An Introduction to Computational Complexity

[version December 5, 2011]

Michael Soltys

Computing and Software, McMaster University, 1280 Main Street West, Hamilton, ON., L8S 4K1, Canada
E-mail address: soltys@mcmaster.ca

To my parents.

Contents

Preface

Complexity theory asks what makes certain problems computationally difficult. It also investigates the relationship between computability and provability; indeed, the famous P versus NP question can be seen in this light: if all solutions to the instances of a given problem can be verified efficiently, then can those solutions be also computed efficiently? The quest of complexity is to assess the computational difficulty of a given problem; for example, how hard is it to determine if a graph can be colored with three colors so that no two adjacent nodes are of the same color?

Complexity upper bounds provide an algorithm that solves the problem in the most efficient way—with respect to resources such as time or space or circuit size. The lower bounds claim that the problem cannot be solved within some quantity of those resources. It is not surprising that upper bounds are easier—we are adept at finding quick algorithms—while good lower bounds are usually very difficult to achieve. Simply put, we are not very proficient at showing that *all* algorithms in a given complexity class are not up to a certain task. After all, lower bounds require a lot of ingenuity in order to show that no ingenuity whatsoever can solve a problem within a given bound on resources. Expressed in another way, existential statements seem more amenable to proofs than universal ones.

The intended audience for this book are graduate students in computer science and mathematics who want to quickly familiarize themselves with computational complexity. As such, this book aims more at depth than breadth. There are many excellent comprehensive guides to computational complexity (in particular classics such as [**Pap94**] and [**Sip06**]). Here, on the other hand, the reader will find major results of complexity presented with a minimum of background.

The highlights are as follows. In chapter 1 we present two applications of the crossing sequences method: we show that a single tape Turing machine requires $\Omega(n^2)$ many steps to decide the language of palindromes (which can also be seen as an application of rudimentary Kolmogorov complexity), and that languages decidable with $o(\log \log n)$ space are in fact regular.

In chapter 2 we use the self-reducibility of SAT to show that tally sets cannot be NP-complete unless $\mathsf{P} = \mathsf{NP}$, and we prove the Karp-Lipton theorem (if $\mathsf{NP} \subseteq \mathsf{P/poly}$, then PH collapses to its second level). We also introduce the so called "padding technique," and use it to prove Ladner's theorem (if $\mathsf{P} \neq \mathsf{NP}$, then there are languages in $\mathsf{NP} - \mathsf{P}$ which are not NP-complete).

In chapter 3 we deal with space complexity, and present Savitch's theorem as well as the inductive counting technique, and prove the Immerman-Szelepcsényi theorem

(i.e., NL is closed under complementation). Chapter 3 ends with interactive proof systems and a proof of IP = PSPACE.

In chapter 4 the Hierarchy theorems are presented, and we prove the somewhat technical result stating that "$\mathsf{P}^A \neq \mathsf{NP}^A$ with probability 1" (with respect to a random oracle A). The Polytime Hierarchy is examined in detail, and we prove some properties of Alternating Turing machines. We end chapter 4 with an application of the so called "Bennett's trick": Nepomnjascij's theorem.

In chapter 5 we have Shannon's (easy) lower bound for circuits, and we introduce the probabilistic method in order to give two different proofs showing that PARITY is not in AC^0. We end the chapter with a characterization of nonuniform computation via Turing machines with advice tapes.

In chapter 6 we prove Haken's lower bound for the size of resolution refutations of the pigeonhole principle. We also give a lower bound for resolution based on the idea of interpolation and Razborov's lower bound for monotone circuits computing CLIQUE.

The last chapter, chapter 7, starts with three examples of randomized algorithms, we introduce the notion of amplification, and present Sipser's theorem ($\mathsf{BPP} \subseteq \Sigma_2^p$) and finish with Toda's theorem ($\mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}}$).

In the Appendix (chapter 8) we collect miscellanea: a section with different NP-complete problems; some number theory (mostly background for the Rabin-Miller algorithm—algorithm 7.2); a section on the RSA public key encryption; the Isolation Lemma (used in the proof of Toda's theorem); and a section on Berkowitz's algorithm (an NC^2 algorithm for computing the determinant of a matrix).

Many standard results, not mentioned in the above outline, are sprinkled throughout the text. On the other hand, there are many complexity classes that do not make an appearance. The tapestry of complexity classes is truly overwhelming; *"But the man who sets himself the task of singling out the thread of order from the tapestry will by the decision alone have taken charge of the world".*[1]

This book contains one semester worth of material, that is, it is intended for a ten to twelve week course, that meets for two to three lecture hours a week.

Notation

The symbol Σ will have several (but all standard) meanings. It will denote a finite alphabet of symbols, such as the binary alphabet $\Sigma = \{0, 1\}$, and Σ^* denotes the set of all finite strings over the alphabet Σ (i.e., Kleene's[2] star of Σ). We shall use Σ_i to denote alternations; for example, Σ_i^p denotes the set of relations expressible by a polytime predicate with i alternations of quantifiers in front of it, where the first quantifier is $\exists$ (see §4.3).

$\vec{x} = x_1, x_2, \ldots, x_n$, i.e., a vector of variables, while $\bar{x}$ denotes the same as $\neg x$, i.e., a negation of a Boolean variable. We use $[n]$ to denote the subset of natural numbers ($\mathbb{N}$) consisting of $\{1, 2, \ldots, n\}$. We use ":=" to make definitions.

---

[1] Cormac McCarthy, "Blood Meridian Or the Evening Redness in the West", Vintage Books, first Vintage edition, May 1992, pg. 199.

[2] After Stephen Cole Kleene. Σ^i often denotes the set of all strings of length i over the alphabet Σ, so Σ^* can be defined as $\bigcup_{i=0}^{\infty} \Sigma^i$, where $\Sigma^0 = \{\varepsilon\}$, and where ε is the (unique) string of length 0.

We use $A \subseteq B$ to denote that A is a subset of B, and possibly $A = B$. We use $A \subset B$ to denote that A is a *proper* subset of B, i.e., $A \subset B$ iff $A \subseteq B$ and $A \neq B$. Also, $A - B$ will denote set difference: all those elements in A which are not in B. We use $\wedge, \vee, \neg, \oplus, \rightarrow, \leftrightarrow$ to denote Boolean and, or, not, xor (exclusive or), implies, equivalent, respectively. Sometime we use the more expressive names, $\texttt{And}, \texttt{Or}, \texttt{Not}, \texttt{Xor}$. We use $\texttt{T}, \texttt{F}$ (as well as $1, 0$) to denote the Boolean constants true and false, respectively, and we use $\Gamma \vDash \alpha$ to denote that the set of formulas Γ logically implies the formula α. We sometimes use $t \vDash \alpha$ to say that the truth assignment t satisfies the formula α; thus "$\vDash$" has a dual meaning.

We use the standard *big-Oh notation*, $g(n) \in O(f(n))$ if there exist constants c, n_0 such that for all $n \geq n_0$, $g(n) \leq cf(n)$, and the *little-oh* notation, $g(n) \in o(f(n))$, which denotes that $\lim_{n\to\infty} g(n)/f(n) = 0$. We also say that $g(n) \in \Omega(f(n))$ if there exist constants c, n_0 such that for all $n \geq n_0$, $g(n) \geq cf(n)$. This is so called *strong* definition of Ω; see page 18 for the *weak* definition. Finally, we say that $g(n) \in \Theta(f(n))$ if it is the case that both $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$.

We shall need a little bit of number theory: let $\mathbb{Z}_n := \{0, 1, 2, \ldots, (n-1)\}$, the set of numbers modulo n, and let $\mathbb{Z}_n^*$ denote the subset of $\mathbb{Z}_n$ consisting of those numbers co-prime with n. We shall write $x \equiv_n y$ and $x \equiv y \pmod{n}$ to denote that $n|(x-y)$, i.e., that n divides $(x-y)$. Let $(n)_b$ be the binary representation of the integer n; for example $(5)_b = 101$.

Our logarithm function, $\log x$, is assumed to be in base 2, i.e., $\log x := \log_2 x$. Using standard notation, we let $\ln x := \log_e x$.

Sans-serif fonts will be used to denote complexity classes, for example NP, and SMALL CAPS FONTS will be used to denote languages (i.e., problems), for example MinFormula.

Acknowledgments

First of all, it will be clear to anyone reading this book that it relies heavily on many other books and papers (each chapter ends with a Notes section that points the reader to the appropriate sources). I am greatly indebted to all those authors.

This book came about as the result of the author teaching a graduate complexity course at McMaster University (Hamilton, Canada) in the years 2002–2006; during a visit at the Algorithmics Research Group of the Jagiellonian University (Kraków, Poland) in the summer 2007; teaching an *Ulam Seminar* at the University of Colorado at Boulder in the Fall 2007; and in February 2008, during the IX Escuela de Verano de Ciencias Informáticas, Universidad Nacional de Río Cuarto, Argentina.

I am grateful to all the students who attended these courses, and in particular, I am deeply grateful to the following proof readers: Lech Duraj, Grzegorz Gutowski, Grzegorz Herman (for reading the *entire* book), Jan Jeżabek, Ryan Lortie, Bartosz Walczak, and Craig Wilson. Others, like Nicholas James, have not taken the course, but have read parts of the book and given valuable feedback.

I am very grateful to Prof. Jan Mycielski for comments and corrections, especially for a very careful proofreading of §7.1.2, and to Prof. Hugo Ryckeboer.

I would like to express my special thanks to Prof. Paweł Idziak for his help in publishing this book, and to Grzegorz Gutowski and Dai Tri Man Lê for a careful proof reading of the final manuscript.

Finally, I would like to thank the students that took my complexity course over the last years. In particular:

McMaster University, Canada	`CAS705`	2002, 2003, 2004, 2005 2006, 2008, 2009, 2011
Universidad Nacional de Río Cuarto, Argentina	Graduate course	2008
University of Colorado at Boulder, U.S.A.	Ulam Seminar	2007
Jagiellonian University, Poland	Graduate Seminar	2007

1

Turing machines

1.1. Definition

A *Turing machine*[1] (TM) is a tuple $(Q, \Sigma, \Gamma, \delta)$ where Q is a finite set of *states* (always including the three special states q_{init}, q_{accept} and q_{reject}), Σ is a finite *input alphabet* (and unless it is otherwise specified it is $\{0, 1\}$), Γ is a finite *tape alphabet*, and it is always the case that $\Sigma \subseteq \Gamma$ (it is convenient to have symbols on the tape which are never part of the input),

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{\text{Left}, \text{Right}\}$$

is the *transition function* which says that when the TM is in state q and the head is reading a symbol σ, it changes to a new state q', overwrites σ with a new symbol σ', and then the head moves left or right. Note that neither q' nor σ' have to be different from q and σ, respectively. The definition of a TM can be adapted to suit any occasion: there can be more than one tape, and the tape(s) can be infinite in one or two directions, etc. From a computational point of view, all these models are equivalent.

A *configuration* is a tuple (q, w, u) where $q \in Q$ is a state, and where $w, u \in \Gamma^*$, the cursor is on the last symbol of w, and u is the string to the right of w. A configuration (q, w, u) *yields* (q', w', u') in one step, denoted as $(q, w, u) \xrightarrow{M} (q', w', u')$ if one step of M on (q, w, u) results in (q', w', u'). Analogously, we define $\xrightarrow{M^k}$, yields in k steps, and $\xrightarrow{M^*}$, yields in any number of steps, including zero steps. The initial configuration, C_{init}, is $(q_{\text{init}}, \triangleright, x)$ where q_{init} is the initial state, x is the input, and $\triangleright$ is the left-most tape symbol, which is always there to indicate the left-end of the tape.[2] For a TM with k tapes, a configuration is given by $(q, w_1, u_1, \ldots, w_k, u_k)$.

Given a string w as input, we "turn on" the TM in the initial configuration C_{init}, and the machine moves from configuration to configuration. The computation ends when either the state q_{accept} is entered, in which case we say that the TM *accepts* w, or the state q_{reject} is entered, in which case we say that the TM *rejects* w. It is

---

[1] Alan Turing in 1936 ([**Tur37**]) was the first to use imaginary computers (which came to be known as *Turing machines*) for characterizing the class of algorithmic functions.

[2] Note that $\triangleright$ is not part of the alphabet Σ, but is part of the *tape alphabet* Γ which always contains Σ. The symbol $\triangleright$ is convenient for denoting the left-end of the tape—and it can be moved to the right, when it is useful to ignore some initial segment of the tape; we follow [**Pap94**, definition 2.1], where, besides the left-end of tape symbol $\triangleright$, the blank square $\sqcup$ is also introduced as part of Γ and not part of Σ. The point is that we can define Turing machines in a convenient way, with small alterations at will.

possible for the TM to never enter q_{accept} or q_{reject}, in which case the computation does not halt.

Given a TM M we define $L(M)$ to be the set of strings accepted by M, i.e., $L(M) = \{x | M \text{ accepts } x\}$, or, put another way, $L(M)$ is the set of precisely those strings x for which $(q_{\text{init}}, \triangleright, x)$ yields an accepting configuration.

EXERCISE 1.1. Give a formal definition of $L(M)$ using $\overset{M^k}{\to}$ and one using $\overset{M^*}{\to}$.

Alan Turing showed the existence of a so called *Universal Turing machine* (UTM); a UTM is capable of simulating any TM from its description. A UTM is what we mean by a computer, capable of running any algorithm. The proof is not difficult, but it requires care in defining a consistent way of presenting TMs and inputs.

EXERCISE 1.2. Convince yourself that one can construct a UTM.

There are many definitions of computation other than the one given by Alan Turing; for example the so called *Unlimited Register Ideal Machine* (URIM). A URIM program P is a sequence of commands $\langle c_1, c_2, \ldots, c_n \rangle$, operating on a finite (but arbitrarily large) set of registers $R_1, R_2, \ldots, R_m$ (which contain natural numbers in, say, unary notation), and each command is one of the following three types:

$$R_i \leftarrow 0,$$
$$R_i \leftarrow R_i + 1,$$
$$\texttt{goto } c_i \texttt{ if } R_j = R_k.$$

In the last command, if $R_j \neq R_k$, then the next command on the list is run. If we run out of commands (or are sent to an $i > n$ by a `goto`) then the program terminates. We can always assume that the input is given in R_1 at the beginning, and the output is given in R_m at the end (with 0 being "no" and 1 being "yes," if we are computing a decision problem and a number if we are computing a function). Note that the subscripts i, j, k are part of the instruction, i.e., they are not variable, but rather "hard-coded" into the program.

Yet another model of computation is given by *recursive functions*. A function f ($f(\vec{x})$, where $\vec{x} = x_1, x_2, \ldots, x_n$) is defined by *composition* from $g, h_1, \ldots, h_m$ if $f(\vec{x}) = g(h_1(\vec{x}), \ldots, h_m(\vec{x}))$, and f is defined by *primitive recursion* from g, h if

$$f(\vec{x}, 0) = g(\vec{x}),$$
$$f(\vec{x}, y+1) = h(\vec{x}, y, f(\vec{x}, y)).$$

There are three *initial functions* $Z, S, I_{n,i}$, where Z is the constant function equal to 0, $S(x) = x + 1$ (the successor function), and $I_{n,i}(\vec{x}) = x_i$ (the projection function). We say that a function is *primitive recursive* if it can be obtained from the initial functions by finitely many applications of primitive recursion and composition. Finally, we say that a function f is defined by *minimization* from g if $f(\vec{x}) = \mu y[g(\vec{x}, y) = 0]$, where $\mu y[g(\vec{x}, y) = 0]$ is the smallest b such that $g(\vec{x}, b) = 0$, and a function is *recursive* if it can be obtained from the initial functions by finitely many applications of primitive recursion, composition, and minimization.[3] Note that primitive recursive functions

[3]It turns out that only one application of minimization is sufficient.

are always total, but recursive functions are not necessarily so—there may be an $\vec{x}$ for which there is no b such that $g(\vec{x}, b) = 0$.

EXERCISE 1.3. Convince yourself that TMs, URIMs, and recursive functions, are all equivalent models of computation.

1.2. Basic properties

We define $\mathsf{TIME}(f(n))$ to be the class of languages decidable by TMs running in time $O(f(n))$ (i.e., TMs that take at most $O(f(n))$ many steps, on any input of size n, before making a decision). $\mathsf{SPACE}(f(n))$ requires a little bit more care because we want to make sense of sub-linear space (i.e., less working space than the actual input size; for example, logarithmic space). So for space we assume that we have a read-only input tape on which the input string is presented, and a work tape on which we bound how much tape we are allowed to use. Thus, we say that $\mathsf{SPACE}(f(n))$ is the class of languages decidable by TMs that use at most $O(f(n))$ squares of the work tape.

THEOREM 1.4. Given a k-tape TM M operating within time $f(n)$, we can construct a single-tape TM M' operating within time $O(f(n)^2)$, such that $L(M) = L(M')$.

EXERCISE 1.5. Prove theorem 1.4.

THEOREM 1.6 (Speed-Up). Suppose that a TM decides a language L in time bounded by $f(n)$. For any $\varepsilon > 0$, there exists a TM that decides L in time $f'(n) = \varepsilon \cdot f(n) + n + 2$.

EXERCISE 1.7. Prove theorem 1.6.

EXERCISE 1.8. What does theorem 1.6 say about languages decidable in time $O(n)$? In general, what does this theorem say about the "big-Oh" notation?

EXERCISE 1.9. Suppose that we insist that the tape alphabet be $\{\triangleright, \sqcup, 0, 1\}$. Can the speed-up theorem be still applied?

In a nondeterministic TM the transition function δ becomes a transition relation Δ, so "yields" is now a relation as well. A nondeterministic computation can be viewed as a tree, where the nodes are configurations, and we branch on all the possible choices allowed by Δ. The machine accepts if at least one branch ends in an accepting configuration. Given a TM M, let d be its *degree of nondeterminism*, meaning that d bounds the number of choices of Δ (this number is a constant, independent of the input, and $d = 1$ for deterministic machines). Formally, $d := \max_{q \in Q, \sigma \in \Gamma} |\Delta(q, \sigma)|$.

We define $\mathsf{NTIME}(f(n))$ to be the class of languages decidable by nondeterministic TMs where each branch is of length bounded by $f(n)$. We define $\mathsf{NSPACE}(f(n))$ to be the class of languages decidable by nondeterministic TMs where the space used on the work-tapes is bounded by $f(n)$ and where the length of each branch is bounded by the number of configurations possible with space $f(n)$.

THEOREM 1.10. $\mathsf{NTIME}(f(n)) \subseteq \bigcup_{c>1} \mathsf{TIME}(c^{f(n)})$.

PROOF. Let $L \in \mathsf{NTIME}(f(n))$, so $L = L(M)$ for some nondeterministic TM that runs in time $f(n)$. Let d be the degree of nondeterminism of M. The computation tree has size at most $d^{f(n)}$, and the tree can be traversed (in a breadth-first manner) with a pointer to a location in the tree. This pointer can be encoded with a number in base d of length $f(n)$. The traversal takes time $O(d^{f(n)})$. □

1.3. Crossing sequences

1.3.1. A lower bound for palindromes. If $x = x_1 x_2 \dots x_n \in \Sigma^*$ is a string, then the *reverse* of x, denoted x^R, is just $x_n \dots x_2 x_1$. We say that a string is a *palindrome* if $x = x^R$, and we define the language of palindromes to be the set $L_{\text{pal}} = \{x \in \{0,1\}^* | x = x^R\}$.

THEOREM 1.11. Suppose that M is a one-tape TM that decides the language L_{pal}. Then, M requires $\Omega(n^2)$ many steps.

PROOF. We first define the *i-th crossing sequence of M on x* to be

$$\{(q_1, \sigma_1), (q_2, \sigma_2), \dots, (q_m, \sigma_m)\},$$

and it means that the first time M crosses from square i to square $i+1$, M leaves σ_1 on the i-th square, and finds itself in state q_1, and then the first time it crosses from square $i+1$ to square i, it arrives at square i in state q_2 and encounters σ_2 on the i-th square, etc.

Note that the odd pairs denote crossings left-to-right, and even pairs denote crossings right-to-left, and that $\sigma_{2i} = \sigma_{2i-1}$.

Now consider inputs of the form $x0^n x^R$, where $|x| = n$. Let $T_M(x)$ be the number of steps that M takes to decide (and accept) $x0^n x^R$.

There must be some i, $n < i \leq 2n$, i.e., some square in 0^n, for which the i-th crossing sequence has length $m \leq \frac{T_M(x)}{n}$. The reason is that the sum of the length of the crossing sequences corresponding to each square in 0^n has to be bounded above by $T_M(x)$, and therefore not all squares can have crossing sequences that are "long" ($> \frac{T_M(x)}{n}$). Let S be this "short" ($\leq \frac{T_M(x)}{n}$) crossing sequence. (Note that we assumed in this analysis that the TM never stays put; it always either moves right or left. But this is not a crucial restriction as we can always "speed-up" a TM, combining several moves into one, so that it never stays put. Besides, in our official definition, TMs always move right or left.)

CLAIM 1.12. M, n, i, S describe x uniquely.

PROOF. To see this, we show how to uniquely "extract" x from M, n, i, S. For each $x \in \{0,1\}^n$, we simulate M on $x0^i$. The first time M wants to cross from square i to square $i+1$, we check that it would have done so with the pair (q_1, σ_1), and we immediately reset the state to q_2 and put the head back on square i and continue. The second time M crosses from square i to square $i+1$ we again check that it does so with (q_3, σ_3), and again we immediately reset the state to q_4 and put the head back on square i and continue. If all the odd-numbered crossing pairs are correct

(i.e., correspond to those in S), we know that we have found x. Otherwise, we move on to examine the next x.

Our procedure identifies x correctly: suppose that some $y \neq x$ passed all the tests. Then M would have accepted $y0^n x^R$ which is *not* a palindrome. This would contradict the correctness of M.

We must modify M to return to $\triangleright$ before accepting, to ensure an even-length crossing sequence, but this adds only a linear number of steps to the computation. □

So M, n, i, S describe x uniquely, and they can be encoded with

$$c_M + \log n + \log n + c \cdot m \leq c_M + 2\log n + c \cdot \frac{T_M(x)}{n}$$

many bits. However, for every n, no matter how we encode strings, there must be a string x_0 whose encoding requires at least n many bits (otherwise, we would have a bijection from the set of strings of length n to the set of strings of length $n-1$, which is not possible).

Therefore, we have that

$$n \leq c_M + 2\log n + c \cdot \frac{T_M(x_0)}{n}, \tag{1}$$

which gives us the result. □

Exercise 1.13. Theorem 1.11 says that any single tape TM M deciding L_{pal} requires $\Omega(n^2)$ steps. This means that there exist constants $b, n_0 \in \mathbb{N} - \{0\}$ such that for any $n \geq n_0$, there exists an $x \in \{0,1\}^n$ such that M takes at least $\frac{1}{b}n^2$ many steps to decide x. Can we really make the step from the conclusion of the above proof to the statement of theorem 1.11?

We say that a model of computation is *robust* if it is insensitive to "small" changes in the definition. This is a vague notion, but there are good examples of robust models: TMs with one or several tapes, infinite in one or two directions, etc., all capture the set of recursive languages; if we restrict these TMs to be polytime, they are still insensitive to these modifications, so polytime TMs are a robust model of computation as well.

Let LT be the class of languages decidable in linear time, i.e., in time $O(n)$. Formally, $\mathsf{LT} = \mathsf{TIME}(n)$; we are going to encounter this class again in §4.5. From theorem 1.11 we know that LT is *not* robust because L_{pal} can be decided in $O(n)$ steps on a two-tape TM, while on a single-tape TM it requires $\Omega(n^2)$ steps. However, NLT (nondeterministic linear time) is quite robust, with respect to the number of tapes, as the next lemma shows.

Lemma 1.14. If M_k^{nlt} is a nondeterministic linear time bounded TM with k tapes, then M_k^{nlt} can be simulated by M_2^{nlt}.

Proof. Let us assume that the input to M_2^{nlt} is written on tape 1. M_2^{nlt} starts by guessing the entire computation of M_k^{nlt}, and writing the guess on tape 2. This computation does not contain the tape configurations of M_k^{nlt}, but rather for each step of M_k^{nlt} it writes down the (supposed) state and symbols scanned by each of the

k heads. Note that this information (if correct) is enough to determine the next move of M_k^{nlt}.

Now M_2^{nlt} checks that the computation it has written is correct, as follows. First it checks that the state sequence is consistent, by making one pass on tape 2, and checking that at each step the next state is what it should be, given the information from the previous step. Next, for each tape i of M_k^{nlt}, it makes a pass on tape 2 and checks that the scanned symbol information it has written for tape i is consistent. It does this by using tape 1 to simulate tape i, and using the information it has written on tape 2 concerning what the other $k-1$ tapes are scanning.

If all these consistency checks are passed, then the information on tape 2 is correct. Hence M_2^{nlt} can figure out what M_k^{nlt} did. □

EXERCISE 1.15. In the above proof, show formally by induction that the t-th step is simulated correctly.

EXERCISE 1.16. Show that L_{pal} is in $\mathsf{TIME}(n) \cap \mathsf{SPACE}(\log n)$.

1.3.2. Little space is no space at all. A language is *regular* if it can be decided by a Deterministic Finite Automaton (DFA). A DFA is just a TM with no tapes besides the input tape, where it scans the input from left-to-right, changing states as it reads the input, but not writing anything—and in particular never turning back, and it enters an accepting or rejecting state immediately after it scanned the last symbol of the input. See [**Sip06**] for the background on DFAs.

In this section we show that if a language can be decided with $o(\log\log n)$ space,[4] then the language is in fact regular.

THEOREM 1.17. If a language L can be decided by a TM M such that M has a read-only input tape, and a work-tape where space is bounded by a function in $o(\log\log n)$, then L is regular.

The next exercise shows that $\log\log n$ is an exact threshold.

EXERCISE 1.18. Consider the language L over $\{0, 1, \#\}$ given by

$$L = \{\#b_k(0)\#b_k(1)\#b_k(2)\#\ldots\#b_k(2^k-1)\#|k \geq 0\},$$

where $b_k(i)$ is the k-bit binary representation of $i \leq 2^k - 1$. Show that this language is (i) not regular, and (ii) decidable in space $O(\log\log n)$.

We show first that if a language is decidable by a TM with work-tape space bounded by a constant, then the language is regular. Next, we show that if a language is not regular, then the work-tape requires $\Omega_{\text{weak}}(\log\log n)$ space. Note that, as the notation indicates, Ω_{weak} is the "weak" version of Ω, which means that for some constant c, $c\log\log n$ is a lower bound for *infinitely many* n's. On the other hand, the standard version of Ω would assure a lower bound for *all* n (sufficiently big), rather than infinitely many. Confer with exercise 1.13.

We assume that our TMs have a read-only input tape, and a work-tape that is read `&` write, and we bound the space on the work-tape.

[4]Recall that $o(f(n))$ is "little-oh" of $f(n)$, and $g(n) \in o(f(n))$ if $\lim_{n\to\infty} g(n)/f(n) = 0$.

CLAIM 1.19. If L can be decided in space bounded by a constant (i.e., there is a constant k bounding the number of work-tape-squares that the TM is allowed to visit during any computation), then L is in fact regular.

PROOF. Note that we can get rid of constant space by keeping a record of all the possible $k \cdot |\Gamma|^k$ configurations (constantly many) by encoding them as states: there are k positions for the work-tape head, and the size of the alphabet is $|\Gamma|$, so $|\Gamma|^k$ are the possible contents of the work-tape. But this is not enough, because the head on the input-tape may move back-and-forth, so we cannot conclude directly that the machine can be simulated by a finite automaton.

We need to show that the machine can be transformed so that the head on the input-tape moves only from left-to-right. This can be accomplished as follows: for any string $s \in \Gamma^*$, define two functions $f_s, g_s : Q \longrightarrow Q$, where $f_s(q_1) = q_2$ if when the machine is started on s in state q_1, with the head on the first symbol of s, then the first time the machine leaves s (by moving from the last symbol of s to the empty square $\sqcup$) or halts, it does so in state q_2. Let g_s be defined similarly, but instead of the head starting on the first symbol of s, it starts on the last symbol of s. Note that there are $|Q|^{|Q|}$ many functions from Q to Q, so they can all be hard-wired into the machine. We can now make the machine move only from left-to-right, and accept or reject after reading the last symbol of s, by doing the following: when we leave the i-th symbol of s, we know the values of $f_{s_1 s_2 \ldots s_i}$ and $g_{s_1 s_2 \ldots s_i}$. We want to compute $f_{s_1 s_2 \ldots s_{i+1}}$ and $g_{s_1 s_2 \ldots s_{i+1}}$. This is easy using the transition function of the original machine.

It is crucial that the two functions $f_{s_1 s_2 \ldots s_{i+1}}$ and $g_{s_1 s_2 \ldots s_{i+1}}$ depend only on the following: (i) the TM M, (ii) the two functions $f_{s_1 s_2 \ldots s_i}$ and $g_{s_1 s_2 \ldots s_i}$, and (iii) the single bit s_{i+1}. In particular, the two new functions *do not depend* on remembering the string $s_1 s_2 \ldots s_i$, and this is very fortunate as arbitrarily long strings cannot be stored by a finite automaton.

At the end, when we have scanned the entire input string, we have $f_{s_1 s_2 \ldots s_n}$. We check that $f_{s_1 s_2 \ldots s_n}(q_0) = q_{\text{accept}}$, and accept iff that is the case.[5] □

EXERCISE 1.20. Explain precisely how to compute $f_{s_1 s_2 \ldots s_{i+1}}$ and $g_{s_1 s_2 \ldots s_{i+1}}$.

CLAIM 1.21. If a language L is not regular, then it requires $\Omega_{\text{weak}}(\log \log n)$ space. In other words, for any TM deciding L, there exists a constant c so that for *infinitely* many n's, there exists an input x of length n on which the machine will take at least $c \cdot \log \log n$ many steps.

PROOF. Suppose that L is not regular. Then, by claim 1.19, we know that it cannot be decided in constant space. Let M be any machine deciding L.

We want to show that there exists an infinite sequence $\{n_i\}$ (and a constant c which depends only on M) such that $\forall n_i$, $\exists x_i$ such that $|x_i| = n_i$, and M requires at least $c \cdot \log \log n_i$ space to decide x_i.

To accomplish this, we use the fact that for any k we choose, we can always find an input x, such that M requires more than k squares of space to decide x. Pick a k_1

[5] The original proof showing that a "one-way-automaton" can simulate a "two-way-automaton" can be found in [**She59**].

(e.g., $k_1 = 100$—it does not matter what it is) and find an x_1 of minimal length n_1, such that M requires $s_1 \geq k_1$ space to decide x_1. We show now that $c \cdot \log\log n_1 \leq s_1$; the c will be fixed, and its value will transpire later in the proof.

For each $j = 1, 2, \ldots, (n_1 - 1)$, let S_j be the sequence of configurations that M is in whenever its head on the input tape crosses from square j to square $j + 1$ or vice-versa. We can think of S_j as a sequence of "snapshots" of M taken at the time when the head on the input tape crosses between squares j and $j+1$. Thus S_j is the crossing sequence associated with square j, but S_j contains more information than the crossing sequence S defined in theorem 1.11: it contains the state, the contents of the entire work-tape (to be more precise, it contains all the squares of the work-tape from the left-most square all the way to the right-most square that was written on up to that point of the computation), and the position of the head on the work-tape.

There are at most

$$N = |Q| \cdot s_1 \cdot (|\Gamma|^1 + |\Gamma|^2 + \cdots + |\Gamma|^{s_1}) \leq |Q| \cdot s_1 \cdot |\Gamma|^{s_1+1},$$

possible snapshots, and there are

$$N^1 + N^2 + \cdots + N^m < (N^{m+1} - 1)/(N - 1)$$

possible crossing sequences of length at most m.

Since x_1 was chosen to be of minimal length, no two crossing sequences on x_1 are equal (for otherwise, the portion of the input between them could be eliminated, obtaining a new shorter input that still requires s_1 squares of space). To see this, suppose that the computation on the middle portion went all the way up to some square r; then the second crossing sequence, S', would have a record of at least the first r squares in some snapshots. But if $S = S'$, where S' is the first crossing sequence, then S' would have at least the first r squares in the same snapshots. This means that we do not shorten the required tape space by removing the middle portion.

So we know that $(n_1 - 1) \leq (N^{m+1} - 1)/(N - 1)$, and so $n_1 \leq N^{m+1}$. On the other hand, $m \leq N$ because otherwise we would have a loop, and M is a decider. Thus, $n_1 \leq N^{N+1}$, i.e.,

$$n_1 \leq (|Q| \cdot s_1 \cdot |\Gamma|^{s_1+1})^{(|Q| \cdot s_1 \cdot |\Gamma|^{s_1+1}+1)}.$$

By taking log of both sides twice we get what we want; note that the constant c arises from $|Q|, |\Gamma|$, and so it depends only on M and not the size of the input.

We now pick k_2 sufficiently large so that there exists an x_2 of minimal length $n_2 > n_1$ such that M requires $s_2 \geq k_2$ space to decide x_2. Since L is not regular, we know that no matter how large k_2 is, we can always find an x_2 that requires at least k_2 space. The only problem is how to ensure that $n_2 > n_1$? As we are always picking an x_2 of minimal length (this is necessary for the argument to work) we might end up with $n_2 = n_1$. But there are finitely many x_2's of length n_1 (i.e., 2^{n_1}), so to make sure that this does not happen, we take a k_2 larger than the required space of any input of length n_1.

This procedure can be repeated *ad infinitum* to obtain $\{n_i\}$. □

If a language L can be decided in $o(\log\log n)$ space, then it is *not* the case that it requires $\Omega_{\text{weak}}(\log\log n)$ space, and so by the contrapositive of claim 1.21 it follows that it must be regular. This proves theorem 1.17. Note that this proof is not constructive, in the sense that we do not obtain a finite automaton from the $o(\log\log n)$-space bounded TM.

1.4. Answers to selected exercises

Exercise 1.1. $\{x \in \Sigma^* | \exists k \in \mathbb{N}, \exists u, v \in \Gamma^* \text{ such that } (q_{\text{init}}, \triangleright, x) \overset{M^k}{\to} (q_{\text{accept}}, u, v)\}$ and $\{x \in \Sigma^* | \exists u, v \in \Gamma^* \text{ such that } (q_{\text{init}}, \triangleright, x) \overset{M^*}{\to} (q_{\text{accept}}, u, v)\}$.

Exercise 1.5. Concatenate all the k tapes into one tape, and simulate one move of M with two passes of the entire tape of M' (the first pass of M' to determine what is underneath the heads on the tapes of M, and the second pass to make the necessary changes). Note that M uses at most $f(n)$ squares of each of its tapes. This is an observation that we make all the time: in t many steps, a TM can write on at most the first t squares.

Exercise 1.7. The idea is to "increase the word size." Introduce new symbols which encode several symbols of M.

Exercise 1.8. It only says that if a language can be decided in time $O(n)$, then it can be decided in time $(1+\varepsilon)n+2$, for any $\varepsilon > 0$, and hence in a time that is almost strictly linear. That is, the constant can be made arbitrarily close to 1. As far as "big-Oh" notation, it is a way of justifying it, as the theorem says that constants are not important.

Exercise 1.9. At least not with the proof given, since it depends crucially on increasing the word size.

Exercise 1.13. We showed that there exists a constant c so that for every n there exists an $x_0 \in \{0,1\}^n$, so that M takes at least $c \cdot n^2$ many steps to decide $x_0 0^n x_0^R$. So technically what we showed is that among inputs of length $3 \cdot n$ (where n is sufficiently big) there exists at least one input on which M takes time $c \cdot n^2$; or, equivalently, given inputs of length n, where n is sufficiently big *and* divisible by 3, there exists at least one such input on which M takes $c \cdot \left(\frac{n}{3}\right)^2 = \left(\frac{c}{9}\right) \cdot n^2$ steps. So, what we really showed is that L_{pal} requires $\Omega_{\text{weak}}(n^2)$ steps to be decided on a single tape TM—see §1.3.2 for a definition of Ω_{weak}. So in the statement of theorem 1.11 we should replace $\Omega(n^2)$ by $\Omega_{\text{weak}}(n^2)$.

Exercise 1.16. L_{pal} is in linear-time because we can copy the input string w to the second tape, move the second head to the end, and then move the two heads simultaneously towards each other, comparing symbols. It is in logspace because we can work in $|w|$ many stages, in stage i we compare w_i to $w_{|w|+1-i}$. We keep track of the stages with one counter ($O(\log|w|)$ many bits, and in each stage compute the positions i and $|w|+1-i$ with a second counter of the same size.

Exercise 1.18. Based on exercise 4, homework 2, in [**Koz06**]; the solution can be found on page 324 of the same book.

Exercise 1.20. Suppose we already have $g_{s_1 s_2 \dots s_{i+1}}$. Then,

$$f_{s_1 s_2 \dots s_{i+1}}(q) = g_{s_1 s_2 \dots s_{i+1}}(f_{s_1 s_2 \dots s_i}(q)).$$

We obtain $g_{s_1 s_2 \dots s_{i+1}}(q)$ as follows: if the head (when started on s_{i+1}) moves right, we are done—we simply take $g_{s_1 s_2 \dots s_{i+1}}(q)$ to be the new state of the machine. But if the head moves left and enters a state q', we must run $g_{s_1 s_2 \dots s_i}(q')$, and repeat. How can we ensure that this procedure ends? That is, how can we ensure that the head will eventually move right? In the proof of claim 1.19 we are transforming a machine M into a new machine M' where the head moves from left-to-right only. In order to ensure that the transformation works properly M must scan the *entire* input before accepting—this forces the head to move right eventually. (Of course, we can force M to scan the entire input by insisting that it goes all the way to the end of the string before accepting.)

1.5. Notes

Exercise 1.3 is based on the presentation of URIM machines in [**Coo08**]. For a detailed proof of theorem 1.4 see [**Pap94**, theorem 2.1], for theorem 1.6 see [**Pap94**, theorem 2.2], and for theorem 1.10 see [**Pap94**, theorem 2.6]. §1.3.1 is based on [**Pap94**, exercise 2.8.5], and it is also presented in [**Koz06**].

An interesting question is what is the smallest possible number of states and symbols necessary to be able to define a UTM. Minsky showed in [**Min62**] that a UTM can be constructed with seven states and a tape alphabet of four symbols. A recent breakthrough seems to suggest that we can get away with two states and three symbols (see the Wolfram Blog on Alex Smith). The *Busy beaver* function $\Sigma(n)$ is related to this question. The value of $\Sigma(n)$ is the largest number of 1s that a TM that halts may print on its tape, when started on the empty tape, having an alphabet of two symbols, $\{0, 1\}$, and using n states. $\Sigma(n)$ is not computable.

Following the historical remarks in [**BM77**, §17], we point out that imaginary computers in the style of URIMs were invented by several people independently (Shepherdson and Sturgis, Lambek, Minsky) in the late 1950s. The first definition of the class of recursive functions was given by Gödel in 1934 following a suggestion by Herbrand. This definition was proven by Kleene in 1936 to be equivalent to the definition which we are presenting here.

An even more ambitious problem than exercise 1.3 would be to show that the Turing machine definition of recursion is equivalent to the ZFC (Zermelo-Fraenkel set theory with the Axiom of Choice) definition of recursion. This definition is given as follows: take the language $\mathcal{L} = \{\in, =\}$, i.e., the language consisting of two binary predicates. We say that the set $S \subseteq \mathbb{N}$ is recursive if there exists a first order formula α over the language $\mathcal{L}$, such that α has a single free variable x and if $m \in S$ then $\text{ZFC} \vdash \alpha(\bar{m})$, and if $m \notin S$ then $\text{ZFC} \vdash \neg\alpha(\bar{m})$. Here $\bar{m}$ is the representation of the ordinal m in the language $\mathcal{L}$. Of course, it would be easier to do this in PA

(Peano Arithmetic)—which is already sufficient for the definition of recursion, and in fact more convenient as it can quite naturally simulate recursive functions. All these different, but equivalent, definitions of recursion should be convincing evidence that we have captured well the notion of "computable."

The invention of palindromes is generally attributed to Sotades the Obscence of Maronea, who lived in the third century BC in Greek-dominated Egypt. Surprisingly, palindromes appear not just in witty word games (such as `madamimadam` in James Joyce's *Ulysses*, or the title of the famous NOVA program, *A Man, a Plan, a Canal, Panama*), but also in the structure of the male defining chromosome. Other human chromosome pairs fight damaging mutations by swapping genes, but because the Y chromosome lacks a partner, genome biologists have previously estimated that its genetic cargo was about to dwindle away in perhaps as little as five million years. However, researchers on the sequencing team discovered that the chromosome fights withering with palindromes. About six million of its fifty million DNA letters form palindromic sequences—sequences that read the same forward as backward on the two strands of the double helix. These copies provide backups in case of a bad mutation. (These observations come from [**Liv05**].)

2

P and NP

2.1. Introduction

The $\mathsf{P} =? \mathsf{NP}$ question is a fundamental open problem of theoretical computer science, and indeed of all of mathematics.[1] It asks whether all problems solvable in polytime on a nondeterministic TM can also be solved in polytime on a deterministic TM, or, more concretely, whether hard problems such as Boolean satisfiability have efficient algorithms. Stephen Cook ([**Coo71**]) and Leonid Levin formulated the problem independently in 1971.

Standard complexity textbooks provide ample material on P and NP; we limit ourselves to a few observations. The class P consists of those languages that can be decided in polynomial time (*polytime*) in the length of the input, and the class NP consists of those languages that can be decided in polytime on a nondeterministic TM, i.e.,

$$\mathsf{P} = \bigcup_{k=1}^{\infty} \mathsf{TIME}(n^k) \qquad \mathsf{NP} = \bigcup_{k=1}^{\infty} \mathsf{NTIME}(n^k).$$

Sometimes it is convenient to use an alternative definition of NP, in terms of proof systems. We say that a language L has a *proof system* if there exists a polytime binary predicate $R(x, y)$, such that $x \in L \iff \exists y R(x, y)$ (here the "y" is called a *proof* or *certificate*).

In order to give an example of a language with a proof system, we define *Boolean formulas* inductively as follows: a variable p and constants `T` (true) and `F` (false) are formulas, and if α, β are formulas, then so are: (α), $(\alpha \vee \beta)$, $(\alpha \wedge \beta)$, $\neg\alpha$. We sometimes omit parenthesis for better readability.

Consider now the language

$$\textsc{Taut} = \{\langle\phi\rangle | \phi \text{ is a tautology}\},$$

where ϕ is a Boolean formula, and $\langle\phi\rangle$ is some reasonable encoding of ϕ as a string (over $\{0, 1\}$). TAUT is a language with a proof system because we can define it as follows: $\textsc{Taut} = \{\langle\phi\rangle | \exists y R(\langle\phi\rangle, y)\}$ where $R(\langle\phi\rangle, y) :=$"$y$ encodes a derivation of ϕ." This derivation could be, for example, a truth table. The predicate $R(\langle\phi\rangle, y)$ checks that y is indeed the encoding of a valid derivation of ϕ. What is important is that

[1] In the year 2000, The Clay Mathematics Institute selected seven Millennium Prize Problems to mark the 100th anniversary of David Hilbert's lecture at the second International Congress of Mathematicians. One of these problems is the $\mathsf{P} =? \mathsf{NP}$ question; see [**Coo00**] for Cook's recent manuscript prepared for the Clay Mathematics Institute. A nice introduction and history of the seven problems is given in [**Dev05**].

given a formula ϕ and an alleged derivation, it is possible to check in polytime that the derivation is indeed a correct derivation of ϕ.

A proof system is *polybounded* (polynomially bounded) if $|y|$ can be bounded by some polynomial in $|x|$. The prototypical example of a language with a polybounded proof system is

$$\text{SAT} = \{\langle\phi\rangle | \phi \text{ is satisfiable}\}.$$

Here the y could simply be the encoding of a truth assignment; a trivial encoding would work: if $y = a_1 a_2 \dots a_n \in \{0,1\}^*$, then variable x_i would be assigned the truth value $\mathtt{T}$ if $a_i = 1$, and $\mathtt{F}$ if $a_i = 0$. Note that the certificate y is short in this case (in fact, of length bounded by $|\langle\phi\rangle|$).

The class NP consists of languages with polybounded proof systems, such as SAT. In lemma 2.2 below we show that the two definitions of NP are equivalent.

Let $\mathsf{co\text{-}NP}$ be the class of languages whose complements are in NP, i.e., $\mathsf{co\text{-}NP} = \{L | \overline{L} \in \mathsf{NP}\}$. Let UNSAT be the language of unsatisfiable Boolean formulas. Then $\text{UNSAT} = \overline{\text{SAT}}$ (see footnote[2]), and so $\text{UNSAT} \in \mathsf{co\text{-}NP}$, and by the same reasoning $\text{TAUT} \in \mathsf{co\text{-}NP}$.

The question $\mathsf{NP} =? \mathsf{co\text{-}NP}$ can be seen as asking whether TAUT has a polybounded proof system (we shall study this question in more detail in chapter 6).

EXERCISE 2.1. Show that if $L \in \mathsf{NP}$ and $L' \in \mathsf{P}$, then $L \cap L' \in \mathsf{NP}$.

LEMMA 2.2. A language L is in NP iff there exists a polytime binary predicate R, and a polynomial p, such that $x \in L \iff (\exists y \le p(|x|))R(x,y)$. In short, a language is in NP iff it has a polybounded proof system.[3]

PROOF. [$\Longrightarrow$] If $L \in \mathsf{NP}$, then there exists a nondeterministic polytime TM M deciding L. Let $R(x,y)$ be the predicate that checks whether y is (an encoding of) an accepting computation of M on x. [$\Longleftarrow$] Let M be a nondeterministic polytime TM which on input x "guesses" a y, and checks $R(x,y)$. When we say "guesses," we mean that the machine examines all the y's, each y on a different branch of the computation. □

LEMMA 2.3. Suppose that the language L has a polybounded proof system V, and suppose that $\mathsf{P} = \mathsf{NP}$. Then there exists a polytime function f, such that for every $x \in L$, $V(x, f(x))$ holds, and for $x \notin L$, $f(x) =$ "no." In other words, if $x \in L$, then $f(x)$ outputs in polytime (in $|x|$) a proof of membership, and if $x \notin L$, then f says so.

PROOF. Let "$\cdot$" denote the *concatenation* of strings, that is, given two strings $x, y \in \Sigma^*$, $x = x_1 x_2 \dots x_n$ and $y = y_1 y_2 \dots y_m$, $x \cdot y = x_1 x_2 \dots x_n y_1 y_2 \dots y_m$. In practice we often omit the dot and write xy instead of $x \cdot y$.

---

[2]This is not quite true, since $\overline{\text{SAT}}$, understood to be $\{0,1\}^* - \text{SAT}$, contains encodings of unsatisfiable formulas as well as "junk" strings, i.e., strings that do not encode a formula. Let WFF (well-formed formula) be the set of strings that encode formulas. When we talk of the complement of SAT we really mean $\overline{\text{SAT}} \cap \mathsf{WFF}$. Clearly, it can be checked in polytime whether $\langle\phi\rangle \in \mathsf{WFF}$. (See exercise 2.1.)

[3]$(\exists y \le p(|x|))R(x,y)$ denotes $\exists y(|y| \le p(|x|) \wedge R(x,y))$, and $(\forall y \le p(|x|))R(x,y)$ denotes $\forall y(|y| \le p(|x|) \to R(x,y))$.

Let Q be a binary predicate such that

$$Q(x, y) := (\exists z \leq (p(|x|) - |y|))V(x, y \cdot z). \tag{2}$$

Note that the language $\{\langle x, y\rangle | Q(x, y)\}$ is in NP, and so by assumption it is in P. The algorithm computing $f(x)$ (recall that ε denotes the empty string) is given below.

ALGORITHM 2.4 (Computing Certificates).
On input x:

1. $p := \varepsilon$
2. if $\neg Q(x, p)$ then return "no"
3. while $(\neg V(x, p))$
4. if $Q(x, p \cdot 0)$ then $p := p \cdot 0$
5. else $p := p \cdot 1$
6. return p

The algorithm tries out consecutive bits of the proof and uses the decider for Q to check if it is following the right path. As a single check can be done in polynomial time and the length of the proof is polynomial in the length of x, f can be computed in polytime. □

EXERCISE 2.5. Is the "$-|y|$" in the right-hand side of the definition given by (2) really necessary?

2.2. Reductions and completeness

In the context of NP, it is customary to use *polytime many-one reductions*, denoted $\leq^m_{\mathsf{P}}$. But standard NP-hardness reductions can be usually carried out in L (logspace), so instead we use *logspace many-one reductions*,[4] denoted $\leq^m_{\mathsf{L}}$. Thus, when we write "$\leq$" we mean "$\leq^m_{\mathsf{L}}$". Formally, $L_1 \leq L_2$ iff there exists a logspace function f such that $x \in L_1 \iff f(x) \in L_2$. A TM that computes a function—rather than decides a language—is often called a *transducer*. A transducer has a read-only input tape, a work-tape, and a write-only output tape; a tape is write-only if its head moves only from left to right.

EXERCISE 2.6. Show that logspace reductions are transitive, i.e., if $A \leq B$ and $B \leq C$, then $A \leq C$. Note that this requires a precise definition of what it means for a function to be computed in logspace (see the first paragraph of §1.2).

A language L is *$\mathcal{C}$-hard*, for some complexity class $\mathcal{C}$, if for every language $L' \in \mathcal{C}$ it is the case that $L' \leq L$. A language is *$\mathcal{C}$-complete* if it is $\mathcal{C}$-hard, and also in $\mathcal{C}$.

The next theorem introduces the notion of circuits which we are going to cover in chapter 5. To refresh the definition of circuits see §5.1; a magnificent introduction to Boolean circuit theory is given in [**Weg87**]. The language CIRCUITVALUE, defined as the set of pairs $\langle C, x\rangle$ such that x is an input that satisfies the circuit C, is complete for the class P.

[4] Logspace reductions make more sense in the context of P, since if we allow polytime reductions we can "hide" all the computation in the reduction, in effect making every P language (except $\emptyset$ and Σ^*) P-complete.

THEOREM 2.7. CIRCUITVALUE is P-complete and remains so with the following two restrictions: all the gates are `Or` and `And` (i.e., C is a *monotone circuit*), and all the gates are arranged in alternating layers of `And` and `Or` gates.

PROOF. Suppose that $L \in \mathsf{P}$, and so L is decided by a TM in time n^k. For all inputs w, $|w| = n$, we can build an $n^k \times n^k$ *computation tableau*, which represents the history of the computation of the machine on a given input of length n. The first row of the tableau is just the initial configuration, and then each row follows from the previous by a transition. Since a computation need not take exactly n^k many steps, but only at most these many, we make the convention that once a halting configuration is reached, it is repeated to fill exactly n^k many rows of the tableau.

It should be clear that for a given n, we can construct a circuit which on input w ($|w| = n$) computes each entry of the tableau. For example, one way to do this is to represent each entry of the tableau with an array of $|Q| + |\Gamma|$ many gates (where Q is the set of states of the machine, and Γ is the tape alphabet), representing states and tape alphabet symbols with Boolean variables set to 0 or 1. Specifically, entry (i, j) in the tableau has output gates g_{ijs}, for each $s \in Q \cup \Gamma$. Only one of these output gates is on (i.e., set to 1), expressing that the corresponding symbol is the one occupying this entry.

This circuit implements the machine's transition function to compute each row of the tableau correctly for the given input w, and at the end, the circuit outputs 1 if the last row of the tableau represents an accepting configuration, and 0 otherwise.[5] Consider the following two consecutive rows, after the head moves right:

$$\begin{array}{cccc|ccc|cccc} \ldots & u_1 & u_2 & u_3 & q & a & v_1 & v_2 & v_3 & \ldots \\ \ldots & u_1 & u_2 & u_3 & b & p & v_1 & v_2 & v_3 & \ldots \end{array}$$

and the following two consecutive rows, after the head moves left:

$$\begin{array}{cccc|ccc|cccc} \ldots & u_1 & u_2 & u_3 & q & a & v_1 & v_2 & v_3 & \ldots \\ \ldots & u_1 & u_2 & q & u_3 & b & v_1 & v_2 & v_3 & \ldots \end{array}$$

Except in the neighborhood of the state q, the other entries remain unchanged. Thus,

$$g_{(i+1)js} = g_{ijs} \wedge \neg \bigvee_{q \in Q, l = j-1, j, j+1} g_{i(j-1)q}$$

expressing that an entry remains unchanged, if the entries right above, to the right and left, do not contain a state symbol. On the other hand, if they do contain a state symbol, they are modified according to the transition function. It is clear that this circuit is extremely uniform; each consecutive pair of rows has the same connections, dictated by the proximity of the head and the transition function.

By de Morgan laws, all the negations in such a circuit can be pushed to the input level, replicating gates on the way down if necessary, and then change the input to consist of two copies of w, where the second copy of w is inverted, i.e., every 1 becomes a 0, and every 0 becomes a 1. Connect the gates that require inputs to w and to inverted w as appropriate.

For the second restriction see lemma 5.6. □

[5] It is not surprising that a computation can be efficiently simulated with circuits; after all, computers are *built* from circuits.

EXERCISE 2.8. Give explicit definitions for all the gates in theorem 2.7, and in particular, define the connections for the gates expressing the transition function of the TM.

THEOREM 2.9. If B is NP-complete, and $B \in \mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$.

EXERCISE 2.10. Prove theorem 2.9.

THEOREM 2.11. If B is NP-complete, and $B \leq C \in \mathsf{NP}$, then C is NP-complete.

EXERCISE 2.12. Prove theorem 2.11.

THEOREM 2.13 (Cook-Levin). SAT is NP-complete.

PROOF. If $A \in \mathsf{NP}$, then $A = L(N)$, where N is a nondeterministic TM that runs in time n^k. Each configuration of N can be described with a string of length n^k (as in n^k-many steps the machine cannot write on more than n^k-many squares of the tape). Any branch of the computation has length at most n^k. So any branch can be described with an $n^k \times n^k$ tableau (same idea as in the proof of theorem 2.7). To determine whether N accepts w, we must determine if an accepting tableau exists.

We construct a reduction $f(\langle w \rangle) = \langle \phi_w \rangle$ such that $w \in A \iff \phi_w$ is satisfiable. The variables are x_{ijs}, where x_{ijs} is true if position (i, j) in the tableau contains the symbol $s \in Q \cup \Gamma \cup \{\#\}$. Then, let ϕ_w be

$$\phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}},$$

where the first formula, ϕ_{cell}, ensures that exactly one symbol is assigned to each cell, the second formula, ϕ_{start}, ensures that the first row of the tableau is the initial configuration, i.e., $(q_{\text{init}}, \triangleright, w_1 w_2 \dots w_n)$, padded at the end with $\sqcup$'s to fill n^k many squares.

The third formula, ϕ_{accept}, ensures that the last row is an accepting row (once we get an accepting row, we repeat it until we reach the n^k-th row). For example, ϕ_{accept} could be given as follows:

$$\bigvee_{1 \leq i,j \leq n^k} x_{ijq_{\text{accept}}}.$$

Finally, ϕ_{move} ensures that each row follows from the previous by a legal transition of N (or, in the case that the previous row was accepting, the current row is its copy). This can be implemented by observing that each row is exactly like the previous one, except in the six squares surrounding the state.

$x_{i(j-1)s_1}$	x_{ijs_2}	$x_{i(j+1)s_3}$
$x_{(i+1)(j-1)s_4}$	$x_{(i+1)js_5}$	$x_{(i+1)(j+1)s_6}$

Let $W_{ij}(s_1, \dots, s_6)$ be the conjunction of the entries of the above table. So, ϕ_{move} is

$$\bigwedge_{i,j} \bigvee_{\substack{s_1, s_2, s_3, s_4, s_5, s_6 \\ \text{legal window}}} W_{i,j}(s_1, s_2, s_3, s_4, s_5, s_6). \tag{3}$$

CLAIM 2.14. ϕ_w can be constructed in logspace in $|w|$.

Proving this claim in detail is the hard part of the proof. But, to be convinced of it, note that in logspace we can maintain constantly many pointers and counters of logarithmic length (and hence of sufficiently many bits to be able to index the tableau and ϕ_w). This is sufficient to construct ϕ_w because its structure is simple.[6] □

A *literal* is a variable or its negation, i.e., x or $\neg x$. It is often convenient to express $\neg x$ as $\bar{x}$; we use the two notations interchangeably. A *clause* is a disjunction of literals, i.e., $(l_1 \vee l_2 \vee \ldots \vee l_n)$ where each l_i is a literal. Every Boolean function can be represented as a Boolean formula in *conjunctive normal form (CNF)*, meaning that it is a conjunction of clauses. An example of a CNF formula is $(x \vee \bar{y} \vee \bar{z}) \wedge (y) \wedge (\bar{x} \vee z)$. A formula is in *disjunctive normal form (DNF)* if it is a disjunction of conjunctions of literals. Furthermore, 3CNF is a CNF formula where each clause has exactly three literals.

Exercise 2.15. Show that any Boolean function has a CNF and a DNF.

Corollary 2.16. Let 3Sat be Sat restricted to 3CNF formulas. Then 3Sat is NP-complete.

Proof. The formula ϕ_w from the proof of theorem 2.13 can be given as a CNF formula without a significant increase in size. A general CNF can be transformed into 3CNF, also with little increase in size. □

Exercise 2.17. Show that ϕ_w can be efficiently transformed (in logspace in $|w|$) into a CNF formula.

Exercise 2.18. Show how to pad clauses with 1 or 2 literals, while preserving satisfiability. Then show how to shrink clauses from $n > 3$ literals to just 3 literals.

Using reductions from Sat and 3Sat we can now start showing many more NP problems to be NP-complete; see §8.1. A comprehensive list of NP-complete problems and reductions can be found in the classic book [**GJ79**]. Some unexpected problems have also been shown to be NP-complete, for example Minesweeper.

Exercise 2.19. Suppose that a Boolean expression on n variables has less than n^k clauses, each with at least $k \log n$ distinct variables. Show that it must have a satisfying truth assignment, and give a polytime algorithm for finding such an assignment.

Let M be a nondeterministic TM, and let $\#\text{accept}_M(x)$ be the number of accepting paths of M on input x. We define the functional class #P to be the class of functions f for which there is a polytime nondeterministic TM M such that $f(x) = \#\text{accept}_M(x)$. The majority of the standard (logspace) many-one reductions for classical NP-complete problems are *parsimonious* (they preserve the number of solutions). Hence they can be used to show that the counting versions of these problems are complete for #P.

[6] Carrying the proof of this claim in detail once in your life is what makes a complexity *Pollywog* into a complexity *Shellback*.

2.3. Self-reducibility of satisfiability

If α is a Boolean formula and v is a variable, then $\alpha[\mathtt{T}/v]$ and $\alpha[\mathtt{F}/v]$ are new Boolean formulas defined as follows: every instance of the variable v is replaced by $\mathtt{T}$ (respectively $\mathtt{F}$) and the resulting formula simplified: subformulas of the form $\mathtt{T}\wedge\beta, \mathtt{T}\vee\beta, \mathtt{F}\wedge\beta, \mathtt{F}\vee\beta$ are replaced by $\beta, \mathtt{T}, \mathtt{F}, \beta$, respectively. We sometimes abbreviate "α is satisfiable" with "$\alpha \in$ SAT."

SAT is *self-reducible* in the following sense: if α is a formula and v is a variable, then

$$\alpha \in \text{SAT} \iff (\alpha[\mathtt{T}/v] \in \text{SAT} \quad \text{or} \quad \alpha[\mathtt{F}/v] \in \text{SAT}).$$

THEOREM 2.20. Consider languages over $\Sigma = \{0,1\}$. Suppose that we have a set $T \subseteq \{1\}^*$ (i.e., T is a set consisting of strings of 1s); such a set is called a *tally* set. If T is NP-hard, then $\mathsf{P} = \mathsf{NP}$.

PROOF. Assume that T is NP-hard, and so SAT $\leq T$, and let g be the function implementing a (logspace) reduction from SAT to T.

We give a polytime algorithm for SAT using this g. The algorithm works in stages: at stage 0, let $C_0 = \{\alpha\}$, where α is the input formula. At stage $(i+1)$, $C_i = \{\alpha_1, \ldots, \alpha_n\}$ (where C_i is the result of the previous stage, i.e., stage i), and we create

$$C' = \{\alpha_1[\mathtt{T}/v_{i+1}], \alpha_1[\mathtt{F}/v_{i+1}], \ldots, \alpha_n[\mathtt{T}/v_{i+1}], \alpha_n[\mathtt{F}/v_{i+1}]\}.$$

Thus, C' contains all the formulas of C_i with v_{i+1} set to $\mathtt{T}$ and to $\mathtt{F}$ (so $|C'| = 2 \cdot |C_i|$), and simplified.

We now prune C' as follows: we compute $g(\beta)$ for every $\beta \in C'$. Whenever we get two formulas that map to the same string in $\{1\}^*$, we keep only one of them. If $g(\beta)$ maps to some string not in $\{1\}^*$, we simply delete β. We let C_{i+1} be the result of this pruning. At the end, when no variables are left, $C_k \subseteq \{\mathtt{T}, \mathtt{F}\}$, and we answer "yes" (i.e., "yes, α is satisfiable") iff $\mathtt{T} \in C_k$.

This algorithm is polytime, because the pruning ensures that the C_i's are always polysize (there are polynomially many unary strings of polynomial size). It is correct since g is assumed to be a correct reduction. □

EXERCISE 2.21. We say that a language is *sparse* if there is a polynomial $p(n)$ such that $|L \cap \{0,1\}^n| \leq p(n)$. Show that if there is a sparse language L which is hard for co-NP with respect to logspace many-one reductions (i.e., $\leq^m_{\mathsf{L}}$), then in fact $\mathsf{P} = \mathsf{NP}$.

THEOREM 2.22 (Mahaney). If there are sparse languages which are hard for NP, then $\mathsf{P} = \mathsf{NP}$.

See [**HO02**, §1.1.2] for a proof of Mahaney's theorem. The *Berman-Hartmanis Isomorphism Conjecture* (IC) states that all NP-complete languages (here it is the $\leq^m_{\mathsf{P}}$ notion of completeness) are polytime isomorphic (inter-reducible with polytime many-one reductions that are bijections, with polytime inverses). In other words, there is only one NP-complete set in many guises ([**HO02**, pp. 26 and 282]). A language is *dense* if it is not sparse. A line of attack on the IC was to show the existence of a sparse NP-complete language, since no dense NP-complete language (such as SAT)

can be polytime isomorphic to a sparse language, thereby showing IC to be false. Mahaney's theorem shows the futility of this line of attack: if sparse NP-complete languages exist, then P = NP, in which case IC fails trivially anyways.

EXERCISE 2.23. Observe that if the IC is true, then $\mathsf{P} \neq \mathsf{NP}$.

The next theorem also uses the idea of self-reducibility of SAT, and it introduces the notion of the Polytime Hierarchy (PH) which we are going to cover in more detail in §4.3. Define Σ_i^p to be the class of languages L for which there is a polytime relation R such that

$$x \in L \iff \exists y_1 \forall y_2 \ldots Q y_i R(x, y_1, y_2, \ldots, y_i),$$

where $R(x, y_1, y_2, \ldots, y_i)$ is decidable in polytime in $|x|$. Let Π_i^p be defined analogously, but starting with a $\forall$ quantifier. Then, $\mathsf{PH} = \bigcup_i \Sigma_i^p = \bigcup_i \Pi_i^p$.

THEOREM 2.24 (Karp-Lipton). If all languages in NP have polysize circuits, that is, $\mathsf{NP} \subseteq \mathsf{P/poly}$, then PH collapses to its second level, that is, $\mathsf{PH} = \Sigma_2^p$.

PROOF. The class P/poly consists of those languages that have polysize circuits; see the definition on page 65 in chapter 5.

It is enough to show that if $\mathsf{NP} \subseteq \mathsf{P/poly}$, then $\Pi_2^p \subseteq \Sigma_2^p$ (see footnote[7]). To show that $\mathsf{NP} \subseteq \mathsf{P/poly} \Rightarrow \Pi_2^p \subseteq \Sigma_2^p$ argue as follows: assume L is in Π_2^p, so $L = \{x | \forall y \exists z R(x, y, z)\}$ (where $|y|, |z|$ are implicitly bounded by a polynomial in $|x|$). Consider the language $L' = \{\langle x, y\rangle | \exists z R(x, y, z)\}$, which is in NP by lemma 2.2. Thus, there exists a polytime function f such that $L = \{x | \forall y\ f(\langle x, y\rangle) \in \text{SAT}\}$.

Note that $x \in L \iff \exists T \forall y [T(y)$ satisfies $f(\langle x, y\rangle)]$, where $T(y)$ is a truth assignment that satisfies $f(\langle x, y\rangle)$ for the given y (if one exists). This is a way of stating that $x \in L$ with the right alternation of quantifiers. The problem is that the naive way of representing T would be as a string of truth assignments for each y, and there are exponentially many y's (in $|x|$), so this does not work.

But we assumed that $\mathsf{NP} \subseteq \mathsf{P/poly}$, and using the self-reducibility of SAT we conclude that for all n, there exists a circuit C_n such that for all ϕ, $|\phi| = n$, C_n outputs a satisfying assignment to ϕ (if one exists).

Now note that $|y| \leq p(|x|)$, for some polynomial p, so for any given x_0, we have that $|f(\langle x_0, y\rangle)| \leq q = q(|x_0|)$, where q is a polynomial that depends on p and the polynomial bounding f. So,

$$x \in L \iff \exists C = \langle C_0 C_1 \ldots C_q\rangle \forall y [C_{|f(\langle x,y\rangle)|}(f(\langle x, y\rangle)) \text{ satisfies } f(\langle x, y\rangle)],$$

where $|C|$ and $|y|$ can be bounded by a polynomial in $|x|$, and the predicate decided in time polynomial in $|x|$, and where $\langle C_0 C_2 \ldots C_1\rangle$ denotes the encoding of a sequence of q circuits. Hence L is in Σ_2^p. □

[7]It follows more or less from definition that if $\Pi_2^p \subseteq \Sigma_2^p$ then in fact the entire hierarchy collapses to Σ_2^p, i.e., $\mathsf{PH} = \Sigma_2^p$, but see §4.3 for the necessary background.

2.4. Padding argument

Define the *padding function* as $\text{pad} : \Sigma^* \times \mathbb{N} \longrightarrow (\Sigma \cup \{\#\})^*$ where $\text{pad}(s, l) = s\#^j$ for $j = \max(0, l - |s|)$. For any language L and function $f : \mathbb{N} \longrightarrow \mathbb{N}$ let

$$\text{pad}(L, f(n)) = \{\text{pad}(s, f(|s|))|\text{where } s \in L\}.$$

Note, for example, that if $L \in \mathsf{TIME}(n^6)$, then $\text{pad}(L, n^2) \in \mathsf{TIME}(n^3)$. To see this, use essentially the same machine that decides L in time n^6 to decide $\text{pad}(L, n^2)$ in time n^3 by making it "ignore" the trailing junk[8] (note that $|\text{pad}(x, |x|^2)| = |x|^2$, and x is decided in time $|x|^6 = (|x|^2)^3$).

This seemingly innocuous trick to reduce the computational time of TMs has some interesting applications.

THEOREM 2.25. If $\mathsf{EXPTIME} \neq \mathsf{NEXPTIME}$, then $\mathsf{P} \neq \mathsf{NP}$.

PROOF. Show the contrapositive: assume $\mathsf{P} = \mathsf{NP}$ and L is in $\mathsf{NEXPTIME}$. Then $\text{pad}(L, 2^{n^k})$ is in NP, and so it is in P, and hence L is in $\mathsf{EXPTIME}$. □

THEOREM 2.26 (Ladner). If $\mathsf{P} \neq \mathsf{NP}$, then there is a language in $\mathsf{NP} - \mathsf{P}$ which is *not* NP-complete.[9]

PROOF. Consider $\text{PADSAT} = \{\text{pad}(\phi, |\phi|^{p(|\phi|)})|\phi \in \text{SAT}\}$, where we next define the padding function $p(n)$. Let $M_1, M_2, M_3, \ldots$ be the list of all Turing machines. This list can be obtained from the universal Turing machine—recall that the UTM takes as input the description of another TM as well as an input to that machine, and simulates the machine on its input. We also want all the machines to occur infinitely often in the list; this can be obtained from the original list $M_1, M_2, M_3, \ldots$ by re-listing them as follows: in the i-th stage list all machines starting with M_1 and ending with M_i.

Let $p(n)$ be the smallest $i < \log\log n$ such that for every $x \in \{0,1\}^{\leq \log n}$ the following holds: M_i halts on x within $(|x|^i + i)$ steps and accepts iff $x \in \text{PADSAT}$. If there is no such i, we let $p(n) = \log\log n$. Note that $p(n)$ is defined in terms of itself, but the recursion is well defined since to compute $p(n)$ we need only consider $p(k)$ for $k \leq \log n$ (when we check that $x \in \text{PADSAT}$).

The idea behind the "strange" definition of $p(n)$ is to have a function that grows fast enough so that PADSAT is not NP-complete, but slowly enough to ensure that it is not in P. In what follows we show that this is indeed the case.

CLAIM 2.27. PADSAT is not in P.

PROOF. Suppose that it is, and it is decided by some M running in time $(n^k + k)$. There is an $i > k$ such that $M = M_i$ (here we use the assumption that each machine

[8]Not quite ignore, as it has to check that the input string is of the form $w = \text{pad}(x, |x|^2)$, and reject if not.

[9]There are candidates for such languages: given two graphs $G = (V, E)$ and $G' = (V', E')$, we say that they are isomorphic ($G \equiv G'$) if there exists a bijection $\pi : V \longrightarrow V'$ such that $(i, j) \in E \iff (\pi(i), \pi(j)) \in E'$. We suspect that $\text{GRAPHISOMORPHISM} = \{\langle G, G' \rangle : G \equiv G'\}$ is an example of a language which is in $\mathsf{NP} - \mathsf{P}$ and yet not NP-complete. It is easy to see that GRAPHISOMORPHISM is in NP: the certificate is π.

occurs infinitely often in our list of machines). Therefore, by the definition of p, for all $n > 2^{2^i}$, $p(n) \leq i$. But this means that for $n > 2^{2^i}$, PadSat is just Sat padded with # to be of length at most n^i. So if PadSat were in P, so would Sat by the following procedure: given ϕ, where $|\phi| = n$, if $n \leq 2^{2^i}$, check if ϕ is satisfiable by examining its truth table; if, on the other hand, $n > 2^{2^i}$, then check if $\phi\#^{n^i - n} \in$ PadSat. This contradicts the assumption that $\mathsf{P} \neq \mathsf{NP}$. □

Claim 2.28. $\lim_{n\to\infty} p(n) = \infty$.

Proof. Since PadSat $\notin \mathsf{P}$ (by claim 2.27), for each i we know that there exists an x such that given time $(|x|^i + i)$, M_i either does not halt or if it does, it gives the incorrect answer to the question whether $x \in$ PadSat. Then, we know (from the definition of p) that for every $n > 2^{|x|}$, $p(n) \neq i$. So for every i there are only finitely many n's such that $p(n) = i$. □

Claim 2.29. PadSat is not NP-complete.

Proof. As $p(n)$ tends to infinity with n (by claim 2.28), the padding is of super-polynomial size. Suppose PadSat is NP-complete, so Sat $\leq$ PadSat. This reduction takes ψ (with $|\psi| = n$), to $\text{pad}(\phi, |\phi|^{p(|\phi|)})$, such that $|\text{pad}(\phi, |\phi|^{p(|\phi|)})|$ is $O(n^k)$ for some fixed k. Therefore, $|\phi|^{p(|\phi|)}$ is $O(n^k)$. But this means that $|\phi|$ must be $o(n)$ (i.e., we can find a fraction $\frac{p}{q}$ such that $p, q \in \mathbb{N}$, and $p < q$, and $|\phi| < n^{\frac{p}{q}}$). Using this fact, we can now design a polytime algorithm for Sat which applies the reduction repeatedly (given a constant a, we want an i such that $n^{\left(\frac{p}{q}\right)^i} < a$; this i is $O(\log\log n)$), each time obtaining a smaller ϕ, until it is of constant size, and can be solved by brute force. It follows that $\mathsf{P} = \mathsf{NP}$; contradiction. □

This ends the proof of Ladner's theorem. □

Exercise 2.30. To finish the above proof, show that PadSat $\in \mathsf{NP}$.

2.5. Answers to selected exercises

Exercise 2.5. No, it only makes the bound tighter, and emphasizes that $|y \cdot z| \leq p(|x|)$.

Exercise 2.6. The difficulty is that while $f(x)$ can be computed in logspace, $|f(x)|$ may still be of polynomial length in $|x|$, and it is perfectly legal to have a long $y = f(x)$ on the output tape. So, when we are computing the composition of two logspace functions f, g, once we have computed $f(x)$, we cannot put it on the work tape, and compute $g(f(x))$ directly. To fix this, we recompute the i-th bit of $f(x)$ each time it is required in the computation of $g(f(x))$.

Exercise 2.17. The formula ϕ_w is almost in CNF form, we only have to iron out a few wrinkles. Note that ϕ_w is composed of four parts:

$$\phi_{\text{cell}}, \phi_{\text{start}}, \phi_{\text{move}}, \phi_{\text{accept}}.$$

We have to make sure that each part is either a disjunction, a conjunction, or a conjunction of disjunctions. The formula ϕ_{accept} is just a disjunction, so it is already in the right form. The formula ϕ_{start} can be given as a conjunction of the variables x_{1js} asserting that the j-th symbol is the j-th symbol of the initial configuration. The formula ϕ_{cell} asserts that each cell contains exactly one symbol, so it is of the form

$$\bigwedge_{i,j} \bigvee_{s} \left(x_{ijs} \wedge \bigwedge_{s' \neq s} \neg x_{ijs'} \right).$$

Using distributivity of $\wedge$ and $\vee$, we can push the $\bigvee_s$ inside, incurring a small increase in size, because the number of symbols s is a constant. This is a good place to make an observation: suppose we have a DNF formula of the form:

$$\overbrace{\underbrace{(\ldots \wedge \ldots)}_{n} \vee (\ldots \wedge \ldots) \vee \ldots \vee (\ldots \wedge \ldots)}^{m},$$

i.e., each clause has n literals, and there are m clauses. Then, using distributivity, this can be transformed into CNF so that each clause has m literals, and there are n^m many clauses. Of course, the same can be done to transform CNF into DNF (we shall use this fact in §5.3.1).

Thus, we can put ϕ_{cell} into CNF effectively, since the number of symbols s is a constant, so from s clauses, by the above observations, we get s^s many clauses—still a constant.

Finally, we deal with ϕ_{move} (see equation (3)). Again, there are constantly many legal windows, so we can push the $\bigvee$ all the way inside incurring little increase in size.

Exercise 2.18. $(l_1 \vee l_2)$ can be transformed to $(l_1 \vee l_2 \vee x) \wedge (l_1 \vee l_2 \vee \bar{x})$ (or simply $(l_1 \vee l_2 \vee l_2)$). For the case (l_1) introduce two new variables x, y and four clauses, each with l_1 and an x-literal and a y-literal, with the four possible arrangements of negations for x and y. Finally, for $n > 3$ literals, send $(a_1 \vee a_2 \vee \ldots \vee a_n)$ to

$$(a_1 \vee a_2 \vee z_1) \wedge (\overline{z}_1 \vee a_3 \vee z_2) \wedge (\overline{z}_2 \vee a_4 \vee z_3) \wedge \ldots \wedge (\overline{z}_{n-3} \vee a_{n-1} \vee a_n),$$

where the z_i's are new variables. The claim is that the new formula ϕ'_w is satisfiable iff ϕ_w is satisfiable (note that the two formulas are not logically equivalent—ϕ'_w has more variables).

Exercise 2.19. If a clause has at least $k \log n$ distinct variables, then the number of truth assignments which falsify it has to be less than $2^{n-k\log n} = \frac{2^n}{n^k}$. Then the number of truth assignments which falsify at least one clause is less than $n^k \frac{2^n}{n^k} = 2^n$, and therefore there must be at least one truth assignment which satisfies the whole formula. Consider the original set of clauses ϕ and set the first variable to either 0 or 1, thereby getting two sets of clauses, ϕ_0 and ϕ_1. If we denote by $f(\psi)$ the number of truth assignments falsifying ψ, we can easily see that $f(\phi) = f(\phi_0) + f(\phi_1)$. But we know that $f(\phi) < 2^n$ and so $f(\phi_i) < 2^{n-1}$, for $i = 0$ or $i = 1$. It turns out, that if we select the i which satisfies the most clauses (and choose i arbitrarily if there is a tie)

we can get a satisfying assignment (see [**Pap94**, theorem 13.2]). Repeating the above step n times we will get a total truth assignment satisfying the original formula.

Exercise 2.21. See the proof of theorem 1.4 in [**HO02**].

Exercise 2.23. Suppose the IC is true, and so is $\mathsf{P} = \mathsf{NP}$. Then every language in P (other than $\emptyset$ and Σ^*) is NP-complete (with respect to polytime reductions), and there are finite languages in P.

2.6. Notes

For the proof of theorem 2.13 we follow [**Sip06**, theorem 7.37]. Exercise 2.19 is [**Pap94**, exercise 11.5.23]. Possible references for theorem 2.20 are [**Pap94**, theorem 14.3, pg. 337] and [**HO02**, theorem 1.2, pg. 3]. Theorem 2.24 is based on [**KL80**]. For a proof of theorem 2.22, Mahaney's theorem, see [**HO02**, §1.1.2]; a new technique is needed for this proof, the so called *left set* technique.

3

Space

3.1. Basic definitions and results

A language L is in the deterministic class $\mathsf{SPACE}(f(n))$ (or nondeterministic class $\mathsf{NSPACE}(f(n))$) if there is a TM with a read-only input tape that decides L and uses at most the first $O(f(n))$ squares on each of the work tapes. The standard space classes are the following:

$$\begin{array}{ll} \mathsf{L} = \mathsf{SPACE}(\log n) & \mathsf{PSPACE} = \bigcup_k \mathsf{SPACE}(n^k) \\ \mathsf{NL} = \mathsf{NSPACE}(\log n) & \mathsf{NPSPACE} = \bigcup_k \mathsf{NSPACE}(n^k), \end{array}$$

and $\mathsf{co\text{-}NL}$ which contains the complements of all the languages in NL.

We say that a function $f : \mathbb{N} \longrightarrow \mathbb{N}$ is *space constructible* if there exists a TM M which on any input of length n (i.e., on any $x \in \{0,1\}^n$) marks off $f(n)$ many squares of the output tape, and the machine uses at most $f(n)$ squares of its tapes to do so. We use space constructible functions as a "ruler" to mark off how much space we have to do the job. Note that the usual functions such as $\log n$, $(\log n)^k$, n^k, $2^{(\log n)^k}$, 2^{n^k}, etc., are all space constructible functions. Thus, unless stated explicitly, we always assume that our space bounds are constructible functions.

There is no guarantee of termination for space-based classes, as the same configuration may be repeated, and so the TM may find itself in a loop. However, when space is bounded by $f(n)$, the number of different configurations that may occur on any branch is bounded by $2^{c \cdot f(n)}$, for some constant c.

Exercise 3.1. Show how to compute this constant c.

So we can implement a counter that increases by one after each step of the computation, and the TM rejects when the counter exceeds $2^{c \cdot f(n)}$ (such a counter requires only $O(f(n))$ space on some dedicated work tape). Note that for a space bounded nondeterministic computation, if an accepting branch exists at all, there is an accepting branch of length at most $2^{c \cdot f(n)}$.

Exercise 3.2. Consider the following three models of computation: (1) Logspace-bounded TMs.

(2) Automata with a two-way read-only input head and a fixed finite number of integer counters. The counters can hold an integer between 0 and n, the length of the input. At each step the automaton may test each of its counters for zero, and based on this information and its current state, it may add one or subtract one from each of the counters, move its read head left or right, and enter a new state.

(3) Automata with a fixed finite number of two-way read-only input heads that may not move outside the input—the idea is that the input is repeated on each tape, but only its length really matters for all tapes but the first one; the other tapes work as counters.

Show that for either deterministic or nondeterministic machines, these three models of computation are equivalent.

Let REACH $= \{\langle G, s, t\rangle | t$ is reachable from $s\}$ be the language of encodings of undirected graphs G with two specified nodes s, t, such that there is a path in G from s to t. The graph G is assumed to be presented in any one of the standard ways of encoding graphs, for example as an adjacency matrix. DIRECTREACH is the problem of reachability where G is a directed graph. Note that DIRECTREACH is also known as s,t-CONNECTIVITY, MAZE, etc.

THEOREM 3.3. DIRECTREACH is NL-complete.

PROOF. The following algorithm decides DIRECTREACH in NL.

ALGORITHM 3.4 (Nondeterministic Reachability).
On input $\langle G, s, t\rangle$:

1. $u := s$
2. while $(u \neq t)$
3. *nondeterministically* select a vertex v
4. if $(u, v) \notin E$, reject
5. else $u := v$
6. accept

Next we show that DIRECTREACH is NL-hard.[1]

To this end we introduce the fundamental concept of a *configuration graph* which, for a given TM M and a given input x, is the directed graph $C_{M,x}$ whose nodes are all the configurations of M on input x, and where there is an edge (C_1, C_2) iff there is a legal transition of M from configuration C_1 to C_2 (i.e., C_1 yields C_2).

It is obvious that the problem of acceptance of a TM M on input x can now be viewed as the problem of directed reachability on the configuration graph $C_{M,x}$ with s being the initial configuration C_{init} and t being the accepting configuration C_{accept}. The only issue is that there may be several accepting configurations (the computation is nondeterministic), and furthermore, we may not know the contents of the tape of an accepting configuration in advanced. We fix this problem by putting the machine in a standard form before accepting; for example by clearing the tapes of all symbols and making the heads move to the left-most square and rest on $\triangleright$. Now for a k tape TM we have that $C_{\text{accept}} = (q_{\text{accept}}, \triangleright, \varepsilon, \ldots, \triangleright, \varepsilon)$.

A configuration graph $C_{M,x}$ for a TM whose space is bounded by $f(|x|)$ has at most $2^{c \cdot f(|x|)}$ many nodes (i.e., configurations), and hence an NL machine has a configuration graph of polynomial size.

[1] With respect to logspace reductions (i.e., $\leq_{\mathsf{L}}^{m}$) which we defined on page 27.

Let $L \in \mathsf{NL}$, and let M be the nondeterministic logspace bounded TM that decides L. We reduce L to DirectReach via

$$g(x) = \langle C_{M,x}, C_{\text{init}}, C_{\text{accept}} \rangle.$$

This proof is finished in the next exercise. □

Exercise 3.5. Complete the last paragraph of the above proof: show that g is a logspace function. In order to do that you will have to define precisely the encoding $\langle C_{M,x}, C_{\text{init}}, C_{\text{accept}} \rangle$.

Exercise 3.6. Show that 2Sat is NL-complete. Hint: given a 2CNF formula ϕ, transform it into a graph G_ϕ, where the nodes of the graph are all the variables of ϕ, as well as the nagations of all the variables of ϕ. Now, given two nodes α and β, we add the edges $(\alpha, \beta), (\overline{\beta}, \overline{\alpha})$ if and only if $\overline{\alpha} \vee \beta$ is a clause of ϕ. The understanding is that $\overline{\alpha}$ is $\neg x$ if $\alpha = x$, and $\overline{\alpha}$ is x if $\alpha = \neg x$. Show that G_ϕ has the following property: ϕ is satisfiable if and only if there is no variable x such that G_ϕ has a path from x to $\neg x$. (Cf. lemma 6.15.)

Theorem 3.7 (Savitch). DirectReach $\in \mathsf{SPACE}(\log^2 n)$.

Proof. In fact Savitch's theorem proves a stronger result: for every $f(n) \geq \log n$, $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}((f(n))^2)$. Thus, we can get rid of nondeterminism at the cost of squaring the space bound.

Define the Boolean predicate $\mathrm{R}(G, u, v, i)$ to be `T` iff there is a path in G from u to v of length at most 2^i. The key observation is that such a path exists if there exists a w which is a mid-point of the path. In other words there exist paths of distance at most 2^{i-1} from u to w and from w to v, i.e.,

$$(\exists w)[\mathrm{R}(G, u, w, i-1) \wedge \mathrm{R}(G, w, v, i-1)].$$

Using the above recursive definition, the following algorithm computes the predicate $\mathrm{R}(G, u, v, i)$.

Algorithm 3.8 (Savitch).
On input G, u, v, i

```
1. R(G,u,v,i):
2.   if i = 0 then
3.     if u = v, then return T
4.     if (u,v) is an edge, then return T
5.   else for each vertex w:
6.     if R(G,u,w,i-1) and R(G,w,v,i-1), then return T
7. return F
```

Let $L \in \mathsf{NSPACE}(f(n))$, and let M be a TM deciding L in nondeterministic $f(n)$-space. We have already seen in the proof of theorem 3.3 that such an M has at most $2^{c \cdot f(n)}$-many configurations. Thus,

$$x \in L \iff \mathrm{R}(C_{M,x}, C_{\text{init}}, C_{\text{accept}}, c \cdot f(n)).$$

The next exercise finishes this proof. Note that the machine implementing Savitch's algorithm need not keep the entire graph $C_{M,x}$ on its tape. The only place where

the graph is consulted is in step 4 of the above algorithm (to establish if (u, v) is an edge), and this can be done on the basis of the transition function of M. □

EXERCISE 3.9. Show that the algorithm in the proof of Savitch's theorem is correct (i.e., it computes $\mathrm{R}(G, u, v, i)$ correctly) and it requires at most $i \cdot s$ space, where s is the number of bits required to keep record of a node.

The surprising consequence of Savitch's theorem is that nondeterminism does not increase the power of polynomial space.

COROLLARY 3.10. PSPACE = NPSPACE.

As we already saw in chapter 2, satisfiability of Boolean formulas (SAT) is NP-complete. On the other hand, satisfiability of quantified Boolean formulas (QSAT) is PSPACE-complete.

A *quantified Boolean formula* (QBF) is defined inductively as ordinary Boolean formulas (see page 25), but we also have two additional cases: if α is a Boolean formula, then so are $(\forall x\alpha)$ and $(\exists x\alpha)$. The semantics of Boolean quantifiers are as follows: $\exists x\alpha(x) \leftrightarrow (\alpha(0/x) \vee \alpha(1/x))$, and $\forall x\alpha(x) \leftrightarrow (\alpha(0/x) \wedge \alpha(1/x))$.

Note that Boolean quantifiers do not increase the power of expressibility; they only seem[2] to allow greater succinctness. For example,

$$\exists x_1 \exists x_2 \ldots \exists x_n \alpha(x_1, x_2, \ldots, x_n) \leftrightarrow \bigvee_{a_1 a_2 \ldots a_n \in \{0,1\}^n} \alpha(a_1/x_1, a_2/x_2, \ldots, a_n/x_n).$$

Clearly, the RHS is greater than the LHS by a factor of 2^n.

Let $\mathrm{QSAT} = \{\langle\phi\rangle | \phi \text{ is a satisfiable QBF}\}$. Note that a quantified Boolean formula may not have any free variables, in which case it is in QSAT if it is true.

THEOREM 3.11. QSAT is PSPACE-complete.

PROOF. To show that QSAT is in PSPACE try all the assignments recursively; this can be accomplished with quadratic space.

Let L be any language decided by TM M in space n^k. We show $L \leq \mathrm{QSAT}$. Let $\phi_{\vec{c}_1, \vec{c}_2, t}$ be a quantified Boolean formula, where: $\vec{c}_1, \vec{c}_2$ are two groups of variables representing configurations of M on a particular w ($|w| = n$), t is a parameter denoting the number of steps of M, so $t \leq 2^{O(n^k)}$, and $\phi_{\vec{c}_1, \vec{c}_2, t}$ is true if and only if $\vec{c}_1$ yields $\vec{c}_2$ in at most t steps. We show how to construct $\phi_{\vec{c}_1, \vec{c}_2, t}$.

The formula $\phi_{\vec{c}_1, \vec{c}_2, 1}$ is quantifier-free, and formalizes the notion that $\vec{c}_1, \vec{c}_2$ are either the same configuration, or $\vec{c}_1$ yields $\vec{c}_2$ in one step. (Recall ϕ_{move} in the proof of theorem 2.13.) For $t > 1$, we build our formula recursively as follows: $\phi_{\vec{c}_1, \vec{c}_2, t}$ is given by: $\exists \vec{m}_1 [\phi_{\vec{c}_1, \vec{m}_1, \lceil \frac{t}{2} \rceil} \wedge \phi_{\vec{m}_1, \vec{c}_2, \lceil \frac{t}{2} \rceil}]$, except that this would double the size of the formula at each recursive step. To avoid this problem we restate this formula as follows:

$$\exists \vec{m}_1 \forall \vec{x} \forall \vec{y} [\{(\vec{x} \leftrightarrow \vec{c}_1 \wedge \vec{y} \leftrightarrow \vec{m}_1) \vee (\vec{x} \leftrightarrow \vec{m}_1 \wedge \vec{y} \leftrightarrow \vec{c}_2)\} \rightarrow \phi_{\vec{x}, \vec{y}, \lceil \frac{t}{2} \rceil}],$$

where $\vec{a} \leftrightarrow \vec{b}$ abbreviates $\wedge_i (a_i \leftrightarrow b_i)$.

Therefore, $w \in L \iff \langle \phi_{\vec{c}_{\mathrm{init}}, \vec{c}_{\mathrm{accept}}, 2^{O(n^k)}} \rangle \in \mathrm{QSAT}$. □

[2] We say "seem" as it is an open question whether Boolean quantification allows to significantly (i.e., super-polynomially) shorten Boolean formulas.

3.2. The inductive counting technique

This section presents the famous Immerman-Szelepcsényi result showing that for $f(n) \geq \log n$, $\mathsf{NSPACE}(f(n)) = \mathsf{co\text{-}NSPACE}(f(n))$. We present the proof for $f(n) = \log n$ in theorem 3.12 below, and leave the extension to $f(n) > \log n$ (for "reasonable" $f(n)$, i.e., constructible $f(n)$) as exercise 3.14.

The class LBA (Linear Bounded Automata) is defined as $\mathsf{LBA} = \mathsf{NSPACE}(n)$. In 1964 Sige-Yuki Kuroda proposed two problems that became known as the "LBA problems." The first LBA question, which to this day remains open, is whether the nondeterminism can be done away with; i.e., $\mathsf{NSPACE}(n) =? \mathsf{SPACE}(n)$.

The second LBA problem is whether $\mathsf{NSPACE}(n) = \mathsf{co\text{-}NSPACE}(n)$, was answered in the affirmative by Immerman and Szelepcsényi (independently of each other).

THEOREM 3.12 (Immerman-Szelepcsényi). $\mathsf{NL} = \mathsf{co\text{-}NL}$.

PROOF. Let DIRECTUNREACH be the complement of DIRECTREACH, i.e., it is the language of encodings of G, s, t such that *there is no* path from s to t. As DIRECTREACH is NL-complete, DIRECTUNREACH is therefore $\mathsf{co\text{-}NL}$-complete, so to prove the theorem it is enough to show that DIRECTUNREACH is in NL.

Suppose that we know that the number of vertices reachable from s is k. How can we use this information to check that t is *not* reachable from s? For each vertex, $s = v_1, v_2, \ldots, v_{n-1}, t = v_n$, we attempt to guess a path from s to v_i. Each time we succeed we increase a counter by 1. If one of these paths contains t we reject. If the counter reaches k, but we have not yet checked $v_n = t$, we know t is *not* reachable from s, so we accept. If after checking $v_n = t$ the counter is $< k$, we reject.

So the question is: how to compute this k in NL? We use the *inductive counting* technique. This technique works in stages, computing at stage i how many nodes are reachable from s with paths of length at most i. First, we check how many nodes are reachable from s with paths of length 0 (answer: 1, just s). Then, we compute how many nodes are reachable with paths of length $(i+1)$ knowing how many nodes are reachable with paths of length i.

Let k_i be the number of nodes reachable from s with paths of length $\leq i$. For each node v, we try to guess a path of length $\leq i$ from s to each u. If we succeed, we increase a counter by 1, and we increase k_{i+1} if (u, v) is an edge. If at the end counter $< k_i$, we reject because we did not account successfully for all the nodes in the i-th layer. We repeat this for each v, and at the end, if all the checks have gone through, we have k_{i+1}. Of course, the number of nodes reachable from s, is $k = k_n$. Here is the algorithm for computing k_{i+1} from k_i.

ALGORITHM 3.13 (Immerman-Szelepcsényi).
On input i, k_i:

1. set $k_{i+1} := 1$
2. for every vertex $v \neq s$:
3. set $k'_i := 0$, TruthValue := `F`
4. for every vertex u:
5. guess whether u is in layer i; if guessed "no" go to next u
6. guess a path from s to u of length up to i; if guess failed, reject

7. set $k'_i := k'_i + 1$
8. if (u, v) is an edge, set TruthValue := T
9. if $k'_i < k_i$, reject
10. if TruthValue = T, set $k_{i+1} := k_{i+1} + 1$

Thus we can establish in NL that there is no path from s to t. □

EXERCISE 3.14. Show that the Immerman-Szelepcsényi theorem can be extended to show that for any $f(n) \geq O(\log n)$, $\mathsf{NSPACE}(f(n)) = \mathsf{co\text{-}NSPACE}(f(n))$.

3.3. Interactive Proof Systems

We introduce a new model of computation (still based on TMs), called interactive proof systems, and the class of languages decidable by such systems, called IP. The main result of this section is that IP = PSPACE. To define IP we introduce the concept of randomness, which we are going to explore further in chapter 7. Essentially, randomness allows us to make some decisions by the flipping of a coin, and accept or reject with a certain probability.

An *interactive proof system* consists of two independent TMs P (the *prover*) and V (the *verifier*). The two machines share a common read-only input tape and a read/write communication tape, but otherwise operate independently. Each machine has its own private work tape. In addition, V has access to a private string of random bits, and it runs in deterministic polytime. On the other hand, P has no time or space bounds, but it must halt, and it must write polysize messages to the communication tape.

The two machines take turns; when it is V's turn, it runs for a polynomial amount of time, accessing the random bits whenever it needs to make a probabilistic decision. At some point it writes a message for P, and then enters a special state that transfers control to P. Then it is P's turn, and they keep taking turns for a polynomial number of rounds. At the end, V decides to accept or reject. We assume that P is omniscient, except that it does not know V's random bits.

Formally, we say that a language L has an IP protocol if there exists a V such that:

- if $x \in L$, then there exists a P such that $\Pr_y[(P, V) \text{ accepts } x] \geq \frac{3}{4}$, and
- if $x \notin L$, then for all P, $\Pr_y[(P, V) \text{ accepts } x] \leq \frac{1}{4}$.

Note that in the second case, P might be a malicious prover bent on misleading V. The y's range over all the random strings up to the polynomial bounding the running time of V.

An example of a language with an IP protocol is SAT. On input ϕ, V requests a truth assignment satisfying ϕ. So P returns a string t—obtained, say, by a brute-force search—and if $t \vDash \phi$ (i.e., t satisfies α), then V accepts, and otherwise, V rejects. To analyze the error, note that if ϕ is satisfiable, then P will always return a satisfying assignment, so in this case the probability of error is 0, and if ϕ is not satisfiable, no matter how evil P is and what devilish tricks it has up its sleeve, at the end V always

tests t, so again the probability of error is 0. Note that in this example V did not make use of its random bits.

A more interesting example is GraphNonIsomorphism, where $\langle G, H\rangle$ is a "yes" instance if G and H are not isomorphic, i.e., there is no bijection $\tau : [n] \longrightarrow [n]$ such that (i, j) is an edge in G iff $(\tau(i), \tau(j))$ is an edge in H. Here V chooses a random permutation τ, applies it to both G and H to obtain G' and H', and then flips a coin to establish which of the two (i.e., G' or H') to send to P. Now P checks if it received an image of G (i.e., G') or of H (i.e., H'). If P answers correctly, V goes to the next round, and otherwise, V rejects. We assume polynomially many rounds.

If the two graphs G, H are indeed non-isomorphic, then P will have no trouble responding correctly (it can compute whether it got a permutation of G or of H, just by doing exhaustive search of all τ). So in this case V accepts with probability 1 (i.e., 0 probability of a false negative). But if they are isomorphic, then all that P can do is guess, and after k rounds it would have had to guess correctly k times in a row to fool V, and so the probability of acceptance (i.e., the probability of a false positive) is 2^{-k}, so for $k = 2$ we conform to the definition, and for $k = 100$ the probability of error becomes negligible.

We now prove the main result of this section, the surprising $\mathsf{IP} = \mathsf{PSPACE}$. We break it up into two parts: $\mathsf{IP} \subseteq \mathsf{PSPACE}$ and $\mathsf{PSPACE} \subseteq \mathsf{IP}$.

3.3.1. $\mathsf{IP} \subseteq \mathsf{PSPACE}$**.** We may assume that P's responses are just one bit long; the protocol can be easily modified so that V asks P for an answer one bit at a time. It is convenient to think of P as an oracle function $P : \Sigma^* \longrightarrow \{0, 1\}$ that, whenever queried on the string $\langle x, m_1, m_2, \ldots, m_k\rangle$, consisting of the input string and the history of the messages exchanged with V thus far, it answers $l_k = P(\langle x, m_1, m_2, \ldots, m_k\rangle)$ where l_k is 0 or 1.

Thus, the protocol may be described as a tree T_P (which depends on P) where the nodes are configurations of V, and where V's query to the tape with random bits is modeled by a binary branch (we assume that V reads the random bits from left to right, never moving left, i.e., it is not "recycling" the random bits). The probability of a path is the product of the edge probabilities on that path ($\frac{1}{2}$ for an edge belonging to a random branch, and 1 otherwise). The probability of a branch is the product of the probabilities on its edges. So the probability of V accepting is the sum of the probabilities of all the accepting branches (i.e., branches where the last configuration is accepting).

Regardless of the actual computational power of P, and without really knowing what P is, we can compute the largest possible number of accepting branches for any polynomially long sequence of query responses, and if their fraction is $\geq \frac{3}{4}$, we accept, and if it is $\leq \frac{1}{4}$ we reject. By the correctness of the original (P, V) protocol, we know that one or the other is always the case.

To compute this optimal number of accepting branches, we examine all possible P's, which in this context means examining all possible strings of polynomial length over $\{0, 1\}$. Each such string models a particular oracle P, where the i-th bit is the answer to the i-th query. In polynomial space we can consider all such P's, and compute the fraction of accepting branches for the corresponding T_P. This can be

done by a depth-first strategy, since each branch is of polynomial length, and each configuration is of polynomial size. Note that this also shows that while we assumed that P is all but omnipotent and omniscient, in fact it is enough to assume that P works in PSPACE.

3.3.2. PSPACE $\subseteq$ IP**.** This direction requires new ideas. We are going to prove that QSAT, which we showed in theorem 3.11 to be PSPACE-complete, is in IP. To this end, we introduce a new technique called *arithmetization*, which consists in translating Boolean formulas into multivariate polynomials.[3]

We take a QBF formula ϕ and first of all modify it so that negations occur only in front of variables (this is easy using de Morgan laws), and then translate it recursively into an algebraic expression p_ϕ. A single variable x is translated into x, and $\neg x$ into $(1-x)$, i.e., $p_x = x$ and $p_{\neg x} = (1-x)$, respectively. Furthermore,

$$\begin{aligned} p_{(\alpha\wedge\beta)} &= p_\alpha \cdot p_\beta, \\ p_{(\alpha\vee\beta)} &= p_\alpha + p_\beta, \\ p_{\forall x\alpha} &= \Pi_{a\in\{0,1\}} p_{\alpha(a/x)}, \\ p_{\exists x\alpha} &= \Sigma_{a\in\{0,1\}} p_{\alpha(a/x)}. \end{aligned}$$

Note that p_α is a syntactic object, just as α is a syntactic object; the Π, Σ are there to shorten the expression.[4] Let value(p_α) be the value of the expression (i.e., the semantic interpretation using standard arithmetic). Note that value$(p_\alpha) \geq 0$.

EXERCISE 3.15. Show that value$(p_\alpha) > 0$ iff α is true. Note that when we say "α is true," the tacit assumption is that it does not have free variables.

EXERCISE 3.16. Show that value$(p_\alpha) < 2^{2^{|p_\alpha|}}$. Note that this bound is too big to be of use to the verifier in an IP protocol.

EXERCISE 3.17. Show that for all a, such that $0 < a < 2^{2^n}$ there exists a prime number $k \in [2^n, 2^{3n}]$ such that $a \not\equiv_k 0$. You may use the fact that for every m there are at least $\sqrt{m}$ prime numbers $\leq m$, and the Chinese Remainder theorem; see theorem 8.23, on page 126.

The protocol starts as shown in figure 1 (see footnote[5]).

So now P must establish for V that it is indeed the case that $\hat{a} \equiv_k$ value(p_α). This is going to be established in several rounds. In each round, one of the following cases takes place.

- If $\alpha = (\alpha_1 \wedge \alpha_2)$ then P sends $\hat{a}_1, \hat{a}_2$ and V verifies that $\hat{a}_1 \cdot \hat{a}_2 \equiv_k \hat{a}$, and now requests that P prove that value$(p_{\alpha_1}) \equiv_k \hat{a}_1$ and value$(p_{\alpha_2}) \equiv_k \hat{a}_2$.

[3] We are also going to be using the technique of arithmetization in §5.3.2 to prove that bounded-depth circuits of polynomial size cannot compute parity.

[4] That is, $\exists x \exists y (x \vee y)$ is now expressed as $\Sigma_{a\in\{0,1\}} \Sigma_{b\in\{0,1\}} (a+b)$, with the Σs there, and not as $(0+0)+(0+1)+(1+0)+(1+1)$.

[5] Note that in light of the recent result showing that PRIMES are in P ([**AKS04**]) it is not really necessary to use Pratt's theorem (theorem 8.19) and a certificate of primality; V can verify directly whether k is a prime. Furthermore, V has a source of random bits, so if V were disposed not to use the polytime algorithm for PRIMES nor Pratt's theorem, V could always use the randomized Rabin-Miller algorithm (see §7.1.2).

Prover	Message	Verifier
Compute $a = \text{value}(p_\alpha)$ find an appropriate prime $k \in [2^n, 2^{3n}]$ and certificate of primality b let $\hat{a} \equiv a \pmod{k}$	$\overset{\hat{a},k,b}{\Longrightarrow}$ $\text{Prove value}(p_\alpha) \equiv_k \hat{a}$ $\overset{}{\Longleftarrow}$	Check that $\hat{a} > 0$, $k \in [2^n, 2^{3n}]$ and b is a correct certificate

FIGURE 1. Initial exchanges of the protocol.

- If $\alpha = (\alpha_1 \vee \alpha_2)$ then same thing happens, except that V verifies that $\hat{a}_1 + \hat{a}_2 \equiv_k \hat{a}$.
- If $\alpha = \forall x \alpha_1(x)$, then P sends a polynomial $\text{poly}_{\alpha_1}(x)$ that purportedly is equal to $p_{\alpha_1}(x)$. Then V checks that $\text{poly}_{\alpha_1}(0) \cdot \text{poly}_{\alpha_1}(1) \equiv_k \hat{a}$. Now V selects a random $r \in \mathbb{Z}_k$, and requests that P prove that $\text{value}(p_{\alpha_1}(r)) \equiv_k \text{poly}_{\alpha_1}(r)$.
- If $\alpha = \exists x \alpha_1(x)$, then the same thing happens as in the previous case, except that V checks that $\text{poly}_{\alpha_1}(0) + \text{poly}_{\alpha_1}(1) \equiv_k \hat{a}$.

At the end, V accepts if the values of the variables returned by P are consistent with the r-assignments given by V.

For example, suppose that the formula in question is $\forall x(x \wedge x)$, which is obviously false. Suppose that P claims that $\hat{a} = \text{value}(p_{\forall x(x \wedge x)}) = 6$, the prime is $k = 7$, and P sends the polynomial $x^2 + 2$, i.e., P claims that $\text{poly}_{(x \wedge x)}(x) = x^2 + 2$. So now V checks that $\text{poly}_{(x \wedge x)}(0) \cdot \text{poly}_{(x \wedge x)}(1) = (0^2 + 2)(1^2 + 2) = 6$, selects $1 \in \mathbb{Z}_7$, and wants to verify that $\text{value}(p_{(x \wedge x)}(1)) = \text{poly}_{(x \wedge x)}(1) = 3$. And now P is in a jam, because there is no way it can convince V that $\text{value}(p_{(x \wedge x)}(1)) = 3$, since $1 \cdot 1 \neq 3$, *not even for ready money.*[6]

Note that there is one potential problem. Consider the translation of the formula $\forall x \forall y_1 \forall y_2 \ldots \forall y_k \alpha$; it is $\Pi x \Pi y_1 \Pi y_2 \ldots \Pi y_k \alpha$, and so the related polynomial in x is of degree 2^k. To keep the degree of the polynomials small, we show that any QBF can be put into an equivalent *simple form* which means that between any variable and its quantifier there is at most one universal quantifier.

EXERCISE 3.18. Show that any QBF can be put in simple form in polytime. Show that the translations of a simple formula of length n are such that the degree of the polynomials are always bounded by $2n$.

Finally, observe that false negatives occur with probability 0, since if α is true, then all that P has to do is behave well, and communicate honestly, and V will accept.

[6]Oscar Wilde's *The importance of being Earnest*, Act 1.

On the other hand, if α is false, then the only sensitive moment is when P sends the polynomial poly_{α_1} (the $\forall$ and $\exists$ cases). It could happen that the polynomial is incorrect, but nevertheless $\text{value}(p_{\alpha_1}(r)) \equiv_k \text{poly}_{\alpha_1}(r)$, i.e., r is a root of the polynomial $\text{value}(p_{\alpha_1}(x)) - \text{poly}_{\alpha_1}(x)$ over $\mathbb{Z}_k$. (Here is where we have made use of the fact that k is a prime, so $\mathbb{Z}_k$ is a field.) Since $k \geq 2^n$ and the degree of the polynomial is at most $2n$, by the Fundamental theorem of Algebra it follows that the probability of this taking place is at most $\frac{2n}{2^n}$, and hence small for large n.

3.4. Answers to selected exercises

Exercise 3.5. Order the configurations lexicographically, so they are numbered $1, 2, \ldots, p(n)$, where p is a polynomial. Construct the adjacency matrix of the graph by putting a 1 in position (i, j) if configuration C_i yields configuration C_j. Note that a logspace bounded TM can still work with a polynomial size configuration graph: the machine need not keep a copy of the entire configuration graph; the graph can be given implicitly by M and x, and whenever the machine needs to establish whether (C_1, C_2) is an edge, it can do so on the basis of M and x, without having to examine the entire graph. Furthermore, it can check whether there is a path from C_{init} to C_{accept} in polynomially many steps.

Exercise 3.18. Using de Morgan laws we can move (in logspace) all the negations to be in front of variables (and cancel double negations, $\neg\neg x = x$). Now we want to transform the formula so that between any variable and its point of quantification, there is at most one universal quantifier. We do this from the outside in, larger subformulas first, introducing a lot of new dummy variables. For every subformula of the form $\forall x \alpha(x, \vec{y})$, where the variables $\vec{y}$ are free, we introduce a new set of variables $\vec{y}'$ and transform the formula to be $\forall x[\exists \vec{y}' \bigwedge_i (y_i \leftrightarrow y_i') \wedge \alpha(x, \vec{y}')]$. Note that the $\vec{y}'$ are quantified *inside* the $\forall x$, and the $\vec{y}$ occur exactly once and just *inside* the $\forall x$.

3.5. Notes

Exercise 3.2 is based on [**Koz06**, chapter 5]. For more details on the proof of theorem 3.12 see [**Pap94**, theorem 7.6]. For more details on the proof of $\mathsf{IP} = \mathsf{PSPACE}$ see [**Koz06**, lectures 16 and 17] and [**SP95**, chapter 21]. Also [**Bab90**] contains an interesting account of the discovery of this result.

The fact that $\mathsf{NSPACE}(n)$ turned out to be equal to $\mathsf{co\text{-}NSPACE}(n)$ is also interesting for the following reason: the class of context-free languages, CFL, is the class of languages decidable by grammars whose rules are of the form $X \to \alpha$, that is, a variable yields an α, where α is a string of variables and terminals. The class of context-sensitive languages, CSL, is the class of languages decidable by grammars whose rules are of the form $\alpha \to \beta$, with the restriction that $|\alpha| \leq |\beta|$; it was known that $\mathsf{CSL} = \mathsf{LBA} = \mathsf{NSPACE}(n)$, and since we know that CFL are not closed under complementation, it was natural to ask whether CSL are closed under complementation—now we know that they are. Finally, if we put no restrictions on the rules of the grammar, i.e., if we allow all $\alpha \to \beta$, where α, β are free to be any strings of variables

and terminals, then we obtain the so called *Semi-Thue system*, also known as a *string rewriting system*. Semi-Thue systems capture the same languages as general TMs; see §1.1 for the different, yet equivalent, models of computation.

4

Diagonalization and Relativization

4.1. Hierarchy theorems

We say that a TM M *recognizes* a language L if $L = L(M)$ (i.e., M halts and accepts all $x \in L$, and M may reject or not halt on $x \notin L$). We say that a language L is *recursively enumerable* if it is recognized by some M. Let RE be the class of recursively enumerable languages.

A TM M is called a *decider* if it halts on every input. We say that a TM M *decides* a language L if $L = L(M)$ and M is a decider (i.e., M recognizes L, and, furthermore, M halts on all inputs—in the accepting state if $x \in L$, and in the rejecting state if $x \notin L$). A language L is called *decidable* (or *recursive*) if there exists a decider M such that $L = L(M)$. Let Rec be the class of recursive (i.e., decidable) languages.[1]

It is well known that the language

$$\mathsf{A}_{\mathrm{TM}} = \{\langle M, x\rangle | M \text{ accepts } x\}$$

is recursively enumerable, but not recursive. A_{TM} is clearly in RE; we use the *diagonal argument* to show that it is not recursive.[2]

THEOREM 4.1. A_{TM} is not recursive.

PROOF. Suppose $M_{\mathsf{A}_{\mathrm{TM}}}$ decides A_{TM}. Define D, the "diagonal" machine, as follows: on input $\langle M\rangle$, D first simulates $M_{\mathsf{A}_{\mathrm{TM}}}$ on $\langle M, \langle M\rangle\rangle$, until $M_{\mathsf{A}_{\mathrm{TM}}}$ is about to halt ($M_{\mathsf{A}_{\mathrm{TM}}}$ is a decider, so by assumption it always halts). If $M_{\mathsf{A}_{\mathrm{TM}}}$ is about to accept, D rejects. If $M_{\mathsf{A}_{\mathrm{TM}}}$ is about to reject, then D accepts.

Consider $D(\langle D\rangle)$. If $D(\langle D\rangle)$ rejects, then $M_{\mathsf{A}_{\mathrm{TM}}}$ must accept $\langle D, \langle D\rangle\rangle$, by definition of D, so $D(\langle D\rangle)$ must accept by definition of $M_{\mathsf{A}_{\mathrm{TM}}}$. If $D(\langle D\rangle)$ accepts, then

[1] Note that all complexity classes are assumed to be contained in Rec, i.e., if $\mathcal{C}$ is a complexity class, then $\mathcal{C} \subseteq \mathsf{Rec}$. For time-bounded computations this follows by definition; for space-bounded computations, we can always modify a space-bounded TM M to be a decider M'. This is so because we can always add a counter which checks that the number of moves does not exceed the number of possible configurations, and halt and reject when it does. Such a counter need not be longer than the space available to the machine.

[2] Note that it follows directly from Cantor's original diagonal argument that there are undecidable languages (in fact that there are not recursively enumerable languages): the set of all TMs has the same cardinality as $\mathbb{N}$, and the set of all languages has the same cardinality as $\mathcal{P}(\mathbb{N})$, the power set of $\mathbb{N}$, and by Cantor's diagonal argument $\mathbb{N} < \mathcal{P}(\mathbb{N})$, so there are languages which do not have a TM. However, theorem 4.1 says more than that; it gives a concrete example of an undecidable language, i.e., A_{TM}, which is recursively enumerable, and in fact complete for the class of recursively enumerable languages.

$M_{\mathsf{A}_{\mathrm{TM}}}$ must reject, by definition of D, so by definition of $M_{\mathsf{A}_{\mathrm{TM}}}$ $D(\langle D\rangle)$ rejects. Thus, $D(\langle D\rangle)$ accepts iff it does not; contradiction. □

We shall now use a slightly modified version of this diagonal argument to show some strong separations between complexity classes; these results go under the name of Hierarchy theorems. Recall the definition of "little-oh" (see page 11), and of space constructible functions (see page 37).

THEOREM 4.2 (Space Hierarchy). Given any space constructible function f, a language A exists that is decidable in space $O(f(n))$ but not in $o(f(n))$.

PROOF. Let D be a deterministic TM which on input $\langle M\rangle\#^m$, $m \geq 0$, and M is a single-tape TM, simulates $M(\langle M\rangle)$ within space bound $f(n)$, $n = |\langle M\rangle\#^m| = |\langle M\rangle| + m$. The suffix of "#" works as padding; its purpose will be clear later. D works as follows.

- If M halts within the alloted space $f(n)$, then D accepts iff M rejects (i.e., D "does the opposite").
- If M does not halt (but stays throughout in the alloted space), D rejects. In order to accomplish this, D keeps a counter of the number of moves of M, and it rejects when the counter reaches $2^{cf(n)}$, for some constant c depending on M. The number $2^{cf(n)}$ is the number of possible configurations of M on inputs of length n; this number requires $cf(n)$ bits to encode.
- If M requires more than $f(n)$ squares of space, then D rejects directly. Note that D keeps track of the space by placing a special symbol ♣ on the square number $f(n)$ of each work-tape, and if the simulation ever wants to move right of this symbol, the computation ends and D rejects. Since f is space constructible, computing which is the $f(n)$-th square can be done in space $c'f(n)$, for some constant c'.

Clearly, D runs in space $\max\{1, c, c'\} \cdot f(n)$, i.e., in space $O(f(n))$.

We argue that the language $L(D)$ cannot be decided in space $o(f(n))$. Suppose that N runs in space $g(n) \in o(f(n))$ (if N is a k-tape TM we can convert it into a single-tape TM which runs in space $kg(n)$, which is still in $o(f(n))$), so $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$, and so for any $\varepsilon > 0$ there exists an n_0 such that for all $n \geq n_0$, $g(n) < \varepsilon f(n)$. Thus, given $\langle N\rangle\#^{n_0}$ as input, D will simulate $N(\langle N\rangle)$ within space

$$g(|\langle N\rangle| + n_0) < \varepsilon f(|\langle N\rangle| + n_0),$$

and so, by selecting $\varepsilon < 1$, we ensure that D and N differ on $\langle N\rangle\#^{n_0}$, and thus $L(N) \neq L$. Note that $|\langle N\rangle|$ could be small, so the padding $\#^{n_0}$ ensures that the input is of sufficient length.

Finally, note that there is a certain (small) overhead associated with the simulation. We can compensate for this by selecting ε suitably small. □

COROLLARY 4.3. For $\varepsilon_1 < \varepsilon_2$ in $\mathbb{Q} \cap [0, \infty)$, $\mathsf{SPACE}(n^{\varepsilon_1}) \subset \mathsf{SPACE}(n^{\varepsilon_2})$.

COROLLARY 4.4. $\mathsf{NL} \subset \mathsf{PSPACE}$.

PROOF. From Savitch's theorem (theorem 3.7) we know that NL is contained in SPACE($\log^2 n$), and from the Space Hierarchy theorem we know that SPACE($\log^2 n$) $\subset$ SPACE(n). □

COROLLARY 4.5. QSAT $\notin$ NL.

PROOF. By corollary 4.4, NL $\subset$ PSPACE, and QSAT is complete for PSPACE by theorem 3.11. □

COROLLARY 4.6. PSPACE $\subset$ EXPSPACE.

PROOF. For all constant k, SPACE(n^k) $\subseteq$ SPACE($n^{\log n}$) $\subset$ SPACE(2^n). □

We can now give an example of a *provably intractable* language. Recall that regular expressions are built up from $\emptyset, \varepsilon, a \in \Sigma$, by using the regular operations $\cup, \cdot, *$ (union, concatenation, and Kleene's star, respectively). Then, $\emptyset, \varepsilon, a \in \Sigma$ are regular expressions, and if R, S are regular expressions, then so are $R \cup S, R \cdot S, R^*, (R)$. The language of a regular expression R, denoted $L(R)$, is defined inductively as follows:

$$\begin{aligned}
&L(\emptyset) = \emptyset, L(\varepsilon) = \{\varepsilon\}, L(a) = \{a\},\\
&L(R \cup S) = L(R) \cup L(S),\\
&L(R \cdot S) = \{xy | x \in L(R) \ \texttt{\&} \ y \in L(S)\},\\
&L(R^*) = \{x_1 x_2 \dots x_n | n \geq 0, x_i \in L(R)\}.
\end{aligned}$$

The language

$$\mathsf{EQ}_{\text{rex}} = \{\langle R, S\rangle | R, S \text{ are regular expressions and } L(R) = L(S)\}$$

is in PSPACE.

If we also introduce the exponentiation operator $R \uparrow k = R \cdot R \cdot \dots \cdot R$ (k-times), and let $\mathsf{EQ}_{\text{rex}\uparrow}$ be the corresponding language, then we have the following theorem.

THEOREM 4.7. $\mathsf{EQ}_{\text{rex}\uparrow}$ is EXPSPACE-complete.

COROLLARY 4.8. $\mathsf{EQ}_{\text{rex}\uparrow}$ is intractable.

PROOF. If it were in P, then it would certainly be in PSPACE, but by completeness it would follow that EXPSPACE $\subseteq$ PSPACE, contradicting the separation we obtained from the Space Hierarchy theorem. □

A function $t : \mathbb{N} \longrightarrow \mathbb{N}$, where $t(n)$ is at least $O(n \log n)$, is *time constructible* if the function that maps any $x \in \{0,1\}^n$ to the binary representation of $t(n)$ is computable in time $O(t(n))$.

THEOREM 4.9 (Time Hierarchy). Given a time constructible function t, there exists a language A that is decidable in time $O(t(n))$ but not in time $o(t(n)/\log t(n))$.

EXERCISE 4.10. Prove the Time Hierarchy theorem.

COROLLARY 4.11. P $\subset$ EXPTIME.

We can also prove Hierarchy theorems for nondeterministic classes. This is a little bit more tricky, since it is not clear how to "flip" the result of the computation efficiently in the nondeterministic case. We demonstrate the main idea with concrete nondeterministic time classes, and then leave the general result as an exercise.

THEOREM 4.12. $\mathsf{NTIME}(n) \subset \mathsf{NTIME}(n^{1.5})$.

PROOF. Let $f : \mathbb{N} \longrightarrow \mathbb{N}$ be defined as follows:

$$f(1) = 2$$
$$f(i+1) = 2^{f(i)^{1.2}}.$$

D will flip the answer of M_i on *some* input in $\{1^n | f(i) < n \leq f(i+1)\}$.

First, assume once again that we have an enumeration $M_1, M_2, M_3, \ldots$ of all TMs such that each TM M appears infinitely often. Thus, given any M, and given any i_0, we can always find an $i \geq i_0$ such that $M = M_i$. Assume also that the degree of non-determinism is bounded *for the entire list*, so it is, say, 2.

Define the diagonal machine D as follows: on input $x = 1^n$ (if $x \notin \{1\}^*$, reject outright), compute an i such that $f(i) < n \leq f(i+1)$ (note that this must, and can, be done in time $O(n^{1.5})$). The machine D now considers two cases.

Case 1. $f(i) < n < f(i+1)$ (i.e., n is strictly in between). Then, D simulates M_i on input 1^{n+1}, using nondeterminism, in time $n^{1.1}$ (if M_i does not halt within this time, D simply accepts) and outputs the same answer.

Case 2. $n = f(i+1)$, then D accepts 1^n iff M_i rejects $1^{f(i)+1}$ in time $(f(i)+1)^{1.1}$. For D to know that M_i rejects, it must go through $2^{(f(i)+1)^{1.1}}$-many branches of M_i on $1^{f(i)+1}$. But this is doable, since the input size is $n = f(i+1) = 2^{f(i)^{1.2}}$.

Thus D is a nondeterministic machine that runs in time $O(n^{1.5})$. We want to show now that $L(D) \notin \mathsf{NTIME}(n)$. Suppose that it is, so we can find an i large enough so that M_i decides $L(D)$ in time $O(n)$ and on inputs of length $n \geq f(i)$, M_i can be simulated in less than $n^{1.1}$ many steps.

We know that for all $f(i) < n < f(i+1)$, $D(1^n) = M_i(1^{n+1})$, and we also know that $D(1^{f(i+1)}) \neq M_i(1^{f(i)+1})$. Together with the fact that $D(x) = M_i(x)$ for all x, this is an untenable situation.

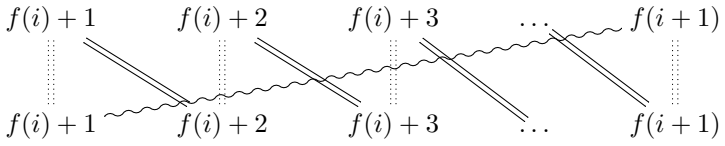


FIGURE 1. All these equalities cannot co-exist with $f(i+1) \neq f(i) + 1$.

The first row of figure 1 consists of the values of D on strings of as many 1s as the given number. The second row represents M_i. Two solid lines represent equality. By transitivity all corresponding values are equal, but then we have a pair that are not; this is not possible. □

4.2. Oracles and Relativization

An *oracle* is a language O, and an *oracle TM* M^O is an ordinary TM with an extra tape called the *oracle tape*. M^O writes a string x on the oracle tape, queries the oracle by entering a special "oracle query state," and gets an answer in one step, i.e., the oracle O writes a 0 or a 1 on the first square of the tape denoting that $x \notin O$ or $x \in O$, respectively. Note that this definition can be extended in the natural way to oracles that output strings. Let P^O and NP^O be the set of languages decidable with polytime deterministic (respectively, nondeterministic) TMs with an oracle for O.

For example, suppose that the oracle O is SAT. Then P^{SAT} is the class of languages decidable by polytime TMs which can query SAT. Note that $\mathsf{NP} \subseteq \mathsf{P}^{\text{SAT}}$, and also $\mathsf{co\text{-}NP} \subseteq \mathsf{P}^{\text{SAT}}$. This is because an oracle for SAT can be used to answer queries about TAUT; to check that $\phi \in$ TAUT just check that $\neg\phi \notin$ SAT. In general, if $L_1 = L(M^{L_2})$, then we say that L_1 is *Turing-reducible* to L_2. Note that the power of the reduction depends on M; so for example, if M were a polytime TM, L_1 would be polytime Turing-reducible to L_2, denoted $\leq_{\mathsf{P}}^T$, where the "T" stands for TM, just like the "m" in $\leq_{\mathsf{P}}^m$ stands for "many-one," and P stands for polytime in both cases.

Turing reductions allow to query a language repeatedly, using the answers to the queries at will. On the other hand, many-one reductions allow exactly one query, and the machine has to answer with the query answer. Therefore, TAUT $\leq_{\mathsf{P}}^T$ SAT, but it is a big open question whether TAUT $\leq_{\mathsf{P}}^m$ SAT, underlying the difference of the two reductions.

Consider the language

$$\text{MINFORMULA} = \{\langle\phi\rangle : \forall\psi[|\psi| < |\phi| \to \psi \not\leftrightarrow \phi]\},$$

i.e., MINFORMULA is the language of encodings of those Boolean formulas for which there does not exist a smaller Boolean formula that computes the same Boolean function; MINFORMULA $\in \mathsf{co\text{-}NP}^{\text{SAT}}$. To see this, note that a $\mathsf{co\text{-}NP}$ machine can examine all ψ such that $|\psi| < |\phi|$, and for each ψ check (using an oracle for SAT) that $\neg(\psi \leftrightarrow \phi)$ is satisfiable.

It may appear strange that NP^{SAT} is believed to be a larger class than NP, but note that in NP^{SAT}, we are allowed to query the oracle repeatedly, and take into account negative answers—this *is more* than is allowed by many-one reductions.

The diagonal argument presented in this chapter appears very powerful; recall the Hierarchy theorems and their consequences. Is it also possible to show that $\mathsf{P} \neq \mathsf{NP}$ with a diagonal argument? This question is somewhat vague, since we have not defined precisely the nature of a "diagonal argument." Still, we can make the observation that diagonal arguments *relativize*. By this we mean that if we can prove the separation of two complexity classes $\mathcal{C}_1$ and $\mathcal{C}_2$, defined in terms of TMs with bounds on resources, then the same argument should go through for showing the separation of $\mathcal{C}_1^O$ and $\mathcal{C}_2^O$, for any oracle O.

Therefore, if we can prove that there exist two oracles A, B such that $\mathcal{C}_1^A = \mathcal{C}_2^A$ and $\mathcal{C}_1^B \neq \mathcal{C}_2^B$, then we can take that as evidence (with a "grain of salt") that a diagonal argument will not work to show the separation of $\mathcal{C}_1$ and $\mathcal{C}_2$.

In the next theorem we show the existence of an oracle A such that $\mathsf{P}^A \neq \mathsf{NP}^A$; this is a very nice construction, but it implies that the P versus NP question will likely not be settled by a diagonal argument: *"A bliss in proof, and proved, a very woe"*.[3]

THEOREM 4.13. There exist oracles A, B such that: $\mathsf{P}^A \neq \mathsf{NP}^A$ and $\mathsf{P}^B = \mathsf{NP}^B$.

PROOF. For the second claim let $B = \text{QSAT}$. Trivially, $\mathsf{P}^{\text{QSAT}} \subseteq \mathsf{NP}^{\text{QSAT}}$, and

$$\mathsf{NP}^{\text{QSAT}} \overset{(1)}{\subseteq} \mathsf{NPSPACE} \overset{(2)}{\subseteq} \mathsf{PSPACE} \overset{(3)}{\subseteq} \mathsf{P}^{\text{QSAT}}.$$

For (1) note that QSAT is in PSPACE, so whenever the NP machine wants to query the QSAT oracle, we simply do the computation directly in PSPACE. For (2) we use corollary 3.10, and for (3) we use the fact that QSAT is PSPACE-complete: theorem 3.11.

For the first claim, given any oracle A, define L_A as follows:

$$\{w | \exists x, x \in A \text{ and } |w| = |x|\}.$$

Clearly, for any A we have $L_A \in \mathsf{NP}^A$: on input w, guess a x such that $|w| = |x|$, and check that $x \in A$. We now construct an A which ensures that $L_A \notin \mathsf{P}^A$.

Let $M_1^{\sqcup}, M_2^{\sqcup}, M_3^{\sqcup}, \ldots$, be a list of polytime oracle TMs. The symbol "$\sqcup$" denotes that these are TMs with a slot for an oracle (think of it as a PCI slot), but without an actual oracle attached (or the empty oracle, i.e., $O = \emptyset$, so the answer to each query is "no"). We make sure that $M_i^{\sqcup}$ runs in time n^i by attaching a counter. We build A and $\hat{A}$ ($\hat{A} \subseteq \overline{A}$) simultaneously in stages to ensure at stage i that $L(M_i^A) \neq L_A$. In the first stage, **stage** 0, we let $A, \hat{A} := \emptyset$.

At **stage** i A and $\hat{A}$ are finite, so we pick an n such that $n > |w|$ for all $w \in A \cup \hat{A}$, and $2^n > n^i$ (where recall that n^i is the running time of $M_i^{\sqcup}$). We now extend A in such a way so that $L(M_i^A) \neq L_A$.

Run $M_i^{\sqcup}(1^n)$: each time M_i queries the oracle with a string y, if the membership of y in A has been determined (i.e., it has been established at an earlier stage whether y is in A or $\hat{A}$), we do nothing to A or $\hat{A}$, and we answer the query consistently with what has been decided earlier. Look at this as intercepting the query to the empty slot, and answering according to the scheme presented here. (Note that because of the way we have chosen n, all the strings whose status has been determined are shorter than n.)

If y's status is undetermined, we answer "no" (i.e., declare $y \notin A$, and thereby $y \in \hat{A}$). If at the end $M_i^{\sqcup}(1^n) =$ "yes," declare all remaining y, $|y| = n$, not to be in A (and thereby in $\hat{A}$). If, on the other hand, $M_i^{\sqcup}(1^n) =$ "no," find a y_0 such that $|y_0| = n$, and y_0 has not been queried (it must exist, because in n^i time $M_i^{\sqcup}$ cannot query all $2^n > n^i$ y's of length n), and put y_0 in A.

Therefore, $M_i^A(1^n) =$ "yes" iff $1^n \notin L_A$. We still need to determine the status of all the y's of length $\leq n$ that have not been considered; say we declare all of them not in A, i.e., in $\hat{A}$. □

THEOREM 4.14. There exist oracles A, B: $\mathsf{NP}^A \neq \text{co-}\mathsf{NP}^A$ and $\mathsf{NP}^B = \text{co-}\mathsf{NP}^B$.

[3] Shakespeare, Sonnet CXXIX.

EXERCISE 4.15. Prove theorem 4.14.

4.2.1. A random oracle separating P and NP. In this section we are going to show that for a random oracle A, we have that $\mathsf{P}^A \neq \mathsf{NP}^A$ with probability 1.

Suppose that we generate a random oracle A by tossing a coin: for every string $x \in \{0,1\}^*$ we toss a coin to determine if $x \in A$. In other words, $\forall x$, $\Pr[x \in A] = \frac{1}{2}$. Then, for such an A, $\Pr[\mathsf{P}^A \neq \mathsf{NP}^A] = 1$, which means that there must exist an oracle separating P and NP (something that we know already from theorem 4.13), and in fact "*most*" oracles separate P and NP.

Just as in the proof of theorem 4.13, let $M_1^{\sqcup}, M_2^{\sqcup}, M_3^{\sqcup}, \ldots$ be a list of polytime oracle TMs. For any given oracle A, $\mathsf{P}^A = \{L(M_i^A) | i \geq 1\}$.

Given a random oracle A, define the language $L(A)$ as follows: arrange all strings in $\{0,1\}^*$ by length and then in lexicographic order. Given an x, $|x| = n$, to determine if $x \in L(A)$, look at the segment of $n2^n$ strings that follow x (in this ordering). Imagine this segment consisting of 2^n blocks, of n strings each. Then, $x \in L(A)$ iff there is at least one such block of n strings all of which are in A.

$$0, 1, 00, 01, \ldots, x = x_i, \underbrace{\underbrace{\ldots\ldots}_{n}, \underbrace{\ldots\ldots}_{n}, \ldots, \underbrace{\ldots\ldots}_{n}}_{2^n}, x_{i+(n2^n+1)}, \ldots$$

Note that for any oracle A, $L(A) \in \mathsf{NP}^A$: the certificate for $x \in L(A)$ is a number $i \leq 2^n$ (giving the index of the block containing only 1s; note that i can be encoded with polynomially many bits in binary), and we verify those n numbers with the oracle A.

Clearly, $\Pr[\mathsf{P}^A = \mathsf{NP}^A] \leq \Pr[L(A) \in \mathsf{P}^A]$, and:

$$\begin{aligned}
&\Pr[L(A) \in \mathsf{P}^A] && (4)\\
=&\Pr[\exists i, L(M_i^A) = L(A)] && (5)\\
\leq&\sum_i \Pr[\forall x, x \in L(M_i^A) \square L(A)] && (6)\\
\leq&\sum_i \Pr[\forall j, x_j \in L(M_i^A) \square L(A)] && (7)\\
=&\sum_i \underbrace{\prod_j \underbrace{\Pr[x_j \in L(M_i^A) \square L(A) \quad | \quad \underbrace{\forall k < j, x_k \in L(M_i^A) \square L(A)}_{(**)}]}_{(*)}}_{} , && (8)
\end{aligned}$$

where the "$\square$" introduced in equation (6) denotes the following operation on sets: $X \square Y = \{w | w \in X \iff w \in Y\}$ (i.e., $X \square Y = \overline{X \triangle Y}$, the complement of the symmetric difference). The x_j's introduced in equation (7) are any subsequence of our ordering of $\{0,1\}^*$, and in equation (8) we introduce the conditional probability, i.e., $\Pr[A|B] = \Pr[A \text{ and } B] / \Pr[B]$, which when it "telescopes" with the $\prod_j$ is equal to the previous line.

Now we are concerned with finding an appropriate subsequence $\{x_j\}$ so that we can bound $(*)$ in equation (8) by 0.9. This will have the desirable effect that $\sum_{i=1}^{\infty}\prod_{j=1}^{\infty} 0.9 = \sum_{i=1}^{\infty} 0 = 0$.

We want to choose the x_j's very far apart from each other, say $|x_{j+1}| > 2^{|x_j|}$, so that the events "$x_j \in L(A)$" are independent (because when the x_j's are this far apart, their respective blocks, used to establish if they are in $L(A)$ or not, do not intersect). It is not possible to make the events "$x_j \in L(M_i^A)$" completely independent, since nothing stops M_i^A from making small queries; but we can make them sufficiently independent by noting that M_i^A cannot make *huge* queries and cannot make *too many* queries.

Consider the following table of probabilities, where events E_1, E_2, E_3, E_4, are taking place under the condition of $(**)$ from equation (8).

	$x_j \in L(M_i^A)$	$x_j \notin L(M_i^A)$
$x_j \in L(A)$	E_1	E_2
$x_j \notin L(A)$	E_3	E_4

We want to show that $\Pr[E_2 \text{ or } E_3]$ is at least 0.1, and so that $(*) \leq 0.9$ (in equation (8)) as desired.

$$\begin{aligned} \Pr[E_1 \text{ or } E_2] &= \Pr[x_j \in L(A) | (**)] = \Pr[x_j \in L(A)] > 0.6, \\ \Pr[E_3 \text{ or } E_4] &= \Pr[x_j \notin L(A) | (**)] = \Pr[x_j \notin L(A)] > 0.3. \end{aligned} \tag{9}$$

First note that we can get rid of the condition $(**)$ since the events "$x_j \in L(A)$" are independent. Then, the probability that a block contains only strings in A is $\frac{1}{2^n}$, so the probability that some block contains only strings in A is $1 - (1 - \frac{1}{2^n})^{2^n}$. Note that $\lim_{m\to\infty}(1 - \frac{1}{m})^m = \frac{1}{e} = 0.367\ldots > 0.3$.

The expression

$$\frac{\Pr[E_2]}{\Pr[E_2] + \Pr[E_4]} \tag{10}$$

represents the probability that $x_j \in L(A)$ while $x_j \notin L(M_i^A)$, all under the assumption $(**)$, i.e., $\Pr[x_j \in L(A) | x_j \notin L(M_i^A), (**)]$, and since "$x_j \in L(A)$" is independent of $(**)$, we conclude $(10) = \Pr[x_j \in L(A) | x_j \notin L(M_i^A)]$.

Now, during the computation of M_i^A on x_j, M_i^A can query at most polynomially many strings of the oracle, and therefore, it can query at most polynomially many strings from the $2^{|x_j|}$ blocks that follow x_j. So consider those blocks where there is a string that is queried by M_i^A to be unavailable for the random experiment that computes the probability of $x_j \in L(A)$. But, as $|x_j|$ grows, we can assume that the polynomial bounding the number of oracle queries of M_i^A is less than $2^{|x_j|}/2$. Thus,

$$(10) = \Pr[x_j \in L(A) | x_j \notin L(M_i^A)] \geq 1 - (1 - 1/2^{|x_j|})^{2^{|x_j|}/2} = 1 - \frac{1}{\sqrt{e}} \geq \frac{1}{3},$$

since the expression $(1 - 1/2^{|x_j|})^{2^{|x_j|}/2}$ converges to $\sqrt{1/e}$ as $|x_j|$ grows.

Consider now two cases. If $\Pr[E_3] \geq 0.1$ then we are done with our quest to show that $\Pr[E_2] + \Pr[E_3] \geq 0.1$. If, on the other hand $\Pr[E_3] < 0.1$, then, since $\Pr[E_3] + \Pr[E_4] > 0.3$ (by (9)), we have that $\Pr[E_4] > 0.2$. Since, $(10) \geq 1/3$, it follows that $\Pr[E_2] > 0.1$.

In either case, $\Pr[E_2] + \Pr[E_3] \geq 0.1$.

EXERCISE 4.16. Show that for a random oracle A, $\Pr[\mathsf{NP}^A \neq \mathsf{co\text{-}NP}^A] = 1$.

The results in this section are related to the *Random Oracle Hypothesis* (ROH). The ROH states that if two complexity classes are equal with respect to a random oracle with probability 1, then they are equal; ROH has been shown to be false—see [**CCG+94**].

4.3. Polynomial time hierarchy

Let $\Sigma_0\mathsf{P} = \Pi_0\mathsf{P} = \mathsf{P}$, and let $\Sigma_{i+1}\mathsf{P} = \mathsf{NP}^{\Sigma_i\mathsf{P}}$, and $\Pi_{i+1}\mathsf{P} = \mathsf{co\text{-}NP}^{\Sigma_i\mathsf{P}}$. We define the *polytime hierarchy* as $\mathsf{PH} = \bigcup_i \Sigma_i\mathsf{P} = \bigcup_i \Pi_i\mathsf{P}$. Note that $\mathsf{NP} = \Sigma_1\mathsf{P}$ and $\mathsf{co\text{-}NP} = \Pi_1\mathsf{P}$, and the language MINFORMULA (defined on page 53) is in $\Pi_2\mathsf{P}$.

An *Alternating Turing Machine* (ATM) is a nondeterministic TM with an additional feature: all its states, except q_{accept} and q_{reject}, are divided into two groups: universal and existential. We give a (recursive) definition of what it means for a node in the configuration graph to be accepting: a leaf node is accepting (rejecting) if it corresponds to an accepting (rejecting) configuration. A universal (existential) node, is accepting if all (some) of its children are accepting. We determine acceptance by looking at the root node; we accept iff the root node is accepting.

Let Σ_i^{Alt} be the class of languages decidable by polytime ATMs where the initial state is existential and there are at most i "runs" of states on any computational branch (i.e., a sequences of existential states, followed by a sequence of universal states, then another sequence of existential states, etc., and there are at most i such sequences). Π_i^{Alt} is defined analogously, except the machines start in a universal state, and $\Sigma_0^{\text{Alt}} = \Pi_0^{\text{Alt}}$ denote the set of languages decidable by deterministic polytime machines.

Recall that Σ_i^p and Π_i^p were defined in §2.3 on page 32.

THEOREM 4.17. For all i, $\Sigma_i\mathsf{P} = \Sigma_i^{\text{Alt}} = \Sigma_i^p$, and the same holds for Π.

PROOF. We prove it by induction on i. The basis case is $i = 0$ and all three are equal to P (by definition). To show the inductive step assume the claim for i, and show it for $(i+1)$.

We show the following chain of containments:

$$\Sigma_{i+1}\mathsf{P} \overset{(1)}{\subseteq} \Sigma_{i+1}^{\text{Alt}} \overset{(2)}{\subseteq} \Sigma_{i+1}^p \overset{(3)}{\subseteq} \Sigma_{i+1}\mathsf{P}.$$

(1) Suppose $L \in \Sigma_{i+1}\mathsf{P}$. Then L is decided by an NP TM with a $\Sigma_i\mathsf{P}$ oracle, i.e., M^O. By IH this oracle O is in Σ_i^{Alt}. Consider the ATM N which simulates M as follows: First, N guesses at most polynomially many queries s_i, *together with* the answers to those queries, and for the "yes" queries special "witnessing" strings w_i. That is, N initially guesses:

$$\{(\langle s_1, w_1\rangle, \text{"yes"}), (s_2, \text{"no"}), \ldots, (\langle s_m, w_m\rangle, \text{"yes"})\},$$

and writes it all down. Second, N simulates M^O, but when M^O is about to query the oracle O for the i-th time, N intercepts this query and checks that the query is

indeed s_i. If this test passes, it responds with the answer it has guessed. The role of the witnesses w_i will become apparent in the checking stage.

Now N must verify that it guessed the correct answers to the queries. So N branches universally on all the m queries (polynomially many of them). If s_i is a "no" query, then N checks if $s_i \in \overline{O}$ (this is a Π_i^{Alt} language since oracle O is a Σ_i^{Alt} language).

If s_i is a "yes" query, N checks if $\langle s_i, w_i \rangle \in O'$, where O' is just like O, except it also takes as input the initial existential nondeterministic choices the machine deciding O (i.e., a Σ_i^{Alt} machine) makes, and thus O' is *de facto* a Π_{i-1}^{Alt} machine. Thus, the w_i are *witnesses* of what the machine for O must do on the initial existential run of the "yes" input s_i, in order to eventually accept.

Thus, N performs a $\Sigma_{i+1}^{\mathrm{Alt}}$ computation, and so $\Sigma_{i+1}\mathsf{P} \subseteq \Sigma_{i+1}^{\mathrm{Alt}}$.

(2) Suppose that L is decided by a $\Sigma_{i+1}^{\mathrm{Alt}}$ machine M. A computational path of M, on input x, can be described with a sequence of $(i+1)$ numbers in base d, where d is the degree of nondeterminism of M. For example, y_1, y_2, y_3 describe a sequence of $|y_1|$-many existential choices, followed by $|y_2|$-many universal choices, followed by $|y_3|$-many existential choices. Let $R(x, y_1, y_2, \ldots, y_{i+1})$ be a predicate which holds whenever the sequence of choices $y_1, y_2, \ldots, y_{i+1}$ on input x leads to an accepting leaf. Since M is polytime, R is polytime. Clearly,

$$M \text{ accepts } x \text{ iff } (\exists y_1)(\forall y_2)\cdots(Qy_{i+1})R(x, y_1, y_2, \ldots, y_{i+1})$$

(bounds on quantified variables omitted). This shows $\Sigma_{i+1}^{\mathrm{Alt}} \subseteq \Sigma_{i+1}^p$.

(3) Finally, show $\Sigma_{i+1}^p \subseteq \Sigma_{i+1}\mathsf{P}$. Suppose $L \in \Sigma_{i+1}^p$, so $x \in L \iff (\exists y_1)\alpha(y_1, x)$, where $\alpha(y_1, x)$ is a Π_i^p formula, so $\neg\alpha(y_1, x)$ is a Σ_i^p formula, and so by induction $\neg\alpha(y_1, x)$ has a $\Sigma_i\mathsf{P}$ oracle. Now decide L with a $\Sigma_{i+1}\mathsf{P}$ machine as follows: nondeterministically guess a b_1 (of polynomial length), query the $\Sigma_i\mathsf{P}$ oracle for $\neg\alpha(b_1, x)$, and accept iff the answer comes back "no."

For Π it follows by complementing all the classes in the equality given in the statement of the lemma. □

THEOREM 4.18. If $\mathsf{P} = \mathsf{NP}$, then PH *collapses* to the ground level, i.e., $\mathsf{PH} = \mathsf{P}$.

PROOF. It suffices to show that for any i, if $\Sigma_i\mathsf{P} = \Pi_i\mathsf{P}$, then $\Sigma_{i+1}\mathsf{P} = \Sigma_i\mathsf{P}$. By theorem 4.17 we can use any of the three formulations of PH. We show that if $\Sigma_i^p = \Pi_i^p$ then $\Sigma_{i+1}^p = \Sigma_i^p$. This is easy, since if $\Sigma_i^p = \Pi_i^p$, then we can "swap" two (opposite) quantifiers (Q_i and Q_{i-1}), and then we can combine the resulting duplicate quantifiers into one, bringing the number of alternations down by one. An inductive procedure now presents itself to finish the proof. □

It is not known whether PH is a sequence of properly contained classes, but it is believed to be so. Let

$$\mathrm{QSAT}_i = \{\psi | \psi = \exists\vec{y}_1 \forall\vec{y}_2 \ldots Q\vec{y}_i \phi(\vec{x}, \vec{y}_1, \vec{y}_2, \ldots, \vec{y}_i) \text{ is satisfiable}\}$$

where ϕ is quantifier free, $Q\vec{z}$ ($Q \in \{\exists, \forall\}$) denotes a vector of variables, i.e., $Q\vec{z}$ is $Qz_1Qz_2 \ldots Qz_n$.

THEOREM 4.19. QSAT_i is $\Sigma_i\mathsf{P}$-complete.

Clearly $\mathsf{PH} \subseteq \mathsf{PSPACE}$, and the containment appears to be proper. Since PH is not believed to collapse at any level, it is not believed to have complete problems, since a complete problem for PH would have to be at some level of PH, and then everything above that level would collapse to it.

The *arithmetical hierarchy* (AH) is defined analogously to PH. Recall that Rec and RE are the classes of recursive and recursively enumerable languages, respectively (see page 49). Then, $\Sigma_0\mathsf{Rec} = \Pi_0\mathsf{Rec} = \mathsf{Rec}$, and $\Sigma_{i+1}\mathsf{Rec} = \mathsf{RE}^{\Sigma_i\mathsf{Rec}}$. Unlike PH, AH provably does not collapse at any level.

THEOREM 4.20. For all i, $\Sigma_{i+1}\mathsf{Rec} \neq \Sigma_i\mathsf{Rec}$.

PROOF. The classes $\Sigma_1\mathsf{Rec} = \mathsf{RE}$ and $\Sigma_0\mathsf{Rec} = \mathsf{Rec}$ were separated by diagonalization in theorem 4.1. The same diagonal argument relativizes to higher levels of the hierarchy. □

In this paragraph we want to convey some intuition about AH, without developing too much logic; see [**BM77**] for the necessary background.[4] Let $\mathcal{L}_A = [0, s, +, \cdot; =, \leq]$ be the standard first order language of arithmetic, containing the constant zero, the successor function, plus, times, equality, and less-than-or-equal. Here is an example of a formula over $\mathcal{L}_A$:

$$(s0 < x) \wedge (\forall y \leq x)(\forall z \leq x)(y \cdot z = x \rightarrow (y = s0 \vee z = s0)), \tag{11}$$

where

$$\begin{array}{lll} s0 < x & \text{abbreviates} & s0 \leq x \wedge \neg(s0 = x), \\ \forall y \leq x\alpha & \text{abbreviates} & \forall y(y \leq x \rightarrow \alpha), \\ \exists y \leq x\alpha & \text{abbreviates} & \exists y(y \leq x \wedge \alpha). \end{array}$$

Let Δ_0 denote the set of formulas where all quantifiers are bounded; (11) is an example of a Δ_0 formula. Let Σ_i^{A} be the set of formulas which are of the form $\exists y_1 \forall y_2 \ldots Q y_i \alpha$, where α is a Δ_0 formula.

We say that a relation $R(x)$ over $\mathbb{N}$ (where numbers are represented in binary) is arithmetical if it can be represented as a formula $A(x)$ over $\mathcal{L}_A$. For example, the relation $\mathrm{Prime}(x)$, which is true iff x is prime, is arithmetical as it can be represented by (11). It should come as no surprise (although it is not trivial to prove) that RE relations can be represented with Σ_1^{A} formulas (but note that $\Sigma_0^{\mathrm{A}} = \Delta_0$ relations are a proper subset of Rec; however, Δ_0 does correspond to LTH, the linear time hierarchy—see theorem 4.26). In general, for $i \geq 1$, $\Sigma_i\mathsf{Rec}$ languages can be given by Σ_i^{A} relations.

While the AH certainly contains all the complexity classes mentioned in these notes, and much more (all recursive languages, and recursively enumerable, etc.), it is not all powerful.

THEOREM 4.21 (Tarski). Let TA be the set of sentences over $\mathcal{L}_A$ which are true in the standard model of arithmetic (i.e., TA is the set of all the theorems of number theory). Clearly, TA can be seen as a language when every such theorem is encoded in some standard way as a string over $\{0,1\}^*$. Then, $\mathrm{TA} \notin \mathsf{AH}$.

[4] Or Stephen Cook's logic notes [**Coo08**].

See, for example, [**BM77**] for a proof of this theorem.

Note that we could have defined $\Sigma_i\mathsf{Rec}$, for $i \geq 1$, with Σ_i^{A} relations; this is a *machine independent* definition of a complexity class. We have thus surreptitiously come into contact with a beautiful area of complexity, known as *Descriptive Complexity*. In Descriptive Complexity we are concerned with the richness of a language necessary to express the computational complexity of certain concepts. One of the first results in the area is Fagin's theorem which shows that NP is precisely the set of languages expressible by second-order existential formulas; a great source for this field is [**Imm99**].

4.4. More on Alternating TMs

Define $\mathsf{ATIME}(f(n))$ and $\mathsf{ASPACE}(f(n))$ to be time and space bounded by $O(f(n))$ on an ATM, without a bound on the number of alternations. Let $\mathsf{AP}, \mathsf{APSPACE}, \mathsf{AL}$ be defined analogously to their non-alternating versions. Note that $\mathsf{PH} \subseteq \mathsf{AP}$.

THEOREM 4.22. For $f(n) \geq n$ we have

$$\mathsf{ATIME}(f(n)) \overset{(1)}{\subseteq} \mathsf{SPACE}(f(n)) \overset{(2)}{\subseteq} \mathsf{ATIME}(f^2(n))$$

and for $f(n) \geq \log n$ we have

$$\mathsf{ASPACE}(f(n)) \overset{(3)}{=} \mathsf{TIME}(2^{O(f(n))}).$$

PROOF. **(1)** Depth first search of the ATM's computation tree, not recording configurations, but only the nondeterministic choice (in base $b =$ degree of nondeterminism).

(2) Given $C_{\mathrm{init}}, C_{\mathrm{accept}}$ of the deterministic TM, the ATM branches existentially to guess a mid-point configuration C, and then branches universally to check that $C_{\mathrm{init}} \rightsquigarrow C \rightsquigarrow C_{\mathrm{accept}}$. The depth of the recursion is

$$\log(\text{number of configurations}) = \log 2^{O(f(n))} = O(f(n)),$$

and at each step we require $O(f(n))$ time to write a configuration.

(3) [$\subseteq$] Construct the entire computation tree ($2^{O(f(n))}$ many nodes), and mark the accepting node (we can assume that there is a unique C_{accept}). Then, scan the graph repeatedly, and if a universal (existential) configuration is such that all (at least one) of its children are (is) accepting, mark it accepting. Stop when a scan introduces no more accepting nodes.

[$\supseteq$] We use a similar idea to the Cook-Levin theorem (theorem 2.13). For time $2^{O(f(n))}$, the computational tableau is of size $2^{O(f(n))} \times 2^{O(f(n))}$, so it cannot be stored in space $f(n)$. However: *"Injurious distance should not stop my way; For then despite of space I would be brought, From limits far remote where thou dost stay"*;[5] four pointers into the tableau can be stored:

s_1	s_2	s_3
	s	

[5]Shakespeare, Sonnet XLIV.

When the first pointer is pointing to s, existentially guess s_1, s_2, s_3, check that they yield s, and then universally branch on each s_i to check that it can be obtained from the initial configuration, i.e., from the first row. Once we are in the first row, the answer can be verified directly since we know the input. We start the recursion in the lower-left corner, again assuming that there is a unique C_{accept} where the head is on the first square and the tape is "clean." □

COROLLARY 4.23. $\mathsf{AL} = \mathsf{P}$, $\mathsf{AP} = \mathsf{PSPACE}$, $\mathsf{APSPACE} = \mathsf{EXPTIME}$.

Define the class $\mathsf{STA}(S(n), T(n), A(n))$ to be the class of languages accepted by ATMs that are simultaneously $S(n)$-space-bounded, $T(n)$-time-bounded, and make at most $A(n)$ alternations on inputs of length n. A "$*$" in any position means "don't care." We also write Σ, Π in the third position to impose that the alternations start with $\exists$ or $\forall$. For example, $\mathsf{L} = \mathsf{STA}(\log n, *, 0)$, and $\mathsf{NP} = \mathsf{STA}(*, n^{O(1)}, \Sigma 1)$.

EXERCISE 4.24. Show that for $S(n) \geq \log n$,

$$\mathsf{STA}(S(n), *, A(n)) \subseteq \mathsf{SPACE}(A(n)S(n) + S(n)^2).$$

4.5. Bennett's Trick

In his 1962 Ph.D. thesis, James Bennett ([**Ben62**]) shows how to encode a long sequence of numbers with shorter sequences of numbers by using alternation of quantifiers. The idea is that $(\exists \langle x_0, x_1, \ldots, x_n \rangle)$ can be expressed with

$$(\exists \langle y_0, y_1, \ldots, y_{\sqrt{n}} \rangle)(\forall i \leq \sqrt{n})(\exists \langle z_0, z_1, \ldots, z_{\sqrt{n}} \rangle).$$

We are going to use this trick to show a nice result about the linear time hierarchy.

Define the class linear time $\mathsf{LT} = \mathsf{TIME}(n)$, and nondeterministic linear time $\mathsf{NLT} = \mathsf{NTIME}(n)$. Let $\Sigma_0^{\mathrm{LT}} = \mathsf{LT}$ and $\Sigma_{i+1}^{\mathrm{LT}} = \mathsf{NLT}^{\Sigma_i^{\mathrm{LT}}}$, which denotes the class of languages decidable in nondeterministic linear time with a Σ_i^{LT} oracle. Define the linear time hierarchy as $\mathsf{LTH} = \bigcup_i \Sigma_i^{\mathrm{LT}}$. Let $\mathsf{NTIMESPACE}(f(n), g(n))$ be the class of languages decided simultaneously in time $O(f(n))$ and space $O(g(n))$ on a nondeterministic (multi-tape) Turing machine.

THEOREM 4.25 (Nepomnjascij). Let $0 < \varepsilon < 1$ be a rational number, and let a be a positive integer. Then, $\mathsf{NTIMESPACE}(n^a, n^\varepsilon) \subseteq \mathsf{LTH}$.

PROOF. In this proof we are going to forgo the notation $\vec{x}$ representing a vector of Boolean variables, as it would clutter the presentation. When we write $\mathbf{x}$ we mean a vector of vectors of Boolean variables.

Suppose M is a nondeterministic TM running in time n^a and space n^ε. Then, M accepts an input x iff

$$\exists \mathbf{y}[\mathbf{y} \text{ represents an accepting computation for } x]. \tag{12}$$

So $\mathbf{y} = y_1 y_2 \ldots y_{n^a}$ where each $|y_i| = n^\varepsilon$, and so $|\mathbf{y}| = n^{a+\varepsilon}$. So $\mathbf{y}$ is too long to verify in linear time (in $n = |x|$). So we use Bennett's trick: let $\mathbf{z} = z_1 z_2 \ldots z_{n^{1-\varepsilon}}$ where the z_i represent every $(n^{a-1+\varepsilon})$-th string in $\mathbf{y}$. So now, (12) can be restated as follows:

$$(\exists \mathbf{z})(\forall i)(\exists \mathbf{u})[\mathbf{u} \text{ shows } z_i \text{ yields } z_{i+1} \text{ in } n^{a-1+\varepsilon} \text{ steps } \& \ z_n \text{ is accepting}].$$

Note that $|\mathbf{z}| = n^{1-\varepsilon}n^{\varepsilon} = n$, so this is fine, but $|\mathbf{u}| = n^{a-1+\varepsilon}n^{\varepsilon} = n^{a-1+2\varepsilon}$. But note that $(a-1+2\varepsilon) < (a+\varepsilon)$ because $0 < \varepsilon < 1$. So we have reduced $\mathbf{u}$ with respect to $\mathbf{y}$, by a factor of $n^{1-\varepsilon}$, so to know how many times we have to repeat the above nesting of quantifiers, we solve for i in the following equation:

$$\frac{n^{a+\varepsilon}}{(n^{1-\varepsilon})^i} = n,$$

and solving, we get $i = (a+\varepsilon)/(1-\varepsilon)$, which is a constant as a, ε are fixed. □

Recall the definition of Δ_0-formulas on page 59. Let $\Delta_0^{\mathbb{N}}$ denote the class of languages that can be given by relations representable with Δ_0 formulas.

THEOREM 4.26. $\mathsf{LTH} = \Delta_0^{\mathbb{N}}$.

An interesting open problem is whether $\mathsf{P} \subseteq \mathsf{LTH}$, or in fact what is the relation of these two complexity classes; are they even comparable?

4.6. Answers to selected exercises

Exercise 4.10. The proof is analogous to the proof of the Space Hierarchy theorem (theorem 4.2), and the division by $\log t(n)$ is now necessary since the machine is "delayed" by keeping a counter.

Exercise 4.15. Let B be QSAT just as in the proof of theorem 4.13.

For any oracle A define the language $L(A) = \{x |\ \{0,1\}^{|x|} \cap A = \emptyset\ \}$. Note that an NP machine with oracle A can decide $\overline{L(A)}$ by guessing y of length $|x|$ and checking if $y \in A$. So, a $\mathsf{co\text{-}NP}$ machine with oracle A can decide $L(A)$. Just as in the proof of theorem 4.13, let $M_1^{\sqcup}, M_2^{\sqcup}, M_3^{\sqcup}, \ldots$ be a list of all oracle TMs, where machine i takes up at most n^i steps to decide any string of length n.

We construct A in stages so that $L(A)$ is not in NP^A. At **stage** 0, $A_0 := \emptyset$, and at **stage** i, we choose an n such that all strings examined (for membership in A) so far are of length $< n$ and furthermore, $n^i < 2^n$. Simulate $M_i^{\sqcup}$ on input 1^n, and each time $M_i^{\sqcup}$ queries the oracle with some y, if y's membership in A has already been established, answer accordingly, and otherwise declare y *not* in A.

If at the end $M_i^{\sqcup}$ rejects (all paths are rejecting), declare the remaining $\{0,1\}^n$ *not* in A. If $M_i^{\sqcup}$ accepts, find an accepting path and some $y \in \{0,1\}^n$ not queried on it, and declare it in A.

Exercise 4.24. Suppose that L is decided with space $S(n)$ and $A(n)$ many alternations, starting with an existential alternation. Let $R(C, C')$ be a predicate that is true iff C, C' are configurations which are not of the same type, meaning that one is existential and the other universal, and C' is reachable from C with a sequence of at most $2^{S(n)}$-many intermediate configurations where all these intermediate configurations are of the same type as C.

Then, R is decidable in nondeterministic space $S(n)$, and hence by Savitch's theorem, in deterministic space $S(n)^2$. Now we implement the alternation with recursion

as follows: check if there exists a C' (by cycling through all configurations lexicographically) such that $R(C_{\text{init}}, C')$ and furthermore if for all C'' we have $R(C', C'')$, then there exists a C''' such that $R(C'', C''')$ and so on.

The depth of this is $A(n)$, and from level to level we only need to record a single configuration, and to decide $R(C_1, C_2)$ we need $S(n)^2$ space that can be reused, so we need $A(n)S(n) + S(n)^2$ deterministic space.

4.7. Notes

For the Hierarchy theorems we follow [**Sip06**, §9.1]. See [**Sip06**, theorem 9.15, pg. 344] for a proof of theorem 4.7. The proof (and statement) of theorem 4.12 follows the exposition in [**AB09**]. §4.2.1 is based on [**SP95**, chapter 22]. The proof of theorem 4.26 can be found in [**CN10**, §3.4]. Exercise 4.24 is from [**Koz06**, homework 4, pg. 279]. The proof of theorem 4.26 can be found in [**CN10**].

5

Circuits

5.1. Basic results and definitions

A *Boolean circuit* can be seen as a directed, acyclic, connected graph in which the input nodes are labeled with variables x_i and constants $1, 0$ (or $\mathtt{T}, \mathtt{F}$), and the internal nodes are labeled with $\mathtt{And}, \mathtt{Or}, \mathtt{Not}$. We often denote $\mathtt{And}, \mathtt{Or}, \mathtt{Not}$ with the standard Boolean connectives $\wedge, \vee, \neg$, respectively. Furthermore, we often use $\bar{x}$ to denote $\neg x$.

Nodes are also called *gates* in the context of circuits. The *fan-in* (i.e., number of incoming edges) of a $\mathtt{Not}$-gate is always 1, and the fan-in of $\mathtt{And}, \mathtt{Or}$ can be anything. The *fan-out* (i.e., number of outgoing edges) of any node can be arbitrary. Each node in the graph can be associated with a Boolean function in the obvious way. The function associated with the output gate(s) is the function computed by the circuit. Note that a Boolean formula can be seen as a circuit in which every node has fan-out 1 (and $\wedge, \vee$ have fan-in 2, and $\neg$ has fan-in 1).

The *size* of a circuit is its number of gates, and the *depth* of a circuit is the largest number of gates on any path from an input gate to an output gate.

Alternatively, circuits (of fan-in 2 and one output) can be defined more formally as follows. A Boolean circuit is a program consisting of finitely many instructions of the form: $P_i := 0$, $P_i := 1$, $P_i := x_l$, $P_i := P_j \wedge P_k$, $P_i := P_j \vee P_k$, $P_i := \neg P_j$, where $j, k < i$ (to ensure acyclicity), and where $x_1, \ldots, x_n$ are the input variables. We want to compute P_m where m is the maximum index.

A *family of circuits* is an infinite sequence $C = \{C_n\} = \{C_0, C_1, C_2, \ldots\}$ of Boolean circuits where C_n has n input variables.

We say that a language $L \subseteq \{0,1\}^*$ has polysize circuits if there exists a polynomial p and a family C such that $|C_n| \leq p(n)$, and $\forall x \in \{0,1\}^*$, $x \in L$ iff $C_{|x|}(x) = 1$. Let P/poly be the class of all those languages which have polysize circuits.

LEMMA 5.1. All languages in P have polysize circuits.

See proof of theorem 2.7.

The converse of the above lemma (i.e., "L has polysize circuits implies $L \in \mathsf{P}$") does not hold, unless we put a severe restriction on how the n-th circuit is generated; as it stands, there are undecidable languages that have polysize circuits.

LEMMA 5.2. There are undecidable languages that have polysize circuits.

PROOF. Recall that we showed in theorem 4.1 that the language A_{TM} is undecidable. We may assume that the instances of A_{TM}, i.e., $\langle M, x \rangle$ are encoded as binary strings where the first bit is 1. This is a minor technical point that makes the rest of the proof simpler. Recall that $(n)_b$ is the binary representation of n.

Let $U = \{1^n | (n)_b \in \mathsf{A}_{\mathrm{TM}}\}$, and since A_{TM} is reducible to U (via an exponential time reduction, which is nevertheless a recursive reduction), it follows that U is also undecidable.

On the other hand, U has a polysize family of circuits $C = \{C_n\}$ deciding it, where C_n is an `And`-gate connected to all the inputs if $1^n \in U$, and a single `F`-gate otherwise. □

The trick in the above proof is to hide the computation in the "nonuniformity" of the circuits; the circuit C_n is indeed small, but given n, we do not know how to construct C_n (i.e., is it the `And`-gate or is it the `F`-gate?). Thus, we make the following definition: a family of circuits $C = \{C_n\}$ is *uniform* if there is a $O(\log n)$-space TM M which on input 1^n, outputs C_n.

THEOREM 5.3. A language L has uniform polysize circuits iff $L \in \mathsf{P}$.

Those languages (or Boolean functions) that can be decided with polysize, constant fan-in, and depth $O(\log^k n)$ circuits, form the class NC^k. The class AC^k is defined in the same way, except we allow unbounded fan-in. We set $\mathsf{NC} = \bigcup_k \mathsf{NC}^k$, and $\mathsf{AC} = \bigcup_k \mathsf{AC}^k$, and while it is easy to see that the uniform version of NC is in P, it is an interesting open question whether they are equal.

LEMMA 5.4. For all k, $\mathsf{AC}^k \subseteq \mathsf{NC}^{k+1} \subseteq \mathsf{AC}^{k+1}$. Thus, $\mathsf{NC} = \mathsf{AC}$.

EXERCISE 5.5. Prove lemma 5.4.

We say that a circuit family is in *layered form* if each circuit in the family has the following three properties: (i) it consists in alternating layers of `And` and `Or` gates, (ii) there are edges only between consecutive layers, and (iii) negations occur only at the input level. Note that in a layered circuit, the depth is the same as the number of layers.

LEMMA 5.6. Any AC circuit family C can be put in layered form incurring a polynomial increase in size, and a constant increase in depth.

PROOF. Let C_i be the i-th circuit in the given family. First, traverse the circuit from its output towards inputs, pushing any `Not` gates through `And` and `Or` gates using de Morgan laws, and replicating gates if necessary (see figure 1). At the end, all variables and negations are at level 0 (that is, layer 0 consists of $x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n$). Now choose if you want to start the alternation at level 1 with `And` or `Or` gates; say we start with `And` gates. Consider all gates of depth 1 (these will be gates connected directly to the inputs, negated or unnegated). If a depth 1 gate is an `And` gate, do nothing. If it is an `Or` gate, then put an `And` gate on each one of its wires (see figure 2). Now consider gates at level 2, and repeat. We may have to collapse several `And` gates (or several `Or` gates) into one `And` (or `Or`), since we are allowed unbounded fan-in. That is, a tree of `And` gates may be collapsed to a single `And` gate which has a lot of inputs. □

LEMMA 5.7. Addition of two integers is in AC^0.

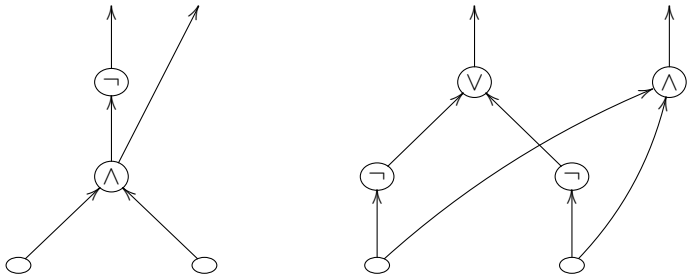


FIGURE 1. Using de Morgan and replicating gates.

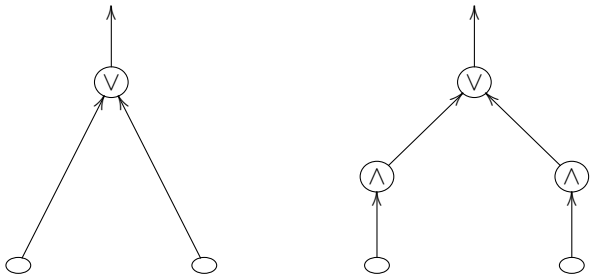


FIGURE 2. Putting `And` gates on the wires of an `Or`.

PROOF. Let $a_{n-1}, \ldots, a_0, b_{n-1}, \ldots, b_0$ be the two inputs (n-bit integers, where a_0, b_0 are the least significant bits).

ALGORITHM 5.8 (Classical Binary Addition).
On input $a_{n-1} \ldots a_0$ and $b_{n-1} \ldots b_0$

1. $y_0 := a_0 \oplus b_0$
2. $\text{carry}_0 := a_0 \wedge b_0$
3. for $i = 1, \ldots, n-1$
4. $\quad y_i := a_i \oplus b_i \oplus \text{carry}_{i-1}$
5. $\quad \text{carry}_i := (a_i \wedge b_i) \vee (a_i \wedge \text{carry}_{i-1}) \vee (b_i \wedge \text{carry}_{i-1})$
6. $y_n := \text{carry}_{n-1}$

Suppose we were to implement the classical algorithm with a circuit. Since carry_i depends on carry_{i-1}, the depth of the resulting circuit would be at least $O(n)$, and hence not an AC^0 circuit.

Instead, note that the i-th bit of the sum is $a_i \oplus b_i \oplus \text{carry}_i$, where carry_i is the carry bit left over after summing the $(i-1)$ first bits. Compute carry_i differently; note that the i-th carry is 1 iff $\exists j \in \{0, \ldots, i-1\}$ such that $a_j \wedge b_j$ and $\forall k \in \{j+1, \ldots, i-1\}$

it is the case that $a_k \vee b_k$. This can be stated with:

$$\text{carry}_i := \bigvee_{0 \le j < i} \left[(a_j \wedge b_j) \wedge \bigwedge_{j<k<i} (a_k \vee b_k) \right].$$

Note that this is a circuit of bounded depth. □

LEMMA 5.9. Repeated addition of integers is in NC^1.

PROOF. Assume that we want to add n n-bit numbers. Using the previous result directly, we can only show that repeated addition is in AC^1, which is a weaker claim. To show that it is in NC^1, we first design an NC^0 circuit which takes as input three integers, and returns two integers whose sum is the same as the sum of the original three: given a, b, c, we produce x, y as follows:

$$a_i, b_i, c_i \mapsto x_i = \underbrace{a_i \oplus b_i \oplus c_i}_{\text{sum } \textit{without} \text{ carries}}, \quad y_{i+1} = \underbrace{(a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i)}_{\text{the carry}}.$$

Each step reduces the number of integers to be added by 1/3. So in a logarithmic number of steps we are left with two integers. We now add those two with an AC^0 circuit, as was shown possible in lemma 5.7. □

We say that a Boolean function is *symmetric* if it only depends on the number of 1s in the input, and not on their order. More formally, if $f = \{f_n\}$, then f is symmetric if for all n, given any permutation $\pi \in S_n$, $f_n(x_1, x_2, \ldots, x_n) = f_n(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)})$.

Examples of symmetric functions are the *parity function*, $\text{Par} = \{\text{Par}_n\}$, where $\text{Par}_n(x_1, \ldots, x_n)$ is 1 if the number of 1s in its input is odd, and the *majority function*, $\text{Maj} = \{\text{Maj}_n\}$, where $\text{Maj}_n(x_1, \ldots, x_n)$ is 1 if at least half of its inputs are 1, and a generalization of majority, namely the *threshold function*, $\text{Th} = \{\text{Th}_{k,n}\}$ which outputs 1 iff at least k of its n variables are 1.

LEMMA 5.10. All symmetric Boolean functions can be computed with NC^1 circuits.

PROOF. Notice that if $f = \{f_n\}$ is a symmetric Boolean function, then the value of f_n depends only on the number of inputs that are set to 1. To construct the circuit that calculates f_n, we treat all inputs as 1-bit numbers and construct a circuit that adds them, thus counting the number of 1's in the input.

From lemma 5.9, we know that this can be done using an NC^1 circuit. The output of this circuit is a $\log n$ bit number. Now look at those $\log n$ many bits as the input to a circuit which gives us the value of f_n; this circuit can be given in CNF, since each clause will have $\log n$ many literals, and there are $O(2^{\log n})$ many clauses, and thus the CNF circuit has size $O(n \log n)$ (and depth 2 when the fan-in is unbounded, and depth $\log n$ when the fan-in is required to be 2).

Altogether we obtain an NC^1 circuit. □

Recall that a deterministic finite automaton (DFA) is a TM that reads the input left-to-right, changing states as it goes, and never using any extra space.

LEMMA 5.11. The set of regular languages is contained in NC^1.

PROOF. Let L be a regular language decided by a DFA M. Define M_a, $a \in \Sigma$, to be the Boolean matrix with a 1 in position (i, j) iff on symbol a machine M moves from state q_i to state q_j.

Given $a \in \{0,1\}^*$, $a = a_1 a_2 \dots a_n$, consider the iterated Boolean matrix product $\mathbf{M} := M_{a_1} M_{a_2} \cdots M_{a_n}$. (Boolean product means that instead of addition we have $\vee$ and instead of multiplication we have $\wedge$.) This product can be clearly computed with an NC^1 circuit as each M_a is of constant size (i.e., $|Q| \times |Q|$).

The output of this circuit is computed as the $\vee$ of all those entries which are in the row corresponding to the initial state, and which also are in the columns corresponding to the accepting states. □

EXERCISE 5.12. Prove by induction on n that the (i, j)-th entry of the product of matrices $M_{a_1} M_{a_2} \dots M_{a_n}$ is 1 iff—when started in state q_i—machine M reaches state q_j after reading $a_1 a_2 \dots a_n$.

LEMMA 5.13. For uniform circuit classes, $\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{NC}^2$.

PROOF. The first containment follows from the fact that an NC^1 circuit can be evaluated by a depth-first manner using only logspace. On the other hand, the circuits are uniform, so on an input $x \in \{0,1\}^n$, C_n can be computed in logspace. However, $|C_n|$ is polynomial in n, so it will not fit in logspace; we deal with that by only generating the parts of the circuit we need at a particular step of the depth-first evaluation.

The second containment is by definition.

The third follows from the fact that DIRECTREACH is in NC^2: matrix product is in NC^1, and therefore iterated matrix product is in NC^2, and so transitive closure can be computed in NC^2. □

5.2. Shannon's lower bound

In the 1960s, Shannon[1] a showed that most Boolean functions require large circuits, but as of today, for concrete Boolean functions, the best lower bounds we have are linear. Let B_n be the set of Boolean functions on n variables ($|B_n| = 2^{2^n}$), and in this section consider circuits with gates $\{\wedge, \vee, \neg, 0, 1\}$ of fan-in exactly two except for fan-in one negations at input level only. Let the size of a circuit be the number of $\{\wedge, \vee\}$ gates (so the negation gates at the input level do not count for size).

CLAIM 5.14. The number of circuits with n variables and size s is bounded above by $(2 \cdot (s + 2n + 2)^2)^s$.

EXERCISE 5.15. Prove claim 5.14.

CLAIM 5.16. For $s = 2^n/(10n)$ the value of the expression $(2 \cdot (s + 2n + 2)^2)^s$ is bounded above by $2^{2^n/5}$, and so "almost all" Boolean functions in B_n require circuits of size $\Omega(2^n/n)$.

[1]See [**Weg87**] for a complete presentation of this material.

PROOF. Setting $s = 2^n/(10n)$, and taking the log of $(2 \cdot (s+2n+2)^2)^s$, we obtain

$$2^n/(10n) \cdot \underbrace{2 \log \left(\sqrt{2} \, (2^n/10n + 2n + 2) \right)}_{(*)}, \tag{13}$$

and for sufficiently large n, $(*) < n$, so for sufficiently large n (13) is less than $2^n/(10n) \cdot 2n$, which is $2^n/5$. □

We know that $n2^n$ many gates are sufficient to compute any Boolean function in B_n, since $n2^n$ is the size of its CNF (or DNF) representation. It is also easy to give a slightly better construction, from which it follows that every Boolean function can be computed with $O(2^n)$ gates. We do this by induction on the number of inputs. Let

$$f = (\neg x_n \wedge f|_{x_n=0}) \vee (x_n \wedge f|_{x_n=1}),$$

where $f \in B_n$ and $f|_{x_n=0}, f|_{x_n=1} \in B_{n-1}$. Let $s(f)$ be the size of the smallest circuit computing the function f. Then

$$s(f) \leq 2 \cdot \max\{s(f|_{x_n=0}), s(f|_{x_n=1})\} + 3.$$

The claim follows from this.

In fact we can get an even tighter upper bound showing that the lower bound $\Omega(2^n/n)$ in claim 5.16 is very exact.

CLAIM 5.17. Every Boolean function f in B_n can be computed with circuits of size $O(2^n/(n - \log n)) \subseteq O(2^n/(n - \frac{1}{2}n)) \subseteq O(2^n/n)$.

PROOF. Observe that for any k there is a circuit with multiple outputs and size $O(2^{2^k})$ such that for every f in B_k there is an output computing f. This is just a CNF-like circuit, with all the possible 2^k different clauses at level 1, and at level 2 it has `Or` gates connected to all the possible subsets of clauses at level 1.

For $f \in B_n$, and for any $k \leq n$, we have that $f(x_1, \ldots, x_n)$ equals

$$\bigvee_{a_1,\ldots,a_k \in \{0,1\}} (x_1^{a_1} \wedge \cdots \wedge x_k^{a_k}) \wedge f(a_1, \ldots, a_k, x_{k+1}, \ldots, x_n),$$

where x^a is x if $a = 1$, and $\neg x$ if $x = 0$. Now construct a circuit with outputs for all functions $x^{a_1} \wedge \ldots \wedge x^{a_k}$ where $a_1, \ldots, a_k \in \{0,1\}^k$. Building this circuit inductively on k, layer by layer, results in a circuit of size $O(2^k)$. On the other hand, all the functions $f(a_1, \ldots, a_k, x_{k+1}, \ldots, x_n)$, for $a_1, \ldots, a_k \in \{0,1\}^k$, can be computed with a circuit of size $O(2^{2^{n-k}})$. Putting it all together, we have a circuit of size $O(2^k) + O(2^{2^{n-k}})$ computing f. Select k so that $n - k = \log(n - \log n)$. □

As we know that most Boolean functions require $\Omega(2^n/n)$ many gates, we can "construct" a hard Boolean function by enumeration: for every n, examine all Boolean functions in B_n, and for each such Boolean function examine all circuits of size up to $\Omega(2^n/n)$, and pick the first Boolean function not computable by such a circuit. This way, we obtain a family $f = \{f_n\}$. The trouble with this "construction" is that it tells us very little about the nature of hard Boolean functions (not to mention the infeasibility of the procedure, as it requires EXPSPACE).

An open question is: can we give a natural Boolean function $f = \{f_n\}$ that is provably hard? By "natural" we mean for example a function $f = \{f_n\}$ which computes if a graph of a given size has a clique; i.e., a Boolean function related to a natural combinatorial problem that requires large circuits.

So far, the best we can do for natural functions is the following linear lower bound. Recall that $\mathrm{Th}_{k,n}(x_1, x_2, \ldots, x_n)$ is true iff at least k of the n inputs are true.

CLAIM 5.18. $\mathrm{Th}_{2,n}$ requires circuit size at least $2n - 4$.

PROOF. We do the proof by induction on n. For $n = 2$ and $n = 3$ it is easy. Otherwise, let C be an optimal circuit for $\mathrm{Th}_{2,n}$, and suppose that the bottom most gate acts on variables x_i, x_j, where $i \neq j$. Under the four possible settings to x_i, x_j, the function $\mathrm{Th}_{2,n}$ has three possible sub-functions: $\mathrm{Th}_{0,n-2}, \mathrm{Th}_{1,n-2}, \mathrm{Th}_{2,n-2}$. It follows that either x_i or x_j fans out to another gate in C, for otherwise C would have only two inequivalent subcircuits under the settings of x_i, x_j. Suppose that it is x_i that fans-out to another gate. Setting x_i to 0 will eliminate the need for at least two gates from C. The resulting function is $\mathrm{Th}_{2,n-1}$, which by induction requires circuits of size $2(n-1) - 4$. Adding the two eliminated gates to this bound shows that C has at least $2n - 4$ gates, which completes the induction. □

5.3. The probabilistic method

In this section we give two different proofs of a super-polynomial lower bound for parity as computed by AC^0 circuits.[2]

5.3.1. First proof of Parity not in AC^0. We show PARITY $\notin \mathsf{AC}^0$, where PARITY is the language of strings over $\{0,1\}$ with an odd number of 1s, i.e., PARITY $= \{x \in \{0,1\}^* | \mathrm{Par}_{|x|}(x) = 1\}$, where the function $\mathrm{Par} = \{\mathrm{Par}_n\}$ was defined on page 68.

All Boolean functions on n variables can be computed by circuits of depth 2 (CNF or DNF). And a CNF circuit (which we shall also call an `And-Or` circuit) can be transformed directly into a DNF circuit (which we shall also call an `Or-And` circuit), and vice-versa, with the distributive laws:

$$\begin{aligned} (x \wedge (y \vee z)) &\leftrightarrow (x \wedge y) \vee (x \wedge z), \\ (x \vee (y \wedge z)) &\leftrightarrow (x \vee y) \wedge (x \vee z). \end{aligned} \tag{14}$$

Suppose that we have an `And-Or` circuit in which all the `Or` gates have fan-in c and the `And`-gate has fan-in d. We use the distributive laws (14) to transform our `And-Or` circuit into an `Or-And` circuit, where now the `And`s have fan-in d and the output `Or` has fan-in c^d (see figure 3).

Consider a polybounded family of circuits $C = \{C_n\}$, where each C_n is an `And-Or` circuit. Suppose that for every C_n, the fan-in of the `Or` gates (in the middle layer) is bounded by c, and the output `And`-gate depends on constantly many input variables, meaning that if you collect all the literals connected to an output `And`-gate (by a path

[2]Two other prominent lower bounds for circuits are Razborov's super-polynomial lower bound for monotone circuits computing CLIQUE (stated, but not proven, as theorem 6.28), and the Razborov-Smolensky results showing that for p, q different primes $\mathsf{AC}^0[p] \neq \mathsf{AC}^0[q]$.

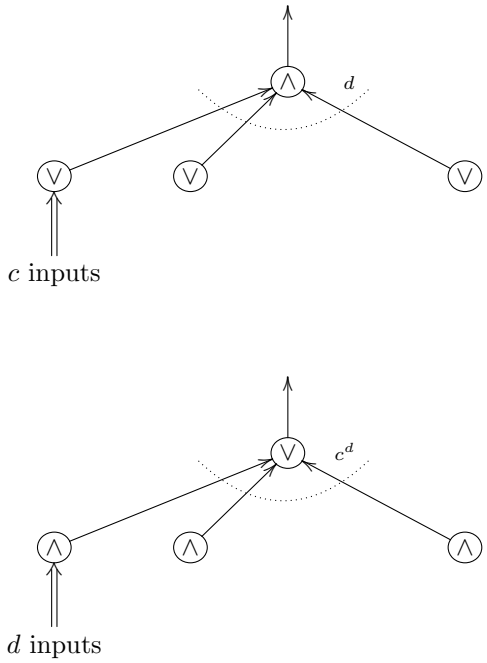


FIGURE 3. `And-Or` as `Or-And`.

from the input layer, passing through the layer of `Or`s) you get a set of variables whose size is always bounded by some constant e.

The following claim holds for such a C (as described in the previous paragraph).

CLAIM 5.19. C can be converted to an `Or-And` family C', still of polynomial size, of constant input fan-in. Also, the claim still holds if we change the definition of "depends on constantly many input variables" to mean a *semantic dependence*. That is, each output `And`-gate might be connected to an arbitrary number of literals, but its value depends on the value of only constantly many input variables.

PROOF. If the `And` gate is connected (through the `Or` gates) to no more than e variables, then there are at most 3^e different (in terms of input literals) `Or` gates (for each variable x the `Or` gate either uses x, its negation, or none of them). Deleting duplicate input connections we can lower the fan-in of the `And` gate to be at most 3^e. The fan-in of the `Or` gates is, of course, bounded by e. Thus, using distributive laws to exchange the `And` and `Or` levels, we will create a single `Or` gate connected to no more than e^{3^e} `And` gates, which is a constant number!

If we consider semantic dependence instead, then if we set all but the e variables to, say, 0, the circuit would compute the same Boolean function, and the output `And` would depend syntactically on no more than e variables. This finishes the proof. □

Note that the last word on the type of results presented in claim 5.19 is the Håstad switching lemma; see [**Bea94**].

CLAIM 5.20. Suppose that we have a circuit for $\overline{\text{Par}}_n$ of depth t and size g. Then there is a circuit of same depth and size for Par_n.

PROOF. Just note that $\overline{\text{Par}}_n(x_1, x_2, \ldots, x_n) = \text{Par}_n(\overline{x}_1, x_2, \ldots, x_n)$. □

CLAIM 5.21. Suppose that we have a circuit C for Par_n, and we set a variable x_i to 0 (or 1) and simplify. Then the resulting circuit C' computes Par_{n-1} or $\overline{\text{Par}}_{n-1}$. Consequently, the same applies if we set any number of variables.

PROOF. This follows immediately from the observation that

$$\text{Par}_n(0, x_2, \ldots, x_n) = \text{Par}_{n-1}(x_2, \ldots, x_n),$$
$$\text{Par}_n(1, x_2, \ldots, x_n) = \overline{\text{Par}}_{n-1}(x_2, \ldots, x_n).$$

We used x_1, but obviously the same works for any x_i. □

CLAIM 5.22. Par_n can be computed with a circuit of polynomial size and depth $O(\log n)$. Thus, PARITY $\in \mathsf{NC}^1$.

PROOF. This has already been shown in lemma 5.10. But this can also be checked directly, as

$$\text{Par}_n(x_1, \ldots, x_n) = \text{Par}_{\lfloor \frac{n}{2} \rfloor}(x_1, \ldots, x_{\lfloor \frac{n}{2} \rfloor}) \oplus \text{Par}_{\left(n - \lfloor \frac{n}{2} \rfloor\right)}(x_{(\lfloor \frac{n}{2} \rfloor + 1)}, \ldots, x_n).$$

Using this fact we can compute Par_n with a tree of `Xor` gates. Note that each $\oplus$ can be replaced (locally) using $x \oplus y \leftrightarrow (x \wedge \bar{y}) \vee (\bar{x} \wedge y)$, and note that all the gates in this circuit have fan-in 2. □

THEOREM 5.23. PARITY $\notin \mathsf{AC}^0$.

The rest of this section is a proof of this theorem. We start with a claim.

CLAIM 5.24. PARITY cannot be decided with a polysize `Or-And` circuit.

PROOF. Suppose that Par_n has an `Or-And` circuit. If one of the `And` gates has as inputs both x_i and $\bar{x}_i$, then eliminate this `And` since its contribution to the output `Or` is zero. If an `And` gate is missing a variable x_i as input (that is, neither x_i nor $\bar{x}_i$ is an input to this `And`) then by setting all the other literal inputs to this `And` to 1, we set the circuit to 1, *regardless* of the value of this x_i, which means that the circuit is not computing the parity correctly. Summing up, this `Or-And` circuit is such that to each `And` gate each variable comes in as an input (either as itself or negated). But this means that each `And` corresponds to "one row" of the truth table for Par_n, i.e., this `Or-And` circuit is the conjunctive normal form of Par_n. Since there are 2^{n-1} rows with value 1, there must be 2^{n-1} `And` gates, and so Par_n requires at least 2^{n-1} gates. □

EXERCISE 5.25. Show that PARITY cannot be computed with a polysize `And-Or` circuit.

Now, theorem 5.23 follows directly from the next claim.

CLAIM 5.26. $\forall t, \forall c, \forall p(n)$, PARITY cannot be computed using a depth t circuit family of size $p(n)$ that has input-level fan-in $\leq c$.

Note that theorem 5.23 follows from this claim because, in order to limit the input-level fan-in, we can always insert a dummy layer of gates at the input level (all of fan-in 1), and increase the depth to $(t+1)$.

The case $t = 2$ was already done (from claim 5.24 and exercise 5.25 we know that PARITY cannot be computed with a polysize circuit of depth 2, let alone one where there is a restriction on the input-level fan-in).

For $t > 2$ we are going to use the following strategy: let t be the least such depth, and suppose that $S_1, S_2, S_3, \ldots$ is a family of circuits of depth t and polysize computing PARITY. Then we are going to produce $S'_1, S'_2, S'_3, \ldots$, of depth $(t-1)$ and polysize, thereby getting a contradiction.

To produce S'_n we are going to take S_{4n^2} and set $(4n^2 - n)$ of its variables to 0 and 1 in such a way that we can use the distributive laws (14) on levels 1 and 2 to exchange these two levels *without increasing the size of the circuit exponentially*. To do this, it is enough to have a situation where *each gate on level 2 depends on only a constant number of variables* (see claim 5.19 and the discussion leading up to it). Then, levels 2 and 3 can be collapsed to a single layer leaving a circuit of depth $(t-1)$.

How can we come up with the right restriction mapping S_{4n^2} to S'_n? We are going to show one exists without explicitly constructing it: we use a *random restriction* and show that the probability of getting an appropriate substitution is positive, so we can conclude that one exists.

The kind of reasoning just described is a very powerful technique in combinatorics, and it goes under the name of a *probabilistic argument*; *"My thoughts and my discourse as madmen's are, at random from the truth vainly express'd"*.[3]

We shall use a family of random restrictions $r = \{r_n\}$, where

$$\Pr[x_i^{r_n} = x_i] = \frac{1}{\sqrt{n}}$$

$$\Pr[x_i^{r_n} = 0] = \Pr[x_i^{r_n} = 1] = \frac{1 - \frac{1}{\sqrt{n}}}{2}.$$

Then, $S_n^{r_n}$ is the result of applying the random restriction r_n to $x_1, x_2, \ldots, x_n$, and simplifying S_n accordingly, so $S_n^{r_n}$ is Par_m or $\overline{\mathrm{Par}}_m$ where $m \leq n$.

Since it will be understood that we are applying r_n to S_n, we are going to drop the subscript of r_n, and write r.

EXERCISE 5.27. As a warm up to the probability that we are going to be using in this section (and in chapter 7) show the *Markov inequality*:

$$\Pr[X \geq a] \leq \frac{E(X)}{a}, \tag{15}$$

and the *Chebyshev inequality*:

$$\Pr[|X - E(X)| \geq a] \leq \frac{V(X)}{a^2}. \tag{16}$$

[3] Shakespeare, Sonnet CXLVII.

CLAIM 5.28. $\Pr[\text{there are fewer than } \frac{\sqrt{n}}{2} \text{ vars in } S_n^r] = O(\frac{1}{\sqrt{n}})$.

PROOF. Consider r acting on S_n. The probability of setting a variable to 0 or 1 is $q = 1 - \frac{1}{\sqrt{n}}$, and the probability of leaving it unset is $p = \frac{1}{\sqrt{n}} = 1 - q$. This is done independently for each variable, and so it is a *binomial distribution* with probability of having exactly k successes (i.e., of leaving exactly k variables unset) at $\binom{n}{k} p^k q^{n-k}$.

Let X be the random variable counting the number of successes (i.e., counting the number of unset variables) in n trials. We want to compute $E(X)$ and $\mathrm{Var}(X)$. Let

$$X_i = \begin{cases} 1 & \text{if } i\text{-th trial is a success, i.e., } x_i^r = x_i \\ 0 & \text{otherwise} \end{cases},$$

then $E(X_i) = 1 \cdot p + 0 \cdot q = p$, and $E(X) = E(\sum_{i=1}^n X_i) = \sum_{i=1}^n E(X_i) = n \cdot p$ and

$$\begin{aligned} \mathrm{Var}(X) &= E(X^2) - E(X)^2 = E((\sum_{i=1}^n X_i)^2) - (n \cdot p)^2 \\ &= E(\sum_{i=1}^n \sum_{j=1}^n X_i X_j) - (n \cdot p)^2 \\ &= n(n-1)p^2 + \underbrace{n \cdot p}_{(*)} - (n \cdot p)^2 = np - np^2 = np(1-p), \end{aligned}$$

where $(*)$ is for when $i = j$, because $E(X_i X_i) = p$.

We are now ready to prove the claim.

$$\begin{aligned} &\Pr[\text{fewer than } \frac{\sqrt{n}}{2} \text{ vars remain in } S_n^r] \\ \leq &\Pr[X \leq \frac{\sqrt{n}}{2}] = \Pr[\sqrt{n} - X \geq \sqrt{n} - \frac{\sqrt{n}}{2}] = \Pr[E(X) - X \geq \frac{\sqrt{n}}{2}] \\ \leq &\Pr[|X - E(X)| \geq \frac{\sqrt{n}}{2}]. \end{aligned}$$

Using Chebyshev inequality, the last line can be bounded by

$$\leq \frac{\mathrm{Var}(X)}{\left(\frac{\sqrt{n}}{2}\right)^2} = \frac{\sqrt{n} - 1}{\left(\frac{n}{4}\right)} \leq 4 \cdot \frac{\sqrt{n}}{n} = 4 \cdot \frac{1}{\sqrt{n}},$$

which finishes the proof of the claim. □

We are now going to show that after a random restriction r, the `And` gates on the second level depend on a constant number of variables with high probability. In other words, we prove by induction on c that the `And` gates depend on "too many variables" with probability $O(n^{-k})$.

The basis case is $c = 1$, where fan-in 1 into the `Or` gates (at level 1) means that these `Or` gates can be disregarded, and the inputs connected directly to the `And` gates.

We now consider two possibilities:

(1) the `And`-gate has a large fan-in ($f \geq 4 \cdot k \cdot \ln n$),

(2) the And-gate has a small fan-in ($f < 4 \cdot k \cdot \ln n$).

CLAIM 5.29. In the first case (large fan-in), we have:

$$\Pr[\texttt{And}\text{-gate is } \textit{not } 0] = O\left(n^{-k}\right).$$

PROOF. This is because it is very likely that a random r would have set one of the And's inputs to 0. So with high probability, the And-gate depends on no variables. We now give the details.

$$\begin{aligned}
&\Pr[\texttt{And}\text{-gate not } 0] \\
\leq &\Pr[\text{all inputs not } 0] \leq \Pr[\text{a fixed input not } 0]^{4k \ln n} \\
\leq &\left(\frac{1 + \frac{1}{\sqrt{n}}}{2}\right)^{4k \ln n},
\end{aligned}$$

and for $n \geq 4$,

$$\leq \left(\frac{3}{4}\right)^{4k \ln n} = n^{4k \ln(\frac{3}{4})} \leq n^{-k},$$

which proves the claim. □

CLAIM 5.30. In the second case (small fan-in), we have:

$$\Pr[\texttt{And}\text{-gate depends on } \textit{more} \text{ than } 18k \text{ inputs}] = O\left(n^{-k}\right).$$

EXERCISE 5.31. Show the following upper bound for the binomial distribution:

$$\Pr[X \geq a] = \sum_{i=a}^{n} \binom{n}{i} p^i (1-p)^{n-i} \leq p^a 2^n. \tag{17}$$

PROOF. (of claim 5.30)

$$\begin{aligned}
&\Pr[\texttt{And}\text{-gate depends on more than } a \text{ variables}] \\
\leq &\sum_{i=a}^{4k \ln n} \binom{4k \ln n}{i} (1/\sqrt{n})^i (1 - 1/\sqrt{n})^{4k \ln n - i},
\end{aligned}$$

and using (17),

$$\leq (1/\sqrt{n})^a 2^{4k \ln n} = n^{-a/2} n^{8k} = n^{8k - a/2}.$$

Now letting $a = 18k$ completes the proof. □

In the induction step assume the result holds for $(c-1)$. Again, there are two cases:

(1) before the random restriction, the And-gate has many Or-gates below it with *disjoint input variables* (at least $d \cdot \ln n$, where $d = k \cdot 4^c$),

(2) before the random restriction, the And-gate has few Or-gates below it with *disjoint input variables* (less than $d \cdot \ln n$).

In the first case (many `Or`-gates below `And`), it is very likely that after the random restriction one of the `Or`-gates will have all of its inputs set to 0, and so the `And`-gate is 0, and so it depends on no variables:

$$\begin{aligned}
&\Pr[\texttt{And}\text{-gate is } not = 0] \\
\leq &\Pr[\text{none of the } \texttt{Or}\text{-gates is } 0] \\
\leq &(\Pr[\text{a fixed } \texttt{Or}\text{-gate is not } 0])^{d \ln n} \\
\leq &(1 - 4^{-c})^{d \cdot \ln n} \quad \text{for } n \geq 4 \\
\leq &n^{d \cdot \ln(1-4^{-c})} \\
\leq &n^{-d \cdot 4^{-c}} \quad \text{since } \ln(1-x) \leq -x \\
= &n^{-k}.
\end{aligned}$$

In the second case (few `Or`-gates below `And`), choose a *maximal* set of `Or`-gates with *disjoint input variables*. Let H be the set of the variables that occur in these `Or` gates.

Since the `Or`-gates' fan-in is c, we know that $|H| \leq c \cdot d \cdot \ln n$. Let $\{\tau_1, \tau_2, \ldots, \tau_l\}$, $l = 2^{|H|}$, be all the truth assignments to variables in H. Let $A_i = C^{\tau_i}$, after simplifying, where C represents the whole `And-Or` circuit. Since each of the `Or`-gates has a variable in H, each `Or`-gate in A_i either disappears or "loses" an input. So the input fan-in in each A_i is at most $(c-1)$. By the induction hypothesis, the probability that A_i^r depends on more than e_{c-1} variables is bounded above by $O(n^{-k})$.

Consider the Boolean function f_C,

$$f_C = \bigvee_{i=1}^{l} \text{clause}_H(\tau_i) \wedge A_i, \tag{18}$$

where $\text{clause}_H(\tau_i)$ is a conjunction of literals over variables in H, in such a way that if $x \in H$, and $\tau_i(x) = 0$, then x appears as $\bar{x}$, and if $\tau_i(x) = 1$, then x appears as x. Therefore, the probability that f_C depends on more than $l \cdot e_{c-1}$ many variables is bounded above by $l \cdot O(n^{-k})$. So, if we can show that with high probability $|H|$ is constant, we will have that with high probability l is constant as well, and then the claim follows.

Let h be the number of variables in H after a random restriction. We show that:

$$\Pr[h > 4cd + 2k] = O\left(n^{-k}\right).$$

To see this, note that $|H| \leq cd \ln n$, so once again using (17) we get that

$$\Pr[h > a] \leq 2^{cd \ln n} \cdot \left(\frac{1}{\sqrt{n}}\right)^a \leq n^{2cd} \cdot n^{-a/2} = n^{2cd - a/2},$$

and solving for a in $2cd - a/2 = k$ we get that $a = 4cd + 2k$.

Thus, with high probability (which always means $O(1 - n^{-k})$) $2^h \leq 2^{4cd+2k}$. Hence, with high probability, $l \leq 2^{4cd+2k}$ in (18), and so, with high probability,

$$e_c = 2^{4cd+2k} \cdot e_{c-1} + (4cd + 2k),$$

where e_c is the constant bounding the number of variables on which f_C depends.

5.3.2. Second proof of Parity not in AC^0. We are going to give a different proof of Parity $\notin \mathsf{AC}^0$, which also uses the probabilistic argument, but with a dash of linear algebra, and employing the idea of arithmetization that was introduced in §3.3.2 (see page 44).

The gate $\mathtt{And}(x_1, \ldots, x_n)$ can be represented by the multivariate polynomial $\Pi_{i=1}^n x_i = x_1 x_2 \ldots x_n$, over $\mathbb{Z}$. Using $(1-x)$ to represent $\mathtt{Not}$, and de Morgan laws, we give the polynomial representation of the $\mathtt{Or}$ function: $1 - \Pi_{i=1}^n (1 - x_i)$. Note that the polynomials thus obtained have degree n; we are interested in lowering this degree, and to that end we use the probabilistic method. We construct a random polynomial as follows: let $S_0 = \{1, \ldots, n\}$. Let $S_i \supseteq S_{i+1}$, for $i \in \{0, \ldots, \log n + 1\}$, where S_{i+1} is chosen randomly so that for all $j \in S_i$, $\Pr[j \in S_{i+1}] = \frac{1}{2}$.

Let $q_i = \sum_{j \in S_i} x_j$ be a random polynomial (of degree 1).[4]

Let $p = \Pi_{i=0}^{\log n + 2}(1 - q_i)$, so it is a polynomial of degree $O(\log n)$.

Claim 5.32. If $\mathtt{Or}(x_1, \ldots, x_n) = 0$, then $1 - p = 0$.

Proof. If $\mathtt{Or}(x_1, \ldots, x_n) = 0$, then $x_1 = \cdots = x_n = 0$, so $q_i = 0$ for all i, so $1 - q_i = 1$ for all i, so their product p is 1, so $1 - p = 0$. □

Claim 5.33. If $\mathtt{Or}(x_1, \ldots, x_n) = 1$, then there is at least one $x_i = 1$. In this case, the probability is $\geq \frac{1}{2}$ that one of the polynomials q_i has the value exactly 1, and so $\Pr[1 - p = 1] \geq \frac{1}{2}$.

Proof. Let T be a set of variables of an $\mathtt{Or}$ that are set to true; we need to argue that for any choice of a non-empty $T \subseteq S_0$, the probability is at least $\frac{1}{2}$ that there is at least one $i \in \{0, 1, \ldots, \log n + 2\}$ such that the size of $T \cap S_i$ is exactly 1. We accomplish this by considering the probability of two cases.

Case 1. For all $i \in \{0, 1, \ldots, \log n + 2\}$, we have that $|T \cap S_i| > 1$.

Since the S_i form a non-ascending chain of subsets, it follows that for all i, $|T \cap S_i| > 1$ iff $|T \cap S_{\log n+2}| > 1$, which is true if at least two variables "survive" all the way from S_0 to $S_{\log n+2}$.

The probability of this happening is bounded by $\binom{n}{2} 4^{-(\log n+2)} < \frac{1}{16}$, since there are $\binom{n}{2}$ ways to choose a pair of variables, and then there is a probability of $\frac{1}{2^{\log n+2}}$ that each "survives" until the end. Note that we could have given a tighter bound, since we are over counting (these pairs intersect). We could have accomplished that with the *Inclusion-Exclusion Principle*, but we do not need such tight bounds in this case.

In fact, the following (even coarser) bound will do just fine for us:

$$\Pr[|T \cap S_{\log n+2}| > 1] \leq \Pr[|T \cap S_{\log n+2} \geq 1] \leq \binom{n}{1} 2^{-(\log n+2)} = \frac{1}{4}.$$

Case 2. There is an $i \in \{0, 1, \ldots, \log n + 2\}$ with $|T \cap S_i| \leq 1$.

[4]The degree of a multivariate polynomial is calculated by writing it out as a sum of monomials—unique up to order of summation—and find the monomial with the highest degree, where the degree of a monomial $(x_1^{a_1} \cdots x_l^{a_l})$ is $a_1 + \cdots + a_l$.

Suppose that $|T \cap S_0| = 1$; then we are set. Otherwise, $|T \cap S_0| = |T| > 1$, and let i be such that $|T \cap S_{i-1}| > 1$ and $|T \cap S_i| \leq 1$. Let $t = |T \cap S_{i-1}|$. The probability that $|T \cap S_i| = 1$ *under the assumption* that $t > 1$ and $|T \cap S_i| \leq 1$ is given by

$$\frac{t2^{-t}}{2^{-t} + t2^{-t}} = \frac{t}{t+1} \geq \frac{2}{3},$$

since $t2^{-t}$ is the probability that one of the t variables "survives," and 2^{-t} is the probability that none of the t variables "survive."

We are now going to put case 1 and 2 together, to obtain a lower bound for $\Pr[\exists i \text{ s.t. } |T \cap S_i| = 1]$. Case 1 does not occur with probability $\frac{3}{4}$, and in case 2 we get what we want with probability $\geq \frac{2}{3}$, and multiplying the two values[5] we obtain $\geq \frac{1}{2}$. □

So the polynomial $(1 - p)$ approximates the `Or`, but with a success rate of only $\frac{1}{2}$. But we can improve this by selecting independent polynomials $p_1, p_2, \ldots, p_t$, and then using $P = 1 - \Pi_{i=1}^{t} p_i$, which has degree $O(t \log n)$. The error probability of P is $\geq \frac{1}{2^t}$. To get the error probability below a given constant ε, we want $\frac{1}{2^t} < \varepsilon$, so $t > \log \varepsilon^{-1}$.

The polynomial for the $\texttt{And}(x_1, \ldots, x_n)$ is just the product of the p_i's with x_j replaced by $(1 - x_j)$, so it is just the dual case of the analysis of the `Or`.

We want to simulate an AC^0 circuit family of size $s = s(n)$ and depth d using our polynomials, so that the error probability is at most ε. We replace all the gates with our polynomials, making sure that the error probability for each gate is $\leq \frac{\varepsilon}{s}$. The degree of the polynomial approximating the `And` and `Or` gates with error $\leq \frac{\varepsilon}{s}$ is $O(\log(\frac{s}{\varepsilon}) \log n)$. So the degree of the polynomial approximating such a circuit of depth d is $O(\log^d(\frac{s}{\varepsilon}) \log^d n) = O(\log^{2d}(\frac{s}{\varepsilon}))$, since $s = s(n)$ is a polynomial in n.

Thus, for every $f \in \mathsf{AC}^0$, we can generate a polynomial p of polylogarithmic degree (in n), with the property that for any $(a_1, \ldots, a_n) \in \{0,1\}^n$ the probability is at least $(1 - \varepsilon)$ that $f(a_1, \ldots, a_n) = p(a_1, \ldots, a_n)$.

Claim 5.34. There exists polynomial $\mathcal{P}$ s.t. $f(a_1, \ldots, a_n) = \mathcal{P}(a_1, \ldots, a_n)$ for all $(a_1, \ldots, a_n)$ in some S, where $|S| \geq (1 - \varepsilon)2^n$.

Proof. For a random polynomial thus generated, the expectation of the number of inputs for which it computes the circuit correctly is $(1 - \varepsilon)2^n$. There has to be a polynomial $\mathcal{P}$ that meets this expectation. □

[5]This is a little bit more subtle than that. You can apply the rule $\Pr[A \wedge B] = \Pr[A] \cdot Pr[B]$ only if the events are independent. In case they are dependent, you can apply the rule (which is *always* valid): $\Pr[A \wedge B] = \Pr[A] \cdot \Pr[B|A]$ where $\Pr[B|A]$ is the conditional probability (i.e., what is the probability of B taking place, given that A has taken place). Let A be the event: "not for all i, $|T \cap S_i| > 1$," and we know from Case 1. that $\Pr[A] > \frac{3}{4}$. Let B be the event: "for some i, $|T \cap S_i| = 1$," and note that in case 2, we are doing our analysis of B taking place, *under* the assumption that event A has taken place, i.e., we computed a lower bound for $\Pr[B|A]$, i.e., $\Pr[B|A] > \frac{2}{3}$. In fact, our events A and B are *not* independent events; B is a "subset" of event A. It follows that $\Pr[B] = \Pr[A \wedge B] = \Pr[A] \cdot \Pr[B|A] > \frac{3}{4}\frac{2}{3} = \frac{1}{2}$.

Identify true with -1 and false with $+1$. The linear function that maps 0 to 1 and 1 to -1 is $x \mapsto 1-2x$; its inverse is $x \mapsto \frac{(1-x)}{2}$. Apply this linear function to the polynomial $\mathcal{P}$ that correctly simulates f on $(1-\varepsilon)2^n$-many inputs to obtain a polynomial $\mathcal{Q}$ of the same degree—which now correctly simulates f transformed to use $\{-1,+1\}^n$, again over $(1-\varepsilon)2^n$-many inputs.

Suppose that the parity function is in AC^0. Then there must be such a polynomial $\mathcal{Q}$ for parity. For $(1-\varepsilon)2^n$-many input strings (in $\{-1,+1\}^n$), $\mathcal{Q}(y_1,\ldots,y_n) = \Pi_{i=1}^n y_i$, i.e., $\mathcal{Q}$ corresponds exactly to multiplication: if the number of 1s is odd, $\mathcal{Q}$ is -1, and if it is even, it is $+1$.

Claim 5.35. There is no polynomial $\mathcal{Q}$ of degree $\frac{\sqrt{n}}{2}$ that correctly represents the function $\Pi_{i=1}^n y_i$ for $(1-\varepsilon)2^n$ strings in $\{-1,+1\}^n$.

Proof. We first give the proof outline, and then fill in the details. Let

$$S = \{(y_1,\ldots,y_n) \in \{-1,+1\}^n | \Pi_{i=1}^n y_i = \mathcal{Q}(y_1,\ldots,y_n)\},$$

where $\mathcal{Q}$ is a polynomial of degree $\frac{\sqrt{n}}{2}$ that correctly represents $\Pi_{i=1}^n y_i$ on $(1-\varepsilon)2^n$ many strings in $\{-1,+1\}^n$. Thus, $|S| \geq (1-\varepsilon)2^n$. We can assume that $\mathcal{Q}$ is *multilinear*, that is, no variable has exponent larger than 1. Let $L(S)$ be the vector space (over $\mathbb{R}$) of functions $f : S \longrightarrow \mathbb{R}$. The dimension of $L(S)$, $\dim(L(S))$, is $|S|$. On the other hand, let POL be the set of n-variable multilinear polynomials of degree $\frac{(n+\sqrt{n})}{2}$. Then, POL is also a vector space (over $\mathbb{R}$), with the usual polynomial addition and multiplication by scalars in $\mathbb{R}$. Then, $\dim(\text{POL})$ is $\sum_{i=0}^{\frac{(n+\sqrt{n})}{2}} \binom{n}{i}$, and using the Stirling approximation we can show that this is strictly smaller than $|S|$. On the other hand, $\dim(L(S)) \leq \dim(\text{POL})$; contradiction.

We now provide some details. The natural basis for $L(S)$ is B given by the set of functions $f_s : S \longrightarrow \mathbb{R}$ where $f_s(s) = 1$ and $f_s(s') = 0$ for $s' \neq s$. Now, any function in $L(S)$ can be written as a linear combination of functions in B.

Note that any f_s can be represented by an n-degree multivariate multilinear polynomial as follows:

$$f_{(s_1,\ldots,s_n)}(x_1,x_2,\ldots,x_n) \mapsto \frac{1}{2^n}(1-s_1\cdot x_1)(1-s_2\cdot x_2)\cdots(1-s_n\cdot x_n),$$

and the expression on the right-hand side can be written out as a sum of monomials. Each such monomial (without the constant coefficient multiplying it in front) is of the form $x_{i_1}x_{i_2}\cdots x_{i_k}$ with $k \leq n$.

Now consider such monomials in the representation of a given f_s. If a monomial has at most $n/2$ many variables, i.e., $k \leq n/2$, then leave it as it is. Otherwise, $k > n/2$, and replace this monomial by the polynomial

$$g = \mathcal{Q}(x_1,x_2,\ldots,x_n)x_{j_1}x_{j_2}\cdots x_{j_{n-k}},$$

where $\{x_{j_1},x_{j_2},\ldots,x_{j_{n-k}}\} = \{x_{i_1},x_{i_2},\ldots,x_{i_k}\}^c$. Two observations about the polynomial g: first, it is of degree $(\sqrt{n}+n)/2$, and second, on S it takes on the same values as the original monomial.

Thus, we just showed that any f_s can be represented with a multivariate multilinear polynomial of degree at most $(\sqrt{n}+n)/2$. We are not quite done yet; to show

that $\dim(L(S)) \leq \dim(\text{POL})$, we need to show that this mapping ($f_s \overset{h}{\mapsto} p \in \text{POL}$, where we extend h from basis($L(S)$) to all of $L(S)$ in the natural way to obtain a vector space homomorphism) is such that $h(S)$ is linearly *independent*.

Suppose that $h(S)$ is linearly dependent, so there are $p_1, p_2, \ldots, p_k \in h(S)$ such that $c_1p_1 + c_2p_2 + \cdots + c_kp_k = 0$ (assume all $c_i \neq 0$), i.e., it is the 0 polynomial. We now examine the preimages of these p_i's in $L(S)$, i.e., we look at $f_{s_1}, f_{s_2}, \ldots, f_{s_k}$, where $h(f_{s_i}) = p_i$ (note that these preimages are unique, i.e., $|S| = |h(S)|$).

It follows that the function $f = c_1f_{s_1} + c_2f_{s_2} + \cdots + c_kf_{s_k}$ is mapped by h to the zero polynomial. It follows therefore that $f = 0$, i.e., f is the zero function. But $f(s_1) = c_1f_{s_1}(s_1) = c_1 \neq 0$; contradiction. □

We showed that the degree of a polynomial approximating a bounded depth circuit is poly-logarithmic. On the other hand, we need degree higher than $\frac{\sqrt{n}}{2}$ to approximate parity. Thus PARITY $\notin \mathsf{AC}^0$.

5.4. Computation with advice

In this section we consider a nonuniform version of TMs which capture the nonuniformity of circuits.

Suppose that our TMs have an extra read-only input tape called the *advice tape*, and let $A(n)$ be a function mapping integers to strings in Σ^*. We say that machine M decides language L with advice $A(n)$ if $x \in L$ implies $M(x, A(|x|)) =$ "yes," and $x \notin L$ implies $M(x, A(|x|)) =$ "no." That is, the advice $A(n)$, specific to the length of the input, helps M decide all strings of length n correctly.

Let $f(n)$ be a function mapping nonnegative integers to nonnegative integers. We say that $L \in \mathcal{C}/f(n)$ if there is an advice function $A(n)$, where $|A(n)| \leq f(n)$ for all $n \geq 0$, and a TM M running in complexity $\mathcal{C}$, such that M decides L with advice A. We write $\mathcal{C}/\text{poly}$ to indicate TMs working in complexity $\mathcal{C}$ with an advice bounded by some fixed polynomial.

CLAIM 5.36. $L \in \mathsf{P}/\text{poly}$ iff L has polysize circuits.

PROOF. If $L \in \mathsf{P}/n^k$ then it can be decided by a deterministic machine with advice. But then for each length n the advice is fixed, so we can encode the whole computation of this machine using a polynomial size circuit. If, on the other hand, L has polynomial circuits, we can construct the following machine with advice to decide it: treat the advice as a description of a circuit, simulate this circuit on your input, and accept iff the result was 1. It is clear that this is a polytime machine which, when given descriptions of circuits for L as the advice, decides L. □

CLAIM 5.37. If SAT $\in \mathsf{P}/\log n$, then $\mathsf{P} = \mathsf{NP}$.

EXERCISE 5.38. Prove claim 5.37.

Let $G = (V, E)$ be a connected undirected regular graph of degree exactly d, i.e., that for each $u \in V$, $|\{(u, v) : \text{for some } v \in V \text{ s.t. } (u, v) \in E\}| = d$.

Assume that for any given u we have ordered the edges incident upon u in some way. A string $U = l_1l_2 \ldots l_m \in \{1, 2, \ldots, d\}^*$ and a node u induce a traversal of the

graph as follows: we start at u, and we take edge l_1 out of node u and arrive at node u_1. Then we take edge l_2 out of node u_1 and arrive at node u_2, etc. (We may visit a node more than once). We say that U traverses G if it visits every node of G.

EXERCISE 5.39. Use a probabilistic argument to show that for each n there is a traversal sequence of length $O(d \cdot n^4)$ that works for all graphs with n nodes and degree d (i.e., a *universal traversal sequence*). What does this tell you about the complexity of undirected reachability?

5.5. Answers to selected exercises

Exercise 5.15. Note that this number is a gross overestimate. We just count graphs with s nodes labeled by a gate with two inputs, not being concerned about the fact that many such graphs are not really circuits because they may contain cycles. Each gate is a $\wedge$ or a $\vee$, and its inputs are two other nodes. Each input can be either a gate (s many choices), a literal ($2n$ choices), or a constant (2 choices). Compounding these choices for s gates gives us the result.

Exercise 5.27. The proof of (15) is:

$$E(X) = \sum_x x \cdot \Pr[X = x] \geq \sum_{x \geq a} x \cdot \Pr[X = x] \geq a \sum_{x \geq a} \Pr[X = x] = a \Pr[X \geq a].$$

Note that $\mathrm{Var}(X) = E((X - E(X))^2) = E(X^2) - (E(X))^2$ which is the average of the square of the distance of each data point from the mean (the second expression is more useful in practice). The proof of (16) is:

$$\Pr[|X - E(X)| \geq a] = \Pr[(X - E(X))^2 \geq a^2] \leq \frac{E((X - E(X))^2)}{a^2} = \frac{\mathrm{Var}(X)}{a^2}.$$

Exercise 5.39. Check in [**Pap94**] that the probability that a fixed node v is *not* visited by a random walk of length dn^2 starting at some node u is $\leq 1/2$. If we repeat this "experiment" m times, or, equivalently, increase the random walk to dn^2m many steps, the probability of not visiting v becomes $\leq 1/2^m$. So the probability of missing some node during the random walk is $\leq n/2^m$. The number of d-regular graphs with n nodes is (grossly) bounded above by n^{dn}, so we want to choose an m such that $(n/2^m)(n^{dn}) < 1$, assuring that there is a universal traversal sequence of all d-regular graphs with n nodes. Choose $m = n^2$. This tells us two things about undirected reachability: (i) it is in RL (randomized logspace, with no false positives), and (ii) it is in L/poly, i.e., logspace with polynomial advice.

5.6. Notes

The proof of lemma 5.2 is based on [**Pap94**, proposition 11.2]; §5.3.1 is based on [**SP95**, chapter 11]; §5.3.2 is based on [**SP95**, chapter 12]; §5.4 is based on [**Pap94**, exercise 11.5.24]. Originally, computation with advice was introduced in [**KL80**]. Claim 5.36 is [**Pap94**, exercise 11.5.24(a)]; Claim 5.37 is [**Pap94**, exercise 11.5.24(b)].

In this chapter we used properties of the logarithm: (i) $\log_c(a) = 1/\log_a(c)$, and (ii) $\log_a(b)/\log_a(c) = \log_c(b)$.

For (i), note that $c^{\log_c(a)} = a$, so $c^{\log_c(a)+1} = ac$, so $\log_a(c^{\log_c(a)+1}) = \log_a(ac)$, so $(\log_c(a)+1)\log_a(c) = 1+\log_a(c)$, and so $\log_c(a)\log_a(c) = 1$. For (ii), note $a^{\log_a(b)} = b$, so $\log_c(a^{\log_a(b)}) = \log_c b$, so $\log_a(b)\log_c(a) = \log_c(b)$, and so $\log_a(b)/\log_a(c) = \log_c(b)$, where we used (i) to derive the last step.

6

Proof Systems

6.1. Introduction

This chapter deals with propositional logic, from the point of view of complexity. The beginnings of propositional logic can be traced to the syllogisms of Aristotle (384–322 BC) and to to the Stoic philosopher Chrysippus (c.280–c.207 BC). A long break of two thousand years followed, until George Boole in the 19th century.[1] There was much activity at the turn of the 20th century with Frege, Russell & Whitehead, Post, and others who were spurred by the problem of laying down the foundations of mathematics. After a break of half a century there was a revival of interest in propositional logic due to Computer Science, and in particular due to the P versus NP problem.

For a fuller history of propositional logic see for example [**Chu96**] and [**HA99**]. For an introduction to propositional proof complexity see [**Urq95**].

We have introduced proof systems briefly in chapter 2 (see page 25). A *proof system* (PS) for a language L is a polytime relation V such that $x \in L \iff \exists p V(x,p)$. Here V is called the *verifier* and p is the encoding of a proof that x is in L. *Soundness* means that $\exists p V(x,p) \Rightarrow x \in L$, and *completeness* means that $x \in L \Rightarrow \exists p V(x,p)$.

The *complexity* of V is $f_V : \mathbb{N} \longrightarrow \mathbb{N}$, where

$$f_V(n) = \max_{\{x \in L, |x|=n\}} \min_{\{p, V(x,p)\}} |p|,$$

and V is *polynomially bounded (polybounded)* iff f_V is bounded by some polynomial, i.e., there exists a polynomial q such that $f_V(n) \leq q(n)$.

As was mentioned in chapter 2, the canonical example of a language with a polybounded PS is SAT. However, the proof systems in which we are most interested are those for TAUT and UNSAT, where UNSAT is the complement of SAT. Thus, we make the following definition: a *propositional proof system* (PPS) is a proof system for TAUT or UNSAT.

THEOREM 6.1 (Cook-Reckhow). A polybounded propositional proof system (PPS) exists iff NP = co-NP.

PROOF. ($\Rightarrow$) Suppose that NP = co-NP, and so TAUT is in NP. Let M be the nondeterministic polybounded TM that decides TAUT. Let $V_M(\phi, p)$ be true iff $p = \langle C_1, C_2, \ldots, C_m \rangle$ encodes an accepting computation of M on ϕ.

---

[1] Boole's work seemed to have no practical application at the time of its invention, but seventy years later, Claude Shannon (see §5.2) in his MIT masters thesis showed how Boolean algebra could optimize the design of systems of electromechanical relays then used in telephone routing switches.

($\Leftarrow$) Suppose that there exists a polybounded PPS, V. Let M be the following nondeterministic TM: on input ϕ, M guesses a p and verifies $V(\phi, p)$. Thus M decides TAUT, and so TAUT is in NP. □

We say that a propositional proof system V is *automatizable*, if there exists an algorithm A, such that given a ϕ in TAUT as input, A outputs a proof p of ϕ in time polynomial in the length of the shortest proof of ϕ. (If $\phi \notin$ TAUT, then A outputs anything.)

LEMMA 6.2. There exists a polybounded automatizable PPS iff $\mathsf{P} = \mathsf{NP}$.

PROOF. ($\Rightarrow$) Suppose that there exists a polybounded automatizable PPS. Given any ϕ in TAUT we know that it has a proof of polynomial size, and that we can find a proof of ϕ in time polynomial in its length. As TAUT is co-NP-complete, we get that $\mathsf{P} = \mathsf{co\text{-}NP}$ and, what follows, $\mathsf{P} = \mathsf{NP}$.

($\Leftarrow$) Let us now assume that $\mathsf{P} = \mathsf{NP}$. Then SAT can be decided by some deterministic polytime Turing machine M. Take any formula ϕ and notice, that it is a tautology iff $\neg\phi$ is not satisfiable. Consider the computation of M on $\neg\phi$. M finishes its work in polynomial time, thus it can use only polynomial space. Therefore the encoding of the computation has polynomial size. But this encoding is a proof that $\phi \in$ TAUT (we can verify it in polynomial time, simply checking that it is a correct computation of M on ϕ), and it can be produced in polynomial time (as M produces it). Thus we have found a polybounded automatizable PPS. □

Let BF be the class of languages recognizable by a family of polynomial size Boolean Formulas. That is, $L \in \mathsf{BF}$ iff there exists a family of Boolean formulas $\Phi = \{\phi_n\}$ such that $\phi_n = \phi_n(x_1, x_2, \ldots, x_n)$, i.e., ϕ_n has n variables, and there is a fixed polynomial p such that $|\phi_n| \leq p(n)$, and

$$a \in L \iff \phi_{|a|}(a_1, a_2, \ldots, a_{|a|}) \text{ is true.}$$

This is analogous to the definition of a language being recognizable by a family of circuits given on page 65.

THEOREM 6.3 (Spira). $\mathsf{NC}^1 = \mathsf{BF}$.

PROOF. Consider an NC^1 circuit; it can be represented by a Boolean formula, by transforming (efficiently) each circuit in the family to be tree-like (a tree-like circuit is just a Boolean formula). To this end, we traverse the circuit from its output towards the inputs, repeating sub-circuits that have been used multiple times. But as the depth of the circuit is bounded by $O(\log n)$, and each gate had at most 2 inputs, the total size of the resulting formula will be bounded by $2^{O(\log n)} = n^{O(1)}$.

We now show by induction that any Boolean formula having no more than n logical connectives (from among $\{\wedge, \vee, \neg\}$) can be represented by a (fan-in 2) circuit of depth not greater than $4\log(n+1)$.

For $n = 0$ (a single variable) it is obviously true.

Fix any Boolean function ϕ with n connectives. The tree representing this formula has n internal nodes. Denote by $s(T)$ the size (number of nodes) of subtree T. Consider the subtree T of smallest size larger than $n/3$. It must be that $n/3 <$

$s(T) \leq 2n/3$ (otherwise one of the subtrees of T would be smaller, while still larger than $n/3$).

Let $\phi = (T \wedge \phi(1/T)) \vee (\neg T \wedge \phi(0/T))$. The formula T has size less than $2n/3$. When we replace T in ϕ by 0 or 1, we will get a formula of size bounded by $2n/3$ (because the size of T was at least $n/3$). Both these formulas can be represented (according to inductive assumption) by circuits of depth bounded by

$$4\log(2n/3+1) \leq 4(\log 2/3 + \log(n+1)) \leq 4\log(n+1) - 2.$$

Thus, we can represent our original formula ϕ by a circuit of depth bounded by

$$4\log(n+1) - 2 + 2 = 4\log(n+1).$$

Our circuit has logarithmic depth and constant fan-in, so its size will be polynomial. □

6.2. Resolution

We start with some definitions. A *literal* is a variable x, or its negation, $\bar{x}$. A *clause* is a set of literals, $\{l_1, l_2, \ldots, l_k\}$. A truth assignment τ *satisfies* a clause C, written $\tau \vDash C$, if it makes at least one literal in C true. A (finite) set of clauses $\mathcal{S}$ is *satisfiable* if there exists a truth assignment τ that satisfies all the clauses in $\mathcal{S}$.

For example, $\mathcal{S} = \{\{x, \bar{y}, \bar{z}\}, \{\bar{x}\}, \{y, z\}\}$ is satisfiable, and the truth assignment τ given by $\tau(x) = \tau(y) = 0, \tau(z) = 1$ satisfies it. On the other hand, $\mathcal{S} = \{\{x\}, \{\bar{x}, y\}, \{\bar{y}, z\}, \{\bar{z}\}\}$ is unsatisfiable.

The *Pigeonhole Principle*, PHP^n_{n-1}, $n > 1$, is the set of clauses given by:

(1) $\{P_{i1}, P_{i2}, \ldots, P_{i(n-1))}\}$, $1 \leq i \leq n$,

(2) $\{\bar{P}_{ik}, \bar{P}_{jk}\}$, $1 \leq i < j \leq n, 1 \leq k \leq (n-1)$.

Note that PHP^n_{n-1} is a set of $n + \binom{n}{2} \cdot (n-1) = O(n^3)$ clauses, and PHP^n_{n-1} is unsatisfiable for every n, because if it were satisfiable, a τ that satisfies it would effectively give us an injective relation on $[n] \times [n-1]$, which is not possible.

Resolution is a propositional proof system for the language of unsatisfiable set of clauses. Given two clauses $C \cup \{x\}, D \cup \{\bar{x}\}$, the *resolution rule* permits us to conclude $C \cup D$ from them. Here $C \cup \{x\}$ indicates that there are no x's in C; however, it may be the case that $\bar{x} \in C$ (and $D \cup \{\bar{x}\}$ indicates that $\bar{x}$ is not in D, but x may be present in D). Note that the resolution rule is *sound*: if τ satisfies $C \cup \{x\}, D \cup \{\bar{x}\}$, then τ must also satisfy $C \cup D$.

A *resolution refutation* (RR) of an unsatisfiable set of clauses $\mathcal{S}$ is a sequence of clauses $C_1, C_2, \ldots, C_n$, where:

(1) each C_i is either in $\mathcal{S}$, or follows by the resolution rule from some C_j, C_k, $j, k < i$,

(2) $C_n = \{\}$, i.e., it is the empty clause.

The *size* of a RR is the number of clauses in it.

For example, consider the pigeonhole principle for $n = 2$, that is,

$$\mathrm{PHP}_1^2 = \{\{P_{11}\}, \{P_{21}\}, \{\bar{P}_{11}, \bar{P}_{21}\}\}.$$

Its RR is:

$$\{P_{11}\}, \{P_{21}\}, \{\bar{P}_{11}, \bar{P}_{21}\}, \{\bar{P}_{21}\}, \{\},$$

and its size is 5.

Resolution is both *sound* and *complete*. To see that it is sound, note that the resolution rule is sound, so it can be shown inductively that if there is a τ that satisfies $\mathcal{S}$, then τ must also satisfy every C_i in the refutation, and in particular $C_n = \{\}$, which is not possible (because to satisfy a clause, τ must make at least one literal in the clause true, and the empty clause $\{\}$ has no literals). So if we can derive $\{\}$ from $\mathcal{S}$, we know that $\mathcal{S}$ cannot be satisfiable.[2]

To show completeness, we use the Davis-Putnam algorithm. The first version, known as DPLL, gives us *tree-like* refutations; the second version, known as DP, gives us *dag-like* refutations.

A tree-like refutation is one where each clause in the proof, except for the original clauses, is used at most once as a premise of a rule. A dag-like refutation can be transformed into a tree-like one by re-deriving a clause each time it is used as a premise; of course, this may give rise to an exponential blow up in size.

6.2.1. DPLL and DP. The most prominent PPS for UNSAT is DPLL (Davis-Putnam-Logemann-Loveland).

ALGORITHM 6.4 (DPLL).
On input $\mathcal{S}$ (a set of clauses):
We pick an $x \in \mathrm{var}(\mathcal{S})$, and branch out on $x = 0$ and $x = 1$. On each branch we continue selecting new variables, until one of two things happen:

(1) a leaf corresponds to a (partial) truth assignment falsifying a clause, in which case we label it with such a clause, or

(2) on the path from that leaf to the root we examined all the variables, in which case we get a (full) truth assignment satisfying the original set of clauses.

If at the end each leaf is labeled with a clause, we know $\mathcal{S}$ is unsatisfiable.

LEMMA 6.5. DPLL is correct, that is, given any set of clauses $\mathcal{S}$, it outputs a refutation if $\mathcal{S}$ is unsatisfiable, and a truth assignment if it is satisfiable.

PROOF. Observe that $\mathcal{S}$ is unsatisfiable iff both $\mathcal{S}|_{x=0}$ and $\mathcal{S}|_{x=1}$ are unsatisfiable, for any $x \in \mathrm{Var}(\mathcal{S})$. $\mathcal{S}|_{x=t}$ denotes $\mathcal{S}$ where each instance of x has been replaced by the truth value $t \in \{0, 1\}$, and the result simplified. The simplification is obvious: if a literal l is made true by setting $x = t$, any clause containing it is eliminated altogether; if it is made false, the literal is eliminated from all the clauses containing it. So, an inductive proof on the number of variables in $\mathcal{S}$, shows the correctness of DPLL. □

[2]On the other hand, the convention is to make an empty set of clauses satisfiable, in fact by any assignment, because no assignment can falsify a clause in an empty set of clauses.

We can now prove the completeness of Resolution.

LEMMA 6.6. A DPLL refutation yields a RR of the same size (in terms of the number of nodes).

PROOF. We can transform a DPLL refutation into a RR. We imagine the tree "upside-down," with the leaves at top, and the root at the bottom, and the leaves are already labeled with clauses (each leaf is labeled by some clause falsified by the partial truth assignment implicit in the branch ending at the leaf—if several clauses are falsified by this truth assignment, we pick any one of them).

If an internal node is labeled with variable x, the idea is to now label it with the clause resulting from resolving on its two parents nodes (to which we already, inductively, assigned clauses) on x. The resulting refutation is tree-like.

More formally, suppose that n is a node with parents n_1, n_2 above it that have clauses C_{n_1}, C_{n_2}, respectively, attached to them. If n was labeled with variable x (in the DPLL tree), it is now labeled with C_n, which is obtained as follows: (i) if $x \in C_{n_1}, \bar{x} \in C_{n_2}$, then C_n is the result of resolving C_{n_1}, C_{n_2} on x. (ii) If neither contains x or $\bar{x}$, let C_n be C_{n_1} (we could have made it C_{n_2}; the point is, it does not matter). It is not possible for both to contain x or for both to contain $\bar{x}$, as the next invariant shows.

The clause C_n is falsified by the (partial) truth assignment given by the path from the root to node n. Thus, if C_{n_1}, C_{n_2} both had x, or both had $\bar{x}$, then one of them would be satisfied by the partial truth assignment. From the construction given in the above paragraph it is easy to show this invariant by induction. □

For example, consider the set of clauses $\{\{x\}, \{\bar{x}, y\}, \{\bar{y}, z\}, \{\bar{z}\}\}$. In figure 1 we show a DPLL to resolution transformation; in brackets we have the corresponding resolution clauses.

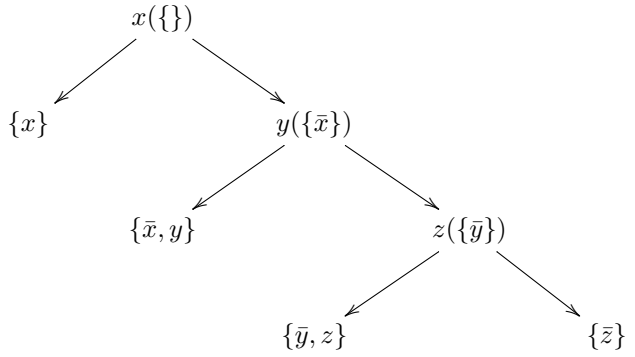


FIGURE 1. From DPLL to resolution.

So Resolution completeness (in fact, completeness of tree-like Resolution) follows from the correctness of DPLL. We can also go efficiently in the reverse direction; from tree-like Resolution to DPLL. However, there is one technical difficulty: a variable in a RR might be resolved on more than once on the same path!

We say that a RR is *regular* if on every path from the leaves to the root ($\{\}$) each variable is resolved on at most once.

THEOREM 6.7. A tree-like RR can be transformed into a regular tree-like RR efficiently.

PROOF. Suppose that we have a path in the refutation labeled by the clauses $C_1, C_2, \ldots, C_k$, $k > 2$, and there is a literal l in C_1 and in C_k, but l is not in any C_j, $1 < j < k$.

(Note that C_i is a premise for C_{i+1}; that's what being a path means here; also C_1 is *not necessarily* a leaf and C_k is *necessarily not* $\{\}$!)

$$\rightsquigarrow l \in C_4 \rightsquigarrow l \notin C_3 \rightsquigarrow l \notin C_2 \rightsquigarrow l \in C_1 \rightsquigarrow$$

FIGURE 2. Tree-like Resolution to regular tree-like Resolution.

Simply discard the first resolution on l. We now have to modify all the C_i's all the way down to $\{\}$. Here is how to do this. In figure 2 discard the right-parent-subtree of C_2, and let $C_2' := C_1$. Since we discarded the right-parent-subtree of C_2, some literals will no longer be in the C_i''s as we make our way to $\{\}$.

Say l_D is such a literal, and in the original refutation we resolve on some C_i ($i > 2$) and some E to get rid of it. Since l_D is no longer there, we simply discard E, and the subtree rooted at E, and let $C_i' := C_{i-1}'$. We follow through all the way to $\{\}$, making such adjustments.

CLAIM 6.8. For all $2 \leq i \leq k$, $C_i' \subseteq C_i \cup \{l\}$, and for $i > k$, $C_i' \subseteq C_i$.

This finishes the proof. □

COROLLARY 6.9. A minimal tree-like RR is regular.

ALGORITHM 6.10 (DP).
On input $\mathcal{S}$:

1. While ($\mathcal{S} \neq \emptyset$ and $\{\} \notin \mathcal{S}$)
2. Choose any $x \in \mathrm{Var}(\mathcal{S})$
3. $\mathcal{S} \leftarrow \mathcal{S} - \{C \in \mathcal{S} : \{x, \bar{x}\} \subseteq C\}$
4. $T \leftarrow \{C \cup D : C \cup \{x\}, D \cup \{\bar{x}\} \in \mathcal{S}\}$
5. $\mathcal{S} \leftarrow T \cup \{C \in \mathcal{S} : \{x, \bar{x}\} \cap C = \emptyset\}$
6. If $\mathcal{S} = \emptyset$ then return "satisfiable"
7. else return "unsatisfiable"

Notice that the DP algorithm produces a dag-like RR, unlike DPLL which produces a tree-like RR.

CLAIM 6.11. DP is correct.

PROOF. If the input $\mathcal{S}$ is an unsatisfiable set of clauses, then the set of clauses resulting from each iteration remains unsatisfiable (this is the *loop invariant*).

Let $\mathcal{S}'$ be the result of one iteration on $\mathcal{S}$. Suppose $\sigma \vDash \mathcal{S}'$. Then, one of the following two holds:

(1) $\sigma \vDash C$ for all C such that $C \cup \{x\} \in \mathcal{S}$, or

(2) $\sigma \vDash D$ for all D such that $D \cup \{\bar{x}\} \in \mathcal{S}$.

Otherwise, $\sigma \nvDash C_0$ and $\sigma \nvDash D_0$, so $\sigma \nvDash C_0 \cup D_0$, contradicting that $\sigma \vDash \mathcal{S}'$.

Therefore, if 1., extend σ to $\hat{\sigma}$ such that $\hat{\sigma}(x) = 0$, and if 2., $\hat{\sigma}(x) = 1$. Clearly, $\hat{\sigma} \vDash \mathcal{S}$; we continue inductively, and show that the original set of clauses $\mathcal{S}$ must be satisfiable. □

The following result indicates that resolution is a general PPS; that is, it can be applied to any Boolean tautology (i.e., not only those in CNF).

LEMMA 6.12. A general Boolean formula ϕ (not necessarily in CNF), can be transformed in polynomial time into a formula in CNF ϕ' such that, ϕ is satisfiable iff ϕ' is satisfiable.

PROOF. Introduce a new variable q_A for any subformula A of ϕ. Let $\mathcal{C}(A)$ be a set of clauses with the property that $\mathcal{C}(A) \cup \{q_A\}$ is satisfiable iff A is satisfiable, and, furthermore, $\mathcal{C}(A)$ holds iff the truth values associated with the variables representing all the subformulas of A are computed correctly in terms of the input values. We define $\mathcal{C}(A)$ inductively on A: if $A = x$, then $\mathcal{C}(A) := \{\{q_A, \bar{x}\}, \{\bar{q}_A, x\}\}$.

If $A = \neg B$, then $\mathcal{C}(A) := \{\{q_A, q_B\}, \{\bar{q}_A, \bar{q}_B\}\} \cup \mathcal{C}(B)$.

If $A = B_1 \vee B_2$, then $\mathcal{C}(A) := \{\{\bar{q}_A, q_{B_1}, q_{B_2}\}, \{q_A, \bar{q}_{B_1}\}, \{q_A, \bar{q}_{B_2}\}\} \cup \mathcal{C}(B_1) \cup \mathcal{C}(B_2)$. Similarly for conjunction. Also need to show inductively that the correctness condition holds. Finally, note that $|\mathcal{C}(A)| = O(|A|)$. □

Suppose that we managed to perform the above transformation *without* introducing any new variables; what would be the consequence of that for propositional proof systems? If we could translate an arbitrary Boolean formula into an equivalent CNF *without* adding new variables and with at most a polynomial increase in size, we would effectively have a polybounded PPS (and consequently $\mathsf{NP} = \mathsf{co\text{-}NP}$). The reason is that a formula in CNF is a tautology iff each clause contains some variable x and its negation; this can be checked in linear time in the length of the formula.

Extended Resolution (ER) is defined as the usual Resolution PPS, together with the extension rule which allows us to abbreviate formulas by new variable names. That is, we allow the introduction of new variables v_{new}, and a definition $v_{\text{new}} \leftrightarrow \alpha$; now we can refer to α by the new name v_{new}.

A clausal form for $v_{\text{new}} \leftrightarrow \alpha$ can be obtained by translating the formula to the equivalent formula $(\neg v_{\text{new}} \vee \alpha) \wedge (v_{\text{new}} \vee \neg\alpha)$.

THEOREM 6.13 (Cook). ER proves PHP_n^{n+1} in polynomial size.

PROOF. This result was shown in [**Coo76**]. The idea is to argue by contradiction: assume that PHP_n^{n+1} holds, and then deduce PHP_{n-1}^n from it. Repeating the process arrive at PHP_1^2, which is easy to refute. Each step of the induction requires polynomially many formulas, so the whole refutation is polynomial in size.

Introduce variables, $B_{i,j}^n$ and let $B_{i,j}^n \leftrightarrow P_{i,j}$ for $i = 1, \ldots, n+1$ and $j = 1, \ldots, n$. Then, for $k = n-1, \ldots, 1$, introduce variables $B_{i,j}^k$, defined as follows

$$B_{i,j}^k \leftrightarrow B_{i,j}^{k+1} \vee (B_{i,k+1}^{k+1} \wedge B_{k+2,j}^{k+1}),$$

for $1 \leq i \leq k+1, 1 \leq j \leq k$.

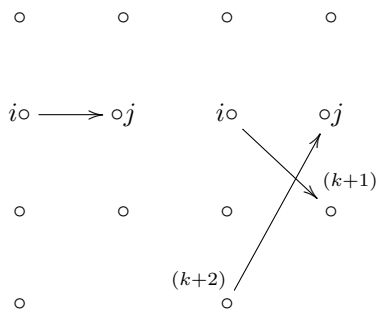


FIGURE 3. Definition of $B^k_{*,*}$ from $B^{k+1}_{*,*}$.

From clauses $\{B^{k+1}_{i,1}, \ldots, B^{k+1}_{i,k+1}\}$, for $i = 1, \ldots, k+2$, and $\{\neg B^{k+1}_{i,l}, \neg B^{k+1}_{j,l}\}$, for $l = 1, \ldots, k+1$, deduce the clauses $\{B^k_{i,1}, \ldots, B^k_{i,k}\}$, for $i = 1, \ldots, k+1$, and $\{\neg B^k_{i,l}, \neg B^k_{j,l}\}\}$, for $l = 1, \ldots, k$.

EXERCISE 6.14. Show how to do this.

At the end, refute the resulting set of clauses $\{\{B^1_{1,1}\}, \{B^1_{2,1}\}, \{\neg B^1_{1,1}, \neg B^1_{2,1}\}\}$, to finish the proof. □

LEMMA 6.15. Let 2SAT be the language of satisfiable formulas in CNF, where each clause has exactly 2 literals. Then, 2SAT in in P.

PROOF. When we resolve two clauses with at most 2 literals each, we obtain a clause with at most two literals: therefore, there are $\binom{2n}{2} = O(n^2)$ possible clauses to be obtained by resolving on the initial set of clause. Thus, we can decide 2SAT with great alacrity by computing all the possible resolvents (stopping to accept when no new clauses are being created), and rejecting if we ever produce the empty clause. □

6.3. A lower bound for resolution

This section consists in the proof of the following theorem.

THEOREM 6.16 (Haken). Any RR of PHP^n_{n-1} requires at least $2^{n/20}$ clauses.

A truth assignment (ta) σ is *i-critical* if σ leaves pigeon i out, and maps the remaining $(n-1)$ pigeons to holes bijectively. See figure 4 for an example.

Two clauses C_1, C_2 are *equivalent wrt critical ta's* if $\sigma \vDash C_1 \iff \sigma \vDash C_2$, for all critical σ. An inference $C_1, C_2 \vdash C_3$ is *sound wrt critical ta's* if whenever a critical σ satisfies C_1, C_2, it also satisfies C_3. Note that this "inference" is not necessarily an application of the resolution rule; C_1, C_2, C_3 are any three clauses with the property that if a critical ta satisfies C_1, C_2 it also satisfies C_3.

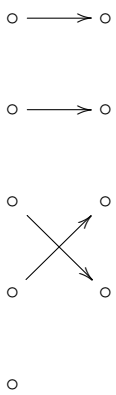


FIGURE 4. A 5-critical truth assignment.

Consider a resolution refutation $\mathcal{R}$ of PHP^n_{n-1}, and in every clause of $\mathcal{R}$ replace $\bar{P}_{ik}$ by the literals $\{P_{lk} | l \neq i\}$, obtaining thus $\hat{\mathcal{R}}$. All clauses in $\hat{\mathcal{R}}$ are *monotone*, equivalent to the corresponding original clauses wrt critical ta's, and all "inferences" in $\hat{\mathcal{R}}$ are sound wrt critical ta's. But note that $\hat{\mathcal{R}}$ is not a resolution refutation *per se*.

Define a clause to be *large* if it contains at least $n^2/10$-many variables, and let $S =$ *the number of large clauses in* $\hat{\mathcal{R}}$.

CLAIM 6.17. There is a P_{ij} contained in at least $S/10$-many large clauses of $\hat{\mathcal{R}}$.

PROOF. There are $< n^2$ variables, and suppose that each of them is in less than $S/10$ many large clauses. Now what is the largest collection of clauses we can form where each clause is required to be large and we have $< S/10$ "copies" of each of the n^2 variables?

$$< \frac{n^2 \cdot S/10}{n^2/10} = S.$$

Contradiction, since we *do* have S large clauses by assumption. □

We are now going to derive a contradiction from $S < 2^{n/20}$. Choose such a P_{ij}, set it to 1, and set to 0 all P_{il} and $P_{l'j}$, where $l \neq j$ and $l' \neq i$. Apply this restriction to $\hat{\mathcal{R}}$, to obtain a monotone refutation $\hat{\mathcal{R}}'$ of PHP^{n-1}_{n-2}. (Note the commutative diagram in figure 5.)

$$\begin{array}{ccc} \mathcal{R} \text{ for } \mathrm{PHP}^n_{n-1} & \xrightarrow{\bar{P}_{ik} \rightsquigarrow \{P_{lk} | l \neq i\}} & \hat{\mathcal{R}} \text{ for } \mathrm{PHP}^n_{n-1} \\ \downarrow & & \downarrow {\scriptstyle P_{ij}=1, \forall l \neq j, l' \neq i, P_{il}=0, P_{l'j}=0} \\ \mathcal{R}' \text{ for } \mathrm{PHP}^{n-1}_{n-2} & \longrightarrow & \hat{\mathcal{R}}' \text{ for } \mathrm{PHP}^{n-1}_{n-2} \end{array}$$

FIGURE 5. Commutative diagram.

The number of large clauses in $\hat{\mathcal{R}}'$ for PHP^{n-1}_{n-2} is at most $9S/10$. Repeat this $\log_{10/9} S$ many times until we knock out all large clauses. So we end up with a monotone RR of $\mathrm{PHP}^{n'}_{n'-1}$ where

$$n' \geq n - \log_{10/9} S > n(1 - (\log_{10/9} 2)/20) > 0.671n, \tag{19}$$

and where there are no large (i.e., of size $\geq n^2/10$) clauses.[3]

LEMMA 6.18. Any $\hat{\mathcal{R}}$ for PHP^n_{n-1} must have a clause with at least $2n^2/9$ literals (which by definition are all positive variables!).

From lemma 6.18 we get a contradiction, since by (19) we have that $n' > 0.671n$, and so $2(n')^2/9 > n^2/10$.

PROOF. (of lemma 6.18) For each C in $\hat{\mathcal{R}}$, let $\mathrm{Complex}(C)$ be the minimum number of clauses in PHP^n_{n-1} that imply C on all critical ta's. Note that only "pigeon" clauses $\{P_{i1}, P_{i2}, \ldots, P_{i(n-1)}\}$ are included in a minimal set implying C (as we restrict ourselves to critical ta's and clauses $\{\bar{P}_{ik}, \bar{P}_{jk}\}$ are always satisfied by critical ta's). The complexity of "pigeon" clauses is 1, and of the empty clause $\{\}$ it is n.

If $C_1, C_2 \vdash C_3$, then $\mathrm{Complex}(C_3) \leq \mathrm{Complex}(C_1) + \mathrm{Complex}(C_2)$.

CLAIM 6.19. $\exists C$, $n/3 < \mathrm{Complex}(C) \leq 2n/3$.

PROOF. Take C to be a clause of complexity greater than $n/3$. $\mathrm{Complex}(\{\}) = n$, so such a clause exists. If both parent clauses are of complexity at most $n/3$, we are done. Otherwise, pick the parent clause of complexity greater than $n/3$, and repeat. This process must end, since the input clauses are of complexity 0 or 1. □

Let P be a minimal subset of "pigeon" clauses that implies C, and let $m = |P| = \mathrm{Complex}(C)$.

CLAIM 6.20. $|C| \geq (n-m)m$.

Note that $(n-m)m \geq 2n^2/9$, since for $0 < n/3 \leq m \leq 2n/3$, as a function of m, it takes its minimum when at the extremes, i.e., when $m = n/3$ or $m = 2n/3$.

PROOF. (of claim 6.20) For any $\{P_{i1}, \ldots, P_{i(n-1)}\} \in P$, consider an i-critical α such that $\alpha \not\models C$. (If every i-critical α satisfied C, we would not need the clause $\{P_{i1}, \ldots, P_{i(n-1)}\}$ in P.)

For each $\{P_{j1}, \ldots, P_{j(n-1)}\} \notin P$ (remember $\mathrm{Complex}(C) \leq 2n/3$), let α' be α except $\alpha'(i) = \alpha(j)$, and α' is j-critical. We have that $\alpha' \vDash C$: since $\{P_{j1}, \ldots, P_{j(n-1)}\} \notin P$, and α' is j-critical, α' satisfies P, so it must satisfy C. But α and α' are the same on all variables, except on P_{jl} and P_{il}. Since $\alpha \not\models C$ but $\alpha' \vDash C$, it follows that $P_{il} \in C$.

---

[3] Actually, after $\log_{10/9} S$ many times you may still have one more large clause left, so really $n' > 0.671n - 1$. But the argument can still be made to work by taking n sufficiently large. More precisely, we are interested in $S \cdot (9/10)^i = 1$, so taking logarithms of both sides we obtain that $i = \frac{\log S}{\log 10/9} = \log_{10/9} S$. Now, $1 - \frac{\log_{10/9}(2)}{20} = 0.6710593\ldots$. If we take this constant, we see that for n sufficiently large (say more than a million), it is clear that $0.6710593n - 1 > 0.671n$.

Using the same α on all $j \notin P$ we get $(n-m)$ *distinct* variables

$$P_{il_1}, P_{il_2}, \ldots, P_{il_{n-m}}$$

in C. Now, repeating the whole argument for each $i \in P$, gives us that there must be at least $(n-m)m$ variables in C. □

This ends the proof of lemma 6.18. □

6.4. Automatizability and interpolation

If C is a clause, let the *width* of C, $w(C)$ denote the number of literals in C. Extend this definition to a set of clauses $\mathcal{S}$ in the obvious way: $w(\mathcal{S})$ is the largest width of all the clauses in $\mathcal{S}$. Also, if P is a resolution refutation, let $w(P)$ be the largest width of all the clauses in P. Finally, let $sw(\mathcal{S})$ be the smallest width of any resolution refutation of $\mathcal{S}$.

LEMMA 6.21. Any tree-like resolution refutation of $\mathcal{S}$ of size s can be converted to one of width bounded above by $(\lceil \log s \rceil + w(\mathcal{S}))$.

PROOF. We prove it by induction on s. If $s = 1$, $\mathcal{S} = \{\{\}\}$, so $\mathcal{S}$ is itself a resolution refutation of width $0 = \lceil \log 1 \rceil + w(\mathcal{S})$.

Now consider a tree-like resolution refutation P of $\mathcal{S}$ of size $s > 1$. Suppose that x is the last variable to be resolved, i.e., the last step of P is

$$\dfrac{\dfrac{\vdots}{\{x\}} \qquad \dfrac{\vdots}{\{\bar{x}\}}}{\{\}}. \tag{20}$$

The dots denote the left and right subtrees, respectively. We want to show that we can transform P to have width $w = \lceil \log s \rceil + w(\mathcal{S})$.

Note that one of the subtrees of (20) has to be of size $< s/2$. Assume that the subtree rooted at $\{\bar{x}\}$ is of size $< s/2$, and the other subtree, the one rooted at $\{x\}$ is of size $< s$. (The opposite case is symmetric.)

Let $\mathcal{S}|_{x=t}$ be the set of clauses $\mathcal{S}$, with x set to $t \in \{0,1\}$, and the natural simplifications done (if the assignment makes a literal false, eliminate it from all the clauses that have it, and if it makes a literal true, eliminate all the clauses that have it).

Thus, $\mathcal{S}|_{x=1}$ has the refutation $P|_{x=1}$ given by the right subtree of (20). By the induction hypothesis, $P|_{x=1}$ can be transformed to P' of width

$$\lceil \log s/2 \rceil + w(\mathcal{S}|_{x=1}) \leq \overbrace{\underbrace{\lceil \log s \rceil + w(\mathcal{S})}_{=w} - 1}^{=(*)}. \tag{21}$$

Now transform P' as follows: to every input clause that originally contained $\bar{x}$, add $\bar{x}$ back in and propagate it down the proof. This way, we obtain a derivation of $\{\bar{x}\}$ of width at most $1 + (*) = w$, from a subset of the original clauses (note that the clauses that contained x were eliminated, and clauses that contain neither $\bar{x}$ nor x

have not changed). How do we know that adding $\bar{x}$ to those input clauses ensures that $\bar{x}$ appears in the conclusion? Since $\bar{x}$ did appear in the conclusion of the original refutation P, there must have been a path from the conclusion of P (i.e., from $\{\bar{x}\}$) to an input clause containing $\bar{x}$; this path remains in P'.

Using the left subtree of (20), we obtain a refutation P'' of $\{\}$ from $\mathcal{S}|_{x=0}$ of width at most w (by induction hypothesis; recall that the left subtree of (20) has size $< s$). But we can obtain $\mathcal{S}|_{x=0}$ from $\mathcal{S}$ using P' to eliminate all the x's from the clauses of $\mathcal{S}$ which have x, and simply ignoring the clauses that have $\bar{x}$. This way, we get a refutation of $\mathcal{S}$ of width w. □

COROLLARY 6.22. Any tree-like resolution refutation of $\mathcal{S}$ requires size

$$\Omega(2^{sw(\mathcal{S})-w(\mathcal{S})}).$$

PROOF. Follows directly from $sw(\mathcal{S}) \leq \lceil \log s \rceil + w(\mathcal{S})$. □

Let $f : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ be a function. We say that a propositional refutation system V is $f(n,s)$-*automatizable* iff there exists an algorithm A_V which on input $\mathcal{S}$, $|\mathcal{S}| = n$, outputs a refutation P of $\mathcal{S}$ in time at most $f(n,s)$ where s is the size of the shortest refutation of $\mathcal{S}$. (Note that we use $\mathcal{S}$, since we are thinking of clauses, but this definition is more general.)

LEMMA 6.23. For k-CNF tree resolution is $s^{O(\log n)}$-automatizable.

EXERCISE 6.24. Prove lemma 6.23.

Let $\alpha(\vec{p},\vec{q}) \wedge \beta(\vec{p},\vec{r})$ be an unsatisfiable CNF formula. A *Craig interpolant* (just *interpolant* from now on) is a function C such that given a value assignment $\vec{p}_0$ to $\vec{p}$:

$$C(\vec{p}_0) = \begin{cases} 0 & \alpha(\vec{p}_0,\vec{q}) \quad \text{is unsatisfiable} \\ 1 & \beta(\vec{p}_0,\vec{r}) \quad \text{is unsatisfiable} \end{cases}.$$

LEMMA 6.25. If for every unsatisfiable formula $\alpha(\vec{p},\vec{q}) \wedge \beta(\vec{p},\vec{r})$ there exists a polytime interpolant (i.e., there is a polytime algorithm computing C), then $\mathsf{NP} \cap \mathsf{co\text{-}NP} \subseteq \mathsf{P/poly}$.[4]

PROOF. Let L be a language in $\mathsf{NP} \cap \mathsf{co\text{-}NP}$. For inputs of a given length n, let $\alpha(\vec{p},\vec{q})$ code the statement "$\vec{q}$ is a witness that $\vec{p}$ is in L", and let $\beta(\vec{p},\vec{r})$ code the statement "$\vec{r}$ is a witness that $\vec{p}$ is *not* in L". (Such formulas exist since $\mathsf{NP} \cap \mathsf{co\text{-}NP} = \Sigma_1^p \cap \Pi_1^p$.)

Since for any pair (α,β) we have a polytime interpolant C, we also have a polysize circuit family $S = \{S_i\}$, where S_i implements C on inputs of length i, and S decides L: on input w, $|w| = n$, compute $S_n(w)$, and accept iff $S_n(w) = 1$, i.e., iff $\alpha(w,\vec{q})$ is satisfiable. A different circuit interpolant exists for different input lengths; we do not know how to generate the interpolants, but we know they are of polynomial size. □

Let $f : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ be a function. We say that a propositional refutation system V has $f(n,s)$-*interpolation* iff given $\alpha(\vec{p},\vec{q}) \wedge \beta(\vec{p},\vec{r})$ (of size n) with minimum refutation size s, there exists a circuit of size at most $f(n,s)$ computing the interpolant

[4]Note that the most prominent problem to be in $\mathsf{NP} \cap \mathsf{co\text{-}NP}$ is factoring; see [**Pap94**, §10.3.].

C for $\alpha(\vec{p},\vec{q}) \wedge \beta(\vec{p},\vec{r})$. We say that V has *feasible interpolation* if f is bounded by a polynomial (in n, s), and *monotone interpolation* if whenever $\vec{p}$ occur only positively in α (or, dually, only negatively in β) the circuit computing the interpolant is monotone (i.e., it has only `And` and `Or` gates).

LEMMA 6.26. If a polybounded refutation system V has feasible interpolation, then $\mathsf{NP} \subseteq \mathsf{P}/\mathrm{poly}$.

EXERCISE 6.27. Prove lemma 6.26.

Recall that a *clique* of size m is a subset of m vertices, which is fully connected (i.e., there is an edge between every pair of vertices), and a *co-clique* of size m is a partition of the vertices of the graph into m sets, so that all edges are between sets, and no edges within any single set. In other words, there is a co-clique of size m iff the chromatic number of the graph is less than m.

THEOREM 6.28 (Razborov). There exists an ε such that for sufficiently large n, and $m = \frac{n}{10}$, any monotone circuit which outputs a 1 on all m-cliques, and a 0 on all $(m-1)$-co-cliques, requires size 2^{n^ε}.

LEMMA 6.29. If V has monotone feasible interpolation, then V is not polynomially bounded.

PROOF. Consider the formula $\alpha(\vec{p},\vec{q}) \wedge \beta(\vec{p},\vec{r})$, where $\vec{p}$ encodes an undirected graph G over n vertices; let (p_{ij}) be the adjacency matrix of a graph, $i, j \in [n]$. We show how to construct $\alpha(\vec{p},\vec{q})$ which asserts that $\vec{p}$ has a clique of size m, and $\beta(\vec{p},\vec{r})$ which asserts that $\vec{p}$ has a co-clique of size $(m-1)$.

We use $\vec{q}$ to describe a clique of size m. Let q_{ij}, for $i \in [n], j \in [m]$ assert that vertex i is the j-th vertex of the clique. We can state that $\vec{p}$ has an m-clique with the following clauses:

(1) For each $j \in [m]$, we have the clause $\{q_{1j}, \ldots, q_{nj}\}$ asserting that some vertex is the j-th vertex of the clique.

(2) For each pair $j \neq j'$ in $[m] \times [m]$, and for each $i \in [n]$, we have the clause $\{\bar{q}_{ij}, \bar{q}_{ij'}\}$ asserting that no vertex is placed in the clique twice.

(3) For each pair $j \neq j'$ in $[m] \times [m]$, and for each $1 \leq i < i' \leq n$, we have $\{p_{ii'}, \bar{q}_{ij}, \bar{q}_{i'j'}\}$, asserting that if two vertices are in the clique, they must be connected by an edge.

All these clauses taken together (i.e., their conjunction) make up α. Note that $\vec{p}$ occur in α only positively.

Now let r_{ij} state that vertex $i \in [n]$ is in the j-th group of the partition, $j \in [m-1]$. The following clauses assert that we have a $(m-1)$-co-clique:

(1) For $i \in [n]$ we have clauses $\{r_{i1}, \ldots, r_{i(m-1)}\}$, asserting that every vertex belongs to some group.

(2) For each pair $i \neq i'$ in $[n] \times [n]$ and $1 \leq j \leq (m-1)$ we have clauses $\{\bar{r}_{ij}, \bar{r}_{i'j}, \bar{p}_{ii'}\}$, asserting that any two vertices in the same group are not connected by an edge.

The conjunction of these clauses makes up β, and $\vec{p}$ occur in β only negatively.

Finally, a feasible monotone interpolant for $\{\alpha \wedge \beta\}_n$, where $m = \frac{n}{10}$, contradicts Razborov's theorem, unless V is not polynomially bounded. □

LEMMA 6.30. Resolution has monotone feasible interpolation.

PROOF. Suppose that P is a resolution refutation of $\alpha(\vec{p}, \vec{q}) \cup \beta(\vec{p}, \vec{r})$, where the p_i's occur only positively in α (the dual case, only negatively in β, is analogous).

Let σ be a truth value assignment to $\vec{p}$. We show how to transform P to obtain a refutation $P|_\sigma$ and at the same time construct a feasible monotone interpolant $\mathbf{I}(\sigma)$. If C was a clause in the original refutation, it will be denoted C' after applying the procedure, and $\mathbf{I}_C(\sigma)$ will be the interpolant associated with clause C.

We say that a clause C' is an α-clause if it contains variables from among $\vec{p}, \vec{q}$ only, and if it only contains variables from $\vec{p}$, it is an α-clause if all its ancestors are α-clauses. Similarly, we define a β-clause symmetrically, with $\vec{r}$ and negations of $\vec{p}$.

We let $\mathbf{I}_C(\sigma)$ be 0 if C' is an α-clause, and 1 if it is a β-clause. Initially, every clause C in α and β stays the same, i.e., $C' = C$, all clauses in α are declared to be α-clauses, and all clauses in β are declared to be β-clauses. We now describe the rest of the procedure.

Suppose a clause C in P is obtained by:

$$\frac{C_1 \cup \{x\} \qquad C_2 \cup \{\bar{x}\}}{C}. \tag{22}$$

Assume inductively that we have already transformed the premises, and we have obtained $(C_1 \cup \{x\})', (C_2 \cup \{\bar{x}\})'$, and we have also declared their sides.

The task is to define C' and $\mathbf{I}_C(\sigma)$. We do it separately for x being a variable in one of the three groups: p_i, q_i, r_i.

In the case when $x = p_i$, let C' be $(C_1 \cup \{p_i\})'$ if $p_i = 0$, and $(C_2 \cup \{\bar{p}_i\})'$ otherwise. One might be tempted to define

$$\mathbf{I}_C := (p_i \vee \mathbf{I}_{(C_1 \cup \{p_i\})}) \wedge (\bar{p}_i \vee \mathbf{I}_{(C_2 \cup \{\bar{p}_i\})}). \tag{23}$$

But we have to be *"Unmoved, cold, and to temptation slow"*;[5] (23) would work if we were not constructing a monotone circuit for the interpolant, but we are, so $\bar{p}_i$ is not allowed. To fix this, keep in mind that one of the goals of the definition of C' is to ensure that it is either an α-clause or a β-clause. Consider the truth table for $\mathbf{I}_C$, given in figure 6, when defined the wrong way as in (23).

Note the two "problem rows," where I_C goes from 1 to 0, despite the fact that p_i changes from 0 to 1. This is the only case where monotonicity is spoiled. But, using the values of $\mathbf{I}$ on the premises, we can make this problem vanish. The point is that we can turn the 1 in the box into a 0, while defining C' consistently. (Note that a "symmetric" answer is possible: turn the 0 below the boxed 1 into a 1.)

Let $\mathbf{I}_C := (p_i \vee \mathbf{I}_{(C_1 \cup \{p_i\})}) \wedge \mathbf{I}_{(C_2 \cup \{\bar{p}_i\})}$, and we let C' be $(C_1 \cup \{p_i\})'$ or $(C_2 \cup \{\bar{p}_i\})'$ as defined the wrong way, except that if $(C_2 \cup \{\bar{p}_i\})'$ is an α-clause, we let C' be equal to it, regardless of the value of p_i. This works, because if $(C_2 \cup \{\bar{p}_i\})'$ is an α-clause, then by definition it cannot have $\bar{p}_i$ in it (it must have disappeared along the way).

[5] Shakespeare, Sonnet XCIV.

p_i	$\mathbf{I}_{(C_1 \cup \{p_i\})}$	$\mathbf{I}_{(C_2 \cup \{\bar{p}_i\})}$	$\mathbf{I}_C$
0	0	0	0
1	0	0	0
0	1	0	$\boxed{1}$
1	1	0	0
0	0	1	0
1	0	1	1
0	1	1	1
1	1	1	1

FIGURE 6. Truth table for $\mathbf{I}_C$.

For $x = q_i$, let C' be the result of resolving $(C_1 \cup \{q_i\})'$ and $(C_2 \cup \{\bar{q}_i\})'$ on q_i if possible (meaning that one still has q_i and the other $\bar{q}_i$), and taking C' to be the β-clause, if one of them is (if they both are, say, the left one), and if neither is a β-clause, take the one without the q_i-literal (say, the left one, if they both miss the q_i-literal). Let $\mathbf{I}_C := \mathbf{I}_{C_1 \cup \{q_i\}} \vee \mathbf{I}_{C_2 \cup \{\bar{q}_i\}}$. The case of $x = r_i$ is the dual case to the q_i, and so the interpolant is defined with a conjunction.

We now show that if C' is an α-clause, then $\alpha|_\sigma \vDash C'|_\sigma$, and same for β-clauses. This can be done with an inductive argument on the depth of the clause.

Basis Case: the claim is clearly true for the input clauses. Suppose that we are deriving as in (22), with $x = q_i$. Suppose that $\hat{\sigma} \supseteq \sigma$ is a truth assignment to all the variables, extending the truth assignment σ to the $\vec{p}$. Suppose both premises are α-clauses, and $\hat{\sigma}$ satisfies them. If C' is the result of resolution on q_i, then by soundness of the resolution rule it is satisfied by $\hat{\sigma}$.

EXERCISE 6.31. Show what to do in the case when x is p_i and r_i.

Define the interpolant for P to be $\mathbf{I}_{\{\}}$, i.e., the value of the interpolant at the final empty clause. It is monotone because we only use $\{\wedge, \vee, 0, 1\}$ and variables (no negations). It is feasible since the number of connectives is linear in the size of the proof P. It works correctly because it establishes that $\{\}' = \{\}$ is an α- or β-clause, and so $\{\}$ is logically implied by either α or β clauses, and so α or β must be unsatisfiable. □

Lemmas 6.29 and 6.30 give us the following corollary, which is an alternative proof of a lower bound for resolution. Note that our original lower bound, given in §6.3, is much simpler; the lower bound just given requires the machinery of interpolation, plus Razborov's lower bound for monotone circuits computing CLIQUE.

COROLLARY 6.32. Resolution is not polynomially bounded.

6.5. Answers to selected exercises

Exercise 6.24. If $w(\mathcal{S}) = k$, i.e., it is constant, we know from lemma 6.21 that a tree-like resolution refutation of size s can be transformed to be of width $O(\log s)$.

Given input size n, there can be at most n-many variables, and hence we can form at most $2^{O(\log s)}\binom{n}{O(\log s)} = n^{O(\log s)} = s^{O(\log n)}$-many clauses of width at most $O(\log s)$. Now, starting with $l = 1$, being careless with space, we use breadth-first search to generate all tree-like resolution refutations with leaves labeled by clauses in $\mathcal{S}$ and with clauses of width at most l. When we can no longer generate new clauses, we increase l by 1.

Exercise 6.27. Suppose that V has feasible interpolation and it is also polybounded. Consider the following two formulas: $\alpha(\vec{p}, \vec{q})$ asserting "$\vec{q}$ is a satisfying truth assignment for the propositional formula encoded by $\vec{p}$", and $\beta(\vec{p}, \vec{r})$ asserting "$\vec{r}$ encodes a V-refutation of $\vec{p}$". Obviously $\alpha \wedge \beta$ is not satisfiable. A polysize interpolant for these formulas would give us a polysize circuit for satisfiability, and hence NP would be contained in P/poly.

6.6. Notes

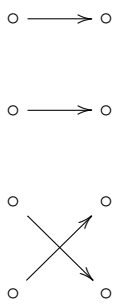
§6.3 is based on [**BP96**]. theorem 6.28 (i.e., Razborov's CLIQUE theorem) is presented in [**Pap94**, §14.4] (albeit, with a slightly different statement). §6.4 is based on [**Pit02**, lectures 6 and 7].

7

Randomized Classes

7.1. Three examples of randomized algorithms

7.1.1. Perfect matching. Consider a bipartite graph G, and its adjacency matrix defined as follows: $(A_G)_{ij} = x_{ij}$ iff $(u_i, v_j) \in E_G$. Then, G has a *perfect matching*



$$\begin{pmatrix} x_{11} & 0 & 0 & 0 \\ 0 & x_{22} & 0 & 0 \\ 0 & 0 & 0 & x_{34} \\ 0 & 0 & x_{43} & 0 \end{pmatrix}$$

FIGURE 1. A bipartite graph and its adjacency matrix.

(i.e., each vertex on the left may be paired with a unique vertex on the right) iff $\det(A_G) = \Sigma_\sigma \mathrm{sgn}(\sigma)\Pi_i (A_G)_{i\sigma(i)} \neq 0$.

Computing the symbolic determinant is computationally very expensive, so instead we randomly assign values to the x_{ij}'s. Let $A_G(x_1, \ldots, x_m)$, $m = |E_G|$, be A_G with its variables renamed to $x_1, \ldots, x_m$. Note that $m \leq n^2$ and each x_l represents some x_{ij}.

Choose m random integers $i_1, \ldots, i_m$ between 0 and $M = 2m$, and compute the integer determinant of $A_G(i_1, \ldots, i_m)$. If $\det(A_G(i_1, \ldots, i_m)) \neq 0$, then "yes," G has a perfect matching. If $\det(A_G(i_1, \ldots, i_m)) = 0$, then "no," G *probably* has no perfect matching. This is a polytime *Monte Carlo algorithm* where "yes" answers are reliable and final, while "no" answers are in danger of a *false negative*. In this case G might have a perfect matching, but unluckily $(i_1, \ldots, i_m)$ may happen to be roots of the polynomial $\det(A_G(x_1, \ldots, x_m))$.

Formally, N is a Monte Carlo TM for L if whenever $x \in L$, then at least half of the computations of N on x halt in "yes." If $x \notin L$, then all computations halt in "no." In other words, a Monte Carlo TM rejects "unanimously," and accepts "by majority."

LEMMA 7.1 (Schwarz-Zippel). Consider only polynomials over the ring $\mathbb{Z}$, and let $p(x_1, \ldots, x_m) \neq 0$ be a polynomial, where the degree of each variable is $\leq d$ (when the polynomial is written out as a sum of monomials), and let $M > 0$. Then the number of m-tuples $(i_1, \ldots, i_m) \in \{0, 1, \ldots, M-1\}^m$ such that $p(i_1, \ldots, i_m) = 0$ is $\leq mdM^{m-1}$.

PROOF. Induction on m (the number of variables). If $m = 1$, $p(x_1)$ can have at most $d \leq dM^0$ many roots, by the Fundamental theorem of Algebra.

Suppose the lemma holds for $(m-1)$, and suppose that $p(i_1, \ldots, i_m) = 0$. There are two cases to consider: express $p(x_1, \ldots, x_m)$ as $y_d x_m^d + \cdots + y_0 x_m^0$, where $y_i = y_i(x_1, \ldots, x_{m-1}) \in \mathbb{Z}[x_1, \ldots, x_{m-1}]$. The first case is the situation where $y_d = \cdots = y_0 = 0$, i.e., the "coefficients" of this polynomial are zero under some value assignment to the x_i's, and so p is zero under that assignment. The probability of this situation happening is certainly bounded above by the probability that $y_d = 0$ (i.e., only the first coefficient is zero under some value assignment). The second case is where $y_d \neq 0$, under some truth value assignment. Thus the probability that $p = 0$ is bounded above by the sum of the probabilities of these two cases.

We now compute the probability of each case. Case (1), the coefficient of the highest degree of x_m (this coefficient is in $\mathbb{Z}[x_1, \ldots, x_{m-1}]$) is zero. By the induction hypothesis, this coefficient is zero for at most $(m-1)dM^{m-2}$ many values, and x_m can take M values, and so the polynomial is zero for at most $(m-1)dM^{m-1}$ values. Case (2), for each combination of M^{m-1} values for $x_1, \ldots, x_{m-1}$, there are at most d roots of the resulting polynomial (again by the Fundamental theorem of Algebra), i.e., dM^{m-1}. Adding the two estimates gives us mdM^{m-1}. □

We apply lemma 7.1 to the Monte Carlo algorithm for matching given above, with $M = 2m$, and obtain that the probability of a false negative is less than or equal to:

$$\frac{m \cdot d \cdot M^{m-1}}{M^m} = \frac{m \cdot 1 \cdot (2m)^{m-1}}{(2m)^m} = \frac{m}{2m} = \frac{1}{2}.$$

Now suppose we perform "many independent experiments," meaning that we perform the above algorithm k many times. Then, if the answer always comes zero we know that the probability of error is $\leq \left(\frac{1}{2}\right)^k = \frac{1}{2^k}$. For $k = 100$, the error becomes *negligible*.

The integer determinant can be computed in NC^2 with Berkowitz's algorithm (this is shown in the Appendix, in §8.5), so this means that perfect matching is in $\mathsf{co\text{-}RNC}^2$ (see the definition of randomized circuit families on page 108). On the other hand, matching is in P; this can be easily seen by using a "max flow algorithm": add two new nodes s, t, and connect s to all the nodes in the left-column of the matching problem, and connect t to all the nodes in the right-column of the matching problem, and give each edge a capacity of 1, and ask if there is a flow $\geq n$ (where n is the number of nodes in each of the two components of the given bipartite graph) from s to t (see figure 2).

Counting the number of perfect matchings is complete for $\#\mathsf{P}$, while the decision problem itself is in P (it is usually the case, for problems which are complete for $\#\mathsf{P}$, such as #SAT, that the related decision problem is NP-complete).

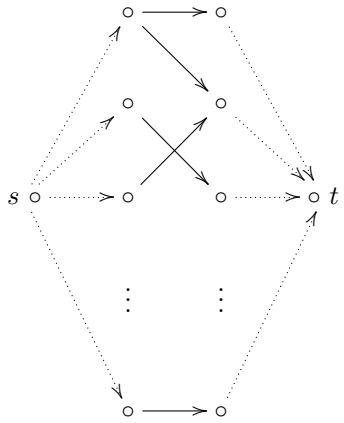


FIGURE 2. Reduction of matching to flow.

The family of Boolean functions f_{pm} computing the existence of a perfect matching (where the input variables denote the presence or absence of edges) is monotone (adding new edges can only improve the chances of the existence of a perfect matching). On the other hand, as we noted before, perfect matching is in P, so we know that we can compute f_{pm} with polysize circuits. It was shown by Razborov, however, that monotone circuits computing f_{pm} are necessarily of exponential size (thus, removing the negation gate increases dramatically the circuit size necessary to compute f_{pm}). This also shows that giving an exponential lower bound for a monotone circuit family deciding a language in NP is not enough to show the separation of P and NP (see theorem 6.28).

One final observation is that perfect matching is not known to be complete for any natural complexity class.

7.1.2. Primality testing. We present the Rabin-Miller randomized algorithm for primality testing. Although a polytime (deterministic) algorithm for primality is now known (see [**AKS04**]), randomized algorithms[1] are simpler and more efficient, and therefore still used in practice. Note that $(n)_b$ denotes the binary encoding of n.

ALGORITHM 7.2 (Rabin-Miller).
On input $(n)_b$:

1. If $n = 2$, accept; if n is even and $n > 2$, reject.
2. Choose at random a positive a in $\mathbb{Z}_n$.
3. If $a^{(n-1)} \not\equiv 1 \pmod{n}$, reject.
4. Find s, h such that s is odd and $n - 1 = s2^h$.
5. Compute the sequence $a^{s\cdot 2^0}, a^{s\cdot 2^1}, a^{s\cdot 2^2}, \ldots, a^{s\cdot 2^h} \pmod{n}$.

[1] In fact it was the randomized test for primality that stirred interest in randomized computation in the late 1970's. Historically, the first randomized algorithm for primality was given by [**SS77**]; a nice self-contained exposition of this algorithm can be found in [**Pap94**, §11.1], and another in [**vzGG99**, §18.5].

6. If all elements in the sequence are 1, accept.
7. If the last element different from 1 is -1, accept. Otherwise, reject.

Note that this is a polytime (randomized) algorithm: computing powers $\pmod n$ can be done efficiently with repeated squaring—for example, if $(n-1)_b = c_r \dots c_1 c_0$, then compute

$$a_0 \equiv_n a, a_1 \equiv_n a_0^2, a_2 \equiv_n a_1^2, \dots, a_r \equiv_n a_{r-1}^2,$$

and so $a^{n-1} \equiv_n a_0^{c_0} a_1^{c_1} \cdots a_r^{c_r}$. Thus obtaining the powers in lines 3 and 5 is not a problem. The highest power of 2 that divides $n-1$ is evident from $(n)_b$, so line 4 is not a problem either. Finally, choosing a non-zero $a \in \mathbb{Z}_n$ in a random way can be done by "flipping a coin" to obtain a string of bits of length $\log n$ (to ensure that a is not zero we choose at random the position of a 1 in the string, and then generate the other bits).

THEOREM 7.3. If n is a prime then the Rabin-Miller algorithm accepts it; if n is composite, then the algorithm rejects it with probability $\geq \frac{1}{2}$.

PROOF. If n is prime, then by Fermat's little theorem $a^{(n-1)} \equiv 1 \pmod n$, so line 3 cannot reject n. Suppose that line 7 rejects n; then there exists a b in $\mathbb{Z}_n$ such that $b \not\equiv \pm 1 \pmod n$ and $b^2 \equiv 1 \pmod n$. Therefore, $b^2 - 1 \equiv 0 \pmod n$, and hence

$$(b-1)(b+1) \equiv 0 \pmod n.$$

Since $b \not\equiv \pm 1 \pmod n$, both $(b-1)$ and $(b+1)$ are strictly between 0 and n, and so a prime n cannot divide their product. This gives a contradiction, and therefore no such b exists, and so line 7 cannot reject n.

If n is an odd composite number, then we say that a is a *witness* (of compositness) for n if the algorithm rejects on a. We show that if n is an odd composite number, then at least half of the a's in $\mathbb{Z}_n$ are witnesses. The distribution of those witnesses in $\mathbb{Z}_n$ appears to be very irregular, but if we choose our a at random, we hit a witness with probability $\geq \frac{1}{2}$.

Because n is composite, either n is the power of an odd prime, or n is the product of two odd co-prime numbers. This yields two cases.

Case 1. Suppose that $n = q^e$ where q is an odd prime and $e > 1$. Set $t := 1 + q^{e-1}$. From the binomial expansion of t^n we obtain:

$$t^n = (1 + q^{e-1})^n = 1 + nq^{e-1} + \sum_{l=2}^{n} \binom{n}{l} (q^{e-1})^l, \tag{24}$$

and therefore $t^n \equiv 1 \pmod n$. If $t^{n-1} \equiv 1 \pmod n$, then $t^n \equiv t \pmod n$, which from the observation about t and t^n is not possible, hence t is a line 3 witness. But the set of line 3 nonwitnesses, $S_1 := \{a \in \mathbb{Z}_n | a^{(n-1)} \equiv 1 \pmod n\}$, is a subgroup of $\mathbb{Z}_n^*$, and since it is not equal to $\mathbb{Z}_n^*$ (t is not in it), by Lagrange's theorem S_1 is at most half of $\mathbb{Z}_n^*$, and so it is at most half of $\mathbb{Z}_n$.

Case 2. Suppose that $n = qr$, where q, r are co-prime. Among all line 7 nonwitnesses, find a nonwitness for which the -1 appears in the largest position in the sequence in line 5 of the algorithm (note that -1 is a line 7 nonwitness, so the set of these nonwitnesses is not empty). Let x be such a nonwitness and let j be

the position of -1 in its sequence, where the positions are numbered starting at 0; $x^{s\cdot 2^j} \equiv -1 \pmod{n}$ and $x^{s\cdot 2^{j+1}} \equiv 1 \pmod{n}$. The line 7 nonwitnesses are a subset of $S_2 := \{a \in \mathbb{Z}_n^* | a^{s\cdot 2^j} \equiv \pm 1 \pmod{n}\}$, and S_2 is a subgroup of $\mathbb{Z}_n^*$.

By the CRT there exists $t \in \mathbb{Z}_n$ such that

$$\begin{array}{ll} t \equiv x \pmod{q} & \\ t \equiv 1 \pmod{r} & \end{array} \quad \Rightarrow \quad \begin{array}{l} t^{s\cdot 2^j} \equiv -1 \pmod{q}, \\ t^{s\cdot 2^j} \equiv 1 \pmod{r}. \end{array}$$

Hence t is a witness because $t^{s\cdot 2^j} \not\equiv \pm 1 \pmod{n}$ (see footnote[2]) but on the other hand $t^{s\cdot 2^{j+1}} \equiv 1 \pmod{n}$.

Therefore, just as in case 1, we have constructed a $t \in \mathbb{Z}_n^*$ which is not in S_2, and so S_2 can be at most half of $\mathbb{Z}_n^*$, and so at least half of the elements in $\mathbb{Z}_n$ are witnesses. □

EXERCISE 7.4. First show that the sets S_1 and S_2 (in the proof of theorem 7.3) are indeed a subgroups of $\mathbb{Z}_n^*$, and that in case 2 all nonwitnesses are contained in S_2. Then show that at least half of the elements of $\mathbb{Z}_n$ are witnesses when n is composite, without using group theory.

Note that by running the algorithm k times on independently chosen a, we can make sure that it rejects a composite with probability $\geq 1 - \frac{1}{2^k}$ (it will always accept a prime with probability 1). So, for $k = 100$ the probability of error, i.e., of a false positive, is negligible. Thus, we have a Monte Carlo algorithm for composites, and therefore PRIMES $= \{(n)_b | n \text{ is primes}\} \in$ co-RP; see §7.2 for a definition of co-RP.

7.1.3. Pattern matching. In this section we design a randomized algorithm for pattern matching. Consider the set of strings over $\{0,1\}$, and let $M : \{0,1\} \longrightarrow M_{2\times 2}(\mathbb{Z})$, that is, M is a map from strings to 2×2 matrices over the integers ($\mathbb{Z}$) defined as follows:

$$M(\varepsilon) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \quad M(0) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}; \quad M(1) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

and for strings $x, y \in \{0,1\}^*$, $M(xy) = M(x)M(y)$, where the operation on the LHS is concatenation of strings, and the operation on the RHS is multiplication of matrices.

First of all, $M(x)$ is well defined because matrix multiplication is associative, and second of all, $M(x) = M(y)$ implies that $x = y$ (i.e., the map M is 1-1). Given $M = M(x)$ we can "decode" x uniquely as follows: if the first column of M is greater than the second (where the comparison is made component-wise), then the last bit of x is zero, and otherwise it is 1. Let M' be M where we subtract the smaller column from the larger, and repeat.

For $x \in \{0,1\}^n$, the entries of $M(x)$ are bounded by Fibonacci number F_n. Let $F_0 = F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 1$. For a given string x, $M(x_1 x_2 \dots x_n)$ is such that the "smaller" column is bounded by F_{n-1} and the "larger" column is

[2]To see why $t^{s\cdot 2^j} \not\equiv \pm 1 \pmod{n}$ observe the following: suppose that $a \equiv -1 \pmod{q}$ and $a \equiv 1 \pmod{r}$, where $\gcd(q,r) = 1$. Suppose that $n = qr | (a+1)$, then $q|(a+1)$ and $r|(a+1)$, and since $r|(a-1)$ as well, it follows that $r|[(a+1)-(a-1)]$, so $r|2$, so $r = 2$, so n must be even, which is not possible since we deal with even n's in line 1 of the algorithm.

bounded by F_n. We can show this inductively: the basis case, $x = x_1$, is obvious. For the inductive step, assume it holds for $x \in \{0,1\}^n$, and show it still holds for $x \in \{0,1\}^{n+1}$: this is obvious as whether x_{n+1} is 0 or 1, one column is added to the other, and the other column remains unchanged.

By considering the matrices $M(x)$ modulo a suitable prime p, we perform efficient randomized pattern matching. We wish to determine whether x is a substring of y, where $|x| = n$, $|y| = m$, $n \leq m$. Let $y(i) = y_i y_{i+1} \dots y_{n+i-1}$, for appropriate i's. Select a prime $p \in \{1, \dots, nm^2\}$, and let $A = M(x) \pmod p$ and $A(i) \equiv M(y(i)) \pmod p$. Note that

$$A(i+1) \equiv M^{-1}(y_i) A(i) M(y_{n+i}) \pmod p,$$

which makes the computation of subsequent $A(i)$'s efficient.

So for all appropriate i's, we check whether $A = A(i)$. If yes, we check whether we did not get a false positive using a bit-by-bit comparison. If they match, we answer "yes," otherwise we change the prime p and continue.[3]

What is the probability of getting a false positive? It is the probability that $A(i) \equiv M(y(i)) \pmod p$ even though $A(i) \neq M(y(i))$. This is less than the probability that $p \in \{1, \dots, nm^2\}$ divides a (non-zero) entry in $A(i) - M(y(i))$. Since these entries are bounded by $F_n < 2^n$, less than n distinct primes can divide any of them. On the other hand, there are $\pi(nm^2) \approx (nm^2)/(\log nm^2)$ primes in $\{1, \dots, nm^2\}$. So the probability of a false positive is $O(1/m)$.

Note that this algorithm has no error; it is randomized, but all potential answers are checked for false positives. Checking for these potential candidates is called *fingerprinting*. The randomness lowers the average time complexity of the procedure.

7.2. Basic Randomized Classes

In this section we present four standard randomized classes: $\mathsf{RP}, \mathsf{ZPP}, \mathsf{BPP}$, and PP. We assume throughout that N is a nondeterministic polytime TM, where at each step there are exactly two nondeterministic choices (i.e., the degree of nondeterminism is always 2), and N is *precise*, i.e., all computations on x halt after the same number of steps, $p(|x|)$.

Let RP be the class of languages with polytime Monte Carlo TMs (see page 101). Note that $\mathsf{P} \subseteq \mathsf{RP} \subseteq \mathsf{NP}$.

RP is a *semantic class*, as opposed to a *syntactic class*. The intuition is that by "examining" a TM, in general we cannot tell if it is a Monte Carlo machine. On the other hand, syntactic classes are such that we can enumerate all the TMs which decide the class; examples of syntactic classes are P and NP, since we can list all polytime deterministic (or nondeterministic) TMs (see page 54 where we list all (oracle) polytime deterministic TMs). The problem with semantic classes is that usually no known complete problems exist for them. On the other hand, every syntactic class

[3]For more details on this question, and in particular for an interesting discussion of why we need to change the prime after a false positive, see [**KR87**].

$\mathcal{M}$ (e.g., nondeterministic polytime) has the following hard problem

$$\{\langle M, x\rangle | M \in \mathcal{M} \text{ and } M(x) = \text{“yes”}\},$$

which for P and NP also turns out to be complete. For semantic classes, this language is usually undecidable.

Is RP closed under complement? The class $\mathsf{co\text{-}RP}$ has false positives, but not false negatives. In §7.1.2 we showed that PRIMES $\in \mathsf{co\text{-}RP}$, although it is now known that PRIMES $\in \mathsf{P}$.

Consider $\mathsf{RP} \cap \mathsf{co\text{-}RP}$. Any problem in this class has two Monte Carlo algorithms, one with no false positives, and the other with no false negatives. If we execute both algorithms independently k times, the probability of not obtaining a definitive answer falls down to 2^{-k}, because it means that one of them is giving the wrong answer k times in the row (see figure 3). Let $\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{co\text{-}RP}$. The class ZPP is a randomized class with zero probability of error and an expected polynomial running time. A ZPP algorithm is called a *Las Vegas algorithm*. At the time of writing these notes, no algorithm class has been named after Baden-Baden.

	$x \in L$	$x \notin L$
RP	$\geq 1/2$ “yes”	all “no”
$\mathsf{co\text{-}RP}$	all “no”	$\geq 1/2$ “yes”

FIGURE 3. ZPP.

The most comprehensive but still plausible notion of realistic computation is the class BPP (Bounded Probability of error and Polynomial running time). This class consists of languages L for which there is a polytime TM N with the following properties: if $x \in L$, then at least $\frac{3}{4}$ of the computations accept, and if $x \notin L$, then at least $\frac{3}{4}$ of the computations reject. (In other words, acceptance and rejection by *clear majority*.)

We say that a language is in PP if there is a nondeterministic polytime TM N (precise—as always when considering randomized algorithms) such that $\forall x$, $x \in L$ iff more than half of the computations of N on input x end up accepting. We say that N decides L “by majority” (as opposed to “by clear majority” as in the case of BPP). PP is a syntactic class since every nondeterministic precise TM N can be used to define a language in PP.

Note that we have:

$$\mathsf{RP} \subseteq \mathsf{BPP} = \mathsf{co\text{-}BPP} \subseteq \mathsf{PP},$$

where the first inclusion follows from the fact that we can run an RP algorithm twice to get the probability of false negatives down to $\leq \frac{1}{4}$, and the second inclusion follows from the fact that a “clear majority” implies a “slim majority.”

A fundamental open problem is whether $\mathsf{BPP} =? \mathsf{P}$ or even $\mathsf{BPP} \subseteq ? \mathsf{NP}$.

EXERCISE 7.5. Show that the classes $\mathsf{RP}, \mathsf{BPP}$, and PP are closed under $\leq_{\mathsf{L}}^{m}$ (logspace many-one reductions).

EXERCISE 7.6. Show that BPP and RP are closed under union and intersection. Also show that PP is closed under complement and symmetric difference.

EXERCISE 7.7. Show that if $\mathsf{NP} \subseteq \mathsf{BPP}$ then $\mathsf{RP} = \mathsf{NP}$.

EXERCISE 7.8. Show that MAJSAT (consisting of ϕ's for which more than half of the truth assignments to $\mathrm{Var}(\phi)$ are satisfying) is PP-complete.

LEMMA 7.9. $\mathsf{NP} \subseteq \mathsf{PP}$.

PROOF. Suppose that L is in NP, decided by a precise N. Then the following N' decides L by majority: add a new initial state to N, and a nondeterministic choice out of it, one to the old initial state which leads to the old computation (i.e., the computation of the original N), and the other leads to a new precise computation of the same length as the old one, where all leaves are accepting.[4] □

A family of circuits $\{C_n\}$ is a *randomized circuit family for* f if in addition to the n inputs $x_1, x_2, \ldots, x_n$, it takes m (random) bits $r_1, r_2, \ldots, r_m$, and furthermore C_n satisfies two properties:

- If $f_n(x_1, \ldots, x_n) = 0$, then $C_n(x_1, \ldots, x_n, r_1, \ldots, r_m) = 0$ *regardless* of the value of the r_i's (i.e., no false positives).
- If $f_n(x_1, \ldots, x_n) = 1$, then $C_n(x_1, \ldots, x_n, r_1, \ldots, r_m) = 1$ with probability at least $1/2$.

This is the nonuniform version of a Monte Carlo algorithm, i.e., the nonuniform version of RP. Let RNC^i to be the class of languages recognizable by randomized NC^i circuit families (see page 102 where we claimed that perfect matching is in RNC^2).

In fact we can derandomize Monte Carlo circuit families (at the cost of losing uniformity—if our circuit family was uniform).

THEOREM 7.10. If f has a randomized polysize circuit family, then it has a polysize circuit family (i.e., it can be *derandomized*).

PROOF. For each n we form a matrix M with 2^n rows, corresponding to each input, and 2^m columns, corresponding to each random input. Let M_{jk} be 1 if the random input corresponding to the k-th column is a *witness* to the input corresponding to the j-th input (i.e., this choice of random bits sets the circuit on that input to be 1). Eliminate all rows for which f_n is zero. At least half of the entries in each surviving row are 1, so there must be a column where at least half of the entries are 1.

Pick the witness $\vec{\mathbf{r}}_1 = r_1, \ldots, r_m$ corresponding to this column; it is a witness for at least half of the inputs, so delete the rows corresponding to those inputs, and repeat.

At each stage, we find a witness to at least half of the remaining inputs, so after $\log 2^n = n$ many steps we will find witnesses $\vec{\mathbf{r}}_1, \ldots, \vec{\mathbf{r}}_n$ for all the inputs. Now let $C'_n(x_1, \ldots, x_n)$ be the `Or` of the circuits $C_n(x_1, \ldots, x_n, \vec{\mathbf{r}}_i)$, $i = 1, \ldots, n$. □

[4]When showing relationships between the different randomized classes one often has to modify the number of accepting and rejecting paths. When doing this, it is important to remember that the different nondeterministic branches are "not aware of each other." So the changes can not be made dependent of the contents of the individual branches.

7.3. The Chernoff Bound and Amplification

Suppose that L is in RP, and A is the Monte Carlo algorithm that decides L. We run A on x (independently) $p(|x|)$-many times. If the answer always come "no," we answer "no." The first time a "yes" answer comes, we answer "yes" and stop running A. If $x \notin L$, the answer will always be "no," and we answer correctly. If $x \in L$, then the probability that we answer "no" at the end is less than or equal to $\left(\frac{1}{2}\right)^{p(|x|)}$, i.e., exponentially small! This is called *amplification* or *error reduction*; at a polynomial cost, it increases the probability of getting the right answer exponentially: *"From fairest creatures we desire increase"*.[5]

LEMMA 7.11 (Chernoff Bound). Suppose that $X_1, X_2, \ldots, X_n$ are independent random variables, taking the value 1 and 0 with probability p and $(1-p)$, respectively, and consider $X = \sum_{i=1}^{n} X_i$. Then, for $0 < \theta \leq 1$,

$$\Pr[X \geq (1+\theta)pn] \leq e^{-\frac{\theta^2}{3}pn}.$$

PROOF.

$$\Pr[X \geq (1+\theta)pn] = \Pr[e^{tX} \geq e^{t(1+\theta)pn}] \leq e^{-t(1+\theta)pn} E(e^{tX})$$

where $t \in \mathbb{R}^+$, and where we applied a version of Markov's inequality (which states that $\Pr[X \geq kE(X)] \leq \frac{1}{k}$; see the proof of claim 5.28, on page 75), with $k = e^{t(1+\theta)pn} E(e^{tX})^{-1}$.

Since $X = \sum_{i=1}^{n} X_i$, where the X_i are independent random variables, we have that

$$E(e^{tX}) = (E(e^{tX_1}))^n = (1 \cdot (1-p) + e^t \cdot p)^n = (1 + p(e^t - 1))^n.$$

Therefore,

$$\Pr[X \geq (1+\theta)pn] \leq e^{-t(1+\theta)pn}(1 + p(e^t - 1))^n.$$

Now note that $(1+y) \leq e^y$, and so $(1 + p(e^t - 1)) \leq e^{p(e^t - 1)}$, and so we can conclude that

$$\Pr[X \geq (1+\theta)pn] \leq e^{-t(1+\theta)pn} e^{pn(e^t - 1)},$$

and $t = \ln(1+\theta)$ minimizes the RHS of the above equation, which finally gives us:

$$\Pr[X \geq (1+\theta)pn] \leq e^{pn(\theta - (1+\theta)\ln(1+\theta))}$$

and $(\theta - (1+\theta)\ln(1+\theta)) \leq \frac{-\theta^2}{3}$. □

COROLLARY 7.12. If $p = \frac{1}{2} + \varepsilon$ for some $0 < \varepsilon < \frac{1}{4}$, then $\Pr[X \leq \frac{n}{2}] \leq e^{-\frac{\varepsilon^2 n}{6}}$.

PROOF. $X = \sum_{i=1}^{n} X_i \leq \frac{n}{2}$ iff $-\sum_{i=1}^{n} X_i \geq -\frac{n}{2}$ iff $n - \sum_{i=1}^{n} X_i \geq n - \frac{n}{2}$ iff $\sum_{i=1}^{n} \underbrace{(1 - X_i)}_{X_i'} \geq \frac{n}{2}$. Thus, X_i' is 1 with probability $p' = \frac{1}{2} - \varepsilon$, and 0 with probability $(1-p')$, and so now we apply the Chernoff bound (lemma 7.11) and obtain

$$\Pr[X' \geq (1+\theta)(\frac{1}{2} - \varepsilon)n] \leq e^{-\frac{\theta^2}{3}(\frac{1}{2} - \varepsilon)n}.$$

[5] Shakespeare, Sonnet I.

To make $(1+\theta)(\frac{1}{2}-\varepsilon)=\frac{1}{2}$, we have to set $\theta=\frac{\varepsilon}{\frac{1}{2}-\varepsilon}$, and now

$$\Pr[X\leq\frac{n}{2}]=\Pr[X'\geq\frac{n}{2}]\leq e^{-\frac{\left(\frac{\varepsilon}{\frac{1}{2}-\varepsilon}\right)^2}{3}(\frac{1}{2}-\varepsilon)n}<e^{-\frac{\varepsilon^2 n}{6}},$$

where the right-most inequality follows from basic algebra. □

Here are some consequences of this corollary.

- We can detect a bias ε in a coin with reasonable certainty by performing about $n=O(\frac{1}{\varepsilon^2})$ many experiments. Say, we let $n=60\frac{1}{\varepsilon^2}$. Then $e^{-\frac{\varepsilon^2 n}{6}}<0.00004$, so the probability that the side with the bias $\frac{1}{2}+\varepsilon$ does not show up in the majority of trials is very small.
- PP is not a "good" randomized class because there the bias ε may be equal to $2^{-p(n)}$, where p is a polynomial (i.e., the majority may be very slim, so if $x\in L$, only half plus 1 are "yes" computations, and half minus 1 are "no" computations, and if $x\notin L$ vice-versa). If ε is so slim, the algorithm then has to be run exponentially many times to get the correct answer with any reasonable degree of confidence.
- We could define BPP to have $\frac{1}{2}+\varepsilon$ as the probability of getting the right answer, even for very small ε. Suppose $0<\varepsilon<\frac{1}{4}$. Suppose that N decides L by majority $\frac{1}{2}+\varepsilon$. Run N n-many times and accept as outcome the majority of outcomes. By choosing n suitably large we can bound the error by $\frac{1}{4}$: by corollary 7.12 we have

  $$\Pr[\text{error}]=\Pr[X\leq\frac{n}{2}]\leq e^{-\frac{\varepsilon^2 n}{6}}.$$

  We want $e^{-\frac{\varepsilon^2 n}{6}}<\frac{1}{4}$, so it is enough to have $\frac{\varepsilon^2 n}{6}>2$, i.e., $\varepsilon^2 n>12$, i.e., $n=\lceil\frac{12}{\varepsilon^2}\rceil$. Note that ε does not even have to be a constant—it could be any inverse polynomial. We can do even better; say we want $e^{-\frac{\varepsilon^2 n}{6}}<\frac{1}{4^m}$, then just let n be greater than $(8/\varepsilon^2)m$.

  In other words, we can always assume that if we have a BPP algorithm for deciding a language, we have a BPP algorithm for deciding the same language where the probability of error is bounded by $\frac{1}{2^{|x|}}$ for any input x. This negligible probability of error (2^{100} is the estimated number of atoms in the observable universe) is what motivates the conjecture that P = BPP.

  Also, in the absence of a proof that P = BPP, this small probability of error suggest that perhaps BPP can replace P as the class of languages that can be recognized feasibly.

7.4. More on BPP

The following theorems show that we can replace randomness with nonuniformity.

THEOREM 7.13. All languages in BPP have polysize circuits, i.e., BPP ⊆ P/poly.

PROOF. $L \in \mathsf{BPP}$, so it is decided by a nondeterministic N by clear majority, say $\frac{1}{4}$ probability of error. We show how to build $C = \langle C_n \rangle$ using the "probabilistic method in combinatorics."[6] Let $p(n)$ be the length of computation of N on inputs of length n (as always, assume N is precise). Consider a sequence $A_n = \{a_1, \ldots, a_m\}$ of bit strings, each $a_i \in \{0,1\}^{p(n)}$, and let $m = 12(n+1)$. Each a_i represents a particular branch of N on an input of length n.

Define $C_n(x)$ to be the majority of outcomes of $N_{a_1}(x), \ldots, N_{a_m}(x)$, where N_{a_i} is a polytime deterministic Turing machine obtained from N by taking the branch specified by a_i, so $N_{a_i}(x)$ can be simulated with a polysize circuit. We argue that for every n there exists an A_n so that C_n works correctly on all inputs x.

CLAIM 7.14. $\forall n > 0$, $\exists A_n$, $|A_n| = m = 12(n+1)$, such that for any given x of length n, less than half of the strings in A_n are *bad for* x (a_i is "bad for x" if $N_{a_i}(x) \neq L(x)$).

PROOF. Generate A_n randomly. We show that the probability that for each $x \in \{0,1\}^n$ more than half the strings in A_n are good is at least $\frac{1}{2}$. Thus, an A_n with at least half the strings being good exists.

For each $x \in \{0,1\}^n$, at most $\frac{1}{4}$ of the computations are bad. So the expected number of bad a_i's is $\frac{m}{4}$. We use the Chernoff bound (lemma 7.11) with the following parameters:

$$X = \sum_{i=1}^{m} X_i, \text{ where } X_i = \begin{cases} 1 & a_i \text{ is bad} \\ 0 & a_i \text{ is good} \end{cases}$$
$$\theta = 1$$
$$p = \frac{1}{4} = \text{probability of } a_i \text{ being bad}$$
$$m = |A_n| = 12(n+1),$$

to obtain

$$\Pr[X \geq (1+\theta)pm] \leq e^{\left(-\frac{\theta^2}{3}pm\right)},$$

and hence $\Pr[\underbrace{X \geq \frac{m}{2}}_{\left[\begin{smallmatrix}\text{prob. } m/2 \text{ or} \\ \text{more } a_i\text{'s are bad}\end{smallmatrix}\right]}] \leq e^{\left(-\frac{m}{12}\right)} < \frac{1}{2^{n+1}}$. Thus, the probability that for *some* $x \in \{0,1\}^n$ at least half of A_n are bad a_i's is at most $2^n \frac{1}{2^{n+1}} = \frac{1}{2}$. □

This ends the proof of theorem 7.13. □

Note that we do not know an efficient way of constructing this A_n, for every n, since otherwise we would have shown that $\mathsf{BPP} = \mathsf{P}$.

THEOREM 7.15 (Sipser). $\mathsf{BPP} \subseteq \Sigma_2^p$.

PROOF. Let $L \in \mathsf{BPP}$. By amplification we know that we can make the probability of error less than $\frac{1}{2^n}$, so let N be the amplified BPP machine deciding L. Let $A(x) \subseteq \{0,1\}^{p(n)}$ be the set of accepting computations for a given x (i.e., strings

[6] Our first example of this method was the lower bound for parity, §5.3.1 (page 71).

representing a sequence of choices that lead to an accepting configuration—remember that our machines are always precise and of degree of nondeterminism exactly 2). We know the following:

- If $x \in L$, then $|A(x)| \geq 2^{p(n)}(1 - \frac{1}{2^n})$, and
- if $x \notin L$, then $|A(x)| \leq 2^{p(n)}\frac{1}{2^n}$.

Let $U = \{0,1\}^{p(n)}$, i.e., U is the set of bit strings of length $p(n)$. For $a, b \in U$, define $a \oplus b$ to be the bit-wise `Xor` of a, b, i.e., the string $(a_1 \oplus b_1) \cdots (a_{p(n)} \oplus b_{p(n)})$. Note that $a \oplus b = c \iff c \oplus b = a$. Thus, for a given fixed b, $f_b(a) = a \oplus b$ is a one-to-one function, and f_b is a convolution, i.e., $f_b^2 = \mathrm{id}$. Further, if $r \in \{0,1\}^{p(n)}$ is a random string, then $f_b(r) = r \oplus a$ is also a random string (because f_b is just a permutation of U).

For any $t \in U$, let $A(x) \oplus t := \{a \oplus t | a \in A(x)\}$. Call this the *translation of* $A(x)$ *by* t, and note that $|A(x) \oplus t| = |A(x)|$.

Claim 7.16. If $x \in L$, we can find a small set of translations of $A(x)$ that covers all of U.

Proof. Suppose $x \in L$, and consider $t_1, \ldots, t_{p(n)} \in U$, obtained uniformly at random. Fix $b \in U$. These translations *cover* b if $\exists j \leq p(n)$ such that $b \in A(x) \oplus t_j$.

$$\Pr[b \notin A(x) \oplus t_j] = \Pr[b \oplus t_j \notin A(x)] \leq \frac{1}{2^n},$$

since $b \in A(x) \oplus t_j \iff b \oplus t_j \in A(x)$, and since $x \in L$. So the probability that b is not covered by any of the t_j's is less than $2^{-n \cdot p(n)}$, and so the probability that some b in U is not covered is at most $2^{p(n)}2^{-n \cdot p(n)}$, and so with overwhelming probability all of U is covered, and so there is a particular $T = \{t_1, \ldots, t_{p(n)}\}$ that covers U. □

On the other hand, if $x \notin L$, then $A(x)$ is an exponential fraction of U, so no polysize set T that covers U can exist. Therefore, there is a sequence T of $p(n)$ translations that cover U iff $x \in L$. Thus, L is the set of strings x such that: $\exists T \in \{0,1\}^{p(n)^2}$, such that $\forall b \in U$, there is a $j \leq p(n)$ such that $b \oplus t_j \in A(x)$. Since the last "there is" can be expressed as an `Or` of polynomially many things, and so can be made part of a polytime relation, we obtain a Σ_2^p predicate. □

Corollary 7.17. $\mathsf{BPP} \subseteq \Sigma_2^p \cap \Pi_2^p$.

Proof. Since $\mathsf{BPP} = \mathsf{co\text{-}BPP}$, the corollary follows directly from theorem 7.15. □

Corollary 7.18. If $\mathsf{P} = \mathsf{NP}$ then $\mathsf{P} = \mathsf{BPP}$.

Proof. If $\mathsf{P} = \mathsf{NP}$, then by theorem 4.18 the polytime hierarchy PH collapses to P, so by theorem 7.15, $\mathsf{BPP} \subseteq \Sigma_2^p \subseteq \mathsf{P}$. Since trivially $\mathsf{P} \subseteq \mathsf{BPP}$, the corollary follows. □

7.5. Toda's theorem

Toda's theorem states that $\mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}}$, which can actually be stated more tightly as follows: $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}[1]}$. ($\#\mathsf{P}[1]$ means that we access the $\#\mathsf{P}$ oracle only once; see page 30 for a definition of $\#\mathsf{P}$.) What this theorem says is that counting is more powerful than any constant number of alternations of quantifiers.

We introduce one more complexity class: $\oplus\mathsf{P}$, Parity P, the class of languages which are decidable by the parity of the number of accepting computations of a nondeterministic polytime TM. The class $\oplus\mathsf{P}$ contains, for example, the graph isomorphism problem.

The proof of Toda's result has two parts.

Part 1. $\forall k$, $\Sigma_k^p \subseteq \mathsf{BPP}^{\oplus\mathsf{P}}$; the consequence of this is that $\mathsf{PH} \subseteq \mathsf{BPP}^{\oplus\mathsf{P}}$.

Part 2. $\mathsf{PP}^{\oplus\mathsf{P}} \subseteq \mathsf{P}^{\#\mathsf{P}[1]}$; the consequence of this is that $\mathsf{BPP}^{\oplus\mathsf{P}} \subseteq \mathsf{P}^{\#\mathsf{P}[1]}$.

Putting the two parts together, we obtain that $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}[1]}$ (since $\mathsf{BPP} \subseteq \mathsf{PP}$). On the other hand, $\mathsf{P}^{\mathsf{PP}} = \mathsf{P}^{\#\mathsf{P}}$ (lemma 7.25), so the advertised result follows. We make some observations:

- Since the permanent (over 0-1 matrices computed over $\mathbb{Z}$) is complete for $\#\mathsf{P}$ (as is $\#\textsc{Sat}$), we could have restated Toda's theorem as claiming that PH fits in P with a single access to an oracle for computing the permanent.
- The permanent can be used to compute the number of perfect matchings in a bipartite graph, and it was already noted that the number of perfect matchings is $\#\mathsf{P}$ complete (see page 102), while the related decision problem (i.e., *is there* a perfect matching in a given graph?) is in P. This is not the case for the decision problem related to $\#\textsc{Sat}$, i.e., $\textsc{Sat}$, which is NP-complete.

To show Part 1, we use induction on k. The Basis Case is that $\Sigma_1^p = \mathsf{NP} \subseteq \mathsf{BPP}^{\oplus\mathsf{P}}$ (which is lemma 7.19). Here is the plan for the proof of the induction hypothesis:

$$\Sigma_{k+1}^p \overset{(1)}{=} \mathsf{NP}^{\Sigma_k^p} \overset{(2)}{\subseteq} \mathsf{NP}^{\mathsf{BPP}^{\oplus\mathsf{P}}} \overset{(3)}{\subseteq} \mathsf{BPP}^{\oplus\mathsf{P}^{\mathsf{BPP}^{\oplus\mathsf{P}}}} \overset{(4)}{\subseteq} \mathsf{BPP}^{\mathsf{BPP}^{\oplus\mathsf{P}^{\oplus\mathsf{P}}}} \overset{(5)}{\subseteq} \mathsf{BPP}^{\oplus\mathsf{P}}, \tag{25}$$

where (1) is by definition, (2) is by the induction hypothesis, (3) is the relativized version of the basis case, i.e., by lemma 7.19 ("Relativized Inclusion," $\mathsf{NP}^O \subseteq \mathsf{BPP}^{\oplus\mathsf{P}^O}$), (4) is by lemma 7.23 (i.e., "Swapping," $\oplus\mathsf{P}^{\mathsf{BPP}^O} \subseteq \mathsf{BPP}^{\oplus\mathsf{P}^O}$), and (5) is by lemma 7.21 (i.e., "Collapsing," $\mathsf{BPP}^{\mathsf{BPP}^O} = \mathsf{BPP}^O$ and $\oplus\mathsf{P}^{\oplus\mathsf{P}} = \oplus\mathsf{P}$).

Part 2 follows from the single lemma 7.24.

LEMMA 7.19. For any oracle O, $\mathsf{NP}^O \subseteq \mathsf{BPP}^{\oplus\mathsf{P}^O}$.

PROOF. In fact we prove something stronger: $\mathsf{NP} \subseteq \mathsf{RP}^{\oplus\mathsf{P}}$ (or even stronger than that: $\mathsf{NP} \subseteq \mathsf{RP}^{\mathsf{UP}}$).[7] Suppose that $L \in \mathsf{NP}$, so there exists a polytime language A such that $x \in L \iff \mathcal{F}(x) := \{y | \langle x, y\rangle \in A\}$ is non-empty. Consider the language

$$B = \{\langle x, W, j\rangle : |\{y \in \mathcal{F}(x) : W(y) = j\}| = 1\}. \tag{26}$$

[7] UP, **U**nambiguous **P**olytime, is the class of languages decidable by a nondeterministic polytime TM where there are either no accepting paths or a single accepting path. Note that $\mathsf{P} \neq \mathsf{UP}$ iff worst-case one way functions exist.

Note that $W(y) = \sum_{i,y(i)=1} W(i)$ is a weight function, which can be encoded efficiently as a string which gives values (in an appropriate polysize range) to all the elements in the set $\{1, 2, \ldots, |y|\}$. In what follows, we are going to apply the Isolation Lemma (see the Appendix, §8.4).

There exists a polynomial $q(n)$ such that

- If $x \in L$, $\Pr_{W,j}[\langle x, W, j\rangle \in B] \geq \frac{1}{q(|x|)}$, where W, j are bounded by an appropriate polynomial. To see this, note that with a probability greater or equal to $\frac{3}{4}$, a W is selected such that $|\{y \in \mathcal{F}(x) | W(y) = j\}| = 1$, *for some* j. Now a j is selected independently at random, among polynomially many values, so we have $\frac{1}{q(|x|)}$ instead.
- If $x \notin L$, then $\Pr = 0$.

Exercise 7.20. Give precise bounds in the above outline, so that the analysis of the probabilities works out (in particular, what is the size of the range of W?).

If we replace the "$= 1$" in (26) with "is an odd number," then $B \in \oplus\mathsf{P}$. Now on input x, do $q(|x|)$-many independent trials of selecting W and j, and for each such pair query $\langle x, W, j\rangle \in B$, and accept iff at least one such trial is successful (in the sense that $\langle x, W, j\rangle \in ?B$ returns a "yes"). If $x \notin L$, clearly $\Pr[\text{accept}] = 0$, and if $x \in L$, then

$$\Pr[\text{reject}] < \left(1 - \frac{1}{q(|x|)}\right)^{q(|x|)} \uparrow_{x\to\infty} \frac{1}{e} < \frac{1}{2}.$$

The same argument can be repeated with an oracle O to obtain the result as advertised in the statement of the lemma. □

Lemma 7.21. For any oracle O, $\mathsf{BPP}^{\mathsf{BPP}^O} = \mathsf{BPP}^O$, and $\oplus\mathsf{P}^{\oplus\mathsf{P}} = \oplus\mathsf{P}$.

Proof. $\mathsf{BPP}^{\mathsf{BPP}} = \mathsf{BPP}$ follows by amplification: if we make the error exponentially small, and then directly simulate the oracle queries in the computation, then the number of incorrect paths (i.e., the paths on which some of the polynomially many oracle queries gave the wrong answer) is a small fraction of all the paths.

Exercise 7.22. Formalize the above paragraph with appropriate parameters and show precise bounds.

To show $\oplus\mathsf{P}^{\oplus\mathsf{P}} = \oplus\mathsf{P}$, observe first that $\oplus\mathsf{P}$ is closed under complementation (just add one more choice from the initial state that lands immediately in an accepting state; this way the total number of accepting states increases by 1 and the parity flips).[8]

Now suppose that $L \in \oplus\mathsf{P}^{\oplus\mathsf{P}}$, so there is an oracle TM M^B such that $x \in L \iff$ the number of accepting paths of M^B on x is odd, where $B \in \oplus\mathsf{P}$. Let N_0 witness B and N_1 witness $\overline{B}$. Let M' be a nondeterministic TM which on input x, simulates M, except when M is about to make a query on y, M' guesses the answer $b \in \{0, 1\}$, and then simulates N_b on y. At the end, M' accepts on a path, if all the computations of N_0, N_1, and M are accepting on that path.

[8]Another way to see it is using the fact that $\#\mathsf{P}$ is closed under addition: if N_0 is a polytime nondeterministic machine, then let N_1 be such that $\#\text{accept}_{N_1} = \#\text{accept}_{N_0} + 1$.

Now, for each accepting branch that ignores the computations of N_0, N_1 (i.e., we only take into account the guess b and continue immediately simulating M; call those branches "trunks"), there was a sequence of queries $y_1, y_2, \ldots, y_n$ on that path, and hence it gets blown up by $\Pi_{i=1}^{n} \#\text{accept}_{N_i}(y_i)$. Note that this number is odd iff all the n guesses are correct. All the trunks that correspond to bad guesses contribute an even factor. □

LEMMA 7.23. For any oracle O, $\oplus\mathsf{P}^{\mathsf{BPP}^O} \subseteq \mathsf{BPP}^{\oplus\mathsf{P}^O}$.

PROOF. We show that $\oplus\mathsf{P}^{\mathsf{BPP}} \subseteq \mathsf{BPP}^{\oplus\mathsf{P}}$, and since the argument relativizes, the lemma follows. Suppose that $L \in \oplus\mathsf{P}^{\mathsf{BPP}}$. Then, there exists a language $A \in \mathsf{P}^{\mathsf{BPP}}$ such that

$$x \in L \iff |\{y : \langle x, y\rangle \in A\}| \text{ is odd.}$$

By lemma 7.21 we know that $\mathsf{P}^{\mathsf{BPP}} \subseteq \mathsf{BPP}^{\mathsf{BPP}} = \mathsf{BPP}$, so in fact $\mathsf{P}^{\mathsf{BPP}} = \mathsf{BPP}$, so we can assume directly that $A \in \mathsf{BPP}$. So we know that there exists a language $B \in \mathsf{P}$ such that

$$\langle x, y\rangle \in A \iff \langle x, y, z\rangle \in B,$$

and this is true for, say, a $(1 - 1/2^{p(|x|)})$-fraction of the z's, where p is any polynomial. We can make the polynomial p sufficiently large, so that in fact for a 3/4-fraction of the z's, we can make the following assertion:

$$\forall y[\langle x, y\rangle \in A \iff \langle x, y, z\rangle \in B].$$

We can make this assertion since the number of y's is itself bounded by $2^{q(|x|)}$, where q is a polynomial. It therefore follows that, for a given x, for a fraction of 3/4 of the z's, we have:

$$|\{y : \langle x, y\rangle \in A\}| = |\{y : \langle x, y, z\rangle \in B\}|$$

and in particular, the LHS is odd iff the RHS is odd. Thus, the following $\mathsf{BPP}^{\oplus\mathsf{P}}$ algorithm decides L: on input x_0, generate a random z_0, and then query the $\oplus\mathsf{P}$ oracle if the set $\{y|\langle x_0, y, z_0\rangle \in B\}$ is odd-sized. On 3/4 of the z's we get the right answer. □

LEMMA 7.24. $\mathsf{PP}^{\oplus\mathsf{P}} \subseteq \mathsf{P}^{\#\mathsf{P}[1]}$.

PROOF. Suppose that $L \in \mathsf{PP}^{\oplus\mathsf{P}}$. Then, there exists a language $A \in \mathsf{P}^{\oplus\mathsf{P}}$, and hence, by lemma 7.21 (part two), $A \in \oplus\mathsf{P}$, such that:

$$x \in L \iff |\{y : \langle x, y\rangle \in A\}| > 2^{p(|x|)-1}.$$

Let M be the polytime nondeterministic TM witnessing that A is in $\oplus\mathsf{P}$. Using some clever arithmetic, it is possible to design a $\#\mathsf{P}$ function g, such that

$$g(\langle x, y\rangle) = \begin{cases} m2^{p(|x|)} + 1 & \text{if } \#\text{accept}_M(\langle x, y\rangle) \text{ is odd} \\ m2^{p(|x|)} & \text{if } \#\text{accept}_M(\langle x, y\rangle) \text{ is even,} \end{cases}$$

and from g we can easily design the $\#\mathsf{P}$ function h,

$$h(x) = \sum_y g(\langle x, y\rangle).$$

The #P machine computing h simply guesses a y (in the appropriate range, of course), and then simulates $g(\langle x, y\rangle)$.

The question is how to define g.

Let $s_0(z) = z$, and let $s_{i+1}(z) = 3s_i(z)^4 + 4s_i(z)^3$. The function s_i has the property that if z is odd, then $s_i(z) - 1$ is divisible by 2^{2^i}, and if z is even, it is divisible by $m2^{2^i}$. On the other hand, we can evaluate a polynomial at a value in #P as follows: suppose we want to compute $a_0 + a_1 z + a_2 z^2 + \cdots + a_m z^m$. For each i, we branch out on all the values $1, 2, \ldots, a_i$, and then we branch out on the value z i-many times.

Let $l_x = \lceil \log p(|x|) + 1 \rceil$ and $r_x(z) = (s_{l_x}(z))^2$, and $g(x) = r_x(\#\text{accept}_M(x))$.

We now show how to put it all together to decide L in $\mathsf{P}^{\#\mathsf{P}[1]}$. On input x, we use the #P oracle exactly once to compute $h(x)$. Once we have $h(x)$ on the oracle tape, we check whether the value encoded in the first $p(|x|)$ many bits is greater than $2^{p(|x|)-1}$, and accept iff it is. □

LEMMA 7.25. $\mathsf{P}^{\mathsf{PP}} = \mathsf{P}^{\#\mathsf{P}}$.

PROOF. Showing $\mathsf{P}^{\mathsf{PP}} \subseteq \mathsf{P}^{\#\mathsf{P}}$ is easy, since a #P oracle gives more information than a PP oracle (we not only know if the majority of computations are accepting; we actually know how many).

To show that $\mathsf{P}^{\mathsf{PP}} \supseteq \mathsf{P}^{\#\mathsf{P}}$, we show that for any polytime nondeterministic TM N, the language $L = \{\langle x, y\rangle | \#\text{accept}_N(x) \geq y\}$ is in PP. This way, if we have a #P function f as oracle, we can compute its value in polytime with a PP oracle using binary search.

We can assume that N is precise, and all computational paths are of length exactly $p(|x|)$. Now define the machine D, which takes input $\langle x, y\rangle$, and which initially branches on $b = 0$ and $b = 1$.

If $b = 0$, it guesses a string $z \in \{0,1\}^{p(|x|)}$, and then accepts iff the rank of z is at most $2^{p(|x|)} - y$. (The *rank* of a string is the number of strings, in lexicographic order by lengths, after that string. In this case it means that there are more than y many strings before z—of course, we work with the finite set $\{0,1\}^{p(|x|)}$.)

If $b = 1$, then D just simulates N on x.

The proportion of accepting computations is:

$$\frac{2^{p(|x|)} - y}{2^{p(|x|)+1}} + \frac{\#\text{accept}_N(x)}{2^{p(|x|)+1}},$$

and it is $> \frac{1}{2}$ iff $\#\text{accept}_N(x) \geq y$. □

7.6. Answers to selected exercises

Exercise 7.4. Showing that S_1, S_2 are subgroups of $\mathbb{Z}_n^*$ is easy; it is obvious in both cases that 1 is there, and closure and existence of inverse can be readily checked.

To give the same proof without group theory, we follow the cases in the proof of theorem 7.3. Let t be the witness constructed in case 1. If d is a (stage 3) nonwitness, we have $d^{p-1} \equiv 1 \pmod{p}$, but then $dt \pmod{p}$ is a witness. Moreover, if d_1, d_2 are

distinct (stage 3) nonwitnesses, then $d_1 t \not\equiv d_2 t \pmod{p}$. Otherwise,

$$d_1 \equiv_p d_1 \cdot t \cdot t^{p-1} \equiv_p d_2 \cdot t \cdot t^{p-1} \equiv_p d_2.$$

Thus the number of (stage 3) witnesses must be at least as large as the number of nonwitnesses.

We do the same for case 2; let d be a nonwitness. First, $d^{s \cdot 2^j} \equiv \pm 1 \pmod{p}$ and $d^{s \cdot 2^{j+1}} \equiv 1 \pmod{p}$ owing to the way that j was chosen. Therefore $dt \pmod{p}$ is a witness because $(dt)^{s \cdot 2^j} \not\equiv \pm 1 \pmod{p}$ and $(dt)^{s \cdot 2^{j+1}} \equiv 1 \pmod{p}$.

Second, if d_1 and d_2 are distinct nonwitnesses, $d_1 t \not\equiv d_2 t \pmod{p}$. The reason is that $t^{s \cdot 2^{j+1}} \equiv 1 \pmod{p}$. Hence $t \cdot t^{s \cdot 2^{j+1} - 1} \equiv 1 \pmod{p}$. Therefore, if $d_1 t \equiv d_2 t \pmod{p}$, then

$$d_1 \equiv_p d_1 t \cdot t^{s \cdot 2^{j+1} - 1} \equiv_p d_2 t \cdot t^{s \cdot 2^{j+1} - 1} \equiv_p d_2.$$

Thus in case 2, as well, the number of witnesses must be at least as large as the number of nonwitnesses.

Exercise 7.6. To prove that PP is closed under complementation, we need to show that it is possible to modify a nondeterministic TM in such a way, that the number of accepting and rejecting computation paths will always differ (of course preserving the inequality between them). To achieve this we first make sure that the left-most computation path is always an accepting one (this can be done preserving the accept/reject relation by splitting the computation tree into 4: an always-accepting, an always-rejecting and two identical to the original one). Then we add an additional bit to the state of the machine to keep track whether the computation follows the left-most path. If this is the case, we nondeterministically accept or reject. Otherwise we do as before (of course we need to create two identical branches to make the machine precise). Now if the original machine would accept on exactly half of the branches, the new one will accept on one less than a half. Therefore we can get a machine for the complement of the original language by reversing the accept and reject answers.

Knowing that PP is closed under complement it is easy to show that it is also closed under symmetric difference. If $\frac{1}{2} < p, q \leq 1$ then it is either the case that $p(1-q) + q(1-p)$ is greater than $\frac{1}{2}$ or $pq + (1-p)(1-q)$ is greater than $\frac{1}{2}$. Therefore for any two languages $L_1, L_2 \in \mathsf{PP}$ we can use one of the following to get their symmetric difference:

$$L_1 \triangle L_2 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2), \qquad L_1 \triangle L_2 = \overline{(L_1 \cap L_2) \cup (\overline{L_1} \cap \overline{L_2})}.$$

Exercise 7.7. We know that $\mathsf{RP} \subseteq \mathsf{NP}$. Assume that $\mathsf{NP} \subseteq \mathsf{BPP}$. Then $\textsc{Sat} \in \mathsf{BPP}$. Consider the following algorithm for SAT: given a formula ϕ we first use the BPP algorithm to see if it is satisfiable. If the answer comes "no," we reject. If the answer comes "yes," we set the first variable x_1 to 0 and 1 and ask the BPP machine again. If both answers are "no," we reject. Otherwise we proceed with x_1 set to the value for which the answer was "yes." We repeat this procedure until we get a total truth assignment. Then we verify (deterministically!) that it really satisfies ϕ and accept if and only if it does. From the last step it is clear, that our algorithm will not return false positive answers. The probability of a false negative can be bounded from above

by the sum of probabilities of errors on each steps (and there are n of them). And as the probability of error of an BPP algorithm can be made exponentially small by amplification, we can easily limit the resulting probability of false negatives by $\frac{1}{2}$. Thus SAT $\in$ RP, but as SAT is NP-complete, NP $\subseteq$ RP and finally, NP = RP.

Exercise 7.8. Consider any language $L \in$ PP. Having the machine solving it we can create a Boolean formula encoding possible computations (as in the Cook's theorem 2.13). Let us consider two parts of this formula: ϕ, meaning "the computation is correct (well-formed)," and ψ, meaning "the computation is accepting." Let us introduce a new variable x and consider the following formula: $\theta := (\phi \to \psi) \wedge (\neg\phi \to x)$. It is easy to see, that if a truth assignment does not satisfy ϕ, then to satisfy θ it must satisfy x. It means that exactly half of these assignments satisfy θ. On the other hand if a truth assignment satisfies ϕ then, to satisfy θ, it must satisfy ψ. It means that more than half of all assignments satisfy θ if and only if more than a half correct computations are accepting. And, as the size of θ is of course polynomial in the size of the instance of L, the reduction works correctly. Thus any problem in PP can be reduced do MAJSAT which means that MAJSAT is PP-complete.

7.7. Notes

The material in §7.1.2 is based on [**Sip06**, § 10.2]. The proof of lemma 8.21 is from [**Sip06**, exercise 10.16]. §8.3 is based on [**MR95**, § 14.4, pg. 410]. §7.1.3 is based on [**KR87**]. Exercise 7.5 is [**Pap94**, exercise 11.5.13], exercise 7.6 is [**Pap94**, exercise 11.5.14 and 15], exercise 7.7 is [**Pap94**, exercise 11.5.18], exercise 7.8 is [**Pap94**, exercise 11.5.16]. The proof of lemma 7.11 is based on [**Pap94**, lemma 11.9], and corollary 7.12 is based on [**Pap94**] as well, but note that in [**Pap94**] the condition $\varepsilon < \frac{1}{4}$ is omitted but it is necessary to ensure that $0 < \theta \leq 1$. §8.4 is based on [**HO02**, chapter 4].

Credit for inventing the Monte Carlo method often goes to Stanisław Ulam, a Polish born mathematician who worked for John von Neumann on the United States Manhattan Project during World War II. Ulam is primarily known for designing the hydrogen bomb with Edward Teller in 1951. He invented the Monte Carlo method in 1946 while pondering the probabilities of winning a card game of solitaire. Quoted in [**Eck87**], Ulam describes the incident as follows: *The first thoughts and attempts I made to practice [the Monte Carlo Method] were suggested by a question which occurred to me in 1946 as I was convalescing from an illness and playing solitaires. The question was what are the chances that a Canfield solitaire laid out with 52 cards will come out successfully? After spending a lot of time trying to estimate them by pure combinatorial calculations, I wondered whether a more practical method than abstract thinking might not be to lay it out say one hundred times and simply observe and count the number of successful plays. This was already possible to envisage with the beginning of the new era of fast computers, and I immediately thought of problems of neutron diffusion and other questions of mathematical physics, and more generally how to change processes described by certain differential equations into an equivalent*

form interpretable as a succession of random operations. Later [in 1946, I] described the idea to John von Neumann, and we began to plan actual calculations.

8

Appendix

8.1. NP-complete problems

If $G = (V, E)$ is an (undirected) graph and $V_0 \subseteq V$, then V_0 is an *independent set* if no two nodes in V_0 are connected by an edge in E.
IndepSet:
Input: $G = (V, E)$, $B \in \mathbb{N}$, $B \leq |V|$.
Question: Does G have an independent set of size B?

Lemma 8.1. IndepSet is NP-complete.

Proof. IndepSet $\in$ NP: guess a set of vertices $V_0 \subseteq V$, check that $|V_0| = B$ and that no edge in E connects two vertices in V_0. We show that 3Sat $\leq_{\mathrm{L}}^{m}$ IndepSet.

Given an input $I = \{C_1, \ldots, C_m\}$ to 3Sat, we must find an input $\langle G_I, B_I \rangle$ to IndepSet such that $I \in$ 3Sat iff $\langle G_I, B_I \rangle \in$ IndepSet.

For each instance $I = \{C_1, \ldots, C_m\}$ of 3Sat, where $C_i = l_{i1} \vee l_{i2} \vee l_{i3}$, $1 \leq i \leq m$, define an instance $G_I = (V_I, E_I)$, B_I of IndepSet as follows: the set V_I of vertices is given by:

$$V_I = \{v_{ij} |\ 1 \leq i \leq m,\ 1 \leq j \leq 3\ \}$$

To construct E_I, connect any two nodes in the same clause, and connect complementary literals (e.g. x and $\bar{x}$). More precisely, $(v_{ij}, v_{kl}) \in E$ iff one of the following holds: $i = k$, that is l_{ij} and l_{kl} are literals in the same clause, or, l_{ij} and l_{kl} are complementary literals (one is the negation of the other).

Finally, let $B_I = m$.

($\Rightarrow$) If I has a satisfying truth assignment t, then G_I has an independent set of size m. To see this, select one literal from each clause made true by t. Two complementary literals cannot both be made true by t, so the result is an independent set.

($\Leftarrow$) Conversely, if G_I has an independent set V_0 of size m, then I has a satisfying truth assignment t. To see this, observe that V_0 selects one occurrence of a literal from each clause. Define t to make all of these literals true. Note that for each l, we cannot select both l and $\bar{l}$ in different clauses because they are connected by an edge. □

If $G = (V, E)$ and $V_0 \subseteq V$, then V_0 is a *vertex cover* in G if every edge in E has at least one end point in V_0.
VertexCover:
Input: $G = (V, E)$, $B \in \mathbb{N}$, $B \leq |V|$.
Question: Does G have a vertex cover of size B?

LEMMA 8.2. VERTEXCOVER is NP-complete.

PROOF. We show that INDEPSET $\leq$ VERTEXCOVER. Observe that if $G = (V, E)$ is a graph, $V_0 \subseteq V$, then V_0 is a vertex cover for G iff $V - V_0$ is an independent set.

So we define the reduction from INDEPSET to VERTEXCOVER as follows: given $I = \langle G, B\rangle$, an input to INDEPSET, let the corresponding input to VERTEXCOVER be $I' = \langle G, B'\rangle$, where $B' = n - B$, $n = |V|$. □

If $G = (V, E)$ and $V_0 \subseteq V$, then V_0 is a *clique* in G iff every pair of distinct nodes in V_0 is connected by an edge in E.

CLIQUE:
Input: $G = (V, E)$, $B \in \mathbb{N}$, $B \leq |V|$.
Question: Does G have a clique of size B?

LEMMA 8.3. CLIQUE is NP-complete.

PROOF. We show that INDEPSET $\leq$ CLIQUE.

Given an instance $I = \langle G, B\rangle$ of INDEPSET, we have to find an instance $I' = \langle G_I, B_I\rangle$ of CLIQUE such that $I \in$ INDEPSET iff $I' \in$ CLIQUE. Let $G_I = \overline{G}$, the complement of G. Let $B_I = B$.

If $G = (V, E)$, then $\overline{G} = (V, \overline{E})$, where $(u, v) \in \overline{E}$ iff $(u, v) \notin E$ (i.e. $\overline{E} = (V \times V) - E$).

Then G has an independent set of size B iff $\overline{G}$ has a clique of size B. □

SUBSETSUM:
Input: $\langle a_1, \ldots, a_m, t\rangle$ $t, a_i \in \mathbb{N}$.
Question: Is there a subset $S \subseteq \{1, \ldots, m\}$ such that $\sum_{i\in S} a_i = t$?

LEMMA 8.4. SUBSETSUM is NP-complete.

EXERCISE 8.5. Prove lemma 8.4

PARTITION:
Input: $\langle a_1, \ldots, a_m\rangle$ $a_i \in \mathbb{N}$.
Question: Is there a subset $S \subseteq \{1, \ldots, m\}$ such that $\sum_{i\in S} a_i = \sum_{j\in\overline{S}} a_j$?

LEMMA 8.6. PARTITION is NP-complete.

PROOF. We show that SUBSETSUM $\leq$ PARTITION we consider two cases: $2t \geq a$ and $2t < a$, where $a = a_1 + \cdots + a_m$. In the first case, given the input instance $\langle a_1, \ldots, a_m, t\rangle$ to SUBSETSUM, let $\langle a_1, \ldots, a_m, a_{m+1}\rangle$ where $a_{m+1} = 2t - a$ be the corresponding input instance to PARTITION. For the second case let $a_{m+1} = a - 2t$. □

A *k-coloring* of a graph $G = (V, E)$ assigns one of k colors to each vertex in G so that adjacent vertices get different colors.

k-COLOR:
Input: $G = (V, E)$.
Question: Can G be colored with k colors?

LEMMA 8.7. 3COLOR is NP-complete.

EXERCISE 8.8. Prove lemma 8.7.

A *Hamiltonian cycle* in a graph G is a cycle which includes every node of G exactly once.
HamCycle:
Input: $G = (V, E)$.
Question: Does G have a Hamiltonian cycle?

Lemma 8.9. HamCycle is NP-complete.

Proof. The reduction VertexCover $\leq$ HamCycle is tricky; see for example [**CLRS09**, §34.5.3, pp. 1008–1012]. □

A *tour* is the same as a Hamiltonian cycle. The input graph G must be *complete*; that is every pair of nodes must be connected by an edge. In this case, a Hamiltonian cycle always exists (if $|V| \geq 3$). The search problem is to find a tour of minimum total cost.
TravelSalesman:
Input: $G = (V, E)$, the complete graph with node set V, with positive integer weights (costs) assigned to each edge, $B \in \mathbb{N}$.
Question: Does G have a tour of total cost at most B?

Lemma 8.10. TravelSalesman is NP-complete.

Proof. We use a reduction from HamCycle as follows. Given an input $I = \langle G = (V, E)\rangle$ to HamCycle, find an input $I' = \langle G_I, C_I, B_I\rangle$ to TravelSalesman (C_I is the cost function) such that G has a Hamiltonian cycle iff G_I has a tour of total cost at most B_I. Let G_I be the complete graph on V, $C_I(e) = 1$ for each edge $e \in E$, $C_I(e) = 2$ for each edge $e \notin E$, $B_I = n$, where n is the number of nodes in G. □

A *Hamiltonian path* of a graph G is a path which includes every node of G exactly once.
HamPath:
Input: $G = (V, E)$.
Question: Does G have a Hamiltonian path?

Lemma 8.11. HamPath is NP-complete.

Proof. We use a reduction from HamCycle as follows. Given an input $I = \langle G\rangle$ to HamCycle, find an input $I' = \langle G_I\rangle$ to HamPath such that G has a Hamiltonian cycle iff G_I has a Hamiltonian path.

Take any vertex $v \in V$ and split it in two, v and v', and add two new nodes a and b.

Let $V_I = V \cup \{v', a, b\}$, $E_I = E \cup \{(a, v), (b, v')\} \cup \{(v', u) | (v, u) \in E\}$.

Then the original graph has a Hamiltonian cycle iff the new graph has a Hamiltonian path. Note that every Hamiltonian path in the new graph must connect a to b (or b to a). □

Let α be a 3CNF formula. A *nae-assignment* (where nae stands for "not all equal") to the variables of α is an assignment where each clause contains two literals with unequal truth values (i.e., one true and one false). Note that the negation of any

nae-assignment is also a nae-assignment ("negation" here means that true and false are interchanged for all variables). Let NAE3SAT be the language of 3CNF formulas with nae-assignments.

LEMMA 8.12. NAE3SAT is NP-complete.

PROOF. $c_i = (l_1 \vee l_2 \vee l_3) \overset{f}{\mapsto} c'_i = (l_1 \vee l_2 \vee z_i) \wedge (\bar{z}_i \vee l_3 \vee b)$ where z_i is a new variable for each clause c_i, and b is a single new variable for all the clauses. The result of the (logspace) mapping f is the conjunction of two clauses c'_i. Correctness: suppose t satisfies $\bigwedge_i c_i$. Let t' be defined as follows $t'(b) = F$, and if $t(l_1 \vee l_2) = T$, then $t'(\bar{z}_i) = T$, and if $t(l_1 \vee l_2) = F$, and $t(l_3) = T$, then $t'(z_i) = T$. Clearly, t' is a nae-assignment. Now we do a similar argument for the other direction. □

The problem DEG2POLY is given by m polynomials over $\mathbb{Z}$ of degree 2 in n variables, and the question is: do they have a common zero?

LEMMA 8.13. DEG2POLY is NP-complete.

PROOF. 3SAT is NP-complete, and given a 3CNF formula we can convert it into a system of polynomials as follows: for each variable x_i that appears, add the polynomial $x_i^2 - x_i$, to ensure that only values $\{0, 1\}$ are taken. Then, for each clause $(l_1 \vee l_2 \vee l_3)$ add the polynomial $m(l_1) \cdot m(l_2) \cdot m(l_3)$, where $m(l) = (1 - x)$ if $l = x$ and $m(l) = x$ if $l = \bar{x}$. The resulting system has a common zero iff the original 3CNF formula is satisfiable. However, the problem is that the polynomials may be of degree 3. We modify this idea of "arithmetization" slightly, and use NAE3SAT instead, with true mapping to 1 and false to -1. □

EXERCISE 8.14. Finish the proof of lemma 8.13.

8.2. A little number theory

We introduce some terminology. Two numbers x, y are *congruent modulo* a third number p if they differ by a multiple of p. We write $x \equiv y \pmod{p}$ (and sometimes $x \equiv_p y$). Every number is congruent modulo p to some number in $\mathbb{Z}_p = \{0, 1, \ldots, (p-1)\}$. We let $\mathbb{Z}_p^*$ be the subset of $\mathbb{Z}_p$ of elements a such that $\gcd(a, p) = 1$. Note that $(\mathbb{Z}_p, +)$ is a group (under addition) and $(\mathbb{Z}_p^*, \cdot)$ is a group (under multiplication).

Note that

$$\begin{aligned} \mathbb{Z}_p^* &= \{a \in \mathbb{Z}_p | \gcd(a, p) = 1\} \\ &= \{a \in \mathbb{Z}_p | a \text{ has a (multiplicative) inverse in } \mathbb{Z}_p\}. \end{aligned} \tag{27}$$

This is why $(\mathbb{Z}_p^*, \cdot)$ is a group; the following lemma justifies this observation.

LEMMA 8.15. $\gcd(a, p) = 1 \iff \exists b \in \mathbb{Z}_p$ such that $ab \equiv 1 \pmod{p}$.

PROOF. $\gcd(a, p) = 1$ iff there exist x, y such that $ax + py = 1$ (and x can be chosen to be in $\mathbb{Z}_p$, by adding or subtracting appropriate multiples of p, and adjusting y by the same amount), and this is the case iff $ax \equiv 1 \pmod{p}$. □

The function $\phi(n)$ is called the *Euler totient function*, and it is the number of elements less than n that are co-prime to n, i.e., $\phi(n) = |\mathbb{Z}_n^*|$. If we are able to factor, we are also able to compute $\phi(n)$: suppose that $n = p_1^{k_1} p_2^{k_2} \cdots p_l^{k_l}$, then it is not hard to see that $\phi(n) = \prod_{i=1}^{l} p_i^{k_i - 1}(p_i - 1)$.

THEOREM 8.16 (Euler's theorem). For every n and every $a \in \mathbb{Z}_n^*$, we have $a^{\phi(n)} \equiv 1 \pmod{n}$.

PROOF. This is an immediate consequence of Lagrange's theorem (which says that the order of any subgroup, and hence the order of any element, divides the order of the group). □

THEOREM 8.17 (Fermat's Little theorem). For every prime p and every $a \in \mathbb{Z}_p$, we have $a^{(p-1)} \equiv 1 \pmod{p}$.

PROOF. An immediate consequence of Euler's theorem. Note that when p is a prime, $\mathbb{Z}_p - \{0\} = \mathbb{Z}_p^*$, and $\phi(p) = (p - 1)$. □

One way to determine whether a number p is prime, is to try all possible numbers $n < p$, and check if any are divisors (also called factors). This algorithm obviously has exponential time complexity in the length of p. We have a good probabilistic algorithm for primality testing (Rabin-Miller), but no probabilistic algorithm for factoring is known.

LEMMA 8.18. A number $p > 1$ is prime iff $\exists 1 < r < p$ such that $r^{p-1} \equiv 1 \pmod{p}$, and furthermore $r^{\frac{p-1}{q}} \not\equiv 1 \pmod{p}$ for all prime divisors q of $p - 1$.

THEOREM 8.19 (Pratt). PRIMES $\in$ NP.

PROOF. Using lemma 8.18 we can construct a short (recursive) *certificate of primality*: $C(p) := (r; q_1, C(q_1), \ldots, q_k, C(q_k))$. □

COROLLARY 8.20. PRIMES $\in$ NP $\cap$ co-NP.

PROOF. It is obvious that PRIMES $\in$ co-NP and Pratt's theorem shows that PRIMES $\in$ NP as well. □

Fermat's little theorem provides a "test" for primality, called the Fermat test; the Rabin-Miller algorithm (algorithm 7.2) is based on this test. When we say that p passes the Fermat test at a, we mean that $a^{(p-1)} \equiv 1 \pmod{p}$. Thus, all primes pass the Fermat test for all $a \in \mathbb{Z}_p - \{0\}$.

Unfortunately, there are also composite numbers n that pass the Fermat tests for every $a \in \mathbb{Z}_n^*$; these are the so called *Carmichael numbers*,[1] for example, 561,

[1] R. D. Carmichael first noted the existence of such numbers in 1910, computed 15 examples, and conjectured that though they are infrequent there were infinitely many. In 1956, Erdös sketched a technique for constructing large Carmichael numbers ([**Hof98**]), and a proof was given by [**AGP94**] in 1994.

1105, 1729, ... and the last number shown on this list is called the Hardy-Ramanujan number.[2]

LEMMA 8.21. If p is a composite non-Carmichael number, then it passes at most half of the Fermat tests in $\mathbb{Z}_p^*$.

PROOF. Call a a *witness* if it fails the Fermat test for p, that is, if $a^{(p-1)} \not\equiv 1 \pmod{p}$.

Consider $S \subseteq \mathbb{Z}_p^*$ consisting of those elements $a \in \mathbb{Z}_p^*$ for which $a^{p-1} \equiv 1 \pmod{p}$. It is easy to check that S is in fact a *subgroup* of $\mathbb{Z}_p^*$. Therefore, using the Lagrange theorem, $|S|$ must divide $|\mathbb{Z}_p^*|$. Suppose now that there exists an element $a \in \mathbb{Z}_p^*$ for which $a^{p-1} \not\equiv 1 \pmod{p}$. Then, S is not "everything" (i.e., not $\mathbb{Z}_p^*$), so the next best thing it can be is "half" (of $\mathbb{Z}_p^*$), so $|S|$ must be at most half of $|\mathbb{Z}_p^*|$. □

EXERCISE 8.22. Give an alternative proof of lemma 8.21 without using group theory.

A number is *pseudoprime* if it is either prime or Carmichael. The last lemma suggests an algorithm for pseudoprimes: on input p, check $a^{(p-1)} \equiv 1 \pmod{p}$ for some random $a \in \mathbb{Z}_p - \{0\}$. If p fails this test (i.e., $\neq 1$), then p is composite for sure. If p passes the test, then p is probably pseudoprime. We show that the probability of error in this case is $\leq \frac{1}{2}$. Suppose p is not pseudoprime. If $\gcd(a,p) \neq 1$, then $a^{(p-1)} \not\equiv 1 \pmod{p}$[3], so assume that p passed the test, we know that $\gcd(a,p) = 1$, so $a \in \mathbb{Z}_p^*$. But then at least half of the elements of $\mathbb{Z}_p^*$ are witnesses of nonpseudoprimeness (what a great word!).

THEOREM 8.23 (Chinese Remainder). Given two sets of numbers of equal size, $r_0, r_1, \ldots, r_n$, and $m_0, m_1, \ldots, m_n$, such that

$$0 \leq r_i < m_i \qquad 0 \leq i \leq n, \tag{28}$$

and $\gcd(m_i, m_j) = 1$ for $i \neq j$, then there exists an r such that $r \equiv r_i \pmod{m_i}$ for $0 \leq i \leq n$.

PROOF. The proof we give is by counting; the distinct values of r, $0 \leq r < \Pi m_i$, represent distinct sequences. To see that, note that if $r \equiv r' \pmod{m_i}$ for all i, then $m_i | (r - r')$ for all i, and so $(\Pi m_i) | (r - r')$ (since the m_i's are pairwise co-prime). So $r \equiv r' \pmod{(\Pi m_i)}$, and so $r = r'$ if both $r, r' \in \{0, 1, \ldots, (\Pi m_i) - 1\}$.

---

[2]1729 is known as the Hardy-Ramanujan number after a famous anecdote of the British mathematician G. H. Hardy regarding a hospital visit to the Indian mathematician Srinivasa Ramanujan. Hardy wrote: *I remember once going to see him when he was ill at Putney. I had ridden in taxi cab number 1729 and remarked that the number seemed to me rather a dull one, and that I hoped it was not an unfavorable omen. "No," he replied, "it is a very interesting number; it is the smallest number expressible as the sum of two cubes in two different ways."*

[3]To see why this is true, assume that $\gcd(a,p) \neq 1$. By the observation (27) we know that if $\gcd(a,p) \neq 1$, then a does not have an inverse in $\mathbb{Z}_p$. Thus, it is not possible for $a^{(p-1)} \equiv 1 \pmod{p}$ to be true, since then it would follow that $a \cdot a^{(p-2)} \equiv 1 \pmod{p}$, and hence a would have a (multiplicative) inverse.

But the total number of sequences $r_0, \ldots, r_n$ such that (28) holds is precisely Πm_i. Hence every such sequence must be a sequence of remainders of some r, $0 \leq r < \Pi m_i$. □

Note that the CRT can be stated in the language of group theory as follows:

$$\mathbb{Z}_{m_1 \cdot m_2 \cdot \ldots \cdot m_n} \cong \mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_n},$$

where the m_i's are pairwise co-prime.

EXERCISE 8.24. The proof of theorem 8.23 (CRT) is non-constructive. Show how to obtain the r that meets the requirement of the theorem—efficiently, i.e., without using brute force search.

8.3. RSA

It is well known that Adam and Eve no longer trust each other.[4] Adam sets up a mechanism whereby he can receive and decode encoded messages from an arbitrary person—and no one else (Eve in particular) can read them. To this end, Adam advertises a function f, and *anyone* can compute $f(m)$ for *any* message m, but only Adam can *efficiently* compute m from $f(m)$ using the function g, where $g(f(m)) = m$.

Choose two odd primes p, q, and set $n = pq$. Choose $k \in \mathbb{Z}^*_{\phi(n)}$, $k > 1$. Advertise f, where $f(m) \equiv m^k \pmod{n}$. Compute $l = k^{-1}$ (inverse of k in $\mathbb{Z}^*_{\phi(n)}$). Now $\langle n, k \rangle$ are public, and the key l is secret, and so is the function g, where $g(C) \equiv C^l \pmod{n}$. (Note that $g(f(m)) \equiv_n m^{kl} \equiv_n m$.)

Note that computing the inverse of k in $\mathbb{Z}^*_{\phi(n)}$, that is l, can be done in poly-time using the extended Euclidean algorithm. Just observe that if $k \in \mathbb{Z}^*_{\phi(n)}$, then $\gcd(k, \phi(n)) = 1$, so $\exists s, t$ such that $sk + t\phi(n) = 1$, and further s, t can be chosen so that s is in $\mathbb{Z}^*_{\phi(n)}$ (first obtain any s, t from the extended Euclidean algorithm, and then just add to s the appropriate number of (positive or negative) multiples of $\phi(n)$ to place it in the set $\mathbb{Z}^*_{\phi(n)}$, and adjust t by the same number of multiples (of opposite sign)). Set $l := s$.

Obviously RSA relies on the hardness of factoring integers for its security; if we were able to factor n, we would obtain p, q, and hence $\phi(n) = \phi(pq) = (p-1)(q-1)$, and so we would be able to compute l.

The first question is: why $m^{kl} \equiv_n m$? Observe that $kl = 1 + (-t)\phi(n)$, where $(-t) > 0$, and so $m^{kl} \equiv_n m^{1+(-t)\phi(n)} \equiv_n m \cdot (m^{\phi(n)})^{(-t)} \equiv_n m$, because $m^{\phi(n)} \equiv_n 1$. Note that this last statement does not follow directly from Euler's theorem, because $m \in \mathbb{Z}_n$, and not necessarily in $\mathbb{Z}^*_n$; in fact m must be in $\mathbb{Z}_n - \{0, p, q, pq\}$, so we could insist that the messages m are small relative to n, so that $0 < m < \min\{p, q\}$—in fact, we break a large message into those small pieces. By Fermat's little theorem, we know that $m^{(p-1)} \equiv_p 1$ and $m^{(q-1)} \equiv_q 1$, so $m^{(p-1)(q-1)} \equiv_p 1$ and $m^{(q-1)(p-1)} \equiv_q 1$, thus $m^{\phi(n)} \equiv_p 1$ and $m^{\phi(n)} \equiv_q 1$. This means that $p | (m^{\phi(n)} - 1)$ and $q | (m^{\phi(n)} - 1)$, so, since p, q are distinct primes, it follows that $(pq) | (m^{\phi(n)} - 1)$, and so $m^{\phi(n)} \equiv_n 1$.

[4] See Genesis 3:15.

The second questions is: how to select random primes? Two random primes are needed to find the public key $n = pq$ for the RSA[5] encryption scheme. It is a non-trivial problem, primarily because verifying the primality of a number is difficult. Here is how we go about it: we know by the prime number theorem that there are about $\pi(n) = n/\log n$ many primes $\leq n$. This means that there are $2^n/n$ primes among n-bit integers, roughly 1 in n, and these primes are fairly uniformly distributed. So we pick an integer at random, in a given range, and apply a primality testing algorithm to it, which in practice means the Rabin-Miller test[6] (see §7.1.2, algorithm 7.2).

We now discuss very briefly two issues related to the security of RSA. The first one is that the primes p, q cannot be chosen "close" to each other. Note that

$$n = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2.$$

Since p, q are close, we know that $s := \dfrac{p-q}{2}$ is small, and $t := \dfrac{p+q}{2}$ is only slightly larger than $n^{\frac{1}{2}}$, and $t^2 - n = s^2$ is a perfect square. So we try the following candidate values for t:

$$\lceil n^{\frac{1}{2}} \rceil, \lceil n^{\frac{1}{2}} \rceil + 1, \lceil n^{\frac{1}{2}} \rceil + 2, \ldots$$

until $t^2 - n$ is a perfect square s^2. Clearly, if s is small, we will quickly find such a t, and then $p = t + s$ and $q = t - s$.

The second issue is the following: suppose that Eve can compute $\phi(n)$ from n. Then she can easily compute the primes p, q (of course, if she can compute $\phi(n)$ she can directly compute l, and she does not need p, q). To see this note that $\phi(n) = \phi(pq) = (p-1)(q-1)$. Then,

$$\begin{aligned} p + q &= n - \phi(n) + 1 \\ pq &= n, \end{aligned} \tag{29}$$

and from these two equations,

$$(x-p)(x-q) = x^2 - (p+q)x + pq = x^2 - (n - \phi(n) + 1)x + n.$$

Thus, we can compute p, q by computing the roots of this last polynomial, and using the quadratic formula $x = (-b \pm \sqrt{b^2 - 4ac})/2a$, we obtain that p, q are

$$\frac{(n - \phi(n) + 1) \pm \sqrt{(n - \phi(n) + 1)^2 - 4n}}{2}.$$

Suppose that Eve is able to compute l from n and k. If Eve knows l, then she knows that whatever $\phi(n)$ is, it divides $kl - 1$, so she has equations (29) but with $\phi(n)$ in the first equation replaced by $(kl-1)/a$, for some unknown a. There is a randomized polytime procedure to find the appropriate a, and obtain p, q, but we do not describe it here.

If Eve is able to factor she can obviously break RSA; on the other hand, if Eve can break RSA (by computing l from n, k), then she would be able to factor in randomized

[5]RSA is named after the first letters of the last names of its inventors: Ron **R**ivest, Adi **S**hamir, and Leonard **A**dleman.

[6]The fact that this method of selecting primes works is attested by the fact that encryption packages such as GPG use it, and they work very well.

polytime. Conceivably, Eve may be able to break RSA *without* computing l, so these observation do not imply that breaking RSA is as hard as factoring.

8.4. The Isolation Lemma

A *weight function* over a finite set U is a mapping from U to the set of positive integers. We naturally extend any weight function over U to one on the power set $\mathcal{P}(U)$ as follows. For each $S \subseteq U$, the weight of S with respect to a weight function W, denoted $W(S)$, is $\Sigma_{x \in S} W(x)$. Let F be a nonempty family of nonempty subsets of U. Call a weight function W *good* for F if there is exactly one minimum-weight set in F with respect to W. Call W *bad* for F otherwise.

LEMMA 8.25 (Isolation). Let U be a finite set. Let $F_1, \ldots, F_m$ be families of nonempty subsets over U, and let $D = |U|$. Let $R > mD$, and let Z be the set of all weight functions whose weights are at most R. Let α, $0 < \alpha < 1$, be such that $\alpha > \frac{mD}{R}$. Then, more than $(1-\alpha)|Z|$ functions in Z are good for all $F_1, \ldots, F_m$.

PROOF. Let F be one family. For a weight function $W \in Z$, let MinWeight_W denote the minimum weight of F with respect to W, i.e., $\text{MinWeight}_W = \min\{W(S) | S \in F\}$, and let MinWeightSet_W denote the set of all minimum-weight sets of F with respect to W, i.e., $\text{MinWeightSet}_W = \{S \in F | W(S) = \text{MinWeight}_W\}$. For $x \in U$, we say that the minimum-weight sets of F with respect to W are *ambiguous about inclusion* of x if there exist some $S, S' \in \text{MinWeightSet}_W$ such that $x \in (S - S') \cup (S' - S)$, i.e., x is in the symmetric difference of S, S'. This "ambiguity" refers to the situation where there exist two different sets in MinWeightSet_W, one with x, the other without. To repeat in yet another way, it means that given x, we cannot say that any minimal weight set must contain it, or any must not contain it.

We need two claims.

CLAIM 8.26. W is bad iff there is some $x \in U$ such that the minimum-weight sets of F with respect to W are ambiguous about inclusion of x.

Let $x \in U$ be fixed. We count the number of weight functions $W \in Z$ such that the minimum-weight sets of F with respect to W are ambiguous about inclusion of x. Let $y_1, \ldots, y_{D-1}$ be an enumeration of $U - \{x\}$ and $v_1, \ldots, v_{D-1} \in [R]$. Let A be the set of all weight functions W such that for all $i \in [D-1]$, $W(y_i) = v_i$. (Note that $|A| = R$, the reason being that for any given $W \in A$, all the values are fixed except for x, which can take one of $[R]$ values.) Suppose that there is a weight function W in A such that the minimum-weight sets of F with respect to W are ambiguous about inclusion of x.

CLAIM 8.27. Let W' be an arbitrary element of $A - \{W\}$. The minimum-weight sets of F with respect to W' are unambiguous about inclusion of x.

This means that there is at most one weight function $W \in A$ such that the minimum-weight sets of F with respect to W are ambiguous about inclusion of x.

For each $i \in [D-1]$ there are R choices for v_i. So, there are at most R^{D-1} weight functions $W \in Z$ such that the minimum-weight sets of F with respect to W are ambiguous about the inclusion of x.

There are R^D weight functions in Z, there are m choices for F, and there are D choices for x. Thus the proportion of

$$\{W \in Z | \text{for some } i \in [m],\ W \text{ is bad for } F_i\}$$

is at most

$$\frac{mDR^{D-1}}{R^D} = \frac{mD}{R} < \alpha.$$

Thus, the proportion of

$$\{W \in Z | \text{for all } i \in [m],\ W \text{ is good for } F_i\}$$

is more than $(1-\alpha)$. □

8.5. Berkowitz's algorithm

In this section we present a very slick algorithm for computing the characteristic polynomial (char poly) of an $n \times n$ matrix A, defined as usual to be

$$p_A(x) = \det(xI - A) = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n.$$

Berkowitz's algorithm, on input A, outputs the coefficients $c_0, c_1, c_2, \ldots, c_n$. It works by computing iterated matrix products—an operation that can be done in the complexity class NC^2 (actually in the class POW, where POW is the class of problems reducible to computing powers of an integer matrix, and $\mathsf{NC}^1 \subseteq \mathsf{POW} \subseteq \mathsf{NC}^2$; see [**Coo85**] for the background on POW, and related complexity classes).

Exercise 8.28. Show that matrix powering is in NC^2.

We give two presentations of Berkowitz's algorithm. The first uses Samuelson's identity, and it is algebraic in flavor. The second uses clow sequences, and it is combinatorial in flavor.

8.5.1. Samuelson's identity. The main idea in the original proof of correctness of Berkowitz's algorithm (see [**Ber84**]) is Samuelson's identity, which relates the char poly of a matrix to the char poly of its principal sub-matrix. Thus, the coefficients of the char poly of an $n \times n$ matrix A below, are computed in terms of the coefficients of the char poly of M:

$$A = \begin{bmatrix} a_{11} & R \\ S & M \end{bmatrix}, \tag{30}$$

where R, S and M are $1 \times (n-1)$, $(n-1) \times 1$ and $(n-1) \times (n-1)$ sub-matrices, respectively. Recall that the adjoint of a matrix A is the transpose of the matrix of cofactors of A; that is, the (i,j)-th entry of $\mathrm{adj}(A)$ is given by $(-1)^{i+j} \det(A[j|i])$. Also recall that $A[k|l]$ is the matrix obtained from A by deleting the k-th row and the l-th column. We also introduce the following notation: $A[-|l]$ denotes A with only the l-th column deleted, and $A[k|-]$ denotes A with only the k-th row deleted, and it makes sense to let $A[-|-] = A$.

LEMMA 8.29 (Samuelson's Identity). Let $p(x)$ and $q(x)$ be the char polys of A and M, respectively. Then:

$$p(x) = (x - a_{11})q(x) - R \cdot \operatorname{adj}(xI - M) \cdot S.$$

PROOF.

$$\begin{aligned} p(x) &= \det(xI - A) \\ &= \det \begin{bmatrix} x - a_{11} & -R \\ -S & xI - M \end{bmatrix}, \end{aligned}$$

using the cofactor expansion along the first row:

$$= (x - a_{11}) \det(xI - M) + \sum_{j=1}^{n-1} (-1)^j (-r_j) \det(\underbrace{-S(xI - M)[-|j]}_{(*)}),$$

where $R = (r_1 r_2 \dots r_{n-1})$, and the matrix indicated by $(*)$ is given as follows: the first column is S, and the remaining columns are given by $(xI - M)$ with the j-th column deleted. We expand $\det(-S(xI - M)[-|j])$ along the first column, i.e., along the column $S = (s_1 s_2 \dots s_{n-1})^T$ to obtain:

$$= (x - a_{11})q(x) + \sum_{j=1}^{n-1} (-1)^j (-r_j) \sum_{i=1}^{n-1} (-1)^{i+1} (-s_i) \det(xI - M)[i|j]$$

and rearranging:

$$\begin{aligned} &= (x - a_{11})q(x) - \sum_{i=1}^{n-1} \left(\sum_{j=1}^{n-1} r_j (-1)^{i+j} \det(xI - M)[i|j] \right) s_i \\ &= (x - a_{11})q(x) - R \cdot \operatorname{adj}(xI - M) \cdot S \end{aligned}$$

and we are done. □

LEMMA 8.30. Let $q(x) = q_{n-1}x^{n-1} + \cdots + q_1 x + q_0$ be the char poly of M, and let:

$$B(x) = \sum_{k=2}^{n} (q_{n-1}M^{k-2} + \cdots + q_{n-k+1}I)x^{n-k}. \tag{31}$$

Then $B(x) = \operatorname{adj}(xI - M)$.

For example, if $n = 4$, then

$$B(x) = Iq_3x^2 + (Mq_3 + Iq_2)x + (M^2q_3 + Mq_2 + Iq_1).$$

PROOF. First note that:

$$\operatorname{adj}(xI - M) \cdot (xI - M) = \det(xI - M)I = q(x)I.$$

Now multiply $B(x)$ by $(xI - M)$, and using the Cayley-Hamilton theorem, we can conclude that $B(x) \cdot (xI - M) = q(x)I$. Thus, the result follows as $q(x)$ is not the zero polynomial; i.e., $(xI - M)$ is *not* singular. □

From lemma 8.29 and lemma 8.30 we have the following identity which is the basis for Berkowitz's algorithm:

$$p(x) = (x - a_{11})q(x) - R \cdot B(x) \cdot S. \tag{32}$$

Using (32), we can express the char poly of a matrix as iterated matrix product. We say that an $n \times m$ matrix is *Toeplitz* if the values on each diagonal are the same. A matrix is *upper (lower) triangular* if all the values below (above) the main diagonal are zero.

If we express equation (32) in matrix form we obtain:

$$p = C_1 q, \tag{33}$$

where C_1 is an $(n+1) \times n$ Toeplitz lower triangular matrix, and where the entries in the first column are defined as follows:

$$c_{i1} = \begin{cases} 1 & \text{if } i = 1 \\ -a_{11} & \text{if } i = 2 \\ -(RM^{i-3}S) & \text{if } i \geq 3 \end{cases}. \tag{34}$$

For example, if A is a 4×4 matrix, then $p = C_1 q$ is given by:

$$\begin{bmatrix} p_4 \\ p_3 \\ p_2 \\ p_1 \\ p_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -a_{11} & 1 & 0 & 0 \\ -RS & -a_{11} & 1 & 0 \\ -RMS & -RS & -a_{11} & 1 \\ -RM^2S & -RMS & -RS & -a_{11} \end{bmatrix} \begin{bmatrix} q_3 \\ q_2 \\ q_1 \\ q_0 \end{bmatrix}.$$

Berkowitz's algorithm consists in repeating this for q (i.e., q itself can be computed as $q = C_2 r$, where r is the char poly of $M[1|1]$), and so on, eventually expressing p as a product of matrices:

$$p = C_1 C_2 \cdots C_n.$$

More precisely, given an $n \times n$ matrix A, over any field K, *Berkowitz's algorithms* computes an $(n+1)\times 1$ column vector p_A as follows: let C_j be an $(n+2-j)\times(n+1-j)$ Toeplitz and lower-triangular matrix, where the entries in the first column are defined as:

$$\begin{cases} 1 & \text{if } i = 1 \\ -a_{jj} & \text{if } i = 2 \\ -R_j M_j^{i-3} S_j & \text{if } 3 \leq i \leq n+2-j \end{cases} \tag{35}$$

where M_j is the j-th principal sub-matrix, so $M_1 = A[1|1]$, $M_2 = M_1[1|1]$, and in general $M_{j+1} = M_j[1|1]$, and R_j and S_j are given by:

$$\begin{pmatrix} a_{j(j+1)} & a_{j(j+2)} & \cdots & a_{jn} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} a_{(j+1)j} & a_{(j+2)j} & \cdots & a_{nj} \end{pmatrix}^t$$

respectively. Then:

$$p_A = C_1 C_2 \cdots C_n. \tag{36}$$

Berkowitz's algorithm works over any field, and in fact, as there are no divisions in the algorithm (i.e., the inverses of field elements are not needed), the algorithm works over any commutative ring. The same can be said of our proof of correctness—it also works over commutative rings (in the proof of lemma 8.30 we argue about the

inverse of $(xI - M)$, which exists, but we need not argue about the inverses of the ring elements). If the additions and multiplications in the commutative ring can be carried out in NC^1, then we have the following lemma.

LEMMA 8.31. Berkowitz's algorithm is an NC^2 algorithm.

PROOF. This follows from (36): p_A equals the product of $C_1, C_2, \ldots, C_n$, all independently computed in NC^2. The product of these matrices can be computed as follows: construct a matrix B by placing the C_i's above the main diagonal and letting all other entries be zero. Then, B^n will contain the product of the C_i's in its upper-right corner. We know that B^n can be computed in NC^2 from exercise 8.28.

Now the entries of each C_i can also be computed using matrix products, again independently of each other. In fact, we can compute the (i, j)-th entry of the k-th matrix very quickly as in (35).

Finally, we can compute additions, additive inverses, and products of the underlying field elements (in fact, more generally, of the elements in the underlying commutative ring, as we do not need divisions in this algorithm). We claim that these operations can be done with small NC^1 circuits (this is certainly true for the standard examples: finite fields, rationals, integers, etc.).

Thus we have "three layers": one layer of NC^1 circuits, and two layers of NC^2 circuits (one layer for computing the entries of the C_j's, and another layer for computing the product of the C_j's), and so we have (uniform) NC^2 circuits that compute the column vector with the coefficients of the char poly of a given matrix. □

8.5.2. Clow Sequences. First of all, a "clow" is an acronym for "closed walk." Clow sequences (introduced in [**MV97**], based on ideas in [**Str83**]), can be thought of as generalized permutations. They provide a combinatorial insight into what is actually being computed in Berkowitz's algorithm.

In section 8.5.1, we derived Berkowitz's algorithm from Samuelson's identity and the Cayley-Hamilton theorem. However, both these principles are in turn proven using Lagrange's expansion for the determinant. Thus, this proof of correctness of Berkowitz's algorithm is indirect, and it does not really show what is being computed in order to obtain the char poly efficiently.

To see what is being computed in Berkowitz's algorithm, and to understand the subtle cancellations of terms, it is useful to look at the coefficients of the char poly of the determinant of a matrix A as given by determinants of minors of A. To define this notion precisely, let A be an $n \times n$ matrix, and define $A[i_1, \ldots, i_k]$, where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, to be the matrix obtained from A by deleting the rows and columns numbered by $i_1, i_2, \ldots, i_k$. Thus, using this notation, $A[1|1] = A[1]$, and $A[2, 3, 8]$ would be the matrix obtained from A by deleting rows and columns $2, 3, 8$.

Now, it is easy to show from the Lagrange's expansion of $\det(xI - A)$, that if $p_n, p_{n-1}, \ldots, p_0$ are the coefficients of the char poly of A, then they are given by the following formulas:

$$p_k = (-1)^{n-k} \sum_{1 \leq i_1 < i_2 < \cdots < i_k \leq n} \det(A[i_1, i_2, \ldots, i_k]), \tag{37}$$

where the summation ranges over all subsets of $[n]$ of k elements. If $k = 0$, then $p_k = p_0 = (-1)^n \det(A)$, and if $k = n$, then $p_k = p_n = (-1)^0 = 1$. This is of course consistent with $\det(xI - A) = p_n x^n + p_{n-1}x^{n-1} + \cdots + p_0$, since p_n is 1, and p_0 is $(-1)^n$ times the determinant of A.

Since $\det(A[i_1, i_2, \ldots, i_k])$ can be computed using the Lagrange's expansion, it follows from (37), that each coefficient of the char poly can be computed by summing over permutations of minors of A:

$$p_{n-k} = \sum_{1 \le i_1 < i_2 < \cdots < i_{n-k} \le n} \sum_{\sigma \in S_k} \operatorname{sign}(\sigma) a_{j_1 \sigma(j_1)} a_{j_2 \sigma(j_2)} \cdots a_{j_k \sigma(j_k)}. \tag{38}$$

The relation between the j-indices and the i-indices is as follows: $\{j_1, j_2, \ldots, j_k\} = [n] - \{i_1, i_2, \ldots, i_{n-k}\}$. When $k = n$ we are simply computing the determinant, since in that case $\{i_1, i_2, \ldots, i_{n-k}\} = \emptyset$, and the other summation spans over all the permutations in S_n:

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sign}(\sigma) a_{1\sigma(1)} \cdots a_{n\sigma(n)}.$$

Finally, note that if $k = 0$, then the result is 1 by convention.

We can interpret $\sigma \in S_n$ as a directed graph G_σ on n vertices: if $\sigma(i) = j$, then (i, j) is an edge in G_σ, and if $\sigma(i) = i$, then G_σ has the self-loop (i, i). In this context, a permutation is called a *cycle cover*. For example, the permutation given by:

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 1 & 2 & 4 & 6 & 5 \end{pmatrix},$$

corresponds to the directed graph G_σ with 6 nodes and the following edges:

$$\{(1,3), (2,1), (3,2), (4,4), (5,6), (6,5)\},$$

where $(4, 4)$ is a self-loop. Given a matrix A, define the weight of G_σ, $w(G_\sigma)$, as the product of a_{ij}'s such that $(i, j) \in G_\sigma$. So G_σ in our running example has a weight given by: $w(G_\sigma) = a_{13}a_{21}a_{32}a_{44}a_{56}a_{65}$. Using the new terminology, we can restate equation (38) as follows:

$$p_{n-k} = \sum_{1 \le i_1 < i_2 < \cdots < i_{n-k} \le n} \sum_{\sigma \in S_k} \operatorname{sign}(\sigma) w(G_\sigma). \tag{39}$$

A problem with (39) is that there are lots of permutations ($k!$), and there is no (known) way of grouping or factoring them, in such a way so that we can compute the summation efficiently. The way to get around this problem is by generalizing the notion of permutation. Instead of summing over cycle covers, we sum over clow sequences; the paradox is that there are many more clow sequences than cycle covers, *but* we can efficiently compute the sums of clow sequences (with Berkowitz's algorithm), making a clever use of cancellations of terms as we go along. We now introduce all the necessary definitions, following [**MV97**].

A *clow* is a walk $(w_1, \ldots, w_l)$ starting from vertex w_1 and ending at the same vertex, where any (w_i, w_{i+1}) is an edge in the graph. Vertex w_1 is the least-numbered vertex in the clow, and it is called the *head* of the clow. We also require that the head occur only once in the clow. This means that there is exactly one incoming edge

(w_l, w_1), and one outgoing edge (w_1, w_2) at w_1, and $w_i \neq w_1$ for $i \neq 1$. The *length* of a clow $(w_1, \ldots, w_l)$ is l. Note that clows are never empty since they must have a head.

A *clow sequence* is a sequence of clows $(C_1, \ldots, C_k)$, where

$$\text{head}(C_1) < \ldots < \text{head}(C_k).$$

The *length* of a clow sequence is the sum of the lengths of the clows (i.e., the total number of edges, counting multiplicities). Note that a cycle cover is a special type of a clow sequence. We define the *sign* of a clow sequence to be $(-1)^k$ where k is the number of clows in the sequence.

Given a matrix A, we associate a weight with a clow sequence that is consistent with the contribution of a cycle cover. Note that we can talk about clows and clow sequences independently of a matrix, but once we associate weights with clows, we have to specify the underlying matrix, in order to label the edges. Thus, to make things more precise, we will sometimes say "clow sequences on A" to emphasize that the weights come from A.

Given a matrix A, the *weight of a clow* C, denoted $w(C)$, is the product of the weights of the edges in the clow, where edge (i, j) has weight a_{ij}.

Consider the clow C given by $(1, 2, 3, 2, 3)$ on four vertices. The head is vertex 1, and the length is 6. The weight is given by: $w((1, 2, 3, 2, 3)) = a_{12}a_{23}^2 a_{32}a_{31}$.

Given a matrix A, the *weight of a clow sequence* C, denoted $w(C)$, is the product of the weights of the clows in C. Thus, if $C = (C_1, \ldots, C_k)$, then:

$$w(C) = \prod_{i=1}^{k} w(C_i).$$

We make the convention that an empty clow sequence has weight 1. Since a clow must consist of at least one vertex, a clow sequence is empty iff it has length zero. Thus, equivalently, a clow sequence of length zero has weight 1. These statements will be important when we link clow sequences with Berkowitz's algorithm.

THEOREM 8.32. Let A be an $n \times n$ matrix, and let $p_n, p_{n-1}, \ldots, p_0$ be the coefficients of the char poly of A given by $\det(xI - A)$. Then:

$$p_{n-k} = \sum_{C \in \mathcal{C}_k} \text{sign}(C) w(C), \tag{40}$$

where $\mathcal{C}_k = \{C | C \text{ is a clow sequence on } A \text{ of length } k\}$.

PROOF. We generalize the proof given in [**MV97**, pp. 5–8] for the case $k = n$. The main idea in the proof is that clow sequences which are not cycle covers cancel out, so the contribution of clow sequences which are not cycles covers is zero.

Suppose that $(C_1, \ldots, C_j)$ is a clow sequence in A of length k. Choose the smallest i such that $(C_{i+1}, \ldots, C_j)$ is a set of disjoint cycles. If $i = 0$, $(C_1, \ldots, C_j)$ is a cycle cover. Otherwise, if $i > 0$, we have a clow sequence which is not a cycle cover, so we show how to find another clow sequence (which is also not a cycle cover) of the same weight and length, but opposite sign. The contribution of this pair to the summation in (40) will be zero.

So suppose that $i > 0$, and traverse C_i starting from the head until one of two possibilities happens: (i) we hit a vertex that is in $(C_{i+1}, \ldots, C_j)$, or (ii) we hit a vertex that completes a simple cycle in C_i. Denote this vertex by v. In case (i), let C_p be the intersected clow ($p \geq i+1$), join C_i and C_p at v (so we merge C_i and C_p). In case (ii), let C be the simple cycle containing v: detach it from C_i to get a new clow.

In either case, we created a new clow sequence, of *opposite sign but same weight and same length k*. Furthermore, the new clow sequence is still not a cycle cover, and if we would apply the above procedure to the new clow sequence, we would get back the original clow sequence (hence our procedure defines an *involution* on the set of clow sequences). □

In [**Val92**] Valiant points out that Berkowitz's algorithm computes sums of what he calls "loop covers." We show that Berkowitz's algorithm computes sums of slightly restricted clow sequences, which are nevertheless equal to the sums of all clow sequences, and therefore, by theorem 8.32, Berkowitz's algorithm computes the coefficients p_{n-k} of the char poly of A correctly. We formalize this argument in the next theorem.

THEOREM 8.33. Let A be an $n \times n$ matrix, and let $p_A = (p_n p_{n-1} \ldots p_0)$, as defined by (36); that is, p_A is the result of running Berkowitz's algorithm on A. Then, for $0 \leq k \leq n$, we have:

$$p_{n-k} = (-1)^{n-k} \sum_{C \in \mathcal{C}_k} \text{sign}(C) w(C), \tag{41}$$

where $\mathcal{C}_k = \{C | C \text{ is a clow sequence on } A \text{ of length } k\}$.

Before we prove this theorem, we give an example. Suppose that A is a 3×3 matrix, $M = A[1|1]$ as usual, and p_3, p_2, p_1, p_0 are the coefficients of the char poly of A and q_2, q_1, q_0 are the coefficients of the char poly of M, computed by Berkowitz's algorithm. Thus:

$$\begin{aligned} \begin{bmatrix} p_3 \\ p_2 \\ p_1 \\ p_0 \end{bmatrix} &= \begin{bmatrix} 1 & 0 & 0 \\ -a_{11} & 1 & 0 \\ -RS & -a_{11} & 1 \\ -RMS & -RS & -a_{11} \end{bmatrix} \begin{bmatrix} q_2 \\ q_1 \\ q_0 \end{bmatrix} \\ &= \begin{bmatrix} q_2 \\ -a_{11}q_2 + q_1 \\ -RSq_2 - a_{11}q_1 + q_0 \\ -RMSq_2 - RSq_1 - a_{11}q_0 \quad (*) \end{bmatrix} \end{aligned} \tag{42}$$

We assume that the coefficients q_2, q_1, q_0 are given by sums of clow sequences on M, that is, by clow sequences on vertices $\{2, 3\}$. Using this assumption and equation (42), we show that p_3, p_2, p_1, p_0 are given by clow sequences on A, just as in the statement of theorem 8.33.

Since $q_2 = 1$, $p_3 = 1$ as well. Note that $q_2 = 1$ is consistent with our statement that it is the sum of restricted clow sequences of length zero, since there is only one empty clow sequence, and by convention its weight is 1.

Consider p_2, which by definition is supposed to be the sum of clow sequences of length one on all three vertices. This is the sum of clow sequences of length one on vertices 2 and 3 (i.e., q_1), plus the clow sequence consisting of a single self-loop on vertex 1 with weight a_{11} and sign $(-1)^1 = -1$. Hence, the sum is indeed $-a_{11}q_2 + q_1$, as in equation (42) (again, $q_2 = 1$).

Consider p_1. Since $p_1 = p_{3-2}$, p_1 is the sum of clow sequences of length two. We are going to show that the term $-RSq_2 - a_{11}q_1 + q_0$ is equal to the sum of clow sequences of length 2 on A. First note that there is just one clow of length two on vertices 2 and 3, and it is given by q_0. There are two clows of length two which include a self loop at vertex 1. These clows correspond to the term $-a_{11}q_1$. Note that the negative sign comes from the fact that q_1 has a negative value, but there are two clows per sequence, so the parity is even. Finally, we consider the clow sequences of length two, where there is no self loop at vertex 1. Since vertex 1 must be included, there are only two possibilities; these clows correspond to the term $-RSq_2$ which is equal to: $-\begin{bmatrix} a_{12} & a_{13} \end{bmatrix}\begin{bmatrix} a_{21} \\ a_{31} \end{bmatrix} = -a_{12}a_{21} - a_{13}a_{31}$ since $q_2 = 1$.

For p_0, note that the summation indicated by $(*)$ includes only those clow sequences which start at vertex 1. This is because, the bottom entry in (42), unlike the other entries, does not have a 1 in the last column, and hence there is no coefficient from the char poly of M appearing by itself. This is not a problem for the following reason: if vertex 1 is not included in a clow sequence computing the last entry, then that clow sequence will cancel out anyways, since a clow sequence of length 3 that avoids the first vertex, cannot be a cycle cover! This observation will be made more explicit in the proof below.

EXERCISE 8.34. Show that $\sum_{C\in\mathcal{C}_k} \operatorname{sign}(C)w(C) = \sum_{C\in\mathcal{C}'_k} \operatorname{sign}(C)w(C)$ where $\mathcal{C}'_k$ is the set of clow sequences on A of length k such that $\operatorname{head}(C_1) = 1$.

PROOF. (of theorem 8.33) We prove this theorem by induction on the size of matrices. The Basis Case is easy, since if A is a 1×1 matrix, then $A = (a)$, so $p_A = \begin{pmatrix} 1 & -a \end{pmatrix}$, so $p_1 = 1$, and $p_0 = -a$ which is (-1) times the sum of clow sequences of length 1.

In the induction step, suppose that A is an $(n+1) \times (n+1)$ matrix and:

$$\begin{bmatrix} p_{n+1} \\ p_n \\ p_{n-1} \\ p_{n-2} \\ p_{n-3} \\ \vdots \\ p_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \dots \\ -a_{11} & 1 & 0 & \dots \\ -RS & -a_{11} & 1 & \dots \\ -RMS & -RS & -a_{11} & \dots \\ -RM^2S & -RMS & -RS & \dots \\ \vdots & \vdots & \vdots & \ddots \\ -RM^{n-1}S & -RM^{n-2}S & -RM^{n-3}S & \dots \end{bmatrix} \begin{bmatrix} q_n \\ q_{n-1} \\ q_{n-2} \\ q_{n-3} \\ \vdots \\ q_0 \end{bmatrix}, \qquad (43)$$

By the induction hypothesis, $q_M = \begin{pmatrix} q_n & q_{n-1} & \dots & q_0 \end{pmatrix}$ satisfies the statement of the theorem for $M = A[1|1]$, that is, q_{n-i} is equal to the sum of clow sequences of length i on $M = A[1|1]$.

Since $p_{n+1} = q_n$, $p_{n+1} = 1$. Since $p_n = -a_{11}q_n + q_{n-1} = -a_{11} + q_{n-1}$ (as $q_n = 1$), using the fact that q_{n-1} = the sum of clow sequences of length 1 on M, it follows that p_n = the sum of clow sequences of length 1 on A.

Now we prove this for general $n + 1 > i > 1$, that is, we prove that p_{n+1-i} is the sum of clow sequences of length i on A. Note that:

$$p_{n+1-i} = -RM^{i-2}Sq_n - RM^{i-3}Sq_{n-1} - \cdots - RSq_{n+2-i} - a_{11}q_{n+1-i} + q_{n-i} \quad (44)$$

as can be seen by inspection from equation (43). Observe that the (i, j)-th entry of M^k is the sum of walks in M that start at vertex i and end at vertex j of length k, and therefore, RM^kS is the sum of clows in A that start at vertex 1 (and of course end at vertex 1, and vertex 1 is never visited otherwise), of length $k + 2$.

Therefore, $RM^{i-2-j}Sq_{n-j}$, for $j = 0, \ldots, i-2$, is the product of the sum of clows of length $i - j$ (that start and end at vertex 1) and the sum of clow sequences of length j on M (by the induction hypothesis), which is just the sum of clow sequences of length i where the first clow starts and ends at vertex 1, and has length $i-j$. Each clow sequence of length i on A starts off with a clow anchored at the first vertex, and the second to last term of equation (44), $-a_{11}q_{n+1-i}$, corresponds to the case where the first clow is just a self loop. Finally, the last term given by q_{n-i} contributes the clow sequences of length i which do *not* include the first vertex.

The last case is when $i = n + 1$, so p_0, which is the determinant of A, by theorem 8.32. As was mentioned at the end of example 8.5.2, this is a *special* sum of clow sequences, because the head of the first clow is always vertex 1. Here is when we invoke the proof of the theorem 8.32: the last entry, p_0 can be shown to be the sum of clow sequences, where the head of the first clow is always vertex 1, by following an argument analogous to the one in the above paragraph. However, this sum is still equal in value to the sum of all clow sequences (of length $n + 1$). This is because, if we consider clow sequences of length $n + 1$, and there are $n + 1$ vertices, and we get a clow sequence C which avoids the first vertex, then we know that C *cannot* be a cycle cover, and therefore it will cancel out in the summation anyways, just as it was shown to happen in the proof of theorem 8.32. □

8.6. Answers to selected exercises

Exercise 8.5. See [**CLRS09**, theorem 34.15, pg. 1014].

Exercise 8.8. See [**CLRS09**, problem 34-3, pg. 1019].

Exercise 8.22. Here is an alternative proof, without group theory (recall that we are concerned with elements of $\mathbb{Z}_p^*$ only!): show that if there is an a such that the test fails for a (i.e., $a^{p-1} \not\equiv 1 \pmod p$), then for every b for which the test passes, there is a c for which the test fails. Suppose the test passes for b. Then, let $c \equiv ab \pmod p$. It remains to show that if the test passes for b_1, b_2, then $ab_1 \not\equiv ab_2 \pmod p$. This follows from the fact that $ab_1 \equiv ab_2 \pmod p$ implies $a(b_1 - b_2) \equiv 0 \pmod p$, so p must divide $a(b_1 - b_2)$, so $pk = a(b_1 - b_2)$, and since $b_1, b_2 \in \mathbb{Z}_p$, $|b_1 - b_2| < p$, so $\operatorname{lcm}(a, p) < ap$, so $\gcd(a, p) > 1$, which is a contradiction, since we assume that $a \in \mathbb{Z}_p^*$.

Exercise 8.24. Construct the r in stages, so that at stage i it meets the first i congruences, that is, at stage i we have that $r \equiv r_j \pmod{m_j}$ for $j \in \{1, 2, \ldots, i\}$. Stage 1 is simple: just set $r \longleftarrow r_1$. Suppose the first i stages have been completed; let $r \longleftarrow r + (\Pi_{j=1}^{i} m_j)x$, where x satisfies $x \equiv (\Pi_{j=1}^{i} m_j)^{-1}(r_{i+1} - r) \pmod{m_{i+1}}$. We know that the inverse of $(\Pi_{j=1}^{i} m_j)$ exists (in $\mathbb{Z}_{m_{i+1}}$) since $\gcd(m_{i+1}, (\Pi_{j=1}^{i} m_i)) = 1$, and furthermore, this inverse can be obtained efficiently with the Extended Euclidean algorithm.

Exercise 8.28. The product of two matrices can be computed with Boolean circuits of polynomial size and logarithmic depth (i.e., in NC^1), and the n-th power of a matrix can be obtained by repeated squaring (squaring $\log n$ many times for a matrix of size $n \times n$).

8.7. Notes

An important application of the Isolation Lemma (lemma 8.25) was the proof of Toda's theorem (§7.5). Another is the proof of $\mathsf{NL}/\text{poly} = \mathsf{UL}/\text{poly}$; see [**RA00**]. It is an interesting open problem whether NL and UL are equal without the advice. Lemma 8.12 is based on [**Sip06**, exercise 7.24]. §8.4 is based on [**HO02**, §4.1.1].

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[AGP94] William R. Alford, Andrew Granville, and Carl Pomerance. There are infinitely many Carmichael numbers. *Annals of Mathematics*, 139(3):703–722, 1994.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[Bab90] László Babai. E-mail and the unexpected power of interaction. In *Proceedings, 5th Structure in Complexity Theory Conference*, pages 30–44, 1990.

[Bea94] Paul Beame. A switching lemma primer. Technical report UW-CSE-95-07-01, Department of Computer Science and Engineering, University of Washington, 1994.

[Ben62] James Bennett. *On Spectra*. PhD thesis, Princeton University, 1962.

[Ber84] Stuart J. Berkowitz. On computing the determinant in small parallel time using a small number of processors. *Information Processing Letters*, 18(3):147–150, 1984.

[BM77] John Bell and Moshé Machover. *A course in mathematical logic*. North-Holland, 1977.

[BP96] Paul Beame and Toniann Pitassi. Simplified and improved resolution lower bounds. In *Proceedings, 37th IEEE Symposium on Foundations of Computer Science*, pages 274–282, 1996.

[CCG+94] Richard Chang, Benny Chor, Oded Goldreich, Juris Hartmanis, Johan Hastad, Desh Ranjan, and Pankaj Rohatgi. The random oracle hypothesis is false. *Journal of Computer and System Sciences*, 49(1):24–39, 1994.

[Chu96] Alonzo Church. *Introduction to Mathematical Logic*. Princeton University Press, 1996.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Book Company, 2009. Third Edition.

[CN10] Stephen A. Cook and Phuong Nguyen. *Logical Foundations of Proof Complexity*. Cambridge Univeristy Press, 2010.

[Coo71] Stephen A. Cook. The complexity of theorem proving procedures. In *Proceedings, 3rd ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[Coo76] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 1976.

[Coo85] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Computation*, 64(13):2–22, 1985.

[Coo00] Stephen A. Cook. The P versus NP problem. Manuscript prepared for the Clay Mathematics Institute for the Millennium Prize Problems and available from Clay Mathematics Institute web site, 2000.

[Coo08] Stephen A. Cook. Computability and logic: Lecture notes. Available at the author's web site, 2008.

[Dev05] Keith Devlin. *The Millennium Problems*. Granta Books, 2005.

[Eck87] Roger Eckhardt. Stan Ulam, John von Neumann, and the Monte Carlo method. *Los Alamos Science*, 15:131–137, 1987.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability*. Bell Telephone Laboratories, 1979.

[HA99] David Hilbert and Wilhelm Ackermann. *Principles of Mathematical Logic*. American Mathematical Society, 1999.

[HO02] Lane A. Hemaspaandra and Mitsunori Ogihara. *The Complexity Theory Companion*. Springer, 2002.

[Hof98] Paul Hoffman. *The Man Who Loved Only Numbers: The Story of Paul Erdős and the Search for Mathematical Truth*. Hyperion, 1998.

[Imm99] Neil Immerman. *Descriptive Complexity*. Springer-Verlag, 1999.

[KL80] Richard M. Karp and Richard J. Lipton. Some connections between nonuniform and uniform complexity classes. In *Proceedings, 12th ACM Symposium on Theory of Computing*, pages 302–309, 1980.

[Koz06] Dexter Kozen. *Theory of Computation*. Springer, 2006.

[KR87] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[Liv05] Mario Livio. *The equation that couldn't be solved*. Simon & Schuster, 2005.

[Min62] Marvin Minsky. Size and structure of universal Turing machines using tag systems. In *Recursive function theory: Proceedings, AMS Symposium in Pure Mathematics*, volume 5, pages 229–238, 1962.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[MV97] M. Mahajan and V. Vinay. Determinant: Combinatorics, algorithms, and complexity. *Chicago Journal of Theoretical Computer Science*, 1997(5):1–26, 1997.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Pit02] Toniann Pitassi. Propositional proof complexity: Lecture notes. Graduate course, CS2429, taught at the University of Toronto in the Fall 2002. Available at the author's web page, 2002.

[RA00] Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. *SIAM Journal on Computing*, 29(4):1118–1131, 2000.

[She59] John C. Shepherdson. The reduction of two-way automata to one-way automata. *IBM Journal of Research and Development*, 3(2):198–200, 1959.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation*. Thompson, 2006. Second Edition.

[SP95] Uwe Schöning and Randall Pruim. *Gems of Theoretical Computer Science*. Springer, 1995.

[SS77] R. Solovay and V. Strassen. A fast monte-carlo test for primality. *SIAM Journal of Computing*, 6:84–86, 1977.

[Str83] Howard Straubing. A combinatorial proof of the Cayley-Hamilton theorem. *Discrete Mathematics*, 43(2–3):273–279, 1983.

[Tur37] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1):230–265, 1937.

[Urq95] Alasdair Urquhart. The complexity of propositional proofs. *Bulletin of Symbolic Logic*, 1(4):425–467, 1995.

[Val92] Leslie G. Valiant. Why is Boolean complexity theory difficult? In *Boolean Function Complexity*, volume 169 of *London Mathematical Society Lecture Notes Series*, pages 84–94. Cambridge University Press, 1992.

[vzGG99] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge University Press, 1999.

[Weg87] Ingo Wegener. *The Complexity of Boolean Functions*. ohn Wiley & Sons Ltd., 1987.

Index