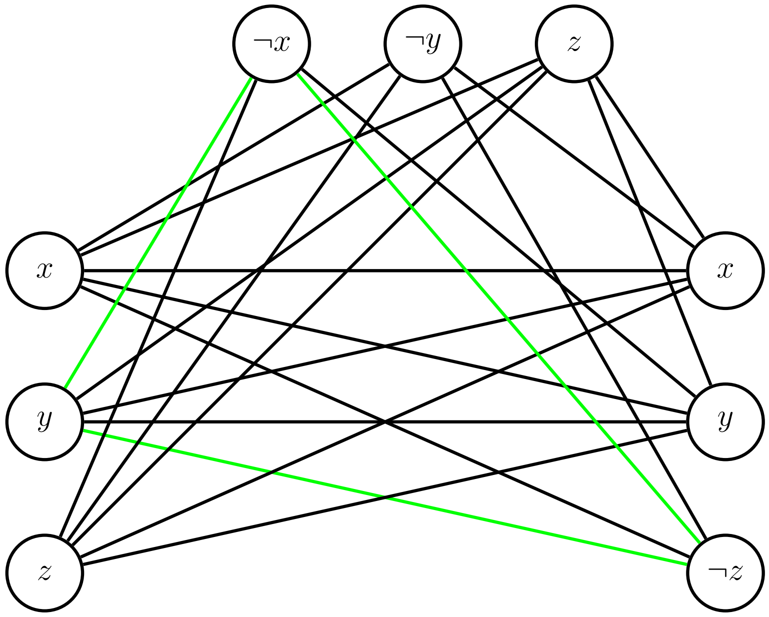
$$\varphi = (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z).$$

The corresponding graph G is:

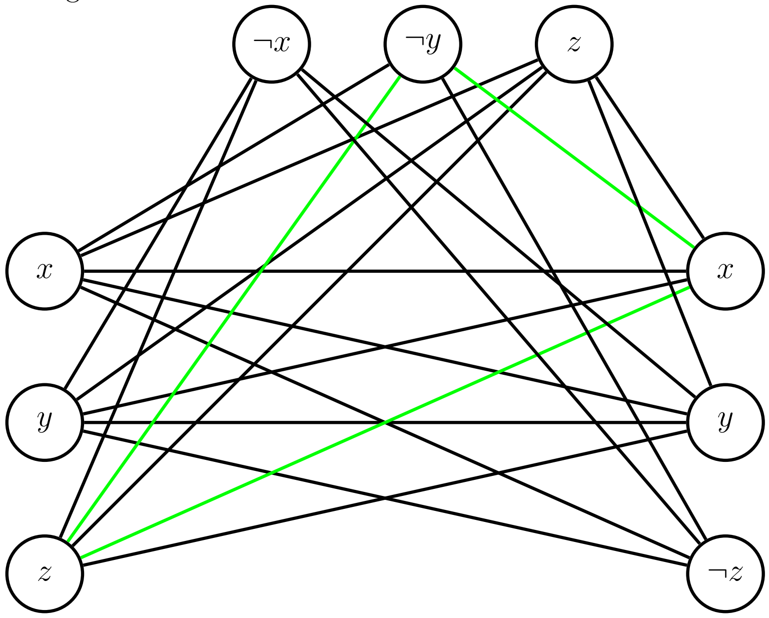


We seek cliques of size $t = k = 3$, a.k.a. triangles.

A satisfying assignment to φ is $x = 0; y = 1; z = 0$. The corresponding clique is shown next in green:



Another satisfying assignment is $x = 1; y = 0; z = 1$. The corresponding clique is shown next in green:



4.3.2 Clique to cover-by-vertexes

TBD

4.3.3 3Sat to Subset-Sum

Definition 4.11. 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 4.3. 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 $t := 15$.

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 4.5. Subset-sum $\in$ P $\Rightarrow$ 3Sat $\in$ 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 power time.

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**

Example 4.4. Let $\varphi := (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z)$. The subset-sum instance is:

	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

A satisfying assignment is $x = 0, y = 1, z = 0$. The corresponding subset is:

		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	
(2x)	$a_{c1} =$	0	0	0	1	0	0	(choose twice)
(2x)	$a_{c2} =$	0	0	0	0	1	0	(choose twice)
(2x)	$a_{c3} =$	0	0	0	0	0	1	
	$t =$	1	1	1	3	3	3	

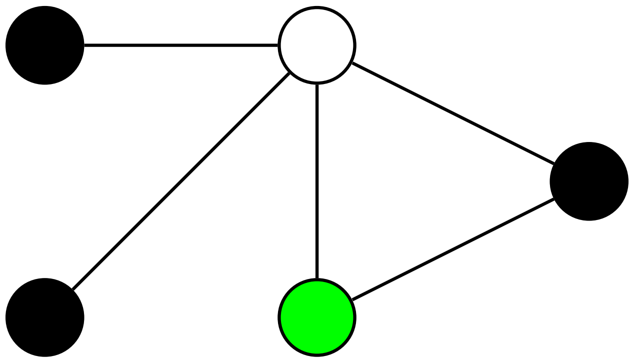
Another satisfying assignment is $x = y = z = 1$ with corresponding subset

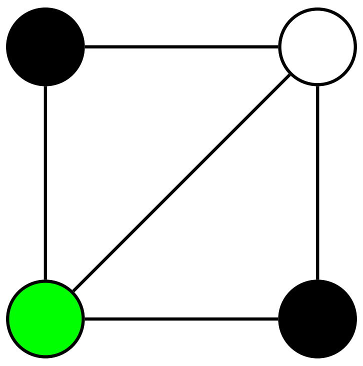
		var x	var y	var z	clause 1	clause 2	clause 3	
	$a_x^T =$	1	0	0	1	0	1	
	$a_x^F =$	1	0	0	0	1	0	
	$a_y^T =$	0	1	0	1	0	1	
	$a_y^F =$	0	1	0	0	1	0	
	$a_z^T =$	0	0	1	1	1	0	
	$a_z^F =$	0	0	1	0	0	1	
(2x)	$a_{c1} =$	0	0	0	1	0	0	
(2x)	$a_{c2} =$	0	0	0	0	1	0	(choose twice)
(2x)	$a_{c3} =$	0	0	0	0	0	1	
	$t =$	1	1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

4.3.4 3Sat to 3Color

Definition 4.12. A 3-coloring of a graph is a coloring of each node, using at most 3 colors, such that no adjacent nodes have the same color. The 3Color problem: Given a graph G, does it have a 3 coloring?

Example 4.5. The following graphs have a 3-coloring, shown:





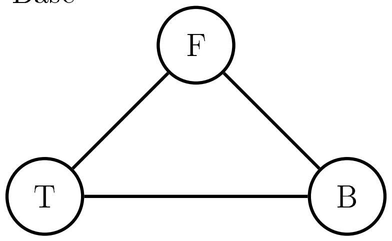
An example of a graph that cannot be 3-colored is a clique of size 4.

Theorem 4.6. 3Color$\in$ P $\Rightarrow$ 3Sat $\in$ P.

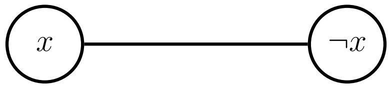
Proof. Given a 3CNF ϕ, we construct a graph G as follows.

Add 3 special nodes called the "palette" in a clique:

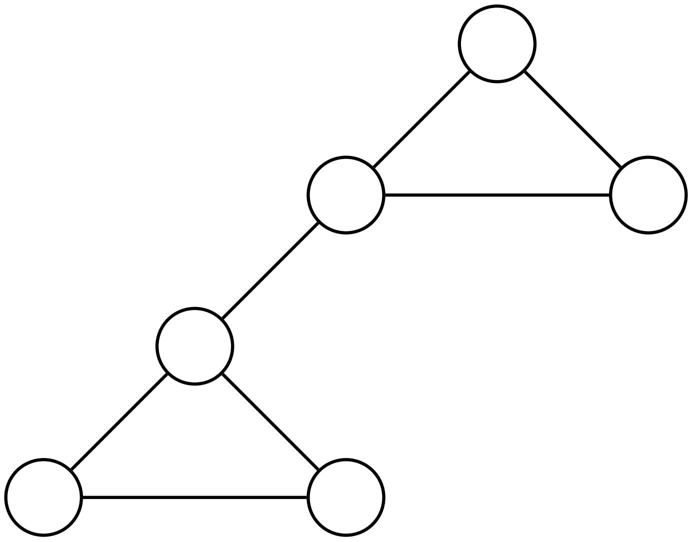
T = True
F = False
B = Base



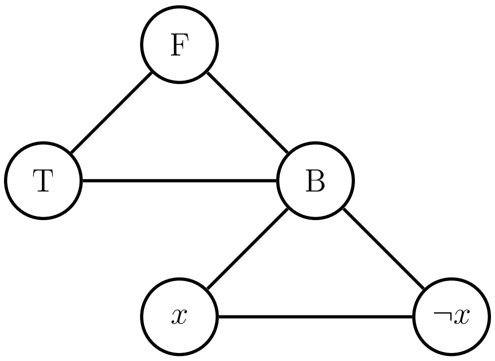
For each variable add 2 literal nodes with an edge between them



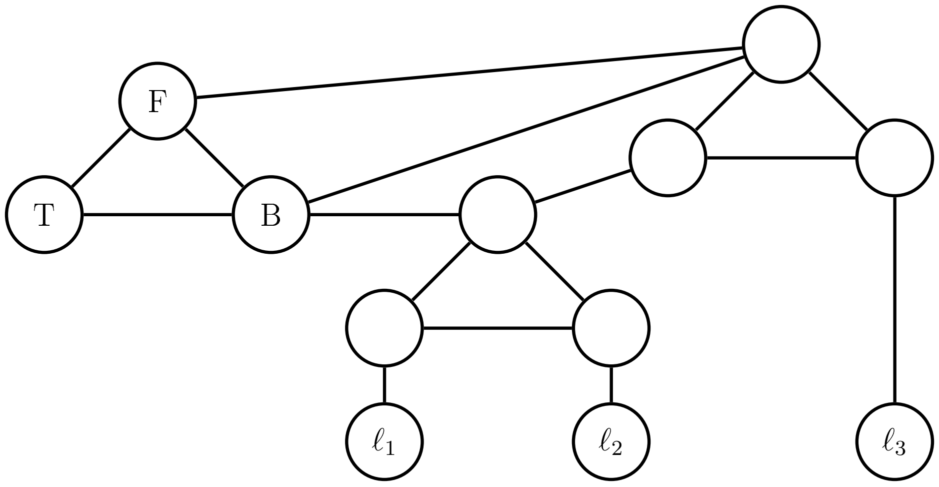
For each clause add the following gadget with 6 nodes



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:

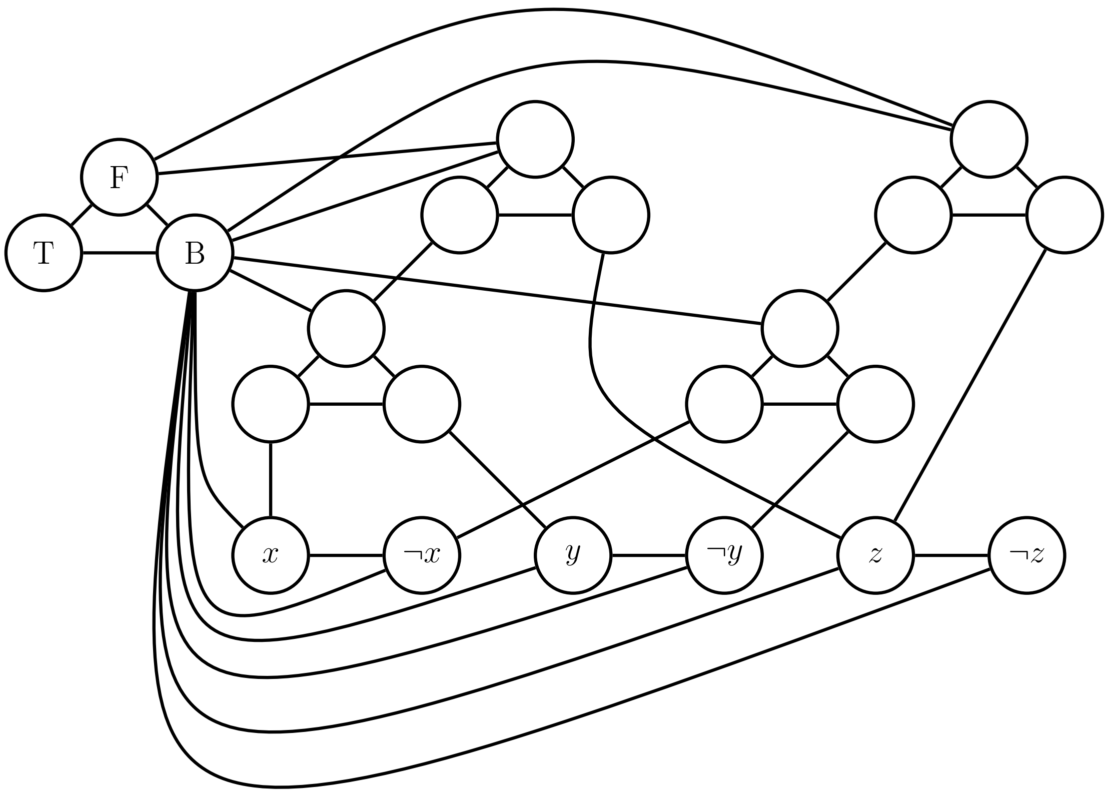


The construction of G is in P. We now prove that φ is satisfiable iff G is 3 colorable. We begin with some preliminary remarks. In the palette, T's color represents TRUE, and F's color represents FALSE. Note in a 3-coloring, all variable nodes must be colored T or F because they are connected to B. Also, 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.

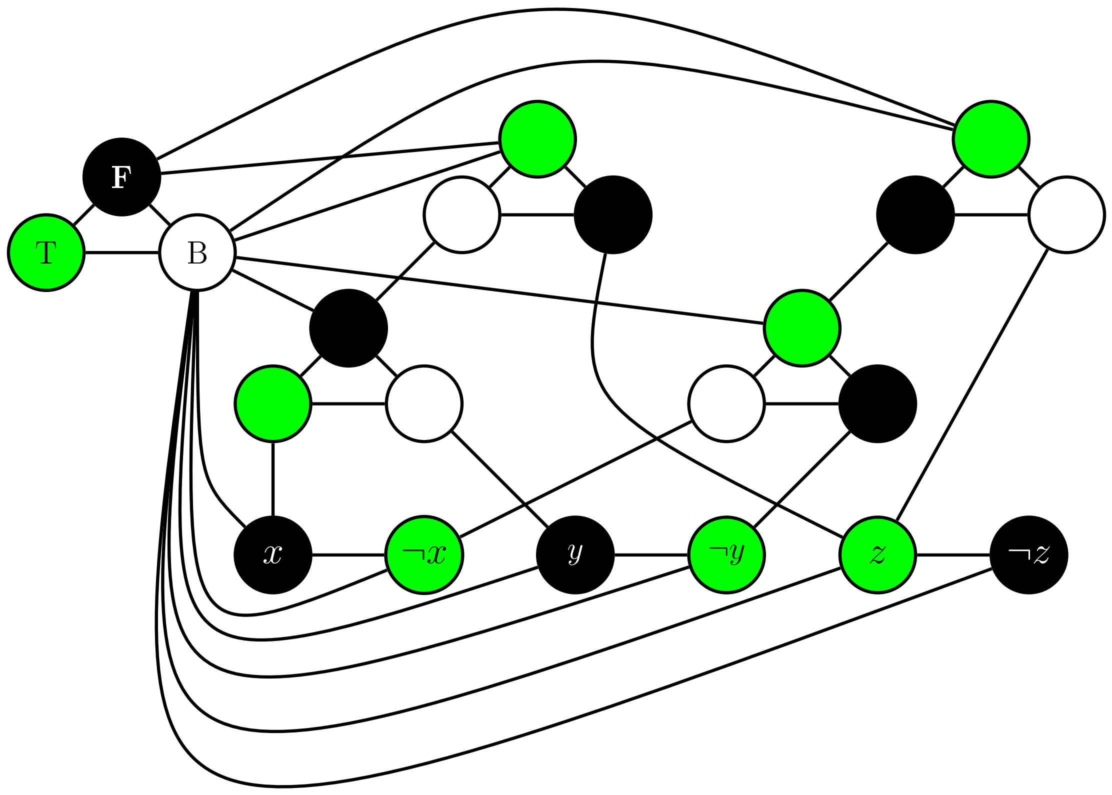
The important claim is that a clause gadget can be 3-colored iff any of the literals connected to it is colored True. This holds because each of the two triangles in a the clause gadget is computing "Or:" In a triangle, the top node is colored according to the Or of the two literals connected to the 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**

Example 4.6. Let $\varphi := (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee z)$. Then G is



A satisfying assignment is $x = y = 0$ and $z = 1$. The corresponding coloring is



Exercise 4.5. The problem System is defined as follows. A *linear inequality* is an inequality involving sums of variables and constants, such as $x + y \geq z$, $x \leq -17$, and so on. A system of linear inequalities has an *integer* solution if it is possible to substitute integer values for the variables so that every inequality in the system becomes true. The language System consists of systems of linear inequalities that 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.

Exercise 4.6. For an integer k, k-Color is the problem of deciding if the nodes of a given undirected graph G can be colored using k colors in such a way that no two adjacent vertices have the same color.

Reduce 3-Color to 4-Color in P.

Reductions in the opposite directions are possible, and so in fact the problems in this section are *power-time equivalent* in the sense that any 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 4.7. 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.
3. Briefly conclude the proof.

Thus, we are identifying a cluster of problems which are all power-time equivalent.

4.4 Power hardness from SETH

In this section we show that a conjecture similar to Conjecture 4.1 can be proved assuming SETH. This is an interesting example of how we can connect different parameter regimes, since SETH is stated in terms of exponential running times. In general, "scaling" parameters is a powerful technique in the complexity toolkit.

Definition 4.13. The Or-Vector problem: Given two sets 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(n^2), in fact the setting is slightly easier as it's more natural to assume that the bit vectors have the same length here, cf. Exercise 4.2. We can show that a substantial improvement would disprove SETH.

Theorem 4.7. Or-Vector $\in$ SubquadraticTime $\Rightarrow$ SETH is false.

Proof. 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 $\exists a \in A, b \in B$ such that $a \vee b = 1^d$.

Constructing these sets takes time Nd^c. If Or-Vector $\in$ Time($n^{2-\epsilon}$) for some $\epsilon > 0$, we can take $k = c_\epsilon$ and rule out SETH. **QED**

4.5 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 4.14. Search-3Sat is the problem: Given a satisfiable 3CNF formula, output a satisfying assignment.

Theorem 4.8. Search-3Sat reduces to 3Sat in P. That is: 3Sat $\in$ P $\Rightarrow$ Search-3Sat $\in$ P.

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 4.8. Show Clique $\in$ P $\Rightarrow$ Search-Clique $\in$ P.

4.5.1 Fastest algorithm for Search-3Sat

A curious fact about many search 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 clock programs, and stop them if they take too long. There is a particular simulation schedule which leads to efficient running times.

Theorem 4.9. There is a RAM U (for "universal") such that on input any satisfiable formula x:

(1) U outputs a satisfying assignment, and

(2) If there is a RAM M that on input x outputs a satisfying assignment for x in t steps then U stops in $c_M t + |x|^c$ steps.

We are taking advantage of the RAM model. On other models it is not known if the dependence on t can be linear.

Proof. For $i = 1, 2, \ldots$ the RAM U simulates RAM i for 2^i steps. Lemma 1.3 guarantees that for each i the simulation takes time $c2^i$. If RAM i stops and outputs y, then U checks in time $|x|^c$ if y is a satisfying assignment. If it is, then U outputs y and stops. Otherwise it continues.

Now let M be as in (2). As before, we work with an enumeration of programs where each program appears infinitely often. Hence we can assume that M has a description of length $\ell := c_M + \log t$. Thus the simulation will terminate when $i = \ell$.

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

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

QED

This result nicely illustrates how "constant factors" can lead to impractical results because, of course, the problem is that the constant in front of t is enormous. Specifically, it is exponential in the size of the program, see Problem 4.8.

4.6 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 [C]. 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 4.15. γ-Gap-3Sat is the 3Sat problem restricted to input formulas f that are either satisfiable or 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. But in fact 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 4.10. [PCP] There is $\gamma < 1$ such that γ-Gap-3Sat $\in$ P $\Rightarrow$ 3Sat $\in$ P.

Similar results can be established for other problems such as 3Color, but the reductions in the previous section don't preserve gaps and can't be immediately applied.

A major application of the PCP theorem is in *inapproximability* results. A typical optimization problem is Max-3Sat.

Definition 4.16. The Max-3Sat problem: given a 3CNF formula, find a satisfying assignment that satisfies the maximum number of clauses.

Solving 3Sat reduces to Max-3Sat (in Chapter 5 we will give a reverse reduction as well). But we can ask for β*-approximating* Max-3Sat, that is, computing an assignment that satisfies a number of clauses that is at least a β fraction of the maximum possible clauses that can be satisfied.

The PCP Theorem 4.10 implies that 3Sat reduces to $\beta-$approximating Max-3Sat, for some constant $\beta < 1$.

It has been a major line of research to obtain tight approximation factors β for a variety of problems. For example, 3Sat reduces to β-approximating Max-3Sat for any $\beta > 7/8$. This constant is tight because a random uniform assignment to the variables satisfies each clause with probability 7/8 and hence expects to satisfy a 7/8 fraction of the clauses.

Exercise 4.9. Turn this latter observation in an efficient randomized algorithm with an approximation factor $7/8 - o(1)$.

4.7 Problems

Problem 4.1. Show that 3Sum (which recall we defined allowing for repeated elements) is SubquadraticBPTime equivalent to 3Sum where repeated elements are not allowed.

Hint: Feel free to assume that the input consists of distinct elements (indeed, it's called a "set"). Prune the input using randomness.

Problem 4.2. Reduce 3Sat in P to the PIT problem (Definition 2.3) over the field with two elements.

Problem 4.3. Prove that 3Sat is not TM-Time($n^{1.99}$). (Hint: 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 3.1. Conclude by giving a suitable reduction.)

Problem 4.4. Consider the problem H: 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 *bad nodes.* The question is whether there are m paths from s to each of the destination nodes. The paths can share edges, but any two paths entering a bad node must leave through the same outgoing edge.

Reduce 3SAT to H in P.

Problem 4.5. 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 4.6. Show that 3Color $\in$ P $\Rightarrow$ Search-3Color $\in$ P.

Problem 4.7. Give an encoding of 3Sat so that the reduction to 3Color in section §4.3 can be computed, for any input length, by a 1-local map (in particular, a circuit of constant depth) (cf. Definition 1.5).

Problem 4.8. Suppose there exists a such that Theorem 4.9 holds with the running time of U replaced with $(|M| \cdot t \cdot |x|)^a$. (That is, the dependence on the program description improved to power, and we allow even weaker dependence on t.) Prove that 3Sat $\in$ P.

Problem 4.9. Use Problem 2.4 and its notation. Assume P = BPP (recall this only refers to boolean functions, see the convention on page 26). Show that given a circuit C and ϵ written in unary s.t. $p_C \geq \epsilon$ we can compute $x : C(x) = 1$ in P. In particular, given a non-zero arithmetic circuit we can find a non-zero assignment.

4.8 Notes

Circuits of size $cn \log n$ for multiplication were obtained in [120]. For the result about RAMs see [229].

Following [63] (discussed in the next chapter), [151] 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: The web of reductions from 3Sat is immense, see [91] for a starter, or the list on wikipedia: https://en.wikipedia.org/wiki/List_of_NP-complete_problems. Amusingly, among these problems are (generalized versions of) several popular games including: Tetris, Lemmings, Sudoku, etc. For an excellent exposition of this type of results see the video https://www.youtube.com/watch?v=oS8m9fSk-Wk.

The ETH and the SETH are from [139] and [141]. 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 [3] and edit distance [29].

The web of reductions of 3Sum, including Theorem 4.2, was first spun in [87] and has grown ever since. Theorem 4.3 is from [278].

Theorem 4.9 is from [168].

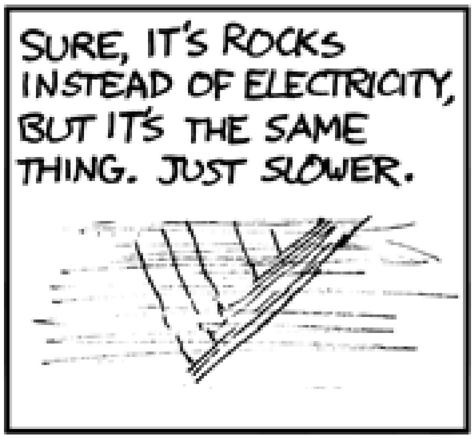
The quote at the beginning of section §4.6 is from [206]. The PCP theorem as stated in Theorem 4.10 is from [21]. A sequence of exciting works preceded and followed it. For an account, as well as a proof of the PCP theorem, see [20].

Problem 4.9 is from [98].

Problem 4.8 is from [259].

Chapter 5

Completeness: Reducing arbitrary computation


SURE, IT'S ROCKS INSTEAD OF ELECTRICITY, BUT IT'S THE SAME THING. JUST SLOWER.

https://xkcd.com/505/ (selection)

In this chapter we show how to reduce arbitrary computation to 3Sat (and hence to the other problems in section §4.3). What powers everything is the following landmark and, in hindsight, simple result which reduces circuit computation to 3Sat.

Theorem 5.1. Given a circuit $C : [2]^n \to [2]$ with s gates we can compute in P a 3CNF formula f_C in $n + s$ variables such that for every $x \in [2]^n$:

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

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 non-input 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 of 3 variables: y_i and the ≤ 2 gates that input g_i, some of which could be input variables. This can be written as a 3CNF by Theorem 1.4. Take an And of all these 3CNFs. Finally, add clause y_o for the output gate g_o. **QED**

Exercise 5.1. Write down the 3CNF for the circuit in figure 1.2, as given by the proof of Theorem 5.1.

Theorem 5.1 is *a depth-reduction* result. Indeed, note that a 3CNF can be written as a circuit of depth $c \log s$, whereas the original circuit may have any depth. This is helpful for example if you don't have the depth to run the circuit yourself. You can let someone else produce the computation, and you can check it in small depth.

We can combine Theorem 5.1 with the simulations in Chapter 1 to reduce computation in other models to 3SAT. In particular, we can reduce MTMs running in time t to 3Sat of size $t \log^c t$. To obtain such parameters we need the quasilinear simulation of MTMs by circuits, Theorem 1.6.

However, recall that a quasilinear simulation of RAMs by circuits is not known. Only a power simulation is (which is obtained by combining the power simulation of RAMs by MTMs, Theorem 1.8, with a simulation of MTMs by circuits). This would reduce RAM computation running in time t to 3CNFs of size t^c. We content ourselves with this power loss for the beginning of this chapter. Later in section §5.3 we will obtain a quasi-linear simulation using an enjoyable argument which also bypasses Theorem 1.6.

In fact, these simulations apply to a more general, *non-deterministic*, model of computation. We define this model next, and then present the simulation with power loss in 5.2.

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 $\exists y : C(x, y)$. Following the discussion before, we could do the same for other models like TMs, MTMs, and RAMs. 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 5.1. NTime($t(n)$) is the set of functions $f : X \subseteq [2]^* \to [2]$ for which there is a RAM M such that:

- $f(x) = 1$ iff $\exists y \in [2]^{t(|x|)}$ such that $M(x, y) = 1$, and

- $M(x, y)$ stops within $t(|x|)$ steps on every input (x, y).

We also define

$$\text{NP} := \bigcup_{d \geq 1} \text{NTime}(n^d),$$

$$\text{NExp} := \bigcup_{d \geq 1} \text{NTime}(2^{n^d}).$$

Note that the running time of M 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. We refer to a RAM machine as in Definition 5.1 as a *nondeterministic machine*, and to the y in $M(x, y)$ 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 2.1. The two definitions are essentially equivalent. Our 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 §4.3 are in NP.

Fact 5.1. 3Sat, Clique, Cover-by-vertexes, 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 5.2. Finish the proof by addressing the other problems in Fact 5.1

5.1.1 How to think of NP

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. However, as we saw above this verification step can be cast 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 computation.

Here's a vivid illustration of NP. Suppose I claim that the following matrix contains a 9:

```
5678856563470563470563748056347 6
```

How can you tell, without tediously examining the whole matrix? However, if I tell you that it's in row 10, 8 digits from the right, you can quickly check that I am right. I won't be able to cheat, since you can check my claims. On the other hand I can provide a proof that's easy to verify.

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 we introduced in Chapter 3. Contrary to the analogous question for BPP, cf. section 2.4, the general belief seems to be that P $\neq$ NP. Similarly to BPP, cf. Theorem 2.7, 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 5.2. P $\subseteq$ NP $\subseteq$ Exp $\subseteq$ NExp.

A consequence of the Time Hierarchy Theorem 3.4 is that P $\neq$ Exp. From the inclusions above it follows that

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

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*.

For multi-tape machines, a separation between deterministic and non-deterministic linear time is in [210, 226].

5.2 NP-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 in Theorem 5.2. Problems like 3Sat admitting such reductions deserve a definition.

Definition 5.2. We call a problem L:

NP-hard if every problem in NP reduces to L in P;

NP-complete if it is NP-hard and in NP. To spell it out, this means that $L \in$ NP and moreover for any $M \in$ NP we have $L \in \mathrm{P} \Rightarrow M \in \mathrm{P}$.

One can define NP-hard (and hence NP-complete) w.r.t. different reductions, cf. Chapter 4, and we will do so later. But the simple choice above suffices for now.

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

Fact 5.3. Suppose L is NP-complete. Then $L \in \mathrm{P} \Leftrightarrow \mathrm{P} = \mathrm{NP}$.

Proof. ($\Leftarrow$) This is because $L \in$ NP.

($\Rightarrow$) Let $L' \in$ NP. Because L is NP-hard we know that $L \in$ P$\Rightarrow L' \in$ P. **QED**

Exercise 5.3. Suppose P = NP. Prove that any problem in NP is NP-complete.

Suppose instead P $\neq$ NP. Let $L \in$ NP. Prove L is NP-complete iff $L \notin$ P.

Fact 5.3 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 5.2. 3Sat is NP-complete.

Proof. 3Sat is in NP by Fact 5.1. Next we prove NP-hardness. The main idea is Theorem 5.1, while the rest of the proof mostly amounts to opening up definitions and using some previous simulations. Let $L \in$ NP and let M be the corresponding TM which runs in time n^d on inputs (x, y) where $|x| = n$ and $|y| = n^d$, for some constant d. We can work with TMs instead of RAMs since they are equivalent up to a power loss, as we saw in Theorem 1.8. We can construct in P a circuit $C(x, y)$ of size $c_M n^{c_d}$ such that for any x, y we have $M(x, y) = 1 \Leftrightarrow C(x, y) = 1$ by Theorem 1.5.

Now, suppose we are given an input w for which we are trying to decide membership in L. 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, with no loss in size. Here by "hard-wire" se mean replacing the input gates x with the bits of w. Now we can apply Theorem 5.1 to this new circuit to produce a 3CNF f_w on variables y and new variables z such that $C_w(y) = 1 \Leftrightarrow \exists z : f(y, z) = 1$, for any y. The size of f_w and the number of variables z is power in the size of the circuit.

We have obtained:

$$w \in L \Leftrightarrow \exists y : M(w, y) = 1 \Leftrightarrow \exists y : C_w(y) = 1 \Leftrightarrow \exists y, z : f_w(y, z) = 1 \Leftrightarrow f_w \in \text{3Sat},$$

as desired. **QED**

In section §4.3 we reduced 3Sat to other problems which are also in NP by Fact 5.1. 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 $C \in$ P then $B \in$ P, and so $A \in$ P.

Corollary 5.1. Clique, Cover-by-vertexes, Subset-sum, and 3Color are NP-complete.

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

Exercise 5.4. Consider the problem, given a RAM M, an input x, and $t \in \mathbb{N}$, where t is written in unary, decide if there is $y \in [2]^t$ such that $M(x, y) = 1$ in t steps. Prove that this is NP-complete.

What if t is written in binary?

The interesting aspect of NP-complete problems such as 3Sat and those in Corollary 5.1 is that they are very simple and structured, and don't refer to computational models. This makes them suitable for reductions, and for inferring properties of the complexity class which are not evident from a machine-based definition.

5.3 From RAM 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 RAM computation to circuits, mentioned at the beginning of the chapter. We now remedy this situation and present a quasi-linear reduction. As we did before, cf. Theorem 5.1 and Theorem 5.2, 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 5.3. Given an input length $n \in \mathbb{N}$, a time bound $t \in \mathbb{N}$, and a RAM M that runs in time t on inputs of n bits, we can compute in time $t' := c_M t(\log t)^c$ a 3CNF f on variables (x, y) where $|y| \le t'$ such that for every $x \in [2]^n$:

$$M(x) = 1 \iff \exists y : f(x, y) = 1.$$

We now present the proof of this amazing result. You may also want to refer back to the Definition 1.6 of a RAM. A key concept in the proof is the following "snapshot" of the RAM computation.

Definition 5.3. The *internal configuration,* abbreviated IC, of a RAM specifies:

- its registers,
- the program counter,
- the word length w, and
- if the current instruction is a Read $r_i := \mu[r_j]$ or Write $\mu[r_j] := r_i$ then the IC includes the content $\mu[r_j]$ of the memory cell indexed by r_j.

Note that at most one memory cell is included in one IC. By contrast, the configuration of a TM (Definition 1.1) includes all its tape cells. Also note that an IC has length $\le c_M + c\log t$ bits, where the c_M is for the program counter, and the $c \log t$ is for the rest, using that the maximum word length of a machine running in time $t \ge n$ is $c \log t$.

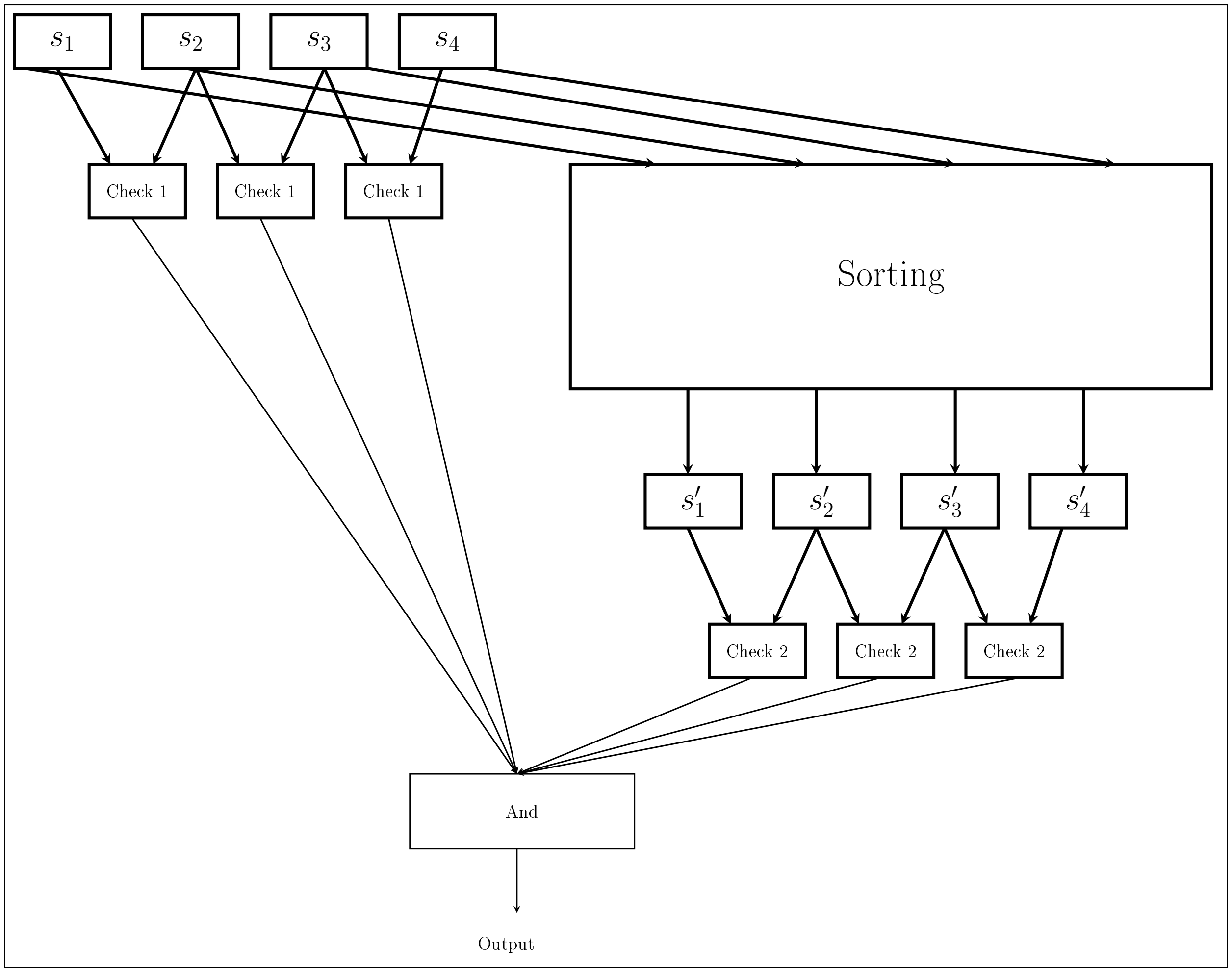


Figure 5.1: Circuit in the proof of Theorem 5.3.

The key idea in the proof. At the high level, the approach is, like in Theorem 5.1, to guess computation and check it efficiently. We are going to *guess* the sequence of ICs, and we need additional ideas to check them efficiently by a circuit. This is not immediate, since, again, the RAM can use direct access to read and write in memory at arbitrary locations, something which is not easy to do with a circuit.

The key idea is to check operations involving memory *independently* from the operations involving registers but not 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 RAM 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 $r_i := \mu[r_j]$ then in s_{i+1} register r_j contains *the most recent value written in memory cell* r_j. In case this cell was never written, then r_j should contain x_j if $j \in \{1, 2, \ldots, n\}$, n if $j = 0$, 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 5.1. It is easy to construct a circuit of quasi-linear size implementing Check 1, since the circuit only has to check adjacent pairs of ICs. As remarked before, these ICs have length $\leq c_M + c \log t$. For fixed i, Check 1 can be implemented by a circuit which depends on the RAM and has size power in the length of an IC. 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 solution is to *sort* the ICs by memory addresses. 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 with a quasi-linear size circuit. Standard sorting algorithms like Mergesort, Heapsort, or Radixsort run in quasi-linear time on a RAM, but rely on direct addressing (cf. section §1.5) and for this reason cannot be easily implemented by a circuit of quasi-linear size. However other algorithms have been developed that do have such an implementation. This gives the following lemma.

Lemma 5.1. 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 5.1 in section §5.3.1.

We summarize the key steps in the proof.

Proof of Theorem 5.3. We construct a circuit C_M as in figure 5.1 (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

$c_M + c\log t$ bits to specify, so we need $\le c_M t \log t$ variables y. The circuit C_M first performs Check (1) above for each adjacent pair (s_i, s_{i+1}) of ICs. This takes size $c_M \log^c t$ for each pair, and so size $c_M t \log^c t$ overall.

Then C_M 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 5.1, using that the memory addresses have $\le c\log t$ 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 is accepting. **QED**

We can now prove completeness in a manner similar to Theorem 5.2, with a relatively simple extension of Theorem 5.3.

Theorem 5.4. Every problem L in NTime(t) map reduces to 3Sat in Time($c_{L,t} t \log^c t$), for every function $t \ge n$ such that $t(x)$ is computable in time $t(x)$ given x.

The assumption on t is similar to that in the hierarchy Theorem 3.4, and is satisfied by all standard functions including all those in this book – cf. discussion after Theorem 3.4.

Proof. Let M be a RAM computing L in the assumed time. Given an input w of length n we have to efficiently compute a 3CNF f such that

$$\exists y \in [2]^{t(n)} : M(w,y) = 1 \iff \exists y \in [2]^{c_{L,t} t(n) \log^c t(n)} : f(y) = 1.$$

First we compute $t(n)$, using the assumption. We now apply Theorem 5.3, but on a new input length $n' := c(n+t) \le ct$, to accommodate for inputs of the form (x,y). This produces a formula f of size $c_{L,t} t (\log t)^c$ in variables (x,y) and new variables z. We can now set x to w and conclude the proof. **QED**

With these sharper results we can now study hardness and completeness within time bounds such as n^2, $n\log^3 n$ etc. We work out an example in the next section.

5.3.1 Efficient sorting circuits: Proof of Lemma 5.1

We present an efficient sorting algorithm for an array $A[n]$ which enjoys the following 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 data,* 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.

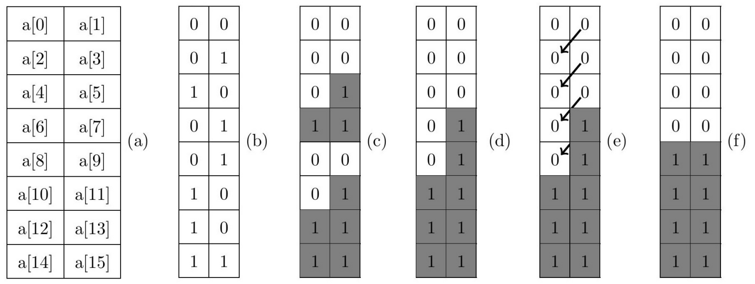


Figure 5.2: Odd-Even-Mergesort

We call an algorithm with this property *oblivious*. Familiar mergesort is not oblivious, because the merge operations performs comparisons which depend on the outcome of previous ones. However a variant of Mergesort [32], called Odd-Even-Mergesort, is oblivious.

Algorithm Odd-Even-Merge(A) merges the two already sorted halves $[a_0, a_1, \ldots, a_{n/2-1}]$ and $[a_{n/2}, a_{n/2+1}, \ldots, a_{n-1}]$ of the sequence $A = [a_0, a_1, \ldots, a_{n-1}]$, resulting in a sorted output sequence. It works in a remarkable and mysterious way. First it merges the *odd* subsequence of the entire array A, then the *even*, and finally it makes $O(n)$ Compare-Exchange-Operations. Throughout, we assume that n is a power of 2.

```
Odd-Even-Merge($A = [a_0, \ldots, a_{(n-1)}]$):
if $n = 2$
 Compare-Exchange($A, 0, 1$)
else {
 Odd-Even-Merge($[a_0, a_2, \ldots, a_{(n-2)}]$, $n/2$) //the even subsequence
 Odd-Even-Merge($[a_1, a_3, \ldots, a_{(n-1)}]$, $n/2$) //the odd subsequence
 for $i \in \{1, 3, 5, 7, \ldots, n-3\}$
  Compare-Exchange($A, i, i+1$)
}
```

We shall now argue that this algorithm is correct.

Lemma 5.2. If $[a_0, a_1, \ldots, a_{n/2-1}]$ and $[a_{n/2}, a_{n/2+1}, \ldots, a_{n-1}]$ are sorted, then Odd-Even-Mergesort($[a_0, a_1, \ldots, a_{n-1}]$) outputs a sorted array.

Proof. To prove this lemma we invoke the so-called "0-1 principle." This principle says that it suffices to prove the lemma when each a_i is either 0 or 1, assuming that the algorithm only accesses the input via Compare-Exchange operations. For completeness we sketch a proof of this principle in this paragraph. Let $A = [a_0, \ldots, a_{n-1}]$ be an input to Odd-Even-Merge, and let $B = [b_0, \ldots, b_{n-1}]$ be the output sequence produced by the algorithm. If the algorithm fails to correctly sort A, then consider the smallest index k such that $b_k > b_{k+1}$. Define a function f such that $f(c) = 1$ if $c \geq b_k$ and $f(c) = 0$ otherwise. For an array $X = [X_0, X_1, \ldots, X_{n-1}]$ let $f(X)$ be the sequence $[f(X_0), f(X_1), \ldots, f(X_{n-1})]$ obtained by applying f to each element of X. Observe that $f(B)$ is not sorted. However it is easy to verify that f commutes with any Compare-Exchange operation applied to any sequence X, i.e.,

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

Because Odd-Even-Merge is just a sequence of Compare-Exchange, we have that

$$f(B) = f(\text{Odd-Even-Merge}(A)) = \text{Odd-Even-Merge}(f(A))$$

and so the algorithm fails to correctly merge the 0-1 sequence $f(A)$. It only remains to notice that $f(A)$ is a valid input for Odd-Even-Merge. This is indeed the case because if a sequence X is sorted then $f(X)$ is also sorted.

We now prove the lemma by induction on n, based on the recursive definition of Odd-Even-Merge. Refer to figure 5.2.

The base case $n = 2$ is clear. Assume that Odd-Even-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 5.2(a)). figure 5.2(b) shows a corresponding 0-1 input, which we can assume w.l.o.g. because of the zero-one principle. figure 5.2(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 Odd-Even-Merge both the left and right (length-($n/2$)) columns are sorted (figure 5.2(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 5.2(e)) yield a sorted sequence (figure 5.2(f)). **QED**

Given Odd-Even-Merge, we can sort by the following algorithm which has the same structure as Mergesort

```
Oblivious-Mergesort($A = [a_0, \ldots, a_{(n-1)}]$):
if $n \geq 2$ {
 Oblivious-Mergesort($[a_0, a_1, \ldots, a_{n/2-1}]$)
 Oblivious-Mergesort($[a_{n/2}, a_{n/2+1}, \ldots, a_{n-1}]$)
 Odd-Even-Merge($[a_0, a_1, \ldots, a_{n-1}]$)
}
```

It only remains to argue efficiency. Let $S_M(n)$ denote the number of Compare-Exchange operations for Odd-Even-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 $S_M(n) = O(n \cdot \log n)$.

Finally, let $S(n)$ denote the number of calls to Compare-Exchange for Oblivious-Mergesort with an input sequence of length n. Then we have the recurrence $S(n) = 2 \cdot S(n/2) + (n \cdot \log n)$, which yields $S(n) = O(n \cdot \log^2 n)$.

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

5.3.2 Quasilinear-time completeness

In this section we use the machinery we just developed to study completeness in quasi-linear time, instead of power time.

Definition 5.4. We define the quasi-linear time complexity classes

$$\text{QLin-Time} := \bigcup_{d \in \mathbb{N}} \text{Time}(n \log^d n) \text{ and}$$
$$\text{QLin-NTime} := \bigcup_{d \in \mathbb{N}} \text{NTime}(n \log^d n).$$

Theorem 5.5. 3Sat is complete for QLin-NTime with respect to mapping reductions in QLin-Time. That is:
- 3Sat is in QLin-NTime, and
- every problem in QLin-NTime map reduces to 3Sat in QLin-Time.

Proof. To show that 3Sat is in QLin-NTime, consider a 3CNF instance f 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 f. 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.

The second part follows from Theorem 5.4, using the fact that the composition of two quasilinear functions is also quasilinear (similarly to the fact that the composition of two power functions is also a power). **QED**

Note that the proof that 3Sat is in QLin-NTime relies on our computational model being a RAM, because we use direct access to fetch the values for the variables in a clause.

We can now give the following quasi-linear version of Fact 5.3. The only extra observation for the proof is again that the composition of two quasi-linear functions is quasi-linear.

Corollary 5.2. 3Sat $\in$ QLin-Time $\Leftrightarrow$ QLin-NTime = QLin-Time.

Exercise 5.5. Prove that Theorem 5.5 holds with 3Color instead of 3Sat. What about Clique and Subset-sum?

Exercise 5.6. Prove that 3Sum reduces to 3Sat in Subquadratic time. That is: 3Sat $\in$ SubquadraticTime$\Rightarrow$ 3Sum $\in$ SubquadraticTime (i.e., Conjecture 4.1 is false).

5.4 Completeness in other classes

The completeness phenomenon is not special to NP but enjoyed by many other classes. In this section we begin to explore completeness for NExp and Exp. One needs to be careful how hardness (and hence completeness) is defined, since these classes are known to be different from P by the hierarchy Theorem 3.4. So defining a problem L to be NExp-hard if $L \in \mathrm{P} \Rightarrow \mathrm{NExp} = \mathrm{P}$ would mean simply that $L \notin \mathrm{P}$. To avoid this in this section hardness (hence completeness) is defined w.r.t. mapping reductions, cf. Chapter 4. (Another option would be to replace P with say BPP, since it is not known if BPP = NExp.)

5.4.1 NExp completeness

Complete problems for NExp include *succinct* versions of problems complete for NP. Here succinct means that rather than giving the input x to the problem in standard format, the input consists instead of a circuit $C : [2]^m \to [2]$ encoding x, for example $C(i)$ equals bit i of x, for every i.

Definition 5.5. The Succinct-3Sat problem: Given a circuit C encoding a 3CNF f_C, does f_C have a satisfying assignment?

Theorem 5.6. Succinct-3Sat is NExp complete with respect to power-time mapping reductions.

Proof sketch.. Let us first show that Succinct-3Sat is in NExp. Given a circuit C of length n, we can run it on every possible input (of length $\le n$) and write down the formula f_C encoded by C. This formula has size $\le 2^n$. We can then use the fact that 3Sat is in NP to decide satisfiability of this formula in non-deterministic power time in 2^n, that is $\mathrm{NTime}(2^{cn}) \subseteq \mathrm{NExp}$.

To prove NExp hardness it is convenient to work with TMs rather than RAMs. The main observation is that in the simulation of a TM M on an input x by a circuit C_M, Theorem 1.5, the circuit is very regular, in the sense that we can construct another circuit S_M which is a succinct encoding of C_M. The circuit S_M is given as input indexes to gates in C_M and outputs the type of the gate and its wires. The size of S_M is power in the index length and M. Thus, if C_M has size t^c, S_M only needs size $\log^c t$. If $t = 2^{n^d}$, S_M has size power in n, as desired. The transformation from circuit to 3CNF in Theorem 5.1 is also regular and can be done succinctly. **QED**

As a consequence, we obtain the following "concrete" problem not in P.

Corollary 5.3. Succinct-3Sat $\notin$ P.

5.4.2 Exp-completeness

Exp-complete problems include several two-player games. The important feature for completeness is that the game may last for an exponential number of steps (otherwise it would belong to a class believed to be stricter which we will investigate in Chapter 7). These games include (generalized versions of) Chess and Checkers.

5.5 Power from completeness

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

5.5.1 Optimization problems

As observed in section §4.6, 3Sat trivially reduces to Max-3Sat. The converse will be shown next.

Theorem 5.7. Max-3Sat reduces to 3Sat in P.

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 f, using binary search and the fact that Atleast-3Sat is in P, we can find in P the largest t s.t. $(f, t) \in$ Atleast-3Sat. Having found this t, there remains to construct an assignment satisfying the clauses. This can be done fixing one variable at the time as in Theorem 4.8. **QED**

5.5.2 NP is as easy as detecting unique solutions

A satisfiable 3CNF can have multiple satisfying assignments. On the other hand some problems and puzzles have unique solutions. In this section we relate these two scenarios.

Definition 5.6. Unique-CktSat is the 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 5.8. 3Sat reduces to Unique-3Sat in BPP.

We in fact reduce 3Sat to Unique-CktSat. Then Unique-CktSat can be reduced to Unique-3Sat observing that the reduction in Theorem 5.1 preserves uniqueness.

The beautiful proof shows how to use hash functions (cf. 2.4) to "isolate" assignments.

Lemma 5.3. 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 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**

Proof of Theorem 5.8. Given a 3Sat instance ϕ with $\leq n$ variables x, we pick a random i from 0 to $n+c$. We then pick a pairwise uniform function mapping $[2]^n$ to $[2]^i$, and consider the circuit

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

This circuit has size n^c.

If ϕ is not satisfiable, C is not satisfiable, for any random choices.

Now suppose that ϕ has $s \geq 1$ satisfying assignment. With prob. $\geq 1/n$ we will have $2^{i-3} \leq s \leq 2^{i-2}$, in which case Lemma 5.3 guarantees that C has a unique satisfying assignment with prob. $\geq c$.

Overall, C has a unique satisfying assignment with prob. $\geq c/n$. Hence the Unique-3Sat algorithm on C outputs 1 with prob. $\geq c/n$. 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**

5.6 Problems

Problem 5.1. In this problem you will explore an alternative proof of the results in section §5.3.

(1) Show Time$(t) \subseteq \exists \cdot c$-TM-Time$(t \log^c t)$. Note the lhs is for RAMs, the rhs is for TMs with c tapes. Hint: Follow the proof structure in section §5.3. Work with plain MergeSort. (In case you are unfamiliar with MergeSort, you might want to review it.)

(2) Prove Theorem 5.3 from (1) using a simulation from Chapter 1.

Problem 5.2. In Theorem 4.8 we reduced Search-3Sat to 3Sat.

(1) Suppose 3Sat is computable by circuits of depth $c \log n$. What would be the depth of the circuits for Search-3Sat given by the reduction?

(2) Reduce Search-3Sat to 3Sat in $\bigcup_{a>0}\text{Depth}(a\log n)$. Hint: First work with randomized circuits. Use ideas in the proof of 5.8. Then explain how to get deterministic circuits.

Note: Depending on how you feel about log-depth circuits, this problem belongs to either Chapter 5 or Chapter 8.

5.7 Notes

NP-completeness and Theorem 5.2 originates in the fundamental works [63, 168]. The first paper proves a version of Theorem 5.1 for TMs, for a more recent and similar exposition see [240]. Theorem 5.3 is from [113, 223]. 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 [32] and following the exposition in [197].

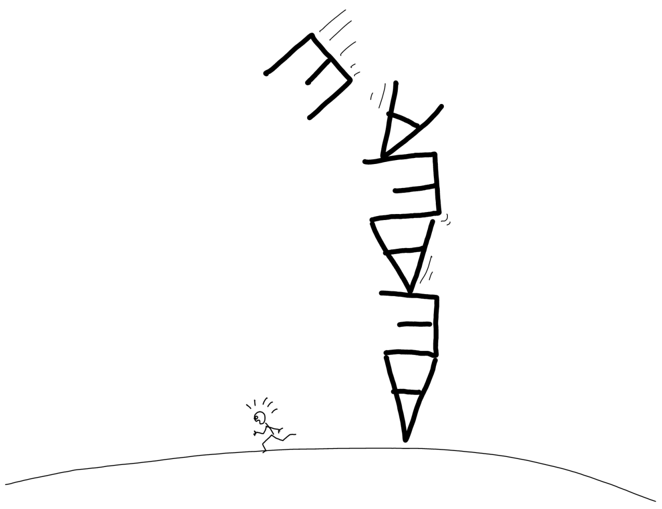
For the Exp-completeness of Chess see [82]; for Checkers [222].

The reduction to unique-3Sat, Theorem 5.8, is from [266] from which we borrowed the section title which, interestingly, emphasizes how easy NP could be, cf. Chapter 17.

Pairwise uniformity was studied as least since [213] and [55]. For background see [262, 125].

Chapter 6

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, space computation, and ACs. In particular, the proofs of several results that do not *prima facie* involve multiple quantifiers will require an excursion to multiple quantifiers (for example, the result that P = NP $\Rightarrow$ P = BPP, Exercise 6.11, or the impossibility results for Sat in section §7.9).

Definition 6.1. $\Sigma_i\text{Time}(t(n))$ is the set of functions $f : X \subseteq [2]^* \to [2]$ for which there is a RAM M such that on input $(x, y_1, y_2, \ldots, y_i)$ stops within $t(|x|)$ steps and

$$f(x) = 1 \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \forall y_4 \ldots Q_i y_i \in [2]^{t(|x|)} : M(x, y_1, y_2, \ldots, y_i) = 1.$$

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

$$\begin{aligned}\Sigma_i\text{P} &:= \bigcup_d \Sigma_i\text{Time}(n^d),\\ \Pi_i\text{P} &:= \bigcup_d \Pi_i\text{Time}(n^d), \text{ and}\\ \text{the power hiearchy PH} &:= \bigcup_i \Sigma_i\text{P} = \bigcup_i \Pi_i\text{P}.\end{aligned}$$

We refer to such computation and the corresponding machines as *alternating,* since they involve alternation of quantifiers and we will soon see a connection with alternating circuits.

The PH is the analog 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.).

As was the case for NP, Definition 5.1, note that the running time of M is a function of $|x|$ only. Again, this difference is inconsequential for Σ_iP, since the composition of two powers is another power. But it is important for a more fine-grained analysis.

Exercise 6.1. Min-Ckt is the problem of deciding if an input circuit has an equivalent circuit which is smaller. It is not known to be in NP. In which of the above classes can you place it?

Exercise 6.2. Show that we can restrict the machine M in Definition 6.1 to read only *one* bit of the input x. The price for this is an extra quantifier, however only over $\log t$ bits. Specifically, show that for every $L \in \Sigma_i$P there exists a RAM M s.t.:

$$\begin{aligned}x \in L \Leftrightarrow &\exists y_1 \in [2]^{t(|x|)} \forall y_2 \in [2]^{t(|x|)} \ldots Q_{i-1} y_{i-1} \in [2]^{t(|x|)}\\ &Q_i(y_i, z) \in [2]^{2t(|x|)} Q_{i+1} y_{i+1} \in [2]^{\log t(|x|)} : M(x, y_1, y_2, \ldots, y_{i+1}) = 1,\end{aligned}$$

and M on input $(x, y_1, y_2, \ldots, y_{i+1})$ stops within $ct(|x|)$ steps and only reads one bit of x. Note the first $i-1$ quantifiers are over t bits and unchanged from Definition 6.1, the next one is over $2t$ bits, written as a pair (y_i, z), and the last is over $\log t$. Hint: The idea is... *you guessed it*!

Exercise 6.3. Prove $\text{P}^{\text{PH}} = \text{PH}$.

6.1 Does the PH collapse?

We refer to the event that $\exists i : \Sigma_i\text{P} = \text{PH}$ as "the PH collapses." It is unknown if the PH collapses. Most people appear to believe that it does not, and to consider statements of the type

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

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

Theorem 6.1. $\mathrm{P} = \mathrm{NP} \Rightarrow \mathrm{P} = \mathrm{PH}$.

The idea in the proof is simply that if you can remove a 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 $L \in \sum_{i+1}\mathrm{P}$, so $\exists a$ and a power-time TM M such that for any $x \in [2]^n$,

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

(As discussed after Definition 6.1 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

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

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

Exercise 6.4. Prove the following strengthening of Theorem 6.1:

$$\bigcup_d \mathrm{NTime}(dn) \subseteq \mathrm{Time}(n^{1+\epsilon}) \Rightarrow \bigcup_d \Sigma_i\mathrm{Time}(dn) \subseteq \mathrm{Time}(n^{1+\epsilon^{c_i}}).$$

Exercise 6.5. Show that if $\Sigma_i\mathrm{P} = \Pi_i\mathrm{P}$ for some i then the PH collapses to $\Sigma_i\mathrm{P}$, that is, $\mathrm{PH} = \Sigma_i\mathrm{P}$.

Theorem 6.2. $\mathrm{NP} \subseteq \mathrm{PCkt} \Rightarrow \mathrm{PH} = \Sigma_2\mathrm{P}$.

Proof. We'll show $\Pi_2\mathrm{P} \subseteq \Sigma_2\mathrm{P}$ and then appeal to Exercise 6.5. Let $f \in \Pi_2\mathrm{Time}(n^d)$ and M be a corresponding machine s.t.

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

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

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

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 the $\Leftarrow$ direction is obvious, by setting $y_2 := C(x, y_1)$. The interesting direction is the $\Rightarrow$. We claim that under the assumption, there is a circuit that given x, y_1 outputs a string y_2 that makes $M(x, y_1, y_2)$ accept, if there is such a string.

To verify this, consider the problems CktSat and Search-CktSat which are analogous to the 3Sat and Search-3Sat problems but for general circuits rather than 3CNF. CktSat is in NP, and so by assumption has power-size circuits. By the reduction in Theorem 4.8, 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 $M(x, y_1, y_2)$, and run S on W. **QED**

Exercise 6.6. Prove that PH $\not\subseteq$ CktGates(n^k), for any $k \in \mathbb{N}$. (Hint: Existentially guess the truth table of a hard function.)

Improve this to Σ_2P $\not\subseteq$ CktGates(n^k).

Exercise 6.7. Prove Exp $\subseteq$ PCkt $\Rightarrow$ Exp $= \Sigma_2$P.

In the next sections we will prove a number of quintessential simulations involving the PH and other classes. A general theme is viewing the simulations as simulations between corresponding *circuit* classes. For example, as suggested by the word alternation in Definition 6.1, the PH and its subclasses correspond to ACs. These circuits have exponential size in the length of the input x. But in fact x plays a limited role and should not be the focus of our attention. Instead, it is natural consider an exponentially longer input (corresponding to the choices for the quantified variables y_i). With respect to this input, the circuits are efficient. The viewpoint of circuits allows us to isolate the core of the proof, and is a necessary condition for a uniform simulation. To obtain the latter, we need the circuit to be sufficiently explicit. There are a few variants of this connection between uniform and circuit classes, depending on what parameters are at premium. The best way to illustrate it all is to dive right into one such simulation.

6.2 BPP in PH

It is not known if BPP is contained in NP. However, we can show that BPP is in PH. More precisely, the following two simulations are known. The first optimizes the number of quantifiers, the second the time. This should be contrasted with various *conditional* results (such as Theorem 2.9) suggesting that in fact a quasilinear deterministic simulation (with no quantifiers) is possible.

Theorem 6.3. For every function t we have:
(1) BPTime$(t) \subseteq \Sigma_2$Time$(t^2 \log^c t)$, and
(2) BPTime$(t) \subseteq \Sigma_3$Time$(t \log^c t)$.

A good way to think of these results is as follows. Fix a BPTime(t) machine M and an input $x \in [2]^n$, and write y for its random bits. 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 a version of the Majority problem, on the exponentially long input

$$(M(x,0), M(x,1), M(x,2), \ldots, M(x,2^t-1)).$$

The alternating machine does not have access to the exponentially-long majority instance, but rather has access to a small circuit $M(x,\cdot)$ s.t. $M(x,y)$ is bit y of the majority instance. The critical aspect is that instances have a *gap*. We define this gap-majority problem next. For convenience we use the letter n to indicate input length. But recall that for proving Theorem 6.3 the input length will be exponential in the running time of the alternating machine.

Definition 6.2. Gap-Maj$_{\alpha,\beta}$ is the problem of deciding if an input $x \in [2]^n$ has weight $|x| \le \alpha n$ or $|x| \ge \beta n$.

As mentioned earlier, it is useful to think of alternating computation as alternating circuits. Indeed, the circuit result that is the starting point of all these simulations is the following somewhat surprising construction of small-depth alternating circuits for Gap-Maj. By contrast, (non-gap) Maj does not have small constant-depth alternating circuits, as we will prove in section §8.5.

Lemma 6.1. Gap-Maj$_{1/3,2/3}(x)$ has alternating circuits of depth 3 and size n^c. Moreover, the gates at distance 1 from the input have fan-in $\le c\log n$.

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* $(\le p, \ge q)$ if $\mathbb{P}_C[C(x)=1] \le p$ and $\mathbb{P}_C[C(y)=1] \ge q$; and we similarly define gives with reverse inequalities. Our goal is to have a distribution that gives

$$(\le 2^{-n}, \ge 1-2^{-n}) \tag{6.1}$$

for every pair $(x,y) \in [2]^n \times [2]^n$ where $|x| \le n/3$ and $|y| \ge 2n/3$. 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

$$\left(\le p, \ge p\cdot n^2\right).$$

Now consider the distribution $C_\vee$ obtained by complementing the circuits in $C_\wedge$. This gives

$$\left(\ge 1-p, \le 1-p\cdot n^2\right).$$

Next consider the distribution $C_{\wedge\vee}$ obtained by taking the And of $m := p^{-1}/n$ independent samples of $C_\vee$. This gives

$$\left(\ge (1-p)^m, \le (1-p\cdot n^2)^m\right).$$

Approximations for the exponential function, Fact A.4, 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.5; 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 (6.1). The bounds on the number of gates and the fan-in holds by inspection. **QED**

Exercise 6.8. Prove Gap-Maj$_{1/2-1/\sqrt{\log n},1/2+1/\sqrt{\log n}}$ has alternating circuits of depth c and size n^c. Hint: First prove it for Gap-Maj$_{1/10,9/10}$. Think of the input string as the outcomes of a BPP algorithm with error $1/10$ for various choices of the randomness. How do we reduce the error probability?

To prove Theorem 6.3 we "only" need the circuits for Gap-Maj in Lemma 6.1 to be sufficiently explicit. Perhaps the simplest notion of explicitness is that the circuit is constructible in power-time in its description. This does not work here because the circuit has exponential size in the input length. Instead, we need a refine notion of explicitness, arguably even more natural. We require that the child of a gate can be computed efficiently given the description of the gate.

Lemma 6.2. Assume the circuit in Lemma 6.1 is explicit in the following sense: Given an index to a gate g of fan-in h and a number $i \leq h$ we can compute the index of child i of g in linear time. Prove Theorem 6.3.

Proof. In time t the machine uses $t' \leq ct \log t$ random bit. Consider the circuit on $T := 2^{t'}$ inputs. To prove (1), use two quantifiers to index an And gate next to the input. This gate has fan-in $h \leq ct'$. For each $i \leq h$, compute child i of the gate, which is just an index to an input bit, which in turn is 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 6.9. Prove (2).

There remains to construct explicit circuits for Gap-Maj. We give a construction which has worse parameters than Lemma 6.1 but is simple and suffices for (1) in Theorem 6.3. The idea is that if the input weight of x is large, 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 6.3. Let $r := \log n$. The following circuit solves $\mathrm{GapMaj}_{1/r^2,1-1/r^2}$ 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. Assume $|x| \le n/r^2$. Each shift s_i contributes at most n/r^2 ones. Hence all the r shifts contribute $\le n/r$ ones, and we do not cover every bit i.

Now assume $|x| \ge n(1 - 1/r^2)$. 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:

$$\begin{aligned}
&\mathbb{P}_{s_1,s_2,\ldots,s_r}[\exists i \in [2]^r : \forall j \in \{1, 2, \ldots, r\} : x_{i\oplus s_j} = 0] \\
\le &\sum_{i\in[2]^r} \mathbb{P}_{s_1,s_2,\ldots,s_r}[\forall j \in \{1, 2, \ldots, r\} : x_{i\oplus s_j} = 0] && \text{(union bound)} \\
= &\sum_{i\in[2]^r} \mathbb{P}_s[x_{i\oplus s} = 0]^r && \text{(independence of the } s_i\text{)} \\
\le &\sum_{i\in[2]^r} (1/r^2)^r && \text{(by assumption on } |x|\text{)} \\
\le &(2/r^2)^r \\
< &1,
\end{aligned}$$

as desired. **QED**

Exercise 6.10. Prove (1) in Theorem 6.3.

Lemma 6.3 is not sufficient for (2) in Theorem 6.3. One can prove (2) by *derandomizing* the shifts in Lemma 6.3. 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 6.3.). This is done in section 11.1.4.

Exercise 6.11. Prove:
(1) $\mathrm{P} = \mathrm{NP} \Rightarrow \mathrm{P} = \mathrm{BPP}$.
(2) $\Sigma_2\mathrm{P} \subseteq \mathrm{BPP} \Rightarrow \mathrm{PH}$ collapses.

6.3 The quantifier calculus

We have extended P with $\exists$ and $\forall$ quantifiers. We have also extended it with randomness to obtain BPP. As alluded to before, we can also think of BPP as a quantifier BP applied to

P. The Unique-3Sat problem (Theorem 5.8) also points to a new quantifier, "exists unique." We now develop a general calculus of quantifiers, and examine fundamental relationships between then. For simplicity, we only consider power-time, total computation.

Definition 6.3. Let C be a class of functions mapping $[2]^* \to [2]$. We define $L \in \mathrm{Op} \cdot C$ if there is $L' \in C$ and $d \in \mathbb{N}$ such that

- $\mathrm{Op} = \mathrm{Maj}$

$$x \in L \Leftrightarrow \mathbb{P}_{y \in [2]^{|x|^d}}[(x,y) \in L'] \geq 1/2.$$

- $\mathrm{Op} = \mathrm{BP}$

$$x \in L \Rightarrow \mathbb{P}_{y \in [2]^{|x|^d}}[(x,y) \in L'] \geq 2/3,$$
$$x \notin L \Rightarrow \mathbb{P}_{y \in [2]^{|x|^d}}[(x,y) \in L'] \leq 1/3.$$

- $\mathrm{Op} = \oplus$ (read: parity)

$$x \in L \Leftrightarrow \text{there is an odd number of } y \in [2]^{|x|^d} : (x,y) \in L'.$$

- $\mathrm{Op} = \exists$

$$x \in L \Leftrightarrow \exists y \in [2]^{|x|^d} : (x,y) \in L'.$$

- $\mathrm{Op} = \forall$

$$x \in L \Leftrightarrow \forall y \in [2]^{|x|^d} : (x,y) \in L'.$$

With this notation we have: $\mathrm{NP} = \exists \cdot \mathrm{P}$, $\mathrm{BPP} = \mathrm{BP} \cdot \mathrm{P}$, $\Sigma_2\mathrm{P} = \exists \cdot \forall \cdot \mathrm{P}$.

More generally, we might be interested in computing the number of y s.t. $(x,y) \in L'$.

Definition 6.4. Let C be a class of functions mapping $[2]^* \to [2]$. We say that $f \in \# \cdot C$ (pronounced sharp C or number C) if there is $L' \in C$ and $d \in \mathbb{N}$ such that $f(x)$ is the number of $y \in [2]^{|x|^d}$ for which $(x,y) \in L'$.

6.4 PH is a random low-degree polynomial

In this section we prove the following result.

Theorem 6.4. $\mathrm{PH} \subseteq \mathrm{BP} \cdot \oplus \cdot \mathrm{P}$.

This is saying that any constant number of $\exists$ and $\forall$ quantifier can be replaced by a BP quantifier followed by a $\oplus$ quantifier. Let's see what this has to do with the title of this section. Where is the polynomial? Consider polynomials over $\mathbb{F}_2 = [2]$. Recall that such a polynomial over n bits 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 $[2]$ 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.

Example 6.1. The And function on n bits can be written as the polynomial

$$\text{And}(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} x_i.$$

The Or function on n bits can be written as the polynomial

$$\text{Or}(x_1, x_2, \ldots, x_n) = 1 + \text{And}(1 + x_1, 1 + x_2, \ldots, 1 + x_n) = 1 + \prod_{i=1}^{n} (1 + x_i).$$

For $n = 2$ we have

$$\text{Or}(x_1, x_2) = x_1 + x_2 + x_1 \cdot x_2.$$

The polynomial corresponding to a PH computation will have an exponential number of terms, so we can't write it down. The big sum over all its monomials corresponds to the $\oplus$ in Theorem 6.4. The polynomial will be sufficiently explicit: we will be able to compute each of its monomials in P. Finally, there won't be just one polynomial, but we will have a distribution on polynomials, and that's the BP part.

Confusing? Like before, a good way to look at this result is in terms of circuits. We state the circuit result behind Theorem 6.4 after a definition. The result is of independent interest and will be useful later in section §8.5.

Definition 6.5. A distribution P on polynomials computes a function $f : [2]^n \to [2]$ with error ϵ if for every x we have

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

Theorem 6.5. Let $C : [2]^n \to [2]$ be an alternating circuit of depth d and size s. Then there is a distribution P on polynomials over $\mathbb{F}_2$ of degree $\log^{d-1} s/\epsilon$ that computes C with error ϵ.

Ultimately we only need constant error, but the construction requires small error. Jumping ahead, this is because we construct distributions for each gate separately, and we need the error to be small enough for a union bound over all gates in the circuit.

The important point in Theorem 6.5 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 power-size alternating circuits of constant depth the degree is power logarithmic for constant error.

Let us slowly illustrate the ideas behind Theorem 6.5 starting with the simplest case: C is just a single Or gate on n bits.

Lemma 6.4. 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.

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 6.1. Hence the final degree is $t = \log 1/\epsilon$. **QED**

Exercise 6.12. Obtain the same result for $C = \mathrm{And}$.

Now we would like to handle general circuits which have any number of And and Or gates. As mentioned earlier, we apply the construction above to every gate, and compose the polynomials. We pick the error at each gate small enough so that we can do a union bound over all gates.

Proof of Theorem 6.5. We apply Lemma 6.4 to every gate in the circuit with error ϵ/s. 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 $\log s/\epsilon$, and we compose $d-1$ times, the final degree is $\log^{d-1} s/\epsilon$. **QED**

6.4.1 Back to PH

We have proved Theorem 6.5 which is a circuit analogue of Theorem 6.4. We now go back to the PH. As in Lemma 6.2, let us first identify how explicit the polynomial needs to be to yield Theorem 6.4. We need to be able to compute monomials efficiently given its index.

Lemma 6.5. Suppose that for every $d \in \mathbb{N}$ we have:

Let C be the AC of depth d where each gate has fan-in 2^{n^d} and the circuit is a tree. Suppose there is a distribution P on polynomials as in Theorem 6.5 but that moreover:

(1) Can be sampled from n^{c_d} random bits r, we write P_r for the (fixed) polynomial given by r, and

(2) Given r and an index i of n^{c_d} we can compute the (coefficient of) monomial i of P_r in time n^{c_d}.

Then Theorem 6.4 follows.

Proof. Let $L \in \Sigma_d\mathrm{P}$. Consider the corresponding alternating circuit C. Similarly to section §6.2, the input consists of the bits

$$M(x, y_1, y_2, \ldots, y_d)$$

over all values of the quantified variables y_i. We use the BP quantifier to select r, and then the $\oplus$ quantifier to pick a monomial. This monomial will correspond to n^{c_d} bits as above. We evaluate each of them and return the result. **QED**

There remains to construct explicit polynomials. Again, this is similar to the way we proceeded in section §6.2: After a non-explicit construction (Lemma 6.1) we then obtained an explicit construction (Lemma 6.3). Though note here we still aim for a distribution.

Let us go back to the simplest case of Or. Recall that the basic building block in the proof of Lemma 6.4 was the construction of a distribution p_A on linear polynomials which are zero on the all-zero input (which just means that they do not have constant terms), and are often non-zero on any non-zero input. We introduce a definition, since now we will have several constructions with different parameters.

Definition 6.6. A distribution p_A on linear polynomials with no constant term has the Or property if $\mathbb{P}_A[p_A(x) = 1] \geq 1/3$ for any $x \neq 0$. We identify p_A with the n bits A corresponding to its coefficients.

The next lemma shows that we can compute distributions on linear polynomials with the Or property from a seed of just $\log n + c$ bits, as opposed to the n bits that were used for A in the proof of Lemma 6.4. This important fact is generalized and put in context in section 11.1.3 (the Or property is a special case of fooling degree-1 polynomials, and the constructions actually establish the latter). Recall that for our application to Lemma 6.4 the polynomials have an exponential number of monomials and so we cannot afford to write them down. Instead we shall guarantee that given a seed r and an index to a monomial we can compute the monomial via a function f in P. In this linear case, for a polynomial in n variables we have $\leq n$ monomials x_i. So the function f takes as input r and a number $i \leq n$ and outputs the coefficient to x_i.

Lemma 6.6. Given n, $i \leq n$, and $r \in [2]^{2\log n + c}$ we can compute in P a function $f(r, i)$ such that for uniform $R \in [2]^{2\log n + c}$ the distribution

$$(f(R, 1), f(R, 2), \ldots, f(R, n))$$

has the Or property.

Proof. Let $q := 2^{\log n + c}$ and identify the field $\mathbb{F}_q$ with bit strings of length $\log q$. We view r as a pair $(s,t) \in (\mathbb{F}_q)^2$. Then we define

$$f((s,t),i) := \langle s^i, t\rangle$$

where s^i is exponentiation in $\mathbb{F}_q$ and $\langle .,. \rangle : (\mathbb{F}_q)^2 \to [2]$ is defined as $\langle u,v\rangle := \sum u_i \cdot v_i$ over $\mathbb{F}_2$.

To show that this has the Or property, pick any non-zero $x \in [2]^n$. We have to show that

$$p := \mathbb{P}_{S,T}[\sum_i \langle S^i, T\rangle x_i = 1] \geq 1/3.$$

The critical step is to note that

$$\sum_i \langle S^i, T\rangle x_i = \sum_i \langle x_i \cdot S^i, T\rangle = \langle \sum_i x_i \cdot S^i, T\rangle.$$

Now, if $x \neq 0$, then the probability over S that $\sum_i x_i \cdot S^i = 0$ is $\leq n/q \leq 1/6$. 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 so the bound follows by Lemma 2.1.

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. Hence our target probability p above satisfies

$$p \geq 1/2 - 1/6$$

as desired. **QED**

Exercise 6.13. Give an alternative construction of a distribution with the Or property following this guideline.

(1) Satisfy the Or property for every input x with weight 1. Note: This can be done with a seed of length 0 (i.e., deterministically), but when solving the next items you might want to get back to this and use a seed of length c to achieve a stronger property.)

(2) For any j, satisfy the Or property for every input x with weight between 2^j and 2^{j+1}, with a seed of length $c \log n$. Use Lemma 5.3.

(3) Combine (2) with various j to satisfy the Or property for every input.

(4) State the seed length for your distribution and compare it to that of Lemma 6.6.

With this in hand, we can now reduce the error in the same way we reduced it in the proof of Lemma 6.4.

Lemma 6.7. Given n, a seed $r \in [2]^{c \log(1/\epsilon) \log n}$, and $m \leq n^{c \log 1/\epsilon}$ we can compute in P a monomial $X_{r,m}$ of degree $c \log 1/\epsilon$ such that the distribution

$$\sum_m X_{R,m}$$

for uniform R computes Or on n variables with error ϵ.

Proof. We use the construction

$$p_{A_1,A_2,\ldots,A_t}(x) := p_{A_1}(x) \vee p_{A_2}(x) \vee \cdots \vee p_{A_t}(x)$$

from the proof of Lemma 6.4, except that each A_i is now generated via Lemma 6.4, and that $t = c\log 1/\epsilon$ (as opposed to $t = \log 1/\epsilon$ before). The bound on the degree is the same as before, as is the proof that it computes Or: The error will be $(1/3)^{c\log 1/\epsilon} \le \epsilon$.

There remains to show that the monomials can be computed in P. For this we can go back to the polynomial for Or in Example 6.1. Plugging that gives

$$p_{A_1,A_2,\ldots,A_t}(x) = \sum_{a\in[2]^t:a\neq 0} \prod_{i\le t}(p_{A_i}(x) + a_i + 1).$$

We can use m to index a choice of a and then a choice for a monomial in each of the t linear factors $p_{A_i}(x) + a_i + 1$. For each factor we can use Lemma 6.6 to compute the monomials. **QED**

We can now compose these polynomials at each gate to obtain an explicit version of Theorem 6.5 which suffices to prove Lemma 6.5 and hence Theorem 6.4.

Claim 6.1. The assumption of Lemma 6.5 is true.

Proof. Replace each gate with the distribution on polynomials given by Lemma 6.7. (Lemma 6.7 only covers Or gates, but And gates are similar, cf. Exercise 6.12.) The desired polynomial is obtained by composing all these polynomials.

Note each of these polynomials is on $2^{n^{c_d}}$variables and we can set the parameter 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 $\le n^{c_d}$.

The seed used to sample the polynomials is re-used across all gates. 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}, giving (1). Degrees multiply as in Theorem 6.5; so the final degree is also n^{c_d}.

It remains to show (2), that is, that we can evaluate the polynomial. We are going to show how we can compute the monomials of the composed polynomial in the same way as we computed monomials in Lemma 6.7. It amounts to parsing the index to the monomial in the natrual way. Some details: Start at the output gate. We use n^{c_d} bits in the given index to choose a monomial in the corresponding polynomial. We write down this monomial, using Lemma 6.7. This monomial is over the $2^{n^{c_d}}$ variables $z_1, z_2, \ldots$ corresponding to the children of the output gate, and as remarked has degree $\le n^{c_d}$. 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}.

We continue in this way until we have monomials just in the bits $M(x, y_1, y_2, \ldots, y_d)$. Those bits can then be computed from x in P running M. **QED**

6.5 Counting

In this section we prove:

Theorem 6.6. $\mathrm{BP} \cdot \oplus \cdot \mathrm{P} \subseteq \mathrm{P}^{\# \cdot \mathrm{P}}$.

Here the right-hand side can be defined as the union of P^f for $f \in \# \cdot \mathrm{P}$. But in fact, the function will have a simple form, making only one oracle query. Combined with Theorem 6.4 we obtain:

Corollary 6.1. $\mathrm{PH} \subseteq \mathrm{P}^{\# \cdot \mathrm{P}}$.

Again, a good way to think of Theorem 6.6 is in terms of circuits, and polynomials. The key is the construction *of modulus-amplifying polynomials*, which allow us to treat the parity as an integer sum and merge it with the BP operator.

Lemma 6.8. [Modulus-amplifying polynomials] For every integer ℓ there is a univariate polynomial F_ℓ of degree $2\ell - 1$ over the integers such that for every input integer y:
- $y \equiv 0 \mod 2 \Rightarrow F_\ell(y) \equiv 0 \mod 2^\ell$, and
- $y \equiv 1 \mod 2 \Rightarrow F_\ell(y) \equiv 1 \mod 2^\ell$.

Given an index to a monomial its coefficient in $[2^\ell]$ can be computed in time ℓ^c.

Proof of Theorem 6.6. Let L be a language in the LHS. We set $\ell = n^{c_L}$. Fix an input x, it suffices to compute a sum

$$\sum_{i \in [2^\ell]} (p_i \mod 2) \tag{6.2}$$

where the p_i are linear polynomials over the integers, over $N = 2^{n^{c_L}}$ input bits.

Lemma 6.8 allows us to write that sum as

$$\left(\sum_{i \in [2^\ell]} F_\ell(p_i) \right) \mod 2^\ell.$$

Indeed, in the last expression we can move the $\mod$ inside, and then apply Lemma 6.8. Now write F_ℓ as a sum of monomials: $F_\ell = \sum_j r_j$. The latter sum can be merged with the one over i, sum over all q_i, and so we obtain

$$\left(\sum_{i \in [2^\ell]} \sum_{j \in [2^\ell]} r_j(p_i) \right) \mod 2^\ell.$$

Now r_j has degree $\le n^{c_L}$. We can expand $r_j(p_i)$ into a sum of $\le 2^{n^{c_L}}$ monomials of power degree, and the result follows. **QED**

Exercise 6.14. Prove Lemma 6.8 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.

Exercise 6.15. Prove:

1. $\text{BPP} \cdot \text{P} \subseteq \text{Maj} \cdot \text{P}$.
2. $\{(M, x, t) :$ the number of $y \in [2]^{|t|}$ such that $M(x, y) = 1$ is $\geq t\} \in \text{Maj} \cdot \text{P}$. Note that the binary representation of t is allowed to have leading zeroes, so $t = 1$ and $|t| = 2^{100}$ is possible. This is just a convenience to avoid throwing in another parameter. If it confuses you, consider the language $\{(M, x, s, t) :$ the number of $y \in [2]^{|s|}$ such that $M(x, y) = 1$ is $\geq t\}$ instead.
3. The same as 2. but with $\geq$ replaced by $\leq$.
4. $\text{NP} \subseteq \text{Maj} \cdot \text{P}$.
5. $\text{P}^{\text{Maj}\cdot\text{P}} = \text{P}^{\#\cdot\text{P}}$

It is not known if NP has linear-size circuits. We saw in Exercise 6.6 that PH does not have circuits of size n^k, for any k. By Exercise 6.15 this holds for $\text{Maj} \cdot \text{Maj} \cdot \text{P}$ as well. The following result improves this $\text{Maj} \cdot \text{P}$. It is particularly interesting because it cannot be established using a well-studied class of techniques which includes all the results about PH we have encountered so far, specifically black-box techniques discussed in section §16.1.

Theorem 6.7. $\text{Maj} \cdot \text{P} \not\subseteq \text{CktGates}(n^k)$, for any $k \in \mathbb{N}$.

The proof is in section §10.4.

6.6 Problems

Problem 6.1. Prove $\Sigma_i\text{Time}(n^\alpha) \not\subseteq \Pi_i\text{Time}(n^\beta)$, for any constants i and any $\alpha > \beta$.

Feel free to work with partial functions and assume $\beta > 1$ if you prefer, because these details are not the main point of the problem.

Problem 6.2. Prove that QLin-NTime = QLin-Time $\Rightarrow$ QLin-BPTime = QLin-Time. (Cf. Definition 5.4, the definition of QLin-BPTime is analogous.)

Problem 6.3. [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 $f' \in \text{PH}$.

Problem 6.4. Prove Theorem 6.5 with the bound on the degree replaced by $(\log^{d-1} cs)c\log 1/\epsilon$ (which is better when ϵ is small).

Problem 6.5. (1) Show that for any constants $\alpha < \beta$, Gap-Maj$_{\alpha,\beta}$ has alternating circuits of depth $c_{\alpha,\beta}$ and size $n^{c_{\alpha,\beta}}$.

(2) Show that Gap-Maj$_{1/2-1/\log n,1/2+1/\log n}$ has alternating circuits of depth c and size n^c. (This can be generalized by replacing the log with $\log^a$ and the c with c_a, but you are not asked to prove this.)

Problem 6.6. Prove $\exists \cdot \text{BP} \cdot \text{P} \subseteq \text{Maj} \cdot \text{P}$.

Problem 6.7. $\text{Maj} \cdot \oplus \cdot\text{P} \subseteq \text{Maj} \cdot \text{Maj} \cdot \text{P}$. (In particular, $\text{PH} \subseteq \text{Maj} \cdot \text{Maj} \cdot \text{P}$ by Theorem 6.4.)

(Hint: Suppose you have an integer w in some range and you want to know if w is odd by just asking questions of the type $w \geq t$ and $w \leq t'$, for various t, t'. You want that the number of questions with answer "yes" only depends on whether w is odd or even.)

Problem 6.8. [Or property vs. error-correcting codes] Actually, explicit constructions of distributions with the Or property (Definition 6.6), or fooling degree-1 polynomials (section 11.1.3), are *equivalent* to error-correcting codes. (The main novelty in theoretical computer science seems the level of explicitness that's typically required in applications.) In this problem you will explore this connection.

A set $C \subseteq [2]^n$ of size 2^k is called *linear* if there exists an $n \times k$ matrix M over $\mathbb{F}_2$ such that $C = \{Mx : x \in [2]^k\}$. Recall error-correcting codes from Exercise 2.12.

1. Prove that a linear set C is an error-correcting code iff the weight of any non-zero string in C is at least $n/3$.

2. Prove the existence of linear error-correcting codes matching the parameters in Exercise 2.12.

3. Let S be a subset of $[2]^k$ s.t. the uniform distribution over S has the Or property. Define $|S| \times k$ matrix M_S where the rows are the elements of S. Prove that $\{M_S x : x \in [2]^k\}$ is an error-correcting code.

4. Give explicit error-correcting codes over ck^2 bits of size 2^k.

5. This motivates improving the parameters of distributions with the Or property. Improve the seed length in Lemma 6.6 to $\log n + c \log\log n$. Hint: What property you need from T?

6. Give explicit error-correcting codes over $k \log^c k$ bits of size 2^k.

6.7 Notes

Two lines of research appear intertwined around few main ideas. The first line is the study of complexity classes defined in terms of operators. The second is that of circuits with various gates. One basic idea is that And (hence, AC) can be approximated by low-degree polynomials. This idea appears in [266] and [216]. Another basic idea is that of modulus amplifying polynomials. They originate from [257] and were studied further in [297, 42], the latter proving Lemma 6.8. A similar proof gives a simulation of ACC (Lemma 8.2). A third basic idea is that there are ACs for gap majority. This is from [7] (where Lemma 6.1 is proved) and [239]. For more constructions, see [275] and [8].

Several proofs of Theorem 6.4 have appeared [149, 79]. Our presentation based on Theorem 6.5 seems a little different ([149, 79] don't cite [216]).

The PH was identified in [247], where Theorem 6.1 is also proved. Theorem 6.2 is from [152]. The study of ACs got a boost from the connection with oracle separations for PH, which was pointed out in [86].

Theorem 6.4 and Theorem 6.6 are from [257].

The first item in the simulation of BPP Theorem 6.3 is from [239]; the second is from [275].

Maj P was defined in [136].

Parity P was defined in [207].

Theorem 6.5 is from [216].

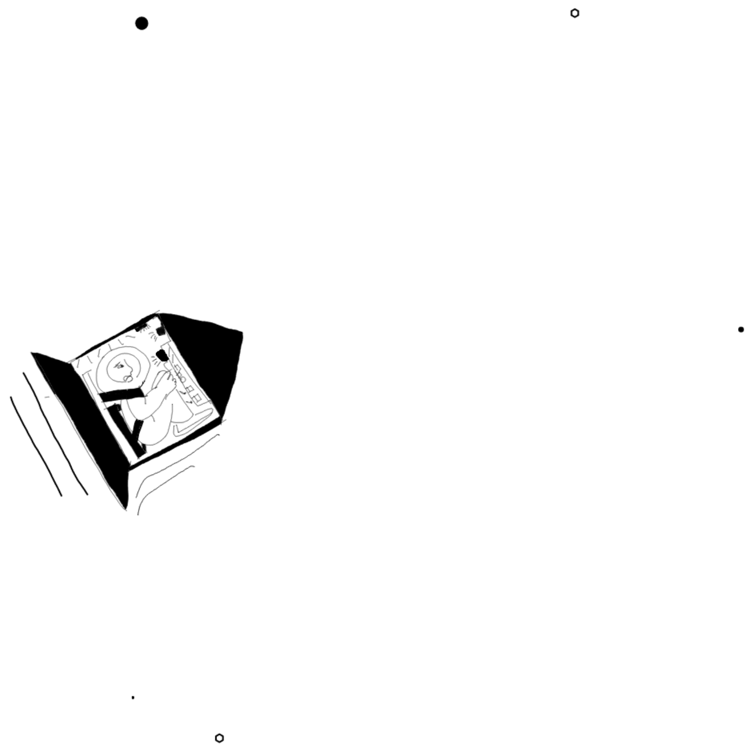
Lemma 6.6 is from [192], where they actually prove stronger results, see Theorem 11.5. The proof we presented is a particularly nice one from the three presented in the follow-up [16].

Theorem 6.7 is from [270].

For a compendium of problems complete for various levels of the PH, in the style of [91], see [228].

Chapter 7

Space



As mentioned in Chapter 1, Time is only one of several resources we with to study. Another important one is *space*, which is another name for the *memory* of the machine. Computing under space constraints may be less familiar to us than computing under time constraints, and many surprises lay ahead in this chapter that challenge our intuition of efficient computation.

If only space is under consideration, and one is OK with a constant-factor slackness, then TMs and RAMs are equivalent; much like P is invariant under power changes in time. In a sense, changing the space by a constant factor is like changing the time by a power; from this point of view the equivalence is not surprising.

We shall consider both space bounds bigger than the input length and smaller. For the latter, we have to consider the input separately. The machine should be able to *read* the input, but not *write* on its cells. One way to formalize this is to consider 2TMs, where one tape holds the input and is *read-only.* The other is a standard *read-write tape.*

We also want to compute functions f whose output is more than 1 bit. One option is to equip the machine with yet another tape, which is *write-only.* We prefer to stick to two tapes and instead require that given x, i the i output bit of $f(x)$ is computable efficiently.

Definition 7.1. A function $f : X \subseteq [2]^* \to [2]^*$ is computable in Space$(s(n))$ if there is a 2TM which on input (x, i) on the first tape, where $x \in X$ and $i \leq |f(x)|$, outputs the i bit of $f(x)$, and the machine never writes on the first tape and never uses more than $s(|x|)$ cells on the second.

We define:

$$\begin{aligned} \mathrm{L} :=& \bigcup_d \mathrm{Space}(d \log n), \\ \mathrm{PSpace} :=& \bigcup_d \mathrm{Space}(n^d). \end{aligned}$$

We investigate next the relationship between space and time. We begin with some basic simulations, the first two of which will be improved right after.

Theorem 7.1. For every functions t and s:

1. kTM-Time$(t) \subseteq$ Space$(c_k t)$,
2. Time$(t) \subseteq$ Space$(ct \log(nt))$,
3. Space$(s) \subseteq$ 2TM-Time$(c^{s(n)} \cdot n^c)$.

Proof. 1. A TM running in time t can only move its heads at most t tape cells. We can write all these contents in one tape. To simulate one step of the kTM we do a pass on all the contents and update them accordingly.

2. A RAM running in time t can only access $\leq t$ memory cells, each containing at most $c \log nt$ bits; the factor n is to take into account that the machine starts with word length $\geq \log n$. We simulate this machine and for each Write operation we add a record on the

tape with the memory cell index and the content, similar to Theorem 1.8. When the RAM reads from memory, we scan the records and read off the memory values from one of them. If the record isn't found, then the simulation returns the value corresponding to the initial configuration.

We also allocate $c \log nt$ bits for the registers of the RAM. It can be shown that the operations among them can be performed in the desired space, since they only involve a logarithmic number of bits. A stronger result is proved later in Theorem 7.4.

3. On an input x with $|x| = n$, a Space(s) machine can be in at most $n^c \cdot c^{s(n)}$ configurations. The first n^c factor is for the head position on the input. The second factor is for the contents of the second tape. Since the machine ultimately stops, configurations cannot repeat, hence the same machine (with no modification) will stop in the desired time. **QED**

For simulating time by space we actually have the following stronger results.

Theorem 7.2. For every functions t and s:
(1) kTM-Time(t) $\subseteq$ Space($c_k\sqrt{t \log t}$),
(2) Time(t) $\subseteq$ Space($ct/\log t$).

Exercise 7.1. Prove (1) for 1TM (i.e., $k = 1$). In fact, obtain the stronger space bound $c\sqrt{t}$.

Proof ideas of the other claims in Theorem 7.2. We only give a sketch of the beautiful proofs which use several results, including some we will prove later. The proof of (1) for kTM is more involved than the proof for 1TMs (cf Exercise 7.1). Under the assumption that $t \geq n^2$, one can prove it by combining Theorem 1.3 and Theorem 9.3 in a relatively simple way.

We sketch a different argument that gives the more modest $\log t$ saving in (2), but avoids Theorem 9.3 and can be extended to RAMs. Let $b := t^c$. Divide each tape into consecutive blocks of $b = t^c$ symbols, and assume that the machine is *block-respecting*, cf. 1.3.

We also divide time into t/b *epochs* of length b. We construct a graph with t/b nodes corresponding to epochs. We are going to place *pebbles* on this graph, where placing a pebble on node v means that we can simulate the machine up to until the end of epoch v, which in turn means computing the state of the machine, which blocks it is working on, and their contents. The rules for pebbling are that we can place a pebble on any node whose predecessors are all pebbled (in particular, nodes with no predecessors), and remove a pebble from any node. We now have to place edges on this graph ensuring our simulation is indeed possible. We have edges $i - 1 \to i$ for every i, because we need to know the state of the machine and the block positions to continue. In addition, we place an edge $i \to j$ if the machine in time block j is working on the same block the machine was working in time block i, and i is the largest index less than j with this property. Note this graph depends on the input x.

It is known that any graph with m nodes can be pebbled using $\leq cm/\log m$ pebbles (which means that every node v that's reachable from a node u can be pebbled starting with a pebble on node u with that many pebbles). Problem 7.1 asks to prove a slightly weaker

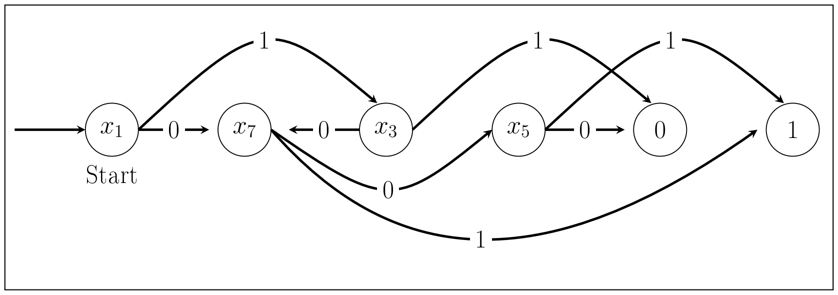


Figure 7.1: Illustration of branching program (Definition 7.2).

bound which suffices to prove the theorem up to a $c \log\log t$ factor, and has a significantly simpler proof. Here $m = t/b$. Moreover, each pebble costs cb bits of space during the simulation. So the total space of the simulation is $m \cdot cb = ct/\log t$.

There remains to compute the pebbling. This is a non-trivial task, since the pebbling can involve an exponential number of moves, but we can use a recursive strategy to solve the following problem (similar to Theorem 7.19): Given a current pebbling of the graph, compute the next move in an optimal pebbling. This can be done in space roughly square the size of the pebbling, which will be within our budget by our choice of b. **QED**

From Theorem 7.1 and the next exercise we have

$$\mathrm{L} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PH} \subseteq \mathrm{PSpace} \subseteq \mathrm{Exp}.$$

Exercise 7.2. Prove PH $\subseteq$ PSpace.

Just like for Time, for space one has universal machines and a hierarchy theorem. The latter 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, but nobody can prove that any specific one is.

7.1 Branching programs

Branching programs are the non-uniform counterpart of Space, just like circuits are the non-uniform counterpart of Time.

Definition 7.2. A *(branching) program* is a directed acyclic graph. A node can be labeled with an input variable, in which case it has two outgoing edges labeled 0 and 1. Alternatively a node can be labeled with 0 or 1, in which case it has no outgoing edges. One special node is the *start* node.

The *space* of the program with S nodes is $\log S$. A program computes a function $f : [2]^n \to [2]$ by following the path from the starting node, following edge labels corresponding to the input, and outputting $b \in [2]$ as soon as it reaches a node labeled b.

See figure 7.1 for an illustration.

Theorem 7.3. Suppose a 2TM M computes $f : [2]^n \to [2]$ in space s. Then f has branching programs of space $c_M(s + \log n)$. In particular, any $f \in$ L has branching programs of size n^{c_f}.

Proof. Each node in the program corresponds to a configuration. **QED**

Definition 7.3. The branching program given by Theorem 7.3 is called the *configuration graph* of M.

7.2 The power of L

Computing with severe space bounds, as in L, seems quite difficult. Also, it might be somewhat less familiar than, say, computing within a time bound. 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 deep mathematical techniques of wide applicability. We hinted at this in Chapter 0. We now give further examples. At the same time we develop our intuition of what is computable with little space.

To set the stage, we begin with a composition result. In the previous sections we used several times the simple result that the composition of two maps in P is also in P. 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. The proof is our first example of a general philosophy which can be cast as follows:

$$\boxed{\text{Unlike time, space can be reused.}}$$

Lemma 7.1. Let $f_1 : [2]^* \to [2]^*$ be in Space(s_1) and satisfy $|f_1(x)| \le m(|x|)$ for a function m. Suppose $f_2 : [2]^* \to [2]$ is in Space(s_2).

Then the composed function g defined as $g(x) = f_2(f_1(x))$ is computable in space $c(s_2(m(n)) + s_1(n) + \log nm(n))$.

In particular, if f_1 and f_2 are in L then g is in L, as long as $m \le n^d$ for a constant d.

Exercise 7.3. Prove this.

7.2.1 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 $\ge n$ to multiply two n-bit integers.

Theorem 7.4. The following problems are in L:

1. Addition of two input integers.
2. Iterated addition: Addition of any number of input integers.
3. Multiplication of two input integers.
4. Iterated multiplication: Multiplication of any number of input integers.
5. Division of two integers.

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 16.

Proof of 1. in Theorem 7.4. We are given as input $x, y \in \mathbb{N}$ 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. **QED**

Exercise 7.4. Prove 2. and 3. in Theorem 7.4.

Proving 4. and 5. is more involved and requires some of those deep mathematical tools of wide applicability we alluded to before. Division can be computed once we can compute 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 Chinese-remaindering representation of integers, abbreviated CRR.

Definition 7.4. We denote by $\mathbb{Z}_m$ the integers modulo m equipped with addition and multiplication (modulo m).

Theorem 7.5. 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 $e_1, ..., e_\ell \leq (p')^c$, 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 7.1. $\mathbb{Z}_6$ 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 CRR are performed in each coordinate separately; how convenient.

To compute iterated multiplication the idea is to move to CRR, perform the multiplications there, and then move back to standard representation. A critical point is that each coordinate in the CRR 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 $\ell := n^c$ and compute the first ℓ prime numbers $p_1, p_2, \ldots, p_\ell$.
2. Convert the input into CRR: 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 CRR: $(\Pi_{i=1}^t x_i \mod p_1), \ldots, (\Pi_{i=1}^t x_i \mod p_\ell)$.
4. Convert back to standard representation.

Exercise 7.5. Prove the correctness of this algorithm.

Now we explain how steps 1, 2, and 3 can be implemented in L. Step 4 can be implemented in L too, but showing this is somewhat technical due to the computation of the numbers e_i in Theorem 7.5. However these numbers only depend on the input length, and so we will be able to give a self-contained proof that iterated multiplication has branching programs of size n^c.

Step 1

By Theorem 2.6, 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 L. 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 7.4. (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 L. 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 L and only take $c \log n$ bits. Multiplying by y_j is also in L. Hence the problem reduces to iterated addition of n numbers which is in L by Theorem 7.4.

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 in L.

Step 4

By Theorem 7.5, to convert back to standard representation from CRR 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 are in L by Theorem 7.4.

Putting the steps together

Combining the steps together we can compute iterated multiplication in L as long as we are given the integers e_i in Theorem 7.5.

Theorem 7.6. Given integers $x_1, x_2, \ldots, x_t$, and given the integers $e_1, e_2, \ldots, e_\ell$ as in Theorem 7.5, where $\ell = n^c$, we can compute $\prod_i x_i$ in L.

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 7.1. Iterated multiplication has branching programs of size n^c.

Exercise 7.6. 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 L, a result which we stated but not completely proved.

Exercise 7.7. Show that the iterated multiplication of $d \times d$ matrices over the integers has branching programs of size n^{c_d}.

7.2.2 Graphs

We now give another example of the power of L.

Definition 7.5. The undirected reachability UConnd problem: Given an undirected graph G and two nodes s and t in G 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.

Theorem 7.7. 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 L.

7.2.3 Linear algebra

Our final example comes from linear algebra. Familiar methods for solving a linear system

$$Ax = b$$

can be done requires 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 7.8. Solving a linear system is computable in Space$(c \log^2 n)$.

7.3 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 $\le t$ 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

$u \rightsquigarrow^t v \Leftrightarrow \exists p_1, p_2, \ldots, p_{b-1} : \forall i \in \{0, 1, b-1\} : p_i \rightsquigarrow^{t/b} p_{i+1},$

where we denote $p_0 := u$ and $p_b := v$.

An important aspect of this technique is that it can be applied *recursively:* We can apply it again 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. We will see several instantiations of this technique, for various settings of parameters, ranging from $b = 2$ to $b = n^c$.

We now utilize the checkpoint technique to show a simulation of small-space computation by small-depth alternating circuits.

Theorem 7.9. A function computable by a branching program with S nodes is also computable by an alternating circuit of depth $c \log_b S$ and size $S^{b \log_b S + c}$, for any $b \le S$.

To illustrate the parameters, suppose $S = n^a$, and let us pick $b := n^\epsilon$ where $n \ge c_{a,\epsilon}$. Then we have ACs of depth $d := ca/\epsilon$ and size $\le S^{n^\epsilon d + c} = 2^{n^{c\epsilon}}$. In other words, we can have depth d and size $2^{n^{ca/d}}$, for every d. Another way of saying this is that the circuit has constant depth and power size on inputs of power-logarithmic length. This is quite useful to design ACs

Exercise 7.8. Prove that for any t and d the Majority function on $\log^d t$ bits can be computed by ACs of size t^{c_d} and depth c_d.

The parameters in the above discussion before the exercise in particular hold for every function in L. We will later give explicit functions (also in P) which cannot be computed by ACs of depth d and size $2^{n^{c/d}}$, "just short" of ruling out L. This state of affairs is worth emphasis:

(1) Every f in L has alternating circuits of depth d and size $2^{n^{c_f/d}}$.
(2) We can prove that there are explicit functions (also in L) which cannot be computed by circuits of depth d and size $2^{n^{c/d}}$.
(3) Improving the constant in the double exponent for a function in P would yield L $\neq$ P. In this sense, the result in (2) is the best possible short of proving a major separation.

Proof. We apply the checkpoint technique to the branching program, recursively, with parameter b. For simplicity we first assume that S is a power of b. Each application of the technique reduces the path length by a factor b. Hence with $\log_b S$ applications we can reduce the path length to 1.

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 $S^{b-1} \cdot b \leq S^b$ leaves. Iterating, the number of leaves will be

$$\left(S^b\right)^{\log_b S}.$$

Each leaf can be connected to the input bit on which it depends. The size of the circuit is at most c times the number of leaves.

If S is not a power of b we can view the branching program as having $S' \leq bS$ nodes where S' is a power of b . **QED**

The following is a uniform version of Theorem 7.9, and the proof is similar. It shows that we can speed up space-bounded computation with alternations.

Theorem 7.10. Any $f \in L$ is also in $\Sigma_{c_f/\epsilon}\mathrm{Time}(n^\epsilon)$, for any $\epsilon > 0$.

Proof. Let G be the configuration graph of f. Note this graph has n^{c_f} nodes. We need to decide if the start configuration reaches the accept configuration in this graph within $t := n^{c_f}$ steps.

We apply to this graph the checkpoint technique recursively, with parameter $b := n^{\epsilon/2}$. Each application of the technique reduces the path length by a factor b. Hence with c_f/ϵ applications we can reduce the path length to

$$\frac{t}{b^{c_f/\epsilon}} = \frac{n^{c_f}}{n^{c_f/2}} \leq 1.$$

Each quantifier ranges over $b \log n^{c_f} = c_f n^{\epsilon/2} \log n \leq n^\epsilon$ bits for large enough n.

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. The machine can compute the two endpoints in time $\log^c m$ where m is the total number of quantified bits, using rapid access. Once it has u and v it can check if u leads to v in one step by reading one bit from the input. Note $m \leq n^\epsilon \cdot c_f/\epsilon$, so $\log^c m \leq c_f n^\epsilon$. **QED**

7.4 The grand challenge for space

We now present impossibility results for space paralleling Chapter 3. Combinatorially, we have a nearly quadratic bound for branching programs.

Theorem 7.11. Branching programs require size $\ge n^2/\log^c n$ to solve the element distinctness problem on n bits arranged as $n/c\log n$ vectors of length $w := c\log n$.

Proof. Given a branching program with S nodes, partition its variable nodes in $n/c\log n$ sets depending on which (bit in a) vector they query. One of the sets has size $S' \le S/(n/c\log n)$. For every fixing of the other vectors, the branching program encodes those vectors, up to permutation. (One can reconstruct the vectors by running the branching program on every possible input.) In particular, it encodes any set subset of $[2]^{c\log n}$ of size $n/c\log n - 1$. On the other hand, the number of branching programs of size S' is $\le 2^{cS'\log S'} \le 2^{(S/n)\cdot\log^c n}$ (for each node we encode the neighbors among $S'+2$ possible nodes). Combining these two facts and taking logs you get

$$(S/n)\cdot\log^c n \ge \log_2\binom{n^c}{n/c\log n - 1}.$$

The binomial in the right-hand side is $\ge \binom{n^2}{n/c\log n} \ge 2^{cn}$ (cf Fact A.3) and the result follows. **QED**

No quadratic bound is known for NP.

Again, we can use diagonalization to prove a space hierarchy. Following section §3.3 we show this for partial functions. The corresponding extension to total functions is routine but tedious, and we will skip it.

Theorem 7.12. There is a partial function in Space($s(n)$) such that any TM M computing it uses space $\ge s(n) - c|M|$, for any $s(n)$.

In other words, Space($s(n)$) $\supsetneq$ Space($s(n) - \omega(1)$).

Proof. Consider a TM H that on input $x = (M, 1^{n-|M|})$ of length n runs M on x until it stops and then complements the answer. (We can use a simple encoding of these pairs, for example every even-position bit of the description of M is a 0.) The TM H is specifically implemented as follows: it begins by making a copy of M. This takes space $c|M|$. Then every step of the computation of M can be simulated by H reading the description of M. (Unlike for time, cf Theorem 3.3, the description of M does not have to accompany the head, but can be left at the beginning of the tape. The head may travel back-and-forth to simulate each step of M, but the space does not increase further.)

Now define X to be the subset of inputs x of length n where the M encoded by x runs in space $\le s(n) - c|M|$ on x. On these inputs, H runs in space $\le s(n)$, as desired. This is because the space overhead is $c|M|$, but M uses space $c|M|$ less than our target of $s(n)$.

Now suppose N computes the same function as H in space $s(n) - c|N|$. Note that $x := (N, 1^{n-|N|})$ falls in the domain X of the function. Now consider running N on x. We have $N(x) = H(x)$ by supposition, but $H(x)$ is the complement of $N(x)$, contradiction. **QED**

Paralleling again the discussion in section 5.1.1, from the space hierarchy it follows that L $\neq$ PSpace. Hence in particular either P $\neq$ L or P $\neq$ 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*.

7.5 Reductions

Again, we can use reductions to relate the space complexity of problems.

7.5.1 P vs. PSpace

Definition 7.6. A problem f is PSpace-complete if $f \in$ PSpace and $f \in$ P $\Rightarrow$ PSpace $=$ P.

Definition 7.7. 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 7.2. $\exists x \forall y \exists z : (x \vee y) \wedge (\neg x \vee z)$ is a true QBF.

Theorem 7.13. QBF is PSpace-complete.

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, let M be a TM using space n^a. And let x be an input. As remarked in Theorem 7.1, the running time of the machine is $\leq c^{n^a}$. We use the checkpoint technique. That is, to know if M goes from configuration C_1 to C_2 in t steps, we guess a middle configuration C_m and check if it goes from C_1 to C_m 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 bits. Repeating $\log(c^{n^a}) \leq cn^a$ times, we have reduced the time to 1. The final check to do is to check 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. By Theorem 5.1 we can introduce another $\exists$ quantifer on n^{ca} bits and write this computation as a 3CNF. **QED**

7.5.2 L vs. P

Definition 7.8. A problem f is P-complete if $f \in$ P and $f \in$ L $\Rightarrow$ L = P.

Definition 7.9. The circuit value problem: Given a circuit C and an input x, compute $C(x)$.

Theorem 7.14. Circuit value is P-complete.

Proof. Circuit value is in P since we can evaluate one gate at the time. Now let $g \in$ P. We can reduce computing g on input x to a circuit value instance, as in the simulation of TMs by circuits in Theorem 1.5. The important point is that this reduction is computable in L. Indeed, given an index to a gate in the circuit, we can compute the type of the gate and index to its children via simple arithmetic, which is in L by Theorem 7.4, and some computation which only depends on the description of the P-time machine for g.n **QED**

Definition 7.10. The monotone circuit value problem: Given a circuit C with no negations and an input x, compute $C(x)$.

Exercise 7.9. Prove that monotone circuit value is P-complete.

Recall from section 7.2.3 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 7.11. The linear inequalities problem: Given a $d \times d$ matrix A of integers and a d-dimensional vector, determine if the system $Ax \le b$ has a solution over the reals.

Theorem 7.15. Linear inequalities is P-complete.

Proof. The ellipsoid algorithm shows that Linear inequalities is in P, but we will not discuss this classic result.

Instead, we focus on showing how given a circuit C and an input x we can construct a set of inequalities that are satisfiable iff $C(x) = 1$.

We shall have as many variables v_i as gates in the circuit, counting input gates as well.

For an input gate $g_i = x_i$ add equation $v_i = x_i$.

For a Not gate $g_i = \text{Not}(g_j)$ add equation $v_i = 1 - v_j$.

For an And gate $g_i = \text{And}(g_j, g_k)$ add equations $0 \le v_i \le 1, v_i \le v_j, v_i \le v_k, v_j + v_k - 1 \le v_i$.

The case of Or is similar, or can be dispensed by writing an Or using Not and And.

Finally, if g_i is the output gate add equation $v_i = 1$.

We claim that in every solution to $Av \leq b$ the value of v_i is the value in [2] of gate g_i on input x. This can be proved by induction. For input and Not gates this is immediate. For an And gate, 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**

Exercise 7.10. Prove that any $f \in \mathrm{L}$ can be map reduced in quasi-linear time to a QBF with $\log^{c_f} n$ variables. Note: Remember that one computation step also depends on an input bit; you need to address the complexity of that.

7.6 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.

For starters, let us define non-deterministic space-bounded computation. A naive approach is to define it using the quantifiers from section §6.3, leading to the class $\exists \cdot \mathrm{L}$. This is an ill-fated choice:

Exercise 7.11. Prove $\exists \cdot \mathrm{L} = \exists \cdot \mathrm{P}$.

Instead, non-deterministic space is defined in terms of non-deterministic TMs.

Definition 7.12. A function $f : [2]^* \to [2]$ is computable in $\mathrm{NSpace}(s(n))$ if there is a two-tape TM which on input x never writes on the first tape and never uses more than $s(n)$ cells on the second, and moreover:

1. The machine is equipped with a special "Guess" state. Upon entering this state, a *guess bit* is written on the work tape under the head.

2. $f(x) = 1$ iff there exists a choice for the guess bits that causes the machine to output 1.

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 7.2.2 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 7.13. 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 7.16. Directed reachability is in NL.

Proof. The proof simply amounts to guessing a path in the graph. The algorithm is as follows:

```
"On input $G, s, t$, let $v := s$.
For $i = 0$ to $|G|$:
    If $v = t$, accept.
    Guess a neighbor $w$ of $v$. Let $v := w$.
If you haven't accepted, reject."
```

The space needed is $|v| + |i| = c \log |G|$. **QED**

We can define NL completeness similarly to NP and P completeness, and have the following result.

Theorem 7.17. Directed reachability is NL-complete. That is, it is in NL and it is in L iff L = NL.

Exercise 7.12. Prove this.

Recall by definition Space($s(n)$) $\subseteq$ NSpace($s(n)$). We showed Space($s(n)$) $\subseteq$ Time($n^c c^{s(n)}$) in Theorem 7.1. We can strengthen the inclusion to show that it holds even for non-deterministic space.

Theorem 7.18. NSpace($s(n)$) $\subseteq$ Time($n^c c^{s(n)}$).

Proof. On input x, we compute the configuration graph G of M 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. Then we solve reachability on this graph in power time, using say breadth-first-search. **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 NP $\subseteq$ Exp. Thus the situation for space is entirely different.

Theorem 7.19. NSpace(s) $\subseteq$ Space(cs^2), for every function $s = s(n) \geq \log n$. 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 N be a non-deterministic TM computing a function in NSpace($s(n)$). We aim to construct a deterministic TM M that on input x returns

$$\text{Reach}(C_{\text{start}}, C_{\text{accept}}, c^{s(n)}),$$

where Reach(u, v, t) decides if v is reachable from u in $\leq t$ steps in the configuration graph of N on input x, and C_{start} is the start configuration, C_{accept} is the accept configuration, and $c^{s(n)}$ is the number of configurations of N.

The key point is how to implement Reach.

```
Computing Reach($u, v, t$)
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 Reach(u, v, t). We have

$$S(t) \le cs(n) + S(t/2).$$

This is because we can re-use the space for two calls to Reach. Therefore, the space for Reach$(C_{\text{start}}, C_{\text{accept}}, c^{s(n)})$ is

$$\le cs(n) + cs(n) + \ldots + cs(n) \le cs^2(n).$$

QED

To set the stage for the next result, recall that we do not know if Ntime(t) is closed under complement. It is generally believed not to be, and we showed that if it is then the PH collapses Exercise 6.5.

What about space? Theorem 7.19 shows NSpace$(s) \subseteq$ 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 7.20. The complement of Path is in NL. In particular, NL is closed under complement.

Proof. We want a non-deterministic 2TM that given $G, s,$ and t accepts if there is no path from s to t in G.

For starters, suppose the machine has 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 $C$:

Initialize Count=0; Enumerate over all nodes $v \neq t$
  Guess a path from $s$ of length $|G|$. 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 $\le i$ 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 (candidate 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. **QED**

Exercise 7.13. Given a graph G and nodes s, t, explain how to compute in L 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 $|G'| \leq |G|^c$. Hint: Use Theorem 7.20 as a black-box.

Now give a direct "algorithmic" proof based on the algorithm in Theorem 7.20 but without using the result as a black-box, or mentioning configuration graphs or completeness. For simplicity, we will assume that we are given the count C: 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 L 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 $|G'| \leq |G|^c$.

7.7 Parity space

Definition 7.14. $\oplus \cdot$ L is defined as NL (cf 7.12) except "a choice" is replaced with "an odd number of choices".

Theorem 7.21. The following problems are complete for $\oplus \cdot$ L. All matrices are over $\mathbb{F}_2$:

- Computing the product of matrices,
- Computing the determinant of a matrix,
- Inverting a matrix.

Corresponding classes for matrices over the *integers* are known as DET (for boolean problems) and GapL (for function problems).

Similarly to section 5.5.2 and Theorem 6.4 one can prove that NL is in the non-uniform analogue of $\oplus \cdot$ L.

7.8 TiSp

So far in this chapter we have focused on bounding the space usage. For this, the TM model was sufficient, as remarked at the beginning. It is natural to consider algorithms that operate in little time *and* space. For this, of course, whether we use TMs or RAMs makes a difference.

Definition 7.15. Let TiSp(t, s) be the functions computable on a RAM that on every input $x \in [2]^n$ runs in time $t(n)$ and does not write on cells with addresses outside of the range $n + 1..n + s(n)$.

In particular, cells $0..n$, which recall from Definition 1.6 contain the input and its length, are read-only (as in Definition 7.1).

Exercise 7.14. Prove L $= \bigcup_d$ TiSp(n^d, d).

An alternative definition of TiSp would allow the RAM to access $s(n)$ cells anywhere in memory. One can maintain a data structure to show that this alternative definition is equivalent to Definition 7.15.

To illustrate the relationship between TiSp, Time, and Space, consider undirected reachability. It is solvable in $\text{Time}(n \log^c n)$ by breadth-first search, and in logarithmic space by Theorem 7.7. But it isn't known if it is in $\text{TiSp}(n \log^a n, a \log n)$ for some constant a.

Exercise 7.15. Prove the following version of Theorem 7.10: $\text{TiSp}(n^a, n^{1-\alpha}) \subseteq \Sigma_{ca/\alpha}\text{Time}(n)$ for any $a \ge 1$ and $\alpha > 0$.

The following is a non-uniform version of TiSp.

Definition 7.16. 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 $|L_i| \le W$ for every i.

Thus t represents the time of the computation, and $\log W$ the space.

Recall that Theorem 7.9 gives bounds of the form $\ge cn^2/\log n$ on the size of branching program (without distinguishing between length and width). For branching programs of length t and width W this bound gives $t \ge cn^2/W \log n$. Note this gives nothing for power width like $W = n^2$. The state-of-the-art for power width is $t \ge \Omega(n\sqrt{\log n/\log\log n})$ (in fact the bound holds even for subexponential width).

With these definitions in hand we can refine the connection between branching programs and small-depth circuits in Theorem 7.9 for circuits of depth 3.

Theorem 7.22. 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 depth-3 circuit with $\le 2^{c\sqrt{t \log W}}$ wires.

We will later show explicit functions that require depth-3 circuits of size $2^{c\sqrt{n}}$. Theorem 7.22 shows that improving this would also improve results for small-width branching programs, a refinement of the message emphasized after Theorem 7.9.

A more general version of Theorem 7.22. states that for any parameter b one can have a depth-3 circuit with

$$2^{b \log W + t/b + \log t}$$

wires, output fan-in W^b, and input fan-in t/b. Interestingly, this tradeoff essentially matches known impossibility results for depth-3 circuits!

Exercise 7.16. Prove Theorem 7.22.

7.9 Three impossibility results for 3Sat

We should turn back to a traditional separation technique – diagonalization.

In this chapter we put together many of the techniques we have seen to obtain several impossibility results for 3Sat. The template of all these results (and others, like those mentioned in section §5.1) is similar. All these results prove time bounds of the form $t \geq n^{1+\alpha}$ where $\alpha \in (0,1)$. One can optimize the methods to push α close to 1, but even establishing $\alpha = 1$ seems out of reach, and there are known barriers for current techniques.

7.9.1 Impossibility I

We begin with the following remarkable result.

Theorem 7.23. 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$(n \log^{10} n)$. In particular, Theorem 7.23 implies that any algorithm for 3Sat either must use super-logarithmic space or time n^{1+c}.

Proof. We assume that what we want to prove is not true and derive the following striking contradiction with the hierarchy Theorem 3.4:

$$\begin{aligned} \text{Time}(n^2) &\subseteq \text{L} \\ &\subseteq \bigcup_d \Sigma_d \text{Time}(n) \\ &\subseteq \text{Time}(n^{1.9}). \end{aligned}$$

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 log-space, by Theorem 5.1 and the discussion after that.

The second inclusion is Theorem 7.10.

For the third inclusion, the assumption that 3Sat $\in$ Time$(n^{1+\epsilon})$ for every ϵ implies that NTime$(dn) \subseteq$ Time$(n^{1+\epsilon})$ for every d and ϵ, by the quasi-linear-time completeness of 3Sat, Theorem 5.4. Now apply Exercise 6.4. **QED**

7.9.2 Impossibility II

We now state and prove a closely related result for TiSp. We seek to rule out algorithms for 3Sat that simultaneously use little space and time, whereas in Theorem 7.23 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 7.24. 3Sat $\notin$ TiSp$(n^{1+c_\epsilon}, n^{1-\epsilon})$, for any $\epsilon > 0$.

The important aspect of Theorem 7.23 is that it applies to the RAM model; stronger results can be shown for space-bounded TMs.

Exercise 7.17. Prove that Palindromes $\notin$ TM-TiSp$(n^{1+c_\epsilon}, n^{1-\epsilon})$, for any $\epsilon > 0$. (TM-TiSp(t, s) is defined as Space(s), cf. Definition 7.1, but moreover the machine runs in at most t steps.) Hint: This problem has a simple solution. Give a suitable simulation of TM-Tisp by 1TM, then apply Theorem 3.1.

Proof. We assume that what we want to prove is not true and derive the following contradiction with the hierarchy Theorem 3.4:

$$\begin{aligned}
\text{Time}(n^{1+\epsilon}) &\subseteq \text{TiSp}(cn^{(1+\epsilon)(1+c_\epsilon)}, cn^{(1+\epsilon)(1-\epsilon)}) \\
&\subseteq \text{TiSp}(n^{1+c_\epsilon}, cn^{1-\epsilon^2}) \\
&\subseteq \Sigma_{c_\epsilon}\text{Time}(n) \\
&\subseteq \text{Time}(n^{1+\epsilon/2}).
\end{aligned}$$

The first inclusion holds by the assumption, padding, and the fact that 3Sat is complete under reductions s.t. each bit is computable in time (and hence space) $n^{o(1)}$, a fact we do not prove here. **QED**

Exercise 7.18. Finish the proof by justifying the remaining inclusions.

7.9.3 Impossibility III

So far our impossibility results required bounds on space. We now state and prove a result that applies to time. Of course, as discussed in Chapter 3, we don't know how to prove that, say, 3Sat cannot be computed in linear time on a 2TM. For single-tape machines, we can prove quadratic bounds, for palindromes (Theorem 3.1) and 3Sat (Problem 4.3). Next we consider an interesting model which is between 1TM and 2TM and is a good indication of the state of our knowledge.

Definition 7.17. A 1.5TM is like a 2TM except that the input tape is read-only.

Theorem 7.25. 3Sat requires time n^{1+c} on a 1.5TM.

Exercise 7.19. Prove Theorem 7.25 following this guideline:

1. Let M be a 1.5TM running in time $t(n)$. Divide the read-write tape of M into consecutive blocks of b cells, shifted by an offset $i < b$. (So the the first cells of the blocks include $i, i + b, i + 2b, \ldots$.) 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 1.5TM-Time$(n^{1.1}) \subseteq \exists y \in [2]^{n^{1-c}}$TiSp$(n^c, n^{1-c})$. (The right-hand side is the class of functions $f : [2]^* \to [2]$ for which there is a RAM M that on input (x, y), where $|x| = n$, runs in time n^c and uses memory cells $0..n^{1-c}$ and s.t. $f(x) = 1 \Leftrightarrow \exists y \in [2]^{n^{1-c}} M(x, y) = 1$.)

3. Conclude the proof.

7.10 Problems

Problem 7.1. Prove that any graph with m edges and in-degree c can be pebbled with $cm(\log\log m)/\log m$ pebbles.

Hint: Prove that any graph of depth d and in-degree c can be pebbled using cd pebbles. Complete the proof using Lemma 8.1.

Problem 7.2. 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 $\text{L} \neq \Sigma_{a(n)}\text{Time}(n^\delta)$ for any growing $a(n)$ and any constant δ. (Hint: Use a slight extension of Problem 6.1.)

Problem 7.3. In this problem we will explore a beautiful illustration of the power of L. The Simplify problem is defined as follows. The input is a string over the alphabet $\{u, v, u^{-1}, v^{-1}\}$. Such a string can be simplified by removing adjacent pairs of the type $uu^{-1}, u^{-1}u, vv^{-1}, v^{-1}v$. The problem asks whether a given input string becomes the empty string after simplification.

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.

We will show that Simplify is in L.

In this problem $|x|$ denotes absolute value.

(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 $\begin{bmatrix} x \\ y \end{bmatrix}$ 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.

Problem 7.4. A family of circuits $C_i : [2]^i \to [2]$ is *log-space uniform* if computing C_n from 1^n is in L.

Prove that any log-space uniform family of circuits has an equivalent family of power-size circuits with the following stronger uniformity condition: There is a linear-time TM with c tapes that on input gates u and v decides if v is an input to u. (Note that log-space uniformity is similar except that the TM runs in linear space.)

Problem 7.5. Given a graph G, nodes s and t, and the number C of nodes reachable from s in G, show how to compute in L a graph G' and nodes s', t' s.t. there is a path from s to t in G iff there is no path from s' to t' in G', and $|G'| \le |G|^c$.

Problem 7.6. Show Space(n) is not contained in 1.5TM-Time($n^{1.99}$). You can use the Space Hierarchy Theorem that, say, Space($n^{0.9}$) $\neq$ Space(n).

7.11 Notes

Theorem 7.2 is from [133, 116, 292]. The first paper gets a logarithmic saving for MTMs. The second paper extends this to RAMs (and other models). The third paper returns to MTMs and gives the square-root saving. It uses [62] (Theorem 9.3) which appeared right before.

Theorem 7.7 is from [220]. The time must have been "ripe:" a concurrent, different proof [260] gives the only slightly weaker space bound $c \log n \log\log n$. The results came quite as a shock during my own PhD, because the proof in [220] is simple. Later a simpler yet proof appeared [225].

Theorem 7.8 follows from [66] which in fact establishes a stronger result, specifically it shows that the problem is in NC^2, a class we encounter in Chapter 8.

The use of CRR for arithmetic is from [37], which also contains several reductions among arithmetical problems. Some of the steps are from the earlier work [184]. For a discussion of the complexity of division, see [12].

Theorem 7.11 is from [195]. Theorem 7.12 is from [245].

For a compendium of P-complete problems see [108].

Theorem 7.10 goes back to [196].

As for NP, a compendium of problems complete for P is available [108].

Theorem 7.19: [227].

Theorem 7.20 was obtained independently in [137, 254]. Naturally, not even this central result came from nowhere: an earlier surprising collapse paved the way, at least for one of the proofs [165]. The proof in [137] uses a logical formalism, the proof we presented is closer to the one in [254]. This question is a.k.a. the second LBA problem, from [162]. As usual, had the answer been different, it would have had applications to the first LBA problem, which is the basically question whether Theorem 7.19 is tight.

See [9, 39] for the state-of-the-art bounds for power-width branching programs.

Complete problems for DET and $\oplus \cdot$ L (including Theorem 7.21) are from [64, 68]. For the relationship with NL see [90].

Problem 7.4 is from [145], where in fact constant-locality uniformity is achieved.

The introductory quote to section §7.9 is from [78], where Theorem 7.23 is proved. This influential work ignited a whole research area. For a survey (not up to date) see [268] or [289]. For the limitations of this type of results, see [54]. Theorem 7.25 is proved in [179] for NTime instead of 3Sat. The result for 3Sat appeared in [269].

Chapter 8

Circuits of small depth

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, Marvin Minsky and Seymour Papert, 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 Minsky and Papert 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, and the PRAM, recent spectacular progress in artificial intelligence has made it even more apparent that small-depth circuits can have amazing capabilities ranging from playing chess, recognizing images, 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.

In this chapter we investigate circuits of small depth, starting with logarithmic depth. Later we move to constant-depth threshold circuits and we link them to neural networks.

We'll introduce several complexity classes, arranged as follows:

$$\mathrm{NC}^0 \subsetneq \mathrm{AC} \subsetneq \mathrm{AC}[2] \subsetneq \mathrm{ACC} \subseteq \mathrm{TC} \subseteq \mathrm{NC}^1 \subseteq \mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{NC}^2 \subseteq \mathrm{NC}^3 \subseteq \cdots \subseteq \mathrm{P}$$

8.1 NC

Let us begin slowly with some basic properties of small-depth circuits so as to get familiar with them. The next exercise shows 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 8.1. A circuit of depth d has size $\leq c^d$ without loss of generality.

The next exercises shows how to compute several simple functions by log-depth circuits.

Exercise 8.2. Prove that the Or, And, and Parity functions on n bits have circuits of depth $c \log n$.

Prove that any $f : [2]^n \to [2]$ computable by an AC of depth d and size $s \geq n$ is also computable by a circuit of depth $cd \log s$ and size s^c.

Next, let us relate these circuits to branching programs. The upshot is that circuits of logarithmic depth are a special case of power-size branching programs, and the latter are a special case of circuits of log-square depth.

Theorem 8.1. Directed reachability has circuits of depth $c \log^2 n$ and size n^c. In particular, the same holds for any function in NL, and any function with power-size branching programs.

Proof. On input a graph G on u nodes and two nodes s and t, let M be the $u \times u$ transition matrix corresponding to G, where $M_{i,j} = 1$ iff edge $j \to i$ is in G.

Transition matrices are multiplied as normal matrices, except that "+" is replaced with "$\vee$," which suffices to know connectivity. To answer directed reachability we compute entry t of $M^u v$, 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^u can be done by squaring $c \log u$ times M. Each squaring can be done in depth $c \log u$, by Exercise 8.2. This establishes the first claim, since $u \leq n$.

The "in particular" follows because those functions can be reduced to directed reachability efficiently. **QED**

Conversely, we have the following.

Theorem 8.2. Any function $f : [2]^n \to [2]$ computed by a circuit of depth d can be computed by a branching program of size 2^d.

In particular, functions computed by circuits of logarithmic depth can be computed by branching programs of power size.

Later in Theorem 9.2 we will prove the stronger and much less obvious result that the equivalence holds even for branching programs of width 5.

Proof. We proceed by induction on the depth of the circuit C. If the depth is 1 then C is either a constant or an input bit, and a branching program of size 1 is available by definition.

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 2^{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. **QED**

Exercise 8.3. Finish the proof by analyzing the case $C = C_1 \vee C_2$.

Definition 8.1. NC^i is the class of functions $f : [2]^* \to [2]^*$ computable by circuits that have depth $a \log^i n$ and size n^a, for some constant a. The circuits are uniform if they can be computed in L.

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 in this class, see for example Theorem 11.4.

Exercise 8.4. Prove that $\mathrm{NC}^0 \neq \mathrm{NC}^1$ (mostly to practice definitions.)

We can equivalently think of NC^1 as power-size formulae.

Definition 8.2. 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. Alternatively, we can think of a formula as a circuit with fan-out 1.

Formulae can have large depth, as in $x_1 \wedge (x_2 \wedge (x_3 \wedge \ldots) \ldots))$. Yet we have:

Theorem 8.3. NC^1 is the same as the class of functions that have power-size formulae.

Proof. Given a circuit of depth d, we build the formula where there is a gate for every path from the output to a gate. The number of such paths is $\leq c^d$. You connect (the gate corresponding to) path p to path p' if p' extends p by one edge. The fan-out is 1 by definition.

Conversely, we prove by induction on s that any formula f of size s has circuits of depth $c \log s$. Find a subformula g of size $\in [cs, cs]$. Let g_0 be f with g replaced by 0, and same for g_1. 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 8.5. Prove that g exists.

8.1.1 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 7.4) can in fact be done in NC^1 as well.

Theorem 8.4. The following problems are in NC^1:

1. Addition of two input integers.
2. Iterated addition: Addition of any number of input integers.
3. Multiplication of two input integers.
4. Iterated multiplication: Multiplication of any number of input integers.
5. Division of two integers.

Exercise 8.6. Prove Item 1. in Theorem 8.4.

Iterated addition is surprisingly non-trivial. We can't use the methods from the proof of Theorem 7.4. Instead, we rely on a new and very clever technique.

Proof of Item 2. in Theorem 8.4.. 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, which we add as in Item 1.

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 8.1. Majority is in NC^1.

Exercise 8.7. Prove it.

Exercise 8.8. Prove Item 3. in Theorem 8.4.

Next we turn to iterated multiplication. The idea is to follow the proof for L in section 7.2.1. We shall use CRR 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 result is useful now:

Theorem 8.5. If p is a prime then $(\mathbb{Z}_p - \{0\})$ is a cyclic group, meaning that there exists a generator $g \in (\mathbb{Z}_p - \{0\}) : \forall x \in (\mathbb{Z}_p - \{0\}), x = g^i$, for some $i \in \mathbb{Z}$.

Example 8.1. For $p = 5$ we can take $g = 2$: $2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8 = 3$.

Proof of Item 4. in Theorem 8.4. We follow the proof for L in section 7.2.1. To compute iterated product of integers $r_1, r_2, \ldots, r_t$ modulo p, use Theorem 8.5 to compute exponents $e_1, e_2, \ldots, e_t$ s.t.

$$r_i = g^{e_i}.$$

Then $\prod_i r_i \mod p = g^{\sum_i e_i}$. We can use Item 2. to compute the iterated addition of the exponents. 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 combining Theorem 1.4 and Exercise 8.2). **QED**

One can also compute division, and make all these circuits uniform, but we won't prove this now.

8.1.2 Linear-size NC^1

It is unknown whether NP has linear-size circuits of logarithmic depth! But there is a non-trivial simulation of such circuits by ACs of depth 3 of sub-exponential size.

Theorem 8.6. Any circuit $C : [2]^n \to [2]$ of size an and depth $a \log n$ has an equivalent AC of depth 3 and size $2^{c_a n/\log\log n}$.

The idea is... yes! Once again, we are going to guess computation. The idea of the simulation is to identify $o(n)$ wires to remove from C so that the resulting circuit becomes very disconnected in the sense that each of its connected components 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 and we refer to figure 8.1.

A circuit can be viewed as an acyclic directed graph with nodes representing gates and directed edges representing the flow of computed values from the output of one gate to the input of the next. The graph corresponding to C in Theorem 8.6 is connected, but we 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.

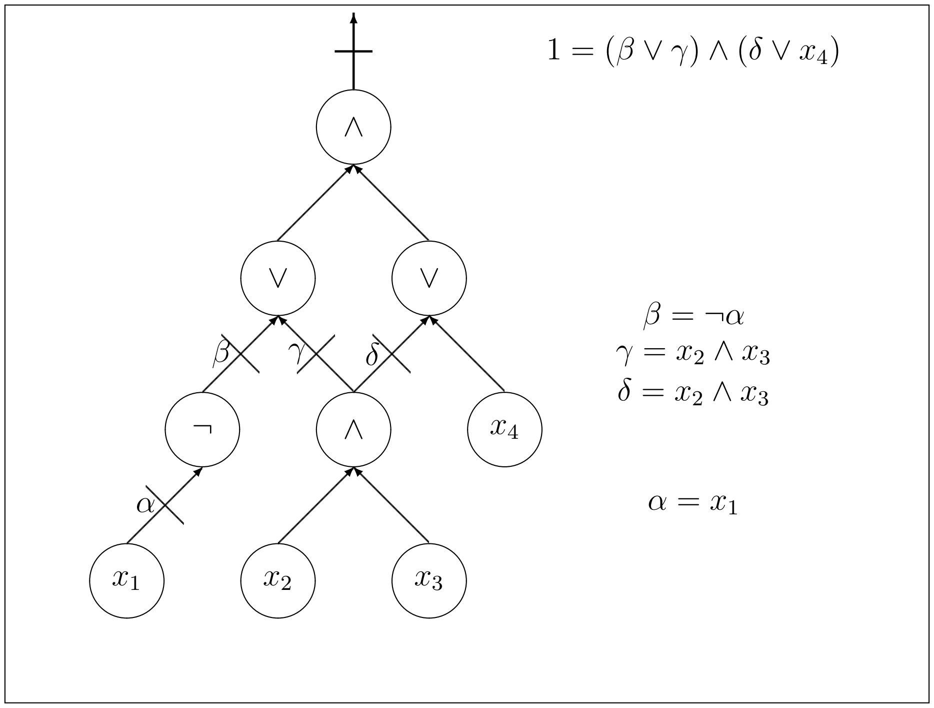


Figure 8.1: The removal of edges a, b, c, and d reduces the depth. The circuit evaluates to 1 if and only if there are $a, b, c, d \in [2]$ satisfying the corresponding equations.

Definition 8.3. 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 8.9. 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 8.1. 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 8.6. 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 $\le a \cdot n/\ell$ edges. Applying the lemma again we reduce the depth to $2^{\ell-2}$ by removing $\le a \cdot n/(\ell - 1)$ 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

$$a \cdot n \left(\frac{1}{\ell} + \frac{1}{\ell - 1} + \ldots + \frac{1}{\ell - \log(2a) + 1} \right) \le a \cdot n \cdot \frac{\log(2a)}{\ell - \log(2a) + 1} = a \cdot n \cdot \frac{\log(2a)}{\log \log n}.$$

For slight convenience we also remove the output edge e_{output} of the circuit; 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 $\le (\log n)/2$ 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 $\le \sqrt{n}$ 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

8.2 TC

Definition 8.4. A *threshold circuit,* abbreviated TC, is a circuit made of Majority gates (of unbounded fan-in). We also allow gates computing constants 0 and 1 (since they aren't immediate to implement using Majority only, unlike for AC). We denote by TC the class of functions f computable by a TC of depth d and size n^d for some constant d.

Exercise 8.10. Prove that AC $\subseteq$ TC $\subseteq$ NC^1.

TCs are one of the frontiers of our knowledge. It isn't known how to prove impossibility results even for TCs of depth 3 and size, say, n^2. As usual, a good explanation for this ignorance is the power of TCs, of which we give several examples next.

Exercise 8.11. 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 TC. Hint: Similar to a part of Exercise 6.15. You may want to start by proving that deciding if the input weight is equal to a fixed value is in TC.

The result PH$\subseteq$ Maj $\cdot$ Maj $\cdot$ P obtained in 6.15 in particular yields the following.

Theorem 8.7. Any function f in AC has TCs of depth 3 and size $2^{\log^{c_f} n}$.

Exercise 8.12. Prove Theorem 8.7 but for depth 4 instead of 3, using only Theorem 6.5 from Chapter 6. Hint: Use Exercise 8.11.

8.2.1 The power of TC: Arithmetic

Theorem 8.8. The following problems are in TC:

1. Addition of two input integers.
2. Iterated addition: Addition of any number of input integers.
3. Multiplication of two input integers.
4. Iterated multiplication: Multiplication of any number of input integers.
5. Division of two integers.

The proof follows closely that for NC^1 in section §8.1.1 (which in turn was based on that for L). Only iterated addition requires a new idea.

Exercise 8.13. Prove the claim about iterated addition. (Hint: Write input as $n \times n$ matrix, one number per row. Divide columns into blocks of $t = c \log n$.)

8.2.2 Neural networks and impossibility results for TC

Neural networks are made of a "small" number of layers, each consisting of a large number of *(artificial) neurons*. Each neuron computes a function from $\mathbb{R}^m \to \mathbb{R}$ as follows. On input $(x_1, \ldots, x_m) \in \mathbb{R}^m$, the neuron computes the weighted sum $s := \sum_i w_i x_i$ where $w_i \in \mathbb{R}$ are weights which define the neuron, and then outputs $\sigma(s)$ where σ is an *activation function*. Several activation functions are considered. In first approximation, we can think of σ as being simply a threshold, i.e., it outputs 1 if $s \geq \theta$ and 0 otherwise, for some threshold θ. Such neurons are also called *weighted thresholds*. In practice, it works better to use an activation function like *ReLU* (rectified linear unit) which is $\sigma(s) := \max\{0, s\}$.

Terminology recap A threshold is similar to a majority gate (one can include constants to the input to majority to shift the threshold). A weighted threshold is like majority but the inputs are weighted. Note that multiple edges from a majority gate to another gate aren't allowed – since we defined size as the number of gates, that would make a majority gate equal to a weighted thresholds. (An equivalent choice would be to allow such edges but define size as the number of edges.)

A neuron is a further generalization where the output is obtained via an activation function like a ReLU.

Exercise 8.14. Prove that neurons with a ReLU activation function are in TC. Feel free to assume that the weights w_i are integers with n^a bits for a constant a.

By applying this exercise to every neuron in a neural network, we see that neural networks can be simulated by threshold circuits of comparable size and depth. In fact, stronger results can be obtained for weighted thresholds, where the depth increases only by one:

Theorem 8.9. A depth-d size-n^a circuit of weighted thresholds has an equivalent threshold circuit of size $n^{c_{a,d}}$ and depth $d+1$.

We do not have impossibility results for depth-3 TC, and in fact we have the following striking version of the grand challenge:

$$\boxed{\text{Prove impossibility results for weighted thresholds of depth 2.}}$$

By Theorem 8.9 this would follow from impossibility results for threshold circuits of depth 3.

Question 8.1. *Can inner product be computed by a power-size weighted threshold of weighted thresholds?*

For depth-2 TC (i.e., *unweighted* thresholds) we can prove impossibility results. Here's how. By Theorem 3.9 it suffices to prove correlation bounds (cf section §3.5) against majority. For example, inner product does not correlate with majority. One way to show this is using communication complexity, cf Chapter 13. That is, majority can be computed with low communication, whereas inner product requires large communication (section 13.2.2).

8.2.3 TC vs. NC^1

Another great question is whether TC $=$ NC^1. For any d, we can show that functions in NC^1, such as Parity, require depth-d TCs of size $\geq n^{1+c\log d}$, and this is tight up to constants. A natural question is whether we can prove stronger bounds for harder functions, in NC^1 or elsewhere. A natural candidate is iterated multiplication of elements from a group, cf section §9.2. The following result shows that, in fact, stronger bounds would already prove "the whole thing," that is, TC $\neq$ NC^1. The proof is not using anything specific about the gates of TCs, but works for other circuit classes such as ACC (cf. section §8.3).

Theorem 8.10. Let G be a group. Suppose that the product of n elements in G can be computed by TCs of size n^k and depth d. Then for any ϵ the product can also be computed by TCs of size $d'n^{1+\epsilon}$ and depth $d' := cdk\log 1/\epsilon$.

For concreteness one can think $G = S_5$, as in section §9.2.

Proof. 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 $(n^{1+\epsilon}/n_i)^{1/k}$ matrices. At the root ($i = 0$) we have $n_0 = 1$.

By the assumption, each product at Level i has TCs of size $n^{1+\epsilon}/n_i$ and depth d. Hence Level i can be computed by TCs 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 $\geq n$ matrices, as required.

Hence the total depth of the circuit is $d \cdot ck\log(1/\epsilon)$, and the total size is the depth times $n^{1+\epsilon}$.

It remains to solve the recurrence. Letting $a_i := \log_n n_i$ we have the following recurrence for the exponents of n_i.

$$a_0 = 0$$
$$a_{i+1} = a_i(1-1/k) + (1+\epsilon)/k = a_i\beta + \gamma$$

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

Were the recursion of the form $a'_{i+1} = a'_i + (1+\epsilon)/k$ then obviously a'_{ck} would already be $\geq 1+\epsilon$. Instead for a_i we need to get to $ck\log(1/\epsilon)$.

8.3 ACC

We denote AC augmented with gates computing $\mod m$ by AC$[m]$. ACC (alternating circuits with counters) refers to any m. We also denote by AC$[m]$ the class of functions computable by AC$[m]$ circuits of size n^d and depth d for some constant d, and similarly for ACC.

Techniques based on polynomials, are effective to prove impossibility results against AC$[m]$ if m is prime or a prime power. These techniques are illustrated in section 8.4 in the fundamental case $m = 2$. They can be extended to any prime power m, but they break down when m is composite. It is consistent with our knowledge that any function in Exp has AC[6] of depth c and size n^c. It is open if Majority has such circuits!

For any m, AC$[m]$ can be simulated by polynomials. The simulation is incomparable to the one for $m = 2$ that we saw previously (Theorem 6.5) and we see again in section 8.4. In the new simulation we work with polynomials over the integers and then map the output to a boolean value. Equivalently, we can think of the polynomial as a depth-2 circuit. On the other hand, this new simulation works for every input as opposed to most inputs.

Lemma 8.2. Any AC$[d]$ of size n^d and depth d has an equivalent depth-2 circuit of size $2^{\log^{c_d} n}$ where the output is a symmetric function (i.e., only depends on the number of bits that are 1 in the input to that gate) and the other gates are And with fan-in $\leq \log^{c_d} n$.

As far as we know, general circuits are equivalent to ACC! Yet there is one thing that we can say about functions computable in ACC that we don't know for PCkt. We can solve ACC-Sat better than brute-force search. Using this, and Lemma 8.2, and diagonalization, one can prove the following result, which we don't know how to prove in other ways.

Theorem 8.11. NExp $\neq$ ACC.

The technique involve guessing circuits so it does not seem applicable to functions in NP.

8.4 AC[2]

AC[2] is a very interesting class because it lies at the frontier of our knowledge. We can prove impossibility results, but not correlation bounds. The following impossibility result is essentially the state-of-the-art. It will give a correlation bound no better than $1/\sqrt{n}$.

Theorem 8.12. Let C be an AC[2] of depth d and size s computing Majority on n bits. Then $\log^d s \geq c\sqrt{n}$.

Recall from section §7.3 that a stronger bound (even for AC) for an explicit function would have major consequences; in particular the function cannot be in L. The proof of Theorem 8.12uses *the polynomial method* (a.k.a. *low-degree approximation*), i.e., the simulation of AC[2] by low-degree polynomials (cf Theorem 6.5). Specifically, we use the following corollary:

Corollary 8.2. Let $C : [2]^n \to [2]$ be an AC[2] of depth d and size s. Then there is a polynomial p over $\mathbb{F}_2$ of degree $\log^d s/\epsilon$ such that $\mathbb{P}_x[C(x) \neq p(x)] \leq \epsilon$.

Proof. Theorem 6.5 gave a distribution P on polynomials s.t. for every x we have

$$\mathbb{P}_P[C(x) \neq P(x)] \leq \epsilon.$$

Averaging over x we also 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 $\leq \epsilon$, yielding the result. **QED**

Exercise 8.15. Theorem 6.5 was stated for AC, not AC[2]. Go back to the proof and explain why it works for AC[2] as well.

We then show that Majority cannot be approximated by such low-degree polynomials. For this the key result is the following:

Lemma 8.3. 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 $\leq n/2$. We claim that for every function $f : M_0 \to [2]$ there is a polynomial p_0 of degree $\leq n/2$ s.t. p_0 and f agree on M_0.

To verify this, consider the monomials of degree $\leq n/2$. 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 result follows. **QED**

Exercise 8.16. Prove the claim in the proof.

Proof of Theorem 8.12. Apply Corollary 8.2 with $\epsilon = 1/10$ to obtain p. Let S be the set of inputs on which $p(x) = C(x)$. By Lemma 8.3, any function $f : S \to [2]$ ca be written as

$$f(x) = p_0(x) + p_1(x) \cdot p(x).$$

The right-hand size is a polynomial of degree $\leq d' := n/2 + \log^d(cs)$. 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 $2^n(1/2 + c \log^d(s)/\sqrt{n})$, since each binomial coefficient is $\leq c2^n/\sqrt{n}$, cf Fact A.2.

On the other hand, $|S| \geq 0.9 \cdot 2^n$.

Combining this we get

$$0.9 \cdot 2^n \leq 2^n(1/2 + c \log^d(s)/\sqrt{n}).$$

This implies

$$0.4 \leq c \log^d(s)/\sqrt{n},$$

proving the theorem. **QED**

Stronger bounds are only known for functions computable in classes related to exponential time.

8.5 AC

In this section we present different techniques to prove impossibility results for AC, also slightly improving the parameters of the bounds established via the polynomial method, Theorem 8.12. To set the stage, let's prove strong results for depth 2, that is, DNFs.

Exercise 8.17. Prove that Majority requires DNFs of size $\geq 2^{cn}$. Hint: What if you have a term with $< n/2$ variables?

As discussed (cf section §8.1.2), 2^{cn} bounds even for depth 3 ACs are unknown, and would imply super-linear lower bounds for log-depth circuits. However, for AC a sharper technique is known that allows us to replace the $\sqrt{n}$ in Theorem 8.12 with n for several functions such as parity.

8.5.1 Restrictions and switching lemmas

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 uniformly at random, and also each unrestricted variable is set to a uniform bit. We denote by N_s the number of restrictions with exactly s stars.

Exercise 8.18. $N_s = 2^{n-s}\binom{n}{s}$.

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. The switching lemma shows that important classes of functions simplify dramatically when "hit" by a random restriction.

Definition 8.5. A *decision tree* of depth d is a branching program where every path has length $\leq d$.

In particular, depth-d decision trees compute 2^d-local functions, cf. Definition 1.5.

Lemma 8.4. *Let $C : [2]^n \to [2]$ equal the Or of functions $f_i : [2]^n \to [2]$ where each f_i is w-local. (The number of such functions is immaterial.) Let ρ be a random restriction with s stars. The probability that C_ρ is not a decision tree of depth d is $\leq (cws/n)^d$.*

To illustrate, f can be a DNF or a CNF with terms of size $w = c \log n$. The lemma says that if we pick a uniform restriction with $s := \sqrt{n}$ stars, and set $d = 10$, then the restricted function on s bits is a decision tree of depth d except with probability $((c \log n)/n)^{10} \leq 1/n^c$. Note in particular the restricted function depends only on $2^d = 1024$ bits, even though the original function may depend on all the bits. If we take larger d, the error probability gets even smaller, so small in fact that we can take a union bound and simultaneously simplify many DNFs. Doing this several times allows us to collapse an AC to a low-depth decision tree. We state and prove this consequence next, trading simplicity of exposition for parameter optimization.

Corollary 8.3. *Let $C : [2]^n \to [2]$ be an AC of depth d size s. Let ρ be a random restriction with n^{c_d} stars. The probability that C_ρ is not a decision tree of depth $\log s$ is $\leq s2^{-n^{c_d}}$.*

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 8.4 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 8.4 gives error $\leq (cw/n^{1/4})^w$, and so on. **QED**

Exercise 8.19. Let $C : [2]^n \to [2]$ be an AC of depth d size $s < 2^{n^{c_d}}$ (note the constant is the same as Corollary 8.3). Show that there is a restriction with n^{c_d} stars s.t. C_ρ is constant.

One can use the switching lemma to prove exponential lower bounds to compute explicit functions by small-depth ACs. The simplest example is parity, given next. In this case, we also prove an exponentially strong correlation bound, cf 3.5. The polynomial method gives weaker correlation bounds. This is a qualitative difference explored more in Chapter 11.

Corollary 8.4. The correlation between parity and an AC of depth d and size s is $\leq s2^{-n^{c_d}}$.

Proof. 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 8.3, after picking the restriction the circuit is a decision tree of depth $\log s$, which is strictly less than the number of remaining starts, except with probability $s2^{-n^{c_d}}$. **QED**

Exercise 8.20. State and prove via a reduction from Corollary 8.4 an impossibility result for Majority. Does a strong correlation bound hold as well?

8.5.2 Proof of Lemma 8.4

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 restrictions 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 8.21. Give it. For concreteness, think $d = 10$ and $s = \sqrt{n}$.

We give an alternative argument which we can then extend to the general case. We are going to encode (or map) such restrictions using (or: to a restriction)... with no stars (that 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. Recall Exercise 8.18. 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.

A critical observation is that we only want to encode 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 (cf. Exercise 8.18). The error probability is then

$$\frac{2^n}{\binom{n}{s}2^{n-s}} = \frac{2^s}{\binom{n}{s}} \leq (2s/n)^s.$$

Here we used $\binom{n}{s} \geq (n/s)^s$. This is stronger than Lemma 8.4. (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, standard calculations show that saving d stars reduces the number of restrictions by a factor $(cs/n)^d$.

Exercise 8.22. Prove $N_{s-d}/N_s \leq (cs/n)^d$. Guideline: Recall Exercise 8.18. To make the calculations as simple as possible, set $s := \sqrt{n}$ and consider the following way of selecting s stars among n variables: First pick $s-d$ stars. Then divide the remaining $n-(s-d)$ variables into d blocks of $(n-(s-d))/d \geq cn/d$ variables. Pick exactly one star in each block. Conclude the proof.

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: Or of functions on any subset of w bits ...isn't really different. First, the number of functions does not play a role, so you can think you have functions on any possible subset of w bits, where some functions may be constant. 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.

8.5.3 The original switching lemma

We now state and prove an earlier, weaker switching lemma, whose proof is however simpler and more intuitive.

Lemma 8.5. Let $f : [2]^n \to [2]$ be a k-CNF. Let ρ be a random restriction with $\mathbb{P}[*] = 1/n^c$. The probability that f_ρ is not c_k-local is $\leq 1/n^k$.

Proof. Induction on k. For $k = 1$, f is an And. If the And is on $\geq ck \log n$ bits then f_ρ will be constant with the desired probability. If the And is on $\leq ck \log n$ bits then the prob. that f_ρ depends on $\geq c_k$ bits is

$$\leq \binom{ck \log n}{c_k} n^{-c \cdot c_k} \leq 1/n^{c_k}. \qquad (8.1)$$

For the induction step, suppose there are $\geq c_k \log n$ Or gates with disjoint inputs. Since each Or gate is 0 after the restriction w.p. $\geq 1/c_k$, the result follows.

Otherwise, there is a set C of $\leq c_k \log n$ variables that touches every Or gate. For every assignment to these variables, we can apply the induction hypothesis, and do a union bound. Also, like in equation (8.1), the prob. that $\geq c_k$ variables in C are set to $*$ by ρ is $\leq 1/n^{c_k}$. **QED**

8.5.4 The power of AC: sampling

We showed in the earlier section that ACs cannot compute parity and majority. However, ACs can sample input-output pairs of these functions. In other words, this is a problem that ACs can't solve, but for which nevertheless they can create instances together with their solutions. These results have applications and again point to the unsuspected power of these circuits. Throughout we assume that the inputs to the circuits are uniform bits.

Exercise 8.23. Give a 2-local map $C : [2]^n \to [2]^{n+1}$ whose output distribution is $(X, \text{parity}(X))$ for uniform $X \in [2]^n$.

Sampling $(X, \text{majority}(X))$ by ACs is more involved and beautiful, and is only known to be possible up to a small error.

Exercise 8.24. Try to sample $(X, \text{majority}(X))$ by ACs for a few minutes. Write down what you tried.

The first step is sampling a uniform permutation.

Lemma 8.6. There is an AC whose output distribution is 2^{-n^c} close to a uniform permutation π over $[n]$, represented as n blocks of $c \log n$ bits where block i has $\pi(i)$ in binary.

Exercise 8.25. Assume Lemma 8.6. For any $i \leq n$ give an AC whose output distribution is 2^{-n^c}-close to a uniform n-bit strings of weight i. Give an AC whose output distribution is 2^{-n^c}-close to $(X, \text{majority}(X))$ for uniform $X \in [2]^n$.

Proof of Lemma 8.6. The main technique is known as "dart throwing:" we view the input random bits as random pointers $p_1, p_2, \ldots, p_n$ into $m \gg n$ cells. We then write i in the p_i-th cell (empty cells get "$*$"). If there are no collisions, the ordering of $[n]$ in the cells 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 cells before p_i are occupied, which cannot be done in small depth.

The key insight is to view the cells as representing the permutation in a different format, one from which we can explicitly write out the permutation in small depth. The format is known as the canonical form for the cyclic notation. We now briefly review it. Just like the standard format, the alternative format represents a permutation via an array $A[1..n]$ 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. The benefit of this format is that it works even if the array A has $m \gg n$ cells, of which $m - n$ are empty and marked by "$*$."

One can now verify that computing the image of i can be done in AC. Here in particular we use the fact that such circuits 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 in parallel, 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.

To conclude the proof of the lemma, generate ℓ uniform and independent sets of pointers $p_1^i, \ldots, p_n^i$, $i = 1, \ldots, \ell$, 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 $p_1^i, \ldots, p_\ell^i$ 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

$$\Pr[\forall i \exists j, k \leq n : p_j^i = p_k^i] = \Pr[\exists j, k \leq n : p_j^1 = p_k^1]^\ell \leq (1/2)^\ell.$$

Choosing $\ell := n$ proves the lemma. **QED**

8.6 Problems

Problem 8.1. Prove that the iterated multiplication of t 3×3 matrices with t-bit integer entries can be computed by power-size circuits of depth $c \log t \log^* t$. Recall $\log^* t$ is the number of times we need to apply log to t to reach c. Feel free to replace $\log^* t$ with $\log \log t$; this contains the main ideas.

Problem 8.2. TBD Show that any function $f \in \mathrm{NC}^1$ is computable on a TM using space zero and n^{c_f} states.

Problem 8.3. Let d be a constant and F a field of size $\leq d$.

You are given as input a vector $v \in F^d$ and a sequence of operations of the type

$$\begin{aligned} v &\to v + s, \\ v &\to Mv; \end{aligned}$$

where $s \in F^d$ and M is a $d \times d$ matrix.

Show how to compute in NC^1 the result of applying the operations to v.

Hint: Write each operation as multiplication by a $(d+1) \times (d+1)$ matrix.

Problem 8.4. Prove that an AC of depth d and size n^d cannot compute Gap-$\mathrm{Maj}_{1/2-\epsilon, 1/2+\epsilon}$ on n bits where $\epsilon = 1/\log^{c_d} n$.

Problem 8.5. (cf. Problem 9.1) Let G be solvable group. Prove that any function computable by a group program over G of length m is computable by an ACC circuit (section §8.3) of size m^{c_G}.

Guideline: First prove it for abelian groups. Second, more interestingly consider the dihedral group D_3 whose elements can be written as (t,b) where $t \in \mathbb{Z}_3$ and $b \in \mathbb{Z}_2$ and $(t,b)(t',b')$ equals

$$\begin{gathered}(t+t',b') \text{ if } b=0,\\(t-t',b'+1) \text{ if } b=1.\end{gathered}$$

In other words, you can move b to the right by flipping the sign of t'.

The proof for the dihedral groups contains all the ideas for the general case, which you can tackle by induction using that G has a normal subgroup H s.t. G/H is abelian.

Problem 8.6. Let G be a group. Show that there are ACs of size $n^{c_{|G|}}$ and depth $c_{|G|}$ whose output distribution on uniform bits is statistically close to $D := (g_1, g_2, \ldots, g_n, \prod_{i \le n} g_i)$ for uniform $g_i \in G$. The distance should be $1/n^{\omega(1)}$. Explain what is the group corresponding to Exercise 8.23, and why for some group the output distribution cannot equal D as in Exercise 8.23.

8.7 Notes

Theorem 8.3 was proved in [243] and apparently rediscovered in [49].

The "2-out-of-3" idea, in the proof of Item 2. in Theorem 8.4, is from [81].

The main ideas behind Theorem 8.6 and its proof are from [73]. The stated version is from [264]. Our exposition is based on [276], apparently the first exposition after [264].

Moving to TC, Theorem 8.9 is from [95].

The $n^{1+c\log d}$ lower bound for computing parity by TCs is from [140].

A weaker version of Theorem 8.10, which however contains the main message, is from [13]. The stated version matches the bound in [95] and is from [58].

Rumor has it that the submission version of [194] was titled along the lines of "The brain can compute pseudorandom functions."

The introductory quote is from [208]. 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 [183]. In the book [224] 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 [208] is [188]. What it showed is impossibility results for minimalistic neural networks, consisting of a threshold gate applied to And gates. 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 neural networks was hard to find. Perhaps a better explanation is that the hardware, data, and the math weren't yet there. The work [41] "vindicate[d] the reputation of the much maligned perceptron" by showing that small, probabilistic perceptrons can simulate AC.

Theorem 8.7 is from [11], but see discussion in section §6.7 for the genesis of these ideas.

The impossibility result for parity in TC is from [140].

Theorem 8.10: [13, 58].

Impossibility results for AC are among the most famous results in complexity and were obtained in the 80's in [86, 7, 294, 216, 241, 122] 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 proof of Theorem 8.12 we gave is from [242], though that paper splashes mathematics in a way that doesn't help me (Hilbert function, Grobner basis, "in the projective case the ideal $I(S)$ is *homogeneous* (or *graded*) if..." etc.). Perhaps one reason behind the aura of the switching lemma is that it's hard to find examples. It would be nice to read: If you have this extreme DNF here's what happens, on the other hand for this other extreme DNF here's what happens, and in general this always works and here's the switching lemma. *Examples are forever* – Erdos. Instead the switching lemma is typically presented as *blam*!: an example-free encoding argument which feels *deus ex machina*, as in this crisp presentation by Thapen. I have tried to give a slightly different exposition of the encoding argument.

Stronger bounds for AC with parity gates for less explicit functions are in [271].

Lemma 8.6 is from [181, 115]. Our presentation follows [280].

Lemma 8.2 is from [297, 42] .

Theorem 8.11 is from [291]. One step of the original proof was somewhat indirect and it was streamlined later in [145].

8.8 Historical vignette: The switching lemma

I must admit I had a good run (private communication)

Random restrictions have been used in complexity theory since at least the 60's [251]. The first dramatic use in the context of AC is due to [86, 7]. These works proved a *switching lemma* the amazing fact that a DNF gets simplified by a random restriction to the point that it can be written as a CNF, so you can collapse layers and induct. (An exposition is given below.) Using it, they proved super-polynomial lower bounds for AC. The proof in [86], presented in section 8.5.3, is very nice and if I want to get a quick intuition of why switching is at all possible, I often go back to it. [7] is also a brilliant paper, and long, unavailable online for free, filled with a logical notation which makes some people twitch. The first symbol of the title says it all, and may be the most obscene ever chosen:

$$\Sigma_1^1.$$

Subsequently, [294] proved exponential lower bounds of the form 2^{n^c}, with a refined analysis of the switching lemma. The bounds are tight, except for the constant c which depends on the depth of the circuit. Finally, the star of this section [121, 122] obtained $c = 1/(\text{depth}-1)$.

Yao's paper doesn't quite state that a DNF can be written exactly as a CNF, but it states that it can be approximated. Hastad's work is the first to prove that a DNF can be written as a CNF, and in this sense his statement is cleaner than Yao's. However, Yao's paper states explicitly that a small circuit, after being hit by a restriction, can be set to constant by fixing few more bits.

The modern formulation of the switching lemma says that a DNF can be written as a *shallow decision tree* (and hence a small CNF). This formulation in terms of decision trees is actually not explicit in Hastad's work. Beame, in his primer [36], credits Cai with this idea and mentions several researchers noted Hastad's proof works in this way.

Another switching lemma trivia is that the proof in Hastad's thesis is actually due to Boppana. Documents detailing Hastad's original argument remain classified, and contentious to this date. All the general public is left with is this remark from [122, 121]:

> *There are two versions of the proof of the main lemma which are almost identical except for notation. Our original proof was in terms of a labeling algorithm used by Yao in his proof. The present version of the proof, avoiding the use of such an algorithm, was proposed by Ravi Boppana.*

The first sentence was later [124] edited to

> *There are two versions of the proof of the Main Lemma that are identical except for notation.*

So, let's recap. Random restrictions are already in [251]. The idea of switching is already in [86, 7]. You already had three analyses of these ideas, two giving superpolynomial lower bounds and one [294] giving exponential. The formulation in terms of decision trees isn't in [122], and the proof that appears in [122] is due to Boppana.

Still, I would guess [122] is more well known than all the other works above combined. [294] did have a following at the time -- I think it appeared in the pop news. But hey -- have you ever heard of Yao's switching lemma?

The current citation counts offer mixed support for my thesis:

FSS: 1351

Y: 732

H - paper "Almost optimal...:" 867

H - thesis: 582

But it is very hard to use citation information. The two H citations overlap, and papers are cited for various reasons. For example FSS got a ton of citations for the connection to oracles (which has nothing to do with switching lemmas).

Instead it's instructive to note the type of citations that you can find in the literature:

> *Hastad's switching lemma is a cornerstone of circuit complexity* [No mention of FSS, A, Y]
>
> *Hastad's Switching Lemma is one of the gems of computational complexity* [Notes below in passing it builds on FSS, A, Y]

The wikipedia entry is also telling:

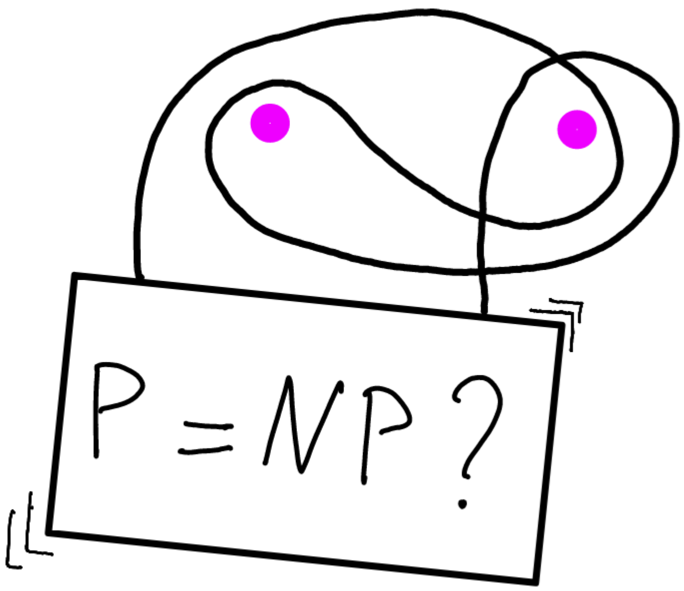
In computational complexity theory, Hastad's switching lemma is a key tool for proving lower bounds on the size of constant-depth Boolean circuits. Using the switching lemma, Johan Haåstad (1987) showed that... [No mention of FSS,A,Y]

I think that 99% of the contribution of this line of research is the *amazing idea* that random restrictions simplify a DNF so that you can write it as a CNF and collapse. 90% of the rest is analyzing this to get superpolynomial lower bounds. And 90% of whatever is left is analyzing this to get exponential lower bounds.

OK -- so maybe this is so, but it must then be the case that [122] is the final word on this stuff, like the ultimate tightest analysis that kills the problem. Actually, it is not tight in some regimes of interest, and several cool works of past and recent times address that. In the end, I can only think of one reason why [122] entered the mythology in ways that other works did not, the reason I have carefully sidestepped so far: *å*.

Chapter 9

Non-commutative magic



This may be my favorite chapter. Many surprises lay ahead, including a solution to the teaser in Chapter 0.

9.1 Computing with 3 bits of memory

We now present a surprising result that in particular strengthens Theorem 8.2. For a moment, let's forget about circuits, branching programs, etc. and instead consider a new, minimalistic type of programs. We will have 3 one-bit registers: R_0, R_1, R_2, operating modulo 2. We allow the following operations

$$R_i += R_j,$$
$$R_i += R_j x_k$$

where x_k is an input bit, for any $i, j \in \{0, 1, 2\}$, with $i \neq j$. (Talk about RISC!) 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.

Definition 9.1. For i, j and $f : [2]^n \to [2]$ we say that a 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.

Also 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. It is convenient to state it for circuits with Xor instead of Or gates. This is without loss of generality since $x \vee y = x + y + x \wedge y$.

Theorem 9.1. Suppose $f : [2]^n \to [2]$ is computable by circuits of depth d with Xor and And gates. For every $i \neq j$ there is a program of length $\leq c4^d$ for

$$R_i+ = R_j f.$$

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. We proceed by induction on d. When $d = 1$ the circuit is simply outputting a constant or one of the input bits, which we can compute with the corresponding instructions. (If the circuit is the constant zero then the empty program would do.)

Proceeding with 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

$$R_i+ = R_k f_1$$
$$R_k+ = R_j f_2$$
$$R_i+ = R_k f_1$$
$$R_k+ = R_j f_2.$$

QED

Exercise 9.1. Prove that the program for $f_1 \wedge f_2$ in the proof works. Write down the contents of the registers after each instruction in the program.

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 8.1 with Theorem 9.1. **QED**

Corollary 9.1. Iterated product of 3x3 matrices over $\mathbb{F}_2$ is complete for NC^1 under projections.

That is, the problem is in NC^1 and for any $f \in$ NC^1 and n one can write a sequence of $t = n^c$ 3x3 matrices $M_1, M_2, \ldots, M_t$ where each entry is either a constant or an input variable x_i s.t. for every $x \in [2]^n$:

$$\prod_{i=1}^{t} M_i \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} f(x) \\ 1 \\ 0 \end{bmatrix}.$$

Exercise 9.2. Prove this.

Recall from Chapter 7 (see in particular section 7.6) that various graph reachability problems are complete for space-bounded computation. In particular, one can reduce any function computable by branching programs of size s to iterated multiplication of $s \times s$ matrices over $\mathbb{F}_2$.

Exercise 9.3. Prove this. Specifically, prove that any function $f : [2]^n \to [2]$ computable by branching programs of size s can be reduced via projections to iterated multiplication of $s \times s$ matrices over $\mathbb{F}_2$. Does a similar result hold for NL?

Hence major open questions about computation are related to the following "purely mathematical question" that doesn't make any direct reference to computation:

Can (any one entry of) the product of s $s \times s$ matrices be reduced via projection to the product of s^d 3×3 matrices, for some constant d? (That is, the former product has s^3 variables x_i and each entry in the latter product is either a variable x_i or 0 or 1, all of this is over $\mathbb{F}_2$.)

Specifically, if the answer is positive then L $\subseteq$ NC^1. And if not then we have an explicit problem that is not in NC^1. More precisely, recall that iterated multiplication of matrices over $\mathbb{F}_2$ is complete for $\oplus$L (Theorem 7.21). Thus the "purely mathematical question" is equivalent to (the non-uniform version of) the question $\oplus$L = NC^1.

9.2 Group programs

The result in section §9.1 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 9.2. 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 9.4. Consider the alternative definitions of group programs:

(1) The program is given by two words $(g_1^{(b)}, g_2^{(b)}, \ldots, g_\ell^{(b)})$ for $b \in [2]$ and the output is $\prod_{i=1}^{\ell} g_i^{(x_{k_i})}$; that is, the bits specify from which word the element is to be taken.

(2) As in (1), but additionally we have another word of constants $(h_1, h_2, \ldots, h_\ell)$ and the output is $\prod_{i=1}^{\ell} h_i g_i^{(x_{k_i})}$.

Prove that (1) and (2) and Definition 9.2 are equivalent (up to changing α in Definition 9.2).

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 depicted at the beginning of the chapter). 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 9.1. This is worked out in Problem 9.1. 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 9.2. 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 4^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 8.2.

Exercise 9.5. Prove the in particular part, assuming the first first part.

To prove this theorem we begin with a lemma stating that the choice of the cycle is immaterial.

Lemma 9.1. 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 α and β are conjugate, that is, $\exists \rho \in S_5$ such that $\alpha = \rho^{-1}\beta\rho$. To see this let

$$\alpha = (\alpha_1, \alpha_2, ..., \alpha_5),$$

$$\beta = (\beta_1, \beta_2, ..., \beta_5),$$

$$\rho := (\alpha_1 \to \beta_1, \alpha_2 \to \beta_2, ..., \alpha_5 \to \beta_5).$$

Now suppose that

$$(g_1^0, ..., g_\ell^0), (g_1^1, ..., g_l^1), (k_1, ...k_\ell)$$

β-computes f; we claim that

$$(\rho g_1^0, ..., g_\ell^0 \rho^{-1}), (\rho g_1^1, ..., g_\ell^1 \rho^{-1})$$

(with the same indices k_i) α-computes f. To see this, note that

$$\prod_{i=1}^{\ell} g_i^{x_{k_i}} = 1_G \Rightarrow \rho^{-1} \prod_{i=1}^{l} g_i^{x_{k_i}} \rho = \rho^{-1} \cdot \rho = 1,$$

$$\prod_{i=1}^{\ell} g_i^{x_{k_i}} = \beta \Rightarrow \rho^{-1} \prod_{i=1}^{l} g_i^{x_{ki}} \rho = \rho^{-1}\beta\rho = \alpha.$$

QED

Lemma 9.2. If $f : \{0,1\}^n \to \{0,1\}$ is α-computable by a group program of length ℓ, so is $1 - f$.

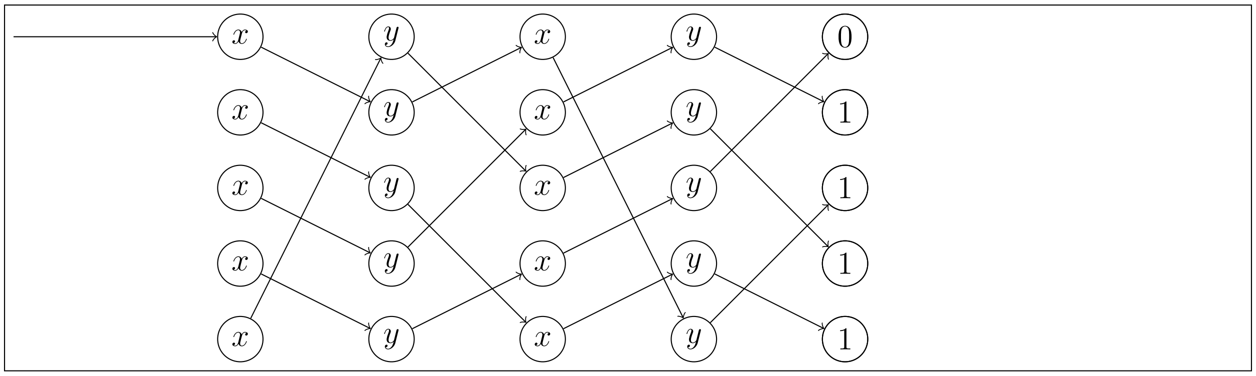


Figure 9.1: The width-5 permutation branching program for And on two bits x and y from the proof of Fact 9.1. All edges drawn have label 1. Every variable node has an edge with label 0 going to the corresponding node in the next column on the right.

Proof. First apply the previous lemma to α^{-1}-compute f. Then multiply last group elements g_ℓ^0 and g_ℓ^1 in the group program by α. **QED**

Lemma 9.3. 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: (α-computes f, β-computes g, α^{-1}-computes f, β^{-1}-computes g). If $f(x) = g(x) = 1$ then the concatenated program evaluates to $(\alpha\beta\alpha^{-1}\beta^{-1})$; otherwise evaluates to 1. **QED**

It only remains to see that we can apply the previous lemma while still computing with respect to a cycle.

Fact 9.1. $\exists \alpha, \beta$ cycles such that $\alpha\beta\alpha^{-1}\beta^{-1}$ is a cycle.

Proof. Let $\alpha := (12345)$, $\beta := (13542)$, we can check that $\alpha\beta\alpha^{-1}\beta^{-1}$ is a cycle. **QED**

Exercise 9.6. Check it.

For an illustration, see figure 9.1. 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 9.2. By induction on d using previous lemmas. **QED**

Exercise 9.7. Give the details of the proof.

The results work over other groups as well, see Problem 9.1.

Exercise 9.8. Describe the group generated by the matrices in Corollary 9.1. Compute the size of the group and compare it to the size of S_5.

The smallest group size that allows for this simulation is 60, met by the alternating group $A_5 \subseteq S_5$.

9.3 Recursive function evaluation

The input to this problem is a function $f : [2]^b \times [2]^b \to [2]^b$ and 2^h inputs $x_i \in [2]^b$. The goal is to compute recursive applications of f corresponding to a full binary tree. Let us introduce notation that is also useful later on. For a bit-string of length $\leq h$ we denote by g_u the constant function x_u if $|h| = u$, and $f(g_{u0}, g_{u1})$ otherwise. The goal is to compute g (of the empty string). This can be done in space cbh by a straightforward recursive approach. Specifically, to compute g_u we can make a recursive call to g_{u0} then *store the b output bits*, make a recursive call to g_{u1}, and finally apply f.

But in fact a better algorithm exists:

Theorem 9.3. Recursive function evaluation is in space $c(h \log b + b)$.

The techniques for proving this theorem are an extension of those in section §9.1. At a very high level, in the space-bounded model (as opposed to the algebraic circuits in section §9.1) we are able to carry through the techniques in section §9.1 bit by bit, thereby accumulating $\log b$ instead of b in recursive applications.

9.3.1 Warm-up: Assume f is linear

This already illustrates the main ideas.

TBD

9.3.2 The general case

TBD

9.4 Average-case complexity

Products over a group G enjoy the useful property of *random self-reducibility.* The basic idea is that given a sequence of group elements

$$g_1, g_2, g_3, \dots, g_n$$

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 $r_1, r_2, \dots, r_{n-1}$ uniformly in G and output

$$g_1 r_1^{-1}, r_1 g_2 r_2^{-1}, r_2 g_3 r_3^{-1}, \dots, r_{n-1}^{-1} g_n. \tag{9.1}$$

Note that the first $n-1$ elements are uniformly distributed, while the product of the n elements is the same as $\prod g_i$.

For groups such as S_5, this property is then inherited by all of NC^1 by Corollary 9.1.

As a first example of the usefulness of this fact we prove a result about the average-case hardness of computing group products. Recall that in Corollary 3.1 we saw that a model can compute a function w.h.p. over any input iff it can correlate well under *every* distribution. For group products, we can substantially strengthen this equivalence: Instead of requiring correlation under every distribution, it suffices to have correlation under the uniform distribution. We will again obtain this type of results in section 11.2.2, but only for models for which we don't know how to prove impossibility results, whereas the reduction here is very simple and in particular can be carried out in models for which we have impossibility results.

Claim 9.1. Let $G = S_5$ and suppose $f : G^n \to G$ equal $\prod g_i$ with probability p over the choice of a uniform input $g_1, g_2, g_3, \ldots, g_n \in G^n$. Then the following distribution on functions computes $\prod g_i$ with probability p on every input: Pick uniform $r_1, r_2, \ldots, r_n$ and compute

$$f(g_1 r_1^{-1}, r_1 g_2 r_2^{-1}, r_2 g_3 r_3^{-1}, \ldots, r_{n-1} g_n r_n^{-1}) \cdot r_n.$$

Exercise 9.9. Prove this.

This result can be used to relate the complexity of NC^1 to its average-case complexity, see Problem 9.3.

9.5 Cryptography in NC^0

One-way functions are easy to compute but hard to invert. Let us first define hard to invert.

Definition 9.3. A function $f : X \to [2]^m$ is ϵ-hard to invert for size s if for every circuit C of size s:

$$\mathbb{P}_{x \in X}[f(C(f(x))) \neq f(x)] \geq \epsilon.$$

In words, the circuit C, given $f(x)$, fails to find a pre-image of $f(x)$, that is, a value y s.t. $f(y) = f(x)$. Most modern cryptography is based on efficient one-way functions. In fact, common candidates one-way functions are computable in NC^1. The following surprising result shows that *if there are one-way functions in NC^1 then there are one-way functions computable with constant locality, i.e., in NC^0*. To simplify the presentation we consider partial functions, see the notes and Problem 9.4.

Theorem 9.4. [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 with locality c and is ϵ-hard to invert for size s', where $n' \leq n + m \cdot (4^d - 1)$, $m' \leq m \cdot 4^d$ and $s' \geq s - cm'$.

Proof. By 9.2, each output bit of f is computable by a group program of length 4^d over S_5 (note that the group program multiplies to 1 of α, so we can think of it as a boolean value). The new function f' takes as input x as well as $m \times (4^d - 1)$ group elements $r_{i,j}$ and outputs m tuples $R^1, R^2, \ldots, R^m$ of 4^d group elements, where tuple i is a uniform tuple except the product equals bit i of $f(x)$, as in 9.1.

Suppose a circuit C' inverts f' with probability $1 - \epsilon$.

Then to invert f we proceed as follows. On input $y \in [2]^m$ run C' on $(R^1, R^2, \ldots, R^m)$ where R^i is a uniform tuple with product equals to y_i, to obtain $(x, r_{i,j})$. Output x.

Note that input distribution to C' is the same as the output of f' on a uniform input. So with probability $1 - \epsilon$ we have that the computed $(x, r_{i,j})$ is a pre-image of f'. From which it follows that x is a valid pre-image of y.

Finally, f' can be computed with locality c because each output element depends only ≤ 3 elements, as in 9.1. **QED**

Exercise 9.10. Suppose we instead define f' to simply output the programs corresponding to the input (i.e., let $r_{i,j} = 1$ everywhere). Which steps breaks in the proof?

The result in Theorem 9.4 that OWF in NC^1 imply OWF in NC^0 can be improved to show that even OWF in $\oplus$·L imply OWF in NC^0. The elegant proof works by considering a suitable group of matrices over $\mathbb{F}_2$.

9.6 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 9.4. 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 9.5. $\oplus \cdot \mathrm{L} \subseteq \mathrm{CL}$.

Using similar techniques in combination with CRR Theorem 7.5 one can extend this to $0-1$ matrices over the integers, in particular obtaining NL $\subseteq$ CL. Even more, one can prove that log-depth threshold circuits (TC^1) is in CL.

Proof. By Theorem 7.21 it suffices to show the iterated product of matrices (of any dimension) over $\mathbb{F}_2$ is in CL. The critical observation is that Theorem 9.1 works over every ring. We consider the ring of $\mathbb{F}_2$ matrices. The extra memory consists of the three registers, each holding a matrix. Before starting the computation, we read off the bit b from the extra memory which will then be xored with the value we want to compute.

We then use Theorem 9.1, for the formula consisting of iterated product of matrices. Xoring with b gives us the output. We then repeat Theorem 9.1 to clear the register and thus restoring the extra memory to its initial state.

Each operation in the program can be computed in log-space **QED**

9.7 Problems

Problem 9.1. (cf. Problem 8.5) Let G be a non-solvable group, equivalently, a group G which has a non-trivial subgroup whose commutator subgroup (i.e., the subgroup generated by commutators) is itself. Prove that any function computable in depth d is α-computable by a program of length $\leq (4g)^d$ over G, for any α.

Problem 9.2. (Cf. Problem 8.5) You are given as input a sequence of elements of two types: (a, b) where $a, b \in \mathbb{Z}$, or z which is a special "flip" symbol. The elements obey the following rules:

$$\begin{aligned} (a, b)(a', b') &= (a + a', b + b'), \\ z(a, b) &= (b, a)z, \\ zz &= (0, 0). \end{aligned}$$

(In group theory terminology, this is the wreath product $\mathbb{Z} \wr \mathbb{Z}_2$.)

Show that deciding if the product of a sequence of elements equals $(0, 0)$ is in TC. For example, $(1, 2)z(-2, -1)z = (0, 0)$.

Hint: First compute in TC an equivalent sequence with at most one z.

Problem 9.3. Suppose there are power-size, constant-depth TCs that compute $\prod_{i\in[n]} g_i$ with probability $1/2 + 1/n^{10}$ over uniform $g_i \in S_5$. Show $\mathrm{NC}^1 = \mathrm{TC}$ for non-uniform circuits.

Problem 9.4. Show that in Theorem 9.4 we can have total functions (i.e., $X = [2]^n$ and $X' = [2]^{n'}$) with a similar construction, if we only want f' in AC and that f' is $(\epsilon - 1/n^{\omega(1)})$-hard to invert. (Hint: Use Problem 8.6.)

9.8 Notes (& mini historical vignette)

Group programs were introduced in the 60's in [182, 161] 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 [244].

Group programs were rediscovered in the 80's as permutation branching programs of constant width. Using essentially the same construction in [182, 161], it was shown in [189] that NC^1 equals the set of functions computable by power-length group programs, over any non-solvable group. ([189] 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 [182, 161], where the domain is G^n.) After [189], several related simulations were discovered, such as [43]. section §9.2 and Problem 9.1 are from [189]. Theorem 9.1 is a variant of this result from [43]. The proof we presented follows [61].

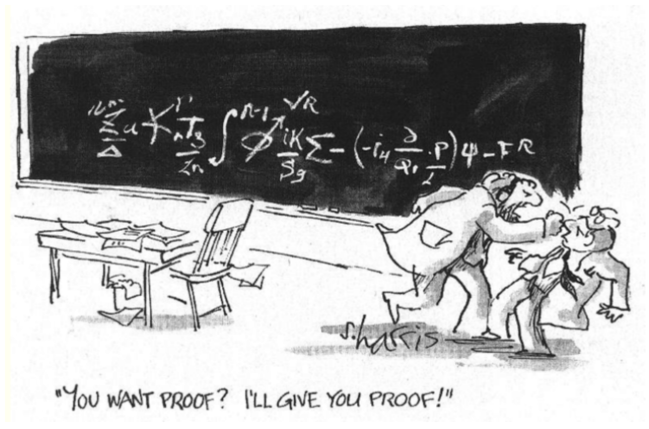
Recursive function evaluation was introduced in [65]. Theorem 9.3 is from [62]. Our exposition builds on [100].

Theorem 9.4 is from [19]. 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 [276].

Catalytic computation was introduced in [51]. The text at the beginning of section §9.6 is their introductory paragraph.

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 section 5.1.1, 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 10.1. For any input length n, V in Definition 10.1 can be taken to be a 3CNF of size n^d.

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 ask for more. 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 10.1. 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 10.2. 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.

Exercise 10.2. Prove Theorem 10.2. For the "only if" use the the (advanced) result that any problem in NP can be map reduced to 0.01-Gap-3Sat (which is essentially Theorem 4.10, except we did not claim map reductions or a specific constant there).

10.2 Proofs that yield nothing but their validity: Zero-knowledge

In Theorem 10.2 the verifier gains "constant 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. 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 learn 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 verifiers 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 a physical implementation that can astonish the right audiences.

10.3 Interactive proofs

We now consider interactive proofs. Here 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 the prover simply sending y to V.

It turns out that it suffices for the verifier to send uniformly random strings $Q_1, Q_2, \ldots$ bits to P. This leads to a simple definition.

Definition 10.2. A function $f : X \subseteq [2]^* \to [2]$ is in *interactive power time* (a.k.a. admits a power-time interactive proof) abbreviated IP, if 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
$$V\left(x, P(Q_1), P(Q_1, Q_2), \ldots, P(Q_1, Q_2, \ldots, Q_b)\right) = 1$$
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
$$\mathbb{P}_{Q_1, Q_2, \ldots, Q_b \in [2]^b}\left[V\left(x, P(Q_1), P(Q_1, Q_2), \ldots, P(Q_1, Q_2, \ldots, Q_b)\right) = 1\right] \le 1/3.$$

The following amazing result shows the power of interactive proofs, compared to non-interactive proofs. We can think of NP as "reading a book" and IP as "going to class and asking questions." We don't yet know how to replace teachers with books.

Theorem 10.3. IP = PSpace.

As a first step towards the proof of Theorem 10.3 we show that IP contains problems not known to be in NP. The protocol in this result is known as the *sum-check protocol*.

Theorem 10.4. Given a field $\mathbb{F}$, an arithmetic circuit $C(x_1, x_2, \ldots, x_v)$ over $\mathbb{F}$ computing a polynomial of degree $d \leq |C|$, and an element $s \in \mathbb{F}$, deciding if

$$\sum_{x_1, x_2, \ldots, x_v \in [2]} C(x_1, x_2, \ldots, x_v) = s \tag{10.1}$$

is in IP, whenever $(1 - d/|\mathbb{F}|)^v \geq 2/3$.

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 polynomial p of degree d in the variable x, which is meant to be

$$s'(x) := \sum_{x_2, x_3, \ldots, x_v \in [2]} C(x, x_2, x_3 \ldots, x_v).$$

V checks if $p(0) + p(1) = s$, and if not rejects. Otherwise, it recursively runs the protocol to verify that

$$\sum_{x_2, x_3, \ldots, x_v \in [2]} C(Q_1, x_2, x_3, \ldots, x_v) = p(Q_1). \tag{10.2}$$

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.

In case equation (10.1) is false, $s'(0) + s'(1) \neq s$. Hence, unless V rejects right away because $p(0) + p(1) \neq s$, we have $p \neq s'$. The polynomials p and s' have degree $\leq d$. Hence by Lemma 2.1

$$\mathbb{P}_{Q_1}[p(Q_1) \neq s'(Q_1)] \geq 1 - d/|\mathbb{F}|.$$

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 $\geq (1 - d/|\mathbb{F}|)^v$. **QED**

Exercise 10.3. 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.

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:

$$\begin{aligned} x &\to x \\ f \wedge g &\to f \cdot g \\ f \vee g &\to f + g - f \cdot g \\ \neg f &\to 1 - f. \end{aligned}$$

Exercise 10.4. Given a 3CNF formula ϕ and $k \in \mathbb{N}$, deciding if ϕ has exactly k satisfying assignments is in IP.

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 Exercise 10.4 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 integer 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 10.1. Let
$$f(x, y_1, \ldots, x_k) := \exists x \forall y_1 y_2 \cdots y_k : x.$$
(This simple example suffices for the point made; one can always use the y_i in trivial ways such as $(y_i \vee \neg y_i)$ to get a more complicated expression involving the y_i making the same point.)

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 10.1. Let $f = Q_1x_1Q_2x_2 \cdots Q_kx_kg(x_1, x_2, \ldots, x_k)$ be a formula where g is quantifier-free. All quantifiers are over [2]. Proceeding from left to right, replace an occurrence of

$$\forall x_i \exists x_{i+1} \cdots Q_kx_k : g(x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \cdots, x_k)$$

with the formula

$$\forall x_i \exists x'_1, x'_2, \ldots, x'_{i-1} : \left(\left(\wedge_{j=1}^{i-1} x_j \iff x'_j \right) \wedge \exists x_{i+1} \cdots Q_kx_k : g(x'_1, x'_2, \ldots, x'_{i-1}, x_i, x_{i+1}, \cdots, x_k) \right).$$

At the end of the process we have obtained f' s.t.:

(1) $f \iff f'$,

(2) $|f'| \le |f| + ck \cdot D$, where D is the number of $\forall$ quantifiers in f, and

(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 $(x_j \wedge x'_j) \vee (\neg x_j \wedge \neg x'_j)$.

Proof. (1) is evident by the transformation rule.

(2) holds because for each $\forall$ quantifier we add a string which is linear in the number of variables in f, which is $\le k$.

(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 $\le$ 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 10.2. Let us return to Example 10.1 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 10.1 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 10.3).

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, cf. Theorem 7.13, and are omitted.

For the other direction it suffices to show that QBF $\in$ IP by Theorem 7.13. Given a QBF f, the verifier applies the transformation in Lemma 10.1 to obtain f'. The verifier then computes the arithmetization g of f'. Recall $\exists x \in [2]$ in f' becomes $\Sigma_{x\in[2]}$ in g and $\forall x \in [2]$ becomes $\prod_{x\in[2]}$. Hence g denotes a number and we note

$$f \text{ is true} \iff f' \text{ is true} \iff g > 0.$$

The prover will show that $g > 0$ by first sending a prime p and then proving $g = K$ over the field $\mathbb{F}_p$ using the sum-prod-check protocol. Note that g can be, and is at most, doubly exponential in n. By Theorem 2.5 a prime of n^c bits can be found so that $g \neq 0$ over $\mathbb{F}_p$. **QED**

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 P 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).

10.4 Merlin-Arthur

The special case of interactive proofs where the number of rounds is bounded is known as Arthur-Merlin protocols. Here Merlin stands for the all-powerful prover and Arthur for the computationally bounded verifier.

A basic class is the one in which Merlin sends a proof, and then Arthur verifies it.

Definition 10.3. A function $f : X \subseteq [2]^* \to [2]$ is in MA if there is $V \in$ P and $d \in \mathbb{N}$ such that for every $x \in [2]^n$, letting $b := n^d$:

- If $f(x) = 1$ then $\exists P \in [2]^b$ such that $V(P, R) = 1$ for every $R \in [2]^b$.
- If $f(x) = 0$ then $\forall P \in [2]^b$ we have $\mathbb{P}_{R\in[2]^b}[V(P, R) = 1] \leq 1/3$.

The definition of MA is similar to that of $\exists \cdot \text{BP} \cdot \text{P}$. But there is one key difference, if $f(x) = 1$ we only have guarantees on the number of R that cause $V(P, R)$ to accept for a valid proof P. But in $\exists \cdot \text{BP} \cdot \text{P}$ we would need guarantees for every P, because in Definition 6.3 we did not consider partial functions.

From the definition, we have MA $\subseteq \Sigma_2$P. Recall Exercise 6.7 showing that if Exp is in PCkt then Exp is in Σ_2P. The same implication holds for other classes. The theory of interactive proofs allows us to strengthen Σ_2P to MA. In fact, we can prove equivalence with MA.

Theorem 10.5. $\text{Maj} \cdot \text{P} \subseteq \text{PCkt} \Rightarrow \text{P}^{\# \cdot \text{P}} \subseteq \text{MA}$.

Proof. The assumption implies that $\text{P}^{\# \cdot \text{P}} \subseteq \text{PCkt}$. Inspection of the protocol in Exercise 10.4 reveals that the prover can be implemented in $\text{P}^{\# \cdot \text{P}}$. So Merlin can send a circuit for the prover, and Arthur can run the protocol by himself. This allows Arthur to answer each query to the $\# \cdot \text{P}$ oracle by himself. Note in Exercise 10.4 the output of the $\# \cdot \text{P}$ query is given as input, but Arthur can run the circuit once to obtain this value, and then verify it. **QED**

Corollary 10.1. $\text{Maj} \cdot \text{P} \subseteq \text{PCkt} \Rightarrow \text{PH} = \text{Maj} \cdot \text{P} = \text{P}^{\# \cdot \text{P}} = \text{MA}$.

Proof. By essentially the same proof as Problem 6.6, we have $\text{MA} \subseteq \text{Maj} \cdot \text{P}$. Also, $\text{MA} \subseteq \text{PH} \subseteq \text{P}^{\# \cdot \text{P}}$, where the first containment is by definition, and the second is Corollary 6.1. The result follows by Theorem 10.5. **QED**

With this machinery we can now prove Theorem 6.7.

Proof of Theorem 6.7. If $\text{Maj} \cdot \text{P}$ is not in PCkt we are done. Otherwise, by Corollary 10.1 $\text{Maj} \cdot \text{P}$ equals PH, and we conclude by Exercise 6.6. **QED**

10.5 Interactive proofs within P

In this section we discuss interactive proofs for problems in P. The goal is having the verifier run in linear or quasi-linear time, and the prover run in power time.

Exercise 10.5. Show that any function in L has interactive proofs where the verifier runs in quasi-linear time. Are the proofs in your protocol computable in P?

The following stronger result is known.

Theorem 10.6. Any function in L has interactive proofs where the verifier runs in quasi-linear time, and moreover when a proof exists it can be computed in P.

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 10.6 displays a beautiful interplay between algebra and computation, and in fact, we will establish stronger results (applying to NL and other classes). Before this, however we give a simple example where an efficient prover exists. This serves as a warm-up for the proof of Theorem 10.6.

10.5.1 Counting triangles

We consider the problem of counting the number of triangles in a graph. Such a problem can be trivially solved in time n^c. Whether faster run time such as $n\log^c n$ are possible is unknown. We show that such time bounds can be achieved via interaction, and moreover the proofs are still feasible.

Theorem 10.7. 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. Moreover, the verifier runs in time $m\log^c m$ and the prover runs in P.

Proof. We construct a suitable arithmetic circuit and then apply the sum-check protocol from Theorem 10.4. 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 name in the graph. Thus, the number of triangles can be written as

$$\sum_{x_1,x_2,\ldots,x_v\in[2]} \prod_{i,j\in[3],i<t} E(y_i, y_j)$$

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).

For this, we define the following polynomial

$$\widehat{E}(z, z') := \sum_{\alpha,\alpha'} (z = \alpha) \cdot (z' = \alpha') \tag{10.3}$$

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'.

With this notation, the verifier needs to verify that

$$\sum_{x_1,x_2,\ldots,x_v\in[2]} \prod_{i,j\in[3],i<t} \widehat{E}(y_i, y_j).$$

Now $\widehat{E}$ is a polynomial of degree $2\log n$. Hence the whole expression is a polynomial of degree $d \leq c\log n$. We run the sum-check protocol over a field $\mathbb{F}$ size $q := \log^c n$. The correctness follows from Theorem 10.4.

Let us now verify the running times. The prover at each round sends a univariate polynomial of degree d which is obtained by summing over n^c values. Hence the prover is in P. The verifier at each round except the last one needs to evaluate a polynomial of degree d over a field of size q, and perform a constant number of field operations. This takes time d^c. At the last round, the verifier needs to evaluate

$$\prod_{i,j\in[3],i<t} \widehat{E}(y'_i, y'_j)$$

for some fixed $y'_i \in \mathbb{F}$. 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. **QED**

Exercise 10.6. Modify the proof so that the protocol has only a constant number of rounds (i.e., the verifier only asks a constant number of questions), while the verifier still runs in time $c_t m \log^c m$. Hint: You may need to modify the sum-check protocol.

10.5.2 Proof of Theorem 10.6

As mentioned, we actually prove stronger results 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 stating this we have to discuss how to encode gates in the circuit.

Encoding gates We shall consider functions computable by circuits of power-size 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 gates into a matrix of $d(n)$ rows and n^e columns, where row 0 corresponds to the output and row $d-1$ to the input. This way, we can always index the gates in a level by $\log n^e$ bits. Unused gates can be set to the constant 0. Any gate in a circuit is indexed by $\log(dn^e)$ bits.

Also, 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

$$m := (\log(dn^e))/\log|\mathbb{H}|$$

elements from $\mathbb{H}$. In terms of bits, this is $\log(dn^e)$ like before.

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 will still be $c\log(dn^e)$ if $|\mathbb{F}| \le |\mathbb{H}|^c$. Indeed, we set

$$|\mathbb{H}| := \log n$$
$$|\mathbb{F}| := \log^{c_e} n.$$

The connection function The *algebraic connection function* $\phi : (u, v, w) \in \mathbb{H}^{3m} \to [2]$ indicates if gate u in C_n takes as input gates v and w.

Definition 10.4. A function $f : [2]^* \to [2]$ computable by circuits of size n^e and depth $\log^e n$ has *efficient algebraic connections* if the circuit C_n for $f : [2]^n \to [2]$ has *an algebraic connection function* $\phi : (u, v, w) \in \mathbb{H}^{3m} \to [2]$ computable by a polynomial that can be written down in time $\log^e n$ and has degree $\log^e n$.

We used the same parameter e for both the complexity of f and 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 is not typically an issue, in the sense that it's hard to find examples where it is hard to satisfy it. Still, let us now verify that the circuits for L indeed have explicit algebraic connections. Rather than working directly over the field $\mathbb{H}$, we shall first prove that the connection function over bits (as opposed to field elements) has efficient ACs, and then use a generic transformation.

Exercise 10.7. Let $f \in L$. The circuits $C : [2]^n \to [2]$ for f can have the following structure (part of which is the same as in Theorem 8.1). Let $a = c_f$. There is a layer A of n^a gates at distance 1 from the input, 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. The output copy has a specified gate as output.

(1) Explain what A and M compute.

(2) Prove that C has efficient AC connections: In time $\log^{c_f} n$ we can compute AC circuits for the connection function $\phi : (u, v, w) \in [2]^{3m \log |\mathbb{H}|} \to [2]$. (In particular, the circuits have size $\log^{c_f} n$.)

(3) Prove that C has efficient algebraic connections.

Main statement and proof Thanks to Exercise 10.7, Theorem 10.6 follows from the following statement about circuits.

Theorem 10.8. Let $f : [2]^* \to [2]$ have circuits of size n^e, depth $\log^e n$, and efficient algebraic connections. Then f has interactive proofs where the verifier runs in time $n \log^{c_e} n$ and the prover is in P.

In the remainder of this section we present the proof of 10.8. Given input x, let α_i denote the value of the gates at distance i from the output. So α_0 is the output and α_{d-1} is the input. The protocol proceeds in d stages. In stage $i-1$ a claim of the form

$$\widehat{\alpha}_{i-1}(z_{i-1}) = b_{i-1}$$

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 $\widehat{\alpha} : \mathbb{F}^m \to \mathbb{F}$ is the arithmetization of α which agrees with α over $\mathbb{H}^m$ and is defined as

$$\widehat{\alpha}(x) := \sum_{y \in \mathbb{H}^m} \mathrm{EQ}(x, y) \cdot \alpha(y).$$

where $\mathrm{EQ}(x, y)$ is a polynomial that, when evaluated over $\mathbb{H}^m$, computes equality. This is known as *low-degree extension*.

Exercise 10.8. Give such a polynomial for EQ that has degree $m(\mathbb{H} - 1)$ and can be evaluated in time $\log^{c_e} n$. In fact, time $m \log^c d(n)$ suffices. Hint: Use that $x^{q-1} = 1$ for $x \neq 0$ in a field of size q.

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 d^c) 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 $i = 1$ we have that z_0 is the output gate and b_0 is the output of the circuit.

Exercise 10.9. Explain how the claim at Stage $i = d$, corresponding to the input level, is verified in time $n \log^{c_e} n$, without further interaction.

The induction step: Stage $i - 1$. W.l.o.g. assume the circuit consists of NAnd gates only. We have

$$\widehat{\alpha}_{i-1}(z) = \sum_{u,v,w \in \mathbb{H}^m} \mathrm{EQ}(z, u) \cdot \phi(u, v, w) \cdot (1 - \widehat{\alpha}_i(v) \cdot \widehat{\alpha}_i(w)).$$

Note that on the rhs we could have written α_i instead of $\widehat{\alpha}_i$. Indeed, they agree on $\mathbb{H}^m$. However, we need to apply the sum-check protocol so we need algebraic computation.

The current claim is that the rhs equals b_{i-1}. 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 claim to

$$\mathrm{EQ}(z, \widehat{u}) \cdot \phi(\widehat{u}, \widehat{v}, \widehat{w}) \cdot (1 - \widehat{\alpha}_i(\widehat{v}) \cdot \widehat{\alpha}_i(\widehat{w})) = b,$$

for some $\widehat{u}, \widehat{v}, \widehat{w} \in \mathbb{F}^m$ and $b \in \mathbb{F}$. (We only stated the protocol for sums over [2], but one can readily extend it to sums over larger sets such as $\mathbb{H}$.) This requires m iterations. 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 the size of $\mathbb{F}$. Overall, the error is $\leq 1/\log^{c_e} n$.

However, the new claim appears to require two evaluations of $\widehat{\alpha}_i$, 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 $\widehat{\alpha}_i$, which is meant to be $p(t) := \widehat{\alpha}_i(\widehat{v} + t(\widehat{w} - \widehat{v}))$. The verifier checks that

$$\mathrm{EQ}(z, \widehat{u}) \cdot \phi(\widehat{u}, \widehat{v}, \widehat{w}) \cdot (1 - p(0) \cdot p(1)) = b$$

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

$$\widehat{\alpha}_i(\widehat{v} + t(\widehat{w} - \widehat{v})) = p(t).$$

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 $\leq \log^e n/\log^{c_e} n \leq \log^{c_e} n$.

10.6 A PCP from the sum-check protocol

First we describe a specific *interactive* proof for the NP-complete problem Quad-Sys, which is slightly more convenient to work with than 3Sat. Then we turn this protocol into a *non-interactive* one, via a trivial brute-force transformation, under the assumption that the proof has some special structure. This protocol is efficient assuming that the prover is restricted to send a low-degree proof. Finally, we show how to remove this restriction on the prover.

10.6.1 An interactive protocol for Quad-Sys (Problem 4.5)

We can write an instance of Quad-Sys of size n as

$$\forall i \in [n] : \sum_{j,k \in [n]} \alpha(i, j, k)x(j)x(k) + \sum_{j \in [n]} \beta(i, j)x(j) + \widehat{\gamma}(i) = 0.$$

Here i indices the equation, $\alpha(i, j, k)$ the coefficient of $x(j)x(k)$ in Equation i, and so on.

As in section 10.5.2 we need low-degree extensions to larger fields. We use the same parameters $\mathbb{H}$, $\mathbb{F}$ there. So can think of i, j, k as ranging in $\mathbb{H}^m$, where $m := \log n/\log |\mathbb{H}|$.

In the protocol, first the prover sends a map $\widehat{w} : \mathbb{F}^m \to \mathbb{F}$, which is meant to be the low-degree extension of a satisfying assignment $w : \mathbb{H}^m \to [2]$. The verifier then needs to verify

$$\forall i \in \mathbb{H}^m : \sum_{j,k \in \mathbb{H}^m} \widehat{\alpha}(i, j, k)\widehat{w}(j)\widehat{w}(k) + \sum_{j \in \mathbb{H}^m} \widehat{\beta}(i, j)\widehat{w}(j) + \widehat{\gamma}(i) = 0. \tag{10.4}$$

To do, it will pick a uniform $i \in \mathbb{F}^m$ and verify the corresponding equation. This verification is done via the sum-check protocol. (One can rewrite the equation as a single sum to match the way we stated the sum-check protocol.)

To analyze the correctness of fixing i, rewrite equation (10.4) as

$$\forall i \in \mathbb{H}^m : f(i) = 0$$

for a suitable f. This is equivalent to

$$\forall i \in \mathbb{F}^m : \widehat{f}(i) = 0$$

as can be verified by the definition of low-degree extension. However, $\widehat{f}$ is a polynomial of degree $\leq m\mathbb{H}$ in each variable. Hence by Lemma 2.1, if it is not zero then $\mathbb{P}_{i \in \mathbb{F}^m}[\widehat{f}(i) = 0] \leq m\mathbb{H}/\mathbb{F}$.

The rest of the correctness follows by the correctness of the sum-check protocol. However this is only guaranteed if the prover sends a low-degree polynomial.

10.6.2 A non-interactive protocol

To make the protocol non-interactive, we simply let the prover send its answers for every verifier's message. Note the number of possible messages from the verifier is $\leq n^c$. Each prover message also has length $\leq n^c$. Hence the total length of the proof is $\leq n^c$.

The verification time is $\leq \log^c n$.

10.6.3 Low-degree testing

Finally, we explain how the verifier can verify that $\widehat{w}$ is indeed a low-degree polynomial.

TBD

10.7 Problems

Problem 10.1. TBD Prove that $\Sigma_2\text{P} \subseteq \text{IP}$.

Prove that $\oplus \cdot \text{P} \subseteq \text{IP}$.

You cannot use the result that IP = PSpace.

Problem 10.2. Improve Theorem 10.8 to constant rounds for functions in AC.

To isolate the essence of the problem, let f be a function computable by power-size, cosntant-depth AC with fan-in $n^{0.1}$. Suppose that the circuit C_n for $f : [2]^n \to [2]$ has *an algebraic connection function* $\phi : (u, w_1, w_2, \ldots, w_t) \in \mathbb{H}^{(t+1)m} \to [2]$ which outputs 1 iff u takes as inputs $w_1, w_2, \ldots, w_t$ and which is computable by a polynomial that can be written down in time $\sqrt{n}$ and has degree $\sqrt{n}$ (cf Definition 10.4).

Prove that f has interactive proofs where (1) the verifier runs in quasi-linear time, (2) the prover is in P, and (3) the number of rounds is constant.

Hint: Modify the protocol in the proof of Theorem 10.8 following Exercise 10.3.

10.8 Notes

Interactive proofs were put forth simultaneously in [27, 106]. The latter paper also introduced zero-knowledge. The zero-knowledge protocol for 3Color is from [101].

Theorem 10.3 is from [177, 231], with the last paper proving it as stated. For the history of this famous result see [22].

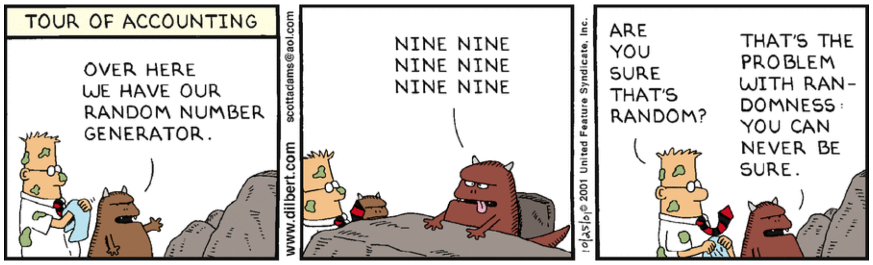
The proof of Theorem 6.7 follows [1].

Interactive proofs for functions in P with efficient verifier were first studied in [105], where essentially Theorem 10.8 appears. Our presentation of this result follows [99]. The latter differs from the former in the way the uniformity of circuits is handled. Our presentation also differs and uses that the circuits for simulating L or NL are sufficiently uniform. In fact, this was studied already in [144] where even constant-locality uniformity is achieved. Theorem 10.7 is from [103]. [221] give protocols that are even constant-round for TiSP. [256] gives alternative arguments (not achieving constant-round) based on matrix powering. Simpler constant-round protocols for smaller classes are in [104]. For a survey of some of these works, see [99].

For more on interactive proofs and zero knowledge see the book [256].

Chapter 11

Pseudorandomness


TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
NINE NINE NINE NINE NINE NINE
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.
www.dilbert.com
scottadams@aol.com
© 2001 United Feature Syndicate, Inc.

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

$$0101010101010101010101010101010101010101.$$

You'd probably think this can't be true. But suppose instead I claim that I got

$$1000111110100010011110100101111101100100.$$

Maybe you would think this is possible then? But why do we 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? Of course, we've been using randomness all along since the first chapter. Still, let's step back and consider three possible answers:

1. **Classical:** Each string has the familiar probability. This viewpoint is useful in mathematics but the example above shows that it doesn't capture our intuitive notion of randomness.

2. **Intrinsic (or ontological):** A string is the less random the shorter 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 1000111110100010011110100101111101100100."

3. **Behaviouristic:** Randomness is in the eyes of the beholder: A string R is random for f if f can't distinguish it from a truly random string. In other words, R fools f into thinking that R is random.

The last answer seems the most useful, and we now make it precise.

Definition 11.1. A distribution R over $[2]^n$ *ϵ-fools* (or is ϵ-pseudorandom for) a function $f : [2]^n \to [2]$ if $|\mathbb{E}[f(R)] - \mathbb{E}[f(U)]| \le \epsilon$, where U is uniform in $[2]^n$. A function f ϵ-breaks (or tells, distinguishes) distributions D and E if $|\mathbb{E}[f(D)] - \mathbb{E}[f(E)]| \ge \epsilon$. If E is omitted it is assumed to be the uniform distribution.

We are naturally interested in distributions that are pseudorandom yet have very little entropy. As in Chapter 3, counting arguments show that very little entropy is needed for non-explicit distributions, about logarithmic in the number of tests to be fooled. But our ability to explicitly construct such distributions is limited by the grand challenge:

Claim 11.1. 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 $F =$ PCkt and R can be sampled in P then $g \in$ NP, and so NP $\not\subseteq$ PCkt.

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 3.2.

To simplify the following discussion, we introduce the notion of a pseudorandom generator which makes it easier to talk about the entropy of the distribution and its explicitness.

Definition 11.2. An algorithm G is a *pseudorandom generator*, abbreviated generator or PRG, that ϵ-fools a class F of functions (or a generator for F with error ϵ) with seed length s (a function of both n and ϵ) if on input n and ϵ, and a uniform seed U of length $s(n, \epsilon)$, G outputs a distribution on n bits that ϵ-fools any function in F on inputs of length n. The *stretch* is $n - s(n, \epsilon)$.

Note that we use n to denote the *output length* of G, because it is the *input length* for a test that's trying to tell G from random. Also, we typically have the output length of G much longer than the input length. For some applications, it suffices if G is computable in power-time in the output length, which can be exponential in the input length. We shall simply say that G is *explicit* in this case. However many generators we present below, especially those for restricted models, are explicit in a stronger sense: Given an input and an index to an output bit, that bit can be computed in P.

Exercise 11.1. Suppose there is $a > 0$ and an explicit generator with seed length $s(n) = a \log n$ that 0.1-fools circuits of size n, for a constant a. Prove that P = BPP.

Note that to eliminate one parameter we set the size of the circuit test equal to the output length of G. Recall from Definition 1.4 that input gates are not counted towards size; the circuit may simply ignore most of its input bits, which makes sense since very few input bits suffice, information-theoretically, to tell the output of G from uniform.

Given Claim 11.1, there are two main avenues for research, closely paralleling the development of earlier chapters. The first is proving unconditional results for restricted models, like AC. Actually, pseudorandomness being a more refined notion of impossibility, even very simple models like local functions are non-trivial, and results for them very useful. The second is proving reductions, that is linking the existence of PRGs to other conjectures. Interestingly, some of the techniques are general and apply to both settings.

11.1 Basic PRGs

In this section we present PRGs for several basic classes of tests. Besides being basic, these tests are the backbone of several other constructions, and somewhat surprisingly suffice to fool apparently stronger classes of tests.

11.1.1 Local tests

The simplest model to consider is perhaps that of local functions.

Definition 11.3. A distribution over $[2]^n$ is *k-wise uniform* if every k bits are uniform in $[2]^k$ (equivalently, any k-local function is 0-fooled). A k-wise generator is a map whose output distribution is k-wise uniform.

Exercise 11.2. Given an explicit 1-wise uniform generator with seed length $s(n) = 1$.

Theorem 11.1. There are explicit k-wise uniform generators with seed length $s = ck \log n$.

Proof. Wlog assume n is a power of 2 and let $\mathbb{F}$ be the field of size n. More generally, the range of G will be $\mathbb{F}^n$, and the distribution of any k coordinates will be uniform over $\mathbb{F}^k$. View the input as coefficients a_i $i \in [k]$ of a polynomial p of degree $k-1$. Define the i output element of G to be $p(i)$. Any k-tuple of field elements is uniform, for if two different polynomials give the same tuple then their difference is a non-zero polynomial of degree $k-1$ with $\geq k$ roots, violating Lemma 2.1. (We can assume $k \leq n$ for else the theorem is trivial.) **QED**

The bound on s is tight for $k \leq n^c$. For k closer to n different arguments apply, but we won't need them here.

Theorem 11.2. The minimum seed length is $\geq ck \log(2n/k)$.

Proof. Think of the support as $\{-1, 1\}^n$, and write down a $2^s \times n$ matrix where row x is $G(x)$. For even k, and for any $T \subseteq [n]$ of size $k/2$, consider the 2^s-long vector v_T obtained by multiplying together the columns indexed in T. Note that the v_T are orthogonal, hence independent (A.13), and so $2^s \geq \binom{n}{k/2}$, whence $s \geq ck \log(2n/k)$. **QED**

Powerlog-wise uniformity suffices to fool AC:

Theorem 11.3. Any $\log(m/\epsilon)^{cd}$-wise uniform distribution over $[2]^n$ ϵ-fools AC of size m and depth d.

In particular, there are explicit generators that ϵ-fool such circuits with seed length $\log(m/\epsilon)^{cd}$. We give generators with this seed length below, via a different route.

11.1.2 The power of NC^0: Local maps can fool local tests

To compute the construction in the proof of Theorem 11.1 it seems we need to read the entire input of $ck \log n$ bits. We now give a different construction where it suffices to read $c \log n$ bits. Note this is a dramatic saving when k is large. In fact we give a general tradeoff between the locality and the seed length.

Theorem 11.4. There are k-uniform generators $G : [2]^s \to [2]^n$ that are d-local, whenever

$$\left(\frac{cdk}{s}\right)^{d/2} \le \frac{1}{n}.$$

For example, we can have $s = ck \log n$ and $d = c \log n$. At the other extreme, we can also have $s = n^{0.1}$ and $d = c$. It even suffices if the seed is dk uniform as opposed to completely uniform.

Proof. Pick a random bipartite graph with s nodes on the left and n nodes on the right. Every node on the right side has degree d and computes the XOR of its neighbors. By Fact A.17 it suffices to show that for any non-empty subset $S \subseteq [n]$ of size $\le k$, the XOR of the corresponding bits is unbiased. For this it suffices that S has a unique neighbor. For that, in turn, it suffices that S has a neighborhood of size greater than $\frac{d|S|}{2}$ (because if every element in the neighborhood of S has two neighbors in S then S has a neighborhood of size $< d|S|/2$). We pick the graph at random and show by standard calculations that it has this property with non-zero probability. Write $N(S)$ for the set of neighbors of nodes in S. We need to bound

$$\mathbb{P}\left[\exists S \subseteq [n], 0 < |S| \le k, \text{ s.t. } |N(S)| \le \frac{d|S|}{2}\right].$$

We can rewrite this as the probability that there is a small T that contains $N(S)$, and then bound the latter:

$$\begin{aligned}
&\mathbb{P}\left[\exists S \subseteq [n], 0 < |S| \le k, \text{ and } \exists T \subseteq [s], |T| \le \frac{d|S|}{2}, \text{ s.t. } N(S) \subseteq T\right]\\
\le&\sum_{i=1}^{k} \binom{n}{i} \cdot \binom{s}{d\cdot i/2} \cdot \left(\frac{d\cdot i}{s}\right)^{d\cdot i}\\
\le&\sum_{i=1}^{k} \left(\frac{e\cdot n}{i}\right)^{i} \cdot \left(\frac{e\cdot s}{d\cdot i/2}\right)^{d\cdot i/2} \cdot \left(\frac{d\cdot i}{s}\right)^{d\cdot i}\\
=&\sum_{i=1}^{k} \left(\frac{e\cdot n}{i}\right)^{i} \cdot \left(\frac{e\cdot d\cdot i/2}{s}\right)^{d\cdot i/2}\\
=&\sum_{i=1}^{k} \left[\underbrace{\frac{e\cdot n}{i} \cdot \left(\frac{e\cdot d\cdot i/2}{s}\right)^{d/2}}_{A}\right]^{i}.
\end{aligned}$$

It suffices to have $A \le 1/2$, so that the probability is strictly less than 1, because $\sum_{i=1}^{k} 1/2^i = 1 - 2^{-k}$. The result follows. **QED**

11.1.3 Low-degree polynomials

Another natural model is that of low-degree polynomials. Chapter 6 and section §8.5 give several applications, and we encounter more below in section 11.1.4.

Theorem 11.5. There are explicit generators that ϵ-fool degree-1 polynomials over $\mathbb{F}_2$ with seed length $s = c\log(n/\epsilon)$.

Such distributions care called *ϵ-bias*, or *small-bias*.

Exercise 11.3. Prove Theorem 11.5 using the construction in the proof of Lemma 6.6.

To fool polynomials of degree $d > 1$, we can take the xor of d independent copies of generators for degree 1. This is known to work for $d < \log n$, and is unknown beyond that.

Theorem 11.6. The sum of d generators that ϵ-fool degree-1 polynomials over $\mathbb{F}_2$, on independent seeds, fools degree-d polynomials with error $\le c\epsilon^{1/2^{d-1}}$.

Question 11.1. *Does this work for $d > \log n$?*

11.1.4 Expander graphs and combinatorial rectangles: Fooling AND of sets

In this subsection we construct a PRG to fool the And of (the indicator functions) of sets. We use "set" and "function" interchangeably in this section. This basic construction ties together many things we have seen and showcases techniques which allow to build even more powerful PRGs. Also, it suffices for the time-efficient simulation of BPP in PH, Item (2) in Theorem 6.3.

Theorem 11.7. There is a PRG G that ϵ-fools the product of t subsets of $[2]^m$ with seed length $m + c(\log t)\log(mt/\epsilon)$. In other words, for any functions $f_i : [2]^m \to [2]$ we have $|\mathbb{E}[\prod_i f_i(U_i)] - \mathbb{E}[\prod_i f_i(X_i)]| \le \epsilon$ where $(X_1, X_2, \ldots, X_t) = G(U)$.

Except for the extra $\log t$ factor, the seed length is good.

Exercise 11.4. Prove Item (2) in Theorem 6.3 assuming Theorem 11.7.

The fundamental case of $t = 2$ is known as *expander graphs.*

Exercise 11.5. [Where is the graph, and why is it expanding?] Let L and R be two disjoint sets with $M := 2^m$ nodes each, and define the graph on vertices $L \cup R$ and edges (x, y) from L to R for any output $(x, y) = G(z)$. For simplicity, further assume that x is uniform for uniform z (a condition satisfied by the construction below). Prove that any set $X \subseteq L$ of αM nodes has $\ge (1 - \epsilon/\alpha)M$ neighbors in R.

For expander graphs, explicit constructions with seed length $m + c\log 1/\epsilon$ are known. We give below a simpler construction with seed length $m + c\log(m/\epsilon)$. Expander graphs have many applications. A simple example is that the general case $t > 2$ is obtained from the $t = 2$ case via recursion.

Recursion

To fool $2t$ sets, first run the generator for 2 sets with error ϵ/c to get two seeds for generators for t sets with error ϵ/c. Then, run twice the generator for t sets on those seeds. Specifically, given $2t$ functions f_i, let $g_1 : [2]^{mt} \to [2]$ be the product of the first t, and g_2 the product of the last t. Let G_t be a generator for the product of t functions. We have:

$$|\mathbb{E}[g_1(U) \cdot g_2(U)] - \mathbb{E}[g_1(G(S_1)) \cdot g_2(G(S_2))]| \le \epsilon/2.$$

To see this, define the "hybrid" distribution $H = g_1(G(S_1)) \cdot g_2(U)$, and note that the distance of $\mathbb{E}[H]$ from each of the expectations inside the absolute value is $\le \epsilon/c$, and use the triangle inequality.

Now the key idea is that we can think of g_1 composed with G as another function h_1, and similarly for g_2. We can fool $h_1 \cdot h_2$ with the generator for two sets with error $\epsilon/2$, obtaining a generator for t sets with error $\epsilon/2 + \epsilon/2 = \epsilon$, as desired.

To analyze the seed length, denote it by $s(m, t, \epsilon)$ for parameters m, t, and ϵ. The definition above gives the recursion

$$s(m, 2t, \epsilon) \le s(s(m, t, \epsilon/c), 2, \epsilon/c) \le s(m, t, \epsilon/c) + c\log s(m, t, \epsilon/c)/\epsilon \le s(m, t, \epsilon/c) + c\log(mt/\epsilon).$$

The second inequality is by the base $t = 2$ case, and the next is because seed mt always suffices, trivially. Iterating $\log_2 t$ times, we obtain seed length

$$\le s(m, 2, \epsilon/t^c) + c\log(t)\log(mt/\epsilon)$$

which is as desired, using again the base case.

Expander graphs

The generator for the base case outputs $(U, U+D)$ where U is uniform and D is a distribution that ϵ-fools linear polynomials over $\mathbb{F}_2$ (!). By Theorem 11.5 the seed length is as desired. To analyze, it is natural to write the functions f_1 and f_2 in terms of polynomials. For slight convenience we think of the inputs in $\{-1,1\}$ instead of $\{0,1\}$, so that multiplication of input bits corresponds to xoring. In particular we will write $U \cdot D$ for $U + D$.

Exercise 11.6 (Hypercube analysis)**.** For $\alpha \subseteq [n]$, we write x^α for $\prod_{i\in\alpha} x_i$, with $x^\emptyset := 1$. Let $f : \{-1,1\}^n \to \mathbb{R}$ be a function.

(1) Show that f can be written as

$$f(x) = \sum_\alpha \hat{f}_\alpha \cdot x^\alpha,$$

for some $\hat{f}_\alpha \in \mathbb{R}$. Guideline: First write $f(x) = \sum_{a\in\{-1,1\}} f(a)I_a(x)$, where $I_a(x) = 1$ if $x = a$ and 0 otherwise.

(2) Show that $\hat{f}_\alpha = \mathbb{E}_x[f(x)x^\alpha]$. In particular and for example, $\hat{f}_\emptyset = \mathbb{E}[f(U)]$.

(3) Show that $\sum_\alpha \hat{f}_\alpha^2 = \mathbb{E}[f^2(U)]$.

Writing $f = f_1$ and $g = f_2$ we need to bound $|\mathbb{E}[f(U)g(U \cdot D)] - \mathbb{E}[f(U)]\mathbb{E}[g(U)]|$. Note that in the first $\mathbb{E}$ the two occurrences of U denote the same sample, whereas in the second they denote independent samples. Using Exercise 11.6, the second summand is $\hat{f}_\emptyset \hat{g}_\emptyset$. The first is

$$\mathbb{E}_{x\leftarrow U,D}[\sum_{\alpha,\beta} \hat{f}_\alpha \hat{g}_\beta x^\alpha (x\cdot D)^\beta].$$

Because $(x \cdot D)^\beta = x^\beta \cdot D^\beta$, the terms with $\alpha \neq \beta$ give 0. So we can rewrite it as

$$\mathbb{E}_D\left[\sum_\alpha \hat{f}_\alpha \hat{g}_\alpha D^\alpha\right].$$

Putting this together, we remove the $\alpha = \emptyset$ term, and our goal is to bound

$$\left|\mathbb{E}_D\left[\sum_{\alpha\neq\emptyset} \hat{f}_\alpha \hat{g}_\alpha D^\alpha\right]\right|.$$

This is at most

$$\begin{aligned}&\sum_{\alpha\neq\emptyset}|\hat{f}_\alpha|\cdot|\hat{g}_\alpha|\cdot|\mathbb{E}_D[D^\alpha]| \text{ (by the triangle inequality)}\\ \leq&\epsilon\sum_{\alpha}|\hat{f}_\alpha|\cdot|\hat{g}_\alpha| \text{ (by the assumption on } D\text{)}\\ \leq&\epsilon\sqrt{\sum_\alpha \hat{f}^2_\alpha}\cdot\sqrt{\sum_\alpha \hat{g}^2_\alpha} \text{ (by Fact A.8)}\\ =&\epsilon\sqrt{\mathbb{E}[f^2(U)]}\cdot\sqrt{\mathbb{E}[g^2(U)]} \text{ (by Exercise 11.6, Item (3))}\\ \leq&\epsilon \text{ (because the range of f and g is [2]).}\end{aligned}$$

11.2 PRGs from hard functions

In this section we present a general paradigm to construct PRGs from hard functions. 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 3.2.

Claim 11.2. Let $f : [2]^n \to [2]$ be a function. We have:

(1) If $C : [2]^{n+1} \to [2]$ ϵ-breaks $G(x) := xf(x)$ then there is $b \in [2]$ s.t. $C'_b : [2]^n \to [2]$ defined as $C'_b(x) := C(xb) \oplus b$ has correlation $\mathbb{E}e[C'_b(x) \oplus f(x)] \geq \epsilon$ with f.

(2) Conversely, suppose $C : [2]^n \to [2]$ has ϵ-correlation with f. Then $C' : [2]^{n+1} \to [2]$ defined as $C'(x, b) := C(x) \oplus b$ $\epsilon/2$-breaks $xf(x)$.

Proof of (1). Pick b uniformly and write

$$\mathbb{E}_{x,b}e[C'_b(x)\oplus f(x)] = \mathbb{E}_{x,b\oplus f(x)}e[C'_{b\oplus f(x)}(x)\oplus f(x)] = \mathbb{E}e[C_{b\oplus f(x)}(x(b\oplus f(x)))\oplus b] = \frac{1}{2}|\mathbb{E}_xC(xf(x))-\mathbb{E}_xC(x\overline{f}(x$$

So there is b s.t. the LHS is at least the RHS. This establishes the first claim. **QED**

Exercise 11.7. Prove (2) in Claim 11.2.

The contrapositive of (1) is that functions with small correlation immediately imply a 1-bit of stretch generator. Naturally, we'd like to increase the stretch. A natural idea is *repetition:* From a pseudorandom distribution D over $[2]^n$, we construct $D^k := D, D, \ldots, D$ over $[2]^{k\cdot n}$.

Claim 11.3. If f ϵ-distinguishes D^k and E^k then a restriction of f ϵ/k-distinguishes D and E.

Proof. Via the "hybrid method," a.k.a. the triangle inequality, cf. proof of Theorem 11.7. Define $H_i := D_0D_1 \cdots D_{i-1}E_iE_{i+1} \cdots E_{k-1}$ 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

$$\begin{aligned}\epsilon &\leq |\mathbb{E}[f(H_0)] - \mathbb{E}[f(H_{k-1})]| \\ &= |\sum_{i\in[k]} \mathbb{E}[f(H_i)] - \mathbb{E}[f(H_{i+1})]| \\ &\leq \sum_{i\in[k]} |\mathbb{E}[f(H_i)] - \mathbb{E}[f(H_{i+1})]|.\end{aligned}$$

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 to ϵ/k. This means the claim is only applicable when ϵ is fairly small. In general, this loss cannot be avoided:

Exercise 11.8. Exhibit a distribution D that is 0.1-pseudorandom (for say PCkt) but D^k is not even 0.9 pseudorandom, for suitable k. Now strengthen this to D of the form $xf(x)$, for some boolean function f.

However, repetition works for *resamplable* functions, like parity. 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 $(x, h(x) \oplus 1)$ – using the same distribution.

Definition 11.4. 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 11.4. Suppose $h : [2]^n \to [2]$ is balanced (i.e., $\mathbb{P}[h(U) = 1] = 1/2$) and resampled by F. Let $D = (X, h(X))$. 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. Because h is balanced, we can sample U_{n+1} by first tossing a coin b, and then outputting $(X, h(X) \oplus b)$. 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. So we can fix those 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. As in the proof of Claim 11.2 it suffices to tell D from $\overline{D} := (X, \overline{h}(X))$. On input $z \in [2]^{n+1}$, we compute

$$f'(F(z), F(z), \ldots, F(z)).$$

If $z = D$ then this is the same as (I), and if $z = \overline{D}$ this is the same as (II). **QED**

Claim 11.5. Parity is resamplable in AC.

Exercise 11.9. Prove this.

Combining the results in this section with the correlation of ACs and parity – Corollary 8.4 – we obtain a PRG with seed length $n - n/\log^{c_d} n$ that fools ACs of size n^d and depth d on n bits. In the next section, leveraging the exponentially-small correlation bounds between ACs and parity, we will obtain a much shorter, logarithmic seed length for ACs.

However, for other classes of circuits like AC[2] such strong correlation bounds are not known. For these classes, the results in this section give the best-known explicit generator. For example, for AC[3] we can again use that parity has correlation $\leq 1/100$ with such circuits, as follows from Exercise **??**, and obtain a generator stretching $n - n/\log^{c_d} n$ bits to n. For AC[2] one can work with a different function and again obtain that stretch. Better stretch is not known.

Question 11.2. *Give an explicit generator with seed length* $0.9n$ *for AC*[2] *circuits of size* n^c *and depth* c *on* n *bits.*

11.2.1 From correlation bounds to stretch: Sets with bounded intersections

The repetition PRG outputs values of a hard function h on independent inputs. We now study a powerful technique which instead outputs values from *dependent* inputs. This gives a better trade-off between seed and output length. It is a derandomized analogue of Claim 11.3. Rather than picking independent inputs as in the repetition generator, we select them based on a collection of subsets of $[u]$, where u is the seed length.

Definition 11.5. Let $S = T_1, T_2, \ldots, T_S$ be collection of subsets of $[u]$ of size ℓ. Then the *bounded-intersection generator*

$$\mathrm{BIG}_S : [2]^u \to \left([2]^\ell\right)^S$$

is defined as $\mathrm{BIG}_S(x) := x_{T_1}, x_{T_2}, \ldots, x_{T_S}$. Here x_{T_j} are the ℓ bits of x indexed by T_j.

For a distribution H on functions from $[2]^\ell \to [2]$ and a generator $G : [2]^u \to \left([2]^\ell\right)^S$ we write $H \circ G(\sigma)$ for the result $H(x_1), H(x_2), \ldots, H(x_S)$ of applying H to the outputs of G, where $G(\sigma) = (x_1, x_2, \ldots, x_S)$ and the occurrences of H denote independent samples.

For example, if H is a uniform function then $H \circ \mathrm{BIG}_S$ is uniform over $[2]^S$. The next key result shows that BIG *preserves indistinguishability*, similar to the repetition generator, as long as the sets in S have small intersections. The intersection size governs the locality (recall Definition 1.5), and hence the complexity, of the reduction.

Theorem 11.8 (BIG PI)**.** Let BIG and S be as in Definition 11.5. Furthermore, suppose $|T_i \cap T_j| \le w$ for any $i \ne j$. Let V and W be two distributions on functions from $[2]^\ell$ to $[2]$.

Suppose f ϵ-tells the distributions $(\sigma, V \circ \mathrm{BIG}_S(\sigma))$ and $(\sigma, W \circ \mathrm{BIG}_S(\sigma))$, over $u + |S|$ bits.

Then there are w-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 $\ge \epsilon/|S|$, where X is uniform in $[2]^\ell$.

Exercise 11.10. Derive Claim 11.3 from Theorem 11.8 for the special case $D = (X, V(X))$ and $E = (X, W(X))$.

Proof. Write $D = (\sigma, V \circ G_S(\sigma)) = D_0 D_1 \cdots D_{|\sigma|+S-1}$ and $E = (\sigma, W \circ G_S(\sigma)) = E_0 E_1 \cdots$. As in the proof of Claim 11.3, define hybrids

$$H_i := D_0 D_1 \cdots D_{i-1} E_i E_{i+1} \cdots E_{|\sigma|+S-1}$$

over $|\sigma| + S$ bits. Note that H_0 is E and $H_{|S|}$ is D. So there is $i \in [S]$ s.t. f distinguishes two adjacent hybrids $H_{|\sigma|+i}$ and $H_{|\sigma|+i+1}$ with advantage $\ge \epsilon/S$. (The first $|\sigma|$ bits in D and E are equal.)

We can fix the $u - \ell$ bits in the seed σ that are not in set T_i. Now every bit in position $j \ne i$ depends on $\le w$ bits in T_i, and so can be computed by a distribution G_j on w-local functions.

The following distribution on circuits tells $(X, V(X))$ from $(X, W(X))$, again with advantage $\ge \epsilon/|S|$: On input (x, b) run f on

$$(G_0(x), G_1(x), \ldots, G_{i-1}(x), b, G_{i+1}(x), G_{i+2}(x), \ldots, G_{|\sigma|+|S|-1}(x)).$$

We can fix the G_i to g_i and maintain the advantage. **QED**

To apply Theorem 11.8 we need a collection S with small intersections. We'd like to have as many sets 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) which are subsets of as small a set 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 collections with great parameters exist. The following is a simple explicit construction.

Lemma 11.1. [Sets with small intersections] There are explicit collections of q^d subsets of $[q^2]$ of size q such that any two sets intersect in $\le d$ elements, for any q that is a power of 2 and $> d$.

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 $\{(x, p(x)) : x \in \mathbb{F}\}$.) 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 Lemma 2.1, the intersection has size $\le d$. **QED**

To illustrate parameters, we can have $m = q^d$ subsets of size $\ell = \sqrt{q}$ from a universe of size $u = \ell^2 = q$ with intersections at most d. For example, given m we can set $d = \log m$ and $q = \log^a m$, and the intersection size is only $\ell^{1/a}$ while the universe is only quadratic in the set size, i.e., $u = \ell^2$.

Corollary 11.1. There are generators $G : [2]^{\log^{c_d} n} \to [2]^n$ that $1/n$-fool ACs of size n and depth d.

As in Exercise 11.1, in this corollary, to eliminate one parameter we set the size of the circuit equal to the output length of G. The same statement holds if the size is n^d instead of n.

Exercise 11.11. Prove Corollary 11.1. Explain how the parameters are set and which results you are combining.

The seed length in Corollary 11.1 is about the best we can do given current impossibility results, and recall once again from section §7.3 that stronger impossibility would imply major separations.

Still, one can ask if PRGs *could* be built *if* we had such stronger results. 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, cf. Exercise 11.1. Lemma 11.1 doesn't give this, since the universe is always at least quadratic in the set size, but the following construction does.

Lemma 11.2. [Sets with small intersections, II] For any a and $n \geq c_a$ there is an explicit collection of n subsets of $[c_a \log n]$ of size $c_a \log n$ with pairwise intersection $\leq a \log n$.

Using this we can prove the following weaker version of Theorem 2.9.

Corollary 11.2. Suppose there is $\epsilon > 0$ and $f \in$ E that on inputs of length n has correlation at most $2^{-\epsilon n}$ with circuits of size $2^{\epsilon n}$. Then P = BPP.

Exercise 11.12. Prove Corollary 11.2 assuming Lemma 11.2.

11.2.2 Turning hardness into correlation bounds

> 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).

We can't expect to prove that correlation bounds under uniform are equivalent to impossibility or hardness results, as one can construct pathological functions which are easy to compute on, say, .75 fraction of the inputs, but impossible to compute on a .76 fraction. So instead our approach will be to *construct* functions which have small correlation under the uniform distribution.

A natural candidate for such a function, starting from a "mildly hard" function $f : [2]^n \to [2]$ is $f' : [2]^{nk} \to [2]$ defined as

$$f'(x_1, \ldots, x_k) := \oplus_{i=1}^{k} f(x_i).$$

An *XOR Lemma* is a statement showing that if f has correlation $\le \epsilon$ with a certain computational model (e.g., PCkt), then the correlation of f' with a related model decays *exponentially fast* with the number k of copies (cf. Definition 3.2 of correlation).

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 this model then simply means that for every constant function $g(x) = 0$ or $g(x) = 1$ we have

$$|\mathbb{E}ef(x) + g(x)| = |\mathbb{E}ef(x)| \le \epsilon.$$

And indeed in this case the correlation decays exponentially fast:

$$|\mathbb{E}ef'(x') + g(x')| = |\mathbb{E}ef'(x')| = |\mathbb{E}ef(x)|^k \le \epsilon^k.$$

More generally, consider that if f has correlation $\le \epsilon$ with small circuits C, then f' indeed has correlation $\le \epsilon^k$ 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

$$|\mathbb{E}ef'(x') + C(x_1, \ldots, x_k)| = \left|\mathbb{E}e \oplus_{i=1}^{k} (f(x_i) \oplus C(x_i))\right| = |\mathbb{E}ef(x) \oplus C(x)|^k \le \epsilon^k.$$

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?

Exercise 11.13. Consider circuits C made of a single majority gate. Prove that the XOR lemma is false for C. Feel free to pick n even and define the value of Majority on inputs of weight $n/2$ to be 1, and recall $\binom{n}{n/2} \cdot \frac{\sqrt{n}}{2^n} \in [c, c]$.

One can extend this result to AC with a small number of majority gates.

Question 11.3. *Does the XOR lemma hold for AC with parity gates, or even constant-degree polynomials over* $\mathbb{F}_2$*?*

But for more powerful models, we can indeed prove the xor lemma, and the proof follows the information-theoretic intuition above. To connect to this intuition, we consider functions which may output a uniform bit on some inputs.

Definition 11.6. 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 $|H| = 2\delta 2^n$ 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 2δ on which it is information-theoretically unpredictable. To illustrate the XOR lemma, suppose that f is δ-random. Then f' will be almost a coin flip. Specifically, the probability that the output is not a coin flip is $(1-2\delta)^k$, the probability that no input falls into H. When some input falls into H, the output is a coin flip, and no circuit, efficient or not, can have non-zero correlation.

This intuition can be formalized via the *hardcore-set* lemma, which allows us to pass from computational hardness to information-theoretic hardness. Before stating the lemma we emphasize an important point:

The hardcore-set lemma is only known to hold for computational models which can compute majority. This is because the proof of correctness uses majority, as will be apparent in section §11.3. So to apply it, we have to start from an impossibility result for circuits that can compute majority. As discussed in Chapter 8, we essentially have no such result. In fact, in some restricted models, the xor lemma is false (cf. Exercise 11.13). So the results in this section are mostly conditional. Still, they allow us to spin a fascinating web of reductions between correlation and randomness, pointing to several challenges.

The following hardcore set lemma says that any function that has somewhat small correlation with small circuits admits a somewhat large hardcore set on which the function has very small correlation with small circuits. To illustrate parameters, note that a δ-random function has correlation $\le 1-2\delta$ with any fixed function (or circuit) (we can extend Definition 3.2 to random functions by taking expectation over both the input and the random function). This is because when the input falls in the set H of density 2δ from Definition 11.6 then the correlation is zero. The hard-core set lemma shows that this is the only way that small correlation may arise: any function with small correlation with small circuits is in fact close to a δ-random function. We state the result in terms of distinguishing input-output pairs, as opposed to computing the function. This is equivalent by an argument similar to Claim 11.2 but is more convenient as it immediately allows us to talk about multiple inputs, as we also do in the next statement. Here we use $\cdot$ to denote concatenation.

Lemma 11.3 (Hardcore Set)**.** Let $f:[2]^n \to [2]$ have correlation $\le 1-2\delta$ with circuits of size s. Then there exists a $c\delta$-random function $g:[2]^n \to [2]$ such that $X\cdot f(X)$ and $X\cdot g(X)$ are ϵ-indistinguishable by circuits of size $cs\epsilon^2\delta^2$, for any ϵ, where $X \equiv U_n$.

In particular, by Claim 11.3,

$$X_1 \cdots X_k \cdot f(X_1)\cdots f(X_k) \text{ and } X_1\cdots X_k \cdot g(X_1)\cdots g(X_k)$$

are $k\epsilon$-indistinguishable for size $cs\epsilon^2\delta^2$, where the X_i's are uniform and independent.

We can now easily formalize the proof of the xor lemma.

Lemma 11.4. Suppose $f:[2]^n \to [2]$ has correlation $\le (1-2\delta)$ 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 $\le (1-c\delta)^k + k/s^c$ with circuits of size $\delta^c s^c$.

For example, if $s = 2^{n^c}$ and $\delta = c$, 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 Lemma 11.3 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 11.2. **QED**

11.2.3 Derandomizing the XOR lemma

A drawback of the xor lemma is that the input length of the new function is $\geq kn$. This prevents us from obtaining correlation 2^{-cn} (as opposed to $2^{-c\sqrt{n}}$) which is important for the flagship conclusion P = BPP, cf. Corollary 11.2. To remedy this we shall use... PRGs! Rather than independently, we will pick the k inputs to f' using a generator. We need two properties from this PRG. First, to behave like repetition, we need BIG (Theorem 11.8). Also, we need to "hit" the hard-core set, for which we need HIT. We can get both properties by xor-ing the generators together. The generator is defined as

$$\text{BIG-HIT}(\sigma_1, \sigma_2) := \text{BIG}_S(\sigma_1) \oplus \text{HIT}(\sigma_2),$$

where HIT is a hitter, given next.

Lemma 11.5. For every ϵ and δ there exists an explicit generator HIT : $[2]^{2n} \to ([2]^n)^s$ with $s = 1/\epsilon\delta$ s.t. for every set $H \subseteq [2]^n$ of size ϵ, $\mathbb{P}_\sigma[\text{HIT}(\sigma)_i \notin H$ for every $i] \leq \delta$.

Proof. Pairwise independence. Consider the field $\mathbb{F}_{2^n}$. The seed σ specifies $a, b \in \mathbb{F}$ and we output $b, a+b, 2a+b, \ldots$. Let X_i be the indicator variable of $\text{HIT}(\sigma)_i \in H$. The X_i are pairwise independent. Their expectation is ϵs. Hence the probability to bound is $\leq \mathbb{P}[|\sum X_i - \epsilon s| \geq \epsilon s]$. Squaring both sides of the inequalities and doing calculations gives the result. **QED**

Exercise 11.14. Do the calculations.

Using this, we can boost correlation $(1-2^{-cn})$ to correlation $\leq 2^{-cn}$. We give an example for an interesting setting of parameters.

Lemma 11.6. Suppose E has a function $f : [2]^* \to [2]$ that on inputs of length n has correlation $(1-2^{-cn})$ with circuits of size 2^{cn}. Then E has a function $f' : [2]^* \to [2]$ that has correlation $\leq 2^{-cn}$ with circuits of size 2^{cn}.

Note the conclusion implies P = BPP by Corollary 11.2.

Proof. Let $\epsilon := 2^{-cn}$ and $\delta := 2^{-cn}$. Define $f' : [2]^{cn} \to [2]$ as $f'(\sigma) := \oplus_{i=1}^{s} f(x_i)$ where BIG-HIT$(\sigma) = (x_1, x_2, \ldots, x_s)$, where $s = 1/\delta\epsilon$ and the set system for BIG is from Lemma 11.2.

We use Lemma 11.3 with $\epsilon := s^c$. Let g the corresponding δ-random function. From Theorem 11.8 it follows that

$$\sigma, f \circ G \text{ and } \sigma, g \circ G$$

are ϵ^c-indistinguishable for size $1/\epsilon^c$. In particular this holds if we take parities, so

$$\sigma, \oplus_{i=1}^{k} f(X_1) \text{ and } \sigma, \oplus_{i=1}^{k} g(X_1)$$

are no more distinguishable, where $(X_1, \ldots, X_k) = G(\sigma)$. By the hitting property of HIT, Lemma 11.5, the chance of not hitting the hardcore set is $\leq \delta$, and we conclude as in the proof of Lemma 11.4. **QED**

Exercise 11.15. Gotcha!? We can't quite use Theorem 11.8 and Lemma 11.5 as stated. Explain why and how to modify the statements and proofs of Theorem 11.8 and Lemma 11.5 so that the above proof of Lemma 11.6 does work.

11.2.4 Encoding the whole truth-table

The results in the previous section give us functions with small correlation starting from functions on n bits with with correlation $(1 - 2^{-cn})$, but not quite from any impossibility results, which only gives correlation $\leq (1 - 2/2^{-n}) < 1$.

Exercise 11.16. Explain where the previous proofs break down.

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 11.9. Suppose there is $f \in$ E that on inputs of length n cannot be computed by circuits of size $s(n)$. Then there is $f' \in$ E that has correlation $(1 - 1/n^c)$ with circuits of size $n^c s(cn)$.

Recall in Theorem 3.9 we saw an equivalence between computing and correlating under every distribution. Had we had that result for the *uniform* distribution we could have skipped all the amplification results, including Theorem 11.9 and constructed PRGs much more directly. However in general we can't guarantee that. In fact, one can construct functions that are very easy over the uniform distribution, say because they are almost always one, but still are hard to compute, say because there is a small set of inputs that makes the function hard.

Proof. Thinks of an n-bit input to f as ℓ variables 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 p_f is done as in section 10.5.2, using Exercise 10.8. That is:

$$f(x) = \sum_{a \in H^\ell} f(a) \cdot \mathrm{EQ}(x, a).$$

Now the gain is that we can think of evaluating p_f over a larger fields. Set $q := n^{10}$ and $d := n^5$ and $\ell := n/\log d$. The new function f' is constructed in two steps. First, we consider inputs over $\mathbb{F}_q^\ell$. Note the length of such inputs is $\le c\ell \log q \le cn$ 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 $\le 1/n^c \le c/d\ell$. Because if it's larger, every such x contributes at least one input (x, i) where C disagrees with f', contradicting the assumption.

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 $C(y + 1s), C(y + 2s), \ldots, C(y + d\ell s)$.

Because each evaluation point is uniform, and $d\ell\alpha \le c$, with prob. $> 1/2$ the evaluations of C will be correct, and equal $p_f(y + 1s), p_f(y + 2s), \ldots, p_f(y + d\ell s)$.

Note that for fixed y and s, $p_f(y + ts)$ is a univariate polynomial q in t of degree $\le d\ell$. We can compute the coefficients of q from its evaluations at $d\ell$ points. (It's a linear system, with a unique solution by Lemma 2.1 because the degree of q is $\le \ell \cdot d < q$.)

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. **QED**

Exercise 11.17. "Put it all together" and prove Theorem 2.9.

11.2.5 Monotone amplification within NP

To increase the hardness of functions in NP we cannot use XOR since NP is not known to be closed under complement. We will use a combination of many things in this chapter – including the (unconditional) generator for AC in Corollary 11.1 – to establish the following.

Theorem 11.10. If NP has a balanced function that has correlation $\le 1/10$ with circuits of size 2^{n^c}, then NP also has a balanced function with correlation $\le 2^{-n^c}$ with circuits of size 2^{n^c}.

Several optimizations have been devised, see the Notes. Still, we don't enjoy the same range as for E:

Question 11.4. *Prove Lemma 11.6 for NP instead of E, even starting from hardness $\delta \geq c$.*

11.2.6 Proof of Theorem 11.10

Rather than XOR, to amplify we use the Tribes function, a monotone read-once DNF.

Definition 11.7. The Tribes function on k bits is:

$$\text{Tribes}(x_1, \ldots, x_k) := (x_1 \wedge \ldots \wedge x_b) \vee (x_{b+1} \wedge \ldots \wedge x_{2b}) \vee \ldots \vee (x_{k-b+1} \wedge \ldots \wedge x_k)$$

where there are k/b clauses each of size b, and b is the largest integer such that $(1-2^{-b})^{k/b} \geq 1/2$. Note that this makes $b \leq c \log k$.

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, that if several bits are uniform, then the output is close to uniform.

Lemma 11.7. Let N_p be a noise vector where each is 1 independently with probability p. Then $\mathbb{E}_{x,N_p} e[\text{Tribes}(x) \oplus \text{Tribes}(x \oplus N_p)] \leq 1/k^{c_p}$.

We shall take k exponentially large. The resulting function is still in NP as we can use non-determinism to pick a clause. We use the generator BIG-AC$(\sigma) = (x_1, x_2, \ldots, x_s)$, which is like BIG-HIT except that HIT is replaced with the generator in Corollary 11.1, for circuits of size $(k2^n)^c$. Note its seed length is n^c for $k \leq 2^{n^c}$.

Define $f' : [2]^{2n} \to [2]$ as $f'(\sigma) := \text{Tribes} \circ (f(x_1), \ldots, f(x_s))$ where BIG-AC$(\sigma) = (x_1, x_2, \ldots, x_s)$, and the set system for BIG is from Theorem 3.1. Following the proof of Lemma 11.6, use Lemma 11.3 with $\epsilon := s^c$. Let g the corresponding δ-random function. From Theorem 11.8 it follows that

$$\sigma, f \circ \text{BIG-AC and } \sigma, g \circ \text{BIG-AC}$$

are ϵ^c-indistinguishable for size $1/\epsilon^c$. In particular this holds if we take Tribes of the output, i.e.What remains to show is that

$$\sigma, \text{Tribes}\circ g \circ \text{BIG-AC}$$

is close to uniform. That is, we have to show that with high probability over σ, just over the choice of g, the value Tribes$\circ g \circ$ BIG-AC is close to a uniform bit.

It suffices to bound

$$\mathbb{E}_\sigma |\mathbb{E}_g e[\text{Tribes}\circ g \circ \text{BIG-AC}(\sigma)]|.$$

Here the inner expectation is over the random choices in all the s evaluations of g. Up to a power, this is

$$\leq \mathbb{E}_\sigma \mathbb{E}_g^2 e[\text{Tribes}\circ g \circ \text{BIG-AC}(\sigma)] = \mathbb{E}_{\sigma,g,g'} e[\text{Tribes}\circ g \circ \text{BIG-AC}(\sigma) \oplus \text{Tribes}\circ g' \circ \text{BIG-AC}(\sigma)].$$

Now the critical step is that Tribes $\circ\, g$ is computable by a distribution on AC of size $(s2^n)^c$ and depth c. Note that the circuit computes g in brute-force, but the dependence on s is good. Because BIG-AC fools such circuits with error 2^{-n^c}, the latter expectation equals

$$\mathbb{E}_{(x_1,\ldots,x_s),g,g'} e[\text{Tribes}\circ g \circ (x_1,\ldots,x_s) \oplus \text{Tribes}\circ g' \circ (x_1,\ldots,x_s)] = \mathbb{E}_{x,N_p} e[\text{Tribes}(x) \oplus \text{Tribes}(x \oplus N_p)],$$

for $p = c$. We conclude by Lemma 11.7.

11.3 Proof of the hardcore-set Lemma 11.3

At the high-level, this is just min-max and concentration of measure, just like the equivalence between computation and correlation in Theorem 3.9. However, the proof is slightly more involved than one might anticipate. We break it up in two claims. First we obtain hardness w.r.t. a "smooth" distribution D: $D(x) \le d/N$ for every x. For example, D could be "flat," i.e. uniform over a set of size N/d (then $D(x)$ is either 0 or d/N). 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 yields the lemma.

Claim 11.6. Suppose $f : [2]^n \to [2]$ is $1/d$-hard for circuits of size s (cf. Definition 3.2). Then there is a distribution D on $[2]^n$ s.t. $D(x) \le d/N$ for every x, and every circuit C of size $s' := s \cdot (\epsilon/\log d)^c$ has $\mathbb{E}_{x \leftarrow D} e[C(x) + f(x)] \le \epsilon$.

Proof. We use the duality Theorem 3.10 where one set consists of sets S of N/d inputs, and the other consists of circuits C of size $\le s'$, and $p(S, C) := \mathbb{E}_{x \in S} e[C(x) + f(x)]$. (In the proof of Theorem 3.9 p was just a "single point," here it's an average.)

Suppose there is a distribution over sets S of size N/d such that for every circuit we have $\mathbb{E}_S[p(S, C)] = \mathbb{E}_S \mathbb{E}_{x \in S} e[C(x) + f(x)] \le \epsilon$. Let D be the induced distribution over inputs x, and note that $D(y) \le d/N$ for every y, and we're done.

Otherwise by the min-max Theorem 3.10 there is a distribution C on circuits s.t. for any set T of size N/d we have

$$\mathbb{E}_C p(T, C) \ge \epsilon. \tag{11.1}$$

Call an input x *hard* if C does not do well on it: $\mathbb{E}_C e[C(x) + f(x)] \le \epsilon/2$. Now, there can't be N/d hard inputs, for else we contradict equation (11.1) for a set T of N/d hard inputs. In fact, we can't even have $(1 - \epsilon/2)N/d$ hard inputs. Otherwise the set T consisting of the N/d elements where $\mathbb{E}_C e[C(x) + f(x)]$ is smallest would have only $(N/d)\epsilon/2$ easy inputs, yielding $\mathbb{E}_C p(T, C) < \epsilon/2 + \epsilon/2 = \epsilon$, again contradicting equation (11.1).

We conclude by observing that for every easy x, picking $\log(d)/\epsilon^c$ samples from C and taking majority gives error prob. $\le \epsilon/2d$, using Theorem 2.1 as in the proof of Theorem 2.2. 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: $(1 - \epsilon/2)/d + \epsilon/2d = 1/d$. This contradicts the hardness of f. **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 11.7. Let D be a distribution over $[N]$ s.t. $D(x) \leq d/N$. Let F be a set of $\leq ce^{c\epsilon^2 N/d^2}$ 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 $|S| \geq cN/d$ 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. $D(x)N/d \in [0, 1]$. 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, $\mathbb{E}[X] = (N/d)\mathbb{E}_D[f(x)]$, and so $|\mathbb{E}[X]| \leq c\epsilon N/d$. By tail bounds, Exercise 2.4:

$$\mathbb{P}_S\left[\left|\sum_{x\in[N]} S(x)f(x)\right| \geq c\epsilon N/d\right] \leq 2e^{-c\epsilon^2 N/d^2}.$$

Also, $\mathbb{E}[\sum_{x\in[N]} S(x)] = N/d]$. And so again by tail bounds the probability that $|S| \leq cN/d$ is, say, $\leq e^{-cN/d^2}$.

By a union bound, there exists S of size $\geq cN/d$ s.t. for every $f \in F$ we have

$$\left|\sum_{x\in[N]} S(x)f(x)\right| \leq c\epsilon N/d.$$

Now it's the moment to connect to $|S|$. Dividing both sides by $|S|$ we have

$$|\mathbb{E}_{x\in S}[f(x)]| \leq c\epsilon(N/d)/|S| \leq c\epsilon,$$

as desired. **QED**

11.4 Problems

Problem 11.1. Let ACSize(d, s) be the functions computable by explicit ACs of size s and depth d. Prove that BP$\cdot$ACSize$(d, s) \subseteq$ ACSize$(d+c, 2^{\log^{c_d} s})$. That is, we can de-randomize small-depth circuits in time much smaller than exponential.

Problem 11.2. Let f be the And function on n bits.

(1) Letting $f = \sum_\alpha \hat{f}_\alpha x^\alpha$ as in Exercise 11.6, give an expression for the $\hat{f}_\alpha$.

(2) Use (1) to show that a pseudorandom generator for degree-1 polynomials fools any And function (on any subset of the bits).

Problem 11.3. Give a "direct" construction of PRGs sufficient to prove P = BPP from a δ-hard function h in E. Guideline: Use BIG-HIT to generate an $n \times n$ matrix of inputs, evaluate h on every input, and then XOR the rows. Start with $\delta = c$. How small can you make δ and still have this construction?

11.5 Notes

For more on unconditional pseurandom generators see [125]. For a broader view of pseudorandomness, with an emphasis on connections between various objects, see [262]. For hypercube analysis, see [204].

For expander graphs see [132]. They have many equivalent presentations, for example in terms of eigenvalues. My presentation is in terms of the mixing lemma, see Section 2.4 in [132]. In my definition I allow for repeated edges. Different notions of explicitness are also natural. In my definition one can output an edge given an index. More stringently, one can ask, given a node and an index to an incident edge, to compute the corresponding neighbor. The construction I presented immediately gives the more stringent explicitness as well.

k-wise uniform distributions were studied before complexity theory, cf. [213]. The complexity viewpoint is from [59, 15].

Generators for degree-1 polynomials originate in [191], with alternative constructions in [16]. The idea of xoring generators for degree-1 polynomials to fool higher-degree polynomials is from [46]. It was studied further in [175, 277], with the latter paper proving Theorem 11.6.

Regarding 11.2: A construction computable in time n^c first appeared in [201], where 11.2 also appears. Alternative constructions computable with small space or with alternations appeared respectively in [158] and [272]. Still, all these constructions use resources at least exponential in the seed length, while for several applications such as Lemma 11.6 one needs power in the seed length. This stronger explicitness is obtained in [114] building on an idea presented in [119] of using error-correcting codes. Specifically, one can use the polynomial code from 2.12 in combination with 11.2 for logarithmic-scale collections (for which the former notion of explicitness is now acceptable).

The XOR lemma was reportedly announced in talks associated with the work [293], cf. [102]. Hardness amplification within NP was first studied in [203] which established correlation about $1/\sqrt{n}$. Exponentially small correlation was achieved in [127], with optimizations in [176, 107]. Our exposition follows [127].

Corollary 3.1 and the connection to min-max is from [295]. Hardcore sets were introduced in [138]. They were optimized and shown to be connected to boosting techniques in machine learning in [157] and subsequent works.

Theorem 11.9 is from [24], building on results in the same spirit from [40, 173, 23, 92].

Other proofs of Theorem 2.9 don't use the derandomized xor lemma, or only use it from constant hardness, and instead rely on results in [138] (see e.g. the original proof [142]) or [252] (see e.g. [262] or [20]).

Problem 11.3 is similar to a construction in [252], except I use XOR instead of extractors, cf. Remark 15 in [252].

For more on hitters, see Appendix C in [96].

The quote at the beginning of section 11.2.2 is from my PhD thesis [273].

11.5.1 Historical vignette: Polylogarithmic independence fools AC (Theorem 11.3)

In 1991 [199] constructed a pseudorandom generator for AC (a.k.a. alternating circuits or AC0 circuits), vastly improving the parameters of the pioneering work [10]. This is one of my favorite papers ever. (Mini historical vignette: A large fraction of papers cite [201] for this result, possibly even the majority. This issue of credit is indeed complicated, since the 1988 conference version of [201] claims ownership for this AC result, and cites an unpublished manuscript with the same title as [199], but with both authors. One can only guess that the authors decided that the AC result should only be attributed to Nisan.)

Nisan's distribution, and even the earlier one in [10], is polylog-wise uniform, that is, any polylog bits are uniform. (The polylog depends on the parameters of the circuit to be fooled in a standard way which is ignored here.) In fact, these results apply to a natural class of polylog-wise distributions: If you pick a uniform sparse linear transformation, the output distribution will be polylog-wise uniform, and Nisan's proof shows that it fools AC.

However, the proof does not show that *every* polylog-wise distribution fools AC. Later, Linial and Nisan [172] conjectured that polylog-wise uniformity suffices to fool AC, which would generalize both [10] and [199].

This problem was somewhat notorious but there was no progress until the paper by Bazzi [35], 15+ years after the conjecture was posed, which proves it for the special case of DNFs.

Bazzi's paper is quite hard to read, and the journal version is also long – 60 pages. Consequently it was hard to find referees, both for the conference and the journal version. Things must have gotten somewhat desperate, because when it finally was my turn to be asked to review the journal version it was deemed appropriate to extract a commitment from me before I could see the submission, something that has never occurred to me for any other paper. My back-and-forth with the author during the refereeing process was then abruptly stopped by, I suspect, the circulation of Razborov's follow up [219] which dramatically simplifies the presentation, especially with an idea by Wigderson. It was then clear that the results were correct and the paper could be accepted, even though I never claimed to understand Bazzi's proof for the non-monotone case.

The message in the papers [35] and [219] was loud and clear: You can make progress with just a little duality. From Razborov's paper:

> *By linear duality, this conjecture is an approximation problem of precisely the kind considered in [LMN93, BRS91, ABFR94]. Therefore, it is quite remarkable that the only noticeable progress in this direction was achieved only last year by Bazzi [Baz07].*

At this point it was clear that the general case of AC might not be that hard. Shortly after Razborov's paper, Braverman [48] indeed proved this, albeit with a quadratic rather than linear dependence on the depth of the circuit. This dependence was later improved.

As usual we can look at citation count:
[48] 143
[34] 91
But more interestingly the literature is full of citations like:

A breakthrough result by Braverman [No mention of Bazzi or Razborov]

My definition of breakthrough result is roughly that of progress on a problem such that many people have thought about it but have been stuck for a long time. This applies to [34].

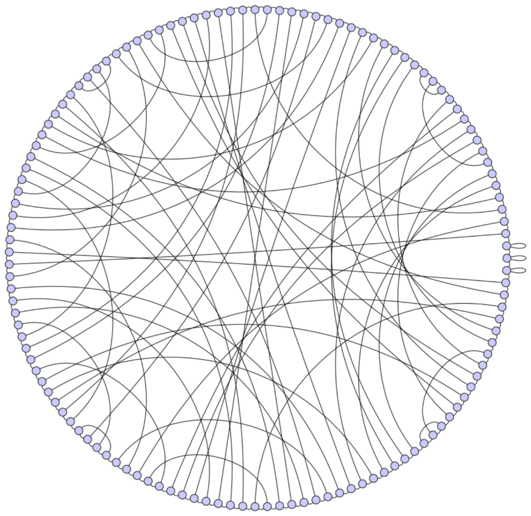
Approximate number of years gap:
[172]-[34]: XXXXXXXXXXXXXXXXXX
[34]-[219]: XX
[219]-[48]: X

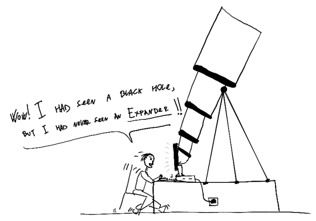
I also think that if a problem was open even for depth 2, then going from 1 to 2 tends to be more fundamental than going from 2 to d. One can think of situations where this wouldn't be the case, for example if the depth-2 case was known for a while, and people were really stuck and couldn't do even depth 3, and that turned out to require a completely different approach. This isn't the case here.

Consider the following example. Tonight a breakthrough lower bound for depth-3 Majority circuits comes out. Then in a year this result is extended to any constant depth with additional but related techniques. Which result, if any, is the breakthrough?

Chapter 12

Eigenvalues, expanders, connectivity in L




Wow! I had seen a black hole,
but I had never seen an Expander !!

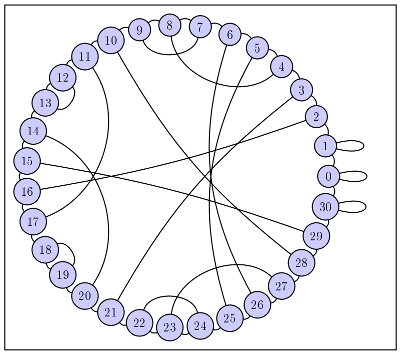


Figure 12.1: A 3-regular expander.

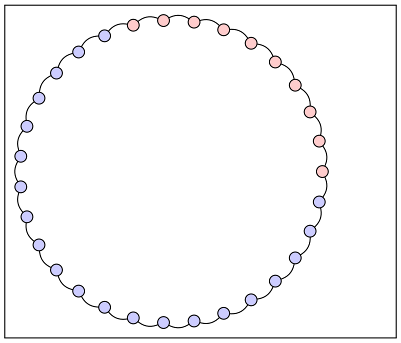


Figure 12.2: The circle graph, on 31 nodes. A set of 10 nodes, shown in red, only has 2 edges leaving it.

12.1 Expanders without eigenvalues

In this section we give a simple and self-contained proof of the existence of constant-degree *expander graphs*. Informally, an expander graph is a graph that is sparse yet "highly connected." Several related notions of expansion exist: vertex, edge, and spectral expansion. In this section we work solely with edge expansion:

Definition 12.1. A d-regular graph $G = (V, E)$ is a *combinatorial edge δ-expander* if for every set $S \subseteq V$ of size $\leq V/2$ its *crossing probability* $\nmid(S)$ that from a uniform $s \in S$ moving to a uniform neighbor takes us out of S is $\geq \delta$. Note

$$\nmid(S) = \frac{E(S, \overline{S})}{dS},$$

where $E(S, T)$ is the set of edges with an endpoint in S and the other in T.

Exercise 12.1. Whereas Definition 12.1 is stated for sets of density $\leq 1/2$, it implies expansion for other sets as well: Let $G = (V, E)$ be a δ-expander and let $S \subseteq V$ of density $\leq (1 - \epsilon)$. Prove $\nmid(S) \geq \delta\epsilon/(1 - \epsilon)$.

Any connected graph $G = (V, E)$ is a δ-expander for noticeable $\delta \geq 1/E$, since at least one edge leaves S. *Expander graphs* are graphs where δ is bounded independent of the size of the graph.

Example 12.1. The circle graph over nodes $\mathbb{Z}_N$ with edges $\{x, x+1\}$, depicted in figure 12.2, is not an expander. For example, $\nmid([N/2]) \to 0$ with $N \to \infty$. On the other hand, the 3-regular expander in figure 12.1, which can be obtained from the circle by adding "chords," can be shown to be an expander. This expander has $\mathbb{F}_p$ as nodes, and the neighbors of x are $x \pm 2$ and $1/x$. For $x = 0$ we replace the undefined $1/x$ with a self-loop, making the graph 3-regular.

Here's another simple expander. TBD the vertex set of a graph G on N nodes is $Z_{\sqrt{N}} \times Z_{\sqrt{N}}$, where $Z_{\sqrt{N}}$ is the ring of the integers modulo $\sqrt{N}$. Each vertex v is a pair $v = (x, y)$ where $x, y \in Z_{\sqrt{N}}$. For matrices T_1, T_2 and vectors b_1, b_2 defined below, each vertex $v \in G_N$ is connected to $T_1 v, T_1 v + b_1, T_2 v, T_2 v + b_2$ and the four inverses of these operations. It can be shown that for the choices $T_1 := \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, $T_2 := \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, $b_1 := \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $b_2 := \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ the resulting graph (which is 8-regular) is an expander.

Pictures are sexy, but we need to compute with expanders. Typically the graph is huge (think Internet) and we have to compute neighbors efficiently given a node name (or index) and an edge name. The expansion properties don't depend on the naming, but the computation does. Several conventions about indexing are possible. *Consistency* is convenient and natural. It asks that an edge name is the same from either endpoint.

Definition 12.2. A d-regular graph $G = (V, E)$ has *neighbor* function $f : V \times [d] \to V$ if $f(u, i)$ for $i \in [d]$ are the d neighbors of u. We say f is *consistent* if whenever $f(u, i) = v$ then also $f(v, i) = u$.

A consistent neighbor function is thus equivalent to an edge coloring of the graph with d colors. While not every d-regular graph admits an edge coloring with d-colors all the graphs in this section have this stronger property.

The main result in this section is that there are expander graphs that are explicit: They have a consistent neighbor function computable in P.

Theorem 12.1. There are c-expanders on 2^n nodes with degree c and a consistent neighbor function in P.

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.

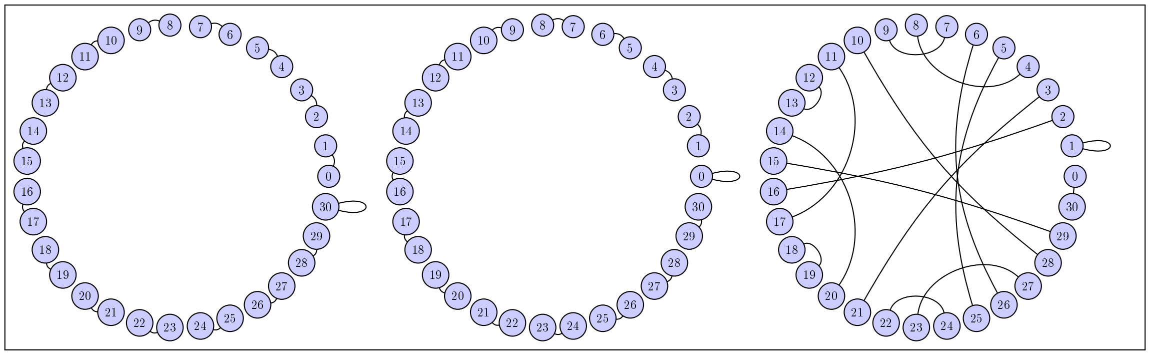


Figure 12.3: The expander in figure 12.1 is the union of these three matchings.

Theorem 12.2. There are c-regular c-expanders on N nodes with a consistent neighbor function computable in time 2^{N^c}.

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. For simplicity, we assume that N is even, and we exclude self-loops from matchings, so a matching has size $N/2$. figure 12.3 illustrates a decomposition in matchings for the expander in figure 12.1; because the number of nodes is odd there, each matching also includes a self-loop. We will soon need to analyze how the distribution of a uniform matching looks locally. 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. We pick the 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 $\leq S/(N - N/4) = S/0.75N =: p \leq 2/3$. These events are not independent, but we can still use the deviation bound Theorem 2.1 via Exercise 2.4. So the prob. that this matching is *bad* in the sense that more than q fraction of these T iterations is matched within S satisfies

$$\mathbb{P}[\text{bad}]^{1/T} \leq \left(\frac{p}{q}\right)^q \left(\frac{1-p}{1-q}\right)^{1-q} \leq \left(\frac{p}{q}\right)^q \left(\frac{1}{1-q}\right)^{1-q} \leq p^{q/2}$$

for $q \in [1-c, 1]$. The latter inequality holds 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 right-had side 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 with lower $\nmid(S)$ is by a union bound and Fact A.3

$$\leq \binom{N}{k} p^{cdT} \leq \left(\frac{eN}{k}\right)^k \left(\frac{k}{0.75N}\right)^{cd\lceil k/2 \rceil} \leq \frac{1}{2^k}$$

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 be checked by again 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**

Exercise 12.2. Repeat this proof with the goal of finding the smallest value d for the degree that works (to obtain expansion dependent on d only).

Next we give explicit expanders with logarithmic degree. We essentially already saw this construction in section 11.1.4.

Theorem 12.3. There are c-expanders on 2^n nodes with degree n^c with a consistent neighbor function computable in P.

Proof. Let Y be an ϵ-biased distribution on $\mathbb{F}_2^n$ (cf Theorem 11.5); the neighbors of $x \in \mathbb{F}_2^n$ are $x + Y$. Theorem 11.5 implies that the graph is explicit. The neighbor function is consistent because $(x+Y)+Y = x$.

To analyze, let S be a set as in Definition 12.1 and let f be the $0-1$ characteristic function of f, and $p := S/N$ the density of S. We have

$$\nmid(S) = p^{-1}\mathbb{E}[f(X)(1 - f(X+Y))] = 1 - p^{-1}\mathbb{E}[f(X)f(X+Y)].$$

We write $f = \sum_\alpha \widehat{f}_\alpha \chi_\alpha$ in the basis of parity functions (cf Exercise 11.6). Note that

$$\widehat{f}_0 = \mathbb{E}[f] = \mathbb{E}[f^2] = \sum_\alpha \widehat{f}_\alpha^2 = p.$$

Hence $\mathbb{E}[f(X)f(X+Y)] = \sum_\alpha \widehat{f}_\alpha^2 \epsilon_\alpha = p^2 + \sum_{\alpha \neq 0} \widehat{f}_\alpha^2 \epsilon_\alpha \leq p^2 + \epsilon(p - p^2)$. Plugging this above we get

$$\nmid(S) \geq 1 - p - \epsilon(1-p) = (1-p)(1-\epsilon).$$

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 12.3. [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 (x, i). 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 repeated 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 3 different neighbor functions.)

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 12.4. 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* (in 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 $X' := \cup_{i \in I'} X_i$ and $X'' := \cup_{i \in I''} X_i$.

If $X'/X \geq \delta_1/10$ 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 12.1, where the 0.5 is for taking a step withing 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 $X''/X \geq (1 - \delta_1/10)$. Let F (for "full") be the union $\cup_{i \in I''} C_i$ 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 $E(F, \overline{F})$, recall $X_i \geq C_i(1 - \delta_1/4)$ for $i \in I''$. Summing over such i we get $F \leq X''/(1 - \delta_1/4) \leq 4X''/3 \leq 4X/3 \leq 2ND/3$. By Exercise 12.1, $E(F) \geq 0.5dF\delta_1$. Next, the term $E(\cup_{i \in I''} \overline{X_i}, \overline{F})$ is $\leq (I'' \cdot D \cdot \delta_1/4)d = (F\delta_1/4) \cdot d$. (Note each node in $\cup_{i \in I''} \overline{X_i}$ has $\leq d$ neighbors outside of F.) So the first difference in the rhs in equation (12.1) above is $\geq (F\delta_1/4) \cdot d$. Because $F \geq (1 - \delta_1/4)X'' \geq (1 - \delta_1/4)(1 - \delta_1/10)X \geq (3/4)(9/10)X \geq 0.5X$, this difference is $\geq Xd\delta_1/8$. Finally, $E(X', \overline{X'})$ is trivially $\leq dX' \leq dX\delta_1/10$. Hence $E(X) \geq Xdc\delta_1$. **QED**

We can now mix these three ingredients to construct explicit expanders.

Proof of Theorem 12.1.. Start with E_1 the expander from Theorem 12.3 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 $\log^c n$. That is, one replacement product allows us to reduce the degree logarithmically. Doing this again, we can reduce the degree to $\log^c \log n$. Finally, we take a replacement product with the non-explicit expander from Theorem 12.2 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 12.4. **QED**

Exercise 12.3. Prove that the neighbor function is computable in L.

12.2 Eigen stuff

We now turn to eigenvalues, a.k.a. spectral, analysis. (I suspect one can also present the main results in the next sections without eigenvalues, but not clear there is much gain and eigenvalues are important for many things, so we won't pursue that direction now.) Let us view a vector as a *probability distribution* over the vertices of a graph G. For example, $u = (1/n, \ldots, 1/n)$ is the uniform distribution over n vertices. For now, we also assume that the graph G is *regular*, i.e. each vertex has degree d, and also that each vertex has a self-loop. We justify this assumption later for our applications. For each graph G, we have a normalized adjacency matrix A, where "normalized" means each entry is divided by the degree d. The following matrix is the adjacency matrix of the graph in Figure 12.4.

$$A = \begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 2/3 & 0 \\ 1/3 & 0 & 2/3 \end{bmatrix}.$$

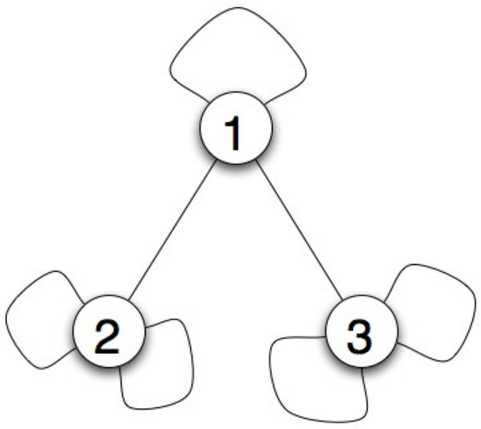


Figure 12.4: Graph example.

For a starting random vector v, Av would be the probability distribution after doing one random step starting at v. For example, for $v = (1, 0, 0)$ we have $Av = (1/3, 1/3, 1/3)$ in the graph above. It would be convenient if the act of multiplying by A corresponded to some simple behavior, like $Av = \lambda \cdot v$ for some scalar λ. Such λ and v are called *eigenvalue* and *eigenvector* respectively. Although every matrix has them, in general they need to be complex, as for example is needed if A corresponds to a rotation of the space. However, the matrices that arise from graphs have a special structure, in particular they are symmetric. One can show that in this case there is a basis of n *real* eigenvectors, called an *eigenbasis*. Moreover, we can choose them to be *orthonormal*, i.e. length-1 vectors that have zero inner product.

Theorem 12.5 (Eigenbasis)**.** Let A be an $n \times n$ real symmetric matrix. Then there exists an orthonormal basis of real eigenvectors $v_1, v_2 \ldots, v_n \in \mathbb{R}^n$.

Some thoughts on the proof are in section A.5.1. This theorem allows us to write any vector v in the eigenbasis and see the act of multiplying the vector by the matrix A as simply multiplying each coordinate of v by the corresponding eigenvalues.

We list several basic properties.

Lemma 12.1. Let A be the normalized random-walk matrix of a d-regular graph G. We have:

(1) The eigenvalues of A are in $[-1, 1]$. This holds more generally for any matrix that is real, symmetric, where each entry is ≥ 0, and each row sums to ≤ 1 (a.k.a. *row sub-stochastic).*

(2) If moreover G has a self-loop on each node and is d-regular then the eigenvalues are in $[-1 + 1/d, 1]$.

(3) If G is regular then the uniform distribution is an eigenvector with eigenvalue 1. Its multiplicity (that is, the dimension of the span of the eigenvectors) equals the number of connected components of G.

(4) G is bipartite iff -1 is an eigenvalue.

Proof. (1) Let v be an eigenvector and wlog let $v_1 = \max_i |v_i|$. As $Av = \lambda v$, in particular $|(Av)_1| = |\lambda v_1|$. We have $|(Av)_1| = |\sum_i A_{1i} v_i| \leq \sum_i A_{1i}|v_i| \leq v_1 \sum_i A_{1i} \leq v_1$. Hence $|\lambda v_1| \leq |v_1|$ and the result follows. **QED**

Exercise 12.4. Prove (2)-(4). Hint: For (2), consider $A - I/d$, then use (1). For (3), first give orthogonal eigenvectors, then prove every other eigenvector with eigenvalue 1 is in the span.

The next useful lemma gives a useful characterization and property of the second largest eigenvalue, in absolute value order. It shows that it gives a bound on how closer we get to uniform after taking a random step. We simply write $|v|$ for the 2-norm $|v|_2 = \sqrt{\sum_i v_i^2}$.

Lemma 12.2. Let A be the normalized adjacency matrix of a connected graph G. Sort the eigenvalues of A by absolute value: $1 = |\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_n| \geq 0$ then

$$|\lambda_2| = \max_{v \perp u} \frac{|Av|}{|v|}$$

where u is the uniform distribution.

In particular, for any probability distribution v we have

$$|Av - u| \le |\lambda_2||v - u|.$$

Proof. Take any vector $v \in \mathbb{R}^n$. Write $v = a_1v_1 + \cdots + a_nv_n$ where the v_i are the eigenvectors of the matrix A, from Theorem 12.5, and $v_1 = u$. Because $v \perp u$ and G is connected we have $a_1 = 0$ by Lemma 12.1. So $v = a_2v_2 + \cdots + a_nv_n$. We now have, by Fact A.11 and orthonormality of the v_i:

$$|Av|^2 = |\lambda_2 a_2 v_2 + \cdots + \lambda_n a_n v_n|^2 = |\lambda_2 a_2|^2 + \cdots + |\lambda_n a_n|^2 \le |\lambda_2|^2|v|^2.$$

This bound is met by $v = v_2$.

The "in particular" part is because $|Av - u| = |A(v - u)|$ and $(v - u) \perp u$ by direct verification. So by the previous claim $|A(v - u)| \le |\lambda_2||v - u|$. **QED**

Exercise 12.5. Let A and G be as in Lemma 12.2. Sort the eigenvalues (without taking absolute values) as $1 = \lambda_1 \ge \lambda_2 \ge \ldots \ge \lambda_n \ge -1$. Prove

$$\lambda_2 = \max_{v \perp u} \frac{\langle Av, v\rangle}{\langle v, v\rangle},$$
$$1 - \lambda_2 = \min_{v \perp u} \frac{\sum_{\{i,j\}\in E}(v_i - v_j)^2}{d \cdot \langle v, v\rangle}. \tag{12.2}$$

The rhs of Exercise 12.5 gives a useful way to think of the *spectral gap* $1 - \lambda$ as of minimizing a certain "energy" function on the edges over the choice of a uniform edge.

Notation 12.1. We write λ_i for the eigenvalues in standard order, and λ'_i for those in absolute value order. We write $\lambda_2(G)$ and $\lambda'_2(G)$ for the maximum non-trivial eigenvalue in each order, and omit G when is clear from the context.

Spectral vs. edge expansion We now show that a graph is an edge expander in the sense of Definition 12.1 iff λ'_2 is small. We first illustrate this via the following "gem" that directly relates the bias of the construction in Theorem 12.3 to the second largest eigenvalue.

Theorem 12.6. 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 $\lambda'_2 \le \epsilon$.

In particular, explicit graphs with λ'_2 exist with degree $(\epsilon^{-1}\log N)^c$.

Proof. By Lemma 12.2 we need to bound $|Av|/|v|$ for any vector $v \perp u$. Let's pick a natural and convenient basis. The 2^n vectors $\chi_S(x) := (-1)^{\langle S,x\rangle}$ of length 2^n, for $S \in [2]^n$, are orthogonal, hence independent (Fact A.13). Let us write v in this basis: $v = \sum a_S \chi_S$. Because $v \perp u$, $a_0 = 0$.

We now have

$$Av = \lambda v = \sum \lambda a_S \chi_S = \sum_{S \neq 0} a_S A \chi_S = \sum a_S \epsilon_S \chi_S,$$

where $\epsilon_S = \mathbb{E}[\chi_S(D)]$ is the bias of χ_S wrt D. is To check the last equation:

$$(A\chi_S)(x) = \sum_y A_{x,y}\chi_S(y) = \sum_y \mathbb{P}[D = x+y]\chi_S(y) = \sum_y \mathbb{P}[D = y]\chi_S(x+y) = \chi_S(x)\mathbb{E}[\chi_S(D)],$$

where we replace y with $x+y$ and then use $\chi(x+y) = \chi(x)\cdot\chi(y)$

Hence

$$\frac{|Av|}{|v|} = \frac{\sqrt{\sum_{S\neq 0} a_S^2 \epsilon_S^2}}{\sqrt{\sum_{S \neq 0} a_S^2}} \leq \epsilon.$$

The "in particular" follows from Theorem 11.5. **QED**

More generally, we have the following connections between λ and δ in Definition 12.1. The bottom line is that each of δ and λ is bounded iff the other is.

Theorem 12.7. [Edge vs. spectral expansion] Let G be a graph, and let δ be the maximum s.t. G is a δ-expander. Then:

(1) [Spectral expansion $\Rightarrow$ edge expansion] $\delta \geq (1-\lambda_2)/2$,

(2) [Edge expansion $\Rightarrow$ spectral bound] $\lambda_2 \leq 1 - \delta^2/2$,and

(3) [Edge expansion + self-loops $\Rightarrow$ absolute spectral expansion] if G has self-loops on each node then

$$\lambda_2' \leq 1 - \min\{\delta^2/2, c/d\}.$$

Proof. 2(1) We use the characterization in Exercise 12.5. Let $x_i := \overline{S}$ if $i \in S$ and $-S$ if $i \in \overline{S}$. Note that x is orthogonal to uniform. By Exercise 12.5, $1 - \lambda_2(G) \geq \sum_{\{i,j\}\in E}(x_i - x_j)^2/d\langle x, x\rangle$.

The numerator of the rhs is $2E(S,T)V^2$ since each edge leaving S contributes V^2. Also note that $\langle x, x\rangle = S\overline{S}^2 + \overline{S}S^2 = S\overline{S}V$. Hence

$$1 - \lambda_2(G) \leq \frac{2E(S,\overline{S})V^2}{dS\overline{S}V} = \frac{2E(S,\overline{S})V}{dS\overline{S}} = \dagger(S)\frac{2V}{\overline{S}} \leq 2\dagger(S).$$

(2) Let $Q := I - A$. Note if λ is an eigenvalue of A then $1-\lambda$ is an eigenvalue of Q. So it suffices to prove that the eigenvalues of Q are $\geq \delta^2/2$. Let z be an eigenvector orthogonal to uniform. This orthogonality implies that z has both coordinates > 0 and also < 0. Sort the coordinates as $z_1 \geq z_2 \geq z_3 \geq ... \geq z_n$. We assume that exactly $m \leq n/2$ of its coordinates are ≥ 0. (Otherwise, one can consider $-z$.) Note that $(Qz)_i = \lambda z_i$ for all i. In particular,

$$\lambda = \frac{\sum_{i\leq m}(Qz)_i z_i}{\sum_{i \leq m} z_i^2}.$$

Consider the numerator, multiplied by d. Plugging the definition of Q and expanding out, and writing E for the multi-set of edges, e.g. $E = \{\{1,2\},\{2\},\{2\},\{1,2\},\{2,3\}\}$ (two self-loops, and two parallel edges), we can write it as

$$\begin{aligned}\sum_{i\le m}(dz_i^2 - \sum_{j:\{i,j\}\in E} z_iz_j) &= \sum_{\{i,j\}\in E: i<j\le m}(z_i - z_j)^2 + \sum_{\{i,j\}\in E: i\le m<j} z_i(z_i - z_j)\\ &\ge \sum_{\{i,j\}\in E: i<j\le m}(z_i - z_j)^2 + \sum_{\{i,j\}\in E: i\le m<j} z_i^2. \qquad (12.3)\end{aligned}$$

Let x be equal to z except negative entries are zero. We claim that we can switch from z to x, and in fact sum over all edges: The rhs above is

$$\ge \sum_{\{i,j\}\in E}(x_i - x_j)^2.$$

To verify this, note that edges with $i < j \le m$ contribute the same in the expression with the z and with the x. The same holds for edges with $i \le m < j$ since $z_j = 0$. And edges with both i and j bigger than 0 contribute nothing.

The key thing is that this sum is now over all edges, so we can rely on the expansion of the graph. For intuition, suppose that x was "flat:" equal to 1 over $[m]$, and 0 otherwise. In this case, the contribution of $x_i - x_j$ is 0 unless $\{i,j\} \in E([1..m],[m+1..n])$, in which case it is c. By edge expansion, the number of crossing edges is $\ge \rho m d$. Hence the numerator is $\ge c\rho m$. The denominator is m. Hence

$$\lambda \ge \frac{c\rho m}{m} \ge c\rho$$

and we are done.

The rest of the proof is for the general case where x may not be flat; it is based on the same idea but it has a few algebraic manipulations. Let x be equal to z except negative entries are zero. We can replace z_i with x_i for any $i \le m$ by definition, and replace $\sum x_i^2$ with $|x|^2$. Hence by above we have

$$\lambda \ge \frac{\sum_{\{i,j\}\in E}(x_i - x_j)^2/d}{|x|^2}. \qquad (12.4)$$

Now our goal is to somehow "turn" $(x_i - x_j)^2$ into $x_i^2 - x_j^2$, because it allows us count more easily the contribution from each term. For this purpose, consider the quantity $\sum_{\{i,j\}\in E}(x_i + x_j)^2/d$. Note that this equals $|x|^2$ up to constants. Indeed, it is larger than $\sum_{\{i,j\}\in E}(x_i^2 + x_j^2)/d = |x|^2$, the equality holding because the degree of each node is d. Also, it is $\le \sum_{i,j} A_{i,j}(x_i + x_j)^2 \le c|x|^2 + c\sum_{i,j} A_{i,j}x_ix_j \le c|x|^2$ by Fact A.8.

Hence multiplying equation (12.4) by $d|x|^4$ we obtain

$$d|x|^4\lambda \ge c\sum(x_i - x_j)^2 \cdot \sum(x_i + x_j)^2 \ge c\left(\sum(x_i - x_j)(x_i + x_j)\right)^2 = c\left(\sum_{\{i,j\}\in E}(x_i^2 - x_j^2)\right)^2 \qquad (12.5)$$

where we use Fact A.8.

Finally, we can rewrite the inner sum telescopically as

$$\sum_{\{i,j\}\in E, i<j} \sum_{k=i}^{j-1} x_k^2 - x_{k+1}^2.$$

Note now that each term $x_k^2 - x_{k+1}^2$ appears as many times as the number of edges $\{i,j\}$ with $i \le k < j$. Hence the double sum equals

$$\sum_{k=1}^{m} E([1..k],[k+1..n])(x_k^2 - x_{k+1}^2);$$

where we use that $x_k = 0$ for $k \ge m$. By expansion and the fact that $m \le n/2$, the E term is $\ge dk\rho$. So the sum is

$$\ge d\rho \sum_{k=1}^{n/2} k(x_k^2 - x_{k+1}^2) = d\rho \left(\sum k x_k^2 - (k-1)x_k^2\right) = d\rho |x|^2.$$

Plugging this bound inside equation (12.5) we obtain $\lambda \ge c\rho^2$. **QED**

Exercise 12.6. Prove equation (12.3) in the proof. Prove (3) in Theorem 12.7.

Exercise 12.7. Prove that a connected graph on n nodes with a self-loop on each node has $\lambda_2' \le 1 - 1/n^c$. Hint: Use Theorem 12.7.

Analysis of the random-walk algorithm Using eigenvalues we can analyze a simple randomized log-space algorithm for UConn (cf Definition 7.5). This *random walk algorithm*, on input a graph on n nodes and vertices s and t 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 12.8. The random walk algorithm runs in logarithmic space and has error prob. $\le$ 1/2.

Exercise 12.8. Prove this. Guideline: Use Lemma 12.2 and Exercise 12.7, Divide the walk of length ℓ in sub-walks of length $\sqrt{\ell}$. First prove that each subwalk has noticeable prob. of reaching t if it is connected to s.

12.3 Robust UConn

To introduce the approach, note that we can solve UConn (cf Definition 7.5) in *deterministic* log-space on graphs with $\lambda_2' \le 1/n^c$. This is because by Lemma 12.2 the distance between Av and u is $\le 1/n^c$. But if Av puts no mass on t the distance would be larger. Hence simply trying all neighbors of s we can determine if s and t are connected. Our input graph doesn't have to satisfy this strong requirement, nevertheless just the fact that it's connected (which is the interesting case of the analysis) implies a noticeable spectral gap (Exercise 12.7).

Thus we would be done if we could somehow turn the graph into a strong expander where every non-trivial eigenvalue is $\leq 1/n^c$ in absolute value. While this can be done, it is slightly more convenient to work with a "robust" version of the problem which will "only" require a constant spectral bound. The following claim also justifies our assumption that graphs are regular, which we used in previous sections.

Claim 12.1. UConn is log-space reducible to the following robust-UConn problem: given a 4-regular graph with a self-loop on each node, and given two connected sets of nodes S and T of density $\geq 1/3$, decide whether S and T are in the same connected component.

Proof. Given an instance G, s, t of UConn, where G has n nodes, we construct G' in the following way. Add n copies of s and n copies of t. Set S consists of all copies of s, and same for T. Put $n/2$ copies of a cycle on S, and similarly on T, as shown in Figure 12.5, so that the extra copies of s and t have degree n. Observe that the degree of each node is $\leq n$

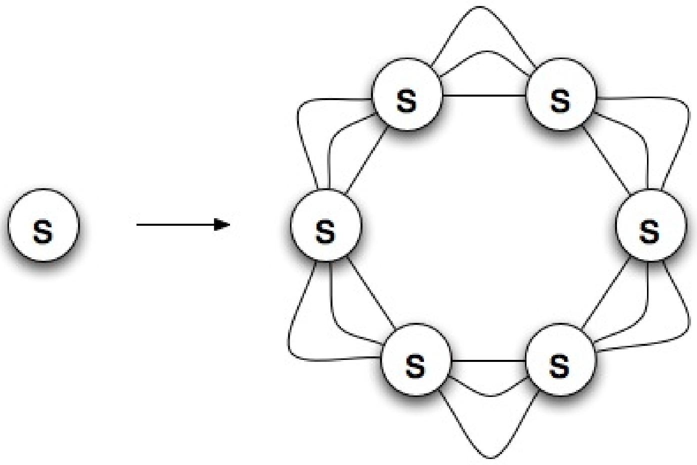


Figure 12.5: Duplicate s and add cycles.

except for s and t which may have degree as large as $2n$. To reduce the degree, replace each node of degree d with a 4-regular graph on d nodes, as shown in Figure 12.6. And call this final graph G'.

By construction, the number of nodes in G' is n^c, G' is 4-regular, and $|S| \geq |G'|/3$, $|T| \geq |G'|/3$. The bound on $|S|, |T|$ follows because (1) this bound holds before reducing the degree, (2) the degree reduction blows up a node by its degree, and (3) every node in $S \cup T$ has degree at least as large as that of any other node in the graph. **QED**

We now show that this more robust version is "easy" when the second eigenvalue is small. This is similar to the previous observation that we can solve UConn easily on graphs with

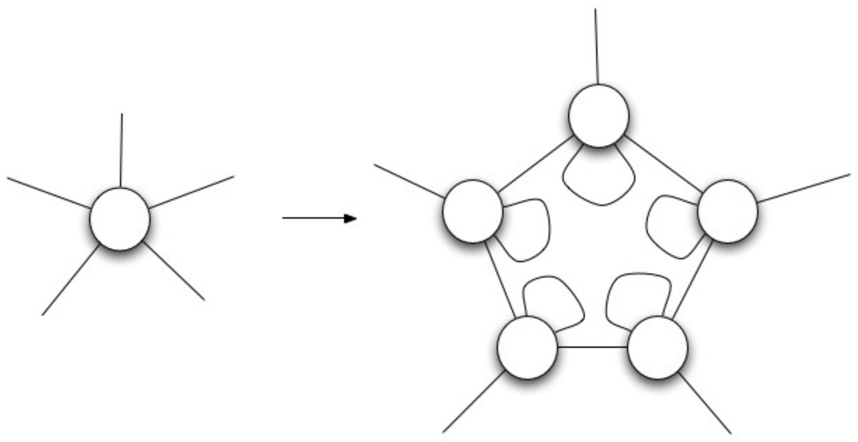


Figure 12.6: Replace each node with a 4-regular graph.

very small eigenvalue bound. However, we will now work with constant eigenvalue bound. We do not restrict the degree: we will apply the following claim to graphs of power degree. Jumping ahead, these will arise by modifying the 4-regular graphs given by Claim 12.1 using appropriate operations that reduce the eigenvalue bound.

Claim 12.2. Let G be a graph with $\lambda_2' \leq 1/10$. Let S and T be connected sets of nodes of density $\geq 1/3$. If some node in S is *connected* to some node in T, then some node in S is *adjacent* to some node in T.

Note the reverse implication is trivial. Therefore, if the neighbors of G are computable in logspace then we can also decide in log-space whether given dense sets S and T are in the same connected component, by cycling over all nodes in S and their neighbors.

The basic idea is that since S is large and λ_2' is small, S has many neighbors ($> 2n/3$), and one of them must be in T.

Exercise 12.9. Prove Claim 12.2. Guideline: Let u represent the uniform distribution (i.e. $u = (1/n, 1/n, \ldots, 1/n)$), v represent the uniform distribution on S (i.e. $v = (3/n, \ldots, 3/n, 0, \ldots, 0)$ where the coordinates with mass $3/n$ are exactly those in S). Our goal is to show Av has non-zero weight on some coordinate in T. Bound $|Av - u|$ from above using 12.2 and from below in case Av has no mass on T.

We are now left with the task of reducing the eigenvalue bound of our graph.

12.3.1 An attempt to reduce the eigenvalue bound

One attempt to reduce the eigenvalue of a graph is by squaring. The squared graph is the graph in which edges correspond to paths of length 2 in the original graph; Figure 12.7 shows an example.

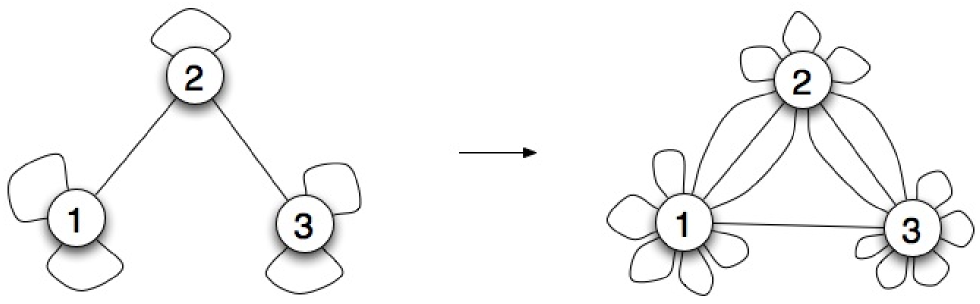


Figure 12.7: Squaring a graph.

Note that the eigenvectors of (the normalized adjacency matrix A of) G^2 are the same as those of G, and the corresponding eigenvalues square, because

$$A^2 v = A(Av) = \lambda A v = \lambda^2 v. \tag{12.6}$$

So if we start with a connected graph with a self-loop at each node – which has $\lambda_2' \leq 1 - 1/n^c$ by Exercise 12.7– and we square it $\ell = c \log n$ we obtain the bound

$$\lambda_2'\left(G^{2^\ell}\right) \leq \left(1 - \frac{1}{n^c}\right)^{2^\ell} = \frac{1}{10}.$$

Although we obtain the desired eigenvalue bound, the degree of the graph G^{2^ℓ} is $D^{2^\ell} \geq D^n$, which is exponential in n. This means we cannot apply Claim 12.2 to determine connectivity in logarithmic space, since the idea there was to cycle over all neighbors of nodes in S.

In order to apply Claim 12.2, we need another graph operation that can decrease λ_2' while at the same time keeping the degree small. Note that an edge in G^{2^ℓ} corresponds to a path of length n^c in G. With the new operation we will still have that an edge in the final graph corresponds to a path of that length in G. But the crucial difference is this: whereas in G^{2^ℓ} we consider all, exponentially many paths, in the new graph we only consider a sparse, polynomial-size collection of paths. We will prove that this sparse collection has the same hitting properties of the collection of all paths, as measured by the eigenvalue bound.

12.3.2 Reducing the eigenvalue via derandomized graph squaring

Derandomized graph squaring is similar to the replacement product (cf Definition 12.3). One advantage is that it does not increase the number of nodes.

Definition 12.4. [Derandomized graph squaring] Let X be a k-regular graph on n nodes, and G be a d-regular graph on k nodes. $X \ominus G$ is a graph on n nodes with degree $k \cdot d$. The neighbors of v in $X \ominus G$ are $v[a][b]$ where b is a neighbor of a in graph G, i.e. $v[a][a[e]]$ where $a \in [k]$ and $e \in [d]$.

Note that in the above definition we see a as both an edge index for X and a node in G. Whereas in graph squaring the neighbors of v are $v[a][b]$ for *every* a, b, in derandomized graph squaring the neighbors are $v[a][b]$ for *some* a, b. The following main result shows that applying derandomized graph squaring we somewhat decrease λ_2:

Theorem 12.9 (Analysis of derandomized graph squaring)**.** If $\lambda_2'(X) = \lambda$ and $\lambda_2'(G) = \mu$, then $\lambda_2'(X \ominus G) \leq (1-\mu)\,\lambda^2 + \mu$.

To illustrate parameters, note when μ is sufficiently small, derandomized squaring essentially squares λ. Numerically, one can verify that if $\lambda_2'(G) = \frac{1}{100}$ and $\lambda_2'(X) = 1 - \gamma \geq 1/10$, then $\lambda_2'(X \ominus G)| \leq 1 - \frac{12}{11}\gamma$. So if we start with $\lambda_2' \leq 1 - 1/n^c$ and repeat this operation $c \log n$ times we will get $\lambda_2' \leq 1/10$, qualitatively the same as graph squaring.

To prove Theorem 12.9 we start with a useful lemma that shows that a random step in a graph G with $\lambda_2'(G) = \lambda$ can be seen as going to the uniform distribution with probability $(1-\lambda)$, and not doing harm otherwise. Denote by J_n the $n \times n$ matrix with $1/n$ everywhere. Multiplying any probability distribution v by J we obtain the uniform distribution u.

Lemma 12.3. Let G be a graph on n nodes with $n \times n$ normalized adjacency matrix A satisfying $\lambda_2'(A) = \lambda$. Then $A = (1-\lambda)J_n + \lambda C$ where $\forall v : |Cv| \leq |v|$.

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 $u \perp w$. Then

$$|Cv|^2 = |au + Aw/\lambda|^2 = |au|^2 + |Aw/\lambda|^2 \leq |au|^2 + |w|^2 = |v|^2.$$

The first equality is obtained by definition and using $Au = u$ and $Jw = 0$. The rest is Fact A.11, Lemma 12.2, and Fact A.11 again. **QED**

Exercise 12.10. Prove the first equality.

To analyze the adjacency matrices arising from derandomized graph squaring we write random-walk matrices in a specific way, also using tensor products (Definition A.1). Let us illustrate in a simple case. Taking a random step in a graph G can be thought as as 3-step process:

$$v \to (v, a) \to (v[a], a) \to v[a],$$

where a is a random edge. This is overkill when dealing with a single random-walk matrix A, but it will be useful when analyzing derandomized graph squaring as we will be able to "remember" the edge label. Each of the three steps above can be implemented by a different matrix, to obtain

$$A = P\tilde{A}L$$

Let G be d-regular on n nodes. The first step is given by the "lift" matrix $L := I_n \otimes (1/d, \ldots, 1/d)^T$ which is $n \cdot d \times n$.

The second step is given by $\tilde{A}$ which is a $n \cdot d \times n \cdot d$ matrix, where $\tilde{A}_{(u,a),(u',a')} = 1$ if and only if $a = a'$ and $u' = u[a]$. This matrix corresponds to taking a step in G *after the choice for the step has been made.* No entropy is added by $\tilde{A}$, which is a permutation matrix.

Finally, the last step is given by the "projection" matrix $P := I_n \otimes (1, \ldots, 1)$ which is $n \times n \cdot d$. Starting with a distribution vector v concentrated on a single node, L spreads the mass onto the d edges, $\tilde{A}$ permutes the node accordingly, keeping the edge label intact, and finally P collects the mass.

Proof of Theorem 12.9. Let A be the normalized adjacency matrix of X, and B be the normalized adjacency matrix of G. We can view a random step in the derandomized-squaring graph $X \ominus G$ as

$$v \to (v, a) \to (v[a], a) \to (v[a], b) \to (v[a][b], b) \to v[a][b]$$

where a is a random node in G and b is a random neighbor of a in G.

We now define matrices that implement each of the above steps.

The first step is given by the "lift" matrix $L := I_n \otimes (1/k, \ldots, 1/k)^T$ which is $n \cdot k \times n$.

The second step is given by $\tilde{A}$ which is a $n \cdot k \times n \cdot k$ matrix, where $\tilde{A}_{(u,a),(u',a')} = 1$ if and only if $a = a'$ and $u' = u[a]$. This matrix corresponds to taking a step in X *after the choice for the step has been made.* No entropy is added by $\tilde{A}$, which is a permutation matrix.

The third step is given by $\tilde{B} = I_n \otimes B$.

The fourth step is $\tilde{A}$ again.

Finally, the fifth step is given by the "projection" matrix $P := I_n \otimes (1, \ldots, 1)$.

The adjacency matrix M of $X \ominus G$ satisfies

$$M = P\tilde{A}\tilde{B}\tilde{A}L.$$

By Lemma 12.3, $B = (1 - \mu)J_k + \mu C$ where $|Cv| \leq |v|$ for all v. It follows that

$$\tilde{B} = I_n \otimes B = (1 - \mu)I_n \otimes J_k + \mu I_n \otimes C.$$

Plugging this into the expression for M one gets

$$M = (1 - \mu)P\tilde{A} \left(I_n \otimes J_k\right) \tilde{A}L + \mu P\tilde{A} \left(I_n \otimes C\right) \tilde{A}L.$$

One can now observe the following:

(1) $I_n \otimes J_k = L \cdot P$;

(2) $P \cdot \tilde{A} \cdot L = A$, as remarked before the proof; and

(3) $D := P\tilde{A} \left(I_n \otimes C\right) \tilde{A}L$ satisfies $|Dv| \leq |v|$ for every v. This can be shown also using the fact that C satisfies this property as we saw before.

Plugging (1) and (2) in the above expression for M, and the definition of D we get

$$M = (1 - \mu)P\tilde{A}LP\tilde{A}L + \mu D = (1 - \mu)A^2 + \mu D.$$

Then, by Lemma 12.2 and the triangle inequality for $|.|$:

$$\lambda_2'(X \ominus G) = \max_{v \perp u} \frac{|Mv|}{|v|} \leq \frac{|(1 - \mu)A^2 v|}{|v|} + \frac{|\mu D v|}{|v|} \leq (1 - \mu)\lambda^2 + \mu.$$

QED

Exercise 12.11. Prove (3). Guideline. For a matrix A define $||A|| := \max_v |Av|/|v|$. This is like λ_2' (cf Lemma 12.2), except v does not need to be perpendicular to u. Our goal is to show $||D|| \le 1$. Prove:

(I) $||A \cdot B|| \le ||A|| \cdot ||B||$,

(II) $||A \otimes B|| \le ||A|| \cdot ||B||$. Hint: Write $|A \otimes Bv|^2 = \sum_{iA,iB} \left(\sum_{jA,jB} A_{iA,jA} B_{iB,jB} v_{jA,jB} \right)^2$,

(III) For a permutation matrix Π such as $\tilde{A}$,$||\Pi|| = 1$,

(IV) $||L|| = 1/\sqrt{k}$, $||P|| = \sqrt{k}$.

Combine this to prove $||D|| \le 1$.

12.4 Proof of Theorem 7.7 that Uconn is in L

We can assume that we are given a graph G and nodes s and t which are connected, because if s and t are not connected, the algorithm we are about to present will never declare them connected. By Claim 12.1, we can focus on 4-regular graph with sets S and T. Call this 4-regular graph X. To start-up our sequence, let $X_1 := X^t$ for a suitable constant t.

Define the sequence of graphs

$$X_{i+1} := X_i \ominus G_i$$

where each G_i is an expander graph with degree c, $\lambda_2' \le 1/100$, on a number of nodes equal to the degree of X_i. Such expanders are obtained from Theorem 12.1. The latter gives edge expansion, and we infer spectral expansion by Theorem 12.7. The specific bound $\lambda_2' \le 1/100$ can be obtained by squaring the graph a few times, using equation (12.6). (To show that UConn is in space $c \log n \log \log n$ the log-degree expanders in Theorem 12.3 suffice.)

The neighbor function of G_i is computable in L (cf Exercise 12.3).

Let $\ell := c \log n$. By Theorem 12.9 (cf. the observation right after its proof) we have $\lambda_2'(X_\ell) \le 1/10$.

Then we can apply Claim 12.2 and solve UConn by going through all $s \in S$ and checking if one of its neighbors in X_ℓ lies in T.

It remains to verify that we can compute neighbors in X_ℓ in L.

The intuition is: if $v \in X_1$, then the neighbors are $v[a_1]$ where $a_1 \in [d]$; if $v \in X_2$, then the neighbors are $v[(a_1, a_2)] = v[a_1][a_1[a_2]]$ where $a_1, a_2 \in [d]$; if $v \in X_3$, then the neighbors are $v[(a_1, a_2, a_3)] = v[(a_1, a_2)][(a_1, a_2)[a_3]] = v[a_1][[a_1[a_2]][(a_1', a_2')]] = v[a_1][a_1[a_2][a_1'][a_1'[a_2']]]$ where $a_1, a_2, a_3 \in [d]$; etc.

An edge in X_ℓ is specified by $(a_1, a_2, \dots, a_\ell)$ where $a_i \in [d]$. To this edge there corresponds a path of length $2^{\ell-1}$ in the graph X_1 with labels $(b_1, b_2, \dots, b_{2^{\ell-1}})$ where $b_i \in [d]$. The associated neighbor of v in X_ℓ is $v[b_1][b_2] \cdots [b_{2^{\ell-1}}]$. So to compute $v[(a_1, \dots, a_\ell)]$, we proceed in 2 phases:

1. compute $(b_1, \dots, b_{2^{\ell-1}})$, and

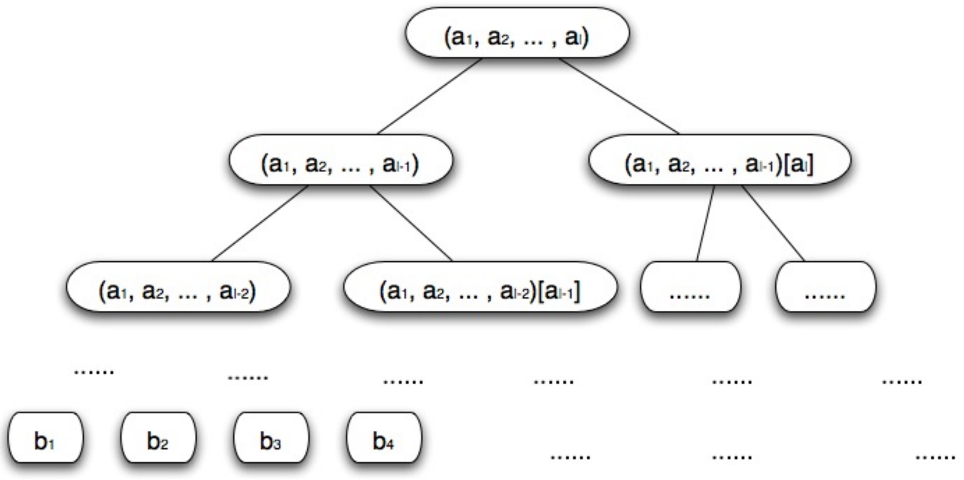


Figure 12.8: Compute b_i's from a_i's.

2. compute $v[b_1][b_2]\cdots[b_{2^{\ell-1}}]$.

We must do this in space about $\log n$, and so we cannot afford to write down the output of the first phase. Instead, the following shows given $(a_1, \ldots, a_\ell)$ and an index $i \leq 2^{\ell-1}$ how to compute $b_i \in [d]$ in space $c \log n$. From this, one can perform Phase 2 one step at the time.

Observe that the the indices b_i are obtained from the indices a_i as in figure 12.8.

So to compute b_i, we just need to go from the root to the leaf b_i in the tree. The space needed for this is just the name of the node in the tree, plus the space needed to compute neighbors in the expander graphs, which is comparable. This completes the proof that UConn is in L.

12.5 Notes

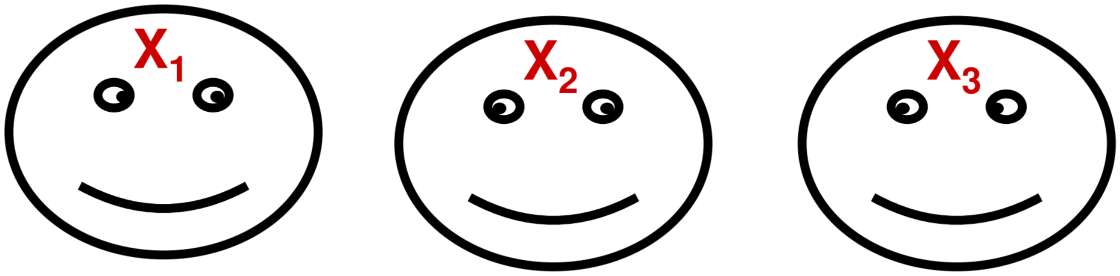
TBD expanders.

The proof of Theorem 7.7 in [220] is similar to the one we presented, but uses different graph operations to reduce the eigenvalue bound. The proof using derandomized graph squaring is from [225]. We follow the presentation in [274]

12.6 Historical vignette: TBD

Chapter 13

Communication complexity



Communication complexity is the study of *rectangles.*

(... alright, let me try to make this sound more fun to the uninitiated; though the ultimate goal is to convince the uninitiated that rectangles are fun.)

Communication complexity is the study of the amount of information that needs to be exchanged among two or more parties (or players) which are interested in reaching a *common* computational goal. The critical difference with interactive proofs is that in communication complexity the parties *cooperate.* By contrast, the setting of cryptographic proofs is *adversarial* or *cryptographic:* The verifier may be interactive with a malicious party. Other than that, the same parameters are studied in both settings, like the number of rounds, the length of the communication (which is a lower bound on the efficiency of the party), etc. But, in communication complexity we can be more basic by disregarding the computational model and instead bestowing unlimited computational power on the parties, and only paying attention to the amount of communication.

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 of two inputs

$$f : X \times Y \to Z$$

where A only knows $x \in X$, and B only knows $y \in Y$. The parties A and B engage in a communication protocol and exchange bits. We can generally assume that the parties alternate sending a message, and that the last message is the output of the protocol. We say that the protocol uses communication d bits if for every input, A and B exchange $\leq d$ bits. The bits exchanged are called *transcript*.

We can visualize a protocol via a binary tree (13.1). Each node is labeled with a function mapping that party's input to the possible messages.

TBD

Figure 13.1: Protocol tree

13.1.1 The communication complexity of equality

Consider the function Equality : $[2]^n \times [2]^n \to [2]$, Equality$(x, y) = 1 \Leftrightarrow x = y$. Trivially, Equality can be computed with communication $n + 1$: A sends her input to B; B then communicates the value of Equality. The same trivial upper bound holds for any function $f : [2]^n \times [2]^n \to [2]$. We now prove the following lower bound.

Theorem 13.1. Any protocol for equality must exchange at least n bits.

Before proving this theorem, we cover some properties of protocols.

Definition 13.1. A rectangle in $X \times Y$ is a subset $R \subseteq X \times Y$ such that $R = A \times B$ for some $A \subseteq X$ and $B \subseteq Y$. 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 13.1. Prove the equivalence.

The connection between rectangles and protocols is the following.

Lemma 13.1. Let P be a protocol that uses d bits, let $t \in [2]^d$ be a transcript. The set of inputs that induce transcript t is a rectangle.

Proof. Let $A \subseteq X \times Y$ be the set of inputs that induce communication t. Suppose that $(x, y), (x', y') \in A$, we want to show that $(x, y') \in A$ (similarly for (x', y)). We prove by induction on i that the i-th bit exchanged by P on input (x, y') is t_i. Of course this means that the protocol exchanges t on input (x, y') and so $(x, y') \in A$ as desired.

For $i = 1$, the bit sent by A only depends on x, but we know $P(x, y)$ exchanges t_1, so we are done.

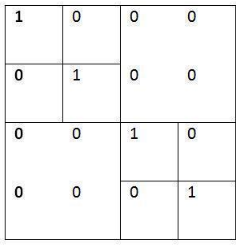
For general i, suppose it is A's turn to speak. The bit she sends is a function of x and the communication so far. By induction hypothesis the communication so far is $t_1, \ldots, t_{i-1}$. So A cannot distinguish between (x, y) and (x, y') and will send t_i as next bit.

If it is B's turn to speak, we reason in the same way replacing (x, y') with (x', y'). **QED**

Corollary 13.1. Suppose $f : X \times Y \to [2]$ is computable by a d-bit protocol, then there is a partition of $X \times Y$ in 2^d rectangles, where each rectangle is f-monochromatic: all inputs in the rectangle give the same value of f.

Proof. For each transcript t, consider $R_t :=$ the set of inputs that induce t. R_t is a rectangle by the previous lemma. It is obviously a partition and f-monochromatic. **QED**

Example 13.1. The equality function, seen as a matrix, is the identity matrix. 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.



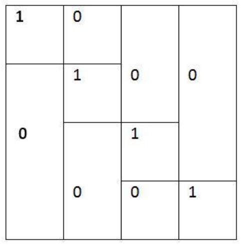


Figure 13.2: Two ways to partition equality in monochromatic rectangles.

We can now prove the lower bound for equality.

Proof of Theorem 13.1. Assume we can partition $X \times Y$ in 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 which implies that any protocol must use at least n bits of communication. **QED**

13.1.2 The power of randomness

We define randomized protocols as a distribution on protocols, similarly to the randomized polynomials in Definition 6.5.

Definition 13.2. A *randomized protocol* P with communication d is a distribution on protocols with A function $f : X \times Y \to Z$ has randomized communication d with error ϵ if there is a randomized protocol P that on every input x computes it correctly w.p. $\geq 1 - \epsilon$, that is, $\mathbb{P}_P[P(x) \neq f(x)] \leq \epsilon$.

The equality function demonstrates the power of randomness in communication:

Theorem 13.2. Equality has randomized protocols with error ϵ and communication $c \log 1/\epsilon$, for any $\epsilon \leq 1/2$.

Exercise 13.2. Prove this.

13.1.3 Public vs. private coins

Our definition of randomized protocols is *public-coin*: the parties share randomness. One can also consider *private-coin* protocols. There are defined like (deterministic) protocols, except that each party's message is a *distribution* on messages.

Exercise 13.3. Suppose that $f : [2]^n \times [2]^n \to [2]$ has a (public-coin) 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 public coin protocol needs only be supported on few protocols.

13.1.4 Disjointness

The *disjointness function* Disj : $[2]^n \times [2]^n \to [2]$ is 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, as we shall see in section 13.1.7.

Theorem 13.3. The randomized communication complexity of Disj with error ϵ is $\geq c_\epsilon n$.

The hard distribution D 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.

13.1.5 Greater than

Another well-studied function is Greater-Than, where the parties wish to determine if $x > y$ as integers.

Theorem 13.4. The randomized communication complexity of greater-than is $\leq c \log n$.

Proof. We sketch the clever protocol. We perform binary search to find the most significant bit where x and y differ. Each comparison during this binary search corresponds is an equality problem, which as we saw has small randomized communication complexity (Theorem 13.2).

The naive way to implement this search is to set the error to $\leq c/\log n$ in Theorem 13.2. But this won't give overall communication $c \log n$.

Instead, we set the error to constant, and perform *binary search with noisy comparisons.* A random-walk-with-backtrack algorithm shows that $c \log n$ 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. **QED**

The above bound is tight.

Theorem 13.5. The randomized communication complexity of greater-than is $\geq c \log n$.

13.1.6 Application to TMs

One-tape TMs have efficient randomized communication protocols. This is essentially the same as the crossing-sequence argument we saw in Chapter 3.

Theorem 13.6. For a function $f : [2]^n \times [2]^n \to [2]$ consider the padded function $p_f : [2]^{3n} \to [2]$ defined as $p_f(x0^ny) = f(x, y)$. 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 $\in [n]$, define the protocol P_i as follows: A is in charge of the first $n + i$ cells (which include x); B is in charge of last $n + (n - i)$ cells (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 cell. 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 $[n]$. **QED**

We will soon exhibit functions which require linear randomized communication, recovering the quadratic impossibility results for TMs from Chapter 3. In fact, we will show stronger results.

13.1.7 Application to streaming

tbd

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. The obvious generalization is to let k players compute a k-argument function $f(x_1, \ldots, x_k)$ where the i-th party only knows the i-th argument x_i. This model is known as "number-in-hand" and useful in some scenarios, but we will focus on a different, fascinating model which has an unexpected variety of applications: the "Number on the Forehead" model . Here, again $f(x_1, \ldots, x_k)$ is a Boolean function whose input is k arguments, and there are k parties. The twist is that the i-th party knows all inputs except x_i, which we can imagine being placed on his forehead. Communication is broadcast.

The grand challenge here is to give an explicit function $f : \overbrace{[2]^n \times ... \times [2]^n}^{k} \to [2]$ that cannot be computed with $k := 2 \log n$ parties exchanging k bits. This would have many applications, one of which is described next.

13.2.1 An application to ACC

Functions computable by small ACC (recall section §8.3) have low communication complexity:

Lemma 13.2. AC$[d]$ on n bits of size n^d and depth d have equivalent protocols with $\log^{c_d} n$ parties communicating $\log^{c_d} n$ bits, for any partition of the input bits.

Proof. By Lemma 8.2 it suffices to prove it for depth-2 circuits consisting of a symmetric gate on s And gates of fan-in t, where s and t are $\le \log^{c_d} n$. 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 $\le s$ of these gates that evaluate to 1. This takes a total of $ct \log s$ bits. **QED**

13.2.2 Generalized inner product is hard

In this section we prove an impossibility result for computing the generalized inner product function $GIP : ([2]^n)^k \to [2]$:

$$\text{GIP}(x_1, \ldots, x_k) := \sum_{i=1}^{n} \bigwedge_{j=1}^{k} (x_j)_i \mod 2.$$

In fact, we shall bound even the correlation between GIP and k-party protocols exchanging d bits, denoted Cor(GIP, d-bit k-party). Recall from section §3.5 that this is defined as the maximum of $|\mathbb{E}_x e[\text{GIP}(x) + f(x)]|$ for any protocol f with corresponding parameters, where x is uniform and $e(z) := (-1)^z$. By (the easy direction of) Corollary 3.1, this implies that the randomized communication complexity of GIP is large. Let us spell out again this implication: Having randomized communication complexity $\le d$ 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 $\le \epsilon$. 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.

Theorem 13.7. $\text{Cor}(\text{GIP}, d\text{-bit } k\text{-party}) \le 2^d \cdot 2^{-cn/4^k}$.

To prove the theorem we associate to any function a quantity $R(f) \in \mathbb{R}$ enjoying the following two lemmas:

Lemma 13.3. $\text{Cor}(f, d\text{-bit } k\text{-party}) \le 2^d \cdot R(f)^{1/2^k}$, for any $f : X_1 \times ... \times X_k \to [2]$.

Lemma 13.4. $R(\text{GIP}) \le 2^{-cn/2^k}$.

The combination of these two facts proves Theorem 13.7.

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 13.4. 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.5.

13.2.3 Proof of Lemma 13.3

We prove this theorem via a sequence of claims.

Definition 13.3. A function $g_i : X_1 \times \ldots \times X_k \to [2]$ is a *cylinder* in the i-th dimension if $\forall (x_1, \ldots, x_k)$ and x_i' we have $g_i(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_k) = g_i(x_1, \ldots, x_{i-1}, x_i', x_{i+1}, \ldots, x_k)$. 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 $S = \{x : \prod g_i(x) = 1\}$.

Recall we saw that a 2-party protocol partitions the input in monochromatic rectangles. The following extension of this fact to k parties is via cylinder intersections.

Claim 13.1. Any d-bit k-party protocol for $f : \overbrace{[2]^n \times \ldots \times [2]^n}^{k} \to [2]$ partitions the inputs in 2^d f-monochromatic cylinder intersections.

Proof. Fix a transcript t, and consider the set A_t of inputs yielding that transcript. We claim that A_t is a cylinder intersection. To see this, consider the cylinder functions $g_i(x) = 1 \Leftrightarrow$ "From the point of view of the i-th party, x could yield transcript t" $\Leftrightarrow \exists x_i'$ such that $P(x_1, \ldots, x_{i-1}, x_i', x_{i+1}, \ldots, x_k)$ yields transcript t.

Obviously if x is in A_t then $g_i(x) = 1$ for all i.

To see the converse, take some input $x = (x_1, \dots, x_k)$ such that $g_i(x) = 1$ for all i. This means that $\exists(x'_1, \dots, x'_k)$ such that
$(x'_1, x_2, \dots, x_k)$ yields t;
$(x_1, x'_2, \dots, x_k)$ yields t;
... ...
$(x_1, x_2, \dots, x'_k)$ yields t.
We must show that x yields t as well, i.e. $x \in A_t$. This is argued by induction on the bits in t, using the same "copy and paste" argument that was used for $k = 2$. **QED**

Using the notion of cylinder intersections we can now relate an arbitrary protocol to a special class of protocols p^*. Each protocol p^* can be written as $p^*(x) = \sum g_i(x) \ mod\ 2$, 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 13.2. $\text{Cor}(f, d-\text{bit}) \leq 2^d \cdot \text{Cor}^*(f)$.

Proof. We use a general trick to turn products $\overbrace{\prod_i g_i(x) = 1}^{cylinder\ \ intersection}$ into sums $\overbrace{\sum g_i(x) \ \ mod\ \ 2}^{p^*}$.
Fix any d-bit protocol, let $\{x : \prod_i g_i^1(x) = 1\}, \dots, \{x : \prod_i g_i^D(x) = 1\}$ be the corresponding $D := 2^d$ f-monochromatic cylinder intersections (by the previous claim). Observe that for a fixed x,

$$\mathop{\mathbb{E}}_{y_1,\dots,y_k \in \{-1,1\}} \left[(y_1)^{1+g_1(x)} \cdot (y_2)^{1+g_2(x)} \cdot \dots \cdot (y_k)^{1+g_k(x)}\right] = \begin{cases} 1 & \text{if } \exists i : g_i(x) = 0 \\ 0 & \text{if } \forall i : g_i(x) = 1. \end{cases}$$

Therefore,

$$e(p(x)) = \sum_{i=1}^{D} r(i) \mathop{\mathbb{E}}_{y_1,\dots,y_k \in \{-1,1\}} \left[(y_1)^{1+g_1^i(x)} \cdot (y_2)^{1+g_2^i(x)} \cdot \dots \cdot (y_k)^{1+g_k^i(x)}\right]$$

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:

$$\begin{aligned}
&\mathbb{E}e[f(x) + p(x)] \\
&= \mathbb{E}_x[e(f(x)) \cdot e(p(x))] \\
&= \mathbb{E}_x \left[e(f(x)) \cdot \textstyle\sum_{i=1}^{D} r(i) \mathop{\mathbb{E}}_{y_1,\dots,y_k \in \{-1,1\}} \left[(y_1)^{1+g_1^i(x)} \cdot (y_2)^{1+g_2^i(x)} \cdot \dots \cdot (y_k)^{1+g_k^i(x)} \right]\right] \\
&= \textstyle\sum_{i=1}^{D} \mathbb{E}_{x,y_1,\dots,y_k \in \{-1,1\}} \left[e(f(x)) \cdot r(i) \cdot (y_1)^{1+g_1^i(x)} \cdot (y_2)^{1+g_2^i(x)} \cdot \dots \cdot (y_k)^{1+g_k^i(x)}\right] \\
&\leq D \cdot \mathbb{E}_{x,y_1,\dots,y_k \in \{-1,1\}} \left[e(f(x)) \cdot r(i) \cdot (y_1)^{1+g_1^{i*}(x)} \cdot (y_2)^{1+g_2^{i*}(x)} \cdot \dots \cdot (y_k)^{1+g_k^{i*}(x)}\right],
\end{aligned}$$

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 13.3. $\mathbb{E}e_x[g(x)] \leq R(g)^{1/2^k}$ 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.7. 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:"

$$\mathbb{E}_{x_1,\ldots,x_k} e[g(x_1, \ldots, x_k)]^2 = \mathbb{E}_{x_1,\ldots,x_{k-1}} [\mathbb{E}_{x_k} e[g(x_1, \ldots, x_k)]]^2 \leq \mathbb{E}_{x_1,\ldots,x_{k-1}} [\mathbb{E}_{x_k} e[g(x_1, \ldots, x_k)]^2]$$

$$= \mathbb{E}_{x_1,\ldots,x_{k-1}} [\mathbb{E}_{x_k^0, x_k^1} e[g(x_1, \ldots, x_{k-1}, x_k^0) + g(x_1, \ldots, x_{k-1}, x_k^1)]].$$

The lemma follows by repeating this k times. **QED**

Claim 13.4. For every function $f : X_1 \times \ldots \times X_k \to [2]$, and every protocol* p^*,

$$R(f \oplus p^*) = R(f),$$

where $f \oplus p^*$ simply is the function whose output is the XOR of f and p^*.

Proof. Suppose $p^*(x) = g_1(x) + \ldots + g_k(x)$, 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,

$$\begin{aligned}
&\sum_{\varepsilon_1,\ldots,\varepsilon_k \in [2]} (f(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k}) + g_k(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k})) \\
&= \sum_{\varepsilon_1,\ldots,\varepsilon_k} f(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k}) + \sum_{\varepsilon_1,\ldots,\varepsilon_k} g_k(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k}) \\
&= \sum_{\varepsilon_1,\ldots,\varepsilon_k} f(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k}) + \sum_{\varepsilon_1,\ldots,\varepsilon_k} g_k(x_1^{\varepsilon_1}, \ldots, x_k^{0}) \\
&= \sum_{\varepsilon_1,\ldots,\varepsilon_k} f(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k}) + 2 \sum_{\varepsilon_1,\ldots,\varepsilon_{k-1}} g_k(x_1^{\varepsilon_1}, \ldots, x_k^{0}) \\
&= \sum_{\varepsilon_1,\ldots,\varepsilon_k} f(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k}) \bmod 2,
\end{aligned}$$

where the second equality holds because g_k does not depend on x_k. **QED**

The straightforward combination of the claims in this section proves Lemma 13.3.

13.2.4 Proof of Lemma 13.4

We have:

$$R(\text{GIP}) = \underset{\substack{x_1^0,\ldots,x_k^0\\ x_1^1,\ldots,x_k^1}}{\mathbb{E}} e\left[\sum_{\varepsilon_1,\ldots,\varepsilon_k\in\{0,1\}}\sum_i\prod_j (x_j^{\varepsilon_j})_i\right] = \mathbb{E}\prod_i e\left[\sum_{\varepsilon_1,\ldots,\varepsilon_k}\prod_j (x_j^{\varepsilon_j})_i\right]$$
$$= \mathbb{E}e\left[\sum_{\varepsilon_1,\ldots,\varepsilon_k}\prod_j (x_j^{\varepsilon_j})_1\right]^n = R\left(\bigwedge_k\right)^n,$$

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:

$$R(\text{GIP}) = \underset{\substack{y_1^0,\ldots,y_k^0\\ y_1^1,\ldots,y_k^1}}{\mathbb{E}} e\left[\sum_{\varepsilon_1,\ldots,\varepsilon_k\in\{0,1\}}\prod_j y_j^{\varepsilon_j}\right]^n.$$

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:

$$R(\text{GIP}) = \mathbb{E}e\left[\sum\prod_j y_j^{\varepsilon_j}\right]^n \leq (-1\cdot 2^{-k} + 1\cdot(1-2^{-k}))^n = (1-2^{-k+1})^n \leq e^{-cn/2^k}.$$

Exercise 13.5. (1) Rewrite the proof of the GIP correlation bound Theorem 13.7 in the case $k = 2$ of inner product $\text{IP}(x, y) := \sum_i x_i y_i \mod 2$, simplifying the notation.

(2) Optimize the constant in the exponent.

(3) Derive the following basic result. Let D be the uniform distribution over a rectangle $X\times Y \subseteq [2]^n \times [2]^n$ where $X = Y = 2^{\alpha n}$. Show $\mathbb{E}e[\text{IP}(D)] \leq c\cdot 2^{(2(1-\alpha)-c)n}$. Show for $\alpha = 1/2$ there is a rectangle for which the expectation is 1.

Exercise 13.6. Formally state and prove, using Theorem 13.7, that TMs running in sub-quadratic time do not correlate with IP.

13.3 Any partition

The results in the previous sections, in particular the correlation bound for GIP in Theorem 13.7 apply to a specific partition of the input in foreheads.

Exercise 13.7. Give a different partition of the input in foreheads s.t. GIP has constant communication.

To generalize these impossibility results to any partition is natural and has a number of applications. It turns out that the same parameters for GIP can be achieved under any partition.

Theorem 13.8. There is an explicit function $h : [2]^n \to [2]$ s.t. for any partition of the input into k sets of equal size, $\text{Cor}(h, d\text{-bit } k\text{-party}) \leq 2^d \cdot 2^{-n/c^k}$.

The construction is a hyper-edge analogues of GIP, and the analysis shows that in every partition we can find a large instance of GIP. In more detail, the construction of h is based on *expander graphs*. The expansion property that suffices is that any 2 sets of size $\geq t$ are connected, closely related to edge expansion Definition 12.1. From any graph, one constructs the hypergraph where the edges are the subsets of size k where one node is adjacent to all other $k-1$. Using the expansion property one can iteratively find

$$\geq \frac{n - kt}{d}$$

disjoint hyperedges, each intersecting each part. We'd like t and d to be as small as possible, and suitable expander graphs of degree d are $cn/\sqrt{d}$-interconnected. Hence setting $d := ck^2$ we find $\geq n/k^c$ disjoint hyperedges. Defining h to be the parity over hyperedges of the Ands over the bits in the hyperedge, we conclude by the GIP bound Theorem 13.7.

13.4 The power of logarithmic players

The impossibility results in the previous sections are effective when the number of players is $k \leq c \log n$, but useless when $k \geq \log n$. We now show that this is for a good reason: there are efficient protocols for large k. For generalized inner product this is unsurprising, since the function is almost always 0 for large k. But this is not clear if we replace, say, And with Majority. In fact, surprisingly there is a general protocol that works for 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, and then evaluating f on that. Here player j has row j of M on the forehead.

We first show the seminal result that there are efficient protocols whenever f and g are both symmetric.

Theorem 13.9. Let $k \geq \log n + 2$. There is a simultaneous k-party protocol with communication $c \log^3 n$ s.t. given a $k \times n$ matrix M (player j sees all M except row j) computes $(y_0, y_1, \ldots, y_k)$ where y_i is the number of columns in M with weight i.

In particular, $f \circ g^{(k)}$ has simultaneous protocols with the same efficiency in case both f and g are symmetric.

Proof. We prove this for $k = \lceil \log n + 2 \rceil$ (cf Exercise 13.8). Each player j communicates the number $a_j(i)$ of columns that they see having weight i, for every i. This takes communication $ck \log n = c \log^2 n$ per player.

We claim that the $a_j(i)$ uniquely specify the y_i, which concludes the proof.

To show this, note

$$b_i := \sum_j a_j(i) = (k-i)y_i + (i+1)y_{i+1}$$

for $i < k$. This is because a column with weight i will be seen as a column of the same weight i by $k - i$ players (those missing a 0) while a column of weight $i + 1$ will seen as having weight just i but $i + 1$ players (those missing a 1).

We in fact claim that even the b_i uniquely specify the y_i. To verify this, assume towards a contradiction that there are y_i and y'_i for $i \le k$ that satisfy these equations, are non-negative, and have the same sum, n, and for some i we have $y_i \neq y'_i$. We derive a contradiction as follows. Let $d_i := y_i - y'_i$. Since both the y_i and the y'_i satisfy the equations above, we get for $i < k$

$$(k-i)d_i + (i+1)d_{i+1} = 0.$$

Hence

$$d_i = -\frac{k-(i-1)}{i} d_{i-1} = (-1)^i \binom{k}{i} d_0.$$

We know that $d_0 \neq 0$ (for else by above d_i would be 0 too, but we assumed it is not) and in fact $d_0 \ge 1$ since it is an integer. Also note that $y_i + y'_i \ge |y_i - y'_i| = |d_i|$.

Hence we obtain the following contradiction

$$2n = \sum_{i=0}^{k} y_i + y'_i \ge |d_i| \ge \sum_{i=0}^{k} \binom{k}{i} = 2^k > 2n.$$

QED

Exercise 13.8. Explain why it indeed suffices to consider $k = \lceil \log n + c \rceil$.

In the next result, g can be arbitrary. Compared to Theorem 13.9, we obtain a worse communication bound and require interaction. However, both shortcomings can be addressed by an extension of the proof (cf Problem 13.1 and Exercise 13.9). I have chosen the simplest exposition giving the main takeaway, which is that when $k = \log^c n$ the communication is $\log^c n$.

Theorem 13.10. Let $k \ge \log n + 2$. 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.

To compute equation (13.1) each player will communicate the sum over $v : g(v) = 1$ of their terms in equation (13.2). The total sum is $\leq 2^k kn$, so $k + \log kn$ bits suffice.

There remains to compute u. Player 1 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**

The protocol in Theorem 13.10 requires interaction. We now explain how we can make it non-interactive, a.k.a. simultaneous. The main tool is the following.

Exercise 13.9. Give a simultaneous version of Theorem 13.10, with communication $c \log^3 n$.

13.4.1 Pointer chasing

We consider the basic problem of following a path in a directed graph. We have k layers, with edges going from nodes in layer i to nodes in layer $i+1$ only. 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.)

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. We shall show that the communication is at least b.

Formally, the input to the pointer-chasing function $G_k^{m,b}$ is a layered graph as above, where Layer i has mb^i nodes, and each node has outdegree 1. In other words, the input are functions g_i for $i \in [k]$ where $g_i : [mb^i] \to [mb^{i+1}]$ for $i \in [k]$, and $g_{k-1} : [mb^{k-1}] \to [2]$.

Exercise 13.10. Let $m = 1$. Give a simultaneous protocol with communication cb.

The next theorem implies that the communication is $\geq c_k mb$.

Theorem 13.11. Let P be a k-party one-way protocol using communication $\leq c_k mb$. Then $\mathbb{P}_x[P(x) \neq G_k^{m,b}] \geq c_k^m$.

For example, for $m = 1$ and fixed k we obtain a tight bound of b up to constant factors.

The total input length is $n \le kb^{k-1}$. Thus for constant k one needs communication $\ge c_k n^{1/(k-1)}$. One can work out the dependence on k and show that the bound remains non-trivial for any $k \le \log^c n$ where n is the total input length.

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 error probability is $\ge 2^{-m}$.

For the induction step, let P use communication t and $p := \mathbb{P}_x[P(x) \ne G_k^{m,b}]$. Write $x = (x_1, y)$ where x_1 is on the forehead of the first party. We have

$$\mathbb{P}_y\left[\mathbb{P}_{x_1}[P(x) \ge G_k^{m,b}] \ge p/2\right] \ge p/2.$$

Let P_a be the protocol P where the first party always communicates string a, regardless of y. We claim there exists a s.t.

$$\mathbb{P}_y\left[\mathbb{P}_{x_1}[P_a(x) = G_k^{m,b}] \ge p/2\right] \ge 2^{-t}p/2.$$

Note that the probability over x_1 is not reduced because the first party's message does not depend on x_1.

Now let $m' := mb$ and define protocol P' for $G_{k-1}^{m',b}$. On input y, P' runs P_a for r 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 bit. In case different runs give different values, the answer 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 $\le rt$.

To analyze the success probability. Fix any y for which $\mathbb{P}_{x_1}[P_a(x) = G_k^{m,b}] \ge p/2$. The probability that all the r runs are correct is $\ge (p/2)^r$. The probability that there are $\ge cm'$ bits that are not pointed to by some x^i is at most the probability that there is a set of size cm' s.t. mr pointers fall there, which is at most

$$\le \binom{m'}{cm'}(1-c)^{mr} \le c^{m'}c^{-mr} \le c^{m'},$$

for $r \ge cb$.

When that does not happen, the random guesses will be correct w.p. $\ge 2^{-cm'}$.

Overall, the success probability over uniform y is

$$\ge 2^{-t}p/2 \cdot ((p/2)^r - c^{m'}) \cdot 2^{-cm'}.$$

For $p \ge 2^{-cm}$ and $t \le cm'$, the overall success probability is $\ge c^{m'}$. **QED**

In the case $k = 3$ according to Theorem 13.11 we need communication $\ge c\sqrt{n}$. As mentioned above, this is tight for G, because of the way the layers are constructed. But one can consider the natural question of chasing pointers where each layer (except the first) has n nodes. It is a tantalizing open question whether there is a protocol with communication $\leq c\sqrt{n}$. The trivial protocol takes communication n, and one might wonder if that's tight. But a clever protocol achieves sublinear communication.

13.4.2 Sublinear communication for 3 player

For $k = 3$ we consider pointer chasing on layers of sizes $1, n, n$. Define G as

$$G(i, g, h) := h(g(i))$$

where $i \in [n]$,$g : [n] \to [n]$, and $h : [n] \to [2]$.

We consider the even more restricted *simultaneous* communication model where the players speak once, non-interactively. Naive intuition suggests that linear communication might be needed. In fact, such bounds were claimed several times, but each time the proof only applied to special cases. Indeed, we have:

Theorem 13.12. *G above has simultaneous communication $o(n)$.*

Proof. TBD Add details

We sketch the ideas in case g is a permutation π. Let H be 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 $d = (1 + \epsilon)pn$ 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.

The protocol is as follows. H is known to all.

Player 1, for each of the r cliques, announces the parity of the bits $h(x)$ for x in the clique.

Player 2 announces $h(x)$ for all the d neighbors x of i in H. This is d bits.

Player 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 Player 2 we know $h(x)$. We can subtract off all these bits to get $h(k)$.

As stated, this protocol is not simultaneous. To make it simultaneous, let Player 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. Then the referee has a bit per clique, knows which bit to look at, and knows which bits of Player 2 to consider.

Player 3 message takes $\log r + d$.

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 $G(n, 1-p^2)$. One can show that w.h.p. $\overline{H}$ has chromatic number at $\le r = o(n)$. 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 players.

Exercise 13.11. TBD Prove that if $\overline{H}$ has chromatic number r then we can partition the nodes of H in r sets so that

13.5 Problems

Problem 13.1. Extend the proof of Theorem 13.10 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.

13.6 Notes

"Because it is *basic*."

Communication complexity was initiated in [296] (to whose author the quote is credited, according to [287]). Exercise 13.3 is from [198].

Several proofs of Theorem 13.3 exist [148, 218, 31], see the books [163, 214] or the survey [57] for two different expositions. For more on disjointness see also the survey [234].

Theorem 13.4 is from [200]. For the random-walk with backtrack, see [75]. The matching lower bound, Theorem 13.5, is from [281].

The number-on-forehead model is from [56]. Computing degree-k polynomials with $k+1$ parties, used in the proof of Lemma 13.2, is from [123].

Theorem 13.7 is from [28]. Several presentations of the proof exist: [60, 215, 285]. We followed the latter.

The any-partition result, section §13.3, is from [126].

Theorem 13.10 is from [4] but our presentation has some minor differences. The result can be extended in various ways, cf. [4] and [117]. Theorem 13.9 is from [25], and is the first amazing protocol in this line of works.

The result can be extended in various ways, such as making the protocol simultaneous, applying different functions g_i to each column, allowing for more complicated matrix structures, and so on. The first amazing protocol in this line of works is from [25] and applies when g is symmetric too.

A proof of Theorem 13.11 in the case $k = 3$ appeared in [26] but did not readily extend to larger k. The proof we presented is a streamlined version of the argument in [286]. The latter paper works with trees instead of graphs to obtain slightly better parameters at the cost of a slightly more involved analysis of the number of bits hit by pointers.

The main idea in Theorem 13.12 for permutations, which we sketched, is from [212]. The extension to general functions is from [50]. These works don't quite give simultaneous protocols though.

Regarding randomness vs. determinism in number-on-forehead protocols, a non-explicit linear separation is in [38]. This work only contains a weaker explicit separation. An explicit, power separation is in [156]. A candidate for an explicit linear separation 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, there are groups where this requires $c \log\log |G| \geq c \log n$ communication, see [283], quantitatively the same explicit separation in [38].

Chapter 14

Algebraic complexity

$$\sum_{S\subseteq[n],|S|=d}\prod_{i\in S}x_i = (-1)^n \sum_{j_1,j_2,\ldots,j_d\geq 0:\sum_{k=1}^{d} kj_k=d}\;\prod_{k=1}^{n}\frac{(-1)^{j_k}}{j_k!k^{j_k}}\left(\sum_{i\in[n]}x_i^k\right)^{j_k}.$$

Stepping back, previous chapters have investigated the complexity of computing strings of length 2^n, corresponding to the truth-table of functions from $[2]^n$ to $[2]$ starting from basic strings (or functions) and changing them via simple operations. For example for boolean circuits the basic functions are the constants $0, 1$ and the variables x_i, and we combine them via And/Or/Not gates.

It is natural to consider other objects and to allow for different operations. In fact, we have already encountered other models which are more algebraic, like polynomials and matrices. In this chapter we explore more algebraic models, and in particular we consider computing other objects, namely polynomials.

Perhaps unexpectedly, the development of this theory closely parallels that of its boolean counterpart. We will encounter again many of the main themes and results seen so far, including depth reduction, impossibility results for small-depth models that are "just short" of proving major separations, the grand challenge, reductions, completeness, the surprising power of restricted models, and an algebraic analogue of NP.

14.1 Linear transformations

A very simple algebraic model consists of *Xor circuits*, made only of Xor gates on 2 bits. Obviously such circuits only compute *linear functions* $M : [2]^n \to [2]^m$. Note that any such linear function can be computed using $\leq mn$ wires. Similarly to Theorem 3.6, most linear functions require about that many wires. The challenge is to come up with an explicit linear function requiring many wires. Again similarly to the boolean setting (cf. section §8.1.2) we do not know of explicit linear functions requiring a super-linear number of wires, even for log-depth circuits (for example for $m = cn$). In this setting, the techniques developed earlier (Theorem 8.6) relate this quest to that of *rigid matrices.*

Theorem 14.1. Let $C : [2]^n \to [2]^{an}$ be a circuit with an wires and depth $a \log n$. Then the matrix M_C corresponding to the linear transformation computed by C can be written as

$$M_C = L + E$$

where 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 errors.

Exercise 14.1. Prove Theorem 14.1 by going through the proof of Theorem 8.6 and explaining what changes are needed.

As before, we can consider small-depth circuits, with unbounded fan-in Xor gates. The trivial upper bound of cnm wires mentioned above can be implemented in depth 1.

A natural candidate hard linear transformation is given by good $(n, an, bn)_2$ codes $C : [2]^{an} \to [2]^n$. Recall this means that a and b are constants independent of n, cf Exercise 2.12.(1). The fact that such codes can be linear is in Problem 6.8.

Exercise 14.2. Prove that any depth-1 Xor circuits computing an $(n, an, bn)_2$-good code of length n has $\geq c_{a,b} n^2$ wires.

You might suspect that the quadratic bound holds even for depth 2 and higher. In fact, even depth 2 suffices for quasi-linear size.

Theorem 14.2. There are good linear codes $(n, cn, cn)_2$ that are computable by depth-2 Xor-circuits with $cn \log^2 n$ wires.

Proof. It suffices to give a construction that for any non-zero input outputs a string with weight $\geq cn$, cf. Problem 6.8. Divide the gates in the middle layer in $c \log n$ *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 $[c, c]$. From this, we can obtain the output layer probabilistically by summing together one uniformly chosen gate from each block, and conclude by the tail bound Theorem 2.1. The output layer has $cn \log n$ wires.

It remains to construct the middle layer. The construction is again probabilistic. Let $X_i \subseteq [2]^k$ be the set of inputs of, say, weight $\in [2^i, 2^{i+1}]$ for $i \in [\log k]$, assuming $k = cn$ is a power of two. For each gate in the block, 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. By the probabilistic method and tail bounds (Theorem 2.1) it suffices to pick $c \log |X_i|$ gates in that block.

Now the intuition is that X_i is about $\binom{k}{2^i}$, which is $\leq (ck/2^i)^{2^i}$by Fact A.3. Hence, $\log |X_i| \leq c 2^i \log k$. 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 blocks, the total number of wires in the middle layer is $ck \log^2 k$. **QED**

Exercise 14.3. Prove $\log |X_i| \le c2^i \log k$.

Exercise 14.4. Come up with an idea to improve the number of edges to $o(n \log^2 n)$ by modifying this construction slightly. This is good practice of balancing parameters. The execution of the idea is slightly technical and deferred to Problem 14.1.

14.2 Computing integers

Another basic algebraic question is that of computing n-bit integers using arithmetic circuits over the integers, with no variables and no constants except 1 and -1. Usual counting argument like that in the proof of Theorem 3.6 show that most n-bit integers require circuits of size $n/\log^c n$. And this is again nearly tight since any integer $t \in [2]^n$ can be computed with cn operations by writing $t = 2^0 t_0 + 2^1 t_1 + \cdots + 2^{n-1} t_{n-1}$ (computing all the 2^i takes cn operations).

As usual, the grand challenge is to exhibit "explicit" integers that are hard to compute. In particular, integers that cannot be computed with $\log^c n$ operations. A prominent integer in this context is the factorial. Recall that $n!$ is a number with $\Theta(n \log n)$ bits. The log factor won't play a role in these connections, so one can informally think of $n!$ as an n-bit number.

Similarly to Theorem 1.7, we can show that if computing factorials is easy then factoring is also easy. The first conclusion of the following theorem gives the simplest setting that conveys the main idea. The second conclusion refutes a popular conjecture in cryptography that is the basis of several cryptosystems.

Theorem 14.3. Suppose $n!$ has algebraic circuits of size $\log^a n$. Then

(1) there are boolean circuits of size n^{c_a} that factor the product of any two n-bit primes p and q s.t. $p \le k < q$, for any k dependent only on n.

(2) let P and Q be i.i.d. n-bit primes, arbitrarily distributed. There is a boolean circuit of power size that given $P \cdot Q$ computes P with prob. $\ge c$.

Proof. (1) By assumption, there are algebraic circuits for $k!$ of size $\log^{c_a} k$. Note $k < q \le 2^n$, so the size is $\le n^{c_a}$. We use this circuit to compute $r := k! \mod x$, as a boolean integer. To do so, we simply run the circuit and compute $\mod x$ at each gate to keep the bit-length feasible. Finally, we output $\gcd(r, x)$ as one of the factors. **QED**

Exercise 14.5. Explain why this proof of (1) works. Hint: Fact A.1.

Prove (2).

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 it factorial is hard then another long-sought separation follows. Hence the complexity of computing factorials appears pivotal.

14.3 Univariate polynomials

A next natural question is computing univariate polynomials. We make two remarks on the model.

First, here the goal is to understand *monomials*, not coefficients, so we allow gates that compute any field element (unlike in section §14.2).

Second, an important 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 Lemma 2.1, 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. Obviously formal identities are also informal, so informal impossibility results are harder to establish than formal. Given this, the cleanest setting may be when the underlying field is infinite.

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 14.6. Let $B : [2]^n \to [2]$ be a (boolean) circuit of size s. Show that over any field there is an algebraic circuit A of size s 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 larger-depth there is a superlinear informal result that does not have a formal counterpart. For several explicit degree-d polynomials in n variables it can prove bounds of the form $cn \log d$. We state one example:

Theorem 14.4. *Computing $\sum_{i\in[n]} x_i^d$ requires size $cn \log d$.*

The beautiful proof is the combination of these two facts.

Lemma 14.1. *Computing a polynomial p and simultaneously all its n partial derivatives w.r.t. the n variables only costs a constant factor more than computing p.*

Lemma 14.2. *Computing the polynomials $p_i(x_1, x_2, \ldots, x_n)$ for $i \in [n]$ where x are n variables requires size $\geq cn \log d$.*

We now turn to constant-depth circuits.

Algebraic impossibility from boolean impossibility

Note that over the field $\mathbb{F}_2$ the informal imp. results obtained in section 8.4 (see especially Theorem 8.12) are algebraic (since And is like multiplication) – and nothing better is known even if one is informal, for small depth. The techniques in section 8.4 can be extended to slightly larger fields. The idea is similar to that in section 8.4: we show that such circuits are approximated by low-degree polynomials. We sketch this idea in the case of $\Sigma\Pi\Sigma$ circuits, highlighting where the field size 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 over a uniform input it's 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 $\le r$ linear forms. Now if we expand the Π gate we will have a sum of products of these r linear forms. Now we can use the fact that over a field of size q we have $X^q = X$, so we reduce the degree of each form in any product to at most $q-1$. Overall, the degree will be $\le (q-1)r$.

Using these ideas one can obtain algebraic impossibility results over small fields. But for larger, or infinite fields a different set of techniques appears necessary, and only formal results are known.

14.4.1 VNP

Similarly to NP, an important class of polynomials can be defined by summing over all boolean values of a set of variables.

Definition 14.1. The Σ-algebraic circuits $S(X_1, \ldots, X_n)$ of size $\le s$ are those that can be written as $\Sigma_{y_1,\ldots,y_s \in [2]} C(X_1, \ldots, X_n, y_1, \ldots, y_s)$ where C is an algebraic circuit of size $\le s$.

Several polynomials of interest that are not known to have small algebraic circuits can be shown to have small Σ-algebraic circuits.

Example 14.1. We show that the *permanent* polynomial in n^2 variables x,

$$p(x) = \sum_{\pi} \prod_{i \in [n]} x_{i,\pi(i)}$$

where π ranges over all permutations of $[n]$, has Σ-algebraic circuits of size n^c. It is an open problem whether it has (plain) arithmetic circuits of power size.

We will 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:

$$p = \sum_{M \in [2]^{n^2}} g(M) \prod_{i \in [n], j \in [n]} x_{i,j} \cdot M_{i,j}.$$

Thus it only remains to show that g has small algebraic circuits.

Exercise 14.7. Finish the example.

Similar to the P vs. NP question, the prominent question here is whether Σ-algebraic and algebraic circuits have similar power (known as the VNP vs. VP question). The next two results prove several consequences of algebraic equivalences.

First we have the following consequence in the boolean world. For simplicity we work on suitable fields.

Exercise 14.8. [Σ-algebraic circuits are easy$\Rightarrow$ Maj $\cdot$ PCkt $\subseteq$ PCkt] Suppose there is a constant a s.t. over any field any Σ-algebraic circuit of size s has an equivalent algebraic circuit of size s^a. Then Maj $\cdot$ PCkt $\subseteq$ PCkt. (Cf. section §6.3 for the definition of the operator.)

The following result connects the power of Σ-algebraic circuits to the complexity of computing integers (section §14.2). One can get similar results for other integers or even univariate polynomials (including those mentioned in 14.3). For simplicity we state this connection for circuits over the integers, and only using fixed constants.

Theorem 14.5. [If Σ-algebraic circuits are easy then so is factorial] Suppose there is a constant a s.t. over the integers, every Σ-algebraic circuit of size s using constants 0 and 1 only has an equivalent algebraic circuit of size n^a using constants 0 and 1 only.

Then $n!$ has algebraic circuits of size $\log^{ca} n$.

The proof is an excellent display of "scaling up and down" and connecting disparate complexity results.

Proof. Rather than giving circuits for $n!$, a number of $\le cn \log n$ 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 in

$$\text{Maj} \cdot \text{Maj} \cdot \dots \cdot \text{Maj} \cdot \text{PCkt},$$

where the number of applications of the Maj operator is c. This follows from the fact that iterated multiplication of integers is in TC (Theorem 8.8). Similarly to Exercise 14.8, the hypothesis implies that Maj $\cdot$ PCkt $\subseteq$ PCkt. Repeating this c times 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 algebraic circuit over the integers and consider

$$S'(X_{n-1}, \dots, X_0) := \sum_{j_0, j_1, \dots, j_{n-1} \in [2]} C(j) X_0^{j_0} X_1^{j_1} \cdot X_{n-1}^{j_{n-1}}$$

in the variables X_i. Note that $S'(2^{n-1}, \dots, 8, 4, 2, 1)$ equals the desired factorial. 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-algebraic circuit. Applying the hypothesis again yields an equivalent n^c-size algebraic circuit C', and then again the desired factorial is $C'(2^{n-1}, \dots, 8, 4, 2, 1)$. The powers of 2 take cn operations. **QED**

14.5 Depth reduction in algebraic complexity

To set the stage, we note that reducing the depth of general algebraic circuits is impossible:

Exercise 14.9. Give an algebraic circuit of size s that does not have an equivalent algebraic 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 14.6. Any algebraic circuit of size s computing a polynomial degree d has an equivalent 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 8.6.

Theorem 14.7. Any n-variate polynomial of degree d computable by a size-n^a arithmetic circuit can be computed by a depth-3 $\Sigma\Pi\Sigma$ of size $n^{c_a\sqrt{d}}$.

We shall see in section §14.8 that impossibility results for circuits of size $n^{c\sqrt{d}}$ computing degree-d polynomials are known. Thus, as mentioned at the beginning of the chapter, the situation in the algebraic world is strikingly analogous to that in the boolean world discussed in section §7.3. We have impossibility results for small-depth circuits that are "just short" of having major consequences for larger-depth models.

14.6 Completeness

The results in section §9.1, extended to other fields, show that iterated product of 3×3 matrices is complete for algebraic 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 in section §14.5, one can extend this completeness to arbitrary circuits *computing low-degree polynomials.*

TBD: Determinant, permanent

14.7 The power of AAC: algebraic AC

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 §14.8. The RHS is an expression for a circuit of size $\geq n^d$. But in fact one can do better.

Theorem 14.8. For any d, $e_{n,d}$ has depth-3 circuits of size cn^2.

Proof. *Linear algebra magic.* Note that

$$p(t,x) := \prod_{i=1}^{n}(1+tx_i) = \sum_{i=0}^{n} t^i e_{n,i},$$

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

$$\sum_i a_i p(\alpha, x) = \sum_i a_i \langle v_i, e\rangle = \langle \sum_i a_i v_i, e\rangle = \langle w_d, e\rangle = e_d.$$

The LHS is an algebraic circuit of size cn^2.

There remains to exhibit such field elements. We can simply pick $\alpha_i = i, i \in [n+1]$, and the vectors v_i are linearly independent (Fact A.12). 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.13). **QED**

We shall see in Claim 14.1 a different approach where the circuit is somewhat larger but has additional structure.

14.8 Impossibility results for small-depth circuits

In this section we prove impossibility results for small-depth algebraic circuits. For simplicity, we only prove such results for circuits of depth 3, and over fields $\mathbb{F}$ of characteristic zero such as $\mathbb{R}$. The argument contains most of the ideas that go into extending the result to higher depth and other fields. The main result we prove is:

Theorem 14.9. There are explicit polynomials over $\mathbb{R}$ of degree $d := c \log n$ in n variables that require depth-3 arithmetic circuits of size $\geq n^{c\sqrt{d}}$.

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 an explicit set-multilinear polynomial for which it is large.

3. We show that the measure is small for efficient set-multilinear circuits.

We now develop each step in turn.

For concreteness, we allow multiplication by arbitrary constants along wires. So for example, if two gates compute polynomials p and q, a Σ gate can compute $ap + bq$ for any $a, b \in \mathbb{F}$, and a Π gate can compute apq for any $a \in \mathbb{F}$.

14.8.1 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 14.2. 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 needs 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 14.3. 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). The second reduces the fan-in of the multiplication gates. The third gets sm.

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 14.1. Suppose a $\Sigma\Pi\Sigma$ circuit of size s compute polynomial p. Then there is a $\Sigma\Pi\Sigma\Pi\Sigma$ homogeneous circuit of size $d^d s^c$ 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.16 below.

Proof. Consider one product gate q. We show how to compute $q^{(k)}$ 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. Assuming $b_j = 1$, we have

$$q = \prod_{i \le s} (\ell_i + 1) = \sum_{S \subseteq [s]} \prod_{i \in S} \ell_i.$$

The k-homogeneous part of q is

$$q^{(k)} := \sum_{S \subseteq [s], |S| = k} \prod_{i \in S} \ell_i.$$

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 §14.7, computing it directly as in the RHS above would give size $\ge n^d$, which we cannot afford. The construction in the proof Theorem 14.8 has smaller size, but it not homogeneous.

We seek an alternative efficient, homogeneous, $\Sigma\Pi\Sigma\Pi\Sigma$ circuit. Towards this, consider the power sum polynomials

$$p_{s,k} := \sum_{i \le s} x_i^k.$$

We have (suppressing the subscript s for simplicity)

$$\begin{aligned} e_1 &= p_1 \\ e_2 &= p_1^2/2 - p_2/2 \\ e_3 &= p_1^3 - 3p_1 p_2/2 + p_3 \\ &\ldots \end{aligned}$$

and so on, which convinces us that we can express the e_k via the p_k, and as it turns out this expression is more efficient.

More in detail, we have by A.16 that

$$k \cdot e_k = \sum_{i=1}^{k} (-1)^{i-1} e_{k-i} \cdot p_i,$$

for all k. Applying this repeatedly, we write e_k as a sum of $\le k^k$ products of $\le k$ polynomials p_i. Since each p_i is naturally a homogeneous $\sum \sum \prod$ 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 $\le k^k \cdot k \cdot s \cdot k \cdot n$, where each factor corresponds to the fan-in at a level.

Because of homogeneity, we only need to consider $k \le d$. Hence the total size is $\le d^d s^c$. **QED**

Exercise 14.10. Show that we can assume that $b_j = 1$.

Exercise 14.11. Explain why the construction in the proof of Theorem 14.8 is not homogeneous.

Claim 14.2. 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 $\le d$.

Proof. Replace all gates computing polynomials of degree $> d$ with the constant 0. The correctness of this step can be argued inductively. For Σ gate computing a polynomial of degree d, it is safe to set to 0 any summand with degree $> d$. This is because by homogeneity, the summands do not have monomials of degree $\le d$, so all their monomials must cancel in the output. For a Π gate the same holds, because multiplication increases degree (Fact A.15), 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 $\le d$ non-constant terms. The constant terms multiply to a constant that can be placed on a wire following our convention. **QED**

Finally, we get sm.

Claim 14.3. 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 $\le d$. 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 g is an input variable $x \in X_i$ we have $g_D = x$ if $D = \{i\}$ and $g_D = 0$ otherwise.

If g is a constant we have $g_\emptyset = g$ and $g_D = 0$ otherwise. (Alternatively, we can disallow constant but define more general addition and multiplication gates with constants in them.)

If

$$g = g_1 + g_2 + \cdots + g_i$$

then simply

$$g_D = g_{1,D} + g_{2,D} + \cdots + g_{i,D}$$

for any D.

Finally, if

$$g = g_1 \cdot g_2 \cdot \cdots \cdot g_i$$

note that

$$g_D = \sum g_{1,D_1} g_{2,D_2} \cdots g_{i,D_i}$$

where the sum is over all partitions $(D_1, \ldots, D_i)$ of D. Now we use the assumption that the fan-in i of this multiplication is $\leq d$. Hence the number of such partitions is at most the number of partitions of $[d]$ into d elements, which is $\leq d^d$.

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 14.2. 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 $\emptyset$.

Note how non-multilinear terms such as x_0^2 are removed during the process.

14.8.2 Step 2

For this step we further partition the blocks $[d]$ in two. We consider blocks with index $i < t$, denoted T_1 and those with $i \geq t$ denoted T_2, for a threshold t that will be set later.

Definition 14.3. 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 monomials with variables X_i for $i \in D \cap T_1$ and the columns by 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 $m_1 m_2$ in p.

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.

The complexity measure $\mu(p)$ is the rank of M_p normalized by the geometric mean of the two sides:

$$\mu(p) := \frac{\operatorname{rank}(M_p)}{\sqrt{\prod_{i \in D \cap T_1} |X_i| \cdot \prod_{i \in D \cap T_2} |X_i|}}.$$

Note that 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 14.3. 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 will have large μ. To construct such polynomials p, we can set the dimensions to be equal:

$$\prod_{i \in T_1} |X_i| = \prod_{i \in T_2} |X_i| \tag{14.2}$$

so that M_p is square; and then pick p so that M_p has full rank, for example it is diagonal or permutation. For such a p we obtain

$$\mu(p) = \frac{\prod_{i\in T_1}|X_i|}{\sqrt{\prod_{i\in T_1}|X_i| \cdot \prod_{i\in T_2}|X_i|}} = \frac{\prod_{i\in T_1}|X_i|}{\prod_{i\in T_1}|X_i|} = 1.$$

This value of μ is maximum as also given by the next lemma which will be used in the next step.

Lemma 14.4. The measure μ enjoys:

1. $\mu(p) \le 1$
2. [Sub-additivity] If p and q are sm w.r.t. the same D, $\mu(p+q) \le \mu(p) + \mu(q)$.
3. [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 14.12. Prove this. For 3., use Fact A.14.

14.8.3 Step 3

We now need to pick the sizes of the X_i so that μ will be small for the efficient circuit, while at the same time keeping equation (14.2). We let X_i with $i \in T_1$ have size m and the others have size $m^{1-\delta}$ where $\delta := 1/(2\sqrt{d})$. The total number of variables is $n = t\cdot m + (d-t)\cdot m^{1-\delta}$, and so $m \ge n^c$.

The parameters m and t are picked so that equation (14.2) is true. This requires

$$m^t = m^{(1-\delta)(d-t)} \iff t = (1-\delta)d/(2-\delta) \iff t = d - d/(2-\delta).$$

We now proceed to show that for such parameters, μ is small.

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 $\le d$ and each f_i is a $\Sigma\Pi\Sigma$ circuit.

We will show that

$$\mu(q) \le 1/m^{c\sqrt{d}} \tag{14.3}$$

from which we conclude the proof of Theorem 14.9 below. To prove equation (14.3) we consider two cases.

Some f_i has degree $\ge c\sqrt{d}$.

Exercise 14.13. Prove this case. Guideline: Let $C_{i,j}$ be a $\prod\sum$ sub-circuit of f_i; prove:

(1) Wlog $C_{i,j}$ has the same degree as f_i (which is $\ge c\sqrt{d}$).

(2) $\mu(C_{i,j}) \le 1/m^{c\sqrt{d}}$. Use Lemma 14.4 and Example 14.3.

(3) $\mu(f_i) \le s/m^{c\sqrt{d}} \le 1/m^{c\sqrt{d}}$.

(4) $\mu(q) \le 1/m^{c\sqrt{d}}$ (equation (14.3)).

Every f_i has degree $\le c\sqrt{d}$. This is where we use the unbalance. At the high-level, the idea is simple. Recall the target matrix is square, and has measure 1, which is the maximum. In the case 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. For perhaps the simplest example of the phenomenon, suppose you have many domino pieces of size p and many of size q, where p and q are primes. By picking q of the former and p of the latter, you can make a $pq \times pq$ square. But, if you use *few* pieces, say $< q$ pieces of length p, you'll never get a square. Returning to the proof, 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

$$\frac{1}{m^{0.5|a_i - b_i(1-\delta)|}}.$$

We now argue that this is small, using that a_i and b_i are integers summing to $\le \sqrt{d}$. The quantity of interest is the distance

$$|a_i - (1-\delta)b_i|,$$

which we need to bound below.

Generally, we claim that if a, b are integers in $[c/\epsilon]$ then $|a - (1-\epsilon)b| \ge c\epsilon(a+b)$.

Exercise 14.14. Prove this.

Using this claim we get

$$\mu(f_i) \le \frac{1}{m^{c(a_i+b_i)/\sqrt{d}}}.$$

By the multiplicativity property from Lemma 14.4 we have

$$\mu(q) \le \prod_i \frac{1}{m^{c(a_i+b_i)/\sqrt{d}}} \le \frac{1}{m^{cd/\sqrt{d}}} \le \frac{1}{m^{c\sqrt{d}}},$$

as desired.

14.8.4 Putting the steps together, proof of Theorem 14.9

By Lemma 14.3, equation (14.3), and the sub-additivity property in Lemma 14.4, the measure of the circuit is

$$\le d^{cd} s^c / m^{c\sqrt{d}}.$$

As remarked earlier, $m \ge n^c$ and so the denominator is $\ge n^{c\sqrt{\log n}}$. This is bigger than the numerator and so the ratio will be < 1.

Picking a target polynomial as in Step 2 for which the measure is 1 completes the proof.

14.9 Algebraic TMs

TBD

14.10 Problems

Problem 14.1. Execute the idea in Exercise 14.4.

Problem 14.2. Prove that Theorem 14.8 is false over $\mathbb{F}_2$.

14.11 Notes

The complexity of linear transformations was studied independently in [109, 264].

Theorem 14.2 is from [88]. For depth 2 they prove a slightly stronger, tight bound; they also show that depth $\log^* n$ suffices for a linear number of wires.

For the connection to factoring see [230, 174].

Impossibility results for univariate polynomials were first studied in [250]. That paper establishes negative results for polynomials with very large coefficients, but the results are non-trivial since arbitrary constants can be used by the circuit. For a survey see the book [53].

For an introduction to multivariate algebraic complexity see [53] and the survey [237]. 14.1 is from [33]. 14.2 is from [249]. For section 14.4 see [110].

Theorem 14.5 is from [160, 52].

For Theorem 14.8 and related constructions, also involving some of the ideas that go into Claim 14.1, see [236]. For more on the power of algebraic AC see [17].

Theorem 14.9 in section §14.8 is from [170]. It builds on a long line of works, see [171] for discussion. The complexity measure originates in [202]. The transformation to sm in the proof of Lemma 14.3 only works for large characteristic, due to the factors arising in Fact A.16 and related steps. [77] proves the lemma over any field, although whether circuits can be made homogeneous over any field remains open, cf. [77] for more discussion.

Theorem 14.7 is the culmination of a line of works about depth reduction that was rekindled by [6]. See discussion of subsequent work in [111] for this statement for depth 3. A formulation for any depth is stated in [171].

A general depth reduction theorem was first established in [135], but the size bounds were not controlled. Theorem 14.6, where the size is simultaneously controlled, is from [265] (see discussion in [265] for more on the history).

Chapter 15

Data 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 data structures

Here we have n bits of input data 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. In general, these words can be read adaptively. But for simplicity we focus on the case in which the locations are fixed by the data structure and the same for every input $x \in [2]^n$. Refer to figure 15.1.

Definition 15.1. A static data-structure problem is simply 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 output bit coordinate of h depends on $\leq t$ input words.

We say *word-space* to emphasize s refers to the number of words, not bits. The *bit-space* of the structure is sw. Sometimes the queries and or the word size are boolean. Another typical setting is $q = 2^w$.

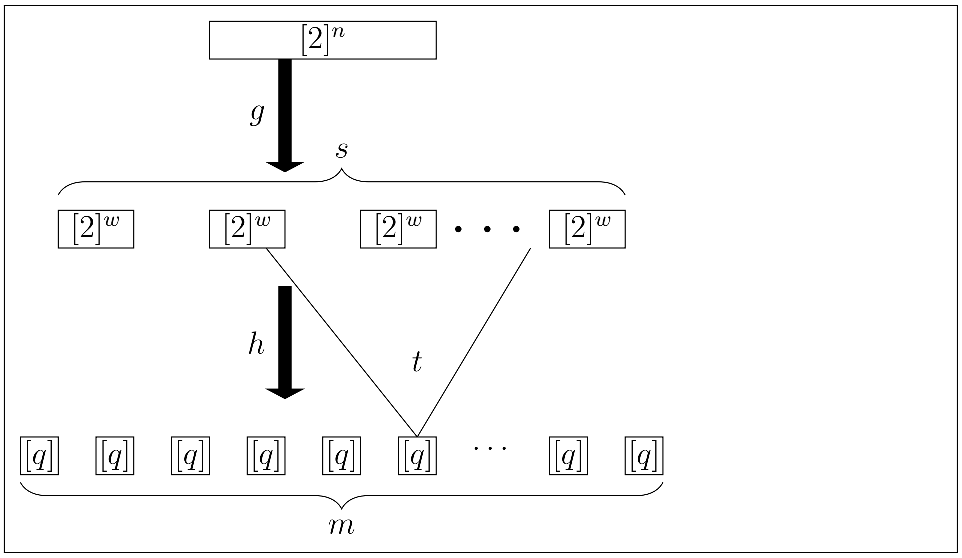


Figure 15.1: The static data-structure problem.

Exercise 15.1. Consider the data structure problem $f : [2]^n \to [2]^m$ where $m = n^2$ 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 $s = n$, $w = 1$, and $t = 2$.

Exercise 15.2. Show that any data-structure problem $f : [2]^n \to [2]^m$ has a data structure with $w = 1$ and the following parameters:

(1) $s = m$ and $t = 1$, and

(2) $s = n$ and $t = n$.

Exercise 15.3. Prove that for every n and $m \le 2^{n/2}$ there exist functions $f : [2]^n \to [2]^m$ s.t. any data structure with space $s = m/2$ and $w = 1$ requires time $t \ge n - c$.

By contrast, next we present the the best known impossibility result. To set the stage, we first show how to establish slightly weaker impossibility results via a calculation-free reduction to communication protocols (cf Chapter 13).

Exercise 15.4. Let f be a data-structure problem as in Definition 15.1. Show that if a f has a data structure with parameter names as in Definition 15.1 then the function $g : [2]^n \times [2]^{\log m} \to [q]$ defined as $g(x, i) := f(x)_i$ has communication complexity $\le ct(\log s + w)$.

Focusing on the natural setting where $\log m \le n$, we have that if g requires maximum communication complexity $\ge c \log m$ then

$$t \ge c\frac{\log m}{\log s + w}.$$

This bound is meaningful when m is large; but in typical settings where $n \le s \le m \le n^c$ it gives nothing.

The next result gives a refined bound where the term $\log s$ in the denominator is replaced with $\log s/n$. This makes for a meaningful bound as long as s is close to n. Settings such as $s = n^2$ are *terra incognita*. The refined bound is established for bounded-uniformity, cf. section 11.1.1 and Definition 11.3.

Definition 15.2. A function $f : [2]^n \to [q]^m$ is d-wise uniform, or simply d-uniform, if any d output coordinates are uniform when the input to f is uniform.

Theorem 15.1. Let $f : [2]^n \to [q]^q$ be d-wise uniform. Let q be a power of 2 and $c \log q \le d$. Then any data structure with $w = \log q$ using word-space s and time t has:

$$t \ge c \frac{\log q}{\log(s/d)}.$$

Let's make sense of the many parameters. Note that such d-wise uniform functions are computable with $n = d \log q = dw$ input bits, interpreting the input as coefficients of a degree $d-1$ univariate polynomial over $\mathbb{F}_q$ and outputting its evaluations, as in the proof of Theorem 11.1. Moreover, this is trivially the smallest possible n. Also, let us set $m = q$ so that $m = q = 2^w$. Plugging these parameters, we obtain for any n and m (powers of 2) an explicit function $f : [2]^n \to [m]^m$ with the data-structure bound

$$t \ge c \frac{\log m}{\log(sw/n)},$$

valid as long as $c \log q \le d \iff cw \le n/w$. A typical setting is $w = 10 \log n$ giving $m = q = n^{10}$. Recall that sw is the bit-space. It follows that if the bit-space is linear in n then $t \ge c \log m$. This result remains non-trivial for s slightly super-linear. But remarkably, if $sw = n^{1+c}$ then nothing is known (for m power in n one only gets $t \ge c$).

Proof. The idea in the proof, known as *cell sampling,* is to find a subset S of less than d memory cells that still allows us to answer $\ge d$ queries. This is impossible, since we can't generate d uniform outputs from less than d memory cells.

Let $p := 1/q^{1/4t}$. Include each memory cell in S with probability p, independently. By Theorem 2.1, $\mathbb{P}[|S| \ge cps] \le 2^{-cps}$.

We are still able to answer a query if all of its queries fall in S. The probability that this happens is $\epsilon := p^t = 1/q^{1/4}$. We now claim that with probability $\ge 0.5\epsilon$, we can still answer $0.5\epsilon q \ge \sqrt{q}$ queries. Indeed, let B be the number of queries we cannot answer. We have $\mathbb{E}[|B|] \le q(1-\epsilon)$. And so

$$\mathbb{P}[|B| \ge q(1-0.5\epsilon)] \le \frac{1-\epsilon}{1-0.5\epsilon} \le 1 - 0.5\epsilon.$$

Thus, if the inequality $2^{-cps} < 0.5\epsilon = 1/q^c$ holds then there exists a set S of cps cells with which we can answer $\ge \sqrt{q} > d$ queries. Since the function is d-uniform, answering the latter queries requires $\ge dw$ input bits, and so $Sw \ge dw \iff S \ge d \iff cps \ge d$.

Therefore we reach a contradiction if the following chain of inequalities is true:

$$c \log q \leq cps < d.$$

Because $d > c \log q$ by assumption, we can say that the first inequality is automatically satisfied. Specifically, we can set s to be the largest s.t. $cps < d$, and the first inequality is satisfied. We reach a contradiction if $cps < d$, and the result follows. **QED**

Surprisingly, this result is nearly tight. There are many parameters floating around, so "tight" should be qualified. But basically, there are bounded-uniform functions f, with nearly optimal parameters, which have data structures with a tradeoff matching Theorem 15.1. One can in fact achieve results in the same spirit for f with optimal parameters (i.e., polynomial evaluation), thus matching the function in Theorem 15.1, but this requires a different construction, and I don't know if constant time is known.

Theorem 15.2. *Let $q = n^{10} = 2^w$. There is $f : [2]^n \to [q]^q$ which is d-wise uniform and has a data structure with w-bit words, time t, word-space $s = cdw \cdot m^{1/t}$, and $n = ctdw$.*

Let us illustrate parameters. Both n and sw (regardless of time) are $\geq dw$, for else you can't generate d uniform coordinates. So there is a d-wise uniform f with nearly optimal n that has a data structure achieving space optimal up to a factor $cm^{1/t}$, and time t. For example, we can have $n = cdw$, word space $dw \cdot q^{0.01}$, and $t = c$. Whereas you could have thought that word space close to q would be necessary. Qualitatively, the space approaches optimal *exponentially* fast with the time t, the same behavior exhibited in the Theorem 15.1. For a proof see Problem 15.1.

15.1.1 Succinct data structures

Succinct data structures are those where the space is close to the minimum, n. Specifically, we let $s = n + r$ for some $r = o(n)$ called *redundancy.* Unsurprisingly, stronger bounds can be proved in this setting. But, surprisingly, again these stronger bounds were shown to be tight. Moreover, it was shown that improving the bounds would imply stronger circuit lower bounds.

To illustrate, consider error-correcting codes, cf Exercise 2.12.

Theorem 15.3. *Any data-structure for an a-good code $f : [2]^n \to [2]^m$ with $w = 1$ using space $n + r$ requires time $\geq c_a n/r$.*

This is nearly matched by the following result. In fact, later we will prove the stronger result that *dynamic* data structures exist for error-correcting codes.

Theorem 15.4. *There is a good code $f : [2]^n \to [2]^m$ that for any r has a data structure with $w = 1$, space $n + r$, and time $c(n/r) \log^3 n$.*

Exercise 15.5. Prove this using the code in Theorem 14.2. Hint: If a circuit has ℓ wires, there are $\leq r$ gates with fan-in $> \ell/r$. That's your redundancy.

The techniques in the hint apply generically. They imply that proving a time lower bound of $(n/r)\log^c n$ would imply new circuit lower bounds, for circuits with XOR gates only, or for circuits with arbitrary gates. For the boolean model, the following connection is also known.

Theorem 15.5. Let $f : [2]^n \to [2]^{am}$ be a function computable by bounded fan-in circuits with bm wires and depth $b \log m$, for constants a, b. Then f has a data structure with space $n + o(n)$ and time $n^{o(1)}$.

Hence, proving n^ϵ time lower bounds for succinct data structures would give functions that cannot be computed by linear-size log-depth circuits, cf. 8.1.2.

15.1.2 Succincter: The trits problem

In this section we present a cute and fundamental data-structure problem with a shocking and counterintuitive solution. The trits problem is to compute $f : [3]^n \to ([2]^2)^n$ where on input n "trits" (i.e., ternary elements) $(t_1, t_2, \ldots, t_n) \in [3]^n$ f outputs their representations using two bits per trit.

Example 15.1. For $n = 1$, we have $f(0) = 00, f(1) = 01, f(2) = 10$.

Note that the input ranges over 3^n elements, and so the minimum space of the data structure is $s = \lceil \log_2 3^n \rceil = \lceil n \log_2 3 \rceil \approx n \cdot 1.584\ldots$ This will be our benchmark for space. One can encode the input to f as before using bits without loss of generality, but the current choice simplifies the exposition.

Simple solutions

- The simplest solution is to use 2 bits per t_i. With such an encoding we can retrieve each $t_i \in [3]$ by reading just 2 bits (which is optimal). The space used is $s = 2n$ and we have linear redundancy.

- Another solution, which we basically already mentioned, is what is called *arithmetic coding*: we think of the concatenated elements as forming a ternary number between 0 and $3^n - 1$, and we write down its binary representation. To retrieve t_i it seems we need to read all the input bits, but the space needed is optimal.

- For this and other problems, we can trade between these two extreme as follows. Group the t_i's into blocks of t. Encode each block with arithmetic coding. The retrieval time will be ct bits and the needed space will be

$$(n/t)\lceil t \log_2 3 \rceil \le n \log_2 3 + n/t$$

(assuming t divides n). In other words, *block-wise arithmetic coding.* This provides a *power* trade-off between time and redundancy, but no more (see the notes).

The shocking solution: An exponential trade-off

We now present an *exponential* trade-off: retrieval time ct bits and redundancy $n/2^t + c$. In particular, if we set $t = c\log n$, we get retrieval time $c\log n$ and redundancy c. Moreover, the bits read are all consecutive, so with word size $w = \log n$ this can be implemented in constant time. To repeat, *we can encode the trits with constant redundancy and retrieve each in constant time.* This solution can also be made dynamic.

Theorem 15.6. The trits problem has a data structure with space $n\log_2 3 + n/2^t + c$ (i.e., redundancy $n/2^t + c$) and time ct, for any t and with word size $w = 1$. For word size $w = \log n$ the time is constant.

Next we present the proof.

Definition 15.3. [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 $\log_2 |B| - \log_2 |A|$.

The following lemma gives the building-block encoding we will use.

Lemma 15.1. 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 $\le c/\sqrt{|Y|}$ and s.t. $x \in X$ can be recovered just by reading the b bits in $f(x, y)$.

Exercise 15.6. Prove $K \le cY$.

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:

$$X \times Y \to X \times C \times K \to [2]^b \times K.$$

There is however a subtle point. If we insist on always having $|C|$ equal to, say, $\sqrt{|Y|}$ or some other quantity, then one can cook up sets that make us waste a lot (i.e., almost one bit) of space. The same of course happens in the more basic approach that just sets $Y = K$ and encodes all of X with b bits. The main idea will be to "reason backwards," i.e., we will first pick b and then try to stuff as much as possible inside $[2]^b$. Still, 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 $c\sqrt{Y} \le d \le c\sqrt{Y}$.

How much can we stuff into B? For a set C of size $|C| = \lfloor B/X \rfloor$, we can encode elements from $X \times C$ in B. The redundancy of such an encoding can be bounded as follows:

$$\begin{aligned}
&\log B - \log X - \log C = \\
=&\log \frac{2^b}{X} - \log \left\lfloor \frac{2^b}{X} \right\rfloor \\
=&\log d - \log\lfloor d \rfloor \\
\leq&\log d - \log(d-1) \\
=&\log\left(1 + \frac{1}{d-1}\right) \\
\leq&\frac{c}{d-1} \\
\leq&\frac{c}{\sqrt{Y}-1} \\
\leq&\frac{c}{\sqrt{Y}}.
\end{aligned}$$

To calculate the total redundancy, we still need to examine the encoding from Y to $C \times K$. Choose K of size $|K| = \lceil Y/C \rceil$, so that this encoding is possible. With a calculation similar to the previous one, we see that the redundancy is:

$$\begin{aligned}
&\log C + \log K - \log Y \\
=&\log \left\lceil \frac{Y}{C} \right\rceil - \log \frac{Y}{C} \\
\leq&\log\left(1 + \frac{C}{Y}\right) \\
\leq&c\frac{C}{Y} \\
\leq&c\frac{\left\lfloor \frac{2^b}{X} \right\rfloor}{Y} \\
\leq&c\frac{2^b}{X \cdot Y} \\
\leq&\frac{c2^{\log\left(X \cdot \sqrt{Y}\right)}}{X \cdot Y} \\
\leq&c\frac{\sqrt{Y}}{Y} \\
=&c\frac{1}{\sqrt{Y}}.
\end{aligned}$$

The total redundancy is then $c/\sqrt{|Y|}$. By construction, $x \in X$ can be recovered from the element of B only. **QED**

Proof of Theorem 15.6. Break the ternary elements into blocks of size t: $(t'_1, t'_2, \ldots, t'_{n/t}) \in T_1 \times T_2 \times \ldots \times T_{n/t}$, where $T_i = [3]^t$ for all i. The encoding, illustrated in figure 15.2, is constructed as follows, where we use f_L to refer to the encoding guaranteed by Lemma 15.1.

Compute $f_L(t'_1, t'_2) = (b_1, k_1) \in B_1 \times K_1$.

For $i = 2, \ldots, n/t - 1$ compute $f_L(k_{i-1}, t'_{i+1}) := (b_i, k_i) \in B_i \times K_i$.

Encode $k_{n/t-1}$ in binary as $b_{n/t}$ using arithmetic coding.

The final encoding is $(b_1, b_2, \ldots, b_{n/t})$. We now compute the redundancy and retrieval time. One can visualize this as a "hybrid argument" transforming a product of blocks of ternary elements into a product of blocks of binary elements, one block at the time.

Redundancy: From (1) in Lemma 15.1, the first $n/t - 1$ encodings have redundancy $c3^{-t/2} \le 1/2^{ct}$. For the last (arithmetic) encoding, the redundancy is at most 1. So the total redundancy is at most $\left(\frac{n}{t} - 1\right) \cdot \frac{1}{2^{ct}} + 1 = \frac{n}{2^{ct}} + c$.

Retrieval Time: Say that we want to recover some t_j which is in block t'_i. To recover block t'_i, Lemma 15.1 guarantees that we only need to read at b_{i-1} and b_i. This is because k_{i-1} can be recovered by reading only b_i, and t'_i can be recovered by reading k_{i-1} and b_{i-1}. Thus to complete the proof it suffices to show that each b_i has length ct.

This is not completely obvious because one might have thought that the K_i become larger and larger, and so we apply the lemma to larger and larger inputs and the B_i get large too. However, recall that each $K_i \le cT_i = c3^t$ from Exercise 15.6. Hence, every time we apply the lemma on an input of size at most $s \le 3^{ct}$. Since the encoding in Lemma 15.1 has small redundancy, none of its outputs can be much larger than its input, and so $B_i = 2^{ct}$. **QED**

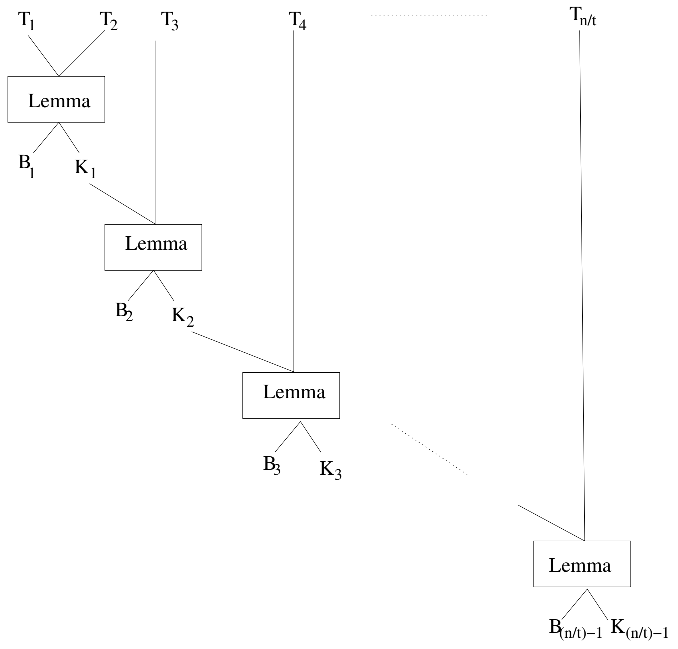


Figure 15.2: Succinct Encoding

15.2 Dynamic data structures

We now study dynamic data structures. As we mentioned, here the input is not fixed but can be modified by the queries. For concreteness, we focus on the specific problem of maintaining a codeword.

Definition 15.4. A dynamic data-structure for a code $f : [2]^n \to [2]^m$ supports two types of operations, starting with the all-zero message $x \in [2]^n$:

$M(i, b)$ for $i \in \{1, 2, \ldots, n\}$ and $b \in [2]$ which sets bit i of the message to b, and

$C(i)$ for $i \in \{1, 2, \ldots, m\}$ which returns bit i of the codeword corresponding to the current message.

The time of a dynamic data structure is the maximum number of operations in memory cells required to support an operation. This is similar to the cell-probe RAM (Definition 1.8) where we ignore details and solely focus on information transfer among cells. Recall for RAMs and *a fortiori* cell-probe RAMs we do not know how to prove impossibility results. This can be considered as an *offline* mode of computation. By contrast the setting of data structure is *online* in that we have to answer queries as they arrive. In this setting it is natural to ask for small running times, like constant, and we can prove non-trivial impossibility results.

Theorem 15.7. Dynamic data-structures for any good code (cf Exercise 2.12) take time $t \ge c \log_w n \ge (c \log n)/\log\log n$ for cell size $w := \log n$.

One might wonder if stronger bounds can be shown. But in fact there exist good codes for which the bounds are nearly tight.

Theorem 15.8. There are dynamic data structures for good codes running in time $c \log^2 n$ with cell size $w = 1$.

Exercise 15.7. Prove Theorem 15.8 using the code in Theorem 14.2. Hint: The data structure is the middle layer. Bound the fan-out of the input gates in the proof of Theorem 14.2.

Proof of Theorem 15.7. Pick $x \in [2]^n$ uniformly and $i \in \{1, 2, \ldots, m\}$ uniformly, and consider the sequence of operations

$$M(1, x_1), M(2, x_2), \ldots, M(n, x_n), C(i).$$

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 which we denote by $C_x := (C_x(1), C_x(2), \ldots, C_x(n)) \in [2]^m$.

We divide the n operations $M(i, x_i)$ into consecutive blocks, called *epochs*. Epoch e consists of n/w^{3e} operations. Hence we can have at least $E := c \log_w n$ 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 cells written by the data structure in future epochs.

A key idea of the proof is to see what happens when the cells written during a certain epoch are ignored, or reverted to their contents right before the epoch. Specifically, after the execution of the M operations, we can associate to each memory cell the last epoch during which this cell was written. Let $D^e(x)$ denote the memory cells of the data structure after the first n operations M, but with the change that the cells that were written last during epoch e are replaced with their contents right before epoch e. Define $C^e_x(i)$ to be the result of the data structure algorithm for $C(i)$ on $D^e(x)$, 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 $C(i)$, executed after the first n operations M, reads a cell that was last written in epoch e, and 0 otherwise. We have

$$t \geq \max_{x,i} \sum_e t(x,i,e) \geq \mathbb{E}_{x,i} \sum_e t(x,i,e) = \sum_e \mathbb{E}_{x,i} t(x,i,e) \geq \sum_e \mathbb{E}_x \Delta(C_x, C^e_x), \qquad (15.1)$$

where the last inequality holds because $C^e_x(i) \neq C_x(i)$ implies $t(x, i, e) \geq 1$.

We now claim that if $t \leq w$ then $\mathbb{E}_x\Delta(C_x, C^e_x) \geq c$ for every e. This concludes the proof.

In the remainder we justify 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 at least

$$2^{n/w^{3e}}$$

codewords. On the other hand, we claim that C^e_x ranges over much fewer strings. Indeed, the total number of cells written in all epochs after e is at most

$$t \sum_{i \geq e+1} n/w^{3i} \leq ctn/w^{3(e+1)}.$$

We can describe all these cells by writing down their indices and contents using $B := ctn/w^{3e+2}$ bits. Note that this information can depend on the operations performed during Epoch e, but the point is that it takes few possible values overall. Since the cells last changed during Epoch e are reverted to their contents before Epoch e, this information suffices to describe $D^e(x)$, and hence C^e_x. Therefore, C^e_x ranges over $\leq 2^B$ strings.

For each string in the range of C^e_x at most two codewords can have relative distance $\leq c$, for else you'd have two codewords at distance $\leq 2c$, violating the distance of the code.

Hence except with probability $2 \cdot 2^B/2^{n/w^{3e}}$ over x, we have $\Delta(C_x, C^e_x) \geq c$. If $t_M \leq w$ then the first probability is ≤ 0.1, and so $\mathbb{E}_x\Delta(C_x, C^e_x) \geq c$, proving the claim. **QED**

Exercise 15.8. Explain how to conclude the proof given the claim.

15.3 Problems

Problem 15.1. Prove Theorem 15.2. Hint: Use the construction Theorem 11.4. How uniform needs the input be?

Problem 15.2. In this problem you will show that proving impossibility results for dynamic data structures is easier (or no harder) than proving them for static data structures.

Let $f : [2]^n \to [2]^m$ be a static data structure problem. Give a set of $n + m$ queries s.t. if a dynamic data structure can support them in time t then f has a data structure with space $s \le tn$ and time t. Feel free to assume word size $w = 1$ and non-adaptive query algorithms throughout (though these cases are not really satisfying, since the power of dynamic data structure relies on larger word size and adaptivity – but one can prove a suitable extension for more general cases). (A non-adaptive query algorithm means that the memory locations accessed depend only on the query, and not on the values of previous queries.)

15.4 Notes

The connection between data structures and communication complexity is from [186] and was studied more in [187]. Theorem 15.1 is from [238]. It was rediscovered in [166]. Efficient data structure for polynomial evaluation (complementing Theorem 15.2) are in [153].

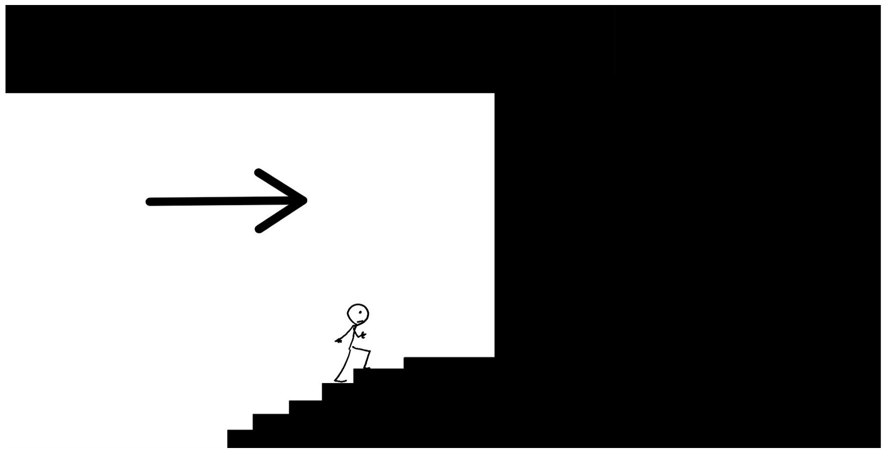
Theorem 15.3: [89]. Efficient data structures for ECC, both static in Theorem 15.4 and dynamic in Theorem 15.8, as well as the connection to circuits, are from [282].

The breakthrough result on the trits problem is from [209]. After that a negative result was proved in [279], whose parameters were then matched by Theorem 15.6, proved in [209, 70]. Our exposition is from [274]. Using number-theoretic results on logarithmic forms, it is shown in [279] that block arithmetic coding does not do better than a power tradeoff.

For dynamic data structures, the technique in the proof of Theorem 15.7 is from [84] and has been applied to many other natural problems. It is not far from the state-of-the art in this area, which is $\log^{1+c} n$ [167].

Chapter 16

Barriers



In an attempt to understand the Grand Challenge (Chapter 3), one can identify several proof techniques and show or speculate that they cannot solve it. Such arguments are known as "barriers." Several barriers have been put forth, and in fact 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.

The two main barriers are the *black-box* (a.k.a. *oracle* or *relativization* barrier) and the *natural proofs* barrier.

16.1 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. How to make this more precise? When programming, we can think of having access to a powerful *library*, a.k.a. *subroutine*, *oracle*, *black box*, etc. If our argument also applies when given access to *any* black box 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.

The precise model is the same we encountered in Chapter 4, see Definition 4.2. We equip our models, such as TMs, with access to a function $f : [2]^* \to [2]$, known as *oracle*. At any point, the model can ask for the value of the function at an input that has been computed. In the case of TMs, we can think of a special oracle tape and a special oracle state. Upon entering the state, the contents $x \in [2]^*$ on the oracle tape are replaced in one step with $f(x) \in [2]$. We can then define corresponding complexity classes, denoted P^f, and so on.

To illustrate, consider the separation $\mathrm{P} \neq \mathrm{Exp}$, which follows from the Time Hierarchy Theorem 3.3. The result relativizes:

Theorem 16.1. For every oracle $f : [2]^* \to [2]$, $\mathrm{P}^f \neq \mathrm{Exp}^f$ (i.e., $\mathrm{P} \neq \mathrm{Exp}$ relativizes).

Proof. Diagonalization and the time hierarchy work just as well for oracle machines. Specifically, when simulating a machine M running in time t with a machine M' running time $t' > t$, the simulation proceeds as before, and if M queries the oracle then M' does that as well. **QED**

Exercise 16.1. Prove $\mathrm{PSpace}^f \subseteq \mathrm{Exp}^f$ for every oracle (i.e., $\mathrm{PSpace} \subseteq \mathrm{Exp}$ relativizes).

Next we argue that relativizing techniques cannot resolve other major questions. Perhaps the simplest example is for P vs. PSpace, because the way the oracle is accessed is clear. We show that there are oracles w.r.t. which the P vs. PSpace question can be resolved either way, so one cannot resolve in a way that extends to all oracles, as in Theorem 16.1.

Theorem 16.2. 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. For example, let f take as input $(M, 1^s, x)$, simulate M using space s on input x for $\leq |M|^s$ steps, and return its answer. If M exceeds space s, the oracle returns 0. We claim that $\mathrm{PSpace}^f = \mathrm{PSpace}$. This is just because the oracle queries can be answered by direct simulation using power space. Further, $\mathrm{PSpace} \subseteq \mathrm{P}^f$, because an algorithm on the rhs can query f on the input corresponding to an algorithm on the lhs. The proof is completed by combining the two claims.

The construction of g is more involved. To illustrate the main idea, let us first assume that oracle machines, 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 machines. 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 doesn't stop, force stop and output, say, 0). 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.

That's the main idea. Now, for completeness, we drop the assumption that on inputs of length n only oracle queries of length n are made. The idea is just to pick sufficiently spaced-out inputs so that the above strategy can be executed again. We set values of the oracle one machine M_i at the time (whereas previously one could have processed all machines simultaneously). For each machine we set the values of the oracle at one more input length, so that on that input length the power-time oracle machine makes a mistake. In the generic iteration, we start with having an oracle s.t. H^g cannot be solved by machine M_j running in time n^j, for any $j < i$, and only the first c_i input lengths of the oracle are set. (The latter condition is w.l.o.g..) 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, consider an input length m 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 set the oracle as before. That is, run 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 inputs of length m, otherwise we set it to 1 on one of the queries of length m that wasn't queried by M_m on input 1^m. **QED**

Exercise 16.2. (1) Write down the definition of NP^f. (2) show that P vs. NP cannot be solved via black-box techniques, by following the proof of Theorem 16.2.

16.2 Natural proofs

The natural proofs barrier aims to explain the limit of *combinatorial* proof techniques. The idea is simple:

(1) Most combinatorial proof techniques against a class of functions F (for example, F are the functions on n bits computable by circuits of size n^{10} and depth $10 \log n$) do more than providing a separation: They yield an *efficient* algorithm that given the truth table of length 2^n of a function can distinguish tables coming from F from those coming from uniformly random functions.

(2) The classes F for which we would like to prove impossibility are believed to be powerful enough to compute *pseudorandom functions*, i.e., truth tables that *cannot* be efficiently distinguished from uniformly random functions.

Note that (2) is not known unconditionally, but just believed to be the case. This is where complexity theory gets quite philosophical. We can't really *prove* (2) without solving the Grand Challenge, in which case these barriers are not actual barriers. On the other hand one can have a *belief* that (2) is indeed true even though we can't prove it, and if that's the case indeed to solve the Grand Challenge one needs to somehow bypass (1) and find alternative techniques. We currently don't seem to have such techniques.

That's not all, however. In some cases we can bypass (2) and claim unconditionally that efficient techniques won't work. The idea is that if lower bounds are not true, we can't prove them; but if lower bounds are true, then we can use them to construct pseudorandom functions.

16.2.1 TMs

We illustrate the natural-proofs barrier for 1TMs. Let us revisit the information bottleneck technique from section §3.1 to show that from it we can extract an efficient algorithm to distinguish truth-tables computed by fast 1TMs from uniform functions. One is tempted to consider a test checking if there is a large set where the function is constant, and moreover X is a product set $X = Y \times Z$. However, it is not clear that this test would be efficient. It is more convenient to use the simulation of 1TMs by low-communication protocols, Theorem 13.6, and use the quantity R from section 13.2.2.

16.2.1.1 Telling subquadratic-time 1TMs from random

Given the truth-table of a function $f : [2]^n \to [2]$, our test D will consider 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 check if $R(f_0) \geq 2^{-cn}$.

First, let us verify that fast 1TMs indeed pass D. Let M be an s-state 1TM running in time t computing $f : [2]^n \to [2]$. By Theorem 13.6, f_0 has 2-party protocols with communication $d := c(\log s)t/n$ and error $\leq 1/2$. By Lemma 13.3, $R(f_0) \geq 2^{d/c} \geq s^{ct/n}$.

Second, let's verify that D is efficiently computable. Indeed, following the definition we can compute D in time 2^{cn}, which is power in the input length 2^n.

Third, and finally, we show that random functions $U : [2]^n \to [2]$ don't pass the test. Indeed, we have

$$\mathbb{E}_U[R(U)] = \mathbb{E}_U \mathbb{E} e_{\substack{x_1^0, x_2^0 \\ x_1^1, x_2^1}} [U(x_1^0, x_2^0) + U(x_1^0, x_2^1) + U(x_1^1, x_2^0) + U(x_1^1, x_2^1)].$$

When the x_1 are distinct and the x_2 are distinct, the expectation is 0. In the other case the expectation is 1, and this happens with prob. $\leq 2^{-cn}$. Therefore $\mathbb{E}_U[R(U)] \leq 2^{-cn}$. Moreover, R is never negative, for $R(f) = \mathbb{E}_{\substack{x_1^0 \\ x_1^1}} \left(\mathbb{E}_y e[f(x_1^0, y) + f(x_2^0, y)]\right)^2$. Hence

$$\mathbb{P}_U[R(U) \geq 2^{-cn}] \leq 2^{-cn}.$$

The upshot of all of the above is that we have devised an efficient test that can distinguish truth tables of functions computed by fast 1TMs from truth tables of uniformly random functions.

16.2.1.2 Quadratic-time 1TMs can compute pseudorandom functions

We now sketch a candidate pseudorandom function computable in quadratic time by 1TMs with cn^2 states. The candidate is an asymptotic generalization of a well-documented and

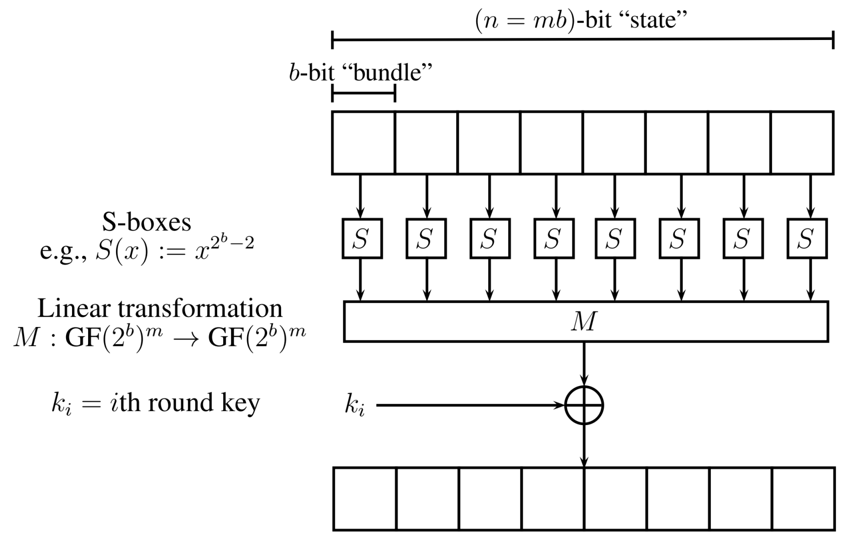


Figure 16.1: One round of an SPN

widely used 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 16.1.

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 $(0 \le i < 4)$ is shifted circularly to the left by i places. In the second step, a maximal-branch-number matrix $\phi \in \mathbb{F}^{4\times 4}$ 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 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 $\le c$ 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.

By the simulation of TMs by circuits, this candidate is also computable by power-size circuits. A naive implementation gives fairly large depth, so next we consider smaller-depth circuits.

16.2.2 Small-depth circuits

In section §8.5 we saw several impossibility results for AC. In the next exercise you are asked that at least one of the proof techniques we saw is natural.

Exercise 16.3. Give an efficient algorithm to distinguish truth tables of functions in AC from uniform. Hint: Use Exercise 8.19.

There are candidate pseudorandom functions computable in TC. Some of them are based on popular conjectures, such as the hardness of factoring, cf Theorem 14.3. The critical feature of TC that enables computing such candidates is iterated multiplication, see Theorem 8.8.

16.3 Notes

In an effort to make progress and understand the reach of current techniques, essentially every technique in complexity theory has been analyzed and, with few exceptions, filed under "black-box" or "natural-proofs." Especially for "black-box" this involved a myriad different oracle constructions. Often, these constructions are related to, and have provided some motivation for the study of, basic complexity classes. For example, PH^f basically corresponds to AC functions of the truth table of f, and one can use results about AC to give various oracle separations for PH and related classes. Indeed, a major motivation for impossibility results for AC was showing that the PH does not collapse w.r.t. some oracles [294]. After this first prolific phase of oracle constructions, starting about half a century ago, a second phase has followed during which it was realized that oracles provide limited information and they were relegated as curiosities. In a more recent third phase they have made a comeback in cryptography and quantum computing, often under the terminological disguise of *black-box*.

Relativization originated in the seminal work [30] and led to countless works on oracles. A variant of relativization where the oracles have additional algebraic structure is sharper for certain proof techniques and is studied in [2].

Natural proofs is from [217]. AES is described in [67]. The SPN structure of alternating "confusion" and "diffusion" steps was put forth already in [232]. The candidate 1TM pseudorandom function in section 16.2.1.2 is from [185].

The PRF in TC is from [193]. It gives TC of size $\geq n^c$. The work [13] considers TC of size $n^{1+\epsilon}$. [185] present a candidate, also based on AES, with these resources.

Chapter 17

I believe P=NP

"[...] 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 [...]." [94]

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 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 [150] 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" [248].

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 conjectured that super-concentrators must have a super-linear number of wires, from which super-linear circuit lower bounds follow [263]. However, he later disproved the conjectured [264]: building on a result of Pinsker 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 recently showed that, in fact, the Hadamard matrix is not rigid [14].

5. Constructing rigid matrices is one of *three* ways to get circuit lower bounds from a graph decomposition in [264]. Another way is via communication lower bounds. Here a specific candidate was the *sum-index* function, but then Sun [253] gave an efficient protocol for sum-index.

6. The LBA problems (Is L = NL? Is NL closed under complement?) A negative solution to the second problem obviously implies a negative solution to the first. A solution to the second problem was found after more than 20 years after the formulation [137, 254, 255]. 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, NL is closed under complement, cf Theorem 7.20.

7. the second problem was solved in the affirmative would imply a negative answer to t TBD

8. 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 [47]. Barrington [189] 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 9.1).

9. Mansour, Nisan, and Tiwari conjectured [180] in 1990 that computing hash functions on n bits requires circuit size $\Omega(n \log n)$. Their conjecture was disproved in 2008 [143] where a circuit of size $O(n)$ was given.

10. 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.

11. Maxflow is a central problem studied since the dawn of computer science. All solutions had running time $\geq n^{1+c}$, until a quasi-linear algorithm obtained in 2022.

12. 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 [25] and subsequent works, where many other counter-intuitive protocols are presented, see section §13.4. For pointer chasing, a similar bound was first claimed and then

retracted [69], then it was made again more recently (personal communication), only to be found in contradiction with the protocol in [212] (Theorem 13.12).

And these are just some of the more famous ones. The list goes on and on. In data structures, would you think it possible to switch between binary and ternary representation of a number using constant time per digit and *zero* space overhead? Turns out it is [209, 70] (see section 15.1.2). Do you believe factoring is hard? Then you also believe there are pseudorandom generators where each output bit depends only on c input bits [19], see section §9.5. Known algorithms for directed connectivity use either super-polynomial time or polynomial memory. But if you are given access to polynomial memory full of junk that you can't delete, then you can solve directed connectivity using only logarithmic (clean) memory and polynomial time [51], section §9.6. And I haven't even touched on the many broken conjectures in cryptography, most recently related to obfuscation.

On the other hand, arguably the main thing that's surprising in the lower bounds we have is that they can be proved at all. The bounds themselves 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 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 (same for the theory of 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.

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, and P=NP. Do I really believe the formal inclusion P=NP? Maybe, let me not pick parameters. What I believe is that 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. It's the upper bounds that are missing.

The "thousand different problems" argument for P $\neq$ NP

"The class NP [...] contains thousands of different problems for which no efficient solving procedure is known."[97]

"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." [134], Page 341.

I find these claims strange. In fact, the theory of NP completeness leads me to an opposite conclusion. As we saw, the problems can all be translated one into the other with extremely simple procedures, essentially doing *nothing*, just maybe complementing a bit. In what sense are they different? I think a good definition of different is that they are not known to be reducible to each other in a simple manner.

The "lots of people tried" argument for $\mathbf{P \neq NP}$

The conjectures above were made by n top scientists in the area. On the other hand, $N \gg n$ people outside of the area attempted and failed to solve NP-hard problems. The fact that they are outsiders can be a strength or a weakness for the argument. It can be a strength, because of the sheer number, and because unshackled by the trends of the community, and without much interaction, the N people have been free to explore radically new ideas:

"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."[97]

But it can also be a weakness, because unaware of the well-studied pitfalls, and with little communication, these N people 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. So it is likely that those N people don't quite count for N distinct attempts, but in fact a much smaller number, quite possibly less than n.

The "catastrophe" argument for $\mathbf{P \neq NP}$

It's easy to consider scenarios in which P = NP would not cause a catastrophe. A trivial scenario is if the algorithms take time n^d for exceedingly large d. A less obvious scenario is that the algorithms use complicated component X (think the classification of 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.

A typical instantiation of the catastrophe is that most cryptography collapses. Again, one can imagine a scenario where it doesn't collapse. For example, the attacks are complicated or impractical. People continue to publish papers and use the protocols regardless. The new result just gives a more nuanced view of security. 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.

My "stop right before major results" argument for $\mathbf{P = NP}$

Why do available techniques for impossibility results stop "right before" proving major results? This phenomenon appears to permeate complexity theory: we saw examples in section §7.3, Chapter 8, and Chapter 14. The most reasonable conclusion, it seems to me, is that this happens because the major results are actually false.

My "get stuck at the same point" argument for $\mathbf{P = NP}$

An issue related to the "stop right before major results" issue 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 can't compute parity, cf section §8.5. The proofs appear genuinely different, I would argue more different than the various NP-complete problems (see the "thousand different problems" argument above). Why should different approaches stop at the same point, except because there is nothing else to prove?

Throughout history, science has often proved wrong those who wouldn't take things at face value.

Complexity theory is perhaps unique in science. It appears that math is not ready for its problems. It is a bulwark against the business approach to science, the frenzy of the illusion of progress. For *ultimately* it doesn't matter how much you rake in or even who is writing bombastic recommendation letters for you, etc. These problems remain untouched. And progress may be more likely to come when you are alone, staring at blank paper:

> "You do not need to leave your room. Remain sitting at your table and listen. Do not even listen, simply wait. Do not even wait, be quiet still and solitary. The world will freely offer itself to you to be unmasked, it has no choice, it will roll in ecstasy at your feet." [147]

References

[1] Scott Aaronson. Oracles are subtle but not malicious. In *CCC*, pages 340–354. IEEE Computer Society, 2006.

[2] 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.

[3] 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.

[4] Anil Ada, Arkadev Chattopadhyay, Omar Fawzi, and Phuong Nguyen. The NOF multiparty communication complexity of composed functions. *Comput. Complex.*, 24(3):645–694, 2015.

[5] Leonard Adleman. Two theorems on random polynomial time. In *19th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 75–83. 1978.

[6] 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.

[7] Miklós Ajtai. Σ_1^1-formulae on finite structures. *Annals of Pure and Applied Logic*, 24(1):1–48, 1983.

[8] Miklós Ajtai. Approximate counting with uniform constant-depth circuits. In *Advances in computational complexity theory*, pages 1–20. Amer. Math. Soc., Providence, RI, 1993.

[9] Miklós Ajtai. A non-linear time lower bound for boolean branching programs. *Theory of Computing*, 1(1):149–176, 2005.

[10] Miklos Ajtai and Avi Wigderson. Deterministic simulation of probabilistic constant-depth circuits. *Advances in Computing Research - Randomness and Computation*, 5:199–223, 1989.

[11] 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.

[12] Eric Allender. The division breakthroughs. *Bulletin of the EATCS*, 74:61–77, 2001.

[13] Eric Allender and Michal Koucký. Amplifying lower bounds by means of self-reducibility. *J. of the ACM*, 57(3), 2010.

[14] Josh Alman and R. Ryan Williams. Probabilistic rank and matrix rigidity. In *ACM Symp. on the Theory of Computing (STOC)*, pages 641–652, 2017.

[15] 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.

[16] 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.

[17] Robert Andrews and Avi Wigderson. Constant-depth arithmetic circuits for linear algebra problems. *Electronic Colloquium on Computational Complexity (ECCC)*, 80, 2024.

[18] Dana Angluin and Leslie G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. *J. Comput. Syst. Sci.*, 18(2):155–193, 1979.

[19] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in NC^0. *SIAM J. on Computing*, 36(4):845–888, 2006.

[20] Sanjeev Arora and Boaz Barak. *Computational Complexity.* Cambridge University Press, 2009. A modern approach.

[21] 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.

[22] László Babai. E-mail and the unexpected power of interaction. In *SCT*, pages 30–44. IEEE Computer Society, 1990.

[23] László Babai, Lance Fortnow, and Carsten Lund. Nondeterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1(1):3–40, 1991.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] Theodore Baker, John Gill, and Robert Solovay. Relativizations of the $P=?NP$ question. *SIAM J. on Computing*, 4(4):431–442, 1975.

[31] 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.

[32] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[33] Walter Baur and Volker Strassen. The complexity of partial derivatives. *Theoret. Comput. Sci.*, 22(3):317–330, 1983.

[34] Louay Bazzi. Polylogarithmic independence can fool DNF formulas. In *48th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 63–73, 2007.

[35] Louay M. J. Bazzi. Polylogarithmic independence can fool DNF formulas. *SIAM J. Comput.*, 38(6):2220–2272, 2009.

[36] 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/.

[37] 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.

[38] 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.

[39] 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.

[40] Donald Beaver and Joan Feigenbaum. Hiding instances in multioracle queries. In *Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 37–48, 1990.

[41] Richard Beigel, Nick Reingold, and Daniel Spielman. The perceptron strikes back. In *Structure in Complexity Theory Conference*, pages 286–291, 1991.

[42] Richard Beigel and Jun Tarui. On ACC. *Computational Complexity*, 4(4):350–366, 1994.

[43] Michael Ben-Or and Richard Cleve. Computing algebraic formulas using a constant number of registers. *SIAM J. on Computing*, 21(1):54–58, 1992.

[44] 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.

[45] Norbert Blum. A boolean function requiring 3n network size. *Theoretical Computer Science*, 28:337–345, 1984.

[46] Andrej Bogdanov and Emanuele Viola. Pseudorandom bits for polynomials. *SIAM J. on Computing*, 39(6):2464–2486, 2010.

[47] 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.

[48] Mark Braverman. Poly-logarithmic independence fools AC^0 circuits. In *24th IEEE Conf. on Computational Complexity (CCC)*. IEEE, 2009.

[49] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

[50] 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.

[51] 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.

[52] Peter Bürgisser. On defining integers and proving arithmetic circuit lower bounds. *Comput. Complex.*, 18(1):81–103, 2009.

[53] Peter Bürgisser, Michael Clausen, and Mohammad Amin Shokrollahi. *Algebraic complexity theory*, volume 315 of *Grundlehren der mathematischen Wissenschaften*. Springer, 1997.

[54] Samuel R. Buss and Ryan Williams. Limits on alternation trading proofs for time-space lower bounds. *Comput. Complex.*, 24(3):533–600, 2015.

[55] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *J. of Computer and System Sciences*, 18(2):143–154, 1979.

[56] 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.

[57] Arkadev Chattopadhyay and Toniann Pitassi. The story of set disjointness. *SIGACT News*, 41(3):59–85, 2010.

[58] 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.

[59] 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.

[60] Fan R. K. Chung and Prasad Tetali. Communication complexity and quasi randomness. *SIAM Journal on Discrete Mathematics*, 6(1):110–123, 1993.

[61] Richard Cleve. Towards optimal simulations of formulas by bounded-width programs.

Computational Complexity, 1:91–105, 1991.

[62] James Cook and Ian Mertz. Tree evaluation is in space o(log n · log log n). In *STOC*, pages 1268–1278. ACM, 2024.

[63] 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.

[64] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Inf. Control.*, 64(1-3):2–21, 1985.

[65] Stephen A. Cook, Pierre McKenzie, Dustin Wehr, Mark Braverman, and Rahul Santhanam. Pebbles and branching programs for tree evaluation. *ACM Trans. Comput. Theory*, 3(2):4:1–4:43, 2012.

[66] L. Csanky. Fast parallel matrix inversion algorithms. *SIAM J. Comput.*, 5(4):618–623, 1976.

[67] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard.* Springer Verlag, 2002.

[68] Carsten Damm. Problems complete for \oplus L. *Inf. Process. Lett.*, 36(5):247–250, 1990.

[69] Carsten Damm, Stasys Jukna, and Jiří Sgall. Some bounds on multiparty communication complexity of pointer jumping. *Computational Complexity*, 7(2):109–127, 1998.

[70] 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.

[71] Apostolos Doxiadis and Christos H. Papadimitriou. *Logicomix: An Epic Search for Truth.* Bloomsbury USA, 2009.

[72] Devdatt Dubhashi and Alessandro Panconesi. *Concentration of measure for the analysis of randomized algorithms.* Cambridge University Press, 2009.

[73] Pál Erdős, Ronald L. Graham, and Endre Szemerédi. On sparse graphs with dense long paths. *Comp. and Maths. with Appls.*, 1:365–369, 1975.

[74] Euclid. *The Thirteen Books of Euclid's Elements, Vol. 2.* Dover Publications, New York, 1956. Originally published in 300 B.C.

[75] Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. Computing with noisy information. *SIAM J. Comput.*, 23(5):1001–1018, 1994.

[76] Magnus Gausdal Find, Alexander Golovnev, Edward A. Hirsch, and Alexander S. Kulikov. A better-than-3n lower bound for the circuit complexity of an explicit function. In Irit Dinur, editor, *IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 89–98. IEEE Computer Society, 2016.

[77] 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.

[78] Lance Fortnow. Time-space tradeoffs for satisfiability. *J. Comput. Syst. Sci.*, 60(2):337–353, 2000.

[79] Lance Fortnow. A simple proof of toda's theorem. *Theory Comput.*, 5(1):135–140, 2009.

[80] 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.

[81] Caxton C. Foster and Fred D. Stockton. Counting responders in an associative memory. *IEEE Trans. Computers*, 20(12):1580–1583, 1971.

[82] Aviezri S. Fraenkel and David Lichtenstein. Computing a perfect strategy for n x n chess requires time exponential in n. *J. Comb. Theory, Ser. A*, 31(2):199–214, 1981.

[83] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

[84] 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.

[85] R. Freivalds. Probabilistic machines can use less running time. In *IFIP Congress*, pages 839–842, 1977.

[86] Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984.

[87] Anka Gajentaan and Mark H. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Comput. Geom.*, 5:165–185, 1995.

[88] 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.

[89] Anna Gál and Peter Bro Miltersen. The cell probe complexity of succinct data structures. *Theoretical Computer Science*, 379(3):405–417, 2007.

[90] Anna Gál and Avi Wigderson. Boolean complexity classes vs. their arithmetic analogs. *Random Struct. Algorithms*, 9(1-2):99–111, 1996.

[91] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[92] 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.

[93] Kurt Godel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme, i. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[94] Kurt Godel, 1956. Letter to John von Neumann. https://www.anilada.com/notes/godel-letter.pdf.

[95] Mikael Goldmann, Johan Håstad, and Alexander A. Razborov. Majority gates vs. general weighted threshold gates. *Computational Complexity*, 2:277–300, 1992.

[96] Oded Goldreich. A sample of samplers - a computational perspective on sampling (survey). *Electronic Coll. on Computational Complexity (ECCC)*, 4(020), 1997.

[97] Oded Goldreich. *Computational Complexity: A Conceptual Perspective.* Cambridge University Press, 2008.

[98] Oded Goldreich. In a world of p=bpp. In *Studies in Complexity and Cryptography*, volume 6650 of *Lecture Notes in Computer Science*, pages 191–232. Springer, 2011.

[99] Oded Goldreich. On doubly-efficient interactive proof systems. *Found. Trends Theor. Comput. Sci.*, 13(3):158–246, 2018.

[100] Oded Goldreich. On the cook-mertz tree evaluation procedure. *Electron. Colloquium Comput. Complex.*, TR24-109, 2024.

[101] 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.

[102] 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/.

[103] 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.

[104] 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.

[105] 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.

[106] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[107] 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.

[108] Raymond Greenlaw, H. James Hoover, and Walter Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. 02 2001.

[109] 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.

[110] 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.

[111] Ankit Gupta, Pritish Kamath, Neeraj Kayal, and Ramprasad Saptharishi. Arithmetic circuits: A chasm at depth 3. *SIAM J. Comput.*, 45(3):1064–1079, 2016.

[112] Yuri Gurevich. Unconstrained church-turing thesis cannot possibly be true. *Bull. EATCS*, 127, 2019.

[113] Yuri Gurevich and Saharon Shelah. Nearly linear time. In *Logic at Botik, Symposium on Logical Foundations of Computer Science*, pages 108–118, 1989.

[114] Dan Gutfreund and Emanuele Viola. Fooling parity tests with parity gates. In *8th Workshop on Randomization and Computation (RANDOM)*, pages 381–392. Springer, 2004.

[115] Torben Hagerup. Fast parallel generation of random permutations. In *18th Coll. on Automata, Languages and Programming (ICALP)*, pages 405–416. Springer, 1991.

[116] 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.

[117] 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.

[118] Kristoffer Arnsfelt Hansen, Oded Lachish, and Peter Bro Miltersen. Hilbert's thirteenth problem and circuit complexity. In Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra, editors, *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, volume 5878 of *Lecture Notes in Computer Science*, pages 153–162. Springer, 2009.

[119] 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.

[120] David Harvey and Joris van der Hoeven. Integer multiplication in time $O(n \log n)$. *Annals of Mathematics*, 193(2):563 – 617, 2021.

[121] Johan Håstad. Almost optimal lower bounds for small depth circuits. In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 6–20. ACM, 1986.

[122] Johan Håstad. *Computational limitations of small-depth circuits.* MIT Press, 1987.

[123] Johan Håstad and Mikael Goldmann. On the power of small-depth threshold circuits. *Computational Complexity*, 1(2):113–129, 1991.

[124] John Hastad. Almost optimal lower bounds for small depth circuits. *Adv. Comput. Res.*, 5:143–170, 1989.

[125] Pooya Hatami and William Hoza. Theory of unconditional pseudorandom generators. *Electron. Colloquium Comput. Complex.*, TR23-019, 2023.

[126] 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.

[127] Alexander Healy, Salil P. Vadhan, and Emanuele Viola. Using nondeterminism to amplify hardness. *SIAM J. on Computing*, 35(4):903–931, 2006.

[128] 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.

[129] F. C. Hennie. One-tape, off-line turing machine computations. *Information and Control*, 8(6):553–578, 1965.

[130] Fred Hennie and Richard Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[131] Fred Hennie and Richard Stearns. Two-tape simulation of multitape turing machines. *J. of the ACM*, 13:533–546, October 1966.

[132] Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bull. Amer. Math. Soc. (N.S.)*, 43(4):439–561 (electronic), 2006.

[133] John E. Hopcroft, Wolfgang J. Paul, and Leslie G. Valiant. On time versus space. *J. ACM*, 24(2):332–337, 1977.

[134] John E. Hopcroft and Jeffrey D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc., 1969.

[135] Laurent Hyafil. On the parallel evaluation of multivariate polynomials. *SIAM J. Comput.*, 8(2):120–123, 1979.

[136] John T. Gill III. Computational complexity of probabilistic turing machines. In *STOC*, pages 91–95. ACM, 1974.

[137] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, 1988.

[138] Russell Impagliazzo. Hard-core distributions for somewhat hard problems. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 538–545, 1995.

[139] Russell Impagliazzo and Ramamohan Paturi. The complexity of k-sat. In *IEEE Conf. on Computational Complexity (CCC)*, pages 237–, 1999.

[140] Russell Impagliazzo, Ramamohan Paturi, and Michael E. Saks. Size-depth tradeoffs for threshold circuits. *SIAM J. Comput.*, 26(3):693–707, 1997.

[141] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Computer & Systems Sciences*, 63(4):512–530, Dec 2001.

[142] 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.

[143] 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.

[144] 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.

[145] Hamid Jahanjou, Eric Miles, and Emanuele Viola. Local reductions. *Information and Computation*, 261(2), 2018. Available at http://www.ccs.neu.edu/home/viola/.

[146] Stasys Jukna. *Boolean Function Complexity: Advances and Frontiers*. Springer, 2012.

[147] Franz Kafka. Aphorisms.

[148] Bala Kalyanasundaram and Georg Schnitger. The probabilistic communication complexity of set intersection. *SIAM J. Discrete Math.*, 5(4):545–557, 1992.

[149] 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.

[150] A. A. Karatsuba. The complexity of computations. *Trudy Mat. Inst. Steklov.*, 211(Optim. Upr. i Differ. Uravn.):186–202, 1995.

[151] 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.

[152] 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.

[153] Kiran S. Kedlaya and Christopher Umans. Fast polynomial factorization and modular composition. *SIAM J. on Computing*, 40(6):1767–1802, 2011.

[154] John Keegan. *The First World War.* Vintage, 2000.

[155] John Keegan. *The Second World War.* Penguin Books, 2005.

[156] Zander Kelley, Shachar Lovett, and Raghu Meka. Explicit separations between randomized and deterministic number-on-forehead communication. *CoRR*, abs/2308.12451, 2023.

[157] Adam Klivans and Rocco A. Servedio. Boosting and hard-core sets. *Machine Learning*, 53(3):217–238, 2003.

[158] 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.

[159] Kojiro Kobayashi. On the structure of one-tape nondeterministic turing machine time hierarchy. *Theor. Comput. Sci.*, 40:175–193, 1985.

[160] Pascal Koiran. Valiant's model and the cost of computing integers. *Comput. Complex.*, 13(3-4):131–146, 2005.

[161] Kenneth Krohn, W. D. Maurer, and John Rhodes. Realizing complex Boolean functions with simple groups. *Information and Control*, 9:190–195, 1966.

[162] S.-Y. Kuroda. Classes of languages and linear-bounded automata. *Inf. Control.*, 7(2):207–223, 1964.

[163] Eyal Kushilevitz and Noam Nisan. *Communication complexity.* Cambridge University Press, 1997.

[164] Gabriel Lamï¿œ. 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.

[165] 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.

[166] Kasper Green Larsen. Higher cell probe lower bounds for evaluating polynomials. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 293–301, 2012.

[167] 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.

[168] Leonid A. Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

[169] Rudolf Lidl and Harald Niederreiter. *Finite fields*, volume 20 of *Encyclopedia of Mathematics and its Applications.* Cambridge University Press, second edition, 1997.

[170] 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.

[171] Nutan Limaye, Srikanth Srinivasan, and Sébastien Tavenas. Guest column: Lower

bounds against constant-depth algebraic circuits. *SIGACT News*, 53(2):40–62, 2022.

[172] Nathan Linial and Noam Nisan. Approximate inclusion-exclusion. *Combinatorica*, 10(4):349–365, 1990.

[173] Richard Lipton. New directions in testing. In *Proceedings of DIMACS Workshop on Distributed Computing and Cryptography*, volume 2, pages 191–202. ACM/AMS, 1991.

[174] Richard J. Lipton. Straight-line complexity and integer factorization. In *International Algorithmic Number Theory Symposium*, pages 71–79, 1994. Cited at 124.

[175] Shachar Lovett. Unconditional pseudorandom generators for low degree polynomials. In *40th ACM Symp. on the Theory of Computing (STOC)*, pages 557–562, 2008.

[176] Chi-Jen Lu, Shi-Chun Tsai, and Hsin-Lung Wu. Improved hardness amplification in NP. *Theor. Comput. Sci.*, 370(1-3):293–298, 2007.

[177] 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.

[178] O. B. Lupanov. A method of circuit synthesis. *Izv. VUZ Radiofiz.*, 1:120–140, 1958.

[179] 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.

[180] Yishay Mansour, Noam Nisan, and Prasoon Tiwari. The computational complexity of universal hashing. *Theoretical Computer Science*, 107:121–133, 1993.

[181] 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.

[182] W. D. Maurer and John L. Rhodes. A property of finite simple non-abelian groups. *Pacific Journal of Mathematics*, 16(2):491–495, 1965.

[183] 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.

[184] Pierre McKenzie and Stephen A. Cook. The parallel complexity of abelian permutation group problems. *SIAM J. Comput.*, 16(5):880–909, 1987.

[185] Eric Miles and Emanuele Viola. Substitution-permutation networks, pseudorandom functions, and natural proofs. *J. of the ACM*, 62(6), 2015.

[186] 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.

[187] 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.

[188] Marvin Minsky and Seymour Papert. *Perceptrons.* MIT Press, Cambridge, MA, 1969.

[189] 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.

[190] Cristopher Moore and Stephan Mertens. *The Nature of Computation.* Oxford University Press, 2011.

[191] J. Naor and M. Naor. Small-bias probability spaces: efficient constructions and appli-

cations. In *22nd ACM Symp. on the Theory of Computing (STOC)*, pages 213–223. ACM, 1990.

[192] Joseph Naor and Moni Naor. Small-bias probability spaces: efficient constructions and applications. *SIAM J. on Computing*, 22(4):838–856, 1993.

[193] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 458–467, 1997.

[194] 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.

[195] E. I. Nechiporuk. A boolean function. *Soviet Mathematics-Doklady*, 169(4):765–766, 1966.

[196] Valery A. Nepomnjaščiĭ. Rudimentary predicates and Turing calculations. *Soviet Mathematics-Doklady*, 11(6):1462–1465, 1970.

[197] NEU. From RAM to SAT. Available at http://www.ccs.neu.edu/home/viola/, 2012.

[198] Ilan Newman. Private vs. common random bits in communication complexity. *Information Processing Letters*, 39(2):67–71, 1991.

[199] Noam Nisan. Pseudorandom bits for constant depth circuits. *Combinatorica. An Journal on Combinatorics and the Theory of Computing*, 11(1):63–70, 1991.

[200] 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.

[201] Noam Nisan and Avi Wigderson. Hardness vs randomness. *J. of Computer and System Sciences*, 49(2):149–167, 1994.

[202] Noam Nisan and Avi Wigderson. Lower bounds on arithmetic circuits via partial derivatives. *Comput. Complexity*, 6(3):217–234, 1996/97.

[203] Ryan O'Donnell. Hardness amplification within *NP*. *J. of Computer and System Sciences*, 69(1):68–94, August 2004.

[204] Ryan O'Donnell. *Analysis of Boolean Functions.* Cambridge University Press, 2014.

[205] Christos H. Papadimitriou. *Computational Complexity.* Addison–Wesley, 1994.

[206] Christos H. Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. *J. Comput. Syst. Sci.*, 43(3):425–440, 1991.

[207] 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.

[208] Seymour Papert. One AI or Many? *Daedalus*, 117, 1988.

[209] Mihai Pătraşcu. Succincter. In *49th IEEE Symp. on Foundations of Computer Science (FOCS).* IEEE, 2008.

[210] 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.

[211] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. of the ACM*, 26(2):361–381, 1979.

[212] Pavel Pudlák, Vojtěch Rödl, and Jiří Sgall. Boolean circuits, tensor ranks, and communication complexity. *SIAM J. on Computing*, 26(3):605–633, 1997.

[213] C. Radhakrishna Rao. Factorial experiments derivable from combinatorial arrangements of arrays. *Suppl. J. Roy. Statist. Soc.*, 9:128–139, 1947.

[214] Anup Rao and Amir Yehudayoff. *Communication complexity.* 2019. https://homes.cs.washington.edu/ anuprao/pubs/book.pdf.

[215] Ran Raz. The BNS-Chung criterion for multi-party communication complexity. *Computational Complexity*, 9(2):113–122, 2000.

[216] 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.

[217] Alexander Razborov and Steven Rudich. Natural proofs. *J. of Computer and System Sciences*, 55(1):24–35, August 1997.

[218] Alexander A. Razborov. On the distributional complexity of disjointness. *Theor. Comput. Sci.*, 106(2):385–390, 1992.

[219] Alexander A. Razborov. A simple proof of Bazzi's theorem. *ACM Transactions on Computation Theory (TOCT)*, 1(1), 2009.

[220] Omer Reingold. Undirected connectivity in log-space. *J. of the ACM*, 55(4), 2008.

[221] Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. *SIAM J. Comput.*, 50(3), 2021.

[222] J. M. Robson. N by N checkers is exptime complete. *SIAM J. Comput.*, 13(2):252–267, 1984.

[223] J. M. Robson. An O(T log T) reduction from RAM computations to satisfiability. *Theoretical Computer Science*, 82(1):141–149, 1991.

[224] Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Spartan Books, 1962.

[225] Eyal Rozenman and Salil P. Vadhan. Derandomized squaring of graphs. In *APPROX-RANDOM*, pages 436–447, 2005.

[226] Rahul Santhanam. On separators, segregators and time versus space. In *IEEE Conf. on Computational Complexity (CCC)*, pages 286–294, 2001.

[227] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[228] M. Schaefer and C. Umans. Completeness in the polynomial-time hierarchy: a compendium. *SIGACT News, Complexity Theory Column*, 2002.

[229] Arnold Schönhage. Storage modification machines. *SIAM J. Comput.*, 9(3):490–508, 1980.

[230] Adi Shamir. Factoring numbers in o(log n) arithmetic steps. *Information Processing Letters*, 8(1):28–31, 1979.

[231] Adi Shamir. IP = PSPACE. *J. of the ACM*, 39(4):869–877, October 1992.

[232] Claude Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28(4):656–715, 1949.

[233] Claude E. Shannon. The synthesis of two-terminal switching circuits. *Bell System Tech. J.*, 28:59–98, 1949.

[234] 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.

[235] Victor Shoup. New algorithms for finding irreducible polynomials over finite fields. *Mathematics of Computation*, 54(189):435–447, 1990.

[236] Amir Shpilka and Avi Wigderson. Depth-3 arithmetic circuits over fields of characteristic zero. *Comput. Complex.*, 10(1):1–27, 2001.

[237] Amir Shpilka and Amir Yehudayoff. Arithmetic circuits: A survey of recent results and open questions. *Found. Trends Theor. Comput. Sci.*, 5(3-4):207–388, 2010.

[238] Alan Siegel. On universal classes of extremely random constant-time hash functions. *SIAM J. on Computing*, 33(3):505–543, 2004.

[239] Michael Sipser. A complexity theoretic approach to randomness. In *ACM Symp. on the Theory of Computing (STOC)*, pages 330–335, 1983.

[240] Michael Sipser. *Introduction to the theory of computation, 3rd ed.* PWS Publishing Company, 1997.

[241] 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.

[242] Roman Smolensky. On representations by low-degree polynomials. In *34th IEEE IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 130–138, 1993.

[243] 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.

[244] A. Spivak. Brainteasers b 201: Strange painting. *Quantum*, page 13, 1997.

[245] Richard Edwin Stearns, Juris Hartmanis, and Philip M. Lewis II. Hierarchies of memory limited computations. In *SWCT*, pages 179–190. IEEE Computer Society, 1965.

[246] 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.

[247] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976.

[248] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[249] Volker Strassen. Die rcrechnungskomplcxittit von elementarsvmmetrischen funktionen und von interpolationskoetfiziearten. *Numer. Math.*, 20:238–251, 1973.

[250] Volker Strassen. Polynomials with rational coefficients which are hard to compute. *SIAM J. Comput.*, 3:128–149, 1974.

[251] B. A. Subbotovskaya. Realizations of linear functions by formulas using +, *, -. *Soviet Mathematics-Doklady*, 2:110–112, 1961.

[252] Madhu Sudan, Luca Trevisan, and Salil Vadhan. Pseudorandom generators without the XOR lemma. *J. of Computer and System Sciences*, 62(2):236–266, 2001.

[253] Xiaoming Sun. A 3-party simultaneous protocol for SUM-INDEX. *Algorithmica*,

36(1):89–111, 2003.

[254] Róbert Szelepcsényi. The moethod of focing for nondeterministic automata. *Bull. EATCS*, 33:96–99, 1987.

[255] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[256] Justin Thaler. Proofs, arguments, and zero-knowledge. *Found. Trends Priv. Secur.*, 4(2-4):117–660, 2022.

[257] Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM J. on Computing*, 20(5):865–877, 1991.

[258] Boris A. Trakhtenbrot. A survey of russian approaches to perebor (brute-force searches) algorithms. *IEEE Ann. Hist. Comput.*, 6(4):384–400, 1984.

[259] Luca Trevisan. The program-enumeration bottleneck in average-case complexity theory. In *CCC*, pages 88–95. IEEE Computer Society, 2010.

[260] Vladimir Trifonov. An o(logn loglogn) space algorithm for undirected st-connectivity. *SIAM J. Comput.*, 38(2):449–483, 2008.

[261] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, 1937.

[262] Salil P. Vadhan. Pseudorandomness. *Foundations and Trends in Theoretical Computer Science*, 7(1-3):1–336, 2012.

[263] Valiant. On non-linear lower bounds in computational complexity. In *ACM Symp. on the Theory of Computing (STOC)*, pages 45–53, 1975.

[264] 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.

[265] 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.

[266] Leslie G. Valiant and Vijay V. Vazirani. NP is as easy as detecting unique solutions. *Theor. Comput. Sci.*, 47(3):85–93, 1986.

[267] J. H. van Lint. *Introduction to coding theory.* Springer-Verlag, Berlin, third edition, 1999.

[268] 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.

[269] Dieter van Melkebeek and Ran Raz. A time lower bound for satisfiability. *Theor. Comput. Sci.*, 348(2-3):311–320, 2005.

[270] N. V. Vinodchandran. A note on the circuit complexity of PP. *Theor. Comput. Sci.*, 347(1-2):415–418, 2005.

[271] 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/.

[272] Emanuele Viola. The complexity of constructing pseudorandom generators from hard functions. *Computational Complexity*, 13(3-4):147–188, 2004.

[273] Emanuele Viola. The complexity of hardness amplification and derandomization. *Ph.D. thesis, Harvard University*, 2006.

[274] 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.

[275] Emanuele Viola. On approximate majority and probabilistic time. *Computational Complexity*, 18(3):337–375, 2009.

[276] Emanuele Viola. On the power of small-depth computation. *Foundations and Trends in Theoretical Computer Science*, 5(1):1–72, 2009.

[277] Emanuele Viola. The sum of d small-bias generators fools polynomials of degree d. *Computational Complexity*, 18(2):209–217, 2009.

[278] Emanuele Viola. Reducing 3XOR to listing triangles, an exposition. Available at http://www.ccs.neu.edu/home/viola/, 2011.

[279] Emanuele Viola. Bit-probe lower bounds for succinct data structures. *SIAM J. on Computing*, 41(6):1593–1604, 2012.

[280] Emanuele Viola. The complexity of distributions. *SIAM J. on Computing*, 41(1):191–218, 2012.

[281] Emanuele Viola. The communication complexity of addition. *Combinatorica*, pages 1–45, 2014.

[282] 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/.

[283] Emanuele Viola. Non-abelian combinatorics and communication complexity. *SIGACT News, Complexity Theory Column*, 50(3), 2019.

[284] Emanuele Viola. Pseudorandom bits and lower bounds for randomized turing machines. *Theory of Computing*, 18(10):1–12, 2022.

[285] Emanuele Viola and Avi Wigderson. Norms, XOR lemmas, and lower bounds for polynomials and protocols. *Theory of Computing*, 4:137–168, 2008.

[286] Emanuele Viola and Avi Wigderson. One-way multiparty communication lower bound for pointer jumping with applications. *Combinatorica*, 29(6):719–743, 2009.

[287] Avi Wigderson. *Mathematics and Computation: A Theory Revolutionizing Technology and Science*. Princeton University Press, 2019.

[288] John Williams. *Stoner*. New York Review Books Classics, 2006.

[289] Ryan Williams. *Algorithms and Resource Requirements for Fundamental Problems*. PhD thesis, Carnegie Mellon University, 2007.

[290] Ryan Williams. Non-uniform ACC circuit lower bounds. In *IEEE Conf. on Computational Complexity (CCC)*, pages 115–125, 2011.

[291] Ryan Williams. Nonuniform ACC circuit lower bounds. *J. of the ACM*, 61(1):2:1–2:32, 2014.

[292] Ryan Williams. Simulating time in square-root space. *Electron. Colloquium Comput. Complex.*, TR25-017, 2025.

[293] Andrew Yao. Theory and applications of trapdoor functions. In *23rd IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 80–91. IEEE, 1982.

[294] Andrew Yao. Separating the polynomial-time hierarchy by oracles. In *26th IEEE*

Symp. on Foundations of Computer Science (FOCS), pages 1–10, 1985.

[295] 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.

[296] 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.

[297] Andrew Chi-Chih Yao. On ACC and threshold circuits. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 619–627, 1990.

"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." [147]

Appendix A

Math facts

Here we collect some additional observations and mathematical facts, sometimes basic, that are used in the main text.

A.1 Integers

Fact A.1. $\gcd(x,y) = \gcd(x \mod y, y)$.

A.2 Basic inequalities

Fact A.2. $\binom{n}{k} \le c2^n/\sqrt{n}$, for all k.

Fact A.3. $(n/k)^k \le \binom{n}{k} \le (en/k)^k$.

Fact A.4. $1+\alpha \le e^\alpha \le 1+\alpha+\alpha^2$, for all $\alpha \le 1$.
The rhs is $\le 1+2\alpha$ for $\alpha \in [0,1]$, and $\le 1+\alpha/2$ for $\alpha \in [-1/2, 0]$ (because $\alpha(1+\alpha) \le \alpha/2 \iff (1+\alpha) \ge 1/2$).

Fact A.5. $(1+\alpha)^r \ge 1+r\alpha$ for all $\alpha \ge -1$ and $r \ge 1$.

Fact A.6. For any $\alpha \in \mathbb{R}$,$1+\alpha \le \frac{1}{1-\alpha} \le 1+(1+\epsilon)\alpha$. The first inequality holds for any $\alpha \in \mathbb{R}$, the second for $\alpha \in [0, \epsilon/(1+\epsilon)]$.

For example, $1/(1-\alpha) \le 1+2\alpha$ $(\epsilon = 1)$ for $\alpha \le 1/2$.

A.2.1 Squaring tricks

Fact A.7. For every real random variable X, $\mathbb{E}^2[X] \le \mathbb{E}[X^2]$.

Proof. $\mathbb{E}[(X-\mathbb{E}[X])^2]$ is ≥ 0. Expand the square. **QED**

This is a special case of:

Fact A.8. 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.3 Probability theory

Developing intuition about random variables is one of the hardest skills to master and even define. To anyone struggling I'd like to mention that my background was null, and in fact I didn't even like the emphasis on randomization, given the status of the field (cf. Chapter 3). Naturally, with effort I grew to like it. 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."

I find the joke at the *incipit* of Chapter 2 a good illustration of how elusive the concept of probability is.

Fact A.9. [Linearity of expectation] TBD

A.4 Groups

A group is a set together with an invertible operation. The theory of groups is *rich and pervasive.*

A.5 Linear algebra

The only game in town.

First we recall list some basic definitions.

- A vector $v = (v_1, v_2, \ldots, v_n) \in \mathbb{R}^n$.
- Inner product $\langle v, w\rangle = \sum_i v_i \cdot w_i$.
- Two vectors are orthogonal, denoted $v \perp w$, if $\langle v, w\rangle = 0$.
- The length of a vector is $|v| = |v|_2 := \sqrt{\sum_i v_i^2} = \sqrt{\langle v, w\rangle}$.

Fact A.10. [Triangle inequality for vectors] $|v + w| \leq |v| + |w|$ for any vectors v, w.

Fact A.11. 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.12. 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). 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 Lemma 2.1. **QED**

Fact A.13. Orthogonal vectors v_i are in particular linearly independent.

Proof. Suppose that $\sum_i a_i v_i = 0$ for some coefficients a_i. Then for any j we have

$$0 = \langle \sum a_i v_i, v_j \rangle = \sum a_i \langle v_i, v_j \rangle = a_j \langle v_j, v_j \rangle,$$

and so $a_j = 0$. **QED**

Definition A.1. 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.1 makes most sense and is almost always more convenient.

Fact A.14. $\text{rank}(A \otimes B) = \text{rank}(A) \cdot \text{rank}(B)$.

A.5.1 The eigenbasis Theorem 12.5

The proof relies on the fundamental theorem of algebra that every polynomial has a complex root. This proves the existence of eigenvectors. Then from first principles one can verify that for symmetric matrices they are real, and one can find an orthonormal basis.

A.6 Polynomials

Fact A.15. 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**

Fact A.16. In the context of the proof of 14.1, $k \cdot e_k = \sum_{i=1}^{k}(-1)^{i-1} e_{k-i} \cdot p_i$, for all k.

Proof. Let $f(x) := (x-x_1)(x-x_2)\cdots(x-x_s)$. The coefficient of x^i in $f(x)$ is $e_{s,s-i}(x_1, x_2, \ldots, x_s) \cdot (-1)^{s-i}$. We also have $f(x_j) = 0$ for any j. Summing over j proves the claim for $k = s$. To prove it for $s > k$, consider any monomial on the LHS. Set to 0 all the other variables. Now the claim reduces to the case $s = k$. This shows that the coefficient of any monomial on the LHS is the same as that on the RHS, finishing the proof. **QED**

A.7 Analysis of boolean functions over groups

A.7.1 Abelian groups

Fact A.17. Let D be a distribution over $[2]^n$ that fools degree-1 polynomials over $\mathbb{F}_2$ with error 0 (a.k.a. 0-biased). Then D is uniform.

Proof. By Exercise 11.6, we write for any $x \in [2]^n$,

$$D(x) = \sum_{\alpha} \widehat{D}_\alpha x^\alpha = \widehat{D}_\emptyset x^\emptyset = \widehat{D}_\emptyset = \mathbb{E}_y[D(y) y^\emptyset] = \mathbb{E}_y[D(y)] = \frac{1}{2^n} \sum_y D(y) = \frac{1}{2^n}.$$

QED

Appendix B

Annotated meta-bibliography

[20] Standard reference for complexity theory. Especially good reference for a proof of the PCP theorem.

[97] Complexity theory. Somewhat narrower in scope compared [20]. Some uncommon details and examples can be found here.

[146] A book "all about proving lower bounds."

[169] Massive reference for finite fields, though the focus is not computational.

[204] All things analysis of boolean functions (Exercise 11.6).

[214] Communication complexity.

[72] The theory of deviation bounds, from the point of view of theoretical computer science.

[287] Broad and enticing overview of the field.

[53] Algebraic complexity.

[262] Pseudorandomness.

[125] Pseudorandomness, with a focus on unconditional pseudorandom generators.

[134] Still a great reference; stay away from the watered-down "revisions."

[240] Standard reference for theory of computation and introduction to complexity theory.

[205] Still conveying passion and originality. Seferis' poem is worth quoting in full:

I wish nothing else but to speak simply
please grant me this privilege
because we have burdened our song with so much music
that it is slowly sinking
and our art has become so ornate
that the makeup has corroded her face
and it is time to say our few simple words
because tomorrow our soul sails away

B.1 Some non-technical books

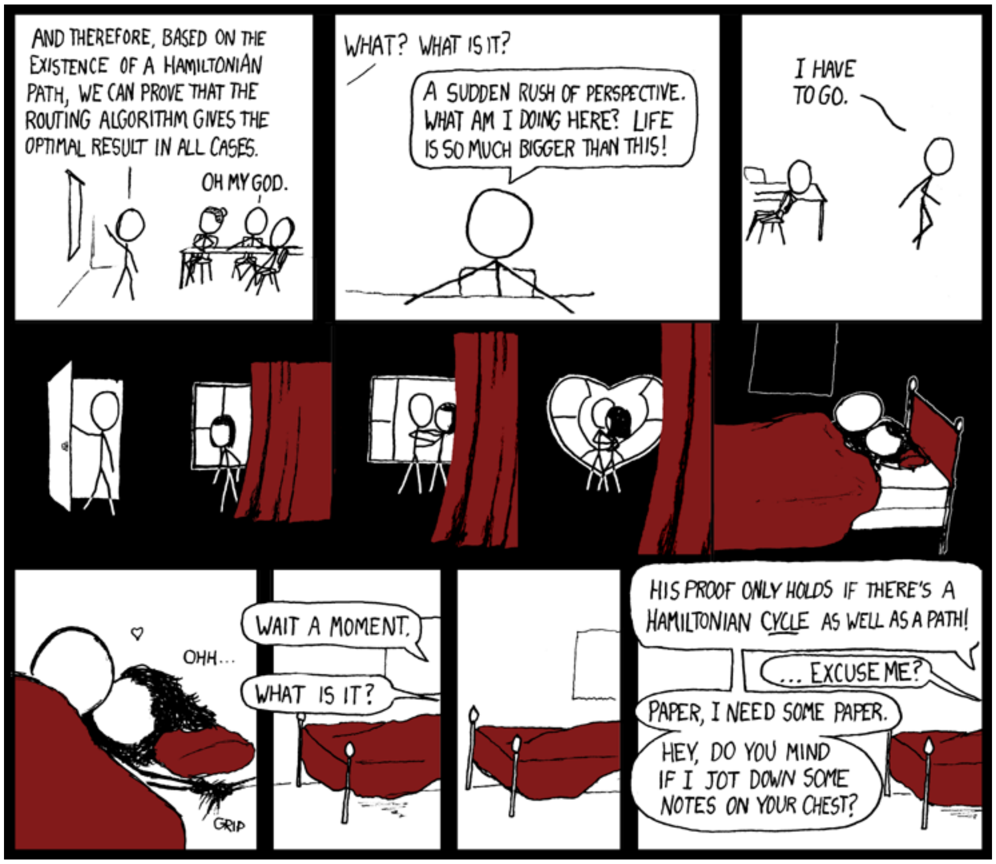
[71]: Another very enjoyable read. Wittgenstein was hilarious, but then again he always is. But if you like me were expecting an Armageddon or a Ragnarok unleashed by Godel's incompleteness, you are going to be disappointed. There are only c scenes with him, and while one does show him on the board scribbling actual lines from him masterwork, the meaning and firepower of his discovery is nowhere to be found in the book, which ends instead with yet another...

[288]: Highly recommended if you ever wonder about the meaning of life, especially academic life, or even marital.

[154, 155]. Masterful accounts of key events that shaped the geography of science production.

Appendix C

Research tips


AND THEREFORE, BASED ON THE EXISTENCE OF A HAMILTONIAN PATH, WE CAN PROVE THAT THE ROUTING ALGORITHM GIVES THE OPTIMAL RESULT IN ALL CASES.
OH MY GOD.
WHAT? WHAT IS IT?
A SUDDEN RUSH OF PERSPECTIVE. WHAT AM I DOING HERE? LIFE IS SO MUCH BIGGER THAN THIS!
I HAVE TO GO.
OHH...
GRIP
WAIT A MOMENT.
WHAT IS IT?
HIS PROOF ONLY HOLDS IF THERE'S A HAMILTONIAN CYCLE AS WELL AS A PATH!
... EXCUSE ME?
PAPER, I NEED SOME PAPER.
HEY, DO YOU MIND IF I JOT DOWN SOME NOTES ON YOUR CHEST?

1. Thou shall not use the letter n.
2. Keep a journal.

Index