Notes on Computational Complexity Theory
CPSC 468/568: Spring 2020

James Aspnes

2019-12-01 19:56

Contents

List of Figures

List of Tables

List of Algorithms

Preface

These are notes for the Spring 2020 semester of the Yale course CPSC 468/568 Computational Complexity. This document also incorporates the lecture schedule and assignments, as well as some sample assignments from previous semesters. Because this is a work in progress, it will be updated frequently over the course of the semester.

Updated versions of these notes will appear at http://www.cs.yale.edu/homes/aspnes/classes/468/notes.pdf. If the Yale CS server becomes inaccessible, a backup copy can be found at https://www.dropbox.com/sh/j98y7z3k7u9iobh/AAAglmWHGH5gdyKoi3rSJewaa?dl=0.

Notes from the Spring 2017 version of the course can be found at http://www.cs.yale.edu/homes/aspnes/classes/468/notes-2017.pdf.

The Spring 2016 version of the course was taught by Joan Feigenbaum, and the organization of this course is in part based on this previous version. Information about the Spring 2016 course, including lecture notes and assignments, can be found at http://zoo.cs.yale.edu/classes/cs468/spr16/.

Much of the course follows the textbook, *Computational Complexity: A Modern Approach*, by Sanjeev Arora and Boaz Barak. In most cases you'll find this textbook contain much more detail than what is presented here, so it is probably better to consider these notes a supplement to it rather than to treat them as your primary source of information.

Syllabus

Description

Introduction to the theory of computational complexity. Basic complexity classes, including polynomial time, nondeterministic polynomial time, probabilistic polynomial time, polynomial space, logarithmic space, and nondeterministic logarithmic space. The roles of reductions, completeness, randomness, and interaction in the formal study of computation. After Computer Science 365 or with permission of the instructor.

Meeting times

Tuesday and Thursday 14:30–15:45 in [[[**To be announced.**]]].

On-line course information

The lecture schedule, course notes, and all assignments can be found in a single gigantic PDF file at http://www.cs.yale.edu/homes/aspnes/classes/468/notes.pdf. You should probably bookmark this file, as it will be updated frequently.

For office hours, see http://www.cs.yale.edu/homes/aspnes#calendar.

Staff

The instructor for the course is James Aspnes. Office: AKW 401. Email: james.aspnes@gmail.com. URL: http://www.cs.yale.edu/homes/aspnes/.

Teaching fellows / ULAs: [[[**To be announced.**]]].

Textbook

The textbook for the class is:

Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009. ISBN 0521424267. QA267.7.A76X 2009 (LC).

The status of this book is a little complicated. It is available on-line in an inconvenient format from Yale campus IP addresses at http://proquest.safaribooksonline.com/9780511530753. A draft version in PDF format is also available at http://theory.cs.princeton.edu/complexity/book.pdf, but this is not identical to the final published version. The final published version is not currently available in printed form. So we will mostly be working from the PDF draft. Where it makes a difference, in the notes we will cite the PDF draft as [AB07] and the print version as [AB09], and we will try to avoid citing the unavailable print version whenever possible.

Reserved books at Bass library

In addition to the textbook, the following books are on reserve at Bass Library:

- Oded Goldreich. *Computational Complexity: A Conceptual Perspective.* Cambridge University Press, 2008.
- Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.
- Michael R. Garey and Davis S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman, 1981.

The first two are other widely-used computational complexity theory textbooks, which may offer perspectives on various topics that complement Arora-Barak and the course notes. The last is a classic collection of known **NP**-hard problems, and can be helpful as a starting point for checking if some problem you are interested is also **NP**-hard.

Other useful resources

- https://complexityzoo.uwaterloo.ca/Complexity_Zoo. On-line catalog of complexity classes.

- http://www.scottaaronson.com/papers/pnp.pdf. Survey of current state of the **P** vs. **NP**problem.

Course requirements

Six homework assignments (60% of the semester grade) plus a final exam (40%).

Use of outside help

Students are free to discuss homework problems and course material with each other, and to consult with the instructor or a TA. Solutions handed in, however, should be the student's own work. If a student benefits substantially from hints or solutions received from fellow students or from outside sources, then the student should hand in their solution but acknowledge the outside sources, and we will apportion credit accordingly. Using outside resources in solving a problem is acceptable but plagiarism is not.

Clarifications for homework assignments

From time to time, ambiguities and errors may creep into homework assignments. Questions about the interpretation of homework assignments should be sent to the instructor at james.aspnes@gmail.com. Clarifications will appear in an updated version of the assignment.

In some circumstances, you may be able to get a faster response using Piazza, at http://piazza.com/yale/spring2017/cpsc468. Note that questions you ask there are visible to other students if not specifically marked private, so be careful about broadcasting your draft solutions.

Late assignments

Late assignments will not be accepted without a Dean's Excuse.

Lecture schedule

As always, the future is uncertain, so you should take parts of the schedule that haven't happened yet with a grain of salt. Readings refer to chapters or sections in the course notes, except for those specified as in AB, which refer to the course textbook [AB07].

Office hours, lecture times, and assignment due dates can be found at http://www.cs.yale.edu/homes/aspnes#calendar.

2020-01-14 [[[**To be announced.**]]]

2020-01-16 [[[**To be announced.**]]]

2020-01-21 [[[**To be announced.**]]]

2020-01-23 [[[**To be announced.**]]]

2020-01-28 [[[**To be announced.**]]]

2020-01-30 [[[**To be announced.**]]]

2020-02-04 [[[**To be announced.**]]]

2020-02-06 [[[**To be announced.**]]]

2020-02-11 [[[**To be announced.**]]]

2020-02-13 [[[**To be announced.**]]]

2020-02-18 [[[**To be announced.**]]]

2020-02-20 [[[**To be announced.**]]]

2020-02-25 [[[**To be announced.**]]]

2020-02-27 [[[**To be announced.**]]]

2020-03-03 [[[**To be announced.**]]]

2020-03-05 [[[**To be announced.**]]]

2020-03-24 [[[**To be announced.**]]]

2020-03-26 [[[**To be announced.**]]]

2020-03-31 [[[**To be announced.**]]]

2020-04-02 [[[**To be announced.**]]]

2020-04-07 [[[**To be announced.**]]]

2020-04-09 [[[**To be announced.**]]]

2020-04-14 [[[**To be announced.**]]]

2020-04-16 [[[**To be announced.**]]]

2020-04-21 [[[**To be announced.**]]]

2020-04-23 [[[**To be announced.**]]]

Chapter 1

Introduction

Last updated 2017. Some material may be out of date.

The basic question that **computational complexity theory** [1] tries to answer is:

> Given a problem X, and a machine model M, how long does it take to solve X using a machine from M?

Unfortunately we can only rarely answer this question. So the real questions of complexity theory are:

1. How can we classify problems into **complexity classes** based on their apparent difficulty?
2. What relationships can we established between these complexity classes?
3. What techniques might we be able to use to resolve relationships between complexity classes whose status is still open?

The most famous of these questions center around the **P** vs. **NP** problem. Here **P** consists of problems that can be *solved* in time polynomial in the

[1] When I started in this business, the field was known as just **complexity theory**. But "complexity theory" is now often used to refer to the study of **complex systems**, a field of research strongly associated with the Santa Fe Institute. At its best, this field represents some of the finest minds in physics seeking to cope with the realization that the behavior of much of the universe is nonlinear and cannot be described succinctly. At its worst, it consists of popular-science writers looking at pictures of fractals and saying "Wow, those things are *really* complex!" We use the term "computational complexity theory" to avoid confusion with this largely unrelated area of research, although when not being careful we will often drop the "computational" part.

size of the input on reasonable computational devices (we will see a more formal definition in Chapter 4), **NP** consists of problems whose solutions can be *verified* in time polynomial in the size of the input (Chapter 5), and the big question is whether there are any problems in **NP** that are not also in **P**. Much of the development of computational complexity theory has arisen from researchers working very hard to answer this question.

Computational complexity theory also includes questions about other computational resources. In addition to time, we can ask about how much **space** it takes to solve a particular problem. The space class analogous to **P** is **L**, the class of problems that can be solved using space logarithmic in the size of the input. Because a log-space machine can only have polynomially-many distinct states, any problem solvable in **L** is also solvable in **P**, and indeed **L** is the largest space complexity class that includes only problems we can expect to solve efficiently in practice. As with **P**, a big open question is whether solving problems in log space is any harder than checking the solutions. In other words, is **L** equal to its nondeterministic counterpart **NL**? Other classes of interest include **PSPACE**, the class of problems solvable using polynomial space, which contains within it an entire hierarchy of classes that are to **NP** what **NP** is to **P**.[2]

In addition to asking about deterministic computations, we can also ask about what happens with randomness. The class **BPP** of bounded-error probabilistic polynomial-time computations, in which we must produce the right answer substantially more often than not in polynomial time, corresponds to typical randomized algorithms. It is an open question whether **BPP** = **P** or not. Cryptographers tend to think they are equal (a good pseudorandom number generator will let you simulate **BPP** in **P**), but we don't really know. A similar question arises for **BQP**, the class of problems solvable with bounded error in polynomial time using a quantum computer. Quantum computers seem pretty powerful, but as far as we know, **BQP** could be equal to **P**.

Complexity theory is a huge field, probably the most technically developed part of theoretical computer science, and we won't be able to cover all of it. But the hope is that this course will give you enough of an introduction to the basic ideas of complexity theory that if you are interested, you will be able to pursue it further on your own.

[2] Which is to say: possibly equal but probably not.

Chapter 2

Problems and languages

Last updated 2017. Some material may be out of date.

For the most part, the kind of problems we study in complexity theory are **decision problems**, where we are presented with an input x and have to answer "yes" or "no" based on whether x satisfies some predicate P. An example is GRAPH 3-COLORABILITY:[1] Given a graph G, is there a way to assign three colors to the vertices so that no edge has two endpoint of the same color?

Most of the algorithms you've probably seen have computed actual functions instead of just solving a decision problem, so the choice to limit ourselves (mostly) to decision problems requires some justification. The main reason is that decision problems are simpler to reason about than more general functions, and our life as complexity theorists is hard enough already. But we can also make some plausible excuses that decision problems in fact capture most of what is hard about function computation.

For example, if we are in the graph-coloring business, we probably want to find a coloring of G rather than just be told that it exists. But if we have a machine that tells use whether or not a coloring exists, with a little tinkering we can turn it into a machine that tells us if a coloring exists consistent with locking down a few nodes to have particular colors.[2] With this modified machine, we can probe potential colorings one vertex at a time, backing off if we place a color that prevents us from coloring the entire graph. Since we

[1] It's conventional to name problems in all-caps.

[2] Since we can't necessarily rewrite the code for our graph-coloring tester, this involves adjusting the input graph. The basic idea is that we can add a triangle off on the side somewhere that gives us nodes with our three colors, and force a node to have a particular color by linking it to the two other colors in the triangle.

have to call the graph-coloring tester more than once, this is more expensive than the original decision problem, but it will still be reasonably efficient if our algorithm for the decision problem is.

Concentrating on decision problems fixes the outputs of what we are doing. We also have to formalize how we are handling inputs. Typically we assume that an instance x of whatever problem we are looking at has an encoding $\llcorner x \lrcorner$ over some **alphabet** Σ, which can in principle always be reduced to just $\{0, 1\}$. Under this assumption, the input tape contains a sequence of symbols from Σ bounded by an infinite sequence of blanks in both directions. The input tape head by convention starts on the leftmost symbol in the input.

We typically don't care too much about the details of this encoding, as long as (a) it is reasonably efficient in its use of space (for example, we'd encode a natural number like 19 using its **binary** representation $\llcorner 19 \lrcorner = 10011$ instead of its **unary** representations $1^{19} = 1111111111111111111$); and (b) it makes the features of the input we want to get at reasonably accessible (if we want to encode two primes, we represent them something like $\llcorner \langle 17, 19 \rangle \lrcorner = 10001, 10011$ instead of $\llcorner 17 \cdot 19 \lrcorner \llcorner 323 \lrcorner = 101000011$, even thought the latter representation in principle lets us recover the former). The justification for being blasé about the details is that we should be able to convert between reasonable representations at low cost; so if we can solve a problem efficiently for one representation of the input, we should also be able to solve it efficiently for any other reasonable representations, and if we can't, that's a sign that there may be something wrong with our problem.[3]

Summing up, on the output side we consider yes/no outputs only, and on the input side we insist that all inputs are encoded as strings of symbols. We can formalize this idea by defining a **language** as a set of finite strings over some alphabet Σ. If $x \in L$ then are supposed to answer "yes" and if $x \notin L$, we are supposed to answer "no." An implementation of a language is a machine of some sort that does this correctly, and a **complexity class** will just be a set of languages whose implementations have some particular complexity property.

If we have a Turing machine M that halts in an accepting state on any input $x \in L$ and halts in a rejecting state on any input $x \notin L$, we say that M **decides** L. Each M that halts on all inputs decides exactly one language L, which we write as $L(M)$.

[3] The technical term here is that we want our problems to be **representation independent**, which is borrowed from the theory of abstract data types; the idea is that we want the meaning of the problem to depend on x and not a particular choice of $\llcorner x \lrcorner$.

Chapter 3

Models of computation

Last updated 2017. Some material may be out of date.

Many models of computation have been proposed over the years. The **Church-Turing thesis** is the observation that all of the ones that are sufficiently powerful and that can plausibly be realized in practice are capable of computing the same predicates.[1] The **extended Church-Turing thesis** says that all reasonable computational models can simulate each other with slowdown at most polynomial in the duration of the computation.

Taken together, these let us ignore the question of precisely which model of computation we are using, and we can settle on whatever model we find convenient (for example, C programs). But when reasoning about specific computations, it can be helpful to settle on a single, mathematically simple model. This usually ends up being a **Turing machine**.

3.1 Turing machines

A **Turing machine** (**TM** for short) consists of one or more a **tapes**, which we can think of as storing an infinite (in both directions) array of symbols from some **alphabet** Γ, one or more **heads**, which point to particular locations on the tape(s), and a **finite-state control** that controls the movement of the heads on the tapes and that may direct each head to rewrite the symbol

[1] Alternatively, the Church-Turing thesis is a declaration of what models we will consider to be plausibly realizable. This roughly means that we can only do a finite amount of work in a finite amount of time, a restriction that may not hold if, for example, you live very close to a rotating black hole or have the ability to carry out **supertasks** in apparent violation of the usual laws of physics.

in the cell it currently points to, based on the current symbols under the heads and its state, an element of its state space Q.

In its simplest form, a Turing machine has exactly one tape that is used for input, computation, and output, and has only one head on this tape. This is often too restrictive to program easily, so we will typically assume at least three tapes (with corresponding heads): one each for input, work, and output. This does not add any significant power to the model, and indeed not only is it possible for a one-tape Turing machine to simulate a k-tape Turing machine for any fixed k, it can do so with only polynomial slowdown. Similarly, even though in principle we can limit our alphabet to just $\{0, 1\}$, we will in general assume whatever (finite) alphabet is convenient for the tape cells.

Formally, we can define a k-tape Turing machine as a tuple $\langle \Gamma, Q, \delta \rangle$, where Γ is the alphabet; Q is the state space of the finite-state controller; and $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{\mathsf{L}, \mathsf{S}, \mathsf{R}\}^k$ is the **transition function**, which specifies for each state $q \in Q$ and k-tuple of symbols from Γ seen at the current positions of the heads the next state $q' \in Q$, a new tuple of symbols to write to the current head positions, and a direction Left, Stay, or Right to move each head in.[2]

More formal definitions of a Turing machine add additional details to the tuple, including an explicit blank symbol $b \in \Gamma$, a restricted input alphabet $\Sigma \subseteq \Gamma$ (which generally does not include b, since the blank regions of the input tape mark the ends of the input), an explicit starting state $q_0 \in Q$, and an explicit list of accepting states $A \subseteq Q$. We will include these details as needed.

To avoid confusion between the state q of the controller and the state of the Turing machine as a whole (which includes the contents of the tapes and the positions of the heads as well), we will describe the state of the machine as a whole as its **configuration** and reserve **state** for just the part of the configuration that represents the state of the controller.

Because we allow the Turing machine to do nothing, we do not need to include an explicit **halting state**. Instead, we will define the machine to

[2] This definition mostly follows the one in §1.2 of Arora and Barak [AB07]. One difference is that we allow the machine to write to all k of its tapes, while Arora and Barak reserve the first tape as an input tape and thus define the transition function as $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{\mathsf{L}, \mathsf{S}, \mathsf{R}\}^k$. The advantage of the more general definition is that it allows for a one-tape machine where the single tape function in all three roles of input, storage, and output. The disadvantage is that if we do want a read-only input tape, which is important when defining sublinear space complexity classes like **L**, we must explicitly require that δ always write to the first tape whatever symbol is already there.

halt if it reaches a configuration where it does not move its heads, change its state, or change any of the tape symbols.

3.1.1 Computations

A computation of a predicate by a Turing machine proceeds as follows:

1. We start the machine in an initial configuration where the first tape contains the input. For convenience we typically assume that this is bounded by blank characters that cannot appear in the input. The head on the first tape starts on the leftmost input symbol. All cells on all tapes, other than the cells representing the input, start off with a blank symbol. The controller starts in the initial state q_0.

2. Each step of the computation consists of reading the k-tuple of symbols under the heads on each of the tapes, then rewriting these symbols, updating the state, and moving the heads according to the transition function.

3. This continues until the machine halts, which we defined above as reaching a configuration that doesn't change as a result of applying the transition function.

3.1.2 Complexity

The **time complexity** of an execution is the number of steps until the machine halts. Typically we will try to bound the time complexity as a function of the **size** n of the input, defined as the number of cells occupied by the input, excluding the infinite number of blanks that surround it.

The **space complexity** is the number of tape cells used by the computation. We have to be a little careful to define what it means to use a cell. A naive approach is just to count the number of cells that hold a non-blank symbol at any step of the computation, but this allows cheating (on a multi-tape Turing machine) because we can simulate an unbounded counter by writing a single non-blank symbol to one of our work tapes and using the position of the head relative to this symbol as the counter value.[3] So instead we will charge for any cell that a head ever occupies, whether it writes a non-blank symbol to that cell or not.

[3] This counter abstraction only supports the operations increment, decrement, and test for zero, but two such counters are enough to simulate an unbounded ordinary memory using a construction of Minsky [Min61], and indeed Minsky's original construction is described in terms of a 2-tape TM with only one non-black cell per tape.

An exception is that if we have a read-only input tape and a write-only output tape, we will only charge for cells on the work tapes. This allows space complexity sublinear in n.

3.1.2.1 Asymptotic notation

In computing time and space complexities, we want to ignore constant factors and performance on small inputs, because we know that constant factors depend strongly on features of our model that we usually don't care much about, and performance on small inputs can always be faked by baking a large lookup table into our finite-state controller. As in the usual analysis of algorithms, we get around these issues by expressing performance using asymptotic notation. This section gives a brief review of asymptotic notation as used in algorithm analysis and computational complexity theory.

Given two non-negative[4] functions $f(n)$ and $g(n)$, we say that:

- $f(n) = O(g(n))$ if there exist constants $c > 0$ and N such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$.
- $f(n) = \Omega(g(n))$ if there exist constants $c > 0$ and N such that $f(n) \geq c \cdot (gn)$ for all $n \geq N$.
- $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, or equivalently if there exist constants $c_1 > 0$, $c_2 > 0$, and N such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq N$.
- $f(n) = o(g(n))$ if for any constant $c > 0$, there exists a constant N such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$.
- $f(n) = \omega(g(n))$ if for any constant $c > 0$, there exists a constant N such that $f(n) \geq c \cdot g(n)$ for all $n \geq N$.

Note that we are using the equals sign in a funny way here. The convention is that given an asymptotic expression like $O(n) + O(n^2) = O(n^3)$, the statement is true if for all functions we could substitute in on the left-hand side in each class, there exist functions we could substitute in on the

[4] This limitation is convenient for talking about time and space complexity, because we don't know how to make anything run using negative time or space. In other contexts, like in analysis (the branch of mathematics), it's common to allow $f(n)$ or $g(n)$ to be negative or even complex-valued, and just put in absolute values everywhere to make the definitions make sense.

right-hand side to make it true.[5]

Intuitively, it may help to think of the five asymptotic operators o, O, Θ, Ω, and ω as mapping to the five comparison relations $<$, $\leq$, $=$, $\geq$, and $>$. When we say $f(n) = o(g(n))$, we mean that $f(n)$ grows strictly more slowly than $g(n)$; $f(n) = O(g(n))$ means it grows no faster than $g(n)$; $f(n) = \Theta(g(n))$ means it grows at about the same rate; etc.; where in each case when we talk about rate of growth we mean the rate of growth ignoring constant factors and small inputs.

If you are familiar with limits, it is also possible to define $f(n) = o(g(n))$ as $\lim_{n\to\infty} f(n)/g(n) = 0$, and similarly $f(n) = \omega(g(n))$ as $\lim_{n\to\infty} g(n)/f(n) = 0$, with the caveat that bad things may happen if $f(n)$ or $g(n)$ are ever 0. This doesn't work so well for something like $f(n) = O(g(n))$. For example, the function

$$f(n) = \begin{cases} n & \text{when } n \text{ is odd} \\ 2n & \text{when } n \text{ is even} \end{cases}$$

is $\Theta(n)$ (it's always between n and $2n$), but $\lim_{n\to\infty} f(n)/n$ doesn't exist since $f(n)/n$ oscillates between 1 and 2.

When expressing complexities in asymptotic form, we usually try to keep the function inside the big O as simple as possible. This means eliminating constants and terms that are dominated by other terms. So complexities like $O(n)$, $O(n \log n)$, $O(n^5)$, and $O(2^n \cdot n^2 \log^5 n)$ are all things you are likely to see in actual papers, but $O(3n)$, $O(n^2 + n)$, and $O(2^n + n^2)$ are not.

3.1.3 Programming a Turing machine

Although one can in principle describe a Turing machine program by giving an explicit representation of δ, no sane programmer would ever want to do this. I personally find it helpful to think about TM programming as if I were programming in a C-like language, where the tapes correspond to (infinite) arrays of characters and the head positions correspond to (highly-restricted) pointers. The restrictions on the pointers are that we can't do any pointer operations other than post-increment, post-decrement, dereference, and assignment through a dereference; these correspond to moving the head left, moving the head right, reading a tape cell, and writing a tape cell.

[5] This particular claim happens to be true: to prove this, we would need to show that if $f(n) = O(n)$ and $g(n) = O(n^2)$, then there is some $h(n) = O(n^3)$ such that $f(n) + g(n) = h(n)$. The good news is that we don't have to guess what $h(n)$ is once we see $f(n)$ and $g(n)$. The bad news is that, after unpacking the definition of $O(-)$ in each case, we have at least six different constants to wrangle to show that $h(n) = O(n^3)$

q	read	q'	write	move
0	$\langle 0, 0, y\rangle$	1	$\langle 0, 0, y\rangle$	$\langle \mathsf{L}, \mathsf{S}, \mathsf{L}\rangle$
0	$\langle x, 0, y\rangle$	0	$\langle x, 0, y\rangle$	$\langle \mathsf{R}, \mathsf{S}, \mathsf{R}\rangle$
1	$\langle 0, 0, y\rangle$	2	$\langle 0, 0, y\rangle$	$\langle \mathsf{R}, \mathsf{S}, \mathsf{S}\rangle$
1	$\langle x, 0, y\rangle$	1	$\langle x, 0, y\rangle$	$\langle \mathsf{L}, \mathsf{S}, \mathsf{S}\rangle$
2	$\langle 0, 0, y\rangle$	2	$\langle 0, 0, y\rangle$	$\langle \mathsf{S}, \mathsf{S}, \mathsf{S}\rangle$
2	$\langle x, 0, y\rangle$	2	$\langle x, 0, x\rangle$	$\langle \mathsf{R}, \mathsf{S}, \mathsf{L}\rangle$

Table 3.1: Turing-machine transition function for reversing the input. In lines where x appears in the input tape, it is assumed that x is not zero. Note this is an abbreviated description: the actual transition function would not use variables x and y but would instead expand out all $|\Sigma|$ possible values for each.

The state of the controller represents several aspects of the program. At minimum, the state encodes the current program counter (so this approach only works for code that doesn't require a stack, which rules out recursion and many uses of subroutines). The state can also be used to hold a finite number of variables that can take on a finite number of values.

3.1.3.1 Example of computing a function

For example, Figure 3.1 is a C program that reverses its input tape to its output tape. The assumption is that blank symbols (including the ends of the input) are represented by null characters. [6] Because this program uses no local variables other than the head position pointers, the state only needs to represent the program counter. A representation of the corresponding transition function is given in Figure 3.1.

Realistically, nobody would ever write out either of these representations, unless they were really trying to be careful about counting states. Instead, a claim that a Turing machine can reverse its input would probably be described in terms of a less formal algorithm:

1. Move the input and output heads to the right until the input head reaches the end of the input.

[6] Note that this function won't work on standard C strings, which are (a) not null-terminated on both sides, and (b) not stored in arrays that are infinite in both directions. Objection (a) is easily dealt with by demanding null-termination at the start from the caller. Objection (b) can in principle be dealt with using a clever dynamically-allocated data structure and C++ overloading magic. As far as I know, nobody has ever bothered to do this.

```
void reverse(char *input, char *work, char *output)
{
    /* move output head to past last position of output */
    /* state 0 */
    while(*input != 0) {
        output++;
        input++;
    }

    /* pull output head left one cell to actual last position */
    /* start moving input head to leftmost input symbol */
    /* state 0 */
    output--;
    input--;

    /* return input head to leftmost input symbol */
    /* state 1 */
    while(*input != 0) {
        input--;
    }

    /* state 1 */
    input++;

    /* copy input to output in reverse order */
    /* state 2 */
    while(*input != 0) {
        *output = *input;
        input++;
        output--;
    }

    /* HALT */
}
```

Figure 3.1: Turing-machine transition function for reversing the input, disguised as a C program

2. Move the output head back one cell to the left.

3. Move the input head back to the leftmost cell of the input.

4. Copy the input to the output, one cell at a time, moving the input head right and the output head left after each step.

As long as it's clear that each of these steps only requires reading and writing cells under the current head positions, each move only moves a head at most one cell left or right, and the number of states needed to keep track of everything is finite,

3.1.3.2 Example of computing a predicate

Here's an example of a predicate computed by a Turing machine. Note that here we don't use the output tape. The halting state determines whether we accept the input or not.

1. Copy the input from the input tape to the work tape, leaving both the input and work tape heads on the blank cell to the right.

2. Move the work tape head back to the blank cell to the left of the copy of the input.

3. Finally, walk the work tape head right across the copy while walking the input tape head left across the input. If we see mismatched symbols before reaching a blank, halt and reject. If we reach the blanks at the end, halt and accept.

Note that most of the "steps" in this algorithm are not steps in the sense of a single move of the TM; instead, we rely on the reader's sense of how to program a TM to interpret statements like "Write $\$_{\mathsf{L}}$ one cell to the right of the rightmost $\$_{\mathsf{R}}$" in terms of a sequence of states and transitions that move the work-tape head to the appropriate location and update the cell. Doing this involves the usual wager in mathematical writing: the writer is betting both their own and the reader's time against the possibility of embarrassment if a procedure that has an obvious implementation does not in fact work.

In this case, the algorithm is simple enough that we can also write out the transition table: see Figure 3.2.

q	read	q'	write	move	note
0	$\langle x, - \rangle$	1	$\langle x, x \rangle$	$\langle \mathsf{R}, \mathsf{R} \rangle$	copy symbol
0	$\langle b, - \rangle$	1	$\langle b, b \rangle$	$\langle \mathsf{S}, \mathsf{L} \rangle$	stop copying
1	$\langle b, b \rangle$	2	$\langle b, b \rangle$	$\langle \mathsf{L}, \mathsf{R} \rangle$	start comparing
1	$\langle b, x \rangle$	2	$\langle b, x \rangle$	$\langle \mathsf{S}, \mathsf{R} \rangle$	move work tape head back to start
2	$\langle b, b \rangle$	3	$\langle b, b \rangle$	$\langle \mathsf{S}, \mathsf{S} \rangle$	reached end, move to accepting state
2	$\langle x, x \rangle$	2	$\langle x, x \rangle$	$\langle \mathsf{L}, \mathsf{R} \rangle$	when $x \neq b$; comparison OK, keep going
2	$\langle x, y \rangle$	2	$\langle x, y \rangle$	$\langle \mathsf{S}, \mathsf{S} \rangle$	when $x \neq y$; halt and reject
3	$\langle b, b \rangle$	3	$\langle b, b \rangle$	$\langle \mathsf{S}, \mathsf{S} \rangle$	halt and accept

Figure 3.2: Recognizing a palindrome using a work tape

3.1.4 Turing machine variants

A lot of early working on Turing machines and related models went into showing that each could simulate the others. Most of these simulations are not very exciting, but knowing they exist can sometimes simplify arguments about what can or cannot be computed by a Turing machine. The following lemma gives a few of the more useful reductions:

Lemma 3.1.1. *Each of the following models computes the same predicates and functions as a standard k-tape Turing machine:*

1. *A machine with a writable input tape and a readable output tape.*
2. *A machine that allows more than one head per tape.*
3. *A machine whose tapes have a left boundary, making them infinite in only one direction.*
4. *A machine that has only a single tape.*
5. *A machine with tape alphabet $\{0, 1\}$.*
6. *A machine that has no work tape, but instead has at least two counters supporting increment, decrement, and test-for-zero operations. (Equivalently, a Turing machine with at least two non-writable work tapes that each contain exactly one non-blank symbol.)*

Proof. 1. The trick here is to add two extra work tapes to the standard machine. The first extra tape holds a copy of the input, the second

a copy of the output. It is straightforward to modify the transition function (with a few extra states) so that the first thing the machine does is copy the non-writable input tape to the first extra work tape, and the last thing the machine does is copy the second extra work tape to the output tape. This incurs a time and space overhead linear in the combined size of the input and output, which is usually not a problem unless we are considering machines with very restricted space complexity.

2. For this we extend the tape alphabet to include markers for the simulated head positions, so that instead of Γ we are using $\Gamma \times \mathcal{P}([\ell])$ where $[\ell] = \{0, 1, \ldots, \ell - 1\}$ and ℓ is the number of heads per tape. We assume, based on the previous construction, that the input and output tapes are writable.

   To execute a step of the multi-head machine, we first send the real heads across each tape to collect all symbols under the simulated heads. This takes $O(T)$ time in the worst case, where T is the running time of the simulated machine. We then compute the new simulated state, moves of the simulated heads, and symbols written, and again send the real heads across each tape to make these updates. The actual mechanics of implementing this are pretty tedious, but it is straightforward to see that only finitely many extra states are needed to keep track of the finite vector of symbols that have been, the finite vector of symbols that need to be written, and which heads still need to move and in which direction. So this is something that a standard Turing machine can do.

   There is no overhead in space complexity (except possibly dealing with the issue of a non-writable input tape with multiple heads), but the time complexity of a computation can easily go from T to $O(T^2)$.

3. Here we take each standard doubly-infinite tape and fold it in half, turning it into a tape that infinite in only one direction.

   The idea is that the i-th cell of the folded tape holds cells at positions $\pm i$ on the original tape, with a special mark in cell 0 to indicate the left edge. If we use an extra bit for this mark, this requires expanding the tape alphabet from Γ to $\Gamma^2 \times \{0, 1\}$. We must also expand the state space to indicate for each head whether it is on the positive or negative half of the simulated tape, and adjust the transition function to keep track of this information, move each head in the appropriate direction, and update whichever side of the simulated tape is appropriate at each

step. These changes will still leave the size of the alphabet, state space, and transition table constant, and there is no slowdown or increase in space complexity as a result of the simulation.

4. To reduce from k tapes to a single tape, pack the tapes consecutively one after each other, with markers as in the previous construction for the k distinct heads. A simulated step consists of scanning the entire tape to find the values under the heads $O(T(n))$ steps, computing the result of applying δ, and then rewriting new values and moving the head markers. This last step takes $O(T(n))$ time if we don't have to increase the space used by any of the work tapes. If we do, we can copy values to the right of the places where we are expanding to make room; doing this from right-to-left requires $O(1)$ storage and $O(T(n))$ time. In each case the total cost of simulating one step of the k-tape machine is bounded by $O(T(n))$, and the total cost of simulating $T(n)$ steps is bounded by $O((T(n))^2)$. This turns out to be optimal, see §3.1.5.

5. To reduce to a two-character alphabet, encode the original alphabet in binary. Let Γ be the original alphabet, and let $k = \lceil \lg|\Gamma| \rceil$ be the minimum number of bits needed to represent each element of Γ uniquely. We will represent each cell of the simulated machine with k cells in the simulating machine, and represent each symbol in Γ with a distinct sequence of k bits (note that this requires cooperation from whoever is supplying our input!)

   At the start of each simulated step, we assume that each head is parked on the leftmost symbol of a k-wide block. To read the simulated symbols under the heads, we move the heads across the blocks collecting bits as they go, then move them back; this takes $2(k-1)$ steps and requires expanding the state space to track what we are doing, but the state space will still be finite. We then compute the transition and store the new symbols to write in the finite-state controller. A second $2(k-1)$-step walk across the blocks writes the new symbols. Finally, we take k steps to move the heads left or right k cells as determined by the simulated transition function.

6. To reduce a TM to two counters, we use a construction of Minsky [Min61].

   The first idea is that we can replace a doubly-infinite tape with a head in the middle with two stacks. To move the head right, we pop from the right stack and push to the left; to move left, we do the reverse.

Next, we can implement a stack with two counters, supporting increment, decrement, and test-for-zero operations. A stack containing symbols $x_0, x_1, \ldots$ is represented by the number $X = \sum_{i=0}^{\infty} s^i x_i$, where $s = |\Gamma|$ and we assume that a blank symbol is represented by 0 to keep the total finite. To read the top symbol x_0, we must compute X mod s, which we can do by copying the counter holding x to a second counter starting at 0 (using X many decrement and increment operations), and tracking the remainder mod s as we go. To pop the stack, we compute $\lfloor X/s \rfloor$ by incrementing the output counter once after every s decrements we manage to do on the input counter, throwing away any remainder at the end. To push a new symbol z, compute $X \cdot s + z$ by incrementing the output counter s times for each time we decrement the input counter, then add an extra z on the end. All of these operations can be managed by a finite-state controller, since we only need to be able to count up to s.

Applying both techniques turns k tapes into $2k$ counters. To reduce to just two counters, use Goedel numbering [Gö31, §V]: pick $2k$ distinct primes $p_1, p_2, \ldots, p_{2k}$ and encode the vector $\langle X_1, \ldots, X_{2k} \rangle$ as $Y = \Pi_{i=1}^{2k} p_i^{X_i}$. We can now test if any particular X_i is 0 by testing if Y is not divisible by p_i (which can be done by computing a remainder while copying Y from one counter to the other), and can increment or decrement X_i by multiplying or dividing by p_i. Again, everything requires a finite amount of state to manage.

The blow-up in time complexity for this simulation is exponential. Going from k tapes to $2k$ counters by itself makes each step cost as much as $O(ks^T)$ counter operations. Each counter operation in turn can take up to $O(p_{2k}^T)$ steps on the two-counter machine, where p_{2k} will be $\Theta(k \log k)$ (a constant!) if we are parsimonious in our choice of primes. Combining these gives a time per step of the original machine of $O(k(sp_{2k})^T)$, which argues for applying this technique only to machines with few tapes and small alphabets, if we have to apply it at all. On the other hand, it shows that even very limited (but unbounded) storage devices, together with a finite-state controller, can compute anything computed by a standard Turing machine or any model it can simulate. This makes general-purpose computation very easy to achieve if we have unbounded space and don't care about time too much.

□

3.1.5 Limitations of simulations

Except for tightly constrained models like Minsky's two-counter machines, most of the models we consider can simulate each other with at most polynomial blow-up. A typical case is our ability to simulate a k-tape Turing machine that runs in $O(T(n))$ time using a one-tape Turing machine that runs in $O((T(n))^2)$ time. In some of these cases, we can show that this blow-up is unavoidable.

For example, we have previously seen (§3.1.3.2 that it is possible to decide the language PALINDROME $= \left\{ x \;\middle|\; x = x^{\mathrm{R}} \right\}$ using a Turing machine with a work tape in $O(n)$ steps. A classic lower bound of Hennie [Hen65] shows that a one-tape machine must take $\Omega(n^2)$ steps to recognize this language.[7] We give an explanation of this result below.

Given a computation of a one-tape Turing machine M on input x, we can consider the sequence of steps that send us back and forth between cells i and $i+1$ on the tape. The crossing sequence $C_i(x)$ is defined as the sequence of states $q_1 q_2 \ldots q_k$ that the finite-state controller holds in each configuration just before moving from i to $i+1$ or from $i+1$ to i. The crossing sequence characterizes what information is carried from the left side of the tape to the right side of the tape and vice versa. When drawing a crossing sequence, we'll often put in arrows indicating which direction the head is moving at each point in the sequence, but this is redundant: we know that the head will be on the left side at the beginning, and each crossing changes which side it is on, so the odd positions in the sequence always correspond to left-to-right transitions and the even positions always correspond to right-to-left transitions.

The length of a crossing sequence $C_i(x)$ may depend on x, since different inputs may result in fewer or more crossings. What makes the crossing sequences useful is that $\sum_i C_i(x)$ is a lower bound on the number of steps taken by the Turing machine.[8] So showing that a computation takes a long time requires "only" showing that it has many long crossing sequences. We can do this for PALINDROME, as well as many similar languages like $\{xx\}$ where recognizing a member of the language requires comparing a lot of information on widely-separated parts of the input tape.

The key step is observing that a crossing sequence fully characterizes what information flows across the boundary between cells i and $i+1$. More

[7] Strictly speaking, Hennie shows [Hen65, Theorem 5] the same lower bound for the language $x2^n x$, where $x \in \{0,1\}^n$. Here the second copy of x is not reversed, but the lower bound argument is pretty much the same.

[8] It is not exact because the sum doesn't include steps where the head doesn't move.

explicitly, we show that if two inputs give the same crossing sequence, we can swap the parts of the input on either side of the boundary without changing the outcome of the computation:

Lemma 3.1.2. *Let M be a one-tape Turing machine. Consider two inputs st and uv, where $|s| = |u| = i$. Then if $C_i(st) = C_i(uv)$ and $M(st) = M(uv)$, then $M(sv)M(st)$.*

Proof. The idea is that we can paste together parts of the computation on st with parts of the computation on uv to get a computation on sv that behaves the same as st on the s side of the boundary and as uv on the v side. Divide the st computation into a sequence of $k = C_i(st) + 1$ fragments $\alpha_0\alpha_1 \ldots \alpha_k$, where the split between each α_j and the following α_{j+1} is at the point where the TM head crosses the i–$(i+1)$ boundary. Similarly divide the uv computation into fragments $\beta_0 \ldots \beta_k$.

We now argue that $\alpha_0\beta_1\alpha_2\beta_3 \ldots \gamma_k$ describes the sv computation, where γ_k is either α_k or β_k depending on whether k is even or odd. The argument is a straightforward induction on the step count so far, with the induction hypothesis stating that at the ℓ-th step of the j-th segment, the active side of the tape and the finite-state controller are both in the same state as in the corresponding step of the appropriate unmixed execution, and the passive side of the tape is in whatever state it would be at the end of the segment it just completed in its own unmixed execution. This is easily shown for transitions that don't cross the boundary. For transitions that cross the boundary, we appeal to $C_i(st) = C_i(uv)$ to argue that the finite-state controller's state is the same coming out of the active side as it should be going into the previously passive side.

It follows that when the machine halts on the s side, it halts in the same state as in $M(st)$; if it halts on the v side, it halts in the same state as in $M(uv)$. In either case it accepts or rejects as in the original unmixed execution. □

How does this help us with PALINDROME? Fix some machine M for recognizing PALINDROME, and consider the set S_n of inputs of the form $x0^n x^{\mathrm{R}}$, for all $x \in \{0,1\}^n$. Let $C_i(S_n) = \{C_i(y) \mid y \in S\}$. We will argue that $|C_i(S)|$ is large for all $i \in n \ldots n2-1$, and thus that most crossing sequences across these boundaries are also large. Since each element of each $C_i(y)$ crossing sequence corresponds to a step of $M(y)$, this will give us an $\Omega(n^2) = \Omega(|y|^2)$ lower bound on $M(y)$ for some $y = x0^n x^{\mathrm{R}}$.

To show $|C_i(S)|$ is large, suppose that there are strings x and y in $\{0,1\}^n$ such that $C_i(x0^n x^{\mathrm{R}}) = C_i(y0^n y^{\mathrm{R}})$. Since these two strings are both

palindromes, Lemma 3.1.2 says that we can split them at i and paste the results together to get a new string $x0^n y^{\mathrm{R}}$ that M accepts. This implies that $x = y$, making C_i injective on S. So $|C_i(S)| = |S| = 2^n$.

Because a crossing sequence is just a list of states, there are at most $|Q|^t$ possible crossing sequences of length t, and at most $\frac{|Q|^{t-1}-1}{|Q|-1} \leq |Q|^t$ crossing sequences of length strictly less than t. Let $t = \lfloor \frac{n-1}{\log|Q|} \rfloor = \Omega(n)$. Then $|Q|^t \leq 2^{n-1}$, and at least half of the y in S give crossing sequences $C_i(y)$ that have length at least $t = \Omega(n)$. If we choose an element y of S uniformly at random, the expected number of positions $i \in \{n \dots 2n-1\}$ for which $C_i(y) \geq t$ is at least $n/2$. It follows that there is some y such that $C_i(y) \geq t$ for at least $n/2$ values if i, giving a total length over all i of at least $tn/2 = \Omega(n^2)$.

3.1.6 Universal Turing machines

One of Turing's most striking observations about Turing machines was that even though any particular Turing machine has a fixed transition table, you can build a **universal Turing machine** U that simulates any other Turing machine M, given a description $\llcorner M \lrcorner$ of that machine on its input tape. This is true even if the simulated machine has a larger tape alphabet than U (although the input will need to be encoded so that U can read it) or uses more tapes.

Specifically, U is **universal** if $U(\llcorner M \lrcorner, \llcorner x \lrcorner) = M(x)$ for any Turing machine M and input x, where $\llcorner M \lrcorner$ and $\llcorner x \lrcorner$ are appropriate encodings of M and x in U's input alphabet.

By an appropriate encoding of M, we want something that specifies:

1. The size of M's state space Q, tape alphabet Γ, and input alphabet Σ.
2. The number of work tapes available to M.
3. The transition table for M. To simplify things, it's usually easiest to assume a standardized form of the transition table, where the states in Q are encoded as binary numbers in the range $0 \dots |Q|-1$, with 0 encoding the initial state q_0, and the alphabet Γ is similarly encoded as $0 \dots |\Gamma|-1$, with 0 representing the blank symbol, $0 \dots |\Sigma|-1$ representing the input alphabet.

Actually programming U is a bit of a nuisance, but if we are not too worried about time complexity, we can store M's work tapes consecutively on a single work tape, using the techniques from Lemma 3.1.1 to simulate

having a larger alphabet than U and separate heads for each simulated tape. This may require copying large section of U's work tape from time to time to expand the storage allocated to a particular simulated tape, but each of these copying operations will only take time $O(S \log|\Gamma|)$ where S is the space complexity of M's computation (which is bounded by the time complexity T of this same computation). To execute a step of M, we gather up the symbols under M's heads onto a second work tape (which we also use to store M's state), and then look for a matching element of $Q \times \Gamma^k$ in M's transition table. This requires scanning our entire storage tape, although for the input we can just use a copy of the original input tape.[9] We then copy the new state, cell contents, and head movements onto the second work tape, and finally run up and down the first work tape to rewrite cells and move the simulated heads. Any output is written directly to the output tape. All of this takes $O(|\llcorner M \lrcorner| + T \log|\Gamma|)$ time assuming the simulated work tapes are reasonably well packed together. The total time to simulate T steps of M is thus $O(CT^2)$ where C is a constant that depends M.

Using the clever amortized data structure of Hennie and Stearns (see Lemma 3.1.1), we can replace the consecutive representations of M's work tapes by interleaved representations and reduce the cost to $O(CT \log T)$, where C is again a constant that depends on M. This improvement in efficiency will turn out to be important when we look at the time hierarchy theorem in §6.1.2.

3.2 Random access machines

A traditional C programmer, presented with the modified version of the language from §3.1.3.1, might initially be overjoyed to realize that having infinite arrays means no possibility of segmentation faults. But their joy would turn to ashes quickly once we reveal that basic array operations like `a[i]` are denied them. If we want to market our models to programmers, we will need to give them more power. Typically this is done using some version of a **random access machine** (**RAM**).

A RAM looks a lot like a typical modern computer in that it has a controller with registers and a memory that looks like a giant unbounded array, except that (as in a Turing machine) its program is encoded directly into the transition table of its finite-state controller. Each register and each memory location holds an integer. While different RAM models use

[9] Using a copy means that we don't need to use the input head to mark where we are in x, and can instead use it to scan through $\llcorner M \lrcorner$.

different instruction sets, the basic idea is that we can do arithmetic on registers, compare registers, and load or store a value from a memory location addressed by a particular register all as a single step. Each such step is determined by the current state, and the transition function specifies the next step (possibly based on the outcome of a test in the case of comparing registers).

We can implement a RAM using a Turing machine by storing a binary representation of the registers and memory on work tapes. The simplest way to do this is probably to assign a separate work tape to each register (there are only finitely many); if this is too profligate, we can use the simulation from Lemma 3.1.1 to reduce to a single work tape. For the memory, we use a single work tape organized as an association list: a non-empty memory location i holding a value x is represented by a sequence $\llcorner i \lrcorner \rightarrow \llcorner x \lrcorner$ where $\llcorner i \lrcorner$ and $\llcorner x \lrcorner$ are binary representations of i and x and $\rightarrow$ is a separator. The list elements are themselves separated by a different separator.

Arithmetic operations on registers are implemented using standard Turing machine programs (possibly using an extra work tape or two). For addition, subtraction, and comparison, this will take $O(\log M)$ time when working on values x with $|x| \leq M$.

Memory operations more painful. To read a memory cell i, we have to scan the entire memory tape to find $\llcorner i \lrcorner$, then copy the corresponding $\llcorner x \lrcorner$ to the desired register tape. To write x to i, we must again scan for the current value of i (if any) and remove it by copying any subsequent association-list pairs down. We can then append the new pair $\llcorner i \lrcorner \rightarrow \llcorner x \lrcorner$ to the end of the list. Both of these operations take time linear in the length of the memory, which will be $O(T \log C)$ if C is an upper bound on the absolute value of any register during the computation. For typical computations, C will be polynomial in T, giving a slowdown of $O(T \log T)$.

The simulation in the other direction is trivial: given a Turing machine, we can assume (based on the simulations in Lemma 3.1.1 that it has a single, half-infinite tape. Store this in memory, one cell per memory location, and use a register to track the position of the head.

Even though RAMs are more natural to program than Turing machines, as a mathematical model they are a bit annoying. The big problem is that the state of a RAM is complicated, which makes it tricky to simulate a RAM using other models, and allowing unbounded values in the registers and memory makes defining space complexity tricky as well. So we will generally think of Turing machines as our fundamental model, and appeal to simulations like the one above (or the extended Church-Turing thesis more generally) to transform algorithms written for more powerful models into

something that can run on a TM.

A second objection to the RAM model is that the ability to access arbitrary memory locations in $O(1)$ steps is not physically realistic. Assuming each bit of memory requires some minimum volume of space (a few thousand cubic nanometers using current integrated circuit technology, or something related to the Planck length based on quantum mechanics), we can only pack $O(1)$ bits of memory within any constant distance of the CPU, and in general we can only pack $O(\ell^3)$ bits within distance ℓ. This means that accessing a memory of size S without faster-than-light communication will require $\Theta(S^{1/3})$ time in the worst case. Turing machines enforce a worse restriction implicitly, since we have to run the head down the tape. This makes a TM arguably a better representation of what a very large computer could do than a RAM.

3.3 The extended Church-Turing thesis

There are many other models of computation we could consider, but with the possible exception of quantum computers, all the ones we can currently imagine implementing fall under the **extended Church-Turing thesis**, which says that

Claim 3.3.1. *Any language that can be decided by a physically realizable computing device M in time $T(n)$ can be decided by a Turing machine in time $O(T(n)^k$ for some fixed k that depends only on M.*

In other words, all physically realizable computing devices are equivalent in power to a Turing machine, up to polynomial slowdown.

The argument for this is that we can imagine that any physically realizable computing device can be simulated by a 1977 TRS-80 Model I home computer equipped with an unboundedly large external storage device,[10] and using a construction similar to the one sketched for random access machines we can imagine that the TRS-80 Model I can be simulated by a Turing machine. So the original device can be simulated by a Turing machine. By itself this claim is just the **Church-Turing hypothesis**; the extended version says that this simulation involves only polynomial slowdown, which appears to be true for everything we've managed to come up with so far.

Note that polynomial slowdown means that things like $O(n)$ or $O(n^2)$ time may depend on our choice of computational model. But if we just talk

[10] If this seems implausible, substitute one of the Zoo computers equipped with an unboundedly large external storage device.

about polynomial time, this is robust against changes in the model. This is why we are so interested in classes like **P** (what we can decide in polynomial time) as opposed to classes like **TIME**(n) (what we can decide in linear time). But these less robust classes are still well-defined as long as we are careful to specify our model.

Chapter 4

Time and space complexity classes

Last updated 2017. Some material may be out of date.

A **complexity class** is a set of languages that are similarly hard in some sense. For example, the class **P** is the set of all languages that can be decided by a Turing machine in **polynomial time**, that is, in $O(n^k)$ time for some k.

To formalize the class of languages that can be decided within some time bound, we need a technical definition to exclude time bounds that produce weird results (say, by being themselves uncomputable). A function $f(n)$ is **time-constructible** if there is a Turing machine that, given input 1^n, computes $1^{f(n)}$ in $O(f(n))$ steps. Similarly, a function $f(n)$ is **space-constructible** if there is a Turing machine that, given input 1^n, computes $1^{f(n)}$ in $O(f(n))$ space. To avoid the issue of reading all the input, we generally also require $f(n) \geq n$ in both cases.

Given a time-constructible function $f(n)$, the complexity class $\mathbf{TIME}(f(n))$ consists of all languages L for which there exists a Turing machine M that decides L while always halting after at most $O(f(n))$ steps. Similarly, given a space-constructible function $f(n)$, the class $\mathbf{SPACE}(f(n))$ consists of all languages L for which there exits a Turing machine M that decides L while always using at most $O(f(n))$ space.

As observed previously, both $\mathbf{TIME}(f(n))$ and $\mathbf{SPACE}(f(n))$ may depend on the specific details of the computational model we are using. For example, it is know that recognizing a palindrome can be done in $O(n)$ time on a Turing machine with a separate input and work tape (easy exercise)

but requires $\Omega(n^2)$ time on a machine with just one tape. For this reason we often work with more robust classes.

The most important class of all, which is generally taken to correspond to what is computationally feasible, is the class

$$\mathbf{P} = \bigcup_{k=1}^{\infty} \mathbf{TIME}(n^k).$$

This consists of all languages that are in $\mathbf{TIME}(n^k)$ for some finite k.

The **extended Church-Turing thesis** says that **P** is robust in the sense that it contains the same languages for any reasonable model of computation. This is not a theorem (although it can be taken as a definition of a reasonable model); instead, it is a hypothesis that follows from the fact that all of the plausible-looking models of computation that have been invented over the years all have the ability to simulate each other up to polynomial slowdown.

Chapter 5

Nonterminism and NP

Last updated 2017. Some material may be out of date.

Historically, a **nondeterministic Turing machine** has been defined as one where the transition function is replaced a **transition relation**: instead of each configuration leading directly to a unique successor configuration, a configuration may have more than one successor configuration. In its simplest form, this means that the next state q' of the Turing machine controller is replaced by two next states q'_0 and q'_1, and the Turing machine can choose between them. In order to avoid having to think about how this choice is made, we imagine that in fact the machine makes *both* choices: we give it the magical ability to split into two copies, each with a different bit telling it what to do next. These copies can then split further into exponentially many copies, or **branches**. If any of the branches accepts, then we say the machine as a whole accepts; if none do, the machine rejects.

This definition is a little awkward to work with, so nondeterminism is now typically represented by giving a machine an extra input, the **certificate** or **witness**. This corresponds to the original definition by having the witness provide the sequence of choices that lead to the accepting branch (if there is one). But instead of having to imagine a ghostly parade of branches, we just think about a single computation that happens to get very lucky in the choice of witness.

Formally, a language L is decided by a nondeterministic machine M if, for every $x \in L$, there exists a witness w such that $M(x, w)$ accepts, and for every $x \notin L$, there does not exist a witness w such that $M(x, w)$ accepts.

This definition is not quite as symmetric as it looks: saying that $x \notin L$ means there is no witness is the same as saying that *all* w cause $M(x, w)$ to reject. So for a "yes" instance of L, one good witness (equivalently, one

accepting branch) is enough, but for a "no" instance, all witnesses must be bad (all branches must reject).

Analogous to **TIME**($f(n)$) and **SPACE**($f(n)$), we have have the nondeterministic time complexity classes **NTIME**($f(n)$) and nondeterministic space complexity classes **NSPACE**($f(n)$). These consist of all languages L that are decided by a nondeterministic Turing machine in $O(f(n))$ time or $O(f(n))$ space, respectively. Here the time or space complexity of machine M on input x is defined as the maximum time or space, respectively, taken by M over all choices of witness w. Note that $|w|$ does not count toward n, which is just $|x|$.

One complication with bounded space complexity is that the time complexity (and thus the number of branchings represented by the witness string) may be much larger than the space complexity (and thus how much of the witness the machine can remember). If we supply the witness on a standard two-way input tape, this allows the machine to go back and revisit its earlier choices in a way that might not be possible in the original branching version of nondeterminism. To avoid this, we supply the witness on a second read-only input tape, whose head can only move right. This also justifies, somewhat, not counting the length of the witness string as part of the input size n.

Like the deterministic time and space complexity classes, nondeterministic time and space complexity classes may depend on the specific details of the model being used. For this reason, we are generally most interested in nondeterministic classes that are more robust against changes in the model. The most important of these is

$$\mathbf{NP} = \bigcup k = 1^{\infty} \mathbf{NTIME}(n^k),$$

the set of all languages that can be decided in **nondeterministic polynomial time**. As far as we know, this class may or may not be the same as **P**, and the most important outstanding problem in complexity theory is showing whether or not $\mathbf{P} = \mathbf{NP}$.

There's an alternative definition of **NP** which puts the input and witness on the same input tape. Here we have to put a bound on the size of the witness to avoid, for example, ending up with a time complexity exponential in $|x|$ because an otherwise-useless w makes n too big. For this definition, we let $L \in \mathbf{NP}$ if there is a polynomial p and a polynomial-time M such that for all x, $x \in L$ if an only if there exists w of length $p(|x|)$ such that $M(x, w)$ accepts. Note that the running time of M is now polynomial in $|x| + |w|$, but this is still polynomial in $|x|$ because $|w|$ is.

This approach doesn't work as well for **NTIME**$(f(n))$ in general, because the size of w would have to be both smaller than n (if it is counted in n) and at least $f(n)$ (if it represents nondeterministic choices that can occur at each of $f(n)$ steps.

5.1 Examples of problems in NP

It's trivial to show that any language L in **P** is also in **NP**: take a poly-time machine $M(x)$ that decides $x \in L$, and convert it to a nondeterministic poly-time machine $M'(x, w)$ that decides $x \in L$ by the simple expedient of ignoring w. But there are a large class of problems that can easily be shown to be in **NP** that we don't know how to solve in **P**.

Typically these are problems where we are asked if some solution exists, and checking the solution (provided as the witness) can be done efficiently. What makes this nice from the point of view of the programmer is that finally we have a logical quantifier that is on our side. No longer must we face the worst-case input, supplied by our adversary $\forall x$, alone. Instead, our good friend $\exists w$ comes to our aid after the adversary makes its play.

For example, suppose we want to solve GRAPH 3-COLORABILITY. On an ordinary Turing machine, we could try out all possible colorings; but for an n-node, m-edge graph, there are 3^n of them. With a nondeterministic Turing machine, we simply summon $\exists w$ and demand it provide us with the correct coloring. This is trivial to check in $O(n + m)$ time, and if for some reason the existential quantifier betrays us, we will be able to recognize in that same time that w is no good. So the beauty of having a nondeterministic machine is that all the hard work of designing an actual algorithm is taken over by whoever provides w; we just need to be able to specify what a correct solution would look like, and write an efficient program to verify candidate solutions.

Many other problems have a similar structure. Want to know if a graph has an INDEPENDENT SET of size k? Have $\exists w$ guess the list of nodes in the independent set. Can your TRAVELING SALESMAN visit every node in a weight graph using a path of total weight W? Have $\exists w$ guess the path. Is your Boolean formula SATISFIABLE? Have $\exists w$ guess the satisfying assignment. In each case the problem of verifying that the guess is correct is straightforward, and we can easily argue that it can be done in polynomial (often only linear) time. So all of these problems are in **NP**. Which means that *if* **P** = **NP**, and we interpret membership in **P** as meaning a problem is easy, then all of these problems are easy. Sadly, there is a very strong chance

that they are not easy.

5.2 Reductions and NP-complete problems

A **polynomial-time many-one reduction** from a language L to a language L' is a deterministic polynomial-time computable function f such that $x \in L$ if and only if $f(x) \in L'$. If there exists a polynomial-time many-one reduction from L to L', we write $L \leq_{\mathbf{P}} L'$.

The idea behind writing a reduction as an inequality is that if we can efficiently reduce L to L', then L is no harder than L', because, given an efficient algorithm M that decides membership in L', then $M \circ f$ is an efficient algorithm that decides membership in L. The $\mathbf{P}$ subscript specifies what complexity class the reduction f lives in; in some cases, we will replace this with other classes to indicate different restrictions on f.

Proving a reduction $L \leq_{\mathbf{P}} L'$ generally involves 3 steps:

1. You have to come up with the mapping f and show that it runs in polynomial time.
2. You have to show that if $x \in L$, then $f(x) \in L'$.
3. You have to show that if $x \notin L$, then $f(x) \notin L'$; *or* that if $f(x) \in L'$, then $x \in L$. The second version is just the contrapositive of the first, but is sometimes easier to see how to do, especially when f is a one-to-one mapping that is easily inverted.

A language L is **NP-hard** if $L' \leq_{\mathbf{P}} L$ for any language L' in **NP**. A language L is **NP-complete** if it is both in **NP** and **NP**-hard. The **NP**-complete languages are the hardest languages in **NP**, in the sense that if we can recognize any **NP**-complete language in polynomial time, then $\mathbf{P} = \mathbf{NP}$.

If $\mathbf{P} \neq \mathbf{NP}$, then there are languages in **NP** that are not **NP**-complete (for example, all the ones in **P**). Under this assumption, there are even languages in $\mathbf{NP} - \mathbf{P}$ that are not **NP**-complete, but they are not very interesting languages.

The **NP**-complete languages, on the other hand, are very interesting: given any **NP**-complete L, the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ question is equivalent to $L \stackrel{?}{\in} \mathbf{NP}$. So instead of having to consider all possible languages in **NP**, it's enough to pick one particular **NP**-complete languages, and show that it either does or does not have a polynomial-time algorithm.

Alternatively, if we believe that $\mathbf{P} \neq \mathbf{NP}$, then this immediately tells us that any **NP**-complete language (more generally, any **NP**-hard language)

will not have a polynomial-time algorithm. Even if we aren't sure if $\mathbf{P} \neq \mathbf{NP}$, we still know that we have no examples so far of a polynomial-time algorithm for any problem that is **NP**-hard. So proving that a particular problem is **NP**-hard means that we can be reasonably confident that we won't find a polynomial-time algorithm for it without some surprising breakthrough.

5.3 The Cook-Levin theorem

None of this is useful unless we can point to an example of an **NP**-complete problem. The **Cook-Levin theorem** gives one such problem, called **3SAT**. This is the problem of testing whether there exists a satisfying assignment to a Boolean formula in **conjunctive normal form** (**CNF**) where each clause contains exactly three literals. A formula in this form is an AND of clauses, each of which is an OR of three variables x_i or their negations $\neg x_i$. The proof of the theorem is a construction that translates any problem in **NP** into such a 3CNF formula.

Theorem 5.3.1. *3SAT is* **NP***-complete.*

Proof. The essential idea is that we can encode the entire computation of an **NP** machine M on a given input x as a single gigantic (but polynomially large) SAT formula.

We'll start by setting up some variables:

Variable	Interpretation
Q_q^t	Finite-state controller is in state q at time t
H_{ij}^t	Head i is in position j at time t
T_{ijs}^t	Tape i, cell j, holds symbol s at time t

If the machine runs for T time, we get $O(T^3)$ variables total. For the tapes, we are relying on the fact that the infinite parts more than T away from the initial head positions don't need to be represented.

We now add some consistency conditions to the SAT formula. For example, to enforce that the machine is in exactly one state at any time, we include the OR-clauses $\bigvee_q Q_q^t$ and $\neg Q_q^t \vee \neg Q_{q'}^t$ for each t, q, and q'. Similar clauses enforce that each head is in exactly one position and each tape cell contains exactly one symbol. The total size of these clauses is again polynomial in T.

For the transition relation, we consider each combination of tape head positions, cell symbols under those positions, and finite-state controller state, and write a clause for each consequence of this combination. For example, if the machine has a single tape, and changes from state 0 to either state 1 or state 2 (it's nondeterministic) when it sees a b at location 5

on this tape, we could write this condition at each time t as $(Q_0^t \wedge H_{1,5}^t \wedge T^t 1, 5, b) \Rightarrow (Q_1^{t+1} \vee Q_2^{t+1})$, which conveniently transforms into the OR clause $\neg Q_0^t \vee \neg H_{1,5}^t \vee \neg T^t 1, 5, b \vee Q_1^{t+1} \vee Q_2^{t+1}$.

There are a lot of these clauses, but for a fixed number of tapes their total size is still polynomial in T.

Finally, we need to assert that the final state accepts (an OR over all accepting states q of Q_q^T) and that the initial configuration of the machine is correct (many one-variable clauses asserting Q_0^0, H_{i0}^0, $S_{0jx_j}^0$, S_{ijb}^0, etc.). The intermediate states are left unconstrained, except by the sanity clauses and transition-relation clauses already described.

If we can satisfy this enormous (but polynomial) formula, then there is a nondeterministic computation by the original machine on the given input that accepts. If we can't, there isn't. So we have successfully reduced the question of whether $M(x)$ accepts, and thus whether $x \in L(M)$, to SAT.

The last step is to replace each clause with more than three literals with a clause with three literals (or fewer, since we can always duplicate a literal to get to exactly three). The trick is that $x_1 \vee x_2 \vee \ldots \vee x_k$ is satisfiable if and only if both of $z \vee x_1 \vee x_2$ and $\neg z \vee x_3 \vee \ldots \vee x_k$ are, where z is a new variable introduced solely for the purpose of splitting up this clause. If z is true, at least one of x_3 through x_k must also be true, making the original clause true; if instead z is false, then at least one of x_1 or x_2 must be true. When we do this split, we new clauses of size 3 and $k-1$. Iterating until we get k down to 3 gets us a 3CNF formula. □

5.4 More NP-complete problems

Once we know that 3SAT is **NP**-complete, showing any other language L is **NP**-complete requires only (a) showing that $L \in$ **NP** and (b) showing that there is a reduction 3SAT $\leq_P L$. The reason why (b) is enough is that $\leq_P$ is transitive: given some language $L' \in$ **NP**, we take some candidate member x of L', run it through one polynomial-time function f to get and instance $f(x)$ of 3SAT, and then run $f(x)$ through another polynomial-time function g to get an instance $g(f(x))$ of L, and we will have $x \in L$ if and only if $g(f(x)) \in L$. The first function f exists because of Theorem 5.3.1. The second function g exists because we showed 3SAT $\leq_P L$.

We don't have to reduce from 3SAT to show L is **NP**-hard. Starting from any known **NP**-hard language L' also works. So it's helpful to have a few known **NP**-hard languages around to get this process started.

Many of the problems below are from Karp's 1972 paper [Kar72] extending

Cook's result for SAT to a variety of other combinatorial problems. (Note that the reductions may not be exactly the same.) A more extended source of core **NP**-complete problems is the classic book of Garey and Johnson [GJ79].

5.4.1 1-IN-3 SAT

An unfortunate feature of 3SAT is that each satisfied clause can have any of 1, 2, or 3 true literals. This turns out to be awkward when we try to reduce to problems that involve exact totals. Fortunately, we can show that 3SAT reduces to its more restrictive cousin 1-OF-3 SAT, defined as the set of all 3CNF formulas that have a satisfying assignment that makes exactly one literal in each clause true.

We do this by converting our original 3CNF formula one clause at a time. This involves adding a few extra variables specific to the representation of that clause.

To show how this works, let's start with $x_1 \vee x_2 \vee x_3$. We'll replace this clause with a new clause $y_1 \vee y_2 \vee y_3$, where $y_i \Rightarrow x_i$ but not necessarily conversely. These y_i variables are new, and we use a separate trio for each original clause. This allows us to pick just one of the true x_i literals to appear in the $y_1 \vee y_2 \vee y_3$ clause if there is more than one. To enforce that y_i can be true only if x_i is also true, we add three new 1-in-3 clauses, with three new variables: $\neg x_1 \vee y_1 \vee z_1$, $\neg x_2 \vee y_2 \vee z_2$, and $\neg x_3 \vee y_3 \vee z_3$. If any x_i is false, this makes $\neg x_i$ in the corresponding clause true, so y_i must also be false: so we can only satisfy $y_1 \vee y_2 \vee y_3$ if we satisfy $x_1 \vee x_2 \vee x_3$. But if x_i is true, we have a choice between making y_i true or z_i true. The z_i variables (which appear nowhere else) act as a sink for excess truth that would otherwise violate the 1-in-3 property.

Since this works for every clause, the output formula is in 1-IN-3 SAT if and only if the input formula is in 3SAT. Since the reduction is obviously polynomial (it's linear), this gives 3SAT $\leq_P$ 1-IN-3 SAT, making 1-IN-3 SAT **NP**-hard. But it's also in **NP**, since in polynomial time we can easily guess the satisfying assignment and check that it has the 1-in-3 property. So 1-IN-3 SAT is **NP**-complete.

5.4.2 SUBSET SUM and PARTITION

SUBSET SUM is the language $\{\langle x_1, x_2, \ldots, x_m, k\rangle \in \mathbb{N}^{m+1} \mid \exists a \in \{0,1\}^m \sum a_i x_i = k\}$. This is trivially in **NP** since we can just guess the vector of coefficients a, and add up the lucky values x_i with $a_i = 1$ to see if we get k.[1] It turns out

[1] This is an $O(n \log n)$ operation if n is the total length of the x_i in bits.

that it is also **NP**-hard, since we can reduce from 1-IN-3 SAT. This makes SUBSET SUM **NP**-complete.

Suppose we are given an instance of 1-IN-3 SAT with n variables $y_1, \ldots, y_n$ and m clauses $C_1, \ldots, C_m$. We will encode each variable y_i with two natural numbers x_i and x_i' written in base 4. The j-th digit of x_i is 1 if y_i appears in C_i and 0 otherwise, and the j-th digit of x_i' is 1 if $\neg y_i$ appears in C_i and 0 otherwise. We also set the $(n+j)$-th digit of both x_i and x_i' to one, and make k be the number written as $n+m$ ones in base 4.

The idea is that the extra ones shared between each x_i and x_i' force us to pick exactly one of them (corresponding to having to choose either y_i or $\neg y_i$ to be true), while the ones mapping to clauses force us to choose exactly one literal per clause to be true. This works because we never have more than 3 ones added together in a single position, so we get no carries in base 4, essentially reducing addition to vector addition.

For example, the suspiciously symmetric formula $(y_1 \vee \neg y_2 \vee \neg y_3) \wedge (\neg y_1 \vee y_2 \vee \neg y_3) \wedge (\neg y_1 \vee \neg y_2 \vee y_3)$, which happens to be in 1-IN-3 SAT because we can set all the y_i to be true, would be represented by the SUBSET SUM problem

$$\begin{aligned} x_1 &= 001001 \\ x_1' &= 001110 \\ x_2 &= 010010 \\ x_2' &= 010101 \\ x_3 &= 100100 \\ x_3' &= 100011 \\ k &= 111111 \end{aligned}$$

This SUBSET SUM problem has the solution $x_1 + x_2 + x_3 = 001001 + 010010 + 100100 = 111111 = k$, from which we can even read off the solution to the original 1-IN-3 SAT problem if we want to.

With a bit of tinkering, we can reduce SUBSET SUM to the even more terrifying **NP**-complete problem PARTITION. This asks, given a sequence of natural numbers $\langle y_1, \ldots, y_n \rangle$, if it is possible to split the sequence into two subsequences that add up to exactly the same total. Given an instance $\langle x_1, \ldots, x_n, k \rangle$ of SUBSET SUM, let $s = \sum x_i$. If $k \leq s$, construct the sequence $\langle x_1, \ldots, x_n, 4s - k, 3s + k \rangle$. This forces PARTITION to put $4s - k$ and $3s + k$ on opposite sides, because if both are on the same side, it will be at least $7s$ while the other side will be at most s even if we put all the x_i

there. But then each side must sum to half of $s + (4s - k) + (3s + k) = 8s$. We can only bring the side with $4s - k$ up to $4s$ by adding x_i to it that sum to k, which we can do if and only if the original SUBSET SUM problem has a subsequence of the x_i that sum to k.

I personally find this problem frightening because it seems like dividing a pile of numbers into two equal-sum piles should not be all that hard. To be fair, for reasonably-sized numbers, it isn't.[2] One of the things that happens along this path of reductions is that the numbers involved get awfully big, since each has a number of digits linear in the size of the output of the Cook-Levin reduction. This is polynomial in the size of the original problem input, but "I have a polynomial number of digits" is not something small numbers tend to say.

5.4.3 Graph problems

Many (but not all!) graph problems that ask if a graph has a subgraph with a particular property turn out to be **NP**-complete. It's trivial to show that such a problem is in **NP** as long as testing a subgraph for the property is in **P**, since we can just guess the winning subgraph. Showing that these problems are **NP**-hard usually requires an explicit reduction.

5.4.3.1 Reductions through INDEPENDENT SET

INDEPENDENT SET is the problem of determining, given G and k, whether G contains an independent set of size k. An **independent set** is a subset S of the vertices of G such that no edge has both endpoints in the subset. We can show INDEPENDENT SET is **NP**-hard by reducing from 3SAT.

The idea is that any clique in G can't contain more than one element of S, so if we can partition G into k non-overlapping cliques, then each of these cliques must contain exactly one element of S. We use this constraint to encode variable settings as 2-cliques (also known as edges): for each x_i, create nodes representing x_i and $\neg x_i$, and put an edge between them. We'll call these the master copies of x_i and $\neg x_i$.

We can then represent a clause $C_j = x \vee y \vee z$ as a 3-clique of copies of x, y, and z (which may be negated variables). Here the member of S in the representation of C_j indicates which of the three literals we are using to

[2] There is a simple dynamic-programming algorithm that solves SUBSET SUM in time $O(nk)$, which looks polynomial but isn't, since the value of k can be exponentially large as a function of its size in bits.

demonstrate that C_j is satisfiable. These are per-clause copies; we make a separate vertex to represent x in each clause it appears in.

The remaining step for the graph is to make sure that whatever literal we chose to satisfy in C_j is in fact assigned a true value in the 2-clique representing the corresponding x_i or $\neg x_i$. We do this by adding an extra edge from the per-clause copy in C_j to the master copy of the opposite value in the 2-clique; for example, if we have a copy of x_i in C_j, we link this copy to the node representing $\neg x_i$, and if we have a copy of $\neg x_i$ in C_j, we link this copy to the node representing x_i.[3]

Finally, we set $k = n + m$, where n is the number of variables (and thus the number of master-copy 2-cliques) and m is the number of clauses (and thus the number of clause 3-cliques). This enforces the one-element-per-clique requirement.

It is easy to see that this reduction can be done in polynomial (probably even linear) time. It remains to show that it maps satisfiable formulae to graphs with independent sets of size k and vice versa.

Let's start with a satisfiable formula. Put the master copy of x_i in the independent set if x_i is true, otherwise put $\neg x_i$ in. This gets us one element per 2-clique. For the clauses, pick some literal in each clause that is assigned the value true and put its copy in the independent set. This gets us our one element per 3-clique, putting us up to k.

However, we still have to worry about marking both endpoints of an edge that crosses between cliques. For each such edge, one of its endpoints is a copy of some x_i, and the other of $\neg x_i$, and we can only put these copies in the independent set if the corresponding variable is true. Since x_i and $\neg x_i$ can't both be true, we are fine.

In the other direction, suppose we have an independent set of size k. We can read off the corresponding variable assignment directly from the 2-cliques. For each clause C_j, there is an independent set element corresponding to some literal in that clause. But we know that this literal is true because it is linked to the master copy of its negation. So every clause is satisfied by at least one literal and the formula is satisfiable.

Knowing that INDEPENDENT SET is **NP**-hard instantly gets us some closely-related problems. These are **CLIQUE**, the problem of determining given (G, k) whether G contains a clique of size k, which reduces from INDEPENDENT SET by replacing the input graph by its complement; and

[3] Essentially the same construction works for SAT, since we just replace the 3-clique for clause C_j with a $|C_j|$-clique, but it doesn't change the ultimate result reducing from 3SAT instead, and it saves worrying about the size of the clauses.

VERTEX COVER, the problem of determining given (G, k) whether G contains a set of k vertices such that every edge is incident to at least one vertex in the set, which is equivalent to asking G contains an independent set of size $n - k$. In both cases the reduction is straightforward and obviously polynomial.

5.4.3.2 GRAPH 3-COLORABILITY

The GRAPH 3-COLORABILITY language consists of all (undirected) graphs that are **3-colorable**: there exists an assignment of colors to vertices so that only three colors are used and no edge has the same color on both endpoints.

It's easy to see that GRAPH 3-COLORABILITY is in **NP**: just guess the coloring. The tricky part as usual is finding a reduction from a known **NP**-hard problem. We will use 3SAT for this.

The main technique is a widget that allows us to build logic gates out of colorings. Consider this graph:

```
A--C
   |\
   | E
   |/
B--D
```

Suppose that we want to color the graph using colors red, green, and blue, and we happen to color both A and B red. Exactly one of C, D, and E must be red, and since E is the only node not adjacent to the two red nodes A and B, E must also be red.

Alternatively, if at least one of A or B is not red, then E doesn't have to be red either.

If we call red false, we've just built an OR gate.[4] Make E one of the inputs to a second copy of the widget, and we get an OR gate over three inputs, enough to handle a single clause of a 3SAT formula.

Colorings are symmetric with respect to permutations of the colors, so we can't actually insist that red is false or green is true. But what we can do is put in a triangle rgb somewhere that acts as a master color wheel, where we just call the color applied to vertex r red, on vertex g green, and on vertex b blue. We can then build variables that are guaranteed to have x be red

[4] Sort of. If red is false and green is true, having one false and one true input allows any of the three possible colors on the output. But for 3SAT it will be enough to force the output to be false if both inputs are false.

and $\neg x$ green or vice versa by building a triangle $x, \neg x, b$ for each x in the formula, where b is the ground vertex from the color wheel.

The last step is to hook up the variables to the OR gates. For each clause $x \vee y \vee z$, build a 3-input OR widget and make its input vertices coincide with vertices x, y, and z. To force the output not to be red, run an edge from the output vertex to vertex r on the color wheel. Now there exists a coloring of the graph if and only if we can assign colors red and green to each x and $\neg x$ so that no 3-input OR widget has all-red inputs. This translates back into being able to assign values false and true to each x and $\neg x$ so that no clause has all-false inputs, which is equivalent to the original formula being satisfiable. We have thus reduced 3SAT to GRAPH 3-COLORABILITY, which completes the proof that GRAPH 3-COLORABILITY is **NP**-complete.

5.5 coNP and coNP-completeness

The class **coNP** consists of the **complements** $\overline{L}$ of all languages L in **NP**. By complement we mean that $x \in \overline{L}$ if and only if $x \notin L$.[5]

We don't know if **coNP** = **NP** or not. If **coNP** = **P** or **NP** = **P**, then **coNP** = **NP** = **P**, since P is closed under complement. So for **coNP** $\neq$ **NP**, we do need **P** $\neq$ **coNP**. However, it is possible for **coNP** = **NP** $\neq$ **P**, although this does collapse the polynomial-time hierarchy (see §8.1).

A language L is **coNP-complete** if L is in **coNP** and $L' \leq_{\mathbf{P}} L$ for every L' in **coNP**. It is not hard to show that L is **coNP**-complete if and only if $\overline{L}$ is **NP**-complete. This instantly gives us many unnatural-looking **coNP**-complete problems like GRAPH NON-3-COLORABILITY and a few natural-looking ones like TAUTOLOGY (the complement of SATISFIABILITY).

We can also characterize problems in **coNP** in terms of witnesses, but the quantifier changes: L is in **coNP** if there is a polynomial-time M and a polynomial $p(|x|)$ such that $x \in L$ if and only if $\forall w \in \{0,1\}^{p(|x|)}$, $M(x, w)$ accepts. An equivalent definition in terms of a nondeterministic Turing machine is L is in **coNP** if there is some nondeterministic Turing machine

[5] A technical issue here is what to do with "junk" inputs that don't actually map to an instance of L because of problem with the encoding. For any reasonable encoding, we can detect a junk input in polynomial tie and reject them in our L machine, but it seems wrong to then accept them with our $\overline{L}$ machine. So we will treat the complement as the complement with respect to all correctly-encoded inputs. This won't affect any results about **coNP**, under the assumption that we can recognize junk inputs efficiently, since we can always map junk inputs to L to some specific rejecting input to $\overline{L}$.

for which *every* branch accepts an input $x \in L$ and at least one branch rejects an input $x \notin L$.

5.6 Relation to EXP

The class $\mathbf{EXP} = \bigcup k = 1^{\infty}\mathbf{TIME}(2^{n^k})$ consists of all languages decidable in **exponential time**. Similarly, the class $\mathbf{NEXP} = \bigcup k = 1^{\infty}\mathbf{TIME}(2^{n^k})$ consists of all languages decided by a nondeterministic Turing machine in exponential time.

It's easy to show that $\mathbf{NP} \subseteq \mathbf{EXP}$, since we can just try all 2^{n^k} witnesses.

A more indirect connection between the classes is that if $\mathbf{EXP} \neq \mathbf{NEXP}$, then $\mathbf{P} \neq \mathbf{NP}$. This makes more sense if we consider the contrapositive: $\mathbf{P} = \mathbf{NP}$ implies $\mathbf{EXP} = \mathbf{NEXP}$. The reason is that we can take any language L in **NEXP**, replace it with the **padded** language $L' = \left\{x; 1^{2^{|x|}} \;\middle|\; x \in L\right\}$. So now if L can be decided by a nondeterministic Turing machine in time $O\left(2^{|x|^k}\right)$, then L' can be decided by a nondeterministic Turing machine in time $O\left(|x; 1^{2^{|x|}}|^k\right)$, putting L' in **NP**. Under the assumption $\mathbf{P} = \mathbf{NP}$, this also makes L' decidable by a deterministic Turing machine in time polynomial in $2^{|x|}$, or exponential in $|x|$. But a machine in **EXP** can simulate such a Turing machine running on just x by first copying x to a work tape and then constructing the suffix $1^{2^{|x|}}$ for itself. This puts L in **EXP** and shows $\mathbf{EXP} = \mathbf{NEXP}$.

Note this doesn't necessarily work in the other direction: for all we know, $\mathbf{P} \neq \mathbf{NP}$ but $\mathbf{EXP} = \mathbf{NEXP}$. The problem is that while we can always pad a short input to make it a long one, we can't "unpad" a long input to make it a short one.

Chapter 6

Diagonalization

Last updated 2017. Some material may be out of date.

The basic idea of **diagonalization** goes back to Cantor's argument that there is no surjective map from any set S to its powerset $P(S)$ [Can91]. The proof is that, given any map $f : S \to P(S)$, the set $T = \{x \in S \mid x \notin f(x)\}$ cannot equal $f(x)$ for any x without producing a contradiction. (If $T = f(x)$, and $x \in T \leftrightarrow x \notin f(x)$, then $x \in T \leftrightarrow x \notin T$.) A similar trick was used by Gödel to to prove his incompleteness theorems in logic [Gö31], and by Turing to prove undecidability of the Halting Problem [Tur37]. In each case the idea is to consider some infinite sequence of candidate objects alleged to have some property, and then carefully construct a new infinite object that shows that none of them in fact have that property. We'll start with Turing's version and then show how to use a similar trick to get impossibility results for various classes of resource-constrained Turing machines.

6.0.1 Undecidability of the Halting Problem

The **Halting Problem** asks whether it is possible to construct a Turing Machine H that, given a representation $\llcorner M \lrcorner$ of a Turing machine M, and an input x, accepts if $M(x)$ halts and rejects if $M(x)$ runs forever. Turing's argument involves constructing a machine that provably does the opposite of what H predicts [Tur37].

This is the equivalent of demonstrating that your local fortune-teller can't really predict the future because they can't tell whether the next thing you say is going to be "yes" or "no." In both cases, the trick only works if you can wait for the predication before choosing what to do. But if H exists, we can do this.

The bad machine M we are going to construct takes as input a description $\llcorner M' \lrcorner$ of some machine M', runs $H(\llcorner M' \lrcorner, \llcorner M' \lrcorner$, then halts if and only if $H(\llcorner M' \lrcorner, \llcorner M' \lrcorner$ rejects. It's not hard to see that we can implement M given H, since the extra work is just a matter of copying the input twice, and instead of halting when H does, using the halting state of H to decide whether to really halt (if H rejects) or not to halt at all (if H accepts), perhaps by moving to a state that just moves one of the tape heads off to the right forever.

So now what happens if we run $H(\llcorner M \lrcorner, \llcorner M \lrcorner)$? This should accept only if and only if $M(\llcorner M \lrcorner)$ halts. But $M(\llcorner M \lrcorner)$ halts if and only if $H(\llcorner M \lrcorner, \llcorner M \lrcorner)$ rejects. So we've constructed a specific machine M and input $\llcorner M \lrcorner$ where H gives the wrong answer. So we have:

Theorem 6.0.1 (Halting Problem)**.** *There does not exist a Turing machine H that always halts, such that $H(M, x)$ accepts if and only if $M(x)$ halts.*

In other words, the Halting Problem is **undecidable**—there is no Turing machine that decides it.

This turns out to have consequences that go beyond just testing if a machine halts or not. Just as polynomial-time reductions from know **NP**-hard problems can show that other problems are **NP**-hard, computable reductions from the Halting Problem can show that other problems are also undecidable.

A very general result that follows from this is **Rice's Theorem**. This says that testing any non-trivial **semantic** property of a Turing machine is also undecidable, where a semantic property depends only whether the machine halts or produces a particular output for each input, and not on the details of the computation, and a property is non-trivial if there is at least one machine for which it holds and at least one for which it doesn't.

Corollary 6.0.2 (Rice's Theorem [Ric53])**.** *Let P be a non-trivial semantic property of Turing machines. Then P is undecidable.*

Proof. Suppose P is true for a machine that never halts. Let M_0 be a machine for which P is false (such a machine exists because P is non-trivial). Suppose there is a machine M_P that decides P. We can use M_P as a subroutine to solve the halting problem.

Let $\langle M, x \rangle$ be a machine-input pair for which we want to know if $M(x)$ halts. Construct a machine M' that takes an input y and runs M on x. If M doesn't halt, neither does M', and so M' has property P. If M does halt, M' switches to running M_0 on y, producig the same output (or failing to

halt on the same inputs) as M_0, giving M' property P. So to decide if $M(x)$ halts, construct M' and run M_P on it. The Turing machine that does this then solves the Halting Problem, contradicting Theorem 6.0.1.

If P is false for a machine that never halts, pick a machine M_1 for which P is true, and apply essentially the same construction. □

Note that Rice's Theorem is only about inputs and outputs. There are non-semantic properties (like "does this machine run in polynomial time on input x?") that are easily decidable even though they are not non-trivial. It also only works because we consider machines that might not halt.

6.1 Hierarchy theorems

In complexity theory we don't care too much about machines that might not halt, because all of our complexity classes only include machines that always halt. But we can use the essentially the same proof as for the Halting Problem to show that there are functions that cannot be computed within a given space or time bound. The tricky part is showing that having a bit more space or time makes these functions computable.

For both theorems the idea is to build a language L consisting of machines with inputs reject within some resource constraint, show that deciding L in less than the resource constraint gives a contradiction, then show that with more resources we can use a universal TM to decide L. Since we didn't do universal machines yet, we'll have to describe them now.

We'll start with the Space Hierarchy Theorem, because the construction is less fiddly.

6.1.1 The Space Hierarchy Theorem

The **Space Hierarchy Theorem** says that getting more than a constant factor more space is enough to solve more problems:

Theorem 6.1.1. *If $g(n)$ is a space-constructible function that is $\Omega(\log n)$, and $f(n) = o(g(n))$, then* $\mathbf{SPACE}(f(n)) \subsetneq \mathbf{SPACE}(g(n))$.

To prove this, we need to find a language L that is in $\mathbf{SPACE}(g(n))$ but not $\mathbf{SPACE}(f(n))$. The following language will have this property:

$$L = \{\langle \llcorner M \lrcorner, x\rangle \mid M \text{ rejects } \langle \llcorner M \lrcorner, x\rangle \text{ using at most } g(n) \text{ space, one work tape, and } |\Gamma| = 2\}$$

The restriction on M's work tape alphabet Γ avoids some annoyances in trying to simulate machines with larger alphabets.

First we'll show that $L \notin \mathbf{SPACE}(f(n))$. Let R be a machine that supposedly decides L using $O(f(n))$ space. From Lemma 3.1.1 there is another machine R' that produces the same output as R in $O(f(n))$ space (with a bigger constant) using one work tape and a two-bit alphabet (not counting the blank symbol). We'll use R' to get a contradiction.

Let x be any string long enough that the space complexity of the execution of R' on $\langle \llcorner R' \lrcorner, x \rangle$ is less than $g(n)$. We know that such a string exists, because R' takes $O(f(n))$ time, and whatever the constants hiding in the O this must drop below $g(n)$ for sufficiently large n. Now let us ask what $R'(\langle \llcorner R' \lrcorner, x \rangle)$ returns.

If it accepts, then $R'(\langle R', x \rangle)$ does not reject and so $\langle \llcorner R' \lrcorner, x \rangle$ is not in L: R' (and thus R) computes the wrong answer.

If it rejects, then $R'(\langle R', x \rangle)$ rejects using at most $g(n)$ space. By construction, R' also uses one work tape and two non-blank work tape symbols. So $\langle R', x \rangle$ is in L. Again, we get the wrong answer. It follows that R gives the wrong answer for at least one input $\langle M, x \rangle$, and so R does not decided L.

Next, we'll show that $L \in \mathbf{SPACE}(g(n))$. This requires constructing a **universal Turing Machine** to simulate M on x. We also need to be able to detect if M uses more than $g(n)$ space, or violates its parole in some other way.

Our machine R^* will use several work tapes:

1. A tape of size $O(\log n)$ to store a binary representation of M's state. The argument for $O(\log n)$ being enough is that $|\llcorner M \lrcorner|$ will be at least linear in the number of states, because of δ.

2. An input pointer tape of size $O(\log n)$. This will store a binary representation of the offset into x corresponding to the position of the simulated input tape head. We need this because we will be using our real tape head to run back and forth between reading $\llcorner M \lrcorner$ and $\llcorner x \lrcorner$, and we can't write to the input tape to mark our position.

3. A second pointer tape that remembers the index of the current head position on the input, relative to the start of x. We need this to know when we have reached the right position to match the first pointer.

4. A copy of M's work tape. We don't have to do anything clever with this, since we can just use our own head to keep track of M's head.

5. A tape of size $g(n)$ holding (after an initial computation) the string $1^{g(n)}$. Whenever we move the simulated work tape head, we'll move this tape's head as well, so if it goes off the end of the string, we'll

know that M is trying to use too much space. The very first thing our simulation does before starting up M is to compute $1^{g(n)}$, which we can do (possibly using an addition $g(n)$ space somewhere, although I think we can borrow the work tape for this) since $g(n)$ is space-constructible.

6. A binary counter of size $g(n) + \log|Q| + \lceil \log g(n) \rceil + \lceil \log n \rceil + 1 = O(g(n) + \log n)$ initialized to all ones. We use this to avoid loops; if we try to decrement the counter and find that it is

Everything else is constant size so we can store it in the finite-state controller. Simulating a step of M consists of:

1. Moving the input head across x until the two input pointers match, and collecting the input symbol.

2. Moving back to $\llcorner M \lrcorner$ and searching for an entry in $\llcorner \delta \lrcorner$ that matches (a) the state $\llcorner q \lrcorner$ on the state tape, (b) the input symbol stored in the finite-state controller, and (c) the work-tape symbol under the head on the simulated work tape. Having found this entry, we copy the new state q' to the state tape, do a write and move on the work tape if we need to, and increment or decrement the input-tape-head pointer as needed to simulate moving the input tape.

3. Decrementing the binary counter. If the counter is already at 0 or the head on the space-bound tape moves onto a blank, reject. In the first case, the machine has run for at least $2 \cdot 2^{g(n)} g(n) n |Q|$ steps, meaning that somewhere during its execution the contents of its work tape, the position of its work tape head, the position of its input tape head, and its state have repeated: it's looping and will never reject. In the second case, it used too much space.

4. If after many such steps the simulation rejects, accept. If it accepts, reject.

This might take a while (I think we can get it each simulated step down to $O(g(n) \log n)$ time if we are careful, and the initialization looks like it takes $O(g(n))$ time altogether), but the important thing is that at no time do we use more than $O(g(n) + \log n)$ space, and R^* decides L. So $L \in \mathbf{SPACE}(g(n) + \log n) = \mathbf{SPACE}(g(n)$ assuming $g(n) = \Omega(\log n)$.

6.1.2 The Time Hierarchy Theorem

The **Time Hierarchy Theorem** is similar to the Space Hierarchy Theorem, but the gap is wider:

Theorem 6.1.2. *If $g(n)$ is a time-constructible function, and $f(n) = o(g(n))$, then* $\mathbf{TIME}(f(n)) \subsetneq \mathbf{TIME}(g(n) \log g(n))$.

By analogy to the proof of the Space Hierarchy Theorem, a first try at a language for this one would be

$$L = \{\langle \llcorner M \lrcorner, x\rangle \mid M \text{ rejects } \langle \llcorner M \lrcorner, x\rangle \text{ using at most } f(n) \text{ time and a tape alphabet of size } 2\}.$$

This will give us something to start with, but getting the full-blown theorem will require some more tinkering. The main issue is that building a time-efficient universal Turing machine is harder than building a space-efficient one (this also accounts for the extra $\log g(n)$ factor), and we've learned from the SHT proof that the diagonalization side of the argument is pretty robust to small changes in L, so it will be helpful to adjust the definition of L a bit after we see the hard parts of the upper bound part of the argument in order to make that part easier.

First, though, the diagonalization part. Suppose R decides L, and R runs in $o(g(n))$ time, then running $R(\langle \llcorner R \lrcorner, x\rangle)$ gives a contradiction when x is large enough that $R's$ running time drops below $g(n)$ exactly. If $R(\langle \llcorner R \lrcorner, x\rangle)$ rejects in such a case, then $\langle \llcorner R \lrcorner, x\rangle$ is in L (by the definition of L), meaning that R just gave the wrong answer on this input. But the same thing happens if it rejects. So we get a contradiction either way, and R either does not decide L or it doesn't run in $o(g(n))$ time.

But now we need to show that a machine that does decide L runs in a reasonable amount of time. We'll start with a simple, direct simulation of the input machine M, and then worry about improving the running time later.[1]

Here's the easy approach: Build a machine R^* that has:

1. A tape storing x. We'll copy x to this tape at the start, costing time $O(n)$. This exists mostly so we can have one tape head pointing into $\llcorner M \lrcorner$ and another (on a separate tape) into x, so we don't have to waste time moving back and forth between two simulated heads.

2. A tape storing M's k work tapes. As in the SHT proof, we'll just store these consecutively, with markers for the k heads.

3. A tape storing the input to δ. This will be a state of M expressed in binary, plus one symbol for each of the $k + 1$ work and input tapes. The total size will be $O(\log|Q_M| + k)$.

[1] This approach is inspired by some lecture notes from Luca Trevisan.

4. A tape storing the output of δ. Pretty much the same as the previous tape, but we also need k symbols from $\{\mathsf{L}, \mathsf{S}, \mathsf{R}\}$ to track how the heads move.

5. A "fuse" tape that holds $01^{g(n)}$. It costs $O(g(n))$ time to set this up, leaving the head on the rightmost cell. Every time we simulate a step of M, we move the head one cell to the left, and when it hits the 0, we know that M took more than $g(n)$ steps and we can reject.

A step of R^* is also similar to a step in the SHT proof: we gather up the input to δ ($O(kg(n))$ time), scan through $\llcorner M \lrcorner$ to find the matching transition ($O(|\llcorner M \lrcorner|) = O(n) = O(g(n))$ time), copy the output to the result tape ($O(n)$ time), and then scan through the work tape to update everything ($O(k^2(g(n))^2)$ time, including the cost of copying cells up to $O(k)$ positions to the right to make room for new cells). Somewhere in here we also check the fuse (free, since we can move the fuse head in parallel with something we are doing anyway).

The total cost per step is $O(k^2 g(n)) = O(n^2 g(n))$, using n as a very crude upper bound on k. Since we have to simulate $O(g(n))$ steps, we've shown that $\mathbf{TIME}(o(g(n))) \subsetneq \mathbf{TIME}(n^2(g(n)))$. This is something, but we can do better.

If we look at the expensive parts of the simulation, the big costs we need to knock down are (a) the $O(kg(n))$ cost to traverse the work tape, and (b) the $O(n)$ cost to traverse δ. For (a), we will use a clever data structure appearing in Hennie and Stearns original THT paper [HS66] to knock the amortized cost per operation down to $O(k \log g(n))$, and make a small tweak to L to get rid of the extra k. There's not much we can do about (b), so we will get around this by another tweak to L to insist that $|\llcorner m \lrcorner|$ is small enough relative to n that the overhead is dominated by the $O(\log g(n))$ per step that we are already paying.

Here is the data structure. The description below follows the presentation in [AB07, §1.A].

We'll first map the k work tapes onto one tape by interleaving: for example, with 3 tapes, we will store their cells as 123123 etc. This means that if we park a head on the leftmost of k cells, we can reach any of the other cells is $O(k)$ time, assuming those tapes' heads are also in the same place. Unfortunately, they probably won't be after the first step.

We deal with this by moving the contents of the tape instead of the head. Doing this naively (for example, shifting every cell on tape 2 one position to the left) will be expensive. So instead we use a representation with gaps in it that we use to store values that we are pushing in one direction, empty out

increasingly large regions in bulk when we need more space. We'll describe how to do this for a single tape, and then apply the interleaving idea to the representation to handle multiple tapes.

The central cell of the tape we will just store by itself. The right-hand side we divide into zones $R_1, R_2, \ldots$, where zone R_i contains 2^i cells. Similarly, the left-hand side is divided into $\ldots, L_2, L_1$. Each zone is either empty (filled with a special not-in-use symbol), half-full (half-filled with that symbol and half with simulated tape symbols), or full (no not-in-use symbols at all). We maintain the invariant that the j-th cell of the L side is full if and only if the j-th cell of the R side is empty: this means that when L_i is empty, R_i is full, etc.

To pop a symbol from R, we look for the leftmost zone R_i that is not empty. This will either contain 2^i symbols, in which case we pop one of them and use the remaining $2^i - 1$ symbols to make all of R_1 through R_i half-full; or it will contain 2^{i-1} symbols, in which case we pop one, use the remaining $2^{i-1} - 1$ symbols to make R_i through R_{i-1} half-full and leave R_i empty. In either case we can do the copying (using an extra work tape) in $O(2^i)$ time. A push is symmetric (and must be because of our invariant): we look for the first non-full zone, fill it either all the way to full or to half-full as needed to clear out earlier zones to half-full, and then add the symbol to R_1.

The important thing is that once we fill in or clean out R_i, we don't do anything to it again until we either fill up or completely empty all zones R_1 through R_{i-1}. So our $O(2^i)$-cost operations on R_i can only happen every $O(2^i)$ steps, giving an amortized cost $O(1)$ per step per zone. We have $O(\log g(n))$ zones, so the cost per step of simulating one tape is $O(\log g(n))$ and of simulating k tapes is $O(k \log g(n))$, giving a total cost across the entire computation of $O(kg(n) \log g(n))$.

If we add in the cost of scanning $\llcorner M \lrcorner$, we get a total cost of $O(kg(n)(\log n + |\llcorner M \lrcorner|))$.

We'll now adjust L to get rid of all the extra junk in this expression. Our new L' will be the set of all inputs $\langle \llcorner M \lrcorner, x \rangle$ where

1. M rejects $\langle \llcorner M \lrcorner, x \rangle$ in time at most $g(n)/k$, where k is the number of work tapes used by M;
2. M has $|\Gamma| = 2$; and
3. $|\llcorner M \lrcorner| < \log n$.

Since the extra conditions can be checked in $O(n) = O(g(n))$ time, and we can compute $1^{g(n)/k}$ in time $g(n)$ without being especially clever, we can easily

decide L' using the above construction in $O(k(g(n)/k)(\log n + \log n) + g(n)) = O(g(n)\log g(n))$ time. The only thing left is to show we didn't break the diagonalization argument.

Suppose R is a machine that decides L in $f(n) = o(g(n))$ time. Then there is also machine R' that decides L in $o(g(n))$ time with a restricted work-tape alphabet. For any fixed k (for example, the number of work tapes possessed by R'), there is some n_0 so that R' decides L in no more than $g(n)/k$ time for all $n \geq n_0$ (this just falls out of the definition of $o(g(n))$. So now we just make $|x| \geq \max(n_0, 2^{|\llcorner M \lrcorner|})$, and get R' to accept $\langle \llcorner R' \lrcorner, x\rangle$ if and only if $\langle \llcorner R' \lrcorner, x\rangle \notin L'$. So $L' \notin \mathbf{TIME}(f(n))$.

6.2 Hierarchy theorems for nondeterminism

So far, we have only considered deterministic computation. We can also ask if there are corresponding space and time hierarchy theorems for nondeterministic computation.

For space, a result very similar to Theorem 6.1.1 holds: $\mathbf{NSPACE}(f(n)) \subsetneq \mathbf{NSPACE}(g(n))$ whenever $f(n) = o(g(n))$, $g(n)$ is space-constructible, and $g(n) = \Omega(\log n)$. Pretty much the same proof works, with one complication: to get the upper bound, we need to build contradiction, we have to build a nondeterministic machine that *accepts* if and only if the nondeterministic machine it is simulating *rejects*, and doing this directly by just using the simulated machine's hint doesn't work. Fortunately, the Immerman-Szelepcsényi Theorem 9.4 says that $\mathbf{NSPACE}(g(n)) = \mathbf{coNSPACE}(g(n))$ when $g(n) = \Omega(\log n)$, so we can just build a simulator that accepts $\langle M, x\rangle$ if M accepts $\langle M, x\rangle$, and appeal to Immerman-Szelepcsényi to reverse the output.

For **NTIME** we can't do this, since we don't think **NTIME** is closed under complement. Instead, we need to use a different technique, called **lazy diagonalization** due to Cook [Coo73]. This gives the result $\mathbf{NTIME}(f(n)) \subsetneq \mathbf{NTIME}(g(n))$ whenever f and g are both time-constructible and $f(n+1) = o(g(n))$. We won't prove this here. See see [AB07, §3.3.3] if you want to know how it works.

6.3 Ladner's Theorem

Ladner's Theorem shows that if $\mathbf{P} \neq \mathbf{NP}$, there are sets in **NP** that are neither polynomial-time computable nor **NP**-complete. The proof is essentially a diagonalization argument, where we construct a single bad

set A by alternating between making it disagree with the output of the next polynomial-time machine M_i and making it fail to decide SAT after running SAT instances through the next polynomial-time function f_i. Some additional trickery puts the set in **NP**, giving the full result.

Theorem 6.3.1. *If* $\mathbf{P} \neq \mathbf{NP}$*, then there is a set* $A \in \mathbf{NP} \setminus \mathbf{P}$ *that is not* **NP***-complete.*

Proof. This particular proof is similar to Ladner's original proof [Lad75], but the presentation below follows a later paper by Downey and Fortnow [DF03], although we change the definition of A slightly to avoid a possible bug in that paper.[2]

Let $M'_1, M'_2, \ldots$ enumerate all Turing machines. Let M_i be the machine that first computes 1^{n^i} on an extra work tape, then runs M'_i for n^i steps, rejecting if it does not terminate in this time. Since n^i is time-constructible, M_i runs in $O(n^i)$ time, and since every Turing machine appears infinitely often in the M'_i list (because we can always pad with extra unused states), every polynomial-time Turing machine appears as some M_i.

Similarly let $f_1, f_2, \ldots$ enumerate all polynomial-time computable functions.

The language A is given by $\{x \mid x \in \text{SAT and } g(|x|) \text{ is even}\}$, where g (which tells us where to put the $g$aps) is a function to be determined. The idea is that the layers of A corresponding g enforce that some M_i can't compute A (because otherwise we could use a modified version of M_i to decide SAT in **P**), while the empty odd layers enforce that some f_i can't reduce SAT to A (because otherwise we could use a modified version of f_i to decide SAT in **NP**). We switch to the next layer once we detect that we have knocked out a particular M_i or f_i, which may require waiting until n is big enough that we can test all inputs up to the first bad one using brute

[2] The issue is that when testing f_i to see if $g(n) = 2i + 1$ should be increased, Downey and Fortnow check all inputs x with $|x| < \log n$ and wait until $\log^{g(n)} n < n$ to ensure that the computation of $f_i(x)$ on each of these inputs is $O(n)$. But it is still necessary to test if $f_i(x)$, which may have size as much as $\log^i |x|$, is in A, which may require testing if it is in SAT. Even taking into account the difference between i and $g(n) = 2i + 1$, we still get outputs from f_i that are of size polynomial in n, so we can't just test for membership in SAT by checking all the assignments. We also can't appeal to the fact that we are ultimately computing g using a nondeterministic machine, because we need to know both when g is positive and when it isn't. The proof given here avoids the issue entirely by putting a time bound on the testing procedure and arguing that n will eventually get big enough that M_i or f_i fails within the time bound, without attempting to compute the bound explicitly. This idea is pretty much the same as the approach used in Ladner's original paper.

force in polynomial time. But since n keeps growing forever, we can do this eventually for any fixed M_i or f_i if we wait long enough.

We'll define g recursively, and show that it is computable in deterministic polynomial time by induction on n. Because this construction depends on the running time of some of its own components, it may be easiest to think of it as defining a particular polynomial-time machine and defining g to be whatever that machine outputs.

Start with $g(0) = g(1) = 2$. We then compute $g(n+1)$ by first computing $g(n)$, and then branching based on whether it is odd or even:

1. If $g(n) = 2i$, lazily generate a list of all inputs x with $|x| < n$ in increasing order by x. For each x, simulate $M_i(x)$, and compare the output to whether or not x is in A, which we can test in time exponential in $|x|$ just by brute-forcing the solution to SAT if $g(|x|)$ is even.

   This procedure is not polynomial-time, but we can truncate it after n steps. If we find a bad x for which M_i gives the wrong answer during those n steps, then we set $g(n+1) = g(n) + 1$ and move on to the next machine. If not, we let $g(n+1) = g(n)$ and figure that we will catch M_i eventually when n is large enough. This works because no matter how long it takes to check each x, as long as that time is finite, eventually n exceeds whatever time is needed to check all the x up to the first bad one.

2. If $g(n) = 2i + 1$, do the same thing to f_i. Enumerate all x, run each through f_i, and test if $|f_i(x)| \leq n$ (so we don't create a cycle trying to compute $g(|f_i(x)|)$) and $x \in \text{SAT} \not\Leftrightarrow g(x_i) \in A$. As before we truncate after n steps.

   If we find such and x, f_i doesn't reduce SAT to A, so we can set $g(n+1) = g(n)$ and keep going. If we dont, we set $g(n+1) = g(n)$ and take solace in the notion that f_i's foot shall slide in due time [Edw41].

Since we bound the testing cost for each n by $O(n)$, the total cost of computing $g(n)$ is bounded by the recurrence the form $g(n+1) = g(n)+O(n)$, which gives $g(n) = O(n^2)$. So A is in **NP**, because a non-deterministic machine can, in polynomial time, compute $g(|x|)$ deterministically, and then either reject (if $g(|x|)$ is odd) or run SAT on x (if $g(|x|)$ is even).

We now argue that one of three things happens:

1. If $g(n) = 2i$ for all n greater than some n_0, then M_i correctly computes SAT in polynomial time for all but a finite number of x with $|x| \leq 0$. Add these cases to M_i using a lookup table to put SAT in **P**.

2. If $g(n) = 2i + 1$ for all n greater than some n_0, then $f_i(x) \in A \leftrightarrow x \in$ SAT for all x, and A is finite. So make a new machine that runs $f_i(x)$ and looks up the result in a table of all members of A to put SAT in **P**.

3. The remaining case is when g is unbounded. Then

   (a) No polynomial-time M_i decides A, so $A \notin \mathbf{P}$,

   (b) No polynomial-time f_i reduces SAT to A, so A is not **NP**-complete, and

   (c) Some polynomial-time nondeterministic Turing machine decides A, so $A \in \mathbf{NP}$.

Since the third case is the only one consistent with $\mathbf{P} \neq \mathbf{NP}$, the theorem holds. □

If $\mathbf{P} \neq \mathbf{NP}$, Ladner's Theorem demonstrates the existence of **NP-intermediate** sets, ones that lie strictly between **P** and the **NP**-complete sets. Indeed, by applying the construction in the proof iteratively, it is possible to show that there is an entire infinite chain of **NP**-intermediate sets, each irreducible to the next, yet all in $\mathbf{NP} \setminus \mathbf{P}$. But these sets are not especially natural, and it is reasonable to ask if we can point to any practically-motivated sets that might be **NP**-complete.

Ladner [Lad75], citing Karp [Kar72], mentions three sets that were candidates for being **NP**-intermediate at the time he proved his theorem: PRIME (is x prime?), LINEAR INEQUALITIES (is a given linear program feasible?), and GRAPH ISOMORPHISM (can we turn G into H just by relabeling the vertices?). All of these were known to be in $\mathbf{NP} \setminus \mathbf{P}$ as of 1975, and none of them were known to be **NP**-complete, but there was no argument that any of them in particular were **NP**-intermediate if $\mathbf{P} \neq \mathbf{NP}$. As of 2017, two of them (PRIME [AKS04] and LINEAR INEQUALITIES [Kha80]) are known to be in **P**. The status of GRAPH ISOMORPHISM is still open, with the best known algorithm running in quasi-polynomial time ($O(n^{\log^c n})$) [Bab15].

Whether there are other potentially **NP**-intermediate problems that are natural depends somewhat on one's definition of "natural." See http://cstheory.stackexchange.com/questions/20930/why-are-so-few-natural-candidates-for-np- for an example of how different people can have very different perspectives on this question.

Chapter 7

Oracles and relativization

Last updated 2017. Some material may be out of date.

Complexity theory is unusual in having a rich collection of barrier results that show that certain proof techniques cannot be use to solve core problems like $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$. One of the most important is **relativization**, where we consider extending our machine model by adding extra information on the side. In some cases, this extra information can make the **relativized** version of **P** provably equal to, or provably not equal to, the similarly relativized verson of **NP**. If this is the case, then we are in trouble if we try to resolve $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ using any technique that doesn't manage to exclude the extra stuff.

We can formalize this idea in terms of **oracle machines**, which we define in the next section.

7.1 Oracle machines

Given a machine M in some model, the machine M^A is obtained by taking M and adding a write-only **oracle tape** that gives M the ability to determine in a single tape if the string x written to the oracle tape is in A. When X and Y are complexity classes, we also write X^Y for the class of languages computable by a machine in class X using an oracle from class Y.

This definition makes X^Y at least as powerful as the stronger of X and Y, and may give it even more power: for example, if $\mathbf{NP} \neq \mathbf{coNP}$, then $\mathbf{NP} \subsetneq \mathbf{P}^{\mathbf{NP}}$ since in $\mathbf{P}^{\mathbf{NP}}$ we can solve any problem in **NP** just by asking the oracle for its opinion, but we can also solve any problem in **coNP** by asking the oracle for its opinion and giving the opposite answer. And this is

all without even taking advantage of the ability to ask multiple questions and have the later questions depend on the outome of earlier ones.

7.2 Relativization

We say that a proof relativizes if it is not affected by adding an oracle, and that a hypothesis relativizes if there is an oracle that makes it true and another oracle that makes it false. If a hypothesis relativizes, it can only be proved or disproved using a technique that doesn't relativize.

All of the techniques we have seen so far relativize, because if we only use the fact that we can simulate a machine M using some machine U, then it also holds that we can simulate M^A using U^A. This is bad news for resolving $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$, because that hypothesis also relativizes.

7.2.1 The Baker-Gill-Solovay Theorem

The **Baker-Gill-Solovay Theorem** [BGS75] says that there exist oracles A and B such that $\mathbf{P}^A = \mathbf{NP}^A$ but $\mathbf{P}^B \neq \mathbf{NP}^B$. This means that $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ cannot be resolved by any technique that relativizes.

Let EXPCOM $= \{\langle M, x, 1^n\rangle \mid M \text{ outputs 1 on } x \text{ within } 2^n \text{ steps}\}$. This gives an oracle $A =$ EXPCOM for which $\mathbf{P}^A = \mathbf{NP}^A$. To prove this, observe that a nondeterministic machine that runs in $O(n^k)$ steps with an EXPCOM oracle can be simulated deterministically by iterating through all possible witness strings (there are at most $2^{O(n^k)}$ of them and simulating each call to the oracle by performing the requested computation directly (there are $O(n^k)$ such calls and each takes $O(2^{O(n^k)})$ time, since we can't write a string of ones longer than $O(n^k)$ on the oracle tape. The total time for this simulation is $O(2^{O(n^k)} \cdot O(n^k) \cdot O(2^{O(n^k)})) = O(2^{O^{n^k}})$. So we can do a single call to EXPCOM on our $\mathbf{P}^{\text{EXPCOM}}$ machine to determine the output of this simulation, making $\mathbf{NP}^{\text{EXPCOM}} \subseteq \mathbf{P}^{\text{EXPCOM}}$.

For the other direction, we construct an oracle for which the language $L_B = \{1^n \mid \exists x \in B, |x| = n\}$ is in $\mathbf{NP}^B \setminus \mathbf{P}^B$. It is easy to show that this language is in $\mathbf{NP}^B$ is in $\mathbf{NP}^B$ for any B, since we can just have our nondeterministic machine guess x and check it using an oracle call. To find a B for which the language is not in $\mathbf{P}^B$, we use diagonalization.

Call the set of all x with $|x| = n$ level n of the oracle. We will set up a sequence of increasing levels $n_1, n_2, \ldots$ such that each polynomial-time machine M_i, where $i = 1, 2, \ldots$, gives the wrong answer on 1^{n_i}. Formally, this construction will involve building an increasing sequence of oracles

$B_1 \subseteq B_2 \subseteq \ldots$ with $B = \bigcup_{i=1}^{\infty} B_i$, where each B_i causes $M_i(1^{n_i})$ to return the wrong value, but at the same time does not change the result of running $M_j(1^{n_j})$ against oracle B_j for $j < i$. We will maintain the invariant that B_i does not include any x with $|x| > i$, and that $x \in B_i$ if and only if $x \in B_{i-1}$ when $|x| < i$. We start with $B_0 = \emptyset$ and $n_0 = 0$.

Let n_i be the smallest value of n such that (a) for every $j < i$, $M_j(1^{n_j})$ running with B_j does not query the oracle on any string x with $|x| \geq n$; and (b) $M_i(1^{n_i})$ running with B_{i-1} does fewer than 2^n queries x with $|x| = n$. If $M_i(1^{n_i})$ accepts when running with B_{i-1}, then let $B_i = B_{i-1}$; from the invariant, there is not $x \in B_{i-1} = B_i$ with $|x| = n_i$, so M_i gives the wrong answer. If instead $M_i(1^{n_i})$ rejects when running with B_{i-1}, pick some y with $cardy = n_i$ such that $M_i(1^{n_i})$ doesn't query y, and let $B_i = B_{i-1} \cup \{y\}$. Now M_i again gives the wrong answer. Since we don't change any values observed by machine M_j for $j < i$, changing from B_{i-1} to B_i doesn't make any of them start working, and when we take the limit B we will have successfully caused every polynomial-time M_i^B to fail on at least one input. It follows that for this particular B, we have $L_B \notin \mathbf{P}^B$, which gives $L_B \in \mathbf{NP}^B \setminus \mathbf{P}^B$.

7.3 The oracle polynomial-time hierarchy

Oracles give us one of several definitions of the polynomial-time hierarchy. Let $\mathbf{\Delta}_0^p = \mathbf{\Sigma}_0^p = \mathbf{\Pi}_0^p = \mathbf{P}$. Then define

$$\mathbf{\Delta}_{k+1}^p = \mathbf{P}^{\mathbf{\Sigma}_k^p}$$
$$\mathbf{\Sigma}_{k+1}^p = \mathbf{NP}^{\mathbf{\Sigma}_k^p}$$
$$\mathbf{\Pi}_{k+1}^p = \mathbf{coNP}^{\mathbf{\Sigma}_k^p}$$

This gives what we believe to be an infinite tower of complexity classes, with each $\mathbf{\Delta}_k^p$ contained in $\mathbf{\Sigma}_k^p$ and $\mathbf{\Pi}_k^p$, and $\mathbf{\Sigma}_k^p$ and $\mathbf{\Pi}_k^p$ contained in $\mathbf{\Delta}_{k+1}^p$. The union of all of these classes is **PH**, the **polynomial-time hierarchy**. For the moment, we will refer to this version specifically as the **oracle polynomial-time hierarchy** , to distinguish from some other definitions, but these definitions will turn out to be equivalent in the end.

Chapter 8

Alternation

Last updated 2017. Some material may be out of date.

In a **NP** computation, the computation tree contains only OR branches. In a **coNP** computation, the computation tree contains only AND branches. We can define still more powerful (we think) classes by allowing both OR and AND branches. We specify how tough a class is by how many times we switch back and forth between OR and AND, that is, by the number of **alternations**alternation between OR and AND.

There are two ways to define the resulting hierarchy: one in terms of alternating quantifiers (corresponding to allowing both OR and AND branches) and one in terms of oracles. These turn out to be equivalent, but the equivalence is not completely trivial. We'll start with the alternating version, which itself splits into two versions, one involving quantifies, and one involving alternating Turing machines.

8.1 The alternating polynomial-time hierarchy

The **alternating polynomial-time hierarchy** is defined by extending the certificate version of the definition of **NP**. In this definition (see [AB07, Definition 5.4], a language L is in $\mathbf{\Sigma}_k^p$ if there is a polynomial p and a polynomial-time machine M such that $x \in L$ if and only if $\exists w_1 \forall w_2 \exists w_3 \ldots Q w_k M((x, w_1, w_2, \ldots, w_k$ accepts, where Q represents either a $\exists$ or a $\forall$ quantifier as appropriate and each w_i has $|w_i| \leq p(|x|)$.

A language is in $\mathbf{\Pi}_k^p$ if its complement is in $\mathbf{\Sigma}_k^p$, or equivalently if there is an alternating formula for membership in L using k quantifiers starting with $\forall$. Unlike the oracle definition, there does not seem to be a natural way to define $\mathbf{\Delta}_k^p$ in terms of alternating formulas.

8.2 Equivalence to alternating Turing machines

So far we have defined the alternating hierarchy in terms of logical formulas, but our original claim was that this had something to do with alternating Turing machines.

An **alternating Turing machine** generalizes a nondeterministic Turing machine by allowing both OR and AND nondeterministic transitions. This produces a branching tree of computations, just as in a nondeterministic machine, but instead of applying the rule that the machine accepts if any branch accepts, we apply a more complicated rule that essentially corresponds to evaluating a game tree.

Each leaf node of the tree, corresponding to a halting configuration of the machine, is assigned a value true or false depending on whether that configuration accepts or rejects. A deterministic node (with exactly one successor) gets the same value as its successor. Nondeterministic OR nodes get the OR if their successors' values, while nondeterministic AND nodes get the AND of their successors' values. The machine as a whole accepts if and only if the root node gets the value true.

An alternating Turing machine can easily implement a sequence of quantifiers, by representing each $\exists$ quantifier as a sequence of OR branches and each $\forall$ quantifier as a sequence of AND branches, where following each branch the machine writes the next bit of the appropriate variable to a work tape. Given a $\mathbf{\Sigma}_k^P$ language, this requires k layers of alternating OR and AND branches, followed by a deterministic polynomial-time computation. So we can represent $\mathbf{\Sigma}_k^p$ languages (and similarly $\mathbf{\Pi}_k^p$ languages) using alternating Turing machines with this restriction on the branching.

In the other direction, suppose that we have an alternating Turing machine, where on each branch of the computation we have a sequence of OR branches, followed by AND branches, followed by OR branches, with at most k such subsequences of branches of the same type and at most polynomially-many steps altogether on each branch. We would like to argue that the language decided by this machine is in $\mathbf{\Sigma}_k^p$. This is not completely trivial because there might be computation interspersed with the branching. However, we can reorganize the machine so that it starts by generating the choices for all the branches ahead of time, writing them to a work tape, and then simulates the original machine by reading the next choice off the work tape as needed. So in fact we get the same class of languages out of alternating formulas as alternating TMs.

8.3 Complete problems

Define $\mathbf{\Sigma}_k$-SAT to be the language consisting of all true $\mathbf{\Sigma}_k$ Boolean formulas, and similarly define $\mathbf{\Pi}_k$-SAT to be the language of all true $\mathbf{\Pi}_k$ Boolean formulas. Then $\mathbf{\Sigma}_k$-SAT is complete for $\mathbf{\Sigma}_k^p$, and $\mathbf{\Pi}_k$-SAT is complete for $\mathbf{\Pi}_k^p$.

The proof of this is essentially the same as for the Cook-Levin theorem, which we can think of as showing that $\mathbf{\Sigma}_1$-SAT (otherwise known as SAT) is complete for $\mathbf{\Sigma}_1^p$ (otherwise known as **NP**). We need to be a little bit careful because the Cook-Levin proof uses the $\exists$ quantifier for two purposes: when converting the $\mathbf{\Sigma}_1^p$ formula $\exists w M(x, w)$ accepts to a $\mathbf{\Sigma}_1$-SAT instance, we introduce extra variables representing the state of M during its computation, and we effectively construct a $\mathbf{\Sigma}_1$ Boolean formula $\exists w \exists c \phi(x, w, c)$. Exactly the same approach works for $\mathbf{\Sigma}_k^p$ formulas when k is odd, because we can combine the new existential quantifier introducing the tableau variables with the last existential quantifier in the $\mathbf{\Sigma}_k^p$ formula; but we can't do this if k is even. Fortunately, for even k we can apply the construction to $\mathbf{\Pi}_k^p$ formulas, and in general we can use the fact that the complement of a $\mathbf{\Pi}_k^p$-complete language is $\mathbf{\Sigma}_k^p$-complete and vice versa to cover all the cases we missed.

8.4 Equivalence to oracle definition

We generally don't distinguish between the alternating and oracle versions of the polynomial-time hierarchy, because they give the same classes.

Theorem 8.4.1. *For all k, $\mathbf{\Sigma}_k^{p,oracle} = \mathbf{\Sigma}_k^{p,alternating}$.*

Proof. By induction on k. When $k = 0$, both are **P**.

For larger k, suppose that the theorem holds for $\mathbf{\Sigma}_{k-1}^p$.

The easy direction is $\mathbf{\Sigma}_k^{p,\text{oracle}} \supseteq \mathbf{\Sigma}_k^{p,\text{alternating}}$. Recall that L is in $\mathbf{\Sigma}_k^{p,\text{alternating}}$ if there is a formula $\exists w P(x, w)$ that is true when $x \in L$, where w has size polynomial in x and P is computable in $\mathbf{\Pi}_{k-1}^{p,\text{alternating}}$. By the induction hypothesis, P is also computable by some machine in $\mathbf{\Pi}_{k-1}^{p,\text{oracle}}$. But then a $\mathbf{NP}^M$ machine can decide L, by guessing w and verifying it using M. This puts L in $\mathbf{NP}^{\mathbf{\Pi}_{k-1}^p} = \mathbf{NP}^{\mathbf{\Sigma}_{k-1}^p} = \mathbf{\Sigma}_k^{p,\text{oracle}}$.

The other direction requires some trickery, because a $\mathbf{NP}^{\mathbf{\Sigma}_{k-1}^p}$ machine can base later oracle calls on the outcome of earlier ones, and can use the result of the oracle calls however it likes. To turn a computation by such a machine into a $\mathbf{\Sigma}_k^p$ formula, we start by guessing, using an existential

quantifier, the entire tableau describing the machine's execution, including guesses for the results of the oracle calls. Verifying that this tableau is consistent requires only a $\mathbf{\Sigma}_0^p$ formula, but we need a $\mathbf{\Sigma}_{k-1}^p$ formula to check each oracle call that returns 1 and a $\mathbf{\Pi}_{k-1}^p$ formula to check each oracle call that returns 0. Fortunately both of these formulas fit in $\mathbf{\Sigma}_k^p$, and taking the AND of all of these formulas is still in $\mathbf{\Sigma}_k^p$. Tacking the guess for the tableau on the front still leaves the formula in $\mathbf{\Sigma}_k^p$. So we have $\mathbf{\Sigma}_k^{p,\text{oracle}} = \mathbf{NP}^{\mathbf{\Sigma}_{k-1}^p} \in \mathbf{\Sigma}_k^{p,\text{alternating}}$. □

8.5 PH ⊆ PSPACE

Problems in the polynomial-time hierarchy can be powerful, but any of them can be solved in polynomial space.

In fact, we can encode any problem in **PH** by reduction to a problem called **TRUE QUANTIFIED BOOLEAN FORMULA** or **TQBF**. TQBF consists of all true formulas of the form $Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \Phi(x_1, \ldots, x_n)$ where each Q_i is either $\forall$ or $\exists$ and each x_i represents a Boolean value.

For example, any problem in $\mathbf{\Sigma}_1^p = \mathbf{NP}$ can be encoded as a formula of the form $\exists x_1 \ldots \exists x_n \Phi(x_1, \ldots, x_n)$, where Φ is the SAT formula produced by the Cook-Levin Theorem. More generally, a $\mathbf{\Sigma}_{k+1}^p$ problem $\exists y M(x, y)$, where M is in $\mathbf{\Pi}_k^p$, can be encoded as a formula of the form $\exists y_1 \ldots \exists y_n \Phi(y_1, \ldots, y_n) \Phi(x, y)$, where Φ encodes M; the same thing works for $\mathbf{\Pi}_{k+1}^p$ formulas except we get universal quantifiers instead of existential quantifiers. In either case we can recurse within Φ until we get down to no quantifiers at all. This makes TQBF hard for any $\mathbf{\Sigma}_k^p$, and thus for all of **PH**.

It can't be **PH**-complete unless the hierarchy collapses. The reason it doesn't immediately collapse is that TQBF allows unbounded layers of alternation.

TQBF is computable in **AP**, **alternating polynomial time**, which is what we get when we have no restriction on an alternating Turing machine except that each branch must run in polynomial time. It is also computable in **PSPACE**, since we can build a recursive procedure that given a formula $Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \Phi(x_1, \ldots, x_n)$ checks if both assignments to x_1 (if $Q_1 = \forall$) or at least one assignment to x_2 (if $Q_1 = \exists$) makes the rest of the formula true. So we can decided any language in **PH** by reducing to TQBF and then solving TQBF in **PSPACE**, which shows **PH** ⊆ **PSPACE**.

In fact, TQBF is **PSPACE**-complete, but we will defer the proof of this to §9.2. A consequence of this is that **AP** = **PSPACE**, which will save us

from having to think about **AP** as a separate class.

Chapter 9

Space complexity

Last updated 2017. Some material may be out of date.

Understanding space complexity requires giving up some preconceptions about time efficiency, since space complexity classes don't care about time efficiency, and indeed many of the known results on space complexity involve spending time like water. On the other hand, if the space complexity is low enough, this puts a bound on time complexity even if our use of time is stupidly inefficient.

Important space complexity classes:

- **PSPACE** $= \bigcup_{c=1}^{\infty}$ **SPACE**(n^c). Contains the entire polynomial-time hierarchy and then some.
- **L** $=$ **SPACE**$(\log n)$. The class of problems solvable in **log-space**. Largest practically-implementable space complexity class (it's contained in **P**). Popular with theoreticians who think about databases. Might be equal to:
- **NL** $=$ **NSPACE**$(\log n)$. **. . .** Like **P** $\stackrel{?}{=}$ **NP**, **L** $\stackrel{?}{=}$ **NL** is open. log-space hierarchy collapse to this class, which is equal to **coNL** (§9.4).

Other noteworthy space complexity classes:

- **SC** $= \bigcup_{c=1}^{\infty}$ **SPACE**$(\log^c n)$. "**Steve's class**", named for Steve Cook. The class of problems solvable in **polylog space**.
- **RL** (**randomized log-space**). log-space version of **RP**, which we'll see more of in Chapter 12. A language is in **RL** if a randomized log-space machine accepts positive instances with constant probability and rejects all negative instances. May be equal to **L**.

- **SL** (**symmetric log-space**). Languages accepted by a reversible log-space machine, which has the property that for any transition $C \to C'$ there is also a reverse transition $C' \to C$. Proposed by Lewis and Papadimitriou [LP82] as an intermediate class between **L** and **NL**. Now known to be equal to **L**. [Rei08]

9.1 Space and time

Claim: $\mathbf{TIME}(f(n)) \subseteq \mathbf{SPACE}(f(n))$. Proof: Can't use more than $O(1)$ new tape cells per step.

Claim: $\mathbf{SPACE}(f(n)) \subseteq \mathbf{TIME}(2^{O(f(n))})$. Proof: A $\mathbf{SPACE}(f(n))$ machine has only $2^{O(f(n))}$ distinct configurations, so if it runs for longer, it's looping and will never halt.

Application: $\mathbf{L} \subseteq \mathbf{P}$, $\mathbf{PSPACE} \subseteq \mathbf{EXP}$. For all we know, equality might hold in both cases. (Current consensus is that it doesn't.)

9.2 PSPACE and TQBF

Main result: TQBF is **PSPACE**-complete. This implies $\mathbf{PH} \subseteq \mathbf{SPACE}$. If they are equal, existence of a **PSPACE**-complete language means that **PH** collapses to a finite level.

Claim: TQBF $\in$ **PSPACE**. Proof: game-tree search.

Claim: TQBF is **PSPACE**-hard. Proof: given a language in **PSPACE**, reduce to TQBF.

Idea: Space complexity is all about reachability in the graph of configurations. If $L = L(M)$ for some **PSPACE** machine M, we can test if $x \in L$ by testing if there is a path from the initial configuration of $M(x)$ to some accepting configuration. To make our life easier later, we will assume that M cleans up on the way out (erasing all the work tapes, moving the heads back to their starting positions) so that the accepting configuration is unique.

The formula, naive version: $C \to^* C'$ expands to $\exists C'' : C \to^* C'' \wedge C'' \to^* C'$. This is too big, so instead we use $\forall$ to test both branches with one formula:

$$C \to^* C' \equiv \exists C'' \forall C_1, C_2 : ((C_1 = C \wedge C_2 = C'') \vee (C_1 = C'' \wedge C_2 = C')) \wedge C_1 \to^* C_2.$$

At the bottom we use Cook-Levin to test if $C_1 \to C_2$ in a single step. This requires depth logarithmic in the length of the execution, and the length is at most exponential, giving depth at most polynomial. Each stage adds a

polynomial size to the formula (those configurations are big, but they are only polynomial), so total size of the formula is polynomial.

9.3 Savitch's Theorem

Savitch's Theorem: If $f(n) = \log n$), then $\mathbf{SPACE}(f(n)) \subseteq \mathbf{NSPACE}((f(n))^2)$.

The proof is similar to the proof of **SPACE**-hardness of TQBF. If $C_0 \rightarrow^{\leq T(n)} C_{\text{accept}}$, then there is some intermediate configuration C' such that $C_0 \rightarrow^{\leq T(n)/2} C' \rightarrow^{\leq T(n)/2} C_{\text{accept}}$. For each possible C', test the left side recursively first, then the right side if the left side worked.

We have to store C' in between these tests, but we can re-use the space we used for the left-side test for the right-side test. So this gives space complexity $S(T(n)) = O(f(n)) + S(T(n/2)) = O(f(n) \log T(n)) = O(f(n) \log 2^{O(f(n))} = O((f(n)^2)$.

Consequences: **NPSPACE** = **PSPACE**, and in fact the entire polyspace hierarchy collapses to **PSPACE**. (Unbounded alternation, which boosts **APSPACE** up to **EXP** (see §9.6.2) may give more power.)

9.4 The Immerman-Szelepcsényi Theorem

Says **NL** = **coNL**. Proof by "inductive counting": Let A_i be set of all configurations reachable in i steps. Given $|A_i|$, we'll compute $|A_{i+1}|$ by enumerating all configurations C that might be in A_i, and for each C try all possible configurations C' that might be predecessors. For each C', we guess and check a path. If we come up with fewer than $|A_i|$ predecessors with paths, we screwed up: reject this computation path and hope one of the other lemmings does better.

The last step is the same but we only look if C_{accept} has no path.

Easy consequence: the alternating log-space hierarchy collapses. Use **NL** = **coNL** to replace last $\forall$ or $\exists$ quantifier with its predecessor and combine them, same as in the proof that $\mathbf{\Sigma}_k^p = \mathbf{\Pi}_k^p$ implies **PH** collapses.

Harder consequence: the oracle log-space hierarchy collapses. This requires some careful looking at how to define oracle access in **NL**, which we will do in the next section.

9.5 Oracles and space complexity

Just as we have to be careful to restrict the input and output tapes of a space-bounded computation to prevent them from doing double duty as work tapes,

we also have to be careful how we define an oracle computation involving a space-bounded machine, particularly if that machine uses nondeterminism. The now-standard definition first appeared in a paper by Ruzzo, Simon, and Tompa [RST84].

The idea is that we have an oracle tape as usual, which we make write-only like the output tape. But we put some restrictions on how a nondeterministic machine can use this tape. These are:

1. Once the controller writes to the oracle tape, it cannot execute any nondeterministic transitions until after it completes and oracle call.

2. An oracle call erases the oracle tape.

Without these restrictions, we would get bizarre results like $\mathbf{NL}^{\mathbf{L}}$ being able to solve SAT even if **NL** couldn't on its own: an unrestricted $\mathbf{NL}^{\mathbf{L}}$ oracle machine could write the problem and a guess at a satisfying assignment on the oracle tape and ask the **L** oracle to verify it [RST84, Example 1]. The Ruzzo-Simon-Tompa definition prevents this, and has the additional advantage of making the contents of the oracle tape, for a $\mathbf{SPACE}(f(n))$ or $\mathbf{NSPACE}(f(n))$ machine, encodable in $O(f(n))$ space provided the decoder has access to the original input.

We illustrate this by showing that $\mathbf{NL}^{\mathbf{NL}} = \mathbf{NL}$ when the oracle calls are subject to the Ruzzo-Simon-Tompa restrictions. Let M be the nondeterministic log-space oracle machine (which is to say the machine that calls the oracle), and let M' be a nondeterministic log-space machine implementing the oracle. We want to build a combined machine N that simulates $M^{M'}$.

The idea is that when N simulates an oracle call, instead of writing to an oracle tape that it doesn't have, it saves a copy of the state of M at the start of the oracle-tape write. This copy now gets called as a kind of co-routine whenever M' looks at a cell on the simulated oracle tape: if M' looks at position i in the oracle tape, N will simulate a fresh copy M starting from its saved state, recording whatever symbols it writes to position i, until simulated-M makes an oracle call. Then N can throw away this copy and continuing simulating M' until the next time it looks at the oracle tape. This is, as usual, tremendously wasteful of time, but it still fits in $\mathbf{NSPACE}(f(n))$.

For the specific case of $\mathbf{NL}^{\mathbf{NL}} = \mathbf{NL}$, we need the additional trick of guessing in advance which way the oracle will answer. If the oracle is expected to answer yes, we just run the straight **NL** computation, and give up if the answer is no (as usual). If the oracle is expected to answer no, we instead run a **coNL** computation converted into **NL** using the Immerman-Szelepcsényi

Theorem. Again we give up if the answer is wrong. Since N can do this for each of M's oracle calls and stay in **NL**, we get $\mathbf{NL}^{\mathbf{NL}} = \mathbf{NL}$ and the log-space oracle hierarchy collapses to **NL**.

Some similar issues come up with **PSPACE**. Here to prevent cheating we restrict the oracle tape to have polynomial size. With this restriction in place, $\mathbf{NPSPACE}^{\mathbf{NPSPACE}} = \mathbf{NPSPACE} = \mathbf{PSPACE}$ by applying the preceding construction and Savitch's Theorem.

9.6 $\mathbf{L} \stackrel{?}{=} \mathbf{NL} \stackrel{?}{=} \mathbf{AL} = \mathbf{P}$

So far we have seen that $\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}$. We know from the Space Hierarchy and Time Hierarchy theorems that some of these inclusions must be strict, since $\mathbf{L} \neq \mathbf{PSPACE}$ and $\mathbf{P} \neq \mathbf{EXP}$. But we don't know which, and given that **L**, **NL**, and **P** are all in some sense "easy" classes, it is not completely ridiculous to ask if $\mathbf{L} \stackrel{?}{=} \mathbf{NL}$ or $\mathbf{L} \stackrel{?}{=} \mathbf{P}$.

9.6.1 Complete problems with respect to log-space reductions

As with $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$, we can make these problems more concrete by concentrating on specific problems that are **complete** for **NL** or **P**. This notion of completeness will be slightly different from the usual notion of completeness used for **NP**, since poly-time reductions from problems in **P** work for just about any target problem: have the reduction do the computation. So instead we will consider log-space reductions.[1]

We write $L' \leq_{\mathbf{L}} L$ if there is a log-space reduction from L' to L, meaning that there is a log-space Turing machine M that transforms instances x of L' to instances $M(x)$ of L, such that $x \in L'$ if and only if $M(x) \in L$.

A language L is **NL-complete** if L is in **NL** and for every L' in **NL**, $L' \leq_{\mathbf{L}} L$. Similarly, a language L is **P-complete** if L is in **P** and for every L' in **P**, $L' \leq_{\mathbf{L}} L$.

In some contexts, it makes sense to consider different restrictions on the reductions, and we will say things like "L is **P**-complete with respect to log-space reductions" or "L is **P**-complete with respect to $\mathbf{NC}^1$ reductions." But for now we will stick with log-space reductions when working with **NL**- or **P**-completeness.

[1] These also work for most **NP**-completeness reductions. For example, the reduction in the Cook-Levin Theorem can easily be made to run in log space, making every language $L \in \mathbf{NP}$ log-space reducible to 3SAT.

9.6.1.1 Complete problems for NL

The classic **NL**-complete problem is s–t **connectivity**, abbreviated as **STCON**. The input to STCON is a triple $\langle G, s, t\rangle$ where G is a directed graph and s and t are vertices, and $\langle G, s, t\rangle$ is in STCON if there is a directed path from s to t in G.

STCON is in **NL** because we can just guess each step of the path. To show that any language $L \in \mathbf{NL}$ reduces to STCON, start with a non-deterministic log-space machine M_L that decides L with a unique accepting state, and build a log-space machine that enumerates all possible pairs of states of M_L (there are only polynomially many, so we can do this in log space), and writes out an edge uv for each pair of states u and v such that v can follow from u given the contents of the input tape (also pretty straightforward to test in log space). Then ask if there is a path from the initial state to the accepting state.[2] This gives $L \leq_{\mathbf{L}}$ STCON, and STCON is **NL**-complete.

We can further reduce STCON to 2SAT to show that 2SAT is **NL**-complete, which gives a nice symmetry with 3SAT being **NP**-complete. An instance of 2SAT is a collection of two-literal OR clauses $x \vee y$ and we want to know if there is a truth assignment that makes all the clauses true.

The idea here is that we can encode an edge uv in our input graph G as an implication $u \Rightarrow v$, which is logically equivalent to the OR clause $\neg u \vee v$. We do this for every edge in G, and add clauses asserting s ($s \vee s$ if we are being picky about exactly two literals per clause) and $\neg t$. It's not too hard to see that we can do this in log space.

If there is no path from s to t in G, then there is a satisfying assignment that sets s true, that sets every vertex reachable from s also true, and sets every vertex not reachable from s (including t false. But if there **is** a path from s to t, we are in trouble. We have to set s true, but then the implications for the edges mean that we also have to set every literal reachable from s, including t true. But then we can't satisfy the $\neg t$ clause.

If you look closely at this argument, you may have noticed that we didn't reduce STCON to 2SAT. Instead we reduced s–t *non*-connectivity to 2SAT, or equivalently we reduced STCON to 2CNF *non-satisfiability*. This means that we showed 2SAT is **coNL**-hard. But **coNL** = **NL**, so it's OK.

To put 2SAT in **NL**, we reverse this construction and translate a 2SAT problem into its **implication graph**, where the implication graph contains nodes for every literal (so both x and $\neg x$ for each variable x), and there is

[2] If we don't want to insist on a unique accepting state, which doesn't really constraint M_L because it can do the usual trick of erasing its work tapes and parking its heads, we can add an extra state t and run an edge from every accepting state to t.

an edge uv for every pair of literals such that $(\neg u \vee v) \equiv (u \Rightarrow v)$ appears in the 2CNF formula. Now there is a satisfying assignment if and only if there is no path from any x to $\neg x$ or vice versa. This is easily testable in $\mathbf{NL}^{\mathbf{NL}} = \mathbf{NL}$, since we can just query an STCON oracle for each pair.

9.6.1.2 Complete problems for P

P-complete problems are similar to **NP**-complete problems in that they tend to involve encoding a poly-time computation in some other process. The difference is that with **P**, there is no nondeterminism, which means that the problem we are reducing to generally won't have many choices left open.

A trivially **P**-complete language is PCOM $= \{\langle M, x, 1^n\rangle \mid M \text{ accepts } x \text{ in } n \text{ steps}\}$. This is in **P** because a universal Turing machine can check if M does in fact accept x in n steps, using time poly in n (and thus the size of the input, since we are expressing n in unary). m It's **P**-complete, because given any particular language L in **P** recognized by a machine M_L that runs in n^c steps, we can reduce L to PCOM using a log space machine that writes out M, copies x, and then computes $1^{|x|^c}$.

Somewhat more interesting is CVP, the **circuit value problem**. This is defined by given a Boolean circuit C and its input x, and asking whether the circuit outputs 1. (Formally, a Boolean circuit is a directed acyclic graph where nodes are labeled with either an input bit or with a function, the value of each node is the value of the function applied to the values of its predecessors, and the value of the circuit is the value of a designated output node.) This is in **P** because we can easily write a poly-time program to solve it. It's **P**-complete because we can take any fixed poly-time M and translate its execution into a circuit that calculates the contents of each tape cell, etc., at each time, and set the output to whether M accepts.

For more elaborate examples, we need to find some clever way to encode circuit gates in whatever problem we are looking at. One cute example is LINEAR INEQUALITIES, where we are given a matrix A and vector b, and want to know if there is a vector x such that $Ax \leq b$. This corresponds to feasibility of a linear program, which is known to be solvable in polynomial time. To show that it is **P**-complete, take CVP for a circuit built from AND and NOT gates, and translate $y = \neg x$ as $y = 1 - x$ and $z = x \wedge y$ as $z \leq x, z \leq y, z \geq x + y$. Then if we peg all inputs at 0 or 1, and put a $0 \leq x \leq 1$ constraint on any variable representing a gate, the unique satisfying variable assignment will set the output variable C equal to the output of the circuit. One more constraint $C \geq 1$ makes the system of linear inequalities feasible if and only if the output is 1.

Many more examples of **P**-complete problems can be found in a classic survey of Greenlaw, Hoover, and Ruzzo [GHR91].

9.6.2 AL = P

With bounded alternation, the log-space hierarchy collapses to **NL**, which we suspect is properly contained in **P**. But unbounded alternation gets us to **P**.

Proof: Let M be a single-tape Turing machine that runs in time n^c for some fixed c. We will show that $L(M) \in \mathbf{AL}$. Define a **tableau** consisting of cells C_{ij} where each C_{ij} describes the state at time i of position j of the tape, including whether the head is at position j and its state if it is there.

Observe that C_{ij} is completely determined by $C_{i-1,j-1}$, $C_{i-1,j}$, and $C_{i-1,j+1}$. We'll assume that M is polite and parks the head on the starting tape position 0 at time n^c, so we can check if M accepts by checking if $C_{n^c,0}$ is in an accepting state.

We do this by building a recursive procedure that checks if a given C_{ij} is in a particular state x_{ij}. For $i = 0$ we can just check x_{ij} against the input tape. To test if C_{ij} is in state x_{ij} when $i > 0$, we first guess (using $\exists$) states $x_{i-1,j-1}$, $x_{i-1,j}$, and $x_{i-1,j+1}$ for $C_{i-1,j-1}$, $C_{i-1,j}$, and $C_{i-1,j+1}$. If these states are not consistent with x_{ij}, we halt and reject. This leaves only branches with consistent predecessor states. We want to verify that all three of these states are correct. We do this recursively, inside a $\forall$ quantifier that picks one of the three states to check. Only if all three checks succeed do we accept.

This fits in log space because when we do the recursive check, we only need to store $i - 1$, the appropriate tape position, and the value x_{ij}. Since $i - 1$ and the tape position are bounded (in absolute value) by n^c, we can do this with $O(c \log n) = O(\log n)$ bits. Checking the predecessor states for consistency doesn't require much additional storage, since we can bake M's transition table into the **AL** machine's finite-state controller and each x_{ij} has constant size. The result is an alternating log-space machine that accepts an input x if and only if M does.

Since we can do this for any language in **P**, we get $\mathbf{AL} \subseteq \mathbf{P}$. In the other direction, a **P** machine can compute the entire state graph for a **AL** machine (it has only polynomially many states), and evaluate whether to accept or reject at each branching node based on whether the successor nodes accept or reject.

This result generalizes. What we've really shown is that $\mathbf{ASPACE}(f(n)) = \mathbf{TIME}(2^{O(f(n))})$, at least for $f(n) = \Omega(\log n)$. (For smaller f we run into

issues keeping track of the position of the input head.) So in addition to **AL** = **P**, we also get **APSPACE** = **EXP**. In both cases, unbounded alternation is necessary unless something surprising happens, since bounded alternation gives us either **NL** or **PSPACE**, which we suspect are different from **P** and **EXP**.

Chapter 10

Circuit complexity

Last updated 2017. Some material may be out of date.

Circuit complexity models computation in terms of **Boolean circuits**, which are formally represented as directed acyclic graphs where each node is either an **input** (with no incoming edges) **gate** (with at least one incoming edge). We think of the edges as wires carrying bits, and typically assume that each input is a bit and each gate computes some **Boolean function** $f : \{0,1\}^k \to \{0,1\}$. If a Boolean circuit has maximum out-degree 1, it is a **Boolean formula**. In both cases, we will often just remember that we are sending bits around everywhere and drop the "Boolean" qualifier. The difference between circuits and formulas is that circuits can share computation.

The in-degree of a gate is called its **fan-in**, and similarly the out-degree is called its **fan-out**. We will often talk about the fan-in or fan-out of a circuit as a whole, meaning the maximum fan-in or fan-out of any gate. For example, formulas have fan-out 1.

Typically, a single gate will be designated the **output** of a circuit. A circuit computes a function by setting the inputs to the circuit to the inputs to the function, computing the output of each gate based on its inputs and the function it implements, and taking the output of the output gate as the value of the function.

The **depth** of the circuit is the length of the longest path from an input to the output. The **size** of a circuit is the number of gates.

Circuits are attractive as a model of computation because they are in a sense more concrete that Turing machines. They also allow us to model parallel computation, since a low-depth circuit corresponds in a natural way to a highly-parallel computation.

Because each circuit has a fixed number of inputs, circuits can't generally compute languages of the sort we are used to studying in complexity theory. So we instead consider **circuit families** $\{C_n\}$, where each circuit C_n is used for inputs of size n. The language decided by a circuit family is $\left\{x \;\middle|\; C_{|x|}(x) = 1\right\}$. Using this approach, we can define complexity classes based on what can be computed by circuit families subject to various size, depth, and fan-in restrictions.

Some of these complexity classes are weak enough that we can prove that actual, non-contrived functions don't live in them, often using techniques that don't relativize.[1] Unfortunately, other obstacles come into play when we try to get lower bound results for stronger circuit complexity classes, such as those corresponding to **P**.

10.1 Polynomial-size circuits

We've seen before when looking at **P**-complete problems (§9.6.1.2) that any computation by a Turing machine can be encoded by a circuit made up of basic Boolean logic gates like AND, OR, and NOT. For a computation that takes $T(n)$ time and uses $S(n)$ space, the circuit will have size $O(T(n) \cdot S(n))$ and depth $O(T(n))$. Both of these quantities will be polynomial if $T(n)$ is polynomial.

This suggests that we can think of **P** as a circuit-complexity class by restricting our attention to families of polynomial-size circuits. This almost works, but as often happens, the simplest definition gives a class that is too powerful.

The reason is that when using a family $\{C_n\}$ of circuits, we have to ask where these circuits come from. If they are arbitrary, even though each C_n may be restricted to size $O(n^c)$, it still may be that C_n encodes a lot of information about n. For example, there is a family of circuits of size 1 where each C_n is a constant circuit, and C_n happens to output 1 if and only if the i-th Turing machine M_i halts on an empty input tape. So polynomial-size circuits by themselves are not restricted enough to avoid

[1] There might not seem like there is much room to sneak an oracle into a circuit, but unless we are very careful to specify exactly what gates we allow (and use this restriction in our proofs), individual gates or small groups of gates working together could do all sorts of tricksy things, including making oracle calls. This is why any technique for proving lower bounds on circuits that relativizes won't help with $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$, because there are oracles A and B that separate small circuits from $\mathbf{NP}^A$ and that supercharge them so that they can easily solve the wimpy problems in $\mathbf{NP}^B$ [Wil85].

computing undecidable predicates, but they do give us a starting point for representing **P**.

10.1.1 P/poly

Polynomial-size circuit families decide languages in a complexity class called **P/poly**, the class of languages that can be computing in polynomial time with polynomial **advice**. Advice is a little bit like the certificates or hints given to nondeterministic Turing machines, with the differences that (a) the advice string can only depend on the size of the input $n = |x|$, and (b) the advice string is trustworthy, in the sense that the Turing machine that handles the advice is not required to do anything sensible if it gets the wrong advice.

We could provide the advice on a separate read-only input tape, but instead we typically just append it to the input. Formally, a language L is in $\mathbf{TIME}(f(n))/g(n)$ if there is machine M and a function $\alpha : \mathbb{N} \to \{0,1\}^*$ where $|\alpha(n)| \leq g(n)$ and $M(x, \alpha(n))$ accepts in $O(f(n))$ steps when $x \in L$ and rejects in $O(f(n))$ steps when $x \notin L$. The class **P/poly** can then be defined as $\bigcup_{a,b} \mathbf{TIME}(n^a)/n^b$.

Given a family $\{C_n\}$ of polynomial-size circuits, the language decided by this family is in **P/poly**: let $\alpha(n)$ be a description of C_n. In the other direction, if M is a poly-time Turing machine that uses advice, we can simulate it by constructing a family of circuits where each C_n has $\alpha(n)$ baked into it. This shows that **P/poly** captures exactly the languages computable by a family of poly-size circuits.

Note that we can always make the advice empty: this gives $\mathbf{P} \subseteq \mathbf{P/poly}$. In §10.2 we will show how to put some constraints on the advice to get a class of polynomial-size circuits that decide exactly the languages in **P**.

10.1.2 Information-theoretic bounds

P/poly is pretty powerful, but it's not all-powerful: almost all functions can't be computed by polynomial-size circuits.

To keep things simple, let's assume that our circuits consist only of 2-input AND gates and NOT gates, since we can simulate AND and OR gates with polynomial fan-in using $O(\log n)$ of these gates and still have a polynomial-size circuit. We can reduce further to 2-input NAND gates (which compute $\neg(x \wedge y)$) since we can build a NOT out of a NAND by wiring both inputs together and build an AND out of two NANDs by using the second one as a NOT to invert the output of the first.

We can specify an all-NAND circuit of size $f(n)$ by listing which gates or inputs supply the input to each gate. This gives us $(f(n)+n)^2$ choices per gate, or $(f(n)+n)^{2f(n)}$ choices total.[2] The number of Boolean functions on n inputs is 2^{2^n}. Taking logs of both quantities gives us $O(f(n)\log f(n))$ bits to represent a circuit of size $f(n)$ (assuming $f(n) = \Omega(n)$) and 2^n bits to represent a function on n inputs. For $f(n) = n^c$, we get $O(n^c \log n) = o(2^n)$ circuits, so most Boolean functions have no circuits of this size.

In fact, this is true for any size bound that is subexponential. On the other hand, at $O(n \cdot 2^n)$ size, we can build a circuit that consists of 2^n separate circuits that each recognize one input using $O(n)$ gates, and a gigantic OR that combines the outputs of all the circuits that recognize a positive input. So exponentially-large circuits can do anything, making **EXP/exp** equal to the set of all functions. For this reason, we will not study **EXP/exp** much.

10.1.3 The Karp-Lipton Theorem

The lower bound in the preceding section is not as exciting as it could be, since it just says that there are many bad languages out there for **P/poly**, but it doesn't say which. The Karp-Lipton Theorem gives evidence that SAT is one of these functions, assuming we don't believe the poly-time hierarchy collapses.

Theorem 10.1.1 (Karp-Lipton [KL80])**.** *If* $\mathbf{NP} \subseteq \mathbf{P/poly}$*, then* $\mathbf{PH} = \mathbf{\Sigma}_2^p$.

Proof. The idea is to show that $\mathbf{\Pi}_2$ SAT, the problem of testing if a formula of the form $\forall x \exists y \Phi(x, y)$ is true, is in $\mathbf{\Sigma}_2^p$.

We do this by guessing a circuit C_n that solves SAT for n big enough to encode $\Phi(x, y)$ for any fixed x and possibly partially-fixed y. Given a polynomial-size circuit C_n that claims to solve SAT, and a circuit $\Phi(x, -)$ it says yes to, we can extract the y that makes $\Phi(x, y)$ true by testing each possible bit one at a time, and then verify for ourselves that the resulting y actually works in $\Phi(x, y)$ (given a fixed x). [3] On the other hand, if C_n doesn't actually work for some particular $\Phi(x, -)$ or its specialization, we can detect this in deterministic polynomial time and refuse to accept.

So now we construct the $\mathbf{\Sigma}_2^p$ machine $\exists C_n \forall x [C_n$ yields a satisfying y for $\Phi(x, y)]$. If $\forall x \exists y \Phi(x, y)$ is true, then when we guess C_n correctly (which we will

[2] We are overcounting a bit here, but it won't matter.

[3] The property of SAT used to extract a satisfying assignment is know as **self-reducibility**. This means that a formula $\Phi(x_1, x_2, \ldots, x_n)$ is satisfiable if and only if at least one of the simpler formulas $\Phi(0, x_2, \ldots, x_n)$ or $\Phi(1, x_2, \ldots, x_n)$ is, meaning we can reduce an instance of SAT to smaller instances of the same problem.

do on at least one branch under the assumption $\mathbf{NP} \subseteq \mathbf{P/poly}$), we will get $\forall x[C_n$ yields a satisfying y for $\Phi(x,y)]$ true, and the machine will accept. If $\forall x \exists y \Phi(x,y)$ is false, then no matter what we guess for C_n, $\forall x[C_n$ yields a satisfying y for $\Phi(x,y)]$ will be false, and the machine will reject.

This gives $\mathbf{\Pi}_2^p \subseteq \mathbf{\Sigma}_2^p$, which also means $\mathbf{\Pi}_2^p = \mathbf{\Sigma}_2^p$. So now given any class above $\mathbf{\Pi}_2^p = \mathbf{\Sigma}_2^p$ in the polynomial-time hierarchy, we can reverse the last two quantifiers using this equivalence, and then drop its number of quantifiers by one by combining the third- and second-to-last quantifiers. So the hierarchy collapses to $\mathbf{\Sigma}_2^p$. □

The Karp-Lipton Theorem suggests that even though $\mathbf{P/poly}$ is in a sense too powerful a class to use in real life, it might still be possible to show $\mathbf{P} \neq \mathbf{NP}$ by proving the just as plausible $\mathbf{P/poly} \neq \mathbf{NP}$. This approach was pretty popular in the late 1980s, in part because of a proof by Razborov that poly-size **monotone circuits** could solve CLIQUE [Raz85]. These are circuits with AND and OR gates, but no NOT gates, and (unlike, say, SAT), it is possible to solve CLIQUE using a big enough monotone circuit (take an OR of ANDs over all sets of edges that might be cliques). So the hope was that some sort of clever non-relativizing circuit sneakery could prove a similar result for general circuits. This hope was ultimately dashed by the Razborov-Rudich **natural proofs** result [RR97], which ruled out a large class of lower bound arguments based on finding testable properties of functions that would imply hardness. (We'll come back to this in Chapter 11.)

On the other hand, Karp-Lipton gives some interesting conditional results. One of these, which Karp and Lipton attribute to Albert Meyer in their paper, is that if $\mathbf{EXP} \subseteq \mathbf{P/poly}$, then $\mathbf{EXP} = \mathbf{\Sigma}_2^p$. But then $\mathbf{P} \neq \mathbf{\Sigma}_2^p$ (Time Hierarchy Theorem), which implies $\mathbf{P} \neq \mathbf{NP}$. So we have the unlikely possibility of being able to prove $\mathbf{P} \neq \mathbf{NP}$ by showing that $\mathbf{P/poly}$ is much more powerful than it has any right to be.

10.2 Uniformity

The reason we get $\mathbf{P/poly}$ instead of $\mathbf{P}$ out of poly-size circuits is that our circuit families are **non-uniform**: we get to pick a different circuit C_n for each input n, and this choice is arbitrary. If we have to actually compute C_n from n, we can restore some sanity.

A circuit family $\{C_n\}$ is **logspace-uniform** or just **uniform** if there is a Turing machine M that computes a description of C_n from 1^n in time logarithmic in n. The description will be a labeled graph, so M needs

to output (1) a count of the number of gates in the circuit, equivalent to counting the number of vertices in the graph; (2) an adjacency list that gives the wires in the circuit; and (3) a label for each gate saying which function (AND, OR, NOT) it computes.[4] As with other computations by log-space machines, we can assume either that the machine is explicitly writing the entire description to a write-only output tape or implicitly providing the description as a Boolean function $f(n, i)$ that gives the i-th bit of the output on input 1^n.

If we consider only logspace-uniform families of circuits, then the families of polynomial size give exactly the class **P**. The reason is that (a) we can simulate any such circuit in **P**, by performing a $\mathbf{L} \subseteq \mathbf{P}$ computation to generate a description of the appropriate C_n, and then evaluating it in the usual way; and (b) we can simulate a poly-time Turing machine M by a family of polynomial-size circuits generated in log space, since each circuit just computes the value of the tape cells, head state, and so forth as in Cook's theorem, and writing out such a circuit will only require logarithmic space to keep track of the loop indices for times and positions.

10.3 Bounded-depth circuits

In addition to restricting the size of a circuit, we can also restrict its depth, the maximum length of any path in the circuit. This gives rise to two important families of complexity classes:

1. $\mathbf{AC}^i$ is the class of languages computed by polynomial-size circuits with depth $O(\log^i n)$.

2. $\mathbf{NC}^i$ is the class of languages computed by polynomial-size circuits with constant fan-in and depth $O(\log^i n)$.

The names **AC** and **NC** are short for **alternating class** and **Nick's class**, respectively.

The intent of these classes is to model problems that can be computed quickly in parallel: a low-depth circuit corresponds to a computation where many activities may be going on at the same time, but the path from any input to an output is short.

[4] If we are willing to limit ourselves to NAND gates, we can compute anything we can compute with AND, OR, and NOT, and skip the labeling. But having explicit AND and OR gates will turn out to be convenient when we look at various restricted circuit complexity classes.

Often we will insist that these circuits are log-space uniform, but this requirement is not part of the standard definitions, and many results about circuit complexity don't require uniformity.

A common convention is to use De Morgan's Laws to push all NOT gates to the inputs, and insist on alternating layers of AND and OR gates. This can be done to either $\mathbf{AC}^i$ or $\mathbf{NC}^i$ circuits, and at most doubles the depth (since we may have to stick a layer of dummy 1-input AND or OR gates in between layers of gates of the same type).

It holds trivially that $\mathbf{NC}^i \subseteq \mathbf{AC}^i$, since any polynomial-size circuit of depth $O(\log^i n)$ with bounded fan-in is also a polynomial-size circuit of depth $O(\log^i n)$ with unbounded fan-in. It is also not hard to show that $\mathbf{AC}^i \subseteq \mathbf{NC}^{i+1}$: replace each AND or OR gate with fan-in k by a tree of AND or OR gates of depth $O(\log k)$.

10.3.1 Parallel computation and NC

The union $\bigcup_{i=1}^{\infty} \mathbf{NC}^i$ of these classes is called **NC**, and is generally taken to be the class of problems that can be computed efficiently in parallel. Note that **NC** is the same as $\bigcup_{i=0}^{\infty} \mathbf{AC}^0$.

The **P**-complete languages (§9.6.1.2) have roughly the same relation to **NC** as the **NP**-complete languages have to **P**: while a **NP**-complete problem is one that we don't expect to be able to compute efficiently at all, a **P**-complete language is one that we expect to be able to compute efficiently, but not in parallel.

10.3.2 Relation to L and NL

A logspace-uniform family of circuits in $\mathbf{NC}^1$ can be simulated in **L**. The basic idea is to use game-tree search, where we are careful about storage. There are basically three things we need to keep track of:

1. An index of the current gate we are working on. This takes $O(\log n)$ bits.

2. A stack describing what path we are currently evaluating. Each edge takes $O(1)$ bits to specify (because we know the parent gate, and this gate has only $O(1)$ inputs) and we have $O(\log n)$ edges per path. So another $O(\log n)$ bits.

3. Storage for communicating with and executing the logspace subroutine for generating the circuit. Also $O(\log n)$ bits, where we use the usual trick of not storing the entire output but only the bits we need.

This doesn't show all of $\mathbf{NC}^1$ is in $\mathbf{L}$, but it does show that logspace-uniform $\mathbf{NC}^1$ is in $\mathbf{L}$. It also doesn't generalize to $\mathbf{AC}^1$: for $\mathbf{AC}^1$, our stack would need $O(\log^2 n)$ space, since we have polynomially many possible inputs at each level.

In the other direction, we can show that $\mathbf{NL} \subseteq \mathbf{AC}^1$, and indeed any $\mathbf{NL}$ function can be computed by a logspace-uniform $\mathbf{AC}^1$ circuit. The idea is to first reduce any problem in $\mathbf{NL}$ to STCON and then solve STCON in $\mathbf{AC}^1$ by repeated matrix squaring, a technique developed for transitive closure by Fischer and Meyer [FM71].

Let A_{ij}^t be a Boolean variable representing whether there is a sequence of t transitions that take us from configuration i to configuration j. Suppose that our logspace machine always finishes in n^c steps. For convenience, let us also structure it so that it has a single accepting configuration a that persists in the sense that $a \to a$ is the only transition leaving a. Then we can determine if the machine accepts by building a circuit to A_{ij}^t for some $t \geq n^c$.

To generate the circuit, we construct a sequence of $O(\log n)$ layers computing $A_{ij}^{2^k}$ for each k, i, and j. The bottom layer ($k = 0$) just computes if $i \to j$ is a transition that is consistent with the input, and we can generate this layer by enumerating all i and j ($O(\log n)$ space for each), and determining which input value if any is necessary to have an $i \to j$ transition. (The subcircuit emitted for A_{ij}^1 will in fact always be one of a constant 0, a constant 1, or x_ℓ or $\neg x_\ell$ for some input bit x_ℓ.) For higher layers, we use the rule

$$A_{ij}^{2^{k+1}} = \bigvee_\ell \left(A_{i\ell}^{2^k} \wedge A_{\ell j}^{2^k} \right).$$

This is a depth-2 circuit using a linear-fanin OR over a bounded-fanin AND, so we can build $O(\log n)$ layers like this and stay in $\mathbf{AC}^1$. This shows that $\mathbf{NL} \subseteq \mathbf{AC}^1$, and if we are careful about checking that the circuit construction can in fact be done in log space, we have $\mathbf{NL}$ is contained in logspace-uniform $\mathbf{AC}^1$.

10.3.3 Barrington's Theorem

Barrington's Theorem [Bar89] shows that the Boolean functions computable in $\mathbf{NC}^1$ are precisely the functions that can be computed by **branching programs** of width 5.

A branching program is a directed acyclic graph where each non-terminal vertex is labeled by an input bit position, and has exactly two outgoing edges labeled 0 and 1. We execute a branching program by starting at an initial

vertex, and at each step we (a) look at the input bit x_i given by the label on the current vertex, and then (b) move to the successor pointed to by the 0 or 1 edge, depending on the value of x_i. When we reach a terminal vertex, we take the label on that vertex (0 or 1) as the output of the program.

A **bounded-width branching program** is one in which the vertices can be arranged into layers, so that each vertex in layer i has successors in layer $i+1$, and there are at most k vertices in each layer. Here k is the **width**width!of a branching program of the branching program.

We can think of a bounded-width branching program as a little like a finite-state machine where the transition function changes at each step. In this view, the k nodes that the program might be in at layer t are treated as k states that the machine might be in at time t, and we think of the labeled outgoing edges as representing functions f_0^t and f_1^t that map the state q_t at time t to the new state q_{t+1} at time $t+1$. So we can program these things by choosing a sequence of inputs x_t and functions f_0^t and f_1^t, and then just iterate applying $f_{x_t}^t$ to the state each each time t starting with some initial state.

Taking one more step down the rabbit hole, we can choose to make these functions invertible, which makes f_0^t and f_1^t both elements of S_k, the **symmetric group** on k elements. Composition of these functions corresponds to multiplication of the group elements, so we can think of the entire computation as evaluating a gigantic word $f_{x_1}^1 f_{x_2}^2 \dots f_{x_{T(n)}}^{T(n)}$ in S_k. So now our programming problem is one of choosing the individual group elements f_0^t and f_1^t and inputs x_t so that this multiplication does what we want.

We will specifically use S_5, and use words that evaluate either to cycles or to the identity. A standard convention in group theory writes an element of S_k as a product of cycles, where each cycle is given by a list of its elements in order in parentheses. For example, (12345) represents the cycle $1 \to 2 \to 3 \to 4 \to 5 \to 1$, while (1)(2)(3)(4)(5) represents the identity e (which sends each position back to itself; it's worth mentioning that usually we would omit trivial cycles with just one element). We can multiply group elements by chasing individual position around the cycles. So, for example, the product (12345)(13542) = (1)(253)(4) = (253), since $1 \to 2 \to 1$, $2 \to 3 \to 5$, and so on.

Each input bit x_i will be represented by a choice of two group elements ξ^{i0} and ξ^{i1}, depending on its value. For a function $f(x_1, \dots, x_n)$ of the input bits, we will say that a word $w = \xi_1 \xi_2 \dots \xi_m$ α-represents f if $w = \alpha$ when $f = 1$ and $w = e$ when $f = 0$. In this representation, we use x_i to refer to

the i-th bit examined by the branching program rather than the i-th bit in the actual input; this saves some extra subscripts and allows us to use a different representation for different occurrences of a particular input bit.

If α and β are cycles, we can convert an α representation to a β representation by conjugation; there is a group element γ such that $\gamma\alpha\gamma^{-1} = \beta$, and when $w = \alpha$ we get $\gamma w\gamma^-1 = \beta$ and when $w = e$ we get $\gamma w\gamma^{-1} = \gamma\gamma^{-1} = e$. Note that this doesn't actually increase the length of w, since we can bake γ and γ^{-1} into the representations of the first and last bits examined by w.

Now we want to represent the gates in our circuit by products of words. Negation is easy: if w α-represents some sub-circuit C, then $w\alpha^{-1}$ α^{-1}-represents $\neg C$. For AND, we take advantage of the fact that there exist cycles $\rho = (12345)$ and $\sigma = (13542)$ in S_5 whose **commutator** $\tau = \rho\sigma\rho^{-1}\sigma^{-1} = (12345)(13542)(54321)(24531) = (13254)$ is also a cycle. So now to compute $C \wedge D$, we let u ρ-represent C, v σ-represent D, and then $w = uvu^{-1}u^{-1}$ τ-represents $C \wedge D$, since if $C = 0$ we get $w = evev^{-1} = vv^{-1} = e$, if $D = 0$ we get $w = ueu^{-1}e = uu^{-1} = e$, and if both are 1 we get $w = \rho\sigma\rho^{-1}\sigma^{-1} = \tau$.

To convert an entire depth d circuit, first convert it to a depth-d formula, then apply the above construction to each subcircuit, adjusting the representation as needed using conjugations. This gives a word w of length 4^d over S_5 that α-represents the computation of the original circuit for some α, and we can compute this using a width-5 branching program by taking any initial state and seeing if it gets mapped elsewhere or not by w.

In the other direction, given a branching program of width k, we can represent it as a word over the semigroup of functions from $\{1 \dots k\}$ to $\{1 \dots k\}$, and use a circuit of depth $O(\log|w|)$ to compute its value by splitting w in half, computing the mapping corresponding to each half, then composing the mappings together (which we can do in constant additional depth). This shows that $O(\log n)$ depth circuits are exactly equivalent in power to polynomial-length width-5 branching programs.

I find this result horrifying. In both directions of the construction, the thing we build has no obvious connection to the thing we started with, and if asked after seeing the state of the branching program coming out of uv what that state means, we would have to admit that we have no ability whatsoever to decode it without running it back through $u^{-1}v^{-1}$. So the answer to pretty much any attempted proof of anything in complexity theory that starts with "I understand how programs work, and at this stage of the program the state must mean this" is "Oh yeah? And what about Barrington's Theorem?"[5]

[5]In fact, applying Barrington's Theorem is a critical step in many known cryptographic techniques for obfuscating code, although in fairness the real obfuscation comes from

10.3.4 PARITY $\notin \mathbf{AC}^0$

Want to show $\mathbf{AC}^0 \neq \mathbf{NC}^1$ by showing PARITY (odd number of inputs are 1) is not in $\mathbf{AC}^0$. This also shows MAJORITY (more than half the inputs are 1) is not in $\mathbf{AC}^0$, since we can use pairs of MAJORITY gates to test if $\sum x_i = k$ for any fixed k and use an OR of $n/2$ of these equality testers to compute PARITY [FSS84].

The first proofs that PARITY $\notin \mathbf{AC}^0$ were given by Furst, Saxe, and Sipser [FSS84] and Ajtai [Ajt83]. We'll give two proofs of this fact, one (§10.3.4.1) based on Håstad's Switching Lemma [Has86], which simplifies circuits by fixing some of their inputs; and one (§10.3.4.3) based on Razborov's approximation of $\mathbf{AC}^0$ circuits by low-degree polynomials over a finite field [Raz87] together with Smolensky's proof that such polynomials can't approximate parity [Smo87].

10.3.4.1 Håstad's Switching Lemma

First, let's start with an easy warm-up: A depth-2 circuit can only compute parity if it has at least 2^{n-1} gates. Proof: If it's an OR of ANDs, any AND with fewer that n inputs gives a false positive on some input, and we need at least 2^{n-1} ANDs on n inputs to cover all odd-parity inputs. If it's an AND of ORs, then do the same thing with false negatives.

Håstad's Switching Lemma uses a **random restriction** to convert an OR of small ANDs into an AND of small ORs. A random restriction fixes a random subset of the inputs to the circuit to random values. By repeatedly switching ANDs with ORs, we can flatten an $\mathbf{AC}^0$ circuit down to something that obviously can't compute parity on the surviving unrestricted variables. This will be be a problem because a random restriction of parity is still parity, and we won't knock out everything.

Håstad's Switching Lemma actually shows something stronger. Given a DNF formula of **width!of a DNF formula** w, meaning an OR of ANDs where each AND has at most w literals, hitting it with a random restriction gives us a **decision tree** of low depth with high probability. This can then be converted into a low-width CNF formula if needed.

Define a random s-restriction as a choice of random values for a random subset of $n - s$ variables, where each of the $\binom{n}{s}2^{n-s}$ possible restrictions are equally likely. (Note that s indicates how many variables survive.) The **restriction** $f|_\alpha$ of f to α is the function on s variables given by fixing the other inputs to f according to α.

disguising the S_5 elements as more complicated algebraic objects. [GGH+16]

Lemma 10.3.1 (Håstad's Switching Lemma)**.** *If f is computed by a w-DNF, and α is a random s-restriction with $s = \sigma n \leq n/5$, then for any $d \geq 0$, the probability that $f|_\alpha$ has no decision tree of depth d or less is at most $(10\sigma w)^d$.*

Proof. The proof is by showing that the set of bad restrictions β that require depth more than d is small. This is done by encoding each such β using an $(s-d)$-restriction supplemented by a small number of extra bits of information. Since the number $\binom{n}{s-d}2^{n-s+d}$ of $(s-d)$-restrictions is much small than the number $\binom{n}{s}2^{n-s}$ of s-restrictions, and the extra bits don't add much to the total count, this will show that the proportion of bad restrictions is small.

Given f and a bad β, we'll define a **canonical decision tree** that may or may not be the best possible decision tree for $f|_\beta$, but that is certainly no better than the best possible tree. In particular, the canonical description tree will contain a path of length $d+1$ or more, and we'll use this path to extract our succinct description of β.

Order the terms $T_1 \vee \ldots T_\ell$ in the w-DNF representation of f in some standard way. We say that β **kills** T_i if β forces T_i to 0. It **fixes** T_i if it instead forces T_i to 1. Since β is bad, it can't fix any term or kill all the terms, because then $f|_\beta$ would be constant and computable by a depth-0 tree.

The canonical decision tree is generated recursively. Start with first term T_i not killed by β, and let T_i have d_i free variables. Build a complete depth-d_i tree that has 2^{d_i} leaves corresponding to each assignment of these variables. Give the leaf of the tree corresponding to a true assignment to T_i the value 1, and recurse on the terms after T_i to get subtrees for each of the other leaves. Note that in this recursion, we restrict the variables we've already examined, so we may kill (or fix) many of the terms after T_i depending on which values we saw. But at the end we get a decision tree that computes $f|_\beta$.

Somewhere in this tree is a path of length $d+1$ or more. Let P be the prefix of this path of length d, and construct a new $(s-d)$-restriction π by adding to β the values of the variables on P.

For the last step, we'll show how to recover β from a *different* $(s-d)$-restriction γ without too much extra information. What γ does is fix all the terms $T_{i_i}, T_{i_2}, \ldots$ whose subtrees are traversed by π, by setting the variables in each T_{i_j} to whatever will make T_{i_j} true. To be able to recover π and β from this, we also write down for each term the set of variables we fixed and what values they have in π: this requires no more than $d_i(\lg w + 2)$ bits if we are careful, for a total of $d(\lg w + 2)$ extra bits over all terms. The extra information lets us reconstruct π from γ.

To recover β, we walk through $f|_\gamma$ looking for clauses fixed by γ. For

each such clause, the extra bits tell us which variables to remove from γ to get β. This shows that the number of bad restrictions is bounded by $\binom{n}{s-d}2^{n-s+d}2^{d(\lg w+2)} = \binom{n}{s-d}2^{n-s+d}(4w)^d$.

Dividing by the total number of restrictions gives a probability of a bad restriction bounded by

$$\begin{aligned}
\frac{\binom{n}{s-d}2^{n-s+d}(4w)^d}{\binom{n}{s}2^{n-s}} &= \frac{\frac{n!}{(s-d)!(n-s+d)!)}}{\frac{n!}{s!}(s-d)!}(8w)^d \\
&= \frac{(s)_d}{(n-s+d)_d}(8w)^d \\
&\leq \left(\frac{s}{n-s+d}\right)^d(8w)^d \\
&\leq \left(\frac{s}{n-s+d}\right)^d(8w)^d \\
&\leq \left(\frac{\sigma}{1-\sigma}\right)^d(8w)^d \\
&\leq \left(\frac{5}{4}\sigma\right)^d(8w)^d \\
&\leq (10\sigma w)^d.
\end{aligned}$$

□

This shows that an OR of small ANDs usually turns into an AND of small ORs when hit by a random restriction. By negating outputs and inputs, we can apply the same technique to turn an AND of small ORs into an OR of small ANDs. We'll alternate between these to flatten $\mathbf{AC}^0$ circuits, as explained below.

10.3.4.2 Application to PARITY

Now we want to use the lemma to beat up an $\mathbf{AC}^0$ circuit C that claims to compute PARITY.

To get the process off the ground, we need to restrict the fan-in of our circuit to 1 at the bottom level.[6] We can trivially do this by adding an extra layer of 1-input gates. We will also assume for simplicity that the circuit is a tree, and that we strictly alternate OR and AND levels, with all negations pushed to the inputs, and AND at the bottom. Let n^b bound the size of the resulting circuit.

[6] Which I didn't do in class on 2017-03-1, leading to much embarrassment.

We now hit the circuit with a random $\sqrt{n}$-restriction, meaning that we set $\sigma = n^{-1/2}$ (this will let us use a union bound over all gates in C later). We'll choose d so that $(10\sigma w)^d = (10n^{-1/2})^d = o(n^{-b})$; $d = 4b$ works, giving a probability of getting depth greater than $4b$ bounded by $10^{4b}n^{-2b} = o(n^{-b})$ since the extra n^{-b} factor starts eating up the constant around $n = 10000$ or so.

If we look at one OR gate at the second level of the circuit and its pile of feeder 1-input AND gates, after applying the restriction we can, with high probability, replace it with an AND of $4b$-input OR gates. All of these ANDs feed into ANDs that used to be at the third level of the circuit, and we can absorb them into their successors without changing the width of the new bottom layer of OR gates. We have just shortened the circuit by one level, at the cost of increasing the width of the input level from 1 to $4b$ and reducing the number of free inputs from $n_0 = n$ to $n_1 = n^{1/2}$.

In general, we'll do the square-root trick whenever we knock out the bottom level. This gives $n_i = n^{2^{-i}}$ as the number of surviving inputs after each level. For the width, let $w_i = 4^i b$. We've already shown that $w_0 = b$ and $w_1 = 4b$ are high-probability bounds on the width after 0 and 1 layers are removed. More generally, the probability that we get more than w_{i+1} width after removing i layers is bounded by

$$\begin{aligned}
\left(10n_i^{-1/2}w_i\right)^{w_{i+1}} &= \left(10n^{-2^{-i-1}}4^i b\right)^{4^{i+1}b} \\
&= (10b)^{4^{i+1}b}2^{2i(i+1)b}n^{-2^{-i-1}4^{i+1}b} \\
&\le (10b)^{4^{i+1}}2^{2i(i+1)}n^{-2^{i+1}b} \\
&= (10b)^{4^{i+1}}2^{2i(i+1)}n^{-2^{i+1}b}
\end{aligned}$$

Since everything up to the n term is a constant, for sufficiently large n it happens that $n^{-2^{i+1}b}$ not only blows it away but puts what's left well under n^{-b}.[7]

Let h be the height of the original circuit. After $h-2$ restrictions, we get, with nonzero probability, a depth-2 circuit with its first layer width bounded by $4^h b$, a constant. Such a circuit must have size at most $1 + (2n+1)^{4^h b}$, a polynomial in n, since with bounded width we can only get so many combinations of inputs before we start getting duplicate gates we can remove. So if PARITY is in $\mathbf{AC}^0$, this construction gives us circuits for computing parity on arbitrarily large numbers of inputs that have depth 2 and polynomial size. But we've already shown this to be impossible. It follows that PARITY $\notin \mathbf{AC}^0$.

[7] We probably could have used a tighter bound on w_i, but $4^i b$ is easy to write down.

For the last step, we can encode the argument that a constant-width depth-2 circuit doesn't compute parity as a further restriction: if we restrict the inputs to the depth-2 circuit to ones that fix one of the bottom-layer gates, we get a constant function. So what we are really showing is that for sufficiently large n, any $\mathbf{AC}^0$ function is constant over some reasonably large subcube of the hypercube of all possible inputs of size n.

This is an example of a proof that doesn't relativize. The reason is that random restrictions don't do anything sensible to arbitrary oracle gates, so we really are using the fact that our original $\mathbf{AC}^0$ circuit is built from AND and OR gates. In the following section, we will see a different proof that also doesn't relativize, because again we are looking at specific properties of AND and OR gates.

10.3.4.3 Low-degree polynomials

A different proof, due to Razborov and Smolensky, shows that PARITY $\notin \mathbf{AC}^0$ based on approximating constant-depth circuits by low-degree polynomials over $\mathbb{Z}_p$ for some prime $p \neq 2$.[8]

The nice thing about this proof is that it (a) gives a lower bound on approximating parity as well as computing parity, and (b) works even if we throw in mod-p gates (that return 0 if the sum of their inputs is a multiple of p and 1 otherwise), for any single prime $p \neq 2$. Throwing in mod-p gates gives us the class $\mathbf{AC}^0[p]$, which is a bit stronger than $\mathbf{AC}^0$ for prime p (for example, $\mathbf{AC}^0[2]$ can compute parity), but probably not by much.

We are going to start by approximating our depth-h circuit by a polynomial that computes a function $\{0,1\}^n \to \{0,1\}$. This 0–1 **basis** is convenient for constructing the approximation, although when we want to show that the polynomial doesn't approximate parity it will be handy to switch to a ± 1 **basis**, which will involve a change of variables to turn the polynomial into a function $\{-1,+1\}^n \to \{-1,+1\}$.

Over the 0–1 basis, here is how to encode a depth-h, size N circuit as a degree $((p-1)\ell)^h$ polynomial that computes the right answer on any fixed input with probability at least $1 - N \cdot 2^{-\ell}$:

1. Encode an input x_i as the variable x_i.

[8] The proof has two parts: Razborov [Raz87] showed that low-degree polynomials over $\mathbb{Z}_p$ can approximate low-depth circuits, and Smolensky [Smo87] shows that low-degree polynomials over $\mathbb{Z}_p$ couldn't approximate PARITY, which implies that constant-depth circuits can't either. Both parts are needed to show PARITY $\notin \mathbf{AC}^0[p]$, so it has become conventional just to refer to the full result as the **Razborov-Smolensky Theorem**.

2. Encode $\neg p$ as $1 - p$. This does not change the degree.

3. For OR, we will do something sneaky to avoid blowing up the degree too much.

   Supposing we are computing $\bigvee_{j=1}^{k} p_{i_j}$, where each p_{i_j} has degree d. Take a random subset S of the i_j, and compute $\sum_{i_j \in S} p_{i_j}$. This has degree d as well, and if all p_{i_j} are 0, it is 0, and if at least one p_{i_j} is not zero, then including the last nonzero p_{i_j} or not in S changes the value of the sum, meaning that there is at least a $1/2$ chance that the sum is nonzero. We can convert this nonzero sum back to 1 using Fermat's Little Theorem: $\left(\sum_{i_j \in S} p_{i_j}\right)^{p-1} = 1 \pmod{p}$ if and only if $\sum_{i_j \in S} p_{i_j} \neq 0 \pmod{p}$. This gives us a polynomial that is 0 for the all-0 input and 1 with probability at least $1/2$ for any nonzero input.

   Now blow this up by choosing ℓ random S_i and let $p = 1 - \prod_{i=1} \ell \left(\left(1 - \sum_{i_j \in S} p_{i_j}\right)^{p-1} \right)$. This has degree $d(p-1)\ell$ and computes OR with probability at least $1 - 2^{-\ell}$.

4. For AND, use De Morgan's Laws and NOT to reduce to OR; the degree is again $d(p-1)\ell$ with probability of error at most $2^{-\ell}$.

5. For mod-p, use $(\sum p_i)^{p-1}$. Degree is $d(p-1) \leq d(p-1)\ell$ and there is no error.

Applying this over depth h gives a degree bound of $((p-1)\ell)^h$, since we go up by at most a factor of $(p-1)\ell$ at each level. Taking a union bound over all gates gives the $N \cdot 2^{-\ell}$ bound on error. This means that each of the 2^n possible inputs contributes $N \cdot 2^{-\ell}$ bad outputs on average, for a total of $2^n \cdot N \cdot 2^{-\ell}$ bad outputs on average. Since this is an average, some specific choice of random subsets must do no worse, showing that there exists a polynomial of degree $((p-1)\ell)^h$ over $\mathbb{Z}_p$ that computes the same function as the original circuit on all but $2^n \cdot N \cdot 2^{-\ell}$ inputs.

Now we need to show that this polynomial can't approximate parity very well. To do so, it is helpful to switch to a ± 1 basis by replacing each occurrence of x_i with $-(y_i + 1) \cdot 2^{-1} = -(y_i + 1) \cdot \frac{p+1}{2}$ and similarly adjusting the output of the resulting polynomial q' by changing it to $q = 1 - 2q'$; this effectively sends every bit b to $(-1)^b$, making parity just the product of all the inputs. Note that neither step affects the degree of the polynomial: we still have a degree $((p-1)\ell)^h$ polynomial that is a good approximation to the circuit.

Now to show that such a polynomial can't be a good approximation to parity, which will show that the circuit doesn't compute parity. Let G be the set of inputs y in $\{-1,+1\}^n$ on which our degree-d polynomial q computes parity. Consider the set of all functions $f : G \to \{-1,+1\}$. For each such f, there is a polynomial $f_p(y)$ (of degree n) over that computes f exactly. Because the inputs to f are all ± 1, we can assume that each monomial in f_p is of the form $\prod_{i \in S} y_i$, since any y_i^2 that appears can be replaced by 1. Assuming we only care about inputs in g, we can make the same assumption about q. We will now use q to smash down f_p to have degree close to $n/2$, and argue that there aren't enough of these lower-degree polynomials to cover all possible $f : G \to \mathbb{Z}_p$.

Take any monomial $\prod_{i \in S} y_i$ where $|S| > n/2$. This is equal to $\prod_{i \notin S} y_i \prod_{i=1}^{n} y_i = \left(\prod_{i \notin S} y_i\right) q(y)$ for any $y \in G$. By applying this transformation to all the high-degree monomials in f_p, we get a new polynomial that computes the same function on G and has degree at most $n/2 + ((p-1)\ell)^h$.

Let $d = ((p-1)\ell)^h$ and $\epsilon = N \cdot 2^{-\ell}$. Then we have at least $p^{2^n(1-\epsilon)}$ possible f (p choices for each element of G) and only $p^{\sum_{i=0}^{d} \binom{n}{i}}$ possible degree-$(n+d)$ polynomials. Let $\ell = \frac{n^{1/(2h}}{p-1}$, so that $d = \sqrt{n}$ and $\epsilon = N \cdot 2^{n^{1/(2h)}/(p-1)} \leq n^{b - n^{1/(2h)}/(p-1)} = o(1)$ (assuming $N \leq n^b$ for some constant b). Then $|\{f\}| = p^{2^n(1-\epsilon)} = p^{2^n(1-o(1))}$ but $|\{f_p\}| = p^{c \cdot 2^n}$ for a constant $c < 1$ (since we are only going out one standard deviation).

At some point the $1 - o(1)$ gets bigger than the c, and we can't cover all functions f with degree $n/2 + \sqrt{n}$ polynomials any more. This implies that q' does not in fact work as well as advertised, meaning that our original circuit fails to compute parity for some input.

With some tinkering, essentially the same argument shows that a $\mathbf{AC}^0[p]$ circuit family can't compute MOD-q when $p \neq q$ and both are prime (or prime powers). We don't know much about what happens with circuits with MOD-m gates where m is not a prime power; polynomials over even $\mathbb{Z}_6$ turn out to be surprisingly powerful. citeBarringtonBR1994.

Chapter 11

Natural proofs

Last updated 2017. Some material may be out of date.

The circuit lower bounds in Chapter 10 generally don't relativize, since oracle gates don't do anything sensible when hit with random restrictions or reduced to polynomials. But they still are unlikely to generalize to produce stronger results, at least if we believe in cryptography. This follows from the **natural proofs** theorem of Razborov and Rudich [RR97].[1]

The idea is to rule out certain "natural" classes of proof strategies, which involve showing that there is a property Φ such that (a) any Boolean function that has this property is hard for some class, and (b) a particular Boolean function has this property. An example would be PARITY's property of not becoming the constant function under a large enough restriction, which is not true of functions in $\mathbf{AC}^0$. The argument is that if we have such a proof strategy that works and satisfies certain technical requirements, then we can use it to break cryptographic primitives. This means that showing that one thing we think is hard is hard, in this particular way, will involve showing that something else we think is hard is easy.

11.1 Natural properties

A **property** Φ is just a set of Boolean functions. We will use Φ_n for the restriction of Φ to functions on n inputs and use f_n to refer to the restriction of a function f to n inputs. A property is **constructive** if the problem

[1] The presentation in this chapter is very, very sketchy, and the presentation in [AB07, Chapter 22] is not much better. I recommend if you are interested in this to look at the original Razborov-Rudich paper [RR97].

$f_n \stackrel{?}{\in} \Phi_n$ is in **P**, where the input to the tester is a string of length 2^n giving the entire truth table for f_n and the time to test is thus polynomial in 2^n; **large** if at least a $2^-O(n)$ fraction of functions on n inputs are in Φ_n; and **useful against P/poly** if $f \in \Phi$ implies $f \notin$ **P/poly**.

If Φ is both constructive and large, we say that it is **natural**[2] The Razborov-Rudich theorem says a natural property useful against **P/poly** breaks certain pseudorandom function generators, which we now take a brief detour to define.

11.2 Pseudorandom function generators

For the purposes of the Razborov-Rudich Theorem, we will consider a class of pseudorandom function generators that are indistinguishable from a random function by a Turing machine running in time $2^{O(n)}$ with oracle access to the function. In this definition, a family of functions $\{f_n\}$ is pseudorandom if, for any machine M running in time $2^{O(n)}$, and n sufficiently large,

$$|\Pr[M(f_n(x) = 1)] - \Pr[M(R) = 1]| \leq 2^{-n^2},$$

where x is chosen uniformly from $\{0,1\}^n$ and R is a random function. A **pseudorandom function generator** is a machine $M(s,x)$ that given any polynomial-size **seed** s becomes a pseudorandom function f_n^s on x.

What is interesting about these generators is that is possible to show that they exist unless problems like FACTORING or DISCRETE LOG can be solved in $O(2^{n^\epsilon})$ time for any $\epsilon > 0$.

11.3 The Razborov-Rudich Theorem

Here we give a formal statement of the Razbov-Rudich theorem. This version is adapted form [AB07, Theorem 22.9]; the original version in the Razbov-Rudich paper [RR97] is stated more generally.

Theorem 11.3.1 (Razbov-Rudich [RR97])**.** *If there is a* **P***-natural property* Φ *that is useful against* **P/poly***, then there are no strong pseudorandom function generators as defined in the preceding section.*

Proof. We'll use Φ to distinguish a pseudorandom function generator from a random function R.

[2] More generally, we say that it is C**-natural** if it is testable in class C and large.

First observe that for any fixed seed s, the function f_n^s is computable in $\mathbf{P/poly}$. So pseudorandom functions do not have property Φ (because of usefulness). On the other hand, a random function does have property Φ with probability at least $1 - 2^{-O(n)}$, since Φ is large. So the probability that Φ distinguishes f_n^s from R is at least $2^{-O(n)} > 2^{-n^2}$. This is a problem because Φ is constructible, so we can use it to distinguish f_n^s from R. □

If we replace $\mathbf{P/poly}$ with some other class (like $\mathbf{AC}^0$), then we get a similar result for pseudorandom function generators that fool the other class instead of $\mathbf{P/poly}$.

11.4 Examples of natural proofs

The random restriction lower bound for PARITY given in §10.3.4.2 can be described by a property Φ where $\Phi(f) = 1$ if f cannot be reduced to a constant function via restricting $n - n^c$ inputs, where $c < 1$. This is large since almost all functions have this property, and constructive since we can check all restrictions from the truth table in time polynomial in 2^n. It is also useful against $\mathbf{AC}^0$ because $\mathbf{AC}^0$ functions don't have the property. But because Φ is large and $\mathbf{P}$-constructible, Theorem 11.3.1 implies that random restrictions won't be useful against $\mathbf{P/poly}$ unless there are no strong pseudorandom function generators.

An example that requires a little more pushing and shoving is Razborov-Smolensky (see §10.3.4.3. The main property used in that proof is that any function $f : \{-1, 1\} \to \mathbb{Z}_p$ can be written as $(\prod_{i=1}^{n})\ell_1 + \ell_2$ where ℓ_1 and ℓ_2 are degree-$(n/2)$ polynomials and the product is the encoding of parity when each bit is represented by an element of $\{+1, -1\}$ This property is unfortunately unique to parity, which would make it not large. So Razborov and Rudich suggest looking at the set of all functions $\{+1, -1\}^n \to \mathbb{Z} - P$ as a 2^n-dimensional vector space and instead using the property $\Phi(g)$ that the dimension of the set of functions $\{g \cdot \ell_1 + \ell_2 \mid \ell_1 \text{ and } \ell_2 \text{ have degree at most } n/2\}$ is at least $(3/4)2^n$. This property does not hold for functions approximated by low-degree polynomials (pretty much the same counting argument for the parity case), so it's useful against $\mathbf{AC}^0[p]$. At the same time it is large (shown in the paper) and constructible (build a $m \times n$ matrix whose $m \leq 2^{2n}$ rows are all functions in the set and check its dimension using Gaussian elimination in time polynomial in 2^n). So if strong pseudorandom function generators exist, it can't be useful against $\mathbf{P/poly}$.

This second example shows that even proofs that look unnatural might have a natural proof buried inside them. One of the things that Razborov and

Rudich do in their paper is go through a catalog of then-known lower bound results, mostly in circuit complexity, and demonstrate that each argument can be interpreted to give a natural property useful against some class. This means that it is not enough to structure a proof that $\mathbf{P} \neq \mathbf{NP}$ (for example) based on a property only applies to a non-large class of functions or is not obviously constructible: it must also be the case that the proof technique used to show this property is hard can't generalize to some other property that is both large and constructible. So far we have not had much luck finding such properties, and the Razborov-Rudich paper itself pretty much put an end to most work on circuit complexity lower bounds.[3]

[3]Curiously, the field of circuit complexity lower bounds can in some sense be traced back to a proof by Minsky and Papert [MP69] that certain classes of depth-2 circuits based on majority gates (**perceptrons**, a kind of **neural network**) couldn't compute PARITY. This put a serious damper on neural network research for a generation and was part of what led to the widespread belief in the "AI winter" era of the late 1980s that artificial intelligence as a field was basically doomed. Nowadays more sophisticated neural networks are the basis of the deep learning methods that have had significant success in solving previously-intractable pattern recognition problems.

And yet these fancier neural networks still can't compute PARITY. It's fortunate in retrospect that the problem of recognizing speech or driving a car doesn't seem to depend on calculating whether the number of one bits in the input is odd or even. Perhaps we can be hopeful that something similar will happen with circuit complexity.

Chapter 12

Randomized classes

Last updated 2017. Some material may be out of date.

In this chapter we look at classes corresponding to **randomized computation** , where we are not guaranteed to get the right answer for all inputs, but instead get the right answer with reasonably high probability assuming we have a source of random bits.

The basic idea of these classes is similar to the certificate or witness version of **NP**: a **randomized Turing machine** is a machine $M(x, r)$ that takes an input x and a string of independent fair random bits r. Typically we assume that M runs in polynomial time and that r has length $p(|x|)$ where p is some polynomial. Alternatively, we can assume that r is unbounded but provided on a one-way input tape. Either way, we get a collection of randomized complexity classes depending on the probability that $M(x, r)$ accepts given that x is or is not in a language L.

12.1 One-sided error: RP, coRP, and ZPP

The simplest randomized complexity class is **RP** (sometimes called **R**. This consists of all languages L such that there exists a polynomial-time randomized machine M that never accepts any x that is not in L, and accepts any x that is in L with probability at least $1/2$.

Such a machine is said to exhibit **one-sided error**. Languages in **RP** have the same asymmetry as languages in **NP**: we can never be tricked into accepting an x we shouldn't, but we might or might not accept an x we should. The difference is that the probability of accepting $x \in L$ is negligible if we feed a random witness string to an **NP** machine.

The complement of a **RP** language is in **coRP**: this means that L is in **coRP** if there is a machine that never rejects $x \in L$, but only rejects $x \notin L$ with probability 1/2 or greater.

12.1.1 P ⊆ RP ⊆ NP

It is trivially the case the **P** ⊆ **RP**, since we can ignore the random bits and just run the **P** computation; it also holds that **RP** ⊆ **NP**, since we can use any sequence of random bits that causes the **RP** machine to accept as a certificate for the **NP** machine. Similarly, we also have **P** ⊆ **coRP** ⊆ **coNP**.

12.1.2 Amplification of RP and coRP

The choice of 1/2 for the probability of success of a **RP** or **coRP** computation is arbitrary: in fact, it's enough to have a probability of accepting $x \in L$ that is polynomial in n. The reason (for **RP**) is that if we accept with probability at least $\epsilon > 0$, then the probability that we fail to accept at least once in k consecutive computations using independent strings $r_1, r_2, \ldots, r_k$ is at most $(1-\epsilon)^k < e^{-\epsilon k} \leq 1/e < 1/2$ for $k \geq 1/\epsilon$. In other words, we can **amplify** polynomially-small probabilities of success up to constant probabilities of success. If we like, we can continue the process to make failure exponentially improbable: if we have a machine that accepts with probability at least 1/2, the probability that it fails to accept in k independent runs is at most 2^{-k}. (For **coRP** machines, the same thing works, except now we are looking at the probability of failing to reject some $x \notin L$.)

12.1.3 Las Vegas algorithms and ZPP

The classes **RP**, **coRP**, and **BPP** all represent **Monte Carlo algorithms**. These are algorithms that produce the right answer with some reasonably high probability, but we can't tell when they produce the right answer.

The other main class of randomized algorithms are known as **Las Vegas algorithms**. In a Las Vegas algorithm, the machine returns the correct answer with probability 1, but there is no fixed bound on its running time. Instead, we ask that the randomized Turing machine accept or reject in polynomial time on average.[1] The class of languages that are decided by

[1] This requires some adjustment to the model, since the machine may consume an unbounded number of random bits. Typically we either assume that the machine executes probabilistic transitions (it flips its own coins) or that it is provided with an infinite sequence of random bits on a one-way input tape.

a randomized Turing machine in polynomial expected time is called **ZPP**, short for **zero-error probabilistic polynomial time**.

The class **ZPP** has an alternative definition that avoids unbounded executions:

Theorem 12.1.1. $\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}$.

Proof. First let us show that any language in **ZPP** is in both of **RP** and **coRP**. Let M be a machine that decides L in expected $T(n)$ time. Construct a new machine M^+ that runs M for at most $2T(n)$, returning the same value as M if M finishes within this bound and returning *accept* if it does not. Similarly let M^- act like M^+, except that it returns *reject* if M runs over time.

By Markov's inequality, the probability that M runs overtime is at most $1/2$. So if $x \notin L$, the probability that M^+ rejects x is at least $1/2$. If instead $x \in L$, the probability that M^+ rejects x is 0. So M^+ puts L in **coRP**. Similarly, M^- puts L in **RP**. This establishes $\mathbf{ZPP} \subseteq \mathbf{RP} \cap \mathbf{coRP}$.

In the other direction, suppose M^+ demonstrates L is in **coRP** and M^- demonstrates L is in **RP**. Given an input x, alternate between running M^+ and M^- on x until either M^+ rejects (which always means $x \notin L$) or M^- accepts (which always means $x \in L$). Accept or reject based on which happens. Since each iteration of this loop has at least a $1/2$ chance of producing an answer, we run 2 iterations on average, giving polynomial expected time since both M^+ and M^- are polynomial-time. This establishes $\mathbf{RP} \cap \mathbf{coRP} \subseteq \mathbf{ZPP}$. □

12.2 Two-sided error: BPP

For some problems, we may have an algorithm that produces both false positives (accepting strings not in L) and false negatives (rejecting strings in L). The class of languages that can be computed with constant two-sided error is called **BPP** (short for "bounded-probability probabilistic polynomial time"), and is defined as the set of all L for which there exists a randomized machine M that accepts with probability at most $1/3$ for inputs not in L and accepts with probability at least $2/3$ for inputs in L.

As with **RP** and **coRP**, the constants $1/3$ and $2/3$ are arbitrary, and can be replaced by $1/2 \pm \epsilon$ for any polynomial $\epsilon > 0$ without changing the class. Here the amplification process is a little more subtle: instead of taking an OR (**RP**) or AND (**coRP**) of the results of k independent computations, we have to take a majority. This can be shown to give the right answer with

high probability for sufficiently large but still polynomial k using Chernoff bounds.[2]

What does not work is to make $\epsilon = 0$. If we do this naively, we can have a machine that accepts exactly $1/2$ the time on all inputs, which tells us nothing. If we insist on some difference in the probability of accepting depending on whether $x \in L$, however small, this gives a different class, known as **PP**. For L to be in **PP**, there must exist a randomized Turing machine that accepts $x \in L$ with probability at least $1/2$ and accepts $x \notin L$ with probability less than $1/2$, leaving a gap that is generally too small to amplify. We'll come back to **PP** when we look at counting classes in Chapter 13.

We generally think of **BPP** as the class of functions that can be efficiently solved using randomization. We also tend to guess that **BPP** is the same as **P**, which would follow from the existence of sufficiently powerful pseudorandom number generators, but we can't prove this. The following sections give two results that show what we *can* prove about the strength of **BPP**.

12.2.1 Adleman's Theorem

This says that polynomial advice is enough to derandomize **BPP**, and is one of the reasons to suspect that **BPP** might in fact be equal to **P**, since there doesn't seem to be anything magical about **BPP** that would let it solve the obvious examples of problems in $\mathbf{P/poly} \setminus \mathbf{P}$.

Theorem 12.2.1 (Adleman [Adl78])**.** $\mathbf{BPP} \subseteq \mathbf{P/poly}$.

Proof. Given a randomized machine $M(x, r)$ that decides some language L in **BPP**, first assume that we've amplified M so that, for each x, $\Pr_r[M(x, r) \neq [x \in L]] < 2^{-|x|}$. Now suppose we try the same r on all x of a given length n: the expected number of such x for which $M(x, r)$ gives the wrong answers is strictly less than $2^n 2^{-n} = 1$, meaning that there is a nonzero probability that we pick an r that works for every x of length n. So such an r exists: make it the advice string α_n. □

[2] The version we want here says that if $X_1, \ldots, X_n$ are 0–1 random variables with $\mathrm{E}[X_i] \leq p_i$ for each i, and $\mu = \sum_{i=1}^{n} p_i$, then $\Pr\left[\sum_{i=1}^{n} X_i \geq (1+\delta)\mu\right] \leq e^{-\mu\delta^2/3}$ for any $0 < \delta < 1.81$. In particular, if $p_i = 1/2 - \epsilon$ is an upper bound on the probability of getting a false positive, then the chance that we get a majority of false positives in k trials is equal to $\Pr\left[\sum X_i \geq 1/2\right] = \Pr\left[\sum X_i \geq \frac{1/2}{1/2-\epsilon}\mu\right]$ which gives $\delta = \frac{1}{1-2\epsilon} - 1 < 2\epsilon$. So for $k = \Omega(n/\epsilon)$ (polynomial in n) we get a probability of getting a false majority value less than $e^{-\Omega(n^2)}$, which is pretty small.

12.3 The Sipser-Gács-Lautemann Theorem

This avoids using advice to derandomized **BPP**, but instead uses alternation. The intuition is that we can test if the set A of r that make $M(x,r)$ accept is large or small by testing if it is possible to cover all possible r by taking the union of a polynomial number of appropriately shifted copies of A.

Theorem 12.3.1 (Sipser-Gács-Lautemann)**.** $\mathbf{BPP} \subseteq \mathbf{\Sigma}_2^p \cap \mathbf{\Pi}_2^p$.

Proof. We'll show $\mathbf{BPP} \subseteq \mathbf{\Sigma}_2^p$; that $\mathbf{BPP} \subseteq \mathbf{\Pi}_2^p$ will then follow from the fact that **BPP** is closed under complement.

Suppose $L \in \mathbf{BPP}$, and that we have a machine $M(x,r)$ that decides if $x \in L$ with probability of error at most $2^-|x|$. For each x, let A_x be the set of random bit-vectors R such that $M(x,r) = 1$. Then if x is in L, $|A_x| \geq (1 - 2^{-|x|})2^{|r|}$, while if x is not in L, $|A_x| \leq 2^{-|x|}2^{|r|}$. We will distinguish these two cases by testing if we can cover all r with $\bigcup_{i=1}^{k}(A_x \oplus t_i)$ for some $t_1, \ldots, t_k$, where $A_x \oplus t_i = \{r \oplus t_i \mid r \in A_x\}$.

First let's suppose $x \in L$. Fix some r, and choose $t_1, \ldots, t_k$ independently at random. Then $\Pr[r \notin A_x \oplus t_i] = \Pr[r \oplus t_i \notin A_x] = |A_x|2^{-|r|} \leq 2^{-|x|}$. Since the t_i are chosen independently, $\Pr[r \notin \bigcup(A_x \oplus t_i)] \leq \left(2^{-|x|}\right)^k = 2^{-k|x|}$. The expected number of r that are not in $\bigcup(A_x \oplus t_i)$ is then bounded by $2^{|r|}2^{-k|x|} = 2^{|r|-k|x|}$. If this is less than 1 for some polynomial k (and it is), then there exists some sequence of bit-vectors $t_1, \ldots, t_k$ that leave no r uncovered, meaning that the $\mathbf{\Sigma}_2^p$ formula $\exists t_1, \ldots, t_k \forall r \bigwedge_{i=1}^{k} M(x, r \oplus t_i) = 1$ is true.

On the other hand, if $x \notin L$, then $|A_x| \leq 2^{|r|-|x|}$ and, for any $t_1, \ldots, t_k$, $|\bigcup_{i=1}^{k}(A_x \oplus t_i)| \leq \sum_{i=1}^{k}|A_x \oplus t_i| \leq k2^{|r|-|x|} = 2^{|r|} \cdot k2^{-|k|}$. For polynomial k, this is strictly less than $|r|$, meaning that no matter what $t_1, \ldots, t_k$ we pick, we can't cover all possible r. In other worse, the formula $\exists t_1, \ldots, t_k \forall r \bigwedge_{i=1}^{k} M(x, r \oplus t_i) = 1$ is now false.

Since we have a $\mathbf{\Sigma}_2^p$ formula that decides L for sufficiently large n, this puts L in $\mathbf{\Sigma}_2^p$, and thus $\mathbf{BPP} \subseteq \mathbf{\Sigma}_2^p$. □

Chapter 13

Counting classes

Last updated 2017. Some material may be out of date.

A *function* is in **FP** ("functional **P**") if and only if there is a polynomial-time Turing machine that, given x on its input tape, writes $f(x)$ to its output tape and halts. The class **FP** is in a sense a functional analog of **P**.

The class of functions #**P** ("sharp **P**") is the functional version of **NP**: A function f is in #**P** if and only if there is a polynomial p and polynomial-time Turing machine $M(x, r)$ such that, for each x, $f(x) = |\{r \mid |r| = p(|x|), M(x, r) = 1)\}|$. In other words, f counts the number of accepting branches of some polynomial-time nondeterministic Turing machine.

13.1 Search problems and counting problems

Formally, the class**NP** consists of decision problems. However, most of these decision problems are the decision version of **search problems** , where we have some relation $R(x, y)$, and given x, we want to find y with length $p(|x|)$, where p is some fixed polynomial, that makes $R(x, y)$ true; or determine that there is no such y. The canonical example is SAT, which in search problem form is a relation $\text{SAT}(\phi, x)$ where ϕ is a CNF formula and x is an assignment that makes ϕ true. But we can similarly define problems like GRAPH 3-COLORABILITY, VERTEX COVER, or HAMILTONIAN CIRCUIT as search problems: in each case, we are given a graph, and want to find some structure within the graph that satisfies a particular predicate.

For any search problem $R(x, y)$, there is a corresponding decision problem given by $\{x \mid \exists y R(x, y)\}$. Similarly, every search problem gives rise to a counting problem, to compute the function $\#_R(x) = |\{y \mid R(x, y)\}|$. So from

SAT we get #SAT (how many satisfying assignment does this formula have?), from GRAPH 3-COLORABILITY we get #GRAPH 3-COLORABILITY (how many 3-colorings does this graph have?), and so on. We can think of #**P** as the class of counting problems corresponding to the search problems in **NP**.

Because every language L in **NP** has a polynomial-time verifier $M(x, y)$ that checks if $x \in L$ based on the existence of a corresponding y, we can in principle turn every decision problem in **NP** into a search problem. This may not always give a search problem that is very interesting, and we usually have to exercise some judgment about which verifier we are willing to apply this transformation to. A SAT verifier that takes as input a CNF formula and a variable assignment gives the obvious search version of SAT; but a SAT verifier that takes as input a CNF formula and a graph coloring for a graph that the CNF formula reduces to defines a different search problem, one that we probably would not think of as the search version of SAT. Hypothetically, we could even imagine problems in **NP** for which there is no reasonable description as a search problem; an example would be every problem in **P**, for which the certificate is useless.

These same considerations apply to counting problems. In defining #SAT, we have to be careful to make sure that we are looking at a reasonable search version of SAT to begin with. And as with search problems, we can imagine decision problems in **NP** that have no reasonable corresponding counting problems.

13.1.1 Reductions

For decision problems, our primary notion of reduction is the **many-one reduction** or **Karp reduction**, where $A \leq_{\mathbf{P}} B$ if there is a polynomial-time computable f such that $x \in A$ if and only if $f(x) \in B$.[1]

For search problems, the corresponding class of reductions are known as **Levin reductions**. If R and S are search problems, a Levin reduction from R to S is a pair of polynomial-time computable functions f and g such that $\langle x, g(x, y)\rangle \in R$ if and only if $\langle f(x), y\rangle \in S$. The idea here is that f translates the problem from R to S, and g translates the solution back from S to R (which is why g needs to know the problem x). An example of a Levin reduction is provided by the Cook-Levin theorem: given a

[1] There is also the stronger notion of a **Cook reduction**, where $x \in A$ is decided by a polynomial-time Turing machine with oracle access to B. Cook reductions have a role in complexity theory, but they most often come up in introductory complexity theory courses when somebody tries to pass off a Cook reduction as a Karp reduction without realizing it.

polynomial-time verifier for R, this constructs a translation from inputs x to 3SAT formulas $f(x)$, and from each satisfying assignment we can extract the relevant variables corresponding to the witness to get a solution $g(x, y)$ to the original problem. So SAT is not only **NP**-complete for decision problems in **NP** (with respect to Karp reductions), but it is also **NP**-complete for search problems in **NP** (with respect to Levin reductions). Many other standard Karp reductions turn out to also be Levin reductions with a tiny bit of additional work.

For counting problems, the corresponding class of reductions are known as **parsimonious reductions**. A parsimonious reduction from a counting problem R to another counting problem S again consists of a pair of polynomial-time computing functions f and g, where $\#_R(x) = g(x, \#_S(f(x)))$. The simplest parsimonious reductions are those for which each solution to $R(x, -)$ maps to exactly one solution to $S(f(x), -)$ (**one-to-one reductions**). But in general we are happy as long as we can recover $\#_R(x)$ from $\#_S(f(x))$ efficiently. The general definition allows us, for example, to parsimoniously reduce #SAT to #UNSAT: if we can count the number of assignments that *don't* satisfy a formula ϕ with n variables, we can compute the number that *do* satisfy ϕ by subtracting from 2^n. This example also shows that parsimonious reductions don't need to correspond to Levin reductions, since we generally don't expect to be able to recover a satisfying assignment from a non-satisfying assignment, and if $\mathbf{NP} \neq \mathbf{coNP}$ we won't be able to find a Levin reduction from SAT to UNSAT.

13.1.2 Self-reducibility

SAT has the desirable property of being **self-reducible**: If we can test for membership in SAT, we can reduce the problem of computing a satisfying assignment to some satisfiable formula ϕ to the smaller problem of computing a satisfying assignment to whichever of $\phi[x = 0]$ or $\phi[x = 1]$ is satisfiable, where x is some variable in ϕ. This means that the search problem version of SAT is solvable by a polynomial-time Turing machine with access to a (decision) SAT oracle: $\mathrm{SAT}_{\mathrm{search}} \in \mathbf{FP}^{\mathrm{SAT}_{\mathrm{decision}}}$ This fact extends to any other problem R that is complete for **NP** (with respect to Levin reductions), because we can use $SAT \leq_P R$ to transform an oracle for the decision version of R to an oracle for the decision version of SAT, and use the Cook-Levin Theorem to transform R into SAT. However, for problems R that are not **NP**-complete, we cannot guarantee that they are self-reducible; it may be that the decision version of R is easier than the search version.

Because the counting version of a problem allows us to solve the decision

version, we can also use the counting version of **NP**-complete problems to solve their search versions. Indeed, with some tinkering, we can even unrank solutions using binary search (for example, find the 937-th satisfying assignment to ϕ). I don't actually know of any good applications for this, but it seems like it ought to be useful for something.

13.2 FP vs #P

It is an open question whether $\#\mathbf{P} \subseteq \mathbf{FP}$. The converse holds, for functions $\{0,1\}^* \to \mathbb{N}$ that do not get too big. The idea is that if $f : \{0,1\}^* \to \mathbb{N}$ is in **FP**, then the machine $M(x,r)$ that accepts if and only if $r < f(x)$ shows that f is in $\#\mathbf{P}$ as well.

An example of a problem that is likely to be in $\#\mathbf{P} \setminus \mathbf{FP}$ is #SAT, which outputs for any 3CNF formula ϕ the number of assignments that satisfy ϕ. It's easy to see that $\#\text{SAT} \in \#\mathbf{P}$, since we can build a machine $M(\phi, r)$ that tests if r is a satisfying assignment for ϕ. The reason we don't think it's in **FP** is because

Theorem 13.2.1. *If* $\#\text{SAT} \in \mathbf{FP}$*, then* $\mathbf{P} = \mathbf{NP}$*.*

Proof. Suppose $\#\text{SAT} \in \mathbf{FP}$. Given a formula ϕ, compute, in polynomial time, the number of satisfying assignments to ϕ. Accept if this number is not zero. We have just solved SAT in **P**. □

More generally, we can show that #SAT is $\#\mathbf{P}$-complete in the sense that any problem in $\#\mathbf{P}$ can be parsimoniously reduced to #SAT. This is an immediate consequence of the Cook-Levin Theorem.

Not all $\#\mathbf{P}$-complete problems arise as the counting versions of **NP**-complete search problems. For example, the #CYCLE problem asks how many simple cycles can be found in a directed graph. This turns out to be $\#\mathbf{P}$-complete, because we can take a directed graph G for which we want to compute the number of Hamiltonian cycles (#HAMILTONIAN-CYCLE), and reduce this problem to #CYCLE by replacing each edge in G with a widget that allows 2^m possible paths between its endpoints. Each n-node cycle in G will give 2^{mn} cycles in this new graph G', while smaller cycles (of which there are at most n^{n-1} will give at most $2^{m(n-1)}$. By choosing m so that $2^{mn} > n^{n-1}2^{m(n-1)}$ ($m > n \lg n$ works), we can divide the output of #CYCLE by 2^{mn}, discarding the remainder, and get the number of Hamiltonian cycles.[2] But even though counting cycles is hard, finding a cycle can be done in linear time.

[2]For #CYCLE to be $\#\mathbf{P}$-complete, we also need to know that #HAMILTONIAN-

13.3 Arithmetic in #P

If f and g are functions in $\#\mathbf{P}$, so are $f+g$ and $f \cdot g$. This is easiest to see if we think of f and g as $\#\mathrm{SAT}(\phi)$ and $\#\mathrm{SAT}(\gamma)$ for appropriate formulas ϕ and γ. Then

1. $f \cdot g = \#\mathrm{SAT}(\phi \wedge \gamma)$ where ϕ and γ have distinct variables. The argument is that if S is the set of satisfying assignments for ϕ, and T the set of satisfying assignments for γ, then $S \times T$ is the set of satisfying assignments for $\phi \wedge \gamma$.

2. $f + g = \#\mathrm{SAT}((x \wedge \phi) \vee (\neg x \wedge \gamma))$, where x is a variable that does not appear in ϕ or γ. Here the OR gives us a union of sets of satisfying assignments, and x guarantees that the union is disjoint.

We can also implement constants. For example, $1 \in \#\mathbf{P}$ because it is $\#\mathrm{SAT}(x)$, $2^k \in \#\mathbf{P}$ because it is $\#\mathrm{SAT}\left(\bigwedge_{i=1}^{k}(x_i \vee \neg x_i)\right)$, and in general we can implement an arbitrary constant ℓ using a formula of size $O(\log^2 \ell)$ by applying the $f+g$ rule to the powers of two that appear in the binary expansion of ℓ. Similar tinkering lets us implement polynomials over $\#\mathbf{P}$ functions as well.

13.4 Counting classes for decision problems

In a sense, the randomized classes of Chapter 12 are already decision versions of counting problems, since **RP** (for example) asks if some machine $M(x,y)$ either never says yes to x or says yes for at least half the possible y. But this doesn't work for an arbitrary M: we need M to promise us that one or the other of these two outcomes occurs (this is an example of a **promise problem**, which we can think of as a search problems with extra constraints on M). For proper counting classes, we'd like to have definitions that work for any polynomial-time verifier $M(x,y)$.

13.4.1 PP

The class **PP** (**probabilistic P**) is the simplest of these. It consists of all languages L of the form $\left\{x \;\middle|\; \#_R(x) \geq \frac{1}{2} \cdot 2^{|x|}\right\}$, where $R(x,y)$ is a polynomial-time computable predicate. In effect, **PP** transforms counting problems in

CYCLE is $\#\mathbf{P}$-complete. This is not entirely obvious, but a parsimonious reduction from #SAT is given by Liśkiewicz, Ogihara, and Toda [LOT03].

$\#\mathbf{P}$ into decision problems by returning only the first bit of the output. We can also think of **PP** as what happens when we start with **BPP** and reduce the gap between negative and positive instances to be as small as possible. Note that unlike **BPP**, **PP** is not entirely symmetric, since we had to make a decision about what to do with an input with exactly $\frac{1}{2}2^n$ witness.[3]

13.4.2 $\oplus\mathbf{P}$

If instead we return the last bit of the output, we get the class $\oplus\mathbf{P}$ (**parity P**), the set of languages L of the form $\{x \mid \#_R(x) \bmod 2 = 1\}$, where as usual $R(x, y)$ is a polynomial-time predicate. The nice thing about $\oplus\mathbf{P}$ is that it retains the arithmetic character of $\#\mathbf{P}$: we can represent a function f in $\oplus\mathbf{P}$ as $\oplus\mathrm{SAT}(\phi)$ for some formula ϕ, where $\oplus\mathrm{SAT}(\phi) = \#\mathrm{SAT}(\phi) \bmod 2$ is the standard $\oplus\mathbf{P}$-complete language, and the same operations on formulas that let us represent $f \cdot g$, $f + g$, etc. using #SAT work for $\oplus$SAT as well.

13.4.3 UP and the Valiant-Vazirani Theorem

The class **UP** (**unique P**) consists of all languages L of the form $\{x \mid \#_R(x) = 1\}$. The idea is to restrict **NP** so we are not confused by extraneous solutions: we only consider some x to be in L if there is a *unique* y such that $R(x, y)$ is true. However, if we are willing to do randomized reductions, we can reduce any problem in **NP** to a problem in **UP**, by randomly restricting the set of solutions until we get 1. This is the Valiant-Vazirani Theorem:

Theorem 13.4.1 (Valiant-Vazirani [VV86])**.** $\mathbf{NP} \subseteq \mathbf{RP}^{\mathbf{UP}}$.

Proof. The idea is to reduce SAT to its unique version USAT by adding some random clauses that knock out the extra solutions. This doesn't always work, but it works with high enough probability that if we try it enough times we will get lucky.

The trick is to use a pairwise-independent random hash function from the set of assignments $\{0,1\}^n$ to $\{0,1\}^k$ for some k, and throw away any solutions that don't hash to zero. If we think of each assignment as a column vector x over $\mathbb{Z}_2$, then we can compute a hash using the matrix formula $Ax + b$, where A is a random $k \times n$ matrix and b is a random k-dimensional column vector. We keep any solutions x for which $Ax + b = 0$, or equivalently for which $Ax = b$.

[3] But in fact it is not too hard to show that we get the same class if we made the other decision; this follows from the arithmetic tricks in §13.3.

Expanding $(Ax)_i = \sum_{j=1}^{n} A_{ij}x_j = \bigoplus_{j=1}^{n}(A_{ij} \wedge x_j)$ gives a Boolean formula for $(Ax)_i = b_i$ that (with a few extra variables and some trickery) we can express in CNF form with a formula of length $O(n)$. Appending one of these for each i gives a restriction ρ with $O(kn)$ length; if ϕ is our original SAT formula, we take $\rho \wedge \phi$ as our candidate USAT formula.

Because b is chosen uniformly at random, any particular x has exactly a 2^{-k} chance of satisfying ρ. We also have pairwise independence: if $x \neq y$, then $\Pr[Ay = b \mid Ax = b] = \Pr[A(x + y) = 0] = 2^{-k}$, since this is just the probability that the sum of the (random) columns corresponding to the bits where x and y differ is exactly 0.

If ϕ has s solutions, then the chance that any particular solution x is the unique solution to $\rho \wedge \phi$ is the probability that (a) x satisfies ρ; and (b) no solution $y \neq x$ satisfies ρ. The probability that x satisfies ρ is exactly 2^{-k}; the probability that at one or more $y \neq x$ satisfies ρ is at most $(s-1)2^{-k}$. So this gives a probability that x is the unique solution of at least $2^{-k}(1 - (s-1)2^{-k})$. If we let A_x be the event that x satisifes ρ, then these events are disjoint for distinct x, so the union of these events has probability at least $s2^{-k}(1 - (s-1)2^{-k}) > s2^{-k}(1 - s2^{-k})$.

Now suppose $2^{k-2} \leq s \leq 2^{k-1}$. Then the probability that we get a unique survivig solution is at least $2^{k-2}2^{-k}(1 - 2^{k-1}2^{-k}) = 2^{-2}(1 - 2^{-1}) = \frac{1}{8}$. So if we are very lucky in our choice of k, we get a $\frac{1}{8}$ chance of a unique solution. But for any particular s, because $2^0 \leq s \leq 2^n$, there is some $k \in \{2, \ldots, n+1\}$ that works. So we can pick one of these n possible values uniformly at random, and get at least a $\frac{1}{8n}$ chance that ρ restricts to a unique solution.

We now ask our **UP** oracle about $\rho \wedge \phi$. If it says that it has a unique solution, we can accept, secure in the knowledge that ϕ must have at least one solution to begin with. If it says no, we reject. This gives a $\frac{1}{8n}$ chance of accepting ϕ if it has a solution, and no chance of accepting ϕ if it doesn't, putting SAT in $\mathbf{RP}^{\mathbf{UP}}$ once we do a little amplification to get the acceptance probability up. □

13.5 Toda's Theorem

Here we want to show that $\mathbf{P}^{\mathbf{PP}}$ can compute any language in the polynomial-time hierarchy **PH**, a result due to Toda [Tod91]. An immediate consequence of this is that if $\mathbf{PP} \subseteq \mathbf{PH}$, then **PH** collapses.[4]

[4]Toda's paper is worth reading, but it's pretty long, and depends on introducing some notation that, while useful, can be confusing. The presentation I will give here is based

13.5.1 Reducing from Σ_k to $\mathbf{BPP}^{\oplus\mathbf{P}}$

The first step is to show that $\mathbf{PH} \subseteq \mathbf{BPP}^{\oplus\mathbf{P}}$. This is done by extending the Valiant-Vazirani proof from §13.4.3. Specifically, we are going to argue that if ϕ is a $\mathbf{\Sigma}_k$ formula, then there is a randomized reduction from ϕ to a formula ψ with no quantifiers, such that if ϕ is satisfiable, then $\oplus\text{SAT}(\psi$ is true with probability at least $2/3$, and if ϕ is not satisfiable, then $\oplus\text{SAT}(\psi)$ is false always.

It is helpful to define a quantifier for keeping track of parity. Let $\oplus x : \phi(x)$ be true if and only if $|\{x \mid \phi(x)\}| \bmod 2 = 1$. Then $\oplus\text{SAT}(\phi(x))$ is true if and only if $\oplus x : \phi(x)$ is true. More generally, if ϕ is a $\mathbf{\Sigma}_k$ formula with a free variable x, we can define $\bigoplus \Sigma_k\text{SAT}(\phi(x))$ to be true if $\oplus x : \phi(x)$ is true. Our strategy for randomly reducing $\Sigma_k\text{SAT}$ to $\oplus\text{SAT}$ will be to first treat a $\Sigma_k\text{SAT}$ problem as a $\bigoplus \Sigma_k\text{SAT}$ problem (we can just add a dummy $\oplus x$ at the front), and show how to peel off the non-parity quantifiers one at a time.

Let's start with $\Sigma_1\text{SAT}$. Here we have a formula of the form $\oplus x \exists y \phi(x, y)$, and we want to get rid of the $\exists y$. Looking at just the $\exists y \phi(x, y)$ part, we know from the proof of Theorem 13.4.1 that there is a randomized restriction ρ such that $\oplus y(\rho(y) \wedge \phi(x, y))$ is true with probability at least $\frac{1}{8n}$ if $\exists y \phi(x, y)$ is true, and with probability 0 otherwise. To amplify this probability, we want to compute $\bigvee_{i=1}^{\ell} \oplus y(\rho_i(y) \wedge \phi(x, y))$ for some large ℓ, where the ρ_i are chosen independently at random. But we can do this using the formula $\phi'(x, y) = \oplus y\Big(1 + \prod_{i=1}^{\ell}(1 + (\rho_i(y) \wedge \phi(x, y)))\Big)$, where addition and subtraction of formulas are defined as in §13.3. This formula will have polynomial size (possibly after applying Cook-Levin to rewrite the unquantified part in CNF, if we insist on this) so long as ℓ is polynomial.

The probability of error for $\phi'(x, y)$ is bounded by $\left(1 - \frac{1}{8n}\right)^{\ell} \leq e^{-\ell/8n}$. We can make this exponentially small by setting $\ell = \Theta(n^2)$. This allows us to argue that the total error over the at most 2^n possible x assignments is still constant, which gives

$$\begin{aligned} \Pr\left[\oplus x \oplus y \phi'(x, y) = 1\right] &\geq 2/3 \qquad \text{when } \oplus x \exists y \phi(x, y) = 1, \text{ and} \\ \Pr\left[\oplus x \oplus y \phi'(x, y) = 1\right] &= 0 \qquad \text{when } \oplus x \exists y \phi(x, y) = 0. \end{aligned}$$

Since we can combine $\oplus x \oplus y$ into a single quantifier $\oplus(x, y)$, this gives a randomized reduction from $\bigoplus \Sigma_1\text{SAT}$ to $\bigoplus \Sigma_0\text{SAT} = \oplus\text{SAT}$. This shows that $\bigoplus \Sigma_1\text{SAT}$ can be decided by a $\mathbf{RP}$ or $\mathbf{BPP}$ machine that is allowed one call to a $\oplus\mathbf{P}$ oracle.

---

in part on some lecture notes of Andrej Bogdanov that I found at http://www.cse.cuhk.edu.hk/~andrejb/courses/s07-198538/lec8.pdf while looking for a simpler version.

For larger k, we can do the same thing to the outermost $\exists$ to reduce $\bigoplus \Sigma_k\text{SAT}$ to $\bigoplus \Pi_{k-1}\text{SAT}$, and then complement the resulting formula (by throwing in a $+1$) to get to $\bigoplus \Sigma_{k-1}\text{SAT}$. By iterating this process (after adjusting ℓ to keep the error at each stage down to something $\ll 1/k$), we get a randomized reduction from $\bigoplus \Sigma_k\text{SAT}$ to $\oplus\text{SAT}$.

13.5.2 Reducing from $\mathbf{BPP}^{\oplus\mathbf{P}}$ to $\mathbf{P}^{\#\mathbf{P}}$

The second step is to use the extra power in $\#\mathbf{P}$ to get rid of the need for randomness.

We will use the following number-theoretic fact:

Lemma 13.5.1. *Given a formula ϕ, let $\psi = \phi^6 + 2\phi^3$. Then $\#\text{SAT}(\phi) = 0 \pmod N$ implies $\#\text{SAT}(\psi) = 0 \pmod{N^2}$ and $\#\text{SAT}(\phi) = -1 \pmod N$ implies $\#\text{SAT}\phi = -1 \pmod{N^2}$.*[5]

Proof. For the first part, factor $\phi^6 + 2\phi^3 = \phi^2(\phi^4 + 2\phi)$; so if ϕ is a multiple of N, ϕ^2 and thus ψ is a multiple of N^2.

For the second part, expand ϕ^6 as $(\phi^3+1)^2-1$. Then if $\phi = -1 \pmod N$, $\phi^3 + 1 = 0 \pmod N$, so $(\phi^3 + 1)^2 = 0 \pmod{N^2}$, and subtracting 1 gets us to $-1 \mod N^2$. □

Now run this $O(\log n)$ times to get a formula $\psi(x)$ such that $\#\text{SAT}\psi(x)$ is always 0 or $-1 \bmod 2^n$, where $n = |x|$, meaning that it always returns either 0 or $2^n - 1$. We now recall that $\psi(x)$ is really $\psi(x, r)$, where r is the sequence of random bits used to construct our original ϕ. Each expansion multiplies the size of the original formula by a constant (we need to AND together six copies to do ϕ^6 and three for ϕ^3, and do some disjoint ORs for the addition), so all $O(\log n)$ stages increase the size by a factor of $O(1)^{O(\log n)}$, which is polynomial.

Using a single call to #SAT, we can compute $\#(x, r) : \psi(x, r) = \sum_r \#x : \psi(x, r)$. Divide this by $(2^n - 1)2^{|r|}$ to get the probability that $\oplus\text{SAT}(\phi(x, r))$ accepts, which decides our original $\Sigma_k\text{SAT}$ problem. This puts $\Sigma_k\text{SAT}$ in $\mathbf{P}^{\#\mathbf{P}}$, which because it works for each k, shows $\mathbf{PH} \subseteq \mathbf{P}^{\#\mathbf{P}} = \mathbf{P}^{\mathbf{PP}}$.

[5] Other polynomials work as well.

Chapter 14

Descriptive complexity

Last updated 2017. Some material may be out of date.

Descriptive complexity is an approach to defining completely classes in terms of classes of logical formulas. From a programmer's perspective, this is a bit like trying to define a programming language that can only be used to write programs that solve problems in **NL**, **P**, **NP**, etc. The idea is to avoid talking specifically about resources like time and space and instead restrict the kinds of formulas we can write so that the resource constraints occur as a consequence of these restrictions. By fixing a particular logic, we can give a characterization of precisely those predicates that are decidable in many standard complexity classes.

The hope is that with such classifications, we can (a) easily determine that particular problems can be solved within a given class, and (b) maybe gain some understanding about why certain classes might or might not be equal to each other. So far descriptive complexity has not had any more luck separating classes than the usual resource-based approach, but it gives an alternative perspective that might be useful for something. (But as with any unfinished branch of mathematics, it is hard to tell when using a different formalism is genuinely going to lead us out of the wilderness or is just a way of complicating our ignorance.)

Our main goal in this chapter is to prove **Fagin's Theorem**, which characterizes the languages in **NP** as precisely those expressible in a particular logic known as **existential second-order existential logic** or **ESO** (sometimes $\exists$**SO** for people who like typesetting, or $SO\exists$ if you think the second-order part should come first).

To explain ESO, we need to start by explaining **second-order logic**, and it will help to start by reviewing **first-order logic**.

14.1 First-order logic

First-order logic is what you have probably seen before. A first-order sentence consists of a quantified Boolean formula over some built-in predicates plus equality, where the quantifiers take on values in the universe under consideration. For example, we can express things like "for every number, there is a bigger number" when the universe is $\mathbb{N}$ and we have the predicate $<$ by writing $\forall x : \exists y : y > x$.

Some languages that we normally think of as computational can be expressed in first-order logic, where we take the universe as consisting of the elements of some structure that is the input to our program. For example, we can describe simple properties of bit-strings using the predicates $x < y$, indicating that x is a smaller index than y in the string, and the function $P(x)$, giving the value of the bit at position x. An example would be the first-order formula

$$\forall x : \exists y : (P(x) = 0) \Rightarrow (P(y) = 1),$$

which says that if there is a 0 in the string, it is followed by at least on 1 later in the string.

Computationally, everything we can write in first-order logic without any extra features fits comfortably inside $\mathbf{AC}^0$: whenever we have a $\forall$, we replace it with an $\wedge$ with fan-in n, and similarly whenever we have a $\exists$, we replace it with a $\vee$. This gives us a tree of constant depth (since the formula is fixed and doesn't depend on the size of the input) and n-bounded fan-in, giving polynomial size. The $<$ or P predicates just turn into constants or inputs.

We can also express properties of ordered graphs (where the order is just some arbitrary ordering of the vertices). For example, if we have a directed graph represented as a collection of vertices ordered by $<$ and a predicate $s \to t$ if there is an edge from s to t, then we can write

$$\forall x : \forall y : \forall z : (x \to y \wedge x \to z) \Rightarrow (y = z)$$

This formula recognizes graphs where every node has out-degree at most one, also known as directed forests. Or we could write

$$\forall x : \forall y : (x \to y) \Rightarrow (x < y)$$

to recognize a directed acyclic graph that has been topologically sorted. These are both properties that can in a sense be verified locally, which is about all that first-order logic can do, since we only have a constant number of variables to work with. Other examples of locally-verifiable graph

properties are being a tree (exactly one vertex has out-degree 0, the rest have out-degree 1) or a path (every vertex has both in-degree and out-degree 1, except for a source with in-degree 0 and out-degree 1 and a sink with in-degree 1 and out-degree 0). But more complicated properties are harder for first-order logic.

14.2 Second-order logic

What if we want to recognize a directed acyclic graph that has not already been sorted for us? It is tempting to say that there exists a partial ordering R of the nodes such that the endpoints of each edge are ordered consistently with R, but we can't say $\exists R$ in first-order logic: the universe consists only of vertices, and doesn't include hypothetical orderings. To say that such an order exists, we need second-order logic, which allows both **first-order quantifiers** over objects in our universe and **second-order quantifiers** over relations (each of a given **arity**, or number of argument). In second-order logic, we can write a formula for a directed acyclic graph as

$$\exists R : (\forall x \forall y \forall z (\neg xRx) \wedge (xRy \Rightarrow \neg yRx) \wedge (xRy \wedge yRz \Rightarrow xRz)) \wedge ((x \to y) \Rightarrow xRy).$$

Here most of the formula expresses that R is in fact irreflexive, antisymmetric, and transitive—that is, strict partial order—while the last part says that edges only go in increasing direction of R.[1]

This formula is in fact an example of a formula in *existential* second-order logic, because we only use existential second-order quantifiers. By just looking at the structure of the formula, and applying Fagin's Theorem, we will see immediately that recognizing a directed acyclic graph is in **NP**. This is true for many graph languages in **NP** (some of which, unlike DAG, are even **NP**-complete). An example would be this ESO formula for HAMILTONIAN PATH:

$$\exists R : \forall x \forall y \forall z (xRy \vee yRx) \wedge ((xRy \wedge x \neq y) \Rightarrow \neg yRx) \wedge (xRy \wedge yRz \Rightarrow xRz) \wedge ((xRy \wedge (\neg \exists q : xRq \wedge qRy)) \Rightarrow$$

This formula says that there is a total order R on all vertices such that whenever x is the immediate predecessor of y in R, there is an edge from x to y. In other worse, there is a path that includes all vertices. So Fagin's Theorem will tell us that HAMILTONIAN PATH is also in **NP**.

Typically, whenever we would want to store something on our Turing machine, we will instead represent it by some predicate whose existence

[1] It's tempting to leave R out of this and just declare that $\to$ is a partial order, but for a general DAG it isn't, because most DAGs aren't transitive.

we assert with a second-order existential quantifier and whose properties we enforce using the first-order formula. For example, here's GRAPH 3-COLORABILITY:

$$\exists R \exists G \exists B \forall x \forall y (Rx \vee Gx \vee Bx) \wedge ((x \to y) \Rightarrow (\neg(Rx \wedge Ry) \wedge \neg(Gx \wedge Gy) \wedge \neg(Bx \wedge By))).$$

This is a little sloppy, since it allows a vertex to have more than one color, but if there exist predicates R, G, and B telling us which vertices are red, green, and blue, we can always prune any multi-coloring we get back to a legal coloring. (If we really cared about avoiding this we could add a few extra clauses to enforce exactly one color per vertex.)

14.3 Counting with first-order logic

Many results in descriptive complexity require the ability to do some sort of counting; a typical application is building indices into some predicate provided to us by a second-order quantifier, where we are interpreting the indices numerically rather than as specific combinations of elements. The trick is to represent numbers as collections of elements, and then show that we can (a) do basic arithmetic on these numbers (mostly limited to adding or subtracting 1); and (b) represent values that are polynomial in the size of the structure n.

Given an ordered structure of size n, we can easily express numbers up to $n-1$ by representing the number i by whichever element x has i smaller elements. This gives us the ability, without any extra magic, to represent predicates like:

$$\begin{aligned} x = 0 &\equiv \forall y \neg(y < x) \\ x = n-1 &\equiv \forall y \neg(x < y) \\ y = x+1 &\equiv (x < y) \wedge (\forall z \neg(x < z \wedge z < uy)) \\ y = x-1 &\equiv x = y+1 \end{aligned}$$

This gives us the ability to use values as indices into $\{0, \ldots, n-1\}$ for which we can express the successors and predecessors. (More sophisticated arithmetic operations will require more power in our logic.)

If we want larger values, we can encode them as vectors of elements. With a vector of length k, we can represent elements in the range $0 \ldots n^k - 1$. The method is to define a comparison predicate $<$ on vectors (reusing notation here) by the rule

$$\langle x_1, \ldots, x_k \rangle < \langle y_1, \ldots, y_k \rangle \equiv \bigvee i = 1^k \Big(\Big(\bigwedge j = 1 i - 1 x_j < y_j \Big) \wedge x_i = y_i \Big).$$

This definition orders tuples lexicographically, and the order is total. So we can use the same construction as for single elements to represent a number i as whatever sequence of k elements has i smaller sequences, and define $\langle x_1, \ldots, x_k \rangle = 0$, etc., as above.

Where this is useful is that if we feed one of these k-tuple numbers to a k-ary relation R, we can treat R as a bit-vector of length n^k. When k is large, this does require the **arity** or number of arguments to R is equally large, but we are fine as long as neither depends on n.

14.4 Fagin's Theorem: ESO = NP

We'll prove Fagin's Theorem for strings, since that simplifies translating between TM inputs and ordered structures. It's not hard to generalize this to any ordered structure (like ordered graphs), but we will omit the details.

Theorem 14.4.1. *Let L be a set of ordered strings. Then L is in* **NP** *if and only if membership in L is expressible by an existential second-order formula.*

Proof. The **ESO** $\subseteq$ **NP** direction is easy: given a formula $\exists R_1 \exists R_2 \ldots \exists R_k \phi$, where ϕ is a first-order formula, an **NP** machine can (a) guess the truth tables for $R_1 \ldots R_k$ (they have polynomial size); then (b) evaluate Φ in deterministic polynomial time (there are only polynomially many choices for the constant number of first-order quantifiers that might appear in ϕ, so we can just enumerate all of them).

For the **NP** $\subseteq$ **ESO** direction, we build a single **ESO** formula that encodes all possible computations of a given **NP** machine M as a gigantic $n^k \times n^k$ table C where $C[s,t]$ is the value of tape cell s (possibly including the state of the controller, if it is parked on cell s) at time t. As usual we will simplify our life by restricting M to use only a single tape, not move off the left end of the tape, and clean up to get some convenient final configuration if it accepts.

The formula will look a little bit like the SAT formula constructed in the Cook-Levin proof, but the difference is that our formula will have fixed size and have to work for all input sizes and all inputs. To build this formula, we will use the following main ideas:

1. For a machine that runs in $n^k - 1$ steps, we can represent times t and positions on s as k-tuples using the technique described in §14.3.

2. Using second-order existential quantifiers, we can guess relations $C_1, C_2, \ldots, C_q$, where $C_i(s,t)$ is true if $C[s,t] = i$.

3. Using first-order universal quantifiers, we can enumerate all positions s and times t, and then write a first-order formula for each $C[s,t]$ saying that it is consistent with $C[s-1,t-1]$, $C[s,t-1]$, and $C[s+1,t-1]$.

4. Finally, we do the usual thing of demanding that the input match $C[0,0]$ through $C[n-1,0]$ and that the final state $C[0,n^k-1]$ is accepting.

The resulting formula will look something like this:

$$\begin{aligned}\exists C_1 \exists C_2 \ldots \exists C_k \forall s \forall t \exists s_{-1} \exists s_{+1} \exists t_{-1} : \\ &[\text{exactly one of } C_i[s,t] \text{ is true}] \\ &\wedge (s = 0 \vee s = s_{-1} + 1) \\ &\wedge (s = n^k - 1 \vee s_{+1} = s + 1 \\ &\wedge (t = 0 \vee t = t_{-1} + 1) \\ &\wedge [C[s,t] \text{ is consistent with } C[s-1,t-1],\ C[s,t-1], \text{ and } C[s+1,t-1]].\end{aligned}$$

We leave filling out the details of the various pieces of this formula and showing that they can in fact be expressed in first-order logic as an exercise that we don't actually expected anybody to do. The messy bit is expressing consistency, but this is basically just a gigantic first-order formula with no quantifiers, which we could easily derive from the transition function for M if we actually had to. □

14.5 Descriptive characterization of PH

We can use Fagin's Theorem to get a descriptive-complexity version of the polynomial-time hierarchy. Recall that L is in **PH** if there is a formula of the form $\forall w_1 \exists w_2 \forall w_3 \ldots \exists w_k P(x, w_1, \ldots, w_k)$ such that each string w_i has length polynomial in $|x|$ and P is polynomial-time computable. Given such a language, we can represent it as a second-order formula $\forall W_1 \exists W_2 \forall W_3 \ldots \exists W_k \exists C_1 \ldots \exists C_q \phi$ where $W_1 \ldots W_k$ are encodings of $w_1 \ldots w_k$ as relations over an appropriate number of variables and the remainder of the formula is as in the proof of Fagin's Theorem. Conversely, if we have a second-order formula, we can use an alternating Turing machine to fill in the polynomial-size truth tables for each quantified relation and check the value of the first-order part, all in polynomial time.

This shows that second-order logic expresses precisely the predicates computable in **PH**, or **SO** = **PH**.

If we look at the construction in more detail, we can actually say something stronger. What we've really shown is that, for odd k, the languages in $\mathbf{\Sigma}_k^p$ are precisely those expressible using a $\mathbf{\Sigma_k SO}$ formula, one that has k alternating second-order $\exists$ and $\forall$ quantifiers starting with $\exists$. This is because we can combine the last $\exists$ quantifier with the $\exists$ quantifier for **ESO**. Similarly, for even k, $\mathbf{\Sigma}_k^p = \mathbf{\Pi_k SO}$. But now we can take complements to cover the missing cases. The final result is that $\mathbf{\Sigma}_k^p = \mathbf{\Sigma_k SO}$ and $\mathbf{\Pi}_k^p = \mathbf{\Pi_k SO}$ for all k.

14.6 Descriptive characterization of NL and L

The idea behind Fagin's theorem is that there is a one-to-one correspondence between relations defined over tuples of elements of a finite structure and bit-vectors of polynomial size. We also used the lemma that we could represent numbers of polynomial size as tuples of elements of the structure. We know that polynomial-size numbers correspond to logarithmic-size bit vectors. So in principle it seems like we ought to be able to encode the state of, say, a log-space machine as a tuple of elements representing a number.

To turn this into a representation of log-space computation, we need two additional pieces of machinery. The first is a predicate $\texttt{BIT}(x, i)$ that extracts the i-th bit of a number x (represented as a tuple of elements). This is not something we can define in standard first-order logic, since it would allow us to compute parity. However, we can do it using the second piece of machinery, which is an extra operator `DTC` (**deterministic transitive closure**) or `TC` (**transitive closure**) that allows us to iterate a formula.

14.6.1 Transitive closure operators

Suppose $\phi(x, y)$ is a formula defined over some logic, where x and y are k-tuples of variables. Then we define $\texttt{TC}(\phi, s, t)$ to be true if and only if there is a sequence of tuples of variables $s = x_0, x_1, \ldots, x_m = t$ such that $\phi(x_i, x_{i+1})$ holds for every i. We similarly define $\texttt{DTC}(\phi, s, t)$ to be true if there is exactly one such sequence, or equivalently if for each x_i there is exactly one x_{i+1} such that $\phi(x_i, x_{i+1})$ is true.

We can now define the class of $\mathbf{FO}(\texttt{TC})$ formulas recursively, as including all statements that we can construct by either applying a built-in predicate like $<$ or P to variables; by taking the negation, AND, or OR of $\mathbf{FO}(\texttt{TC})$ formulas; by applying $\exists x$ or $\forall x$ to an $\mathbf{FO}(\texttt{TC})$ formula; or by applying `TC` to an $\mathbf{FO}(\texttt{TC})$ formula. The class $\mathbf{FO}(\texttt{DTC})$ is defined the same way, except using `DTC` instead of `TC`.

14.6.2 Arithmetic in FO(DTC)

We want to show that we can implement BIT in **FO**(TC) (and thus also in **FO**(DTC). We've already shown how to implement 0 and successor over tuples of elements using just **FO**. The next step is to implement addition:

$$x + y = z \equiv \mathtt{DTC}(\phi(\langle x, y\rangle, \langle x', y'\rangle), \langle x, y\rangle, \langle 0, z\rangle)$$

where

$$\phi(\langle x, y\rangle, \langle x', y'\rangle) \equiv (x = x' + 1) \wedge (y' = y + 1)$$

Note that, as with successor, we are really implementing a predicate $+(x, y, z)$ that is true whenever $x+y$ happens to be equal to z. That the above definition works is easily shown by induction, using the hypothesis each tuple $\langle x, y\rangle$ in the sequence has the same sum.

Now we can use addition to implement parity:

$$(x \bmod 2) = 1 \equiv \exists y : y + y = x$$

and division by 2

$$: \lfloor x/2 \rfloor = y \equiv \exists r : (r = 0 \vee \exists z : (z = 0 \wedge z + 1 = r)) \wedge \exists y_2 (y + y = y_2 \wedge x = y_2 + r),$$

and, as originally promised, BIT:

$$\mathtt{BIT}(x, i) \equiv \exists y : \mathtt{DTC}(\phi, \langle x, i\rangle, \langle y, 0\rangle) \wedge (y \bmod 2) = 1$$

where

$$\phi(\langle x, i\rangle, \langle x', i'\rangle) \equiv (\lfloor x/2 \rfloor = x') \wedge (i' + 1 = i).$$

14.6.3 Expressing log-space languages

Suppose we have a deterministic log-space machine M, and we want to write a formula that expresses whether or not $M(x)$ accepts. We can do this in **FO**(DTC) using the following approach. As usual we assume that M is a one-tape machine just to make our life easier, and we will also assume that whether or not it accepts can be detected by observing a particular bit of its configuration that occurs only in terminal accepting states.

1. We represent configurations of M as bit-vectors of length $O(\log n)$, corresponding to a sequence of tape cell states possible with a tape head state attached.

2. Using the BIT operator and our ability to do arithmetic, we can write a formula ϕ in **FO**(DTC) such that $\phi(x, x')$ holds if and only if x' is a configuration of the machine that can follows from x.

3. Also using BIT plus arithmetic, construct a predicate α that recognizes the initial configuration of the machine (based on the input as accessed through P or $\rightarrow$).

4. Evaluate the formula $\exists s \exists t : \alpha(s) \wedge \texttt{DTC}(\phi, s, t) \wedge \texttt{BIT}(t, i)$, where i is the fixed bit that indicates acceptance.

Because we are using DTC, this formula will be true if and only if there is a sequence $s = x_0 \ldots x_m = t$ such that s is the initial configuration, each x_i leads deterministically to x_{i+1}, and $x_m = t$ is an accepting state. This puts $\mathbf{L} \subseteq \mathbf{FO}(\texttt{DTC})$. If we adjust ϕ to allow nondeterministic transitions and use TC instead of DTC, then we get $\mathbf{NL} \subseteq \mathbf{FO}(\texttt{TC})$ instead. We will show in the next section that both of these containments are in fact equality.

14.6.4 Evaluating FO(TC) and FO(DTC) formulas

To show that we can evaluate a formula ϕ in **FO**(TC), we apply **structural induction**, where our induction hypothesis is that any subformula can be evaluated in **L** and we must show that the formula as a whole can be. The possible structures we can have are:

1. $P(i)$ for some i. Here we just check the i-th bit of the input. (If we have a graph formula, this is messier, but still clearly in **L**.)

2. $\neg\phi$, $\phi \wedge \rho$, $\phi \vee \rho$: In each case, we carry out one or two log-space computations and combine the results as appropriate.

3. $\exists x \phi$, $\forall x \phi$. Using $O(\log n)$ space to hold x, enumerate all possibilities and evaluate ϕ (also using $O(\log n)$ space) on each. Return the OR or AND of the results depending on the quantifier.

4. $\texttt{DTC}(\phi, s, t)$. Use $O(\log n)$ space to hold a pair of tuples x and x', plus a counter c. Initialize x to s and c to $n^c > 2^{|s|}$. For each iteration, enumerate all possible x' and test $\phi(x, x')$. If we get 0 or more than one solution, return false; if we reach t, return true; if c drops to 0, return false (we are looping). Otherwise, set x to the unique solution x', decrement c, and try again.

This shows that $\mathbf{FO}(\texttt{DTC}) \subseteq \mathbf{L}$ and thus that $\mathbf{FO}(\texttt{DTC}) = \mathbf{L}$.

The reason this works is that $\mathbf{L}^{\mathbf{L}} = \mathbf{L}$, so whenever we need to call a **L** subroutine to evaluate some subformula, we can do so and stay within **L**. For **NL**, this is less obvious, but we have Immerman-Szelepcsényi to save us: since $\mathbf{NL}^{\mathbf{NL}} = \mathbf{NL}$, we can call a **NL** subroutine to evaluate a subformula and stay in **NL**. This covers pretty much everything we did in the above list, with the only thing being missing an implementation of TC. But this is a straightforward modification of the DTC code: instead of enumerating all possible x' and counting those for which $\phi(x, x')$ is true, we nondeterministically guess x' and reject if $\phi(x, x')$ is false. This gives $\mathbf{FO}(\texttt{TC}) \subseteq \mathbf{NL}$ and thus $\mathbf{FO}(\texttt{DTC}) = \mathbf{NL}$.

14.7 Descriptive characterization of PSPACE and P

Fixed-point operators give the ability to work on sequence of relations each define in terms of previous one. This gives power similar to the transitive closure operators, but with a sequence of relations instead of a sequence of tuples of elements. Since a relation can represent polynomially-many bits, it is not surprising that using a fixpoint operator gives us **PSPACE**.

14.7.1 $\mathbf{FO}(\texttt{PFP}) = \mathbf{PSPACE}$

The PFP (**partial fixed point**) operator of applies to a formula $\phi(P, x)$ where P and x are free variables (for x, possibly a tuple of free variables). To evaluate $\texttt{PFP}(\phi, y)$, let P_0 be the empty relation (that is false on all arguments). Then let $P_{i+1}(x) = \phi(P_i, x)$; this means that to compute whether P_{i+1} is true on x, ϕ can use the entire truth table for P_i, plus the value of x. If at some point $P_{i+1} = P_i$, then $\texttt{PFP}(\phi, y) = P_i(y)$ for that value of i. If this does not occur, then $\texttt{PFP}(\phi, y)$ is false.

It's easy to evaluate $\texttt{PFP}(\phi, y)$ in **PSPACE**: since the truth table for P_i has polynomial size, we can store it and evaluate $\phi(P_i, y)$ on all (polynomially-many) y to compute P_{i+1}. We can then re-use the space for P_i to store P_{i+2}, and so on. Either we eventually reach a fixed point $P_{i+1} = P_i$, and can read $\texttt{PFP}(\phi, y)$ directly from the truth table, or we reach $i > 2^{|P|}$; in the latter case, we are looping, and so we can return false. Since we can evaluate the rest of first-order logic in **PSPACE** using essentially the same approach as we used to put $\mathbf{FO}(\texttt{DTC})$ in **L**, this gives $\mathbf{FO}(\texttt{PFP}) \subseteq \mathbf{PSPACE}$.

In the other direction, given a **PSPACE** machine M, we can construct

a first-order formula ϕ such that $P_{i+1}(x) = \phi(P_i, x)$ is true if and only if the x-th bit of the state of M at time $i+1$ is 1, given that P_i describes the state of M at time i. This is essentially the same construction as we used to show $\mathbf{L} \subseteq \mathbf{FO}(\texttt{DTC})$; the only difference is that now we are using a relation to store the state instead of encoding it in a variable.[2] This gives $\mathbf{SPACE} \subseteq \mathbf{FO}(\texttt{PFP})$ and thus $\mathbf{PSPACE} = \mathbf{FO}(\texttt{PFP})$.

14.7.2 $\mathbf{FO}(\texttt{LFP}) = \mathbf{P}$

The LFP (**least fixed point**) operator is similar to PFP, but when computing P_{i+1} from P_i we let $P_{i+1}(x) = P_i(x) \vee \phi(P_i, x)$. This means once we set a bit in the relation, we can't unset it later, and make LFP less of a full-blown iteration operator and more of a tool for building up definitions recursively. Formally, we again start with P_0 empty, iterate until $P_{i+1} = P_i$ and define $\texttt{LFP}(\phi, y) = P_i(y)$. Note that because the sequence of relations is non-decreasing, we must eventually hit some fixed point, so there is no need to define what happens if we don't.

One way to think about LFP is that it expresses recursive definitions, where we say that x has some property P if we can show some collection of of precursors $a, b, c, \ldots$ have property P. For example, if we want to define the property that x is even (given a successor operation), we can write it as $\texttt{LFP}(\phi, x)$ where $\phi(P, x) \equiv (x = 0) \vee Px \vee (\exists y \exists z : Py \wedge z = y + 1 \wedge x = z + 1)$.

Computationally, the nice thing about LFP is that we always reach the fixed point in polynomially-many iterations, because there are only polynomially-many bits to set in each P_i. This means that we can evaluate $\texttt{LFP}(\phi, y)$ in $\mathbf{P}$, and more generally we have $\mathbf{FO}(\texttt{LFP}) \subseteq \mathbf{P}$.

In the other direction, given an machine M and input x, we can use LFP to fill in the tableau for the Cook-Levin theorem. The idea is that we let P_i contain the first i rows of the tableau, corresponding to the first i steps of M. We can easily build a formula ϕ that extends this by one row at each iteration. This gives $\mathbf{P} \subseteq \mathbf{FO}(\texttt{LFP})$ and thus $\mathbf{P} = \mathbf{FO}(\texttt{LFP})$.

A consequence of this fact is that we can recast the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem in terms of the relative expressiveness of different logics: $\mathbf{P} = \mathbf{NP}$ if and only if $\mathbf{FO}(\texttt{LFP}) = \mathbf{SO}$, or in other words if the full power of second-order logic (when applied to finite models) adds nothing to the ability to write recursive definitions.

[2] Conveniently, this means we don't need to build BIT, since we can index P_i directly.

Chapter 15

Interactive proofs

Last updated 2017. Some material may be out of date.

An **interactive proof** [GMR89] generalizes the notion of certificates in **NP**; it involves a **verifier** V, which is a randomized polynomial-time Turing machine, and a **prover** P, which is an arbitrary collection of functions that respond to a sequence of questions posed by the verifier.

Each **round** of the interactive proof consists of V flipping some coins to produce a question, and P responding. For a k-round protocol we have a sequence of $2k$ messages alternately going from the V to P and vice versa. At the end of this sequence of messages, V decides to accept or reject its original input x.

An **interactive proof system** is just the code for V, allowing it to carry out interactive proofs. An interactive proof system for a language L is **complete** if, whenever x is in L, there exists a prover P that causes V to accept with probability at least $2/3$, and **sound** if every prover P causes V to accept with probability at most $1/3$. We say that L is in **IP** if there is an interactive proof system that uses polynomially-many rounds and that is both complete and sound for L.

The reason we make the verifier randomized is that a deterministic verifier only gives us **NP**: An **NP**-machine can guess the entire sequences of questions and answers, and check that (a) each question is in fact what the original verifier would have asked at that step; and (b) the final decision is accept. Since a deterministic verifier accepts or rejects with probability 1, if there is any set of answers from the prover that works, then the **NP**-machine will find it, and if there isn't, it won't.

It's also worth noting that the probability bounds are the same as for **BPP**, and as with **BPP**, we can repeat a protocol multiple times and take

majorities to amplify them. So we will be happy if we get any probability bounds that are separated by at least a polynomial in n, and will assume if needed that we can make the actual gap be $\epsilon, 1 - \epsilon$ where ϵ may be exponentially small.

15.1 Private vs. public coins

The original definition of **IP** assumed **private coins**: when the prover chooses its answers, it can see the input and the questions from the verifier, but it can't tell what coins the verifier used to generate the questions.

A similar class, the class of **Arthur-Merlin games** [BM88], assumes instead that the protocol uses **public coins**, where the prover can observe both the questions the verifier asks and the coins used to generate them.

This actually allows for a very simple definition of the class. There is no need for Arthur (the verifier) to actually pose any questions: since Merlin is all-powerful and can observe Arthur's (past) coins, Merlin can simply answer the question Arthur would have asked. This means that we can model Arthur in terms of a nondeterministic Turing machine as a node that averages over all choices of coins and Merlin as a node that takes the maximum probability of causing the machine to accept. A language L is decided by such a machine if it accepts any $x \in L$ with probability at least $2/3$ and any $x \notin L$ with probability at most $1/3$, where in both cases Merlin (the maximizer nodes) are trying the maximize the probability of acceptance.

The class **AM** consists of all games where Arthur goes first and Merlin responds. The class **MA** has Merlin go first, generating a single, deterministic witness intended to maximize the probability that Arthur accepts: this is essentially **NP** with a **BPP** verifier instead of a **NP** verifier. In general, **AM**$[k]$ consists of k layers of alternating averaging and maximizing nodes, with maximizing nodes coming last. This makes **AM** = **AM**[2].

In principle, private coins (as in **IP**) would seem to provide more power to the verifier than public coins (as in **AM**), since it should be easier for the verifier to catch a cheating prover's lies if the verifier has extra private information that the prover doesn't know about. This turns out not to be the case. Below, we will give an example of how we can replace a private-coin protocol for a particular problem with a public-coin protocol, and then talk a bit about how this generalizes.

15.1.1 GRAPH NON-ISOMORPHISM with private coins

The **GRAPH NON-ISOMORPHISM** (**GNI**) problem takes as input two graphs G and H, and asks if G is not isomorphic to H ($G \not\simeq H$). Recall that $G \simeq H$ if there is a permutation of the vertices of G that makes it equal to H. An **NP**-machine can easily solve the GRAPH ISOMORPHISM problem by guessing the right permutation; this puts GRAPH NON-ISOMORPHISM in **coNP**. We can also give a simple one-round protocol using private coins that puts it in **IP**.

Given G and H, the verifier picks one or the other with equal probability, then randomly permutes its vertices to get a test graph T. It then asks the prover which of G or H it picked.

If the graphs are not isomorphic, the prover can distinguish $T \simeq G$ from $T \simeq H$ and answer the question correctly with probability 1. If they are isomorphic, then the prover can only guess: T gives no information at all about which of G or H was used to generate it. So in this case it answers correctly only with probability $1/2$. By running the protocol twice (which can even be done in parallel), we can knock this probability down to $1/4$, which gets us outside the $(1/3, 2/3)$ gap needed for soundness and completeness.

15.1.2 GRAPH NON-ISOMORPHISM with public coins

The problem with the preceding protocol using public coins is that the prover can ignore T and just tell the verifier the outcome of its initial coin-flip. This is not very convincing. So instead we will have the prover do something else: it will show that size of the set of $S = \{T \mid T \simeq G \vee T \simeq H\}$ possible test graphs T is large ($2n!$) when $G \not\simeq H$, in a way that does not allow it to do so when S is small ($n!$) when $G \simeq H$.

The method for doing this is to use an approximate counting protocol due to Goldwasser and Sipser [GS86], who in fact extended this to convert any private-coin protocol into a public-coin protocol. The Goldwasser-Sipser protocol in general allows a prover to demonstrate that a set S is big whenever it can prove membership in S. The intuition is that if S makes up a large part of some universe Ω, then the verifier can just pick some ω uniformly in Ω and have the prover demonstrate that ω is in S, which will cause the verifier to accept with probability $|S|/|\Omega|$. The problem is finding an Ω so that this probability will show a polynomial-size (ideally constant gap) between large and small S.

For example, with graphs, the verifier could try to generate a random graph R and ask the prover to demonstrate that R is isomorphic to at least

one of G and H. But this gives bad probabilities: since there are $2^{\binom{n}{2}} \gg 2n!$ possible graphs R, it is exponentially improbable that R would be isomorphic to either. So we need to crunch the space down to make S take up a larger proportion of the possible choices.

We can do this using a pairwise-independent random hash function. Let n be the number of bits used to represent each element of S. Let N be such that a large S has size at least N and a small S has size at most $N/2$. (For GRAPH NON-ISOMORPHISM, $N = 2n!$.) To distinguish whether S is large or small, pick m such that $2^{n-1} \leq N \leq 2^n$, and consider the family of hash functions $h_{ab} : \{0,1\}^n \to \{0,1\}^m$ given by $h_{ab}(x) = (ax + b) \bmod 2^m$ where multiplication and addition are done over the finite field $GF[2^n]$ and a and b are chosen uniformly and independently at random from $GF[2^n]$. These hash functions have the property of **pairwise-independence**: if $x \neq y$, for any x' and y' the probability $\Pr[h_{ab}(x) = x' \wedge h_{ab}(y) = y']$ is exactly 2^{-2m}. To prove this, observe first that for any x'' and y'', $\Pr[ax + b = x'' \wedge ax + b = y'']$ is exactly 2^{-2n}, since given $x \neq y$, x'', and y'', we can solve for the unique values of a and b that make the system of linear equations hold. Now sum over all $2^{2(n-m)}$ choices of x'' and y'' that map through $\bmod 2^m$ to a specific pair x' and y'.

Now the idea is that the verifier will pick a random hash function h, and ask the prover to find a graph R such that $h(R) = 0$, together with an explicit isomorphism $R \simeq G$ or $R \simeq H$. If $G \simeq H$, the chance that the prover can pull this off will be smaller than if $G \not\simeq H$, since there will be fewer candidate graphs R.

For any fixed graph R, the probability that $h(R) = 0$ is exactly 2^{-m}. If $G \simeq H$, there are at most $n!$ possible graphs R that the prover could choose from, giving a probability of at most $2^{-m}n!$ that it convinces the verifier. If $G \not\simeq H$, then there are $2n!$ possible choices. The event that at least one such choice maps to 0 is bounded from below by $2n! \cdot 2^{-m} - \binom{2n!}{2}2^{-2m} > 2n! \cdot 2^{-m} - \frac{1}{2}(2n! \cdot 2^{-m})^2$ by the inclusion-exclusion principle. If we let $p = 2n! \cdot 2^{-m}$, we get a gap between p and $2(p - p^2)$ for the cases where G and H are or are not isomorphic. This gives a gap of $p - 2p^2$, which will be nonzero for $p < 1/4$. We can't set p arbitrarily, since we can only make 2^{-m} a power of 2, but by picking m to make $1/16 \leq p \leq 1/8$ we can get a nontrivial gap that we can then amplify by repetition if needed.

15.1.3 Simulating private coins

It turns out that we can simulate private coins with public coins in general, not just for GRAPH ISOMORPHISM, a result due to Goldwasser and

Sipser [GS86]. This means that we don't need to make a distinction between the public coins used in Arthur-Merlin games and the private ones used in interactive proofs, and can pick either model depending on which is more convenient.

We give a very sketchy description of the argument below, specialized for the one-round case. This description roughly follows some lecture notes of Madhu Sudan; for the full argument, see the paper. It helps to amplify the probabilities a lot first: we will assume that we have a private-coin interactive proof where the probability that the verifier fails to recognize a valid proof is exponentially small.

The intuition is that the same hashing trick used to approximately count graphs in GRAPH NONISOMORPHISM works for approximately counting anything: if the verifier wants to distinguish many objects with some property from few, it can pick a random hash function, appropriately tuned, and ask the prover to supply an object that has the property and hashes to a particular value. To get rid of private coins, what we want to do is get the prover to demonstrate that there are many choices of private coins that will cause the verifier to accept.

Recall that in a one-round private-coin protocol, V chooses some random r and computes a question $q(x, r)$, P responds with an answer $a(x, q)$, and V then chooses whether to accept or not based on x, q, and r. What we want to do is get the prover to demonstrate that there are many questions for which it has good answers, that is, answers which are likely to cause the verifier to accept.

Let S_{qa} be set of random bits that cause verifier to ask q and accept answer a. Let $N_q = \max_a |S_{qa}|$. Then the maximum probability that the prover can get the verifier to accept in the original private-coin protocol is $2^{-|r|} \cdot \sum_q N_q$. To show that the sum is big, we'll pick some reasonable value N, and have the prover demonstrate that there are many q such that $N_q \geq N$.

For any particular q, the prover can demonstrate that $N_q \geq N$ by picking the best possible a (which it will tell to the verifier) and showing that S_{qa} is large (using the hashing trick). To show that there are many good q, we wrap this protocol up another layer of the hashing trick. So the full protocol looks like this:

1. The verifier picks a hash function h_1.
2. The prover responds with a question q such that $h_1(q) = 0$ and an answer a.

3. The verifier picks a second hash function h_2.

4. The prover responds with an $r \in S_{qa}$ such that $h_2(r) = 0$.

If the prover fails to get $h_1(q) = 0$ or $h_2(r) = 0$, the verifier rejects, otherwise it accepts. Assuming we tune the ranges of h_1 and h_2 correctly, the prover can only win with high probability if there are many q (first round) such that N_q is large (second round). But this implies $\sum_q N_q$ is large, meaning that the original $x \in L$.

15.2 IP = PSPACE

In this section, we sketch a proof of Shamir's surprising result that **IP** = **PSPACE** [Sha92]. This had a big impact when it came out, because (a) the result showed that **IP** had more power than anybody expected, (b) the technique used doesn't relativize in any obvious way, and (c) it was one of the few results out there that shows equality between two complexity classes that don't seem to be obviously connected in some way. As a bonus, if for some reason you don't like one of **IP** or **PSPACE**, you can forget about it and just use the other class instead.

15.2.1 IP $\subseteq$ PSPACE

This is the easy direction: express a **IP** computation as polynomially-deep game tree of averages (verifier moves) and maxima (prover moves) over polynomially-sized choices. The obvious recursive algorithm evaluates the probability that the verifier accepts, assuming an optimal prover; we can then just check if this probability is nonzero or not.

15.2.2 PSPACE $\subseteq$ IP

To show **PSPACE** $\subseteq$ **IP**, we'll give an interactive proof system for the **PSPACE**-complete language TQBF of true quantified Boolean formulas. The technique involves encoding a quantified Boolean formula as a (very large) polynomial over $\mathbb{Z}_p$ for some suitable prime p, and then using properties of polynomials to get the prover to restrict the original problem down to a case the verifier can check, while using properties of polynomials to keep the prover from cheating during the restriction process. We will start by showing how to do this for #SAT, which will give the weaker result $\mathbf{P}^{\#\mathbf{P}} \subseteq \mathbf{IP}$.

15.2.2.1 Arithmetization of #SAT

Here we want to show that if a Boolean formula has exactly k solutions, the prover can convince the verifier of this. The main technique is **arithmetization**: we replace the Boolean operations in the formula, which apply only to 0 or 1, with arithmetic operations over a field $\mathbb{Z}_p$ that give the same answers for 0 and 1. This is done in the obvious way:

$$\begin{aligned} x \wedge y &\equiv x \cdot y \\ \neg x &\equiv 1 - x \\ x \vee y &\equiv 1 - (1-x) \cdot (1-y) = x + y - x \cdot y \end{aligned}$$

When arithmetizing a formula, we won't actually rewrite it, because this would make the formula bigger if it involves ORs. Instead, we will use the same Boolean operators as in the original formula and just remember their arithmetic interpretations if we need to apply them to numbers that aren't 0 or 1.

We now want to play something like the following game: Given a formula $\phi(x_1, \ldots, x_m)$, the verifier asks the prover to tell it how many solutions it has. To check this number, the verifier will ask the prove to split this total up between $\phi(0, x_2, \ldots, x_m)$ and $\phi(1, x_2, \ldots, x_m)$. It can then pick one of these subtotals and split it up into two cases, repeating the process until it gets down to a fixed assignment to all of the variables, which it can check for itself.

The problem is that the prover can lie, and it's hard for the verifier to catch the lie. The prover's trick will be to offer answers for $\#\text{SAT}(\phi(0, x_2, \ldots, x_m))$ and $\#\text{SAT}(\phi(1, x_2, \ldots, x_m))$ that add up to the claimed total, and make one of them be the real answer. This gives it a 50% chance at each step of having the verifier recurse into a subtotal about which the prover is not actually lying. Over m variables, there is only a 2^{-m} chance that the verifier picks the bogus answer at each step. So when it gets to the bottom, it is likely to see a true answer even if the initial total was wrong.

This is where the polynomials come in. We will use the fact that, if $p(x)$ and $q(x)$ are polynomials of degree at most d in a single variable $x \in \mathbb{Z}_p$, and $p \neq q$, then $\Pr[p(r) = q(r)]$ for a random $r \in \mathbb{Z}_p$ is at most d/p. This follows from the Fundamental Theorem of Arithmetic, which says that the degree-d polynomial $p - q$ has at most d zeros. To apply this fact, we need to get the prover to express their claim about $\#\text{SAT}(\phi)$ in terms of a polynomial, so we can use polynomial magic to force its lies to be consistent until they get small enough that we can detect them.

First observe that the arithmetized version of ϕ is a degree-n multivariate polynomial in $x_1, \ldots, x_m$, where n is the size of ϕ. This follows via a straightforward induction argument from the fact that ϕ contains at most n AND or OR operations, and each such operation introduces exactly one multiplication. Now define the family of polynomials $f_0, f_1, \ldots, f_m$, where

$$f_i(x_1, \ldots, x_i) = \sum_{y_{i+1} \in \{0,1\}} \sum_{y_{i+2} \in \{0,1\}} \cdots \sum_{y_m \in \{0,1\}} \phi(x_1, \ldots, x_i, y_{i+1}, \ldots, y_m).$$

These polynomials correspond to the stages of restricting the prover's answers: $f_0()$ just gives the total number of solutions to ϕ, while $f_i(x_1, \ldots, x_i)$ gives the total number of solutions with given fixed values for the first i variables. Because they are all sums over restrictions of the degree-n polynomial ϕ, they all have degree at most n.

Unfortunately, we can't just ask the prover to tell us all the f_i, because a degree-n *multivariate* polynomial can still be exponentially large as a function of the number of variables. So we will instead supply the prover with fixed values for all but the last variable in f_i, and ask it to tell us the univariate degree-n polynomial that only depends on the last variable, this being the variable for which we just got rid of the summation.[1] Formally, we define

$$g_i(z) = f_i(r_1, \ldots, r_{i-1}, z) = \sum_{y_{i+1} \in \{0,1\}} \sum_{y_{i+2} \in \{0,1\}} \cdots \sum_{y_m \in \{0,1\}} \phi(r_1, \ldots, r_{i-1}, z, y_{i+1}, \ldots, y_m),$$

where $r_1, \ldots, r_{i-1}$ will be random elements of $\mathbb{Z}_p$ chosen during the computation.

Here is the actual protocol:

1. The verifier asks the prover to supply $k = f_0() = \#\text{SAT}(\phi)$, as well as a convenient prime p between 2^n and 2^{n-1}. (Such a prime always exists by Bertrand's Postulate.) It checks that p is in fact prime.

2. At stage $i = 1$, the verifier asks the prover to supply $g_1(z) = f_1(z)$, and tests it for consistency by checking $g_1(0) + g_1(1) = k$.

3. At each stage $i > 1$, the verifier chooses a random $r_{i-1} \in \mathbb{Z}_p$, sends it to the prover, and asks for the polynomial $g_i(z) = f_i(r_1, \ldots, r_{i-1}, z)$. It tests g_i for consistency by checking $g_i(0) + g_i(1) = g_{i-1}(r_{i-1})$.

[1] Later, we will look at more complicated formulas, where we may want to test different variables at different times. But the idea will be that there is always some single variable that we were previously summing or quantifying over that we now need to plug in 0 or 1 for, and in subsequent rounds we will free up that slot by making it random.

4. After all m stages, the verifier checks that $g_m(z)$ is in fact the same polynomial as $f_1(r_1, \ldots, r_{m-1}, z)$. It could do this probabilistically by setting z to a random r_m, or it could be lazy and just insist that the prover provide $g_m(z)$ in a form that is syntactically identical to $\phi(r_1, \ldots, r_{m-1}, z)$.

If all the tests pass, the verifier accepts the prover's claim about k. Otherwise it rejects.

Technical note: As described, this protocol doesn't really fit in **IP**, because #SAT(ϕ) is not a decision problem. To make it fit, we'd have to have the verifier supply k and consider the decision problem #SAT$_D(\phi, k)$ of determining if ϕ has exactly k solutions. But in the context of a $\mathbf{P}^{\#\mathbf{P}}$ problem, we can get away with just using the above protocol, since to simulate $\mathbf{P}^{\#\mathbf{P}}$ the prover is trying to convince the verifier that there is a sequence of oracle calls with corresponding answers $k_1, k_2, \ldots$ that would cause the oracle machine to accept, and for this purpose it's fine to have the prover supply those answers, as long the verifier can check that they actually work.

If the prover is not lying, it just supplies the correct value for k and the correct g_i at each step. In this case the verifier accepts always.

If the prover is lying, then in order to avoid getting caught it must stop lying at some stage in the protocol. A lying prover supplies a sequence of polynomials g'_1, g'_2, $\ldots g'_m$, and it's in trouble if $g'_m \neq g_m$. So for the cheating prover to get away with it, there has to be some i for which $g'_{i-1} \neq g_{i-1}$ but $g'_i = g_i$.

Suppose that this occurs. Then $g_i(0) + g_i(1) = g_{i-1}(r_{i-1)}$, from the definition of g_i. But we also have $g_i(0) + g_(1) = g'_{i-1}(r_{i-i1})$, since this is tested explicitly by the verifier. The means the prover gets away with swapping in a correct value for g'_i only if $g'_{i-1}(r_{i-1}) = g_{i-1}(r_{i-1})$. But the prover hasn't seen r_{i-1} yet when it picks g'_{r-1}, so r_{i-1} is independent of this choice and has only a $d/p < n/2^n$ chance of making this equation true. It follows that a lying prover has at most an $n/2^n$ chance of successfully escaping at each step, for an at most $n^2/2^n$ chance of convincing the verifier overall.

This is exponentially small for sufficiently large n, so the probability of error is still exponentially small even if we use the same protocol polynomially many times to simulate polynomially many #**P** oracle calls. So this gives $\mathbf{P}^{\#\mathbf{P}} \subseteq \mathbf{IP}$, and in fact the protocol has the even stronger property of having only one-sided error (which turns out to be feasible for any problem in **IP**).

15.2.3 Arithmetization of TQBF

Now we want to do the same thing to TQBF that we did to #SAT. We can arithmetize the Boolean $\forall x$ and $\exists y$ operators the same way that we arithmetized $\wedge$ and $\vee$:

$$\forall x \in \{0,1\}\, \phi(x) \equiv \prod_{x \in \{0,1\}} \phi(x),$$
$$\exists x \in \{0,1\}\, \phi(y) \equiv \coprod_{y \in \{0,1\}} \phi(x) = 1 - \prod_{y \in \{0,1\}} (1 - \phi(y)).$$

(It should be noted that using the **coproduct** $\coprod$ for $\exists$ is not standard notation, but it seems like the right thing to do here.)

The problem with this arithmetization is that if we apply it directly to a long quantified Boolean formula, we double the degree for each quantifier and get an exponentially high degree overall. This is going to be trouble for our polynomial-time verifier, even aside from giving a lying prover exponentially many zeros to play with. Curiously, this does not happen if we limit ourselves to Boolean values, because then $x^2 = x$ and we can knock ϕ down to a multilinear polynomial with degree at most m. But this doesn't work over $\mathbb{Z}_p$ unless we do something sneaky.

The sneaky thing to do is to use a **linearization operator** L_x that turns an arbitrary polynomial p into a polynomial $L_x(p)$ that is (a) linear in x, and (b) equal to p for $x \in \{0,1\}$. This operator is easily defined by

$$L_x(p(x, y_1, \ldots, y_m)) = (1 - x) \cdot p(1, y_1, \ldots, y_m) + x \cdot p(0, y_1, \ldots, y_m).$$

Using this linearization operator, we will push down the degree of each subformula of ϕ whenever some nasty quantifier threatens to push it up.

Suppose that ϕ has the form $Q_1 x_1 Q_2 x_2 \ldots Q_m x_m \phi(x_1, \ldots, x_m)$, where each Q_i is $\forall$ or $\exists$. Write L_i for L_{x_i}. Then construct a polynomial $f_0() = (Q_1 x_1) L_1 (Q_2 x_2) L_1 L_2 \ldots (Q_n x_n) L_1 \ldots L_n \phi(x_1, \ldots, x_n)$, and derive from it a sequence of polynomials $f_1, f_2, \ldots$ where each f_i strips off the first i operators.

We now have to do a test at each stage similar to the test for #SAT that $f_i(x_1, \ldots, x_i) = f_{i-1}(x_1, \ldots, x_{i-1}, 0) + f_{i-1}(x_1, \ldots, x_{i-1}, 1)$. But now the test depends on which operator we are removing:

- For $\forall x_j$, we check $f_{i-1}(x_1, \ldots, x_{j-1}) = f_i(x_1, \ldots, x_{j-1}, 0) \cdot f(x_1, dots, x_{j-1}, 1)$. In doing this check, we make all variables except x_j be random values set previously, and what the prover sends is the univariate polynomial $g_i(z)$ that fixes each other $x_{j'}$ to its most recent random assignment. After doing the check, we set $x_j = r_i$ for some random i.

- For $\exists x_j$, we check $f_{i-1}(x_1, \ldots, x_{j-1}) = 1(1 - f_i(x_1, \ldots, x_{j-1}, 0)) \cdot (1 - f(x_1, dots, x_{j-1}, 1))$. As in the previous case, $g_i(z)$ is the univariate polynomial that fixes all variables except x_j.

- For L_{x_j}, we check $f_{i-1} = (1 - x_j) f_i(x_1, dots, x_{j-1}, 0, x_{j+1}, \ldots, x_k) + x_j f_i(x_1, dots, x_{j-1}, 1, x_{j+1}, \ldots, x_k)$. Here $g_i(z)$ is again a univariate polynomial in x_j. What is a little odd is that x_j may be a variable we previously fixed, but that doesn't stop us from doing the test. It does mean that for subsequent stages we need to assign x_j a new random value r_i independent of its previous value or values, to prevent any possibility of the prover exploiting its prior knowledge.

In each of these cases, checking that $g_i(r_i)$ gives a consistent value enforces that the prover tells a consistent story unless r_i happens to land on a zero of the difference between a correct and bogus polynomial. The error analysis is essentially the same as for the #SAT cases; over polynomially many tests we get a total probability of missing a cheating prover's lies of $n^c/2^n = o(1)$, assuming as before that $p > 2^n$. This puts TQBF in **IP** and thus gives **PSPACE** $\subseteq$ **IP**.

Chapter 16

Probabilistically-checkable proofs and hardness of approximation

Last updated 2017. Some material may be out of date.

In this chapter, we discuss results relating the hardness of various approximation problems to the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ question. In particular, we will show how a result known as the **PCP theorem** can be used to prove the impossibility of getting tight approximations for many common approximation problems assuming $\mathbf{P} \neq \mathbf{NP}$. The PCP theorem shows that the certificates provided to a **NP**-machine can be replaced by **probabilistically-checkable proofs**, which can be verified by a randomized Turing machine that uses r random bits to select q bits of the proof to look at, and accepts bad proofs with less than some constant probability ρ while accepting all good proofs.

This turns out to have strong consequences for approximation algorithms. We can think of a probabilistically-checkable as a kind of constraint satisfaction problem, where the constraints apply to the tuples of q bits that the verifier might look at, the number of constraints is bounded by the number of possible random choices 2^r, and each constraint enforces some condition on those q bits corresponding to the verifier's response to seeing them. If we can satisfy at least $\rho \cdot 2^r$ of the constraints, we've constructed a proof that is not bad. This means that there is a winning certificate for our original **NP** machine, and that whatever input x we started with is in the language accepted by that machine. In other words, we just converted an approximation algorithm for a particular constraint-satisfaction algorithm

(assuming it runs in deterministic polynomial time) into a polynomial-time procedure to solve a **NP**-hard problem. Conversely, if $\mathbf{P} \neq \mathbf{NP}$, we can't construct such an approximation algorithm.

This is only a sketchy, high-level overview of where we are going. Below we fill in some of the details. We are mostly following the approach of [AB07, §§18.1–18.4].

16.1 Probabilistically-checkable proofs

A $\langle r(n), q(n), \rho \rangle$-**PCP verifier** for a language L consists of two polynomial-time computable functions f and g, where:

- $f(x, r)$ takes an input x of length n and a string r of length $r(n)$ and outputs a sequence i of $q(n)$ indices $i_1, i_2, \ldots, i_{q(n)}$, each in the range $0 \ldots q(n) \cdot 2^{r(n)}$; and
- $g(x, \pi_i)$ takes as input the same x as f and a sequence $\pi_i = \pi_{i+1}\pi_{i_2} \ldots \pi_{i_{(q)n}}$ and outputs either 1 (for accept) or 0 for (reject); and
- if $x \in L$, then there exists a sequence π that causes $g(x, \pi_{f(x,r)})$ to output 1 always (**completeness**); and
- if $x \notin L$, then for any sequence π, $g(x, \pi_{f(x,r)})$ outputs 1 with probability at most ρ (**soundness**).

We call the string π a **probabilistically-checkable proof** or **PCP** for short.

Typically, ρ is set to $1/2$, and we just write $\langle r(n), q(n) \rangle$-**PCP** verifier for $\langle r(n), q(n), 1/2 \rangle$-**PCP** verifier.

The class $\mathbf{PCP}(r(n), q(n))$ is the class of languages L for which there exists an $\langle O(r(n)), O(q(n)), 1/2 \rangle$-**PCP** verifier.

The **PCP theorem** says that $\mathbf{NP} = \mathbf{PCP}(\log n, 1)$. That is, any language in **NP** can be recognized by a **PCP**-verifier that is allowed to look at only a constant number of bits selected using $O(\log n)$ random bits from a proof of polynomial length, which is fooled at most half the time by bad proofs. In fact, 3 bits is enough [Hås01]. We won't actually prove this here, but we will describe some consequences of the theorem, and give some hints about how the proof works.

16.1.1 A probabilistically-checkable proof for GRAPH NON-ISOMORPHISM

Here is a probabilistically-checkable proof for GRAPH NON-ISOMORPHISM, based on the interactive proof from §15.1.1. This is not a very good proof, because it is $2^{\binom{n}{2}}$ bits long and requires $\Theta(n \log n)$ random bits to query. But it only requires the verifier to check one bit.

Recall that in the interactive proof protocol for GRAPH NON-ISOMORPHISM, the verifier picks one of the two input graphs G and H, permutes its vertices randomly, shows the permuted graph T to the prover, and asks the prover to guess whether the chosen graph was G or H. If the graphs are not isomorphic, the (infinitely powerful) prover can win this game every time. If they are, it can only win half the time, since T is isomorphic to both G and H and gives no information about which one was picked.

To turn this into a probabilistically-checkable proof, have the prover build a bit-vector π indexed by every possible graph on n vertices, writing a 1 for each graph that is isomorphic isomorphic to H. Now the verifier can construct T as above, look it up in this gigantic table, and accept if and only if (a) it chose G and $\pi[T] = 0$, or (b) it chose H and $\pi[T] = 1$. If G and H are non-isomorphic, the verifier accepts every time. But if they are isomorphic, no matter what proof π' is supplied, there is at least a $1/2$ chance that $\pi'[T]$ is wrong. This puts GRAPH NON-ISOMORPHISM in $\mathbf{PCP}(n^2, 1)$ (the n^2 is to allow the proof to be long enough).

16.2 $\mathbf{NP} \subseteq \mathbf{PCP}(\text{poly}(n), 1)$

Here we give a weak version of the PCP theorem, showing that any problem in **NP** has a probabilistically-checkable proof where the verifier uses polynomially-many random bits but only needs to look at a constant number of bits of the proof: in other words, $\mathbf{NP} \subseteq \mathbf{PCP}(\text{poly}(n), 1)$.[1] The proof itself will be exponentially long.

The idea is to construct a $\langle \text{poly}(n), 1) \rangle$-PCP for a particular **NP**-complete problem; we can then take any other problem in **NP**, reduce it to this problem, and use the construction to get a PCP for that problem as well.

[1] This is a rather weaker result, since (a) the full PCP theorem gives **NP** using only $O(\log n)$ random bits, and (b) $\mathbf{PCP}(\text{poly}(n), 1)$ is known to be equal to **NEXP** [BFL91]. But the construction is still useful for illustrating many of the ideas behind probabilistically-checkable proof.

16.2.1 QUADEQ

The particular problem we will look at is QUADEQ, the language of systems of quadratic equations over $GF(2)$ that have solutions.

This is in **NP** because we can guess and verify a solution; it's **NP**-hard because we can use quadratic equations over $GF(2)$ to encode instances of SAT, using the representation 0 for false, 1 for true, $1-x$ for $\neg x$, xy for $x \wedge y$, and $1-(1-x)(1-y) = x+y+xy$ for $x \vee y$. We may also need to introduce auxiliary variables to keep the degree from going up: for example, to encode the clause $x \vee y \vee z$, we introduce an auxiliary variable q representing $x \vee y$ and use two equations

$$x + y + xy = q,$$
$$q + z + qz = 1$$

to enforce the constraints $q = x \vee y$ and $1 = q \vee z = x \vee y \vee z$. It will be helpful later to rewrite these in a standard form with only zeros on the right:

$$q + x + y + xy = 0$$
$$q + z + qz + 1 = 0.$$

This works because we can move summands freely from one side of an equation to the other since all addition is mod 2.

16.2.2 The Walsh-Hadamard Code

An **NP** proof for QUADEC just gives an assignment to the variables that makes all the equations true. Unfortunately, this requires looking at the entire proof to check it. To turn this into a $\mathbf{PCP}(\text{poly}(n), 1)$ proof, we will make heavy use of a rather magical **error-correcting code** called the **Walsh-Hadamard code**.

This code expands an n-bit string x into a 2^n-bit **codeword** $H(x)$, where $H(x)_i = x \cdot i$ when x and the index i are both interpreted as n-dimensional vectors over $GF(2)$ and $\cdot$ is the usual vector dot-product $\sum_{i=1}^{n} x_j i_j$. This encoding has several very nice properties, all of which we will need:

1. It is a linear code: $H(x+y) = H(x) + H(y)$ when all strings are interpreted as vectors of the appropriate dimension over $GF(2)$.

2. It is an error-correcting code with distance 2^{n-1}. If $x \neq y$, then exactly half of all i will give $x \cdot i \neq y \cdot i$. This follows from the **subset sum principle**, which says that a random subset of a non-empty set S is

equally likely to have an even or odd number of elements. (Proof: From the Binomial Theorem, $\sum_{\text{even } i} \binom{n}{i} - \sum_{\text{odd } i} \binom{n}{i} = \sum_{i=0}^{n} (-1)^n \binom{n}{i} = (1 + (-1))^n = 0^n = 0$ when $n \neq 0$.) So for any particular nonzero x, exactly half of the $x \cdot i$ values will be 1, since i includes each one in x with independent probability 1/2. This makes $d(H(0), H(x)) = 2^{n-1}$. But then $d(H(x), H(y)) = d(H(x), d(H(x+y))) = 2^{n-1}$ whenever $x \neq y$.

3. It is **locally testable**: Given an alleged codeword w, we can check if w is close to being a legitimate codeword $H(x)$ by sampling a constant number of bits from w. (We do not need to know what x is to do this.)

   Our test is: Pick two indices i and j uniformly at random, and check if $w_i + w_j = w_{i+j}$. A legitimate codeword will pass this test always. It is also possible to show using Fourier analysis (see [AB07, Theorem 19.9]) that if w passes this test with probability $\rho \geq 1/2$, then there is some x such that $\Pr_i[H(x)_i = w_i] \geq \rho$ (equivalently, $d(H(x), w) \leq 2^n(1-\rho)$, in which case we say w is **ρ-close** to $H(x)$.

4. It is **locally decodable**: If w is ρ-close to $H(x)$, then we can compute $H(x)_i$ by choosing a random index r and computing $H(x)_r + H(x)_{i+r}$. This will be equal to $H(x)_i$ if both bits are correct (by linearity of H). The probability that both bits are corrects is at least $1 - 2\delta$ if $\rho = 1 - \delta$.

5. It allows us to check an unlimited number of linear equations in x by looking up a single bit of $H(x)$. This again uses the subset sum principle. Give a system of linear equations $x \cdot y_1 = 0, x \cdot y_2 = 0, \ldots x \cdot y_m = 0$, choose a random subset S of $\{1, \ldots, m\}$ and query $H(x)_{\sum_{i \in S} y} = \sum_{i \in S} x \cdot y$. This will be 0 always if the equations hold and 1 with probability 1/2 if at least one is violated.

   This gets a little more complicated if we have any ones on the right-hand side. But we can handle an equation of the form $x \cdot y = 1$ by rewriting it as $x \cdot y + 1 = 0$, and then extending x to include an extra constant 1 bit (which we can test is really one by looking up $H(x)_i$ for an appropriate index i).

16.2.3 A PCP for QUADEC

So now to construct a PCP for QUADEC, we build:

1. An n-bit solution u to the system of quadratic equations, which we think of as a function $f : \{0,1\}^n \to \{0,1\}$ and encode as $f = H(x)$.

2. An n^2-bit vector $w = u \otimes u$ where $(u \otimes u)_{ij} = u_i u_j$, which we encode as $g = H(x \otimes x)$.

To simplify our life, we will assume that one of the equations is $x_1 = 1$ so that we can use this constant 1 later (note that we can trivially reduce the unrestricted version of QUADEC to the version that includes this assumption by adding an extra variable and equation). A different approach that does not require this assumption is given in [AB07, §18.4.2, Step 3].

To test this PCP, the verifier checks:

1. That f and g are $(1 - \delta)$-close to real codewords for some suitably small δ.

2. That for some random r, r', $f(r)f(r') = g(r \otimes r')$. This may let us know if w is inconsistent with u. Define W as the $n \times n$ matrix with $W_{ij} = w_{ij}$ and U as the $n \times n$ matrix $U = u \otimes u$ (so $U_{ij} = u_i u_j$). Then $g(r \otimes r') = w \cdot (r \otimes r') = \sum_{ij} w_{ij} r_i r'_j = rWr'$ and $f(r)f(r') = (u \cdot r)(u \cdot r') = (\sum_i u_i r_i)\left(\sum_j u_j r'_j\right) = \sum_{ij} r_i U_{ij} r_j = rUr'$. Now apply the random subset principle to argue that if $U \neq W$, then $rU \neq rW$ at least half the time, and if $rU \neq rW$, then $rUr' \neq rUr'$ at least half the time. This gives a probability of at least $1/4$ that we catch $U \neq W$, and we can repeat the test a few times to amplify this to whatever constant we want.

3. That our extra constant-1 variable is in fact 1 (lookup on u).

4. That w encodes a satisfying assignment for the original problem. This just involves checking a system of linear equations using w.

Since we can make each step fail with only a small constant probability, we can make the entire process fail with the sum of these probabilities, also a small constant.

16.3 PCP and approximability

Suppose we want to use the full **PCP** theorem $\mathbf{NP} = \mathbf{PCP}(\log n, 1)$ to actually decide some language L in **NP**. What do we need to do?

16.3.1 Approximating the number of satisfied verifier queries

If we can somehow find a PCP for $x \in L$ and verify it, then we know $x \in L$. So the obvious thing is to try to build an algorithm for generating PCPs.

But actually generating a PCP may be hard. Fortunately, even getting an good approximation will be enough. We illustrate the basic idea using MAX SAT, the problem of finding an assignment that maximizes the number of satisfied clauses in a 3CNF formula.

Suppose that we have some language L with a PCP verifier V. If $x \in L$, there exists a proof of polynomial length such that every choice of q bits by V from the proof will be accepted by V. We can encode this verification step as a Boolean formula: for each set of bits $S = \{i_1, \ldots, i_q\}$, write a constant-size formula ϕ_S with variable in π that checks if V will accept $\pi_{i_1}, \ldots, \pi_{i_q}$ for our given input x. Then we can test if $x \in L$ by testing if $\phi = \bigwedge_S \phi_S$ is satisfiable or not.

But we can do better than this. Suppose that we can approximate the number of ϕ_S that can be satisfied to within a factor of $2 - \epsilon$. Then if ϕ has an assignment that makes all the ϕ_S true (which followed from completeness if $x \in L$), our approximation algorithm will give us an assignment that makes at least a $\frac{1}{2-\epsilon} > \frac{1}{2}$ fraction of the ϕ_S true. But we can never make more than $\frac{1}{2}$ of the ϕ_S true if $x \notin L$. So we can run our hypothetical approximation algorithm, and if it gives us an assignment that satisfies more than half of the ϕ_S, we know $x \in L$. If the approximation runs in **P**, we just solved SAT in **P** and showed **P** = **NP**.

16.3.2 Gap-preserving reduction to MAX SAT

Maximizing the number of subformulas ϕ_S that are satisfied is a strange problem, and we'd like to state this result in terms of a more traditional problem like MAX SAT. We can do this by converting each ϕ_S into a 3CNF formula (which makes ϕ also 3CNF), but the cost is that we reduce the **gap** between negative instances $x \notin L$ and negative instances $x \in L$.

The **PCP** theorem gives us a gap of $(1/2, 1)$ between negative and positive instances. If each ϕ_S is represented by k 3CNF clauses, then it may be that violating a single ϕ_S only maps to violating one of those k clauses. So where previously we either satisfied at most $1/2$ of the ϕ_S or all of them, now we might have a negative instance where we can still satisfy a $1 - \frac{1}{2k}$ of the clauses. So we only get **P** = **NP** if we are given a poly-time approximation algorithm for MAX SAT that is at least this good; or, conversely, we only show that **P** $\neq$ **NP** implies that there is no MAX SAT approximation that gets more than $1 - \frac{1}{2k}$ of optimal.

This suggests that we want to find a version of the **PCP** theorem that makes k as small as possible. Fortunately, Håstad [Hås01] showed that it is possible to construct a PCP-verifier for 3SAT with the miraculous property

that (a) q is only 3, and (b) the verification step involves testing only if $\pi_{i_1} \oplus \pi_{i_2} \oplus \pi_{i_3} = b$, where i_1, i_2, i_3, and b are generated from the random bits.

There is a slight cost: the completeness parameter of this verifier is only $1-\epsilon$ for any fixed $\epsilon > 0$, meaning that it doesn't always recognize valid proof, and the soundness parameter is $1/2 + \epsilon$. But checking $\pi_{i_1} \oplus \pi_{i_2} \oplus \pi_{i_3} = b$ requires a 3CNF formula of only 4 clauses. So this means that there is no approximation algorithm for MAX SAT that does better than $7/8 + \delta$ of optimal in all cases, unless $\mathbf{P} = \mathbf{NP}$. This matches the 7/8 upper bound given by just picking an assignment at random.[2]

This is an example of a reduction argument, since we reduced 3SAT first to a problem of finding a proof that would make a particular PCP-verifier happy and then to MAX SAT. The second reduction is an example of a **gap-preserving reduction**, in that it takes an instance of a problem with a non-trivial gap $(1/2 + \epsilon, 1 - \epsilon)$ and turns it into an instance of a problem with a non-trivial gap $(7/8 + \epsilon, 1 - \epsilon)$. Note that do be gap-preserving, a reduction doesn't have to preserve the value of the gap, it just has to preserve the existence of a gap. So a **gap-reducing reduction** like this is still gap-preserving. We can also consider **gap-amplifying reductions**: in a sense, Håstad's verifier gives a reduction from 3SAT to 3SAT that amplifies the reduction from the trivial $(1 - 1/m, 1)$ that follows from only being able to satisfy $m - 1$ of the m clauses in a negative instance to the much more useful $(1/2 + \epsilon, 1 - \epsilon)$.

16.3.3 Other inapproximable problems

Using inapproximability of MAX SAT, we can find similar inapproximability results for other **NP**-complete optimization problems by looking for gap-preserving reductions from MAX SAT. In many cases, we can just use whatever reduction we already had for showing that the target problem was **NP**-hard. This gives constant-factor inapproximability bounds for problems like GRAPH 3-COLORABILITY (where the value of a solution is

[2] It's common in the approximation-algorithm literature to quote approximation ratios for maximization problems as the fraction of the best solution that we can achieve, as in a 7/8-approximation for MAX SAT satisfying 7/8 of the maximum possible clauses. This leads to rather odd statements when we start talking about lower bounds ("you can't do better than $7/8 + \delta$") and upper bounds ("you can get at least 7/8"), since the naming of the bounds is reversed from what they actually say. For this reason complexity theorists have generally standardized on always treating approximation ratios as greater than 1, which for maximization problems means reporting the inverse ratio, $8/7 - \epsilon$ in this case. I like 7/8 better than 8/7, and there is no real possibility of confusion, so I will stick with 7/8.

the proportion of two-colored edges) and MAXIMUM INDEPENDENT SET (where the value of a solution is the size of the independent set). In each case we observe that a partial solution to the target problem maps back to a partial solution to the original SAT problem.

In some cases we can do better, by applying a gap-amplification step. For example, suppose that no polynomial-time algorithm for INDEPENDENT SET can guarantee an approximation ratio better than ρ, assuming $\mathbf{P} \neq \mathbf{NP}$. Given a graph G, construct the graph G^k on $\binom{n}{k}$ vertices where each vertex in G^k represents a set of k vertices in G, and ST is an edge in G^k if $S \cup T$ is *not* an independent set in G. Let I an independent set for G. Then the set I^k of all k-subsets of I is an independent set in G^k ($S \cup T \subseteq I$ is an independent set of any S and T in I^k). Conversely, given any independent set $J \subseteq G^k$, its union $\bigcup J$ is an independent set in G (because otherwise there is an either within some element of J or between two elements of J). So any maximum independent set in G^k will be I^k for some maximum independent set in G.

This amplifies approximation ratios: given an independent set I such that $|I|/|OPT| = \rho$, then $|I^k|/|OPT^k| = \binom{|I|}{k}/\binom{|OPT|}{k} \approx \rho^k$. If k is constant, we can compute G^k in polynomial time. If we can then compute a ρ^k-approximation to the maximum independent set in G^k, we can take the union of its elements to get a ρ-approximation to the maximum independent set in G. By making k sufficiently large, this shows that approximating the maximum independent set to within any constant $\epsilon > 0$ is **NP**-hard.

There is a stronger version of this argument that uses expander graphs to get better amplification, which shows that $n^{-\delta}$ approximations are also **NP**-hard for any $\delta < 1$. See [AB07, §18.3] for a description of this argument.

16.4 Dinur's proof of the PCP theorem

Here we give a very brief outline of Dinur's proof of the **PCP** theorem [Din07]. This is currently the simplest known proof of the theorem, although it is still too involved to present in detail here. For a more complete description, see §18.5 of [AB07], or Dinur's paper, which is pretty accessible.

A **constraint graph** is a graph $G = (V, E)$, where the vertices in V are interpreted as variables, and each edge uv in E carries a **constraint** $c_{uv} \subseteq \Sigma^2$ that specifies what assignments of values in some **alphabet** Σ are permitted for the endpoints of uv. A **constraint satisfaction problem** asks for an assignment $\sigma : V \to \Sigma$ that minimizes $\mathrm{UNSAT}_\sigma(G) = \Pr_{uv \in E}[\langle\langle \sigma(u), \sigma(v)) \notin c_{uv}]$, the probability that a randomly-chosen constraint is unsatisfied. The quantity $\mathrm{UNSAT}(G)$ is defined as minimum value of $\mathrm{UNSAT}_\sigma(G)$: this is

the smallest proportion of constraints that we must leave unsatisfied. In the other direction, the **value** $\text{val}(G)$ of a constraint satisfaction problem is $1 - \text{UNSAT}(G)$: this is the largest proportion of constraints that we can satisfy. [3]

An example of a constraint satisfaction problem is GRAPH 3-COLORABILITY: here $\Sigma = \{r, g, b\}$, and the constraints just require that each edge on the graph we are trying to color (which will be the same as the constraint graph G!) has two different colors on its endpoints. If a graph G with m edges has a 3-coloring, then $\text{UNSAT}(G) = 0$ and $\text{val}(G) = 1$; if G does not, then $\text{UNSAT}(G) \geq 1/m$ and $\text{val}(G) \leq 1 - 1/m$. This gives a $(1 - 1/m, 1)$ gap between the best value we can get for non-3-colorable vs. 3-colorable graphs. Dinur's proof works by amplifying this gap.

Here is the basic idea:

1. We first assume that our input graph is k-regular (all vertices have the same degree k) and an expander (every subset S with $|S| \leq m/2$ has $\delta|S|$ external neighbors for some constant $\delta > 0$). Dinur shows how these assumptions can be enforced without breaking **NP**-hardness.

2. We then observe that coloring our original graph G has a gap of $(1-1/m, 1)$, or that $\text{UNSAT}(G) \geq 1/m$. This follows immediately from the fact that a bad coloring must include at least one monochromatic edge.

3. To amplify this gap, we apply a two-stage process.

   First, we construct a new constraint graph G' (that is no longer a graph coloring problem) with n vertices, where the constraint graph has an edge between any two vertices at distance $2d + 1$ or less in G, the label on each vertex v is a "neighborhood map" assigning a color of every vertex within distance d of v, and the constraint on each edge uv says that the maps for u and v (a) assign the same color to each

[3] Though Dinur's proof doesn't need this, we can also consider a **constraint hypergraph**, where each q-**hyperedge** is a q-tuple $e = v_1 v_2 \dots v_q$ relating q vertices, and a constraint c_e is a subset of Σ^q describing what assignments are permitted to the vertices in e. As in the graph case, our goal is to minimize $\text{UNSAT}_\sigma(G)$, which is now the proportion of hyperedges whose constraints are violated, or equivalently to maximize $\text{val}_\sigma(G) = 1 - \text{UNSAT}_\sigma(G)$. This gives us the q-CSP problem: given a q-ary constraint hypergraph, find an assignment σ that maximizes $\text{val}_\sigma(G)$. An example of a constraint hypergraph arises from 3SAT. Given a formula ϕ, construct a graph G_ϕ in which each vertices represents a variable, and the constraint on each 3-hyperedge enforces least one of the literals in some clause is true. The MAX 3SAT problem asks to find an assignment σ that maximizes the proportion of satisfied clauses, or $\text{val}_\sigma(G_\phi)$.

vertex in the overlap between the two neighborhoods, and (b) assigns colors to the endpoints of any edge in either neighborhood that are permitted by the constraint on that edge. Intuitively, this means that a bad edge in a coloring of G will turn into many bad edges in G', and the expander assumption means that many bad edges in G will also turn into many bad edges in G'. In particular, Dinur shows that with appropriate tuning this process amplifies the UNSAT value of G by a constant. Unfortunately, we also blow up the size of the alphabet by $\Theta(k^d)$.

So the second part of the amplification knocks the size of the alphabet back down to 2. This requires replacing each node in G' with a set of nodes in a new constraint graph G'', where the state of the nodes in the set encodes the state of the original node, and some coding-theory magic is used to preserve the increased gap from the first stage (we lose a little bit, but not as much as we gained).

The net effect of both stages is to take a constraint graph G of size n with $\text{UNSAT}(G) \geq \epsilon$ and turn it into a constraint graph G'' of size cn, for some constant c, with $\text{UNSAT}(G'') \geq 2\epsilon$.

4. Finally, we repeat this process $\Theta(\log m) = \Theta(\log n)$ times to construct a constraint graph with size $c^{O(\log n)}n = \text{poly}(n)$ and gap $(1/2, 1)$. Any solution to this constraint graph gives a PCP for GRAPH 3-COLORABILITY for the original graph G.

16.5 The Unique Games Conjecture

The **PCP** theorem, assuming $\mathbf{P} \neq \mathbf{NP}$, gives us fairly strong inapproximability results for many classic optimization problems, but in many cases these are not tight: there is a still a gap between the lower bound and the best known upper bound. The **Unique Games Conjecture** of Khot [Kho02], if true, makes many of these bounds tight.

The Unique Games Conjecture was originally formulated in terms of an interactive proof game with two provers. In this game, the verifier V picks a query q_1 to ask of prover P_1 and a query q_2 to ask of prover P_2, and then checks consistency of the prover's answers a_1 and a_2. (The provers cannot communicate, and so answer the queries independently, although they can coordinate their strategy before seeing the queries.) This gives a **unique game** if, for each answer a_1 there is exactly one answer a_2 that will cause V to accept.

Equivalently, we can model a unique game as a restricted 2-CSP: the labels on the vertices are the answers, and the consistency condition on each edge is a bijection between the possible labels on each endpoint. This corresponds to the two-prover game the same way PCPs correspond to single-prover games, in that a labeling just encodes the answers given by each prover.

A nice feature of unique games is that they are easy to solve in polynomial time: pick a label for some vertex, and then use the unique constraints to deduce the labels for all the other vertices. So the problem becomes interesting mostly when we have games for which there is no exact solution.

For the Unique Games Conjecture, we consider the set of all unique games with gap $(\delta, 1-\epsilon)$; that is, the set consisting of the union of all unique games with approximations with ratio $1-\epsilon$ or better and all unique games with no approximations with ratio better than δ. The conjecture states that for any δ and ϵ, there exists some alphabet size k such that it is **NP**-hard to determine of these two piles a unique game G with this alphabet size lands in.[4]

Unfortunately, even though the Unique Games Conjecture has many consequences that are easy to state (for example, the usual 2-approximation to MINIMUM VERTEX COVER is optimal, as is the 0.878-approximation to MAX CUT of Goemans and Williamson [GW95]), actually proving these consequences requires fairly sophisticated arguments. So we won't attempt to do any of them here, and instead will point the interested reader to Khot's 2010 survey paper [Kho10], which gives a table of bounds known at that time and citations to where they came from.

There is no particular consensus among complexity theorists as to whether the Unique Games Conjecture is true or not, but it would be nifty if it were.

[4]Note that this is not a decision problem, in that the machine M considering G does not need to do anything sensible if G is in the gap; instead, it is an example of a **promise problem** where we have two sets L_0 and L_1, $M(x)$ must output i when $x \in L_i$, but $L_0 \cup L_1$ does not necessarily cover all of $\{0,1\}^*$.

Appendix A

Assignments

Assignments are typically due Thursdays at 23:00.

Assignments should be uploaded to Canvas in PDF format.

Do not include any identifying information in your submissions. This will allow grading to be done anonymously.

Make sure that your submissions are readable. You are strongly advised to use LaTeX, Microsoft Word, Google Docs, or similar software to generate typeset solutions. Scanned or photographed handwritten submissions often come out badly, and submissions that are difficult for the graders to read will be penalized.

Sample solutions will appear in this appendix after the assignment is due. To maintain anonymity of both submitters and graders, questions about grading should be submitted through Canvas.

[[[**To be announced.**]]]

Appendix B

Sample assignments from Spring 2017

B.1 Assignment 1: due Wednesday, 2017-02-01 at 23:00

B.1.1 Bureaucratic part

Send me email! My address is james.aspnes@gmail.com.

In your message, include:

1. Your name.
2. Your status: whether you are an undergraduate, grad student, auditor, etc.
3. Anything else you'd like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

B.1.2 Binary multiplication

A **finite-state transducer** (FST) is a Turing machine with a read-only input tape, a write-only output tape, no work tapes, and heads that can only stay put or move right at each step. We would like to get a finite-state transducer to multiple binary numbers by 3.

1. Suppose that the input and output are both given most-significant-bit (MSB) first: for example, the input 6 is represented by $\llcorner 6 \lrcorner = 110$ and the corresponding output $3 \cdot 6 = 18$ is represented by $\llcorner 18 \lrcorner = 10010$. Give a program for a finite-state transducer that multiplies its input by 3 using this representation, or show that no such program is possible.

2. Suppose instead that the input is given least-significant-bit (LSB) first: now 6 is represented by $\llcorner 6 \lrcorner^{\mathrm{R}} = 011$ and $3 \cdot 6 = 18$ is represented by $\llcorner 18 \lrcorner^{\mathrm{R}} = 01001$. Give a program for a finite-state transducer that multiplies its input by 3 using this representation, or show that no such program is possible.

Solution

1. A finite-state transducer cannot multiple binary numbers by 3 with the MSB-first representation.

   Let x_k, for each integer $k > 0$, be the input with binary representation $(10)^k$. Then $x_k = 2\sum_{i=0}^{k-1} 4^i = 2 \cdot \frac{4^k-1}{3}$, and $3x_k = 2 \cdot (4^k - 1)$, which has the binary representation $1^{2k-1}0$.

   Now compare with $x_k + 1$: $\llcorner x_k + 1 \lrcorner = (10)^{k-1}11$ differs from $\llcorner x_k \lrcorner = (10)^{k-1}10$ only in the last bit, but $\llcorner 3(x_k + 1) \lrcorner = 10^{2k}1$ already differs from $\llcorner 3x_k \lrcorner = 1^{2k-1}0$ on the second bit. We will use this fact to argue that any FST for this problem must give the wrong answer for some input x_k or $x_k + 1$.

   Fix some FST M. Consider executions of M on x_k and x_{k+1}. The second bit in M's output is the complement of the last bit it reads, so if it outputs more than one bit before reading the last bit of its input, it will be incorrect in one of the two executions. It follows that a correct M cannot output more than one bit without reading its entire input.

   Now consider executions with inputs x_k, where k ranges over all positive integers. Let q_k be the state of the finite-state controller when M first reaches a configuration where the input head is over the last bit of the input. There are infinitely many k, but only finitely many possible q_k, so there must be two values $k \neq k'$ such that $q_k = q_{k'}$. We have previously established that when M reaches the last input bit in either x_k or $x_{k'}$, it is in state q_k and has output at most one bit. Any subsequent bits it outputs depend only on q_k and the remaining input bit (0), since it can't move the input head left to see any of the other bits. So the outputs $M(x_k)$ and $M(x_{k'})$ differ by at most the presence

q	read	q'	write	move
$\langle 0,0\rangle$	0	$\langle 0,0\rangle$	0	R
$\langle 0,0\rangle$	1	$\langle 0,1\rangle$	1	R
$\langle 0,0\rangle$	b	$\langle 0,0\rangle$	b	S
$\langle 0,1\rangle$	0	$\langle 0,0\rangle$	1	R
$\langle 0,1\rangle$	1	$\langle 1,1\rangle$	0	R
$\langle 0,1\rangle$	b	$\langle 0,0\rangle$	1	R
$\langle 1,0\rangle$	0	$\langle 0,0\rangle$	1	R
$\langle 1,0\rangle$	1	$\langle 1,1\rangle$	0	R
$\langle 1,0\rangle$	b	$\langle 0,0\rangle$	1	R
$\langle 1,1\rangle$	0	$\langle 1,0\rangle$	0	R
$\langle 1,1\rangle$	1	$\langle 1,1\rangle$	1	R
$\langle 1,1\rangle$	b	$\langle 1,0\rangle$	0	R

Table B.1: Transition table for multiplying by 3 (LSB first)

or absence of a single initial bit. But $\llcorner 3x_k \lrcorner$ and $\llcorner 3x_{k'} \lrcorner$ differ by at least two bits, so M gives the wrong answer for at least one of them.

2. But with the LSB-first representation, there is no problem. One way to see this is that we can compute $3x$ as $x + 2x$, and $\llcorner 2x \lrcorner^{\mathrm{R}} = 0\llcorner x \lrcorner^{\mathrm{R}}$ is just the input shifted right one position.

   If we are processing the input from LSB to MSB, at each step we need to add together (a) the current input bit, (b) the previous input bit (for $2x$), and (c) whatever carry bit we have from the previous position. This gives us a value at most 3; we write the low-order bit and keep the high-order bit as the carry for the next iteration. Between tracking the and the previous bit we need four states, which turns out to be enough for the entire computation. A transition table is given in Figure B.1; here each state is $\langle \text{carry}, \text{previous}\rangle$.

   It is possible to optimize this a bit further. We can notice that the behavior of the TM is the same in states $\langle 0,1\rangle$ and $\langle 1,0\rangle$. So we could actually reduce to just three states, representing a combined carry and shifted input value of 0, 1, or 2.

B.1.3 Transitivity of O and o

Use the definitions given in §3.1.2.1 to show that:

1. If $f(n) = o(g(n)$, then $f(n) = O(g(n))$.

2. If $f(n) = o(g(n))$ and $g(n) = O(h(n))$, then $f(n) = o(h(n))$.

Solution

1. Fix some $c > 0$. Then $f(n) = o(g(n))$ means that there is some N such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$. But the existence of c and N with this property means that $f(n) = O(g(n))$.

2. We want to show that for any $c > 0$, there exists N such that $f(n) \leq c \cdot h(n)$ for all $n \geq N$. Fix some such c. Let c_2 and N_2 be such that $g(n) \leq c_2 \cdot h(n)$ for all $n \geq N_2$. Let $c_1 = c/c_2$ and let N_1 be such that $f(n) \leq c_1 g(n)$ for all $n \geq N_1$. Then for any $n \geq \max(N_1, N_2)$, we have $f(n) \leq c_1 g(n) \leq c_1 c_2 h(n) = (c/c_2)c_2 h(n) = ch(n)$ as required.

It's worth noting that essentially the same argument for part 2 shows that $f(n) = O(g(n))$ and $g(n) = o(h(n))$ together imply $f(n) = o(h(n))$, but one tedious proof is enough.

B.2 Assignment 2: due Wednesday, 2017-02-15 at 23:00

B.2.1 A log-space reduction

The usual definition of **NP**-completeness uses polynomial-time reductions, where $A \leq_{\mathbf{P}} B$ if there is a polynomial-time computable f such that $x \in A$ if and only if $f(x) \in B$. But we could also consider other kinds of reductions, characterized by a different restriction on f.

One of these is a **log-space reduction**. We write $A \leq_{\mathbf{L}} B$ if there is a function f computable on a standard Turing machine using $O(\log n)$ space such that $x \in A$ if and only if $f(x) \in B$.

Show that INDEPENDENT SET $\leq_{\mathbf{L}}$ CLIQUE, where the input k and $G = (V, E)$ to both problems is given by 1^k, followed by a delimiter of some sort (say, ";"), followed by $1^{|V|}$, followed by another delimiter, and finally a sequence of pairs of vertex ids representing the edges (in no particular order), with each vertex id represented as a binary number in the range $0 \ldots |V| - 1$, terminated by the delimiter.

You do not need to (and probably shouldn't, unless you are bored and immortal) give an explicit transition table, but you should describe the

workings of a log-space Turing machine that computes f in enough detail that you can argue that it does in fact run in logarithmic space.

Solution

The usual polynomial-time reduction from INDEPENDENT SET to CLIQUE replaces G with its complement $\overline{G}$, which has the same vertices but contains each possible edge uv if and only if G does not contain uv. So we would like to implement this mapping in log space.

For $\llcorner k \lrcorner$ and $\llcorner |V| \lrcorner$, we can just copy the input until we reach the second delimiter. This requires no space beyond a few states in the finite-state controller. So the only difficult part is complementing E.

One what to do this is to use three tapes to keep track of binary representations of $v = |V|$, and counters i and j that run from 0 to $|V| - 1$. Observe that using only a finite number of states in the controller, we can do all of the following tasks in space $O(\log|V|)$:

1. Set a counter to 0.
2. Increment v, i, or j by 1.
3. Compare i or j to v.
4. Search E for an edge ij or ji.
5. Write edge ij to the output tape.

We can then run the algorithm given in Algorithm B.1. Each of the steps in this algorithm can be done without using any work tape space beyond the space needed to represent v, i, and j, so the total space complexity is $O(\log|V|)$. This is logarithmic in the size of the input because $|V|$ is expressed in unary.[1]

B.2.2 Limitations of two-counter machines

Recall that a **two-counter machine** consists of a finite-state controller, a read-only input tape, a write-only output tape, and two counters, that support increment and decrement operations, and that can be tested for equality with zero by the finite-state controller.

[1] It is convenient for us that whoever picked the representation of G made this choice. If k and $|V|$ were presented in binary, we might have to worry about what happens with very sparse graphs.

```
1  Copy ⌞k⌟ and ⌞|V|⌟ to the output tape.
2  v ← 0
3  i ← 0
4  for each 1 in the representation of |V| do
5      Increment v
6  while i ≠ v do
7      j ← 0
8      while j ≠ v do
9          if ij ∉ E and ji ≠ E then
10             Write ij to the output
11         Increment j
12     Increment i
```

Algorithm B.1: Log-space reduction from INDEPENDENT SET to CLIQUE

Show that if $f(n) \geq n$ is time-constructible (by a standard Turing machine) then there exists a language L that can be decided by a two-counter machine eventually, but that cannot be decided by a two-counter machine in $o(f(n))$ time.

Solution

Let $L = \{\langle \llcorner M \lrcorner, x\rangle \mid M \text{ is a two-counter machine that rejects } x \text{ in at most } f(n) \text{ steps}\}$.

Claim: If R is a two-counter machine that runs in $o(f(n))$ steps, then $L(R) \neq L$. Proof: Let x be large enough that $R(\langle \llcorner R \lrcorner, x\rangle)$ runs at most $f(n)$ steps. Then $\langle \llcorner R \lrcorner, x\rangle$ is in L if and only if $R(\langle \llcorner R \lrcorner, x\rangle)$ rejects. Either way, $L(R) \neq L$.

Now we have to show that L can be decided by a two-counter machine without the time bound. We could build a universal two-counter machine directly, but this is overkill. Instead, let's take this approach to show we can decide L using a Turing machine:

1. We can translate a representation $\llcorner M \lrcorner$ of a two-counter machine to a representation of $\llcorner M' \lrcorner$ of a Turing machine using two work tapes with a single mark to represent zero and the head position to represent the counter value.

2. We can then use one of our previous constructions to show that we can decide using a Turing machine if M' rejects x in $f(n)$ steps.

This means that there exists a Turing machine that decides L. Since we can simulate Turing machines using two-counter machines (Lemma 3.1.1), there is also a two-counter machine that decides L.

A better solution: Though the preceding works, it's overkill, since we can avoid diagonalizing over two-counter machines entirely.[2]

The Time Hierarchy Theorem says that if $f(n)$ is time-constructible, then there is a language L that cannot be decided by a Turing machine in $o(f(n))$ time but can be decided by a Turing machine eventually.

Since a Turing machine can simulate a two-counter machines with no slowdown (use two work tapes with a single mark on each for the counter), if L can be decided by a two-counter machine in $o(f(n))$ time, it can also be decided by a Turing machine in $o(f(n))$ time, but it can't. So L is not decided by a two-counter machine in $o(f(n))$ time. On the other hand, it is decided by some Turing machine eventually, and a two-counter machine that simulates that Turing machine will also decided it eventually.

B.3 Assignment 3: due Wednesday, 2017-03-01 at 23:00

B.3.1 A balanced diet of hay and needles

Call an oracle A **balanced** if, for each $n \geq 1$, $|\{x \in A \mid |x| = n\}| = 2^{n-1}$.

Show that there exist balanced oracles A and B such that $\mathbf{P}^A = \mathbf{NP}^A$ and $\mathbf{P}^B \neq \mathbf{NP}^B$.

Solution

Rather than build new oracles from scratch, we'll show how to encode any oracle so that it is balanced, and use this to reduce to Baker-Gill-Solovay.

Given an oracle A, define its balanced version $\breve{A} = \{x0 \mid x \in A\} \cup \{x1 \mid x \notin A\}$.

Since every x with $|x| = n$ produces exactly one element of $\breve{A}$ with $|x| = n + 1$, we get exactly 2^n elements of size $n + 1$, making $\breve{A}$ balanced.

Any machine that uses A can be modified to use $\breve{A}$ instead, by writing an extra 0 to the end of each oracle call, thus replacing a class to $A(x)$ with

[2] I would like to thank Aleksandra Zakrzewska for pointing this out in her solution, unlike the rest of us lemmings who blindly went ahead and diagonalized over two-counter machines.

$A(x0)$. In the other direction, a machine that uses $\breve{A}$ can be modified to use A instead, by replacing the response to each call to $\breve{A}(xb)$ with $A(x) \oplus b$. These modifications are easily made to either a **P** or **NP** machine.

Now let A and B be the oracles from the Baker-Gill-Solovay Theorem for which $\mathbf{P}^A = \mathbf{NP}^A$ and $\mathbf{P}^B \neq \mathbf{NP}^B$. Then $\breve{A}$ and $\breve{B}$ are balanced oracles for which $\mathbf{P}^{\breve{A}} = \mathbf{NP}^{\breve{A}}$ and $\mathbf{P}^{\breve{B}} \neq \mathbf{NP}^{\breve{B}}$.

B.3.2 Recurrence

Call a vertex u in a directed graph G **recurrent** if a random walk starting at u eventually returns to u with probability 1. (This is true if and only if u is reachable from any node v that is reachable from u.) Let $L = \{\langle G, u\rangle \mid u \text{ is recurrent in } G\}$. Show that L is **NL**-complete with respect to log-space reductions.

Solution

We need to show $L \in \mathbf{NL}$ and $\forall L' \in \mathbf{NL} L' \leq_{\mathbf{L}} L$.

- L is in **NL**. Proof: We can test u in $\mathbf{L}^{\mathbf{NL}}$ by iterating over all v and checking using an STCON oracle if there is a path from u to v and a path from v to u. If we find the former but not the latter, $\langle G, u\rangle \notin L$.

  This shows $L \in \mathbf{L}^{\mathbf{NL}} \subseteq \mathbf{NL}^{\mathbf{NL}} = \mathbf{NL}$.

- L is **NL**-hard. Proof: We'll show it's **coNL**-hard by log-space reduction from the complement of STCON. Given a graph $G = (V, E)$ and nodes s and t, construct a new graph $G' = (V, E')$ where

$$E' = E \cup \{vs \mid v \in V, v \neq t\} \setminus \{tv \mid v \in V\}.$$

  Let $u = s$.

  Then every $v \neq t$ can reach u in G'. So the only way that u is not recurrent is if $u = s$ can reach t in G', which has no outgoing edges. If there is an s–t path in G, then there is an s–t path in G', since if some G path uses an outgoing edge from t we can truncate to the shortest prefix that reaches t and get an s–t path that doesn't. Conversely, any s–t path in G' gives an s–t path in G by taking the shortest suffix that contains s. So u is recurrent in G' if and only if there is no s–t path in G.

  This makes L **coNL**-hard. But **coNL** = **NL**, so L is also **NL**-hard.

B.4 Assignment 4: due Wednesday, 2017-03-29 at 23:00

B.4.1 Finite-state machines that take advice

A manufacturer of low-budget Turing machines decides that advice is so powerful, that a machine that uses it might still be useful even without any write heads.

Define the class **FSM/poly** to be the set of languages L decided by a Turing machine M with two read-only input tapes, one of which is a two-way tape that contains the input x and one of which is a one-way tape that contains an advice string $\alpha_{|x|}$, where both tape alphabets consist of bits. We say that M decides L if and only if there is a family of advice strings $\{\alpha_n\}$ of size polynomial in n that cause M to always output the correct value when presented with x and α_n.

(The advice tape is one-way, only allowing its head to move to the right, to keep us from using it as a counter. The input tape is two-way because making it one-way would just be cruel.)

Show that $\mathbf{NC}^1 \subseteq \mathbf{FSM/poly} \subseteq \mathbf{AC}^1$.

Solution

We can show that $\mathbf{NC}^1 \subseteq \mathbf{FSM/poly}$ by showing that a machine as defined above can simulate a bounded-width branching program, which can in turn simulate anything in $\mathbf{NC}^1$ using Barrington's Theorem. We make the advice be a sequence of instructions of the form: (a) move the input tape head one cell to the left; (b) move the input tape head one cell to the right; (c) update the state of the branching program according to permutation ξ_0 or ξ_1 depending on the bit in the current input cell; or (d) halt and accept if the state of the branching program is not equal to its initial state. Each of these instructions can be encoded in a constant number of bits,[3] so the finite-state controller can read in an instruction and execute it without needing too many states.

To encode the branching program, we replace each instruction $(i_j, \xi_{0j}, \xi_{1j})$ with a sequence of up to $n-1$ left or right moves to position the head on top of input bit i (this is a fixed sequence, because we just need to adjust the position by the constant offset $i_j - i_{j-1}$) and then execute the instruction for ξ_0, ξ_1. At the end we put in the halt instruction. We can easily show by

[3] At most $\lceil \lg\left((5!)^2 + 3\right) \rceil = 16$, so maybe not something we can run on an 8-bit controller, but all the cool kids got 16-bit controllers decades ago.

induction on j that the simulated state of the branching program inside our controller after j such steps is correct, which means that we get the right answer when we halt.

To show **FSM/poly** $\subseteq$ $\mathbf{AC}^1$, given an **FSM/poly** machine, represent its execution as a path in a graph whose vertices are labeled by (i, t, q) where i is the position of the input head, t is the position of the advice head, and q is the state of the finite-state controller. If $|\alpha_n|$ is bounded by n^c, then there are $O(n^{c+1})$ vertices in this graph.[4]

Presence or absence of edges that depend on the input can be computed directly from individual input bits. There is an edge from (i, t, q) to (i', t', q') if M can make this transition after observing a particular value b at position i on the input tape, which means that we can compute the existence of this edge in our circuit by either taking x_i directly or running it through a NOT gate, depending on b. We also have that M accepts if and only if there is a path from the initial configuration of M to some accepting configuration, which we can solve using STCON.[5] But STCON is in $\mathbf{AC}^1$, so we can detect if M accepts (and thus decide L) using an $\mathbf{AC}^1$ circuit.

B.4.2 Binary comparisons

Suppose that we extend $\mathbf{AC}^0$ by adding **binary comparison gates** . A $2m$-input binary comparison gate takes inputs $x_{m-1}, \ldots, x_0$ and $y_{m-1}, \ldots, y_0$, and returns 1 if and only if $\sum_{i=0}^{m-1} 2^i x_i > \sum_{i=0}^{m-1} 2^i y_i$.

Define the complexity class $\mathbf{ACBC}^0$ to consist of all circuits of polynomial size and constant depth consisting of unbounded fan-in AND, OR, and NOT gates, and polynomial fan-in binary comparison gates.

Prove or disprove: PARITY $\in \mathbf{ACBC}^0$.

[4] There is a small technical issue here, that I will confess to not noticing until Lee Danilek pointed it out. We can't necessarily assuming that the input head position is bounded by n, because in general we allow the input tape to a Turing machine to contain infinite regions of blanks extending past the actual input. We can deal with this without adding any extra vertices by arguing that an **FSM/poly** machine that moves the head off the end of the input executes a sequence of steps that do not depend on the input until the head moves back, and thus we can replace this entire sequence of steps by a single edge to the resulting configuration. Equivalently, we could use essentially the same idea to replace whatever bits of the advice are consumed during the off-input rampage by a single instruction on the advice tape that causes the controller to update its state and input head position appropriately.

[5] We probably need to link all the accepting configurations to a single extra sink vertex to make this work, because a general **FSM/poly** may not be smart enough to park its input head in a standard place if we only allow 0 and 1 input bits.

Solution

We'll show PARITY $\notin \mathbf{ACBC}^0$, by showing that $\mathbf{ACBC}^0 = \mathbf{AC}^0$.

Observe that $x > y$ if and only if there is some position i such that $x_j = y_j$ for all $j > i$, $x_i = 1$, and $y_i = 0$. We can test $x_j = y_j$ using the formula $(x_j \wedge y_j) \vee (\neg x_j \wedge \neg y_j)$, and we can test $x > y$ using the formula $\bigvee_{i=0}^{m-1}(x_i \wedge \neg y_i \wedge \bigwedge j = i + 1^{m-1}(x_j = y_j))$. By replacing each binary comparison gate with a circuit computing this formula, we transform any $\mathbf{ACBC}^0$ circuit into a $\mathbf{AC}^0$ circuit while maintaining polynomial size and constant depth. It follows that $\mathbf{ACBC}^0 = \mathbf{AC}^0$ as claimed. But since PARITY is not in $\mathbf{AC}^0$, it can't be in $\mathbf{ACBC}^0$ either.

B.5 Assignment 5: due Wednesday, 2017-04-12 at 23:00

B.5.1 $\mathbf{BPP}^{\mathbf{BPP}}$

Show that $\mathbf{BPP}^{\mathbf{BPP}} = \mathbf{BPP}$.

Solution

It holds trivially that $\mathbf{BPP} \subseteq \mathbf{BPP}^{\mathbf{BPP}}$, so we only need to show that $\mathbf{P}^{\mathbf{BPP}} \subseteq \mathbf{BPP}$.

Suppose L is in $\mathbf{BPP}^{\mathbf{BPP}}$. Then there is a polynomial-time randomized oracle Turing machine M such that M accepts any x in L with probability at least $2/3$ and accepts any x not in L with probability at most $1/3$, given an oracle that computes some language L' in $\mathbf{BPP}$.

Let M' be a polynomial-time randomized Turing machine that decides L' with probability of error at most $1/3$. The oracle calls always give the right answer, so we can't necessarily drop M' into M without blowing up the error probability for M. But we can make it work using amplification.

Recall that we can reduce the error of M' to ϵ by running M' $\Theta(log(1/\epsilon))$ times and taking the majority. The original machine M makes at most n^c oracle calls for some c. We can replace each such call with a sequence of $\Theta(\log n^{c+1})$ simulations of M', whose majority value will be incorrect with probability at most n^{-c-1}. Taking a union bound, the probability of error over all these simulated oracle calls is at most $1/n$, giving a probability of error for the computation as a whole bounded by $1/3 + 1/n$. This is bounded away from $1/2$ by a constant for sufficiently large n, so we can amplify to

get it under 1/3. Since the entire construction still takes polynomial time, this puts L in **BPP**.

B.5.2 coNP vs RP

Show that if **coNP** $\subseteq$ **RP**, then **NP** = **ZPP**.

Solution

We already have **coRP** $\subseteq$ **NP**, so adding the assumption gives **coRP** $\subseteq$ **coNP** $\subseteq$ **RP**. Take complements to get **RP** $\subseteq$ **coRP**, which gives **RP** = **coRP**. We then have **coRP** = **RP** = **NP**, and **ZPP** = **RP** $\cap$ **coRP** = **NP**.

B.6 Assignment 6: due Wednesday, 2017-04-26 at 23:00

B.6.1 NP $\subseteq$ $\mathbf{P}^{\mathbf{SquareP}}$

A secretive research lab claims that it is possible to use quantum interference to build a machine that decide any language in the class **SquareP**, the set of languages decided by a polynomial-time nondeterministic Turing machine that accepts if the number of accepting paths is a perfect square $(0, 1, 4, 9, \dots)$ and rejects otherwise.

Show that this machine would give a practical procedure for all problems in **NP**, by showing that **NP** $\subseteq$ $\mathbf{P}^{\mathbf{SquareP}}$.

Solution

There are a lot of ways to solve this. The easiest is probably to note that x and $2x$ (more generally, px for any prime p) are both squares if and only if $x = 0$. So if we can tinker with some **NP**-complete problem to increase the number of solutions by a prime factor, we can recognize an instance with no solutions by detecting that the number of solutions x to the original instance and the number of solutions px to the modified instance are both squares. This is enough to put **NP** $\subseteq$ $\mathbf{P}^{\mathbf{SquareP}}$, since two oracle queries is well within the polynomial bound on how many we are allowed to do.

With some additional sneakiness, we can do the same thing with just one oracle query.

Let SQUARE SAT = $\{\phi \mid \phi$ has a square number of satisfying assignments$\}$. It's easy to see that SQUARE SAT is in **SquareP** (build a machine that

guesses each possible assignment and verifies it). It's also the case that a $\mathbf{P}^{\mathbf{SquareP}}$ can do poly-time reductions. So we can show $\mathbf{NP} \subseteq \mathbf{P}^{\mathbf{SquareP}}$ by giving a $\mathbf{P}^{\text{SQUARESAT}}$ machine to solve SAT.

Given a formula ϕ with n variables $x_1, \ldots, x_n$, construct a new formula $\phi' = (z \wedge y_1 \wedge y^2 \wedge \ldots \wedge y_n \wedge \phi) \vee \neg z$, where z and $y_1, \ldots, y_n$ are new variables not appearing in ϕ. Then if ϕ has k satisfying assignments, ϕ' has $k + 2^{2n}$ satisfying assignments, consisting of (a) k assignments satisfying ϕ with z and all y_i true, and (b) 2^{2n} assignments with z false and the remaining $2n$ variables set arbitrarily.

If $k = 0$, then then number of satisfying assignments for ϕ' is $2^{2n} = (2^n)^2$, a square If $k > 0$, then the number of satisfying assignments is $2^{2n} + k \leq 2^{2n} + 2^n < 2^{2n} + 2 \cdot 2^n + 1 = (2^n + 1)^2$. Since this last quantity is the smallest perfect square greater than 2^{2n}, $2^{2n} + k$ is not a square. It follows that our machine can correctly identify whether ϕ is satisfiable by feeding ϕ' to the SQUARE SAT oracle and accepting if and only if the oracle rejects.

B.6.2 NL ⊆ P

Recall that $\mathbf{NL} = \mathbf{FO}(\texttt{TC})$ and $\mathbf{P} = \mathbf{FO}(\texttt{LFP})$. Show that $\mathbf{NL} \subseteq \mathbf{P}$ by giving an explicit algorithm for converting any $\mathbf{FO}(\texttt{TC})$ formula to a $\mathbf{FO}(\texttt{LFP})$ formula.

Solution

For each $\mathbf{FO}(\texttt{TC})$ formula ϕ, let $\hat{\phi}$ be the corresponding $\mathbf{FO}(\texttt{LFP})$ formula, constructed recursively as described below:

1. If ϕ is $P(x)$ or $x < y$, then $\hat{\phi} = \phi$.
2. If ϕ is $\neg\rho$, then $\hat{\phi} = \neg\hat{\rho}$.
3. If ϕ is $\rho \vee \sigma$, then $\hat{\phi} = \hat{\rho} \vee \hat{(}\sigma)$.
4. If ϕ is $\rho \wedge \sigma$, then $\hat{\phi} = \hat{\rho} \wedge \hat{(}\sigma)$.
5. If ϕ is $\texttt{TC}(\rho, x, y)$, then $\hat{\phi}$ is $\texttt{LFP}(\sigma, y)$, where $\sigma(P, z) \equiv (z = x) \vee (\exists q : P(q) \wedge \hat{\rho}(q, z))$.

We claim by induction on formula size that ϕ is true if and only if $\hat{\phi}$ is. Except for the TC implementation, this is immediate from the definition above. For the TC implementation, we wish to argue by induction on i that if $P_0, P_1, \ldots$ is the sequence of relations generated by the LFP operator, then

$P_i(y)$ holds precisely if there is a sequence $x = x_1, x_2, \ldots, x_j = y$ with $j \leq i$ such that $\phi(x_k, x_{k+1})$ holds for each k.

This is clearly true for $i = 0$ (P_0 is empty, and there is no sequence) and $i = 1$ (P_1 contains only x). For larger i, suppose that there is a sequence of length $j \leq i$. If $j < i$, then $P_{i-1}(y)$ holds already, and since `LFP` can only add new elements to P_i, $P_i(y)$ holds as well. If $j = i$, then $P_{i-1}(x_{i-1})$ holds by the induction hypothesis, and the formula make $P_i(y)$ true because $P_{i-1}(x_{i-1})$ and $\hat{\rho}(x_{i-1}, y)$ holds (because $\rho(x_{i-1}, x_i)$ does). In the other direction, if $P_i(y)$, then either $P_{i-1}(y)$ holds, and there is a sequence of length $j \leq i - 1 \leq i$, or $P_{i-1}(y)$ does not hold. In the latter case, either $y = x$ or there exists q such that $P_{i-1}(q)$ holds and $\rho(q, y)$ is true; either way we get a sequence ending in y of length at most i.

B.7 Final Exam

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are three problems on this exam, each worth 20 points, for a total of 60 points. You have approximately three hours to complete this exam.

B.7.1 $\mathbf{L}^A = \mathbf{PH}^A$

Show that there exists an oracle A such that $\mathbf{L}^A = \mathbf{PH}^A$.

Solution

Let A be EXPCOM $= \{\langle M, x, 1^n\rangle \mid M \text{ accepts } x \text{ in at most } 2^n \text{ steps}\}$. Because a machine that runs in log space also runs in poly time, we have $\mathbf{L}^A \subseteq \mathbf{P}^A \subseteq \mathbf{PH}^A$.

For the other direction, we will show that:

1. Any language in $\mathbf{NP}^{\text{EXPCOM}}$ can be decided by a single call to EXPCOM, with an appropriate input $\langle M, x, 1^n\rangle$, and
2. An $\mathbf{L}^{\text{EXPCOM}}$ machine can generate this input.

Let $L \in \mathbf{PH}^{\text{EXPCOM}}$. Then $L \in (\mathbf{\Sigma}_k^p)^{\text{EXPCOM}}$ for some fixed k. This means that L can be decided by an alternating oracle Turing machine M that computes $\exists y_1 \forall y_2 \exists y_3 \ldots Q y_k M'(x, y_1, \ldots, y_k)$ where each y_i has length $p(n)$ for some polynomial p and M' is a machine with access to an EXPCOM oracle that runs in time $q(n)$ for some polynomial n.

Each call M' makes to the oracle has length at most $q(n)$, so it can be simulated in time $2^{O(q(n))}$, and since M' makes at most $q(n)$ many calls, an entire execution of M' can be simulated in time $O\Big(q(n)2^{O(q(n)}\Big) = 2^{O(q(n))}$. There are $2^{kp(n)}$ choices of $y_1, \ldots, y_k$, so enumerating all possible choices and simulating M' on each takes time $2^{O(kp(n)q(n))}$. Let M'' be the machine that does this on input x.

Our $\mathbf{L}^{\mathrm{EXPCOM}}$ can thus decide L for sufficiently large inputs x by making an oracle call with input $\langle M'', x, 1^{n^c}\rangle$ where c is chosen so that 2^{n^c} exceeds the maximum time $2^{O(kp(n)q(n)}$ of M''. For smaller inputs, we can just hard-code the answers into the transition table. This shows $L \in \mathbf{L}^{\mathrm{EXPCOM}}$ and thus $\mathbf{PH}^{\mathrm{EXPCOM}} \subseteq \mathbf{L}^{\mathrm{EXPCOM}}$. It follows that we have an oracle A for which $\mathbf{L}^A = \mathbf{PH}^A$.

B.7.2 A first-order formula for MAJORITY

Suppose you have predicates P and $<$, where $<$ is a total order on the universe. Suppose also that the universe is finite and non-empty.

Find a first-order formula ϕ, using P, $<$, and the usual logical machinery $\forall$, $\exists$, $\neg$, $\wedge$, $\vee$, such that ϕ is true if and only if $P(x)$ is true for at least half of all possible x, or show that no such formula exists.

Solution

There is no such formula. Proof: We know that $\mathbf{AC}^0$ can't compute PARITY. This means $\mathbf{AC}^0$ can't compute MAJORITY either, because we could use n circuits for MAJORITY plus a few extra gates to compute PARITY. But $\mathbf{FO} \subseteq \mathbf{AC}^0$, since we can implement $\forall$ and $\exists$ as (very wide) $\wedge$ and $\vee$ gates, the other logical connectives as gates, instances of $<$ as constants, and instances of P as input wires. So $\mathbf{FO}$ can't compute MAJORITY either.

B.7.3 On the practical hardness of BPP

Show that there is a language L that is (a) contained in $\mathbf{BPP}$; and (b) not decidable in $O(n^{100})$ time using a deterministic Turing machine.

Solution

The Time Hierarchy Theorem says that there exists a language L that can be decided by a deterministic Turing machine in $O(n^{101})$ time but not in $O(n^{100})$ time. This language is in $\mathbf{P} \subseteq \mathbf{BPP}$.

Bibliography

[AB07] Sanjeev Arora and Boaz Barak. Computational complexity: A modern approach. Unpublished draft available at http://theory.cs.princeton.edu/complexity/book.pdf, 2007.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[Adl78] Leonard Adleman. Two theorems on random polynomial time. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 75–83, Washington, DC, USA, 1978. IEEE Computer Society.

[Ajt83] Miklós Ajtai. Σ_1^1-formulae on finite structures. *Annals of pure and applied logic*, 24(1):1–48, 1983.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of mathematics*, pages 781–793, 2004.

[Bab15] László Babai. Graph isomorphism in quasipolynomial time. *CoRR*, abs/1512.03547, 2015.

[Bar89] David A Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc1. *Journal of Computer and System Sciences*, 38(1):150–164, 1989.

[BFL91] László Babai, Lance Fortnow, and Carsten Lund. Non-deterministic exponential time has two-prover interactive protocols. *computational complexity*, 1(1):3–40, 1991.

[BGS75] Theodore Baker, John Gill, and Robert Solovay. Relativizations of the P=?NP question. *SIAM Journal on computing*, 4(4):431–442, 1975.

[BM88] László Babai and Shlomo Moran. Arthur-merlin games: A randomized proof system, and a hierarchy of complexity classes. *Journal of Computer and System Sciences*, 36(2):254 – 276, 1988.

[Can91] Georg Cantor. Über eine elementare Frage der Mannigfaltigkeitslehre. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 1(1):75–78, 1891.

[Coo73] Stephen A Cook. A hierarchy for nondeterministic time complexity. *Journal of Computer and System Sciences*, 7(4):343–353, 1973.

[DF03] Rod Downey and Lance Fortnow. Uniformly hard languages. *Theoretical Computer Science*, 298(2):303–315, 2003.

[Din07] Irit Dinur. The pcp theorem by gap amplification. *Journal of the ACM (JACM)*, 54(3):12, 2007.

[Edw41] Jonathan Edwards. *Sinners in the Hands of an Angry God. A sermon, preached at Enfield, July 8th, 1741. At a Time of great Awakenings; and attended with remarkable Impressions on many of the Hearers.* S. Kneeland and T. Green, Boston, in Queen-Street over against the Prison, 1741.

[FM71] Michael J Fischer and Albert R Meyer. Boolean matrix multiplication and transitive closure. In *Switching and Automata Theory, 1971, 12th Annual Symposium on*, pages 129–131. IEEE, 1971.

[FSS84] Merrick Furst, James B Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Theory of Computing Systems*, 17(1):13–27, 1984.

[GGH+16] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. *SIAM Journal on Computing*, 45(3):882–929, 2016.

[GHR91] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. A compendium of problems complete for p (preliminary). Technical Report 91-11, University of Alberta, December 1991.

[GJ79] Michael R Garey and David S Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on computing*, 18(1):186–208, 1989.

[GS86] Shafi Goldwasser and Michael Sipser. Private coins versus public coins in interactive proof systems. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 59–68. ACM, 1986.

[GW95] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, 1995.

[Gö31] Kurt Gödel. Über formal unentscheidbare Sätze der "Principia Mathematica" und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[Has86] Johan Hastad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 6–20. ACM, 1986.

[Hås01] Johan Håstad. Some optimal inapproximability results. *Journal of the ACM (JACM)*, 48(4):798–859, 2001.

[Hen65] F. C. Hennie. Crossing sequences and off-line Turing machine computations. In *Proceedings of the 6th Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1965)*, FOCS '65, pages 168–172, Washington, DC, USA, 1965. IEEE Computer Society.

[HS66] F. C. Hennie and R. E. Stearns. Two-tape simulation of multitape Turing machines. *Journal of the ACM*, 13(4):533–546, October 1966.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[Kha80] Leonid G Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980.

[Kho02] Subhash Khot. On the power of unique 2-prover 1-round games. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 767–775. ACM, 2002.

[Kho10] Subhash Khot. On the unique games conjecture. In *2010 25th Annual IEEE Conference on Computational Complexity*, pages 99–121, 2010.

[KL80] Richard M Karp and Richard J Lipton. Some connections between nonuniform and uniform complexity classes. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 302–309. ACM, 1980.

[Lad75] Richard E. Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, 22(1):155–171, 1975.

[LOT03] Maciej Liśkiewicz, Mitsunori Ogihara, and Seinosuke Toda. The complexity of counting self-avoiding walks in subgraphs of two-dimensional grids and hypercubes. *Theoretical Computer Science*, 304(1-3):129–156, 2003.

[LP82] Harry R. Lewis and Christos H. Papadimitriou. Symmetric space-bounded computation. *Theoretical Computer Science*, 19(2):161 – 187, 1982.

[Min61] Marvin L. Minsky. Recursive unsolvability of Post's problem of "Tag" and other topics in theory of Turing machines. *Annals of Mathematics*, 74(3):437–455, November 1961.

[MP69] Marvin Minsky and Seymour Papert. *Perceptrons*. MIT press, 1969.

[Raz85] Alexander A Razborov. Lower bounds on the monotone complexity of some boolean functions. *Dokl. Akad. Nauk SSSR*, 281(4):798–801, 1985.

[Raz87] Alexander A Razborov. Lower bounds on the size of bounded depth circuits over a complete basis with logical addition. *Mathematical Notes of the Academy of Sciences of the USSR*, 41(4):333–338, 1987.

[Rei08] Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):17:1–17:24, September 2008.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[RR97] Alexander A Razborov and Steven Rudich. Natural proofs. *Journal of Computer and System Sciences*, 55(1):24–35, 1997.

[RST84] Walter L Ruzzo, Janos Simon, and Martin Tompa. Space-bounded hierarchies and probabilistic computations. *Journal of Computer and System Sciences*, 28(2):216–230, 1984.

[Sha92] Adi Shamir. IP = PSPACE. *Journal of the ACM (JACM)*, 39(4):869–877, 1992.

[Smo87] Roman Smolensky. Algebraic methods in the theory of lower bounds for boolean circuit complexity. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 77–82. ACM, 1987.

[Tod91] Seinosuke Toda. Pp is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 20(5):865–877, 1991.

[Tur37] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[VV86] Leslie G Valiant and Vijay V Vazirani. NP is as easy as detecting unique solutions. *Theoretical Computer Science*, 47:85–93, 1986.

[Wil85] Christopher B. Wilson. Relativized circuit complexity. *Journal of Computer and System Sciences*, 31(2):169–181, 1985.

Index