Complexity Theory

Course Notes

Sebastiaan A. Terwijn

Radboud University Nijmegen
Department of Mathematics
P.O. Box 9010
6500 GL Nijmegen
the Netherlands
terwijn@math.ru.nl

Version: November 2015.

Contents

Chapter 1

Introduction

1.1 Complexity theory

Complexity theory is concerned with the resources, such as time and space, needed to solve computational problems. After the success of the general theory of computability, that gave us a precise definition of the notion of *algorithm* and fundamental insights into the notion of mathematical proof and undecidability, it was only natural to also consider the notion of *efficient* computation, and ask why some problems that are computable in principle resist all attempts to compute them in a reasonable amount of time. There is an abundance of problems from mathematics and computer science that are trivially solvable by brute force search of an exponential number of instances, but for which currently no efficient algorithm is known. Complexity theory is the appropriate setting for the study of such problems. It is also the home of one of the most fundamental open problems in mathematics, namely the famous NP versus P problem. Some 40 years after the discovery of this problem, complexity theory has matured into an extremely rich and fascinating subject, despite the lack of progress on some of its most fundamental problems.

1.2 Preliminaries

In the following sections we list some preliminaries and notation. Most of the material treated here can be found in textbooks such as [2], [4], [29], and [32]. The notation used below is mostly standard. Given a set Σ, the set of all finite strings of elements from Σ is denoted by Σ^*. (This set is also referred to as the set of all *words* over the alphabet Σ.) The $*$-operator is called *Kleene star*. We usually work over the set of finite binary strings $\{0,1\}^*$, that can be interpreted as binary representations of the natural numbers $\mathbb{N}$. The length of a string x is denoted by $|x|$, and $\{0,1\}^n$ denotes the set of strings of length n. The concatenation of strings x and y is denoted by $x\widehat{\ }y$ or simply by xy. The

i-fold concatenation of x with itself is denoted by x^i. Whether x^i denotes the i-fold concatenation of the string x or the i-th power of the natural number x will always be clear from the context. We let $x \sqsubseteq y$ denote that the string x is an initial segment of the string y. Throughout, $\langle \cdot\,, \cdot \rangle$ denotes a bijective *pairing function* from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$ (or equivalently from pairs of strings to strings) defined by

$$\langle x, y \rangle = \frac{1}{2}\Big((x+y+1)^2 - (x+y+1)\Big) + x = \frac{1}{2}\Big((x+y)^2 + 3x + y\Big).$$

Objects such as formulas and Turing machines that can be coded using strings from $\{0,1\}^*$ are routinely handled as such. Subsets $A \subseteq \{0,1\}^*$ are variously called *set*, *problem* (in case we think of it as a decision problem, namely to decide for a given element x whether $x \in A$), or *language*. We often identify a set A with its characteristic sequence, i.e. we write $A(x) = 1$ if $x \in A$ and $A(x) = 0$ otherwise. For a given set A, $\overline{A}$ denotes the complement of A, and $A{\restriction}x$ denotes the finite set $A \cap \{0 \ldots x\}$, or equivalently, the finite string $A(0)A(1)\ldots A(x)$. For a given class $\mathcal{C}$ of sets, co-$\mathcal{C}$ denotes the class $\big\{A : \overline{A} \in \mathcal{C}\big\}$ of all sets whose complement is in $\mathcal{C}$. $|A|$ denotes the cardinality of A.

Each chapter contains a section with exercises. Exercises that are more challenging or that require a more elaborate solution are marked with * .

1.3 Turing machines

Preliminaries about Turing machines can be found in [37]. (Note however that in complexity theory Turing machines are allowed to have any finite number of tapes.)

Given a Turing machine M and an input x, we use the notation $M(x)\!\downarrow$ to denote that the computation of M on x halts in a finite number of steps, and we write $M(x)\!\uparrow$ if this is not the case. A set or a function is *computable* if there is a Turing machine computing it.[1] For notational convenience, we often identify machines with the sets they compute. A set is *computably enumerable (c.e.)* if it is empty or the range of a computable function (cf. also Exercise 2.7.12.)

Given a computable set A and a machine M that computes A, we also say that M *recognizes* A, or that M *accepts* A.

A Turing machine is *nondeterministic* if at any computation step, there is a *set* of next possible states, rather than a single next state as in a de-

[1] In computability theory it is essential to work with *partial* computable functions, i.e. functions that need not be defined on all arguments. Partial functions that are everywhere defined are called *total*, and a computable function is total by definition. In complexity theory, because of the presence of time and space bounds, the distinction between total and partial functions is less important.

terministic computation. That means that, on any given input, there is a set of possible computation paths, rather than a single one. By definition, a nondeterministic Turing machine accepts an input when *some* computation path accepts it.

We will often use the existence of *universal* Turing machines, that is, the existence of a machine M with two arguments such that for any Turing machine M' there exists $i \in \mathbb{N}$ (thought of as a code for M') such that $M(i, x) = M'(x)$ for any x. Here $M(i, x)$ is defined as $M(\langle i, x\rangle)$. This means that there is an computable list $\{M_i\}_{i\in\mathbb{N}}$ of all Turing machines. This also holds for Turing machines with resource bounds, cf. Theorem 2.1.1.

1.4 Big O and small o

We use the following common notation to compare the asymptotic behavior of functions on $\mathbb{N}$. The phrase "for almost all $n \in \mathbb{N}$" means "for all n except perhaps finitely many".

$f \in O(g)$ if there is a constant $c \in \mathbb{N}$ such that for almost all n we have $f(n) \leqslant cg(n)$.

$f \in o(g)$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

We can rephrase this as follows: for every real constant $r > 0$ it holds that for almost every n, $f(n) \leqslant rg(n)$.

The Big-O notation is often used to introduce constants without having to name them: $f \in O(1)$ means that f is bounded by a constant. For example, $g(n) \leqslant n^{O(1)}$ means that g is at most of polynomial growth.

Big O and small o also have a dual notation, denoted with omegas:

$f \in \omega(g)$ if for every constant $c \in \mathbb{N}$ there are infinitely many n such that $f(n) \geqslant cg(n)$.

$f \in \Omega(g)$ if there is a real constant $r > 0$ such that for infinitely many n, $f(n) \geqslant rg(n)$.[2]

We also use the notation $\forall^\infty x\varphi(x)$ to denote that $\varphi(x)$ holds for almost all x. Similarly, $\exists^\infty x\varphi(x)$ denotes that there are infinitely many x such that $\varphi(x)$ holds.

1.5 Logic

We use the following terminology from logic. We assume the reader is familiar with the basics of propositional logic. Propositional formulas are also called

[2]This is the Hardy-Littlewood definition of Ω, see also Exercise 1.7.1. Knuth uses the version with "almost every n" instead of "infinitely many n", which is not equivalent.

Boolean formulas. A *literal* is a Boolean variable or the negation thereof. For a literal x its negation is denoted by $\bar{x}$. A *clause* is a disjunction of literals. Clauses are often denoted using set notation, so the set of literals $\{x_1, \ldots, x_k\}$ denotes the clause $x_1 \vee \ldots \vee x_k$. $\square$ denotes the empty clause, and stands for false. A Boolean formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. A formula is in *disjunctive normal form* (DNF) if it is a disjunction of conjunctions of literals.

1.6 Number theory

In this section we summarize some preliminaries from number theory that we will use later. The reader can skip this section and refer back to it later when needed.

For any integer n, $\mathbb{Z}/n\mathbb{Z}$ is the additive group of integers modulo n. $(\mathbb{Z}/n\mathbb{Z})^*$ denotes the multiplicative group of integers x with $\gcd(x, n) = 1$.

Theorem 1.6.1. (Chinese remainder theorem) *For all n and m such that $\gcd(n, m) = 1$ we have $(\mathbb{Z}/nm\mathbb{Z}) \cong (\mathbb{Z}/n\mathbb{Z}) \times (\mathbb{Z}/m\mathbb{Z})$.*

Proof. Define the isomorphism by $x \bmod nm \mapsto (x \bmod n, x \bmod m)$. $\square$

The order (cardinality) of $(\mathbb{Z}/n\mathbb{Z})^*$ is denoted by $\varphi(n)$. The function φ is called Euler's function. For p prime we have $\varphi(p) = p - 1$, as is easily seen from the definition of $(\mathbb{Z}/p\mathbb{Z})^*$.

Theorem 1.6.2. (Euler) *$x^{\varphi(n)} \equiv 1$ mod n for all x and n with $\gcd(x, n) = 1$.*

Proof. By definition, $\varphi(n)$ is the order of $(\mathbb{Z}/n\mathbb{Z})^*$. By Lagrange's theorem, the order of every element in the group divides the order of the group. $\square$

A special case of Theorem 1.6.2 is Fermat's little theorem: if p is prime then $a^{p-1} \equiv 1 \bmod p$ for every a with $\gcd(a, p) = 1$.
We will apply the above results later in section 8.1.

Definition 1.6.3. Given a prime $p > 2$, an element $r \in (\mathbb{Z}/p\mathbb{Z})^*$ is a *primitive root* modulo p if it has order $p - 1$, i.e.

(i) $r^{p-1} \equiv 1 \bmod p$, and

(ii) $r^i \not\equiv 1 \bmod p$ for every $1 < i < p - 1$.

Note that we can replace (ii) in the above definition by

(ii)′ $r^{p-1/q} \not\equiv 1 \bmod p$ for every prime factor q of $p - 1$.

Namely, if i is least such that $r^i \equiv 1$ and $r^{p-1} \equiv 1 \bmod p$ then $i \mid p - 1$, and also $r^{ai} \equiv 1$ for every multiple ai of i. So if there exists $i < p - 1$ with $r^i \equiv 1 \bmod p$ then there is a prime factor q of $p - 1$ with $r^{p-1/q} \equiv 1 \bmod p$.

Theorem 1.6.4. (Fermat) *A number $p > 2$ is prime precisely when there is a primitive root modulo p.*

Proof. Suppose that (ii)′ above holds. We prove that p is prime. If $r^{p-1} \equiv 1 \bmod p$ then the order of r divides $p-1$. By (ii)′ it cannot be a proper divisor, so the order of r is $p-1$. Hence $(\mathbb{Z}/p\mathbb{Z})^*$ has order $p-1$, and p is prime.

Conversely, suppose that n is not prime, and suppose that $r \in (\mathbb{Z}/n\mathbb{Z})^*$ satisfies $r^{n-1} \equiv 1 \bmod n$. By Theorem 1.6.2 we also have $r^{\varphi(n)} \equiv 1 \bmod n$, and since n is not prime, $\varphi(n) < n-1$. Since the order of r divides both $\varphi(n)$ and $n-1$ it is strictly smaller than $n-1$, hence (ii)′ does not hold. □

There exist also primitive roots modulo numbers n that are not prime: r is a *primitive root* modulo n if $r^i \not\equiv 1 \bmod n$ for every $i \mid \varphi(n)$. We will later use the fact that for every $p > 2$ prime, there is a primitive root modulo p^2, cf. Exercise 1.7.3.

1.7 Exercises

Exercise 1.7.1. (a) Show that $f \in \omega(g)$ if and only if $f \notin O(g)$ (for nonzero functions).
(b) Show that $f \in \Omega(g)$ if and only if $f \notin o(g)$.

Exercise 1.7.2. Prove by formula induction that every propositional formula can be put into conjunctive normal form.

Exercise 1.7.3. Suppose that $p > 2$ is prime. In this exercise we show that there exists a primitive root modulo p^2. Let r be a primitive root modulo p.
(a) Show that r is a primitive root modulo p^2 if both $r^p \not\equiv 1 \bmod p^2$ and $r^{p-1} \not\equiv 1 \bmod p^2$.
(b) Show that $r^p \not\equiv 1 \bmod p^2$.
(c) Show that $(r+p)^p \equiv r^p \bmod p^2$. Conclude that also $(r+p)^p \not\equiv 1 \bmod p^2$.
(d) Suppose that r is not a primitive root modulo p^2. Show that in that case $r+p$ is a primitive root modulo p^2.

Chapter 2

Basics

2.1 Time and space bounds

In the following we will consider computations with bounded resources. It is customary to measure the complexity of computations in the size of the input, which is usually its *length* when represented as a binary string. This means in particular that the size of a number $n \in \mathbb{N}$ is of order $\log n$. (Unless stated otherwise, log is always to the base 2.) We only want to consider resource bounds that are reasonable in the following sense.

- $t : \mathbb{N} \to \mathbb{N}$ is *time constructible* if there is a Turing machine which on every input of length n halts in precisely $t(n)$ steps.
- $s : \mathbb{N} \to \mathbb{N}$ is *space constructible* if there is a Turing machine that on every input of length n halts in a configuration with exactly $s(n)$ non-blank tape cells and that used no other cells during the computation.[1]

For time constructible bounds it is possible to construct an effective enumeration of all Turing machines that are *clocked* with the given bound. Thus we can avoid decidability problems arising when considering arbitrary Turing machines. From now on we assume that all complexity bounds are time or space constructible. What may happen in absence of this assumption is discussed in section 5.5.

Given such constructible resource bounds we define the following classes:

- TIME(t) is the class of all sets that are accepted by a Turing machine within running time $t(n)$ for all inputs of length n.

[1] Note that an input of length n automatically uses n tape cells of the input tape. In order to let *sublinear* space bounds such as $\log n$ make sense, it is customary in complexity theory to equip Turing machines with a separate input tape, and to not count the input tape when counting the space used in a computation.

- NTIME(t) is defined as TIME(t), but now with nondeterministic Turing machines.
- SPACE(s) is the class of all sets that are accepted by a Turing machine using space at most $s(n)$ for all inputs of length n.
- NSPACE(s) is defined as SPACE(s), but now with nondeterministic Turing machines.

We will often use the following basic result about universal Turing machines. A similar result holds for nondeterministic Turing machines.

Theorem 2.1.1. *There exists an enumeration $\{M_e\}_{e\in\mathbb{N}}$ of all Turing machines such that the set*

$$\left\{\langle e, x, 1^t\rangle : M \text{ accepts } x \text{ in} \leqslant t \text{ steps}\right\}$$

is in TIME(n^2).

Proof. This follows in the same way as the existence of universal Turing machines. The bound n^2 is necessary for the simulation of Turing machines with many tapes by a Turing machine with two tapes, though better bounds are possible. For details see e.g. [19, p292] or [28, p82]. □

2.2 Inclusions between classes

We have the following inclusions between classes:

- TIME(t) $\subseteq$ NTIME(t), SPACE(s) $\subseteq$ NSPACE(s).
- TIME(t) $\subseteq$ SPACE(t), NTIME(t) $\subseteq$ NSPACE(t). This holds because a machine cannot use more space than time.
- SPACE($c \cdot s$) $\subseteq$ SPACE(s) for any $c \in \mathbb{N}$. This result is called the *tape compression theorem*. The idea of the proof is to increase the tape alphabet by introducing a symbol for every block of length c. Idem for NSPACE. The result means that we can work with $O(s)$ rather than s in the future.
- TIME($c \cdot t$) $\subseteq$ TIME(t) for any $c \in \mathbb{N}$ and t with $n \in o(t)$, i.e. such that $\forall r > 0 \forall^\infty n(n < r \cdot t(n))$. This result is called the *linear speed-up theorem*. The proof is slightly more complicated than that of the previous item, but the idea is similar: again use symbols for c-blocks, and simulate several steps with one by monitoring the actions of the tape head within a given region.

- NTIME(t) $\subseteq$ SPACE(t). Reuse space! We only need to keep a counter of length t for the current path. This gives NTIME(t) $\subseteq$ SPACE($2t$) = SPACE(t).
- NSPACE(s) $\subseteq$ TIME($2^{O(s)}$). This holds because there are only $2^{s(n)}$ configurations of length $s(n)$. Starting with the initial configuration, inductively make a list of all configurations reachable by the nondeterministic machine from the ones already found, and see if there is an accepting one. This takes at most $2^{s(n)}$ rounds, and every round has at most $2^{s(n)}$ steps. Note that it follows from this and the previous item that NTIME(t) $\subseteq$ TIME($2^{O(t)}$).
- NSPACE(s) $\subseteq$ SPACE(s^2). This is *Savitch's Theorem* proved below.

Savitch's Theorem can be proved using that the path problem for graphs is solvable in space $\log^2 n$, cf. Theorem 2.2.1 and Exercise 2.7.1 below. The problem PATH (also known as reachability or connectivity) is defined by
INPUT: a directed graph G, and nodes $x, y \in G$,
QUESTION: is there a path from x to y?

Theorem 2.2.1. PATH $\in$ SPACE($\log^2 n$).

Proof. Let Path(x, y, i) express that there is a path from x to y of length $\leqslant 2^i$. Since there are n nodes, we will try to determine whether Path$(x, y, \lceil \log n \rceil)$.
If $i = 0$ we can check Path(x, y, i) directly from the input.
If $i > 0$, check for all $z \in G$ whether Path$(x, z, i-1)$ and Path$(z, y, i-1)$.
For the second, we can *reuse* the space of the first!
We have to keep a stack of answers to the recursive calls. The depth of the recursion is $\log n$, and the current value is of size $3 \log n$ (because we work with triples (x, y, i)), so the cost is at most $3 \log^2 n$. □

Note that in the path problem, the path from x to y can have length n, which does not fit in space $\log^2 n$, so that the algorithm from Theorem 2.2.1 can really only give a yes/no answer.

2.3 Hierarchy theorems

The following hierarchy theorems were proven by Hartmanis and Stearns in 1965, and can be seen as the start of the field of complexity theory.

Theorem 2.3.1. (Space Hierarchy Theorem) SPACE(s) $\not\supseteq$ SPACE(s') *for every* $s' \in \omega(s)$, *i.e. such that* $\forall c\ \exists^\infty n\ (s'(n) > c \cdot s(n))$.

Theorem 2.3.2. (Time Hierarchy Theorem) TIME(t') $\not\subseteq$ TIME(t) *for every* $t' \in \omega(t \log t)$. *In particular, if also* $t' \geqslant t$ *then* TIME(t) $\subsetneq$ TIME(t').

Proof of the hierarchy theorems. We only sketch the proof of the time hierarchy theorem, since the space version is similar. Given time bounds t and t' as above, consider an effective enumeration $\{M_i\}_{i\in\mathbb{N}}$ of all Turing machines working in time t. (Cf. Theorem 2.1.1. Note that we are using the time-constructibility of t here.) On input $w = 1^i 0u$ run $M_i(w)$ for $t'(|w|)$ steps. If M_i halts within this time, let $M(w) = 1 - M_i(w)$, and let $M(w) = 0$ otherwise. Then $M \neq M_i$ for every i, because t' is infinitely often large enough to complete a computation of M_i. The set computed by M is in $\mathrm{TIME}(t')$; the extra factor $\log t$ is needed to simulate many tapes by a fixed number of tapes. □

The proof just given is a variant of Cantor's famous diagonal argument showing that the set of reals is uncountable. The diagonalization method has been very fruitful in computability and complexity theory. We will come back to it in Chapter 5.

There are also hierarchy theorems for the nondeterministic classes. The problem is how to do the diagonalization step $M(w) = 1 - M_i(w)$ from the proof above. In case of nondeterministic space, this can be arranged in a nice way with an appeal to Theorem 2.6.1. The following hierarchy theorem for nondeterministic time was proven by Cook [12].

Theorem 2.3.3. $\mathrm{NTIME}(n^2) \not\subseteq \mathrm{NTIME}(n)$.

Proof. Let $\{M_i\}_{i\in\mathbb{N}}$ be an enumeration of all nondeterministic Turing machines working in linear time. Such an enumeration can certainly be constructed in time n^2, that is, there is a nondeterministic Turing machine working in time n^2 that, given a code i and input x, simulates $M_i(x)$. We want to construct a nondeterministic machine M working in time n^2 such that for every i the requirement

$$R_i : \qquad \exists x\big(M(x) \neq M_i(x)\big)$$

is satisfied. Given i, it seems problematic to satisfy R_i directly, as in the proof of Theorem 2.3.2. Instead we pick a fresh candidate witness x and ensure the following:

(i) $M(y) = M_i(y+1)$ for all $y \in [x, 2^x)$,

(ii) $M(2^x) \neq M_i(x)$.

Item (ii) can be easily achieved by direct diagonalization, since in time 2^n there is enough time to simulate all computation paths of $M_i(x)$. To see that this suffices, suppose that $M_i(y) = M(y)$ for all $y \in [x, 2^x]$. Then in particular $M_i(y) = M_i(y+1)$ for every $y \in [x, 2^x)$ by (i), and hence $M_i(x) = M_i(2^x)$. But by (ii) we have $M_i(2^x) \neq M_i(x)$, a contradiction.

To satisfy all the requirements R_i in one construction, all we need to do is pick the witnesses x in such a way that the intervals $[x, 2^x]$ are disjoint. □

Using a more careful analysis, a somewhat tighter version of the previous hierarchy theorem can be obtained. Of course, the proof generalizes also to larger time bounds.

2.4 Central complexity classes

There are many important complexity classes, but any list of them would certainly include the following.

- $\mathrm{P} = \bigcup_i \mathrm{TIME}(n^i)$ is the class of *polynomial time computable sets*.
- $\mathrm{NP} = \bigcup_i \mathrm{NTIME}(n^i)$ is the class of *nondeterministic polynomial time computable sets*.
- $\mathrm{PSPACE} = \bigcup_i \mathrm{SPACE}(n^i)$ is the class of *polynomial space computable sets*.
- $\mathrm{EXP} = \bigcup_i \mathrm{TIME}(2^{n^i})$ is the class of *exponential time computable sets*.

Note that by the results of the previous section we have

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP} \tag{2.1}$$

and that by Savitch's Theorem it holds that PSPACE = NPSPACE (since $(n^i)^2$ is still polynomial). Note also that by the time hierarchy theorem, $\mathrm{P} \subsetneq \mathrm{EXP}$, hence some of the inclusions in (2.1) must be strict! Unfortunately, we have no clue which ones, but the conjecture is that in fact all the inclusions are strict.

One of the central themes in complexity theory is the difference between determinism and nondeterminism, and the tradeoff between time and space. This translates into the most prominent questions $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ and $\mathrm{P} \stackrel{?}{=} \mathrm{PSPACE}$. P is often identified with the class of problems that are solvable in "reasonable" time, although it may of course depend on the context what one finds reasonable. In contrast, NP corresponds to the class of problems of which it may be hard to find solutions, but of which one can efficiently check whether a candidate solution really is one. Because it does seem not very useful if one cannot even check whether a solution is correct, one could with a bit of exaggeration say that NP is the largest class of problems that one would be interested in knowing the answers to. Given a nondeterministic Turing machine M working in time n^c, one can view a computation of M as a binary tree of depth n^c, so that every binary string of length n^c corresponds with a computation path, and acceptance means that there is an accepting path, cf. Figure 2.1. (Cf. also Exercise 2.7.4.) In discussing NP-problems one often says: "guess" a path and then perform the polynomial computation. An accepting path is sometimes referred to as a *certificate* for the acceptance of the computation.

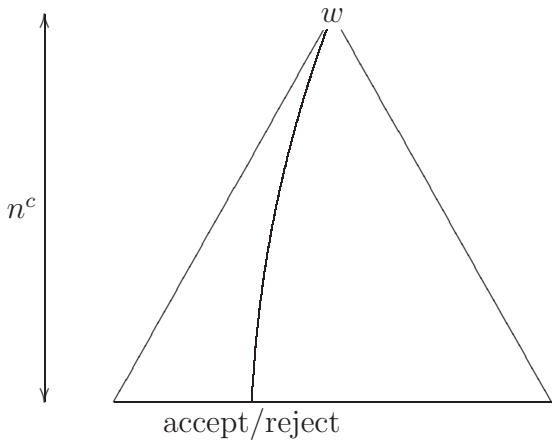


Figure 2.1: A nondeterministic computation

2.5 Problems from logic, algebra, and graph theory

The satisfiability problem SAT is defined by
INPUT: a Boolean formula φ,
QUESTION: is there a satisfying assignment of φ ?
It is easy to check that SAT $\in$ NP: Given φ of length n, φ has at most n propositional variables. Nondeterministically guess an assignment, and check that it satisfies φ. If so, accept, if not, reject.

The problem PRIMES is defined by:
INPUT: $n \in \mathbb{N}$,
QUESTION: is n prime ?
Obviously, PRIMES is in co-NP (guess a decomposition). It was a longstanding open problem whether PRIMES is in P, until this was finally solved in [1]. The proof is too long to include here, but we prove instead the weaker result that PRIMES is in NP. Note that NP$\cap$co-NP is not known to be equal to P. In section 6.4 we will discuss a randomized algorithm for PRIMES.

Theorem 2.5.1. (Pratt) PRIMES *is in* NP.

Proof. We use Theorem 1.6.4, namely that $p > 2$ is prime precisely when there is a primitive root modulo p, that is, an element $1 < x < p$ of order $p-1$:
(i) $x^{p-1} \equiv 1 \bmod p$, and
(ii) $x^{p-1/q} \not\equiv 1 \bmod p$ for every prime factor q of $p-1$.
Now given p, guess x and verify (i) and (ii). (i) poses no problem. To check (ii), guess a prime factorization of $p-1$ and recursively check that all the factors q are prime and that (ii) holds. □

Since graphs can be used to model so many discrete problems, it should not

come as a surprise that numerous problems in complexity theory come from graph theory. Here are two examples.

The HAMILTON PATH problem is defined by:
INPUT: a graph G,
QUESTION: does G have a Hamilton path, i.e. a path that visits every node exactly once?
HAMILTON PATH is in NP: simply guess a path of length $|G|$ and check that it is a Hamilton path.

The traveling salesman problem TSP is defined by:
INPUT: a complete undirected weighted graph G, that is, every two nodes i and j have a *distance* $d_{i,j}$ (with $d_{i,j} = d_{j,i}$), and $k \in \mathbb{N}$,
QUESTION: is there a tour of length (i.e. weight) at most k ? (A *tour* is a path without repetitions visiting all nodes, starting and finishing in the same node. So this is the same as a Hamilton circuit, cf. Exercise 3.5.19.)[2]
The problem TSP is in NP: Guess a permutation of the nodes of G and check that it defines a tour of length at most k. Note that in contrast to this, Dijkstra's algorithm shows that the problem of finding shortest paths in graphs is in P, cf. Exercise 2.7.7.

2.6 The Immerman-Szelepcsényi Theorem

The following theorem was proven independently in 1987 by Immerman [21] and Szelepcsényi [36]. It says that the nondeterministic space classes are closed under complementation. Note that if $A \in \mathrm{NSPACE}(f)$, an application of Savitch's Theorem would only give that $\overline{A} \in \mathrm{NSPACE}(f^2)$.

Theorem 2.6.1. *Let f be a space constructible function such that $f(n) \geqslant \log n$. Then* $\mathrm{NSPACE}(f) = \text{co-NSPACE}(f)$.

Proof. Suppose that machine M accepts A in nondeterministic space $f(n)$. Note that on a given input x of length n there are $2^{f(n)}$ possible configurations. We define a nondeterministic machine $\overline{M}$ that accepts the complement $\overline{A}$, i.e. $x \notin A$ if and only if $\overline{M}$ has a path accepting x. The idea is to cycle through all possible configurations σ of M, for each one checking whether it is a final configuration that is reachable from the initial configuration by a computation of M. In case no such accepting configuration is found we know $x \notin A$ so we let $\overline{M}$ accept x. The problem is that to see if σ is reachable we can only guess a path of M.[3] If a wrong path was guessed we may falsely think that

[2]Note that the parameter k is used to phrase the problem as a *decision* problem. Formally, all our complexity classes are classes of sets, that can be thought of as coding decision problems with yes/no answers. One can also formulate complexity classes of *functions* rather than sets, but we will not do this here.

[3]Note that trying all paths is too expensive: with $2^{f(n)}$ possible configurations there are simply too many paths, so that a counter would be too long.

σ was unreachable. (Note that we will never think that an unreachable σ is reachable.) Hence, in doing so for every σ we may *underestimate* the number $n(x)$ of reachable σ's. If we *knew* the number $n(x)$ in advance, we could match the number of σ's found in the above procedure against it, and let $\overline{M}$ reject x in case the numbers did not match. In case the numbers matched we would know we had seen all reachable configurations, and if none of them were accepting we could safely declare $x \notin A$. If indeed $x \notin A$, for *some* path of $\overline{M}$ all the guessed paths for the σ's would be correct and we would correctly declare $x \notin A$.

So it remains to be shown that we can nondeterministically compute the number $n(x)$ of configurations σ that are reachable by M from the initial configuration. This is done by "inductive counting". Let $m(x,t)$ be the number of configurations that are reachable in at most t steps. We can deterministically compute $m(x,1)$. Suppose that we know $m(x,t)$. Configuration σ is reachable in at most $t+1$ steps if σ is initial or there is a τ reachable in at most t steps such that σ is reachable from τ in one more step. We compute $m(x,t+1)$ from $m(x,t)$ as follows. Cycle trough all configurations σ. For every σ, cycle through all possible τ, each time guessing a path to reach τ in at most t steps. If a τ is found to be reachable, and if σ is reachable from τ in one step, increase the counter $m(x,t+1)$ by one. If at the end the number of τ's found does not match $m(x,t)$, reject x (since we know a mistake was made). If there is a match with $m(x,t)$ we know that also $m(x,t+1)$ is correct. Finally, $n(x)$ is equal to $m(x,t)$ for the first t such that $m(x,t) = m(x,t+1)$. □

Theorem 2.6.1 has an important consequence for the theory of formal languages, namely that the class of languages that can be generated with a context-sensitive grammar is closed under complements. This had been an open problem for about 25 years.

Corollary 2.6.2. *The class of context-sensitive languages is closed under complements.*

Proof. It is a result of formal language theory that the class of context-sensitive languages is precisely the class of sets accepted by linearly bounded automata, that is, NSPACE(n), so the result follows from Theorem 2.6.1. □

Note that while Theorem 2.6.1 shows the closure of the nondeterministic space classes under complementation, the same result for the nondeterministic time classes remains one of the fundamental open problems in complexity theory. It seems unlikely that a similar result holds in this case, e.g. that NP = co-NP would hold.

2.7 Exercises

N.B. Exercises marked with * may be more challenging.

Exercise 2.7.1. Derive Savitch's Theorem from Theorem 2.2.1. (Hint: Given a nondeterministic machine we can apply Theorem 2.2.1 to the configuration graph of the machine on a given input by asking whether an accepting configuration is reachable.)

Exercise 2.7.2. The following problem is related to PATH. Suppose we are given the adjacency matrix A of a binary relation $\prec$ on n points. Show that it is possible to construct the adjacency matrix of the transitive closure of $\prec$ in polynomial time. (Hint: A^k contains the information whether from node i we can reach node j in k steps.)

Exercise 2.7.3.* In this exercise we prove that the class of regular sets equals SPACE(k), the class of sets recognizable in constant space (for any fixed constant k). By definition, the regular sets are recognizable by one-way Turing machines (i.e. tape heads can only move right) with a read-only input. Note that SPACE(k) = SPACE(1) by the tape compression theorem.

(a) Show that regular sets can be recognized using constant space. (Recall from page 6 that the space taken by the input does not count.)

(b) Given L, define an equivalence relation on Σ^* by $x \equiv_L y$ precisely when for all z, $xz \in L \leftrightarrow yz \in L$. Prove the Myhill-Nerode Theorem: L is regular if and only if $\equiv_L$ has only finitely many equivalence classes.

(c) (Shepherdson) Prove that two-way machines with a read-only input and working in constant space can be simulated by one-way machines. (Hint: For the equivalence defined in (b) we have $x \equiv_L y$ if and only if x and y induce the same mapping of states of M as follows. Define mappings f_x and g_x by $f_x(q) = p$ if whenever M enters x at the right in state q it leaves x at the left in state p. Similar for g_x but now from right to left. Acceptance of xz depends only on f_x and g_x and not on x, and there are only finitely many such functions since the number of states is finite.)

Exercise 2.7.4. In general the computation tree of a nondeterministic computation in NTIME(n^c) is k-splitting, where k is the number of states in the Turing machine. Argue that we may think of the tree as a *binary* tree, where every path has length $\lceil \log k \rceil n^c$.

Exercise 2.7.5. Deciding whether $\gcd(x, y) = 1$ is in P. (Hint: The Euclidean algorithm uses at most a logarithmic number of divisions $x = ny + r$, since each time the remainder r is $\leqslant \frac{1}{2}x$.)

Exercise 2.7.6. (a) TSP was defined using tours. What is the total number of tours in TSP?

(b)* Note that, despite the bound from (a), TSP is solvable in time $2^{O(n)}$.

Exercise 2.7.7. (Dijkstra) Given a weighted graph G and $x, y \in G$, show that the problem of finding a shortest path from x to y is in P. (If you like, you can first phrase this as a decision problem.)

Exercise 2.7.8. Show that NP is closed under Kleene star. That is, if A is in NP, then so is the set A^* consisting of strings obtained by concatenating a finite number of strings from A.

Exercise 2.7.9.* In this exercise we show that P is closed under Kleene star. This is much harder than the previous exercise!

(a) Given $L \in \mathrm{P}$ and a string w of length n, let $w[i, j]$ denote the substring from bit i to j. Define a binary relation $\prec$ on pairs (i, j) by $(i, j) \prec (i', j')$ if $i' = j + 1$ and $w[i, j]$ and $w[i', j']$ are both in L. Let $\prec^*$ denote the transitive closure of $\prec$. Show that $w \in L^*$ if and only if there exist j and k at most n such that $(1, j) \prec^* (k, n)$.

(b) Suppose that $L \in \mathrm{P}$. Show that $L^* \in \mathrm{P}$. (Hint: Use Exercise 2.7.2.)

Exercise 2.7.10. Is every subset of a set in NP automatically in NP ?

Exercise 2.7.11. Classify the following KNAPSACK problem: Given a set $\{1, \ldots, n\}$ of items such that item i has value v_i and weight w_i for every i. Given a knapsack that can hold a total weight of at most W, the problem is to find a subset $S \subseteq \{1, \ldots, n\}$ of maximal value that fits into the knapsack. Put more mathematically: maximize $\sum_{i \in S} v_i$ subject to the condition $\sum_{i \in S} w_i \leqslant W$.

Exercise 2.7.12. A function $g : \{0,1\}^* \to \{0,1\}^*$ is called *honest* if there is a polynomial p such that $p(|g(x)|) \geqslant |x|$, that is, g cannot shrink its input by more than a polynomial factor. This exercise indicates the interest in the notion of honesty for complexity theory.

(i) By definition, a set is *computably enumerable (c.e.)* if it is empty or the range of a computable function. Show that a set A is c.e. if and only if it is empty or the range of a polynomial time computable function g, i.e. $A = \{y : \exists x\ g(x) = y\}$.

(ii) Show that a set A is in NP if and only if it is empty or the range of an *honest* polynomial time computable function g.

Chapter 3

Reductions and completeness

One of the goals of complexity theory is to classify problems according to their complexity. The main tool for doing this is to consider effective reductions between problems. A key insight is that classes such as NP contain hardest problems. Reductions also allow us to substantiate the idea that various problems, though differently formulated, are actually the same.

3.1 Many-one reductions

Definition 3.1.1. Given sets A and B, A is many-one reducible or simply *m-reducible* to B, written $A \leqslant_m B$, if for some computable function f,

$$x \in A \iff f(x) \in B$$

for every x. If f is in addition polynomial time computable then we say that A is *p-m-reducible* to B, written $A \leqslant^p_m B$. We write $A \equiv^p_m B$ if both $A \leqslant^p_m B$ and $B \leqslant^p_m A$. The set $\{B : A \equiv^p_m B\}$ is called the *p-m-degree* of A.

The *join* $A \oplus B$ of two sets of strings A and B is defined as

$$A \oplus B = \{x\hat{}0 : x \in A\} \cup \{x\hat{}1 : x \in B\}.$$

Viewing strings as numbers, this is the same as putting A on the even numbers and B on the odd ones. The set $A \oplus B$ contains precisely all the information from A and B "zipped" together.

Proposition 3.1.2. (i) *$\leqslant^p_m$ is reflexive and transitive.*

(ii) *$A \leqslant^p_m B$ precisely when $\overline{A} \leqslant^p_m \overline{B}$.*

(iii) *If $A \in$ P then $A \leqslant^p_m B$ for every set B that is not empty or equal to $\{0,1\}^*$.*

(iv) *If both* $A \leqslant_m^p C$ *and* $B \leqslant_m^p C$ *then* $A \oplus B \leqslant_m^p C$. *Since clearly* A, $B \leqslant_m^p A \oplus B$ *this shows that on the p-m-degrees* $\oplus$ *gives the* least upper bound *of* A *and* B.

Proof. Exercise 3.5.1. □

The graph problem 3COLORING is defined by
INPUT: a graph G,
QUESTION: can G be colored with 3 colors, that is, does there exist a 3-coloring of the vertices such that no pair of adjacent vertices have the same color?

Proposition 3.1.3. 3COLORING $\leqslant_m^p$ SAT.

Proof. Suppose the given graph G has k vertices. Introduce variables $x_{i,j}$, $i = 1, \ldots, k$ and $j = 1, 2, 3$, meaning "vertex i has color j". We define an instance of SAT by the following set of clauses:

$$C(i) = \{x_{i,1}, x_{i,2}, x_{i,3}\}, \quad i = 1 \ldots k, \text{ "every vertex has a color"}$$
$$T(i) = \{\bar{x}_{i,1}, \bar{x}_{i,2}\}, \qquad i = 1 \ldots k, \text{ "}i\text{ not both color 1 and 2"}$$
$$U(i) = \{\bar{x}_{i,1}, \bar{x}_{i,3}\},$$
$$V(i) = \{\bar{x}_{i,2}, \bar{x}_{i,3}\}.$$

Together these clauses express that every vertex has exactly one color. Next, for every edge e from u to v and every color j define

$$D(e, j) = \{\bar{x}_{u,j}, \bar{x}_{v,j}\}$$

that is, u and v should not both have color j. We can write down these clauses in linear time, and the set of clauses is satisfiable precisely when G is 3-colorable. □

Proposition 3.1.3 is an example of the expressivity of SAT. In the following we show that SAT can express *all* NP-problems.

Definition 3.1.4. A set A is called *hard* for a class $\mathcal{C}$ if $C \leqslant_m^p A$ for every $C \in \mathcal{C}$. A is *complete* for $\mathcal{C}$ if in addition $A \in \mathcal{C}$.

Note that if A is hard and $A \leqslant_m^p B$ then also B is hard. Also note that all the classes P, NP, co-NP, PSPACE are downwards closed under $\leqslant_m^p$, that is, if $A \leqslant_m^p B$ and B is in the class then also A belongs to it.

The notion of completeness is one of the central notions of our subject. It allows us to study complexity classes by focussing on certain individual elements in them. Before we can do so, we first have to show that such complete elements exist.

3.2 NP-complete problems

One can easily define an NP-complete problem as follows. Let

$$K = \big\{\langle M, x, 1^t\rangle : M \text{ is a nondeterministic TM that accepts } x \text{ in } \leqslant t \text{ steps}\big\}.$$

It is debatable whether K is a "natural" NP-problem. Of course, K is natural in our given context of complexity theory. It is the direct analogue of the halting problem in computability theory. However, sometimes the term "natural" is reserved for problems that arose in some other context and were not defined directly using the notions at hand.

Theorem 3.2.1. *K is* NP*-complete.*

Proof. $K \in$ NP: Given input $\langle M, x, 1^t\rangle$, guess a computation path of M of length t and check whether it accepts x. This can be done in time $O(n^2)$ (cf. Theorem 2.1.1).

K is NP-complete: Let A be an arbitrary set in NP. We show that $A \leqslant_m^p K$. Let M be a nondeterministic machine accepting A in time $p(n)$. Let $f(x) = \langle M, x, 1^{p(|x|}\rangle$. Then $x \in A \Leftrightarrow f(x) \in K$. □

The following problem has perhaps a better claim to the label "natural". It is one of the most famous NP-complete problems, and it was also the first one.

Theorem 3.2.2. (Cook [11])[1] SAT *is* NP*-complete.*

Proof. Let $L(M)$ be a language in NP accepted by a nondeterministic machine M working in time p. Given x of length n, we can effectively define the following Boolean formulas. Let α and β denote vectors of Boolean variables of length $p(n)$.

1. Initial(α, x) expresses that α codes an initial configuration of M on input x.
2. Next(α, β) expresses that β is a configuration following α after one possible computation step of M.
3. Accept(α) expresses that α codes an accepting configuration.

Now combine these formulas to form

$$\text{Accepted}(x) = \text{Initial}(\alpha_1, x) \wedge \bigwedge_{i=1}^{p(n)-1} \text{Next}(\alpha_i, \alpha_{i+1}) \wedge \text{Accept}(\alpha_{p(n)}).$$

Note that we can write this formula down in polynomial time, and that it is satisfiable precisely when there are assignments α_i encoding a complete accepting computation of M on x. Hence $x \in L(M)$ if and only if Accepted$(x) \in$ SAT. □

[1] Because of similar work by Levin this is sometimes called the Cook-Levin theorem.

3.3 More decision problems from logic

We have already encountered the set SAT of all satisfiable Boolean formulas. Consider the following variants of SAT.

- SAT-CNF is the subset of SAT consisting of all satisfiable formulas that are in conjunctive normal form.
- nSAT is the subset of SAT-CNF where every clause has at most n literals.
- QBF is the set of satisfiable quantified Boolean formulas. The set of q.b.f.'s is defined by

  (i) 0,1,x (variable) are q.b.f.'s,

  (ii) the set of q.b.f.'s is closed under $\wedge$, $\vee$, $\neg$,

  (iii) if F is a q.b.f. then also $\exists xF$ and $\forall xF$ are q.b.f.'s.

  $\exists xF$ is *true* if F with $x = 0$ is true or F with $x = 1$ is true. $\forall xF$ is true if both F with $x = 0$ is true and F with $x = 1$ is true. As for SAT, a q.b.f. is satisfiable if there is an assignment of the free variables that makes the formula true. Note that it also makes sense to talk about the truth of *closed* q.b.f.'s, i.e. those without free variables. E.g. $\forall x(x \vee \neg x)$ is true.

Note that putting a Boolean formula in CNF using the distributive law

$$(\varphi \wedge \psi) \vee \chi \longleftrightarrow (\varphi \vee \chi) \wedge (\psi \vee \chi)$$

can be expensive: it may result in an exponential blow-up of the formula because of the repetition of χ on the right hand side. For QBF, however, we have the following result:

Proposition 3.3.1. *For every Boolean formula F of size m and containing variables $x_1 \dots x_m$ there is an equivalent q.b.f. $\exists y_1 \dots \exists y_k F'$ with variables $x_1 \dots x_m$, $y_1 \dots y_k$ such that F' is in CNF and of size at most $c \cdot m^2$, where c is a constant independent of F.*

Proof. Without loss of generality $\neg$ occurs only directly in front of variables in F. The proof is by formula induction on F:

The base case where F is a literal or a variable is clearly o.k.

$F = F_1 \wedge F_2$. By induction F_1 and F_2 have the required form. Now just put their $\exists$'s in front, after a possible renaming of variables.

$F = F_1 \vee F_2$. By induction $F_1 = \exists \vec{y} F_1'$ and $F_2 = \exists \vec{z} F_2'$. Introduce a new variable x to indicate whether F_1 or F_2 is true: Convert every clause C in F_1'

to $x \vee C$ and every C in F_2' to $\neg x \vee C$ and let F' be the conjunction of all these new clauses. Then $\exists x \exists \vec{y} \exists \vec{z} F'$ is equivalent to F. Note that the $\vee$-step adds a constant to the size of every clause, hence at most $c \cdot m$ to the formula. Since there are at most m such steps, the total size of the new formula is bounded by $c \cdot m^2$. □

Corollary 3.3.2. SAT $\leqslant_m^p$ SAT-CNF. *Hence* SAT-CNF *is* NP*-complete.*

Proof. Given F, Proposition 3.3.1 gives an equivalent q.b.f. $\exists \vec{y} F'$ with F' in CNF. Now just drop the $\exists$'s: $F \in$ SAT if and only if $F' \in$ SAT. □

The problem CLIQUE is defined by
INPUT: a graph G and $k \in \mathbb{N}$,
QUESTION: does G have a complete subgraph of size k ?

Theorem 3.3.3. CLIQUE *is* NP*-complete.*

Proof. That the problem is in NP is obvious: simply guess a subgraph of size k and check that it is complete. We show that SAT-CNF$\leqslant_m^p$ CLIQUE. Given a formula F with literals $x_1, \ldots, x_r$ and clauses $C_1, \ldots, C_s$, let

$$V = \big\{(x_i, C_j) : x_i \in C_j\big\}$$
$$E = \big\{\big((x_i, C_j), (x_m, C_n)\big) : j \neq n \wedge \bar{x}_i \neq x_m\big\}$$
$$k = s$$

We leave it as Exercise 3.5.12 to check that this is a p-m-reduction. □

Theorem 3.3.4. 3SAT *is* NP*-complete.*

Proof. We show that SAT-CNF$\leqslant_m^p$ 3SAT, so the result follows from Corollary 3.3.2. Given an instance of SAT-CNF, first note that an assigment satisfies a clause $\big\{x_1, x_2, \ldots, x_k\big\}$ if and only if the same assignment satisfies the conjunction of $\big\{x_1, x_2, z\big\}$ and $\big\{\bar{z}, x_3, \ldots, x_k\big\}$ for a suitable value of z. By iterating this, we replace every clause $\big\{x_1, x_2, \ldots, x_k\big\}$ with more than 3 literals by the $k-2$ new clauses

$$\big\{x_1, x_2, z_1\big\}, \big\{x_3, \bar{z}_1, z_2\big\}, \big\{x_4, \bar{z}_2, z_3\big\}, \ldots, \big\{x_{k-2}, \bar{z}_{k-2}, z_{k-3}\big\}, \big\{x_{k-1}, x_k, \bar{z}_{k-3}\big\}.$$

Then the formula obtained in this way is equisatisfiable with the original one. □

In contrast to Theorem 3.3.4 we have

Theorem 3.3.5. 2SAT *is in* P.

Proof. Exercise 3.5.13. □

The proof of the following theorem has a certain similarity with the proof of Savitch's theorem.

Theorem 3.3.6. (Meyer and Stockmeyer [22]) QBF *is* PSPACE-*complete.*

Proof. First note that QBF $\in$ SPACE(n): try all (linear size) assignments, using a (linear sized) counter.

For the completeness, let M be a given machine with space bound p. A computation of M on an input of size n has at most $2^{p(n)}$ configurations, and is accepting if an accepting configuration is reachable in at most $2^{p(n)}$ steps from the initial configuration. So the theorem follows if we can efficiently construct a q.b.f. $\text{Access}_{2^m}(\alpha, \beta)$ expressing that α and β code configurations of M and β is reachable from α in $\leqslant 2^m$ steps. We already saw in the proof of Theorem 3.2.2 that there are Boolean formulas expressing that α is an (initial or accepting) configuration of M.

$m = 0$: We can use the formula $\text{Next}(\alpha, \beta)$ from the proof of Theorem 3.2.2 to express that β is reachable from α in at most one step.

$m > 0$: Given $\text{Access}_{2^{m-1}}(\alpha, \beta)$, a bad attempt would be to write

$$\text{Access}_{2^m}(\alpha, \beta) = \exists\gamma\big(\text{Access}_{2^{m-1}}(\alpha, \gamma) \wedge \text{Access}_{2^{m-1}}(\gamma, \beta)\big)$$

because then the length of the formula doubles with every step. Instead write

$$\text{Access}_{2^m}(\alpha, \beta) = \exists\gamma\forall\alpha', \beta'\big((\alpha' = \alpha \wedge \beta' = \gamma) \vee (\alpha' = \gamma \wedge \beta' = \beta) \rightarrow \text{Access}_{2^{m-1}}(\alpha', \beta')\big).$$

Note that the extra quantification is used so that we have to write the previous formula $\text{Access}_{2^{m-1}}$ only once instead of twice. This makes the recursion a linear affair. □

3.4 Completeness of Hamilton path and TSP

We can view reductions as a translation from the language of the easier problem to that of the harder one. As we have seen, these reductions are a useful tool in proving that certain problems are complete, using problems that we already know to be complete. The reductions we have seen so far were rather easy to describe. In this section we give an example of a more elaborate reduction, showing completeness of HAMILTON PATH and TSP.

Theorem 3.4.1. (Karp [23]) HAMILTON PATH *is* NP-*complete.*

Proof. We follow the proof in [29]. That the problem is in NP was already discussed on page 12. We prove that 3SAT $\leqslant_m^p$ HAMILTON PATH, so that the completeness follows from Theorem 3.3.4. Suppose that φ is an instance

Figure 3.1: The truth module

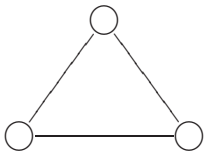


Figure 3.2: The clause module

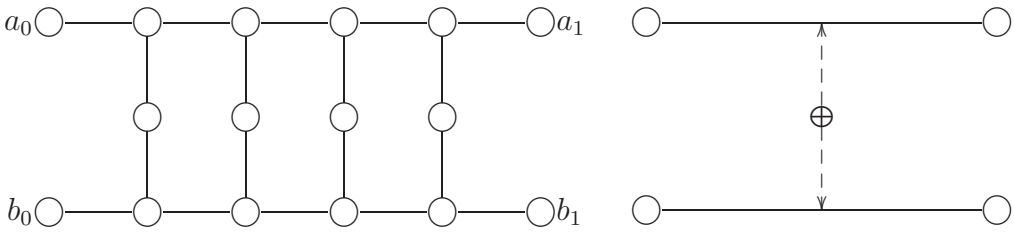


Figure 3.3: The XOR module

of 3SAT, with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$. Without loss of generality, every clause contains exactly three variables (see Exercise 3.5.17). We construct an instance $f(\varphi)$ of HAMILTON PATH by piecing together some basic modules. Truth or falsity of a variable is coded by the truth module from Figure 3.1. Any Hamilton path has to choose between the left and the right edge. We let the left one stand for "true" and the right one for "false". We represent the clauses by triangles, as in Figure 3.2, where each side of the triangle represents a literal from the clause. We will arrange things such that only sides for which the corresponding literal is false have to be traversed. Note that by definition no Hamilton path can traverse all sides of a triangle. Hence if all literals are false there will be no Hamilton path.

Next consider the module on the left hand side of Figure 3.3. Any Hamilton path that does not start or end in a node of this module has only two possible ways of routing through it: Either entering in a_0 and exiting in a_1 or entering in b_0 and exiting in b_1. (Cf. Exercise 3.5.18.) Thus the module acts as a kind of XOR gate forcing a choice between two "edges", viz. the upper and the lower level. We abbreviate this module by the picture on the right of Figure 3.3. It has the property that in each Hamilton path, one of the edges is traversed, and the other is not.

We put all the pieces together as follows. The graph $f(\varphi)$ has n truth modules, one for each variable $x_1, \ldots, x_n$, connected in series, as on the left

side of Figure 3.4. Call the first node 1 and the last one 2. It also has m triangles, one for each clause. Every side of a triangle represents a literal in the clause. If the literal is x_i, we connect the side with a XOR module to the "true" side of the truth module of x_i, and if the literal is $\neg x_i$ we connect it to the "false" side. Also, we add an extra node 3. We pairwise connect all $3m$ nodes of the triangles plus nodes 2 and 3. Finally, we add a node 4 only connected to node 3. An example of the final graph thus obtained is pictured in Figure 3.4.

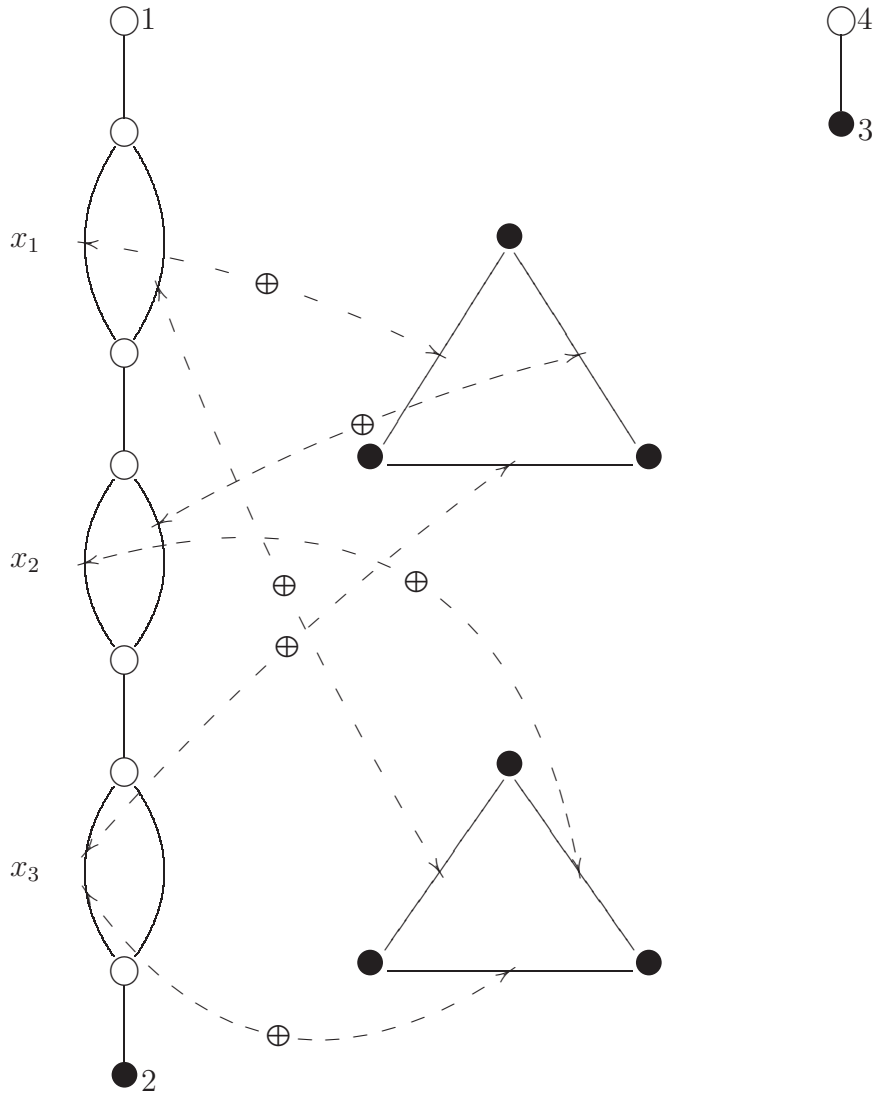


Figure 3.4: Graph for the formula $\big(x_1 \vee \neg x_2 \vee x_3\big) \wedge \big(\neg x_1 \vee x_2 \vee x_3\big)$. The upper triangle is for the first clause, the lower for the second. Also, every pair of black dots is connected.

We verify that the construction works. Suppose that the graph $f(\varphi)$

possesses a Hamilton path. Then its two ends must be 1 and 4. Without loss of generality suppose it starts in 1. The only way to get to node 2 is to traverse the series of truth modules. This defines a valuation v of the variables x_i. We claim that v satisfies φ. The use of the XOR module ensures that a side of a triangle is not traversed if and only if the corresponding literal is true under v. Since no Hamilton path can traverse all edges of a triangle, every triangle has an untraversed edge, which means that v makes the corresponding literal true. Hence φ is satisfied.

Conversely, suppose that there is a valuation v satisfying φ. Then $f(\varphi)$ has a Hamilton path as follows. First traverse the truth modules according to v, choosing left if $v(x_i) = 1$ and right if $v(x_i) = 0$. Since every clause has a literal that is true under v, for no triangle we have to traverse all edges. Since all edges between the nodes of the triangles and nodes 2 and 3 are present, we easily see from this that we can reach 4 visiting all nodes exactly once. □

Corollary 3.4.2. TSP *is* NP*-complete.*

Proof. We show that HAMILTON PATH $\leqslant_m^p$ TSP. Given a graph G with n nodes, construct an instance $f(G)$ of TSP by letting the cities be the nodes of G. Let the distance between two cities be 1 if the nodes are connected in G, and 2 otherwise. Then G has a Hamilton path if and only if there is a tour of length at most $(n-1) \cdot 1 + 2 = n + 1$. □

3.5 Exercises

Exercise 3.5.1. Prove Proposition 3.1.2.

Exercise 3.5.2. (a) Show that for every L in PSPACE there is an L' in SPACE(n) such that L p-m-reduces to L'. (Hint: define L' by prefixing the elements of L with strings of polynomial length. This technique is called *padding*.)

(b) Suppose that $\mathcal{C}$ is closed downwards under p-m-reducibility and that SPACE(n) $\subseteq \mathcal{C} \subseteq$ PSPACE. Show that $\mathcal{C} =$ PSPACE.

(c) Show that P $\neq$ SPACE(n) and that NP $\neq$ SPACE(n). (Hint: use (b) and the space hierarchy theorem.)

Exercise 3.5.3. Use the time hierarchy theorem to show that TIME(n^k) is not closed downwards under $\leqslant_m^p$.

Exercise 3.5.4. Define analogous versions of the set K that are p-m-complete for co-NP, PSPACE, and EXP.

Exercise 3.5.5. Think about the details of Theorem 3.2.2.

Exercise 3.5.6. Show that P = NP if and only if SAT $\in$ P.

Exercise 3.5.7. In a 1956 letter to von Neumann (reprinted in [31]) Gödel mentioned the following problem: Given a predicate formula φ and $n \in \mathbb{N}$, does φ have a proof (in a fixed proof system for predicate logic) of length at most n ? Gödel explicitly mentions the possibility that the problem could be solvable in quadratic time, as opposed to the exponential bound resulting from brute force search. Show that the problem is in NP, and that it is NP-complete by reducing SAT to it.

Exercise 3.5.8. A set A is called *self-reducible* if $A \leqslant_m^p A$ via a reduction f such that $f(x) \neq x$ for all x. Show that SAT is self-reducible.

Exercise 3.5.9. There are two exponential time classes in the literature:

$$\mathrm{E} = \bigcup_c \mathrm{TIME}(2^{cn})$$
$$\mathrm{EXP} = \bigcup_c \mathrm{TIME}(2^{n^c}).$$

These are often abbreviated to $\mathrm{E} = 2^{\mathrm{linear}}$ and $\mathrm{EXP} = 2^{\mathrm{polynomial}}$. Show that EXP is equal to the downward closure of E under $\leqslant_m^p$.

Exercise 3.5.10. Let $\mathrm{LIN} = \mathrm{TIME}(n)$, and let $\mathrm{NLIN} = \mathrm{NTIME}(n)$. It is known that LIN $\neq$ NLIN, see e.g. vol. II of [4] for a proof. In this exercise we prove that P $\neq$ NLIN.

(i) If NLIN $\subseteq$ P then P = NP.

(ii) If P $\subseteq$ NLIN then P $\neq$ NP.

(iii) P $\neq$ NLIN.

Exercise 3.5.11. Positive reductions are a generalization of m-reductions defined as follows: $A \leqslant_{\mathrm{pos}}^p B$ if $A = M^B$ for a deterministic Turing machine M working in polynomial time such that

$$C \subseteq D \Longrightarrow M^C \subseteq M^D.$$

Show that NP is downwards closed under $\leqslant_{\mathrm{pos}}^p$.

Exercise 3.5.12. Finish the proof of Theorem 3.3.3.

Exercise 3.5.13. Prove Theorem 3.3.5, namely that 2SAT is in P. (Hint: Given φ, draw a directed graph of all literals, with an arrow from x to y if $\{\bar{x}, y\}$ is a clause of φ. Argue that φ is satisfiable if and only if there is no literal x for which there is both a path from x to $\bar{x}$ and from $\bar{x}$ to x. Then use Exercise 2.7.2.)

Exercise 3.5.14. Let kDNF be the set of all satisfiable formulas in disjunctive normal form (i.e. written as a disjunction of conjunctions) where every conjunct has at most k literals. Show that kDNF is in P for every k.

Exercise 3.5.15. HORNSAT is the subset of SAT-CNF consisting of the satisfiable Horn formulas, i.e. those where every clause contains at most one positive literal. Show that HORNSAT is in P. (Hint: Given φ, define the minimal truth assignment T satisfying φ. Hence φ is satisfiable if and only if T satisfies φ.)

Exercise 3.5.16. Show that one can define the logical constant $\bot$ (the formula that is always false) with an nSAT-formula in which every clause contains exactly n variables. Namely, show that there is an nSAT-formula φ in which every clause contains exactly the variables $x_1, \ldots, x_n$ such that $\varphi(x_1, \ldots, x_n)$ holds precisely if $\bar{x}_1 = \ldots = \bar{x}_n = \bot$.

Exercise 3.5.17. Consider the following variants of nSAT.
nSAT : every clause has at most n literals.
nSAT$'$: every clause has exactly n variables.
nSAT$''$: every clause has exactly n literals.
Show that $n\text{SAT} \equiv^p_m n\text{SAT}' \equiv^p_m n\text{SAT}''$. (Note that by definition clauses are *sets*, so that repeating a literal in a clause makes no difference to the number of elements in it. Instead, use Exercise 3.5.16.)

Exercise 3.5.18. Check the claim from Theorem 3.4.1 that there are only two possible ways for a Hamilton path to traverse the module of Figure 3.3, given that the path enters in either a_0 or b_0 and the internal nodes of the module (i.e. those different from the end points of the module) are not connected to any nodes outside the module.

Exercise 3.5.19. Define the problem HAMILTON CIRCUIT by:
INPUT: a graph G,
QUESTION: does G have a cycle, starting and returning in the same node and visiting every other node exactly once?
Define the variant H by:
INPUT: graph G and two nodes $x_0, x_1 \in G$,
QUESTION: does G have a Hamilton path starting in x_0 and ending in x_1?
Show that the problems HAMILTON PATH, HAMILTON CIRCUIT, and H all have the same p-m-degree.

Chapter 4

Relativized computation and the polynomial hierarchy

4.1 Relativized computation

One of the central notions in computability theory is Turing's notion of relative computability, defined using oracle Turing machines. Such a machine has an extra tape on which the (infinitely many) bits of a oracle set can be written, and that can be used during the computation. The information coded by this oracle set can be thought of as given for free, and the sets computable in this fashion as being computable *relative* to the oracle set. The resource bounded version of this notion is defined as follows. The classical formulation with oracle tapes is not entirely appropriate in this setting, since running up and down the oracle tape takes too long. Instead, we extend our Turing machines with a query tape and query states. The machine can write queries to the oracle on the query tape, and when the machine is in a query state the answer to the query is received instantaneously, so that the queries to the oracle count as a single computation step. Given an oracle machine M, let M^B denote the function computed by M using oracle B. Then a set A *Turing reduces* to B in polynomial time, denoted $A \leqslant_T^p B$, if $A = M^B$ for some machine M working in polynomial time.

Note that we have that

$$A \leqslant_m^p B \Longrightarrow A \leqslant_T^p B.$$

Turing reducibility is much more liberal than many-one reducibility: To answer questions of the form "$x \in A$?" we not only can query the oracle B more than once, but the queries can also be *adaptive*, i.e. depend on the answer to previous queries. By diagonalization, one can show that $\leqslant_T^p$ differs from $\leqslant_m^p$ on the computable sets, or even EXP (see Theorem 5.4.1). It is open if the two notions are different on NP. Of course, if P = NP they are the same on

NP. By Exercise 4.4.1, NP is closed under $\leqslant_T^p$ if and only if NP = co-NP. Hence if NP $\neq$ co-NP then $\leqslant_T^p$ and $\leqslant_m^p$ differ on NP.

The notion of reducibility $\leqslant_T^p$ satisfies the same properties as those of $\leqslant_m^p$ listed in Proposition 3.1.2, except that now we have in addition

$$A \leqslant_T^p B \Longrightarrow \overline{A} \leqslant_T^p B.$$

Again, $A \oplus B$ is a least upper bound operator. We also have an equivalence relation $\equiv_T^p$ induced by $\leqslant_T^p$, and $\{B : A \equiv_T^p \}$ is called the *p-T-degree* of A.

We employ the following notation in the presence of oracles:

- $\mathrm{P}^A = \mathrm{P}(A) = \{B : B \leqslant_T^p A\}$,
- $\mathrm{NP}^A = \mathrm{NP}(A) = \{B : B = M^A \text{ for some nondeterministic } M\}$,
- $\mathrm{P}^{\mathcal{C}} = \mathrm{P}(\mathcal{C}) = \bigcup \{\mathrm{P}^C : C \in \mathcal{C}\}$.

Proposition 4.1.1. $\mathrm{NP}(\mathrm{P}^{\mathcal{C}}) = \mathrm{NP}(\mathcal{C})$.

Proof. ($\supseteq$) Obvious since $\mathcal{C} \subseteq \mathrm{P}^{\mathcal{C}}$.
($\subseteq$) Suppose $B = M_0^A$, where M_0 is a nondeterministic machine, and $A = M_1^C$ for a $C \in \mathcal{C}$ and a deterministic machine M_1. Nondeterministically compute B as follows: guess a computation path of M_0. For every query to A, insert the equivalent computation of M_1^C. The resulting path still has polynomial length. □

4.2 The Polynomial Hierarchy

We define a hierarchy, the *polynomial time hierarchy*, or simply the *polynomial hierarchy*, that is the direct analogue of Kleene's arithmetical hierarchy from computability theory. In the analogy P corresponds to the computable sets and NP to the computably enumerable sets. The analogy only goes so far. For starters, we do not know whether P $\neq$ NP. Also, a set A is computable precisely if both A and its complement $\overline{A}$ are c.e. In contrast with this, it is believed that NP $\cap$ co-NP $\neq$ P.

Definition 4.2.1. (The Polynomial Hierarchy) We inductively define a hierarchy as follows.

- $\Sigma_0^p = \Pi_0^p = \mathrm{P}$.
- $\Sigma_{n+1}^p = \mathrm{NP}(\Sigma_n^p)$.
- $\Pi_{n+1}^p = \text{co-}\Sigma_{n+1}^p$.
- $\Delta_{n+1}^p = \mathrm{P}(\Sigma_n^p)$.

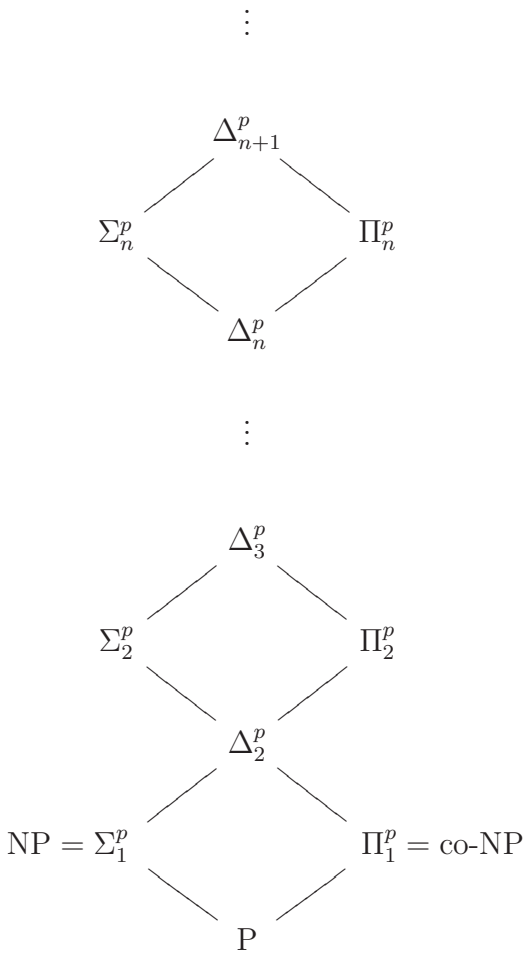


Figure 4.1: The polynomial hierarchy

- $\mathrm{PH} = \bigcup_{n \geqslant 0} \Sigma^p_n$.

The inclusions $\Sigma^p_n \subseteq \Delta^p_{n+1} \subseteq \Sigma^p_{n+1}$ and $\Pi^p_n \subseteq \Delta^p_{n+1} \subseteq \Pi^p_{n+1}$ are immediate from the definition. Figure 4.1 is a picture of the levels of this hierarchy and their inclusions.

Proposition 4.2.2. (i) $\Delta^p_1 = \mathrm{P}$.

(ii) $\Pi^p_{n+1} = \text{co-NP}(\Sigma^p_n)$.

(iii) $\Sigma^p_{n+1} = \mathrm{NP}(\Pi^p_n)$.

(iv) $\Delta^p_{n+1} = \mathrm{P}(\Pi^p_n)$.

(v) $\Sigma^p_{n+1} = \mathrm{NP}(\Delta^p_{n+1})$.

Proof. Exercise 4.4.5. □

The levels of the arithmetical hierarchy can be defined using quantification over previous levels. This is also true for the polynomial time hierarchy. The analogue of classical quantification is *bounded* quantification.

Definition 4.2.3. $\exists^{p(n)} x\, R(x)$ denotes that there is a string x with $|x| \leqslant p(n)$ with property R. If no confusion can arise we will simply write $\exists^p$. $\forall^{p(n)} x\, R(x)$ of course means that for all x with $|x| \leqslant p(n)$ property R holds. For a given class $\mathcal{C}$, $\exists\mathcal{C}$ denotes the class of sets of the form

$$\left\{x : \exists^{p(|x|)} y\ (x, y) \in B\right\}$$

where p can be any polynomial and $B \in \mathcal{C}$. The class $\forall\mathcal{C}$ is defined analogously.

Proposition 4.2.4. (i) $\exists \mathrm{P} = \mathrm{NP}$, $\forall \mathrm{P} = \text{co-NP}$.

(ii) $\exists\Sigma^p_n = \Sigma^p_n$, $\forall\Pi^p_n = \Pi^p_n$, *for* $n > 0$.

(iii) $\exists\Pi^p_n = \Sigma^p_{n+1}$, $\forall\Sigma^p_n = \Pi^p_{n+1}$.

Proof. Exercise 4.4.6. □

It follows from Proposition 4.2.4 that *a set is in* Σ^p_n *precisely if it can be defined by a formula with* n *quantifier alternations, starting with* $\exists$. Similarly for Π^p_n, but now the first quantifier is $\forall$.

In contrast to the situation in computability theory, we do not know whether the polynomial time hierarchy is proper, since we do not even know whether P = NP. Of course, when P = NP then the whole hierarchy collapses to P, as we invite the reader to check. The conjecture is that PH does not collapse, i.e. that $\Sigma^p_n \neq \Sigma^p_{n+1}$ for all n. However, it is also possible that P $\neq$ NP and that the hierarchy collapses to some higher level. Whatever is in the hierarchy, it takes place within PSPACE:

Proposition 4.2.5. PH $\subseteq$ PSPACE.

Proof. The proof is by induction on the levels Σ^p_n of PH.
$n = 0$: By definition, $\Sigma^p_0 = \mathrm{P} \subseteq \mathrm{PSPACE}$.
$n+1$: By induction hypothesis, $\Sigma^p_{n+1} = \mathrm{NP}(\Sigma^p_n)$ is included in NP(PSPACE).
But NP(PSPACE) $\subseteq$ PSPACE(PSPACE) = PSPACE. □

By Proposition 4.2.5 we see that the statement PH = PSPACE expresses that PH is as large as possible. Amusingly, this implies its collapse, i.e. if PH is large then it is already large at one of its finite levels:

Proposition 4.2.6. *If* PH = PSPACE *then* PH *collapses.*

Proof. Consider the PSPACE-complete set QBF from Theorem 3.3.6. If PH = PSPACE then QBF $\in \Sigma^p_n$ for some level n. By completeness of QBF and $\Sigma^p_{n+1} \subseteq$ PSPACE this implies that all sets Σ^p_{n+1} reduce to a set in Σ^p_n. Since Σ^p_n is downwards closed under p-m-reductions we obtain $\Sigma^p_{n+1} \subseteq \Sigma^p_n$. □

There is a whole array of intermediate notions between the two extremes of p-m-reducibility and p-T-reducibility. There are situations in which $\leqslant^p_m$ is too restrictive, and $\leqslant^p_T$ too liberal. Useful variations such as 1-1, truth-table, and bounded truth-table reducibility can be found in several of the standard textbooks.

4.3 Relativization

We have seen that the classes P and NP have relativized versions P^A and NP^A for any oracle A. In general we can relativize any class $\mathcal{C}$ that is defined in terms of Turing machine computations, simply by writing A on the oracle tape. Given an oracle A, the relativized class is denoted by $\mathcal{C}^A$.

Often a result about complexity classes holds in fact for all the relativized versions of the classes as well. In this case we say that *the result relativizes*. For example, the strict inclusion $\mathrm{P} \subsetneq \mathrm{EXP}$ relativizes, i.e. $\mathrm{P}^A \subsetneq \mathrm{EXP}^A$ for any A. In fact, we can obtain this result by using the *same* proof as before, just adding the oracle A in the appropriate places. In this case we say that *the proof relativizes*.

Sometimes we need to be more specific about what we mean by relativization. For example, Theorem 3.2.1 relativizes to: K^A is NP^A-complete for every A. But what do we mean by NP^A-complete? It could be taken to mean complete under the unrelativized $\leqslant^p_m$, or complete under m-reductions computed by functions in P^A. In this case the stronger version, using unrelativized reductions, holds, but in general one has to be specific about this.

We may not know whether $\mathrm{P} \neq \mathrm{NP}$, but we know something about a possible proof: Whether P = NP or $\mathrm{P} \neq \mathrm{NP}$, this *cannot be proven using a proof that relativizes*. This will follow from the results in section 5.3.

4.4 Exercises

Exercise 4.4.1. NP is closed under $\leqslant^p_T$ if and only if NP = co-NP.

Exercise 4.4.2. Show that a class $\mathcal{C}$ is countable if and only if there is a set A that is p-T-hard for $\mathcal{C}$.

Exercise 4.4.3.* Show that the class COMP of all computable sets does not have p-m-complete sets. Show the same result for p-T-completeness. (You can either prove this by direct diagonalization, or by using the fact that there is no universal computable set.)

Exercise 4.4.4. Explain why we did not define a polynomial space hierarchy.

Exercise 4.4.5. Prove Proposition 4.2.2.

Exercise 4.4.6. Prove Proposition 4.2.4. (Hint: Item (ii) follows by taking two quantifiers of the same kind together using the pairing function $\langle \cdot\,, \cdot \rangle$. For (iii), prove $\Sigma^p_{n+1} \subseteq \exists\Pi^p_n$ by induction. Separate the positive and the negative queries.)

Exercise 4.4.7. Let VALID be the set of all valid propositional formulas.

(i) Show that $\overline{\text{SAT}} \equiv^p_m \text{VALID}$.

(ii) For which level of PH is VALID complete?

Exercise 4.4.8. Prove the following:

- If $\Sigma^p_i = \Pi^p_i$ then $\text{PH} = \Sigma^p_i = \Pi^p_i$.
- If $\Sigma^p_i = \Sigma^p_{i+1}$ then $\text{PH} = \Sigma^p_i$.
- If $\Pi^p_i = \Pi^p_{i+1}$ then $\text{PH} = \Pi^p_i$.

Exercise 4.4.9. Prove that $\Sigma^p_i(\text{NP}) = \Sigma^p_{i+1}$ for every i.

Exercise 4.4.10. Show that the following results relativize:

(i) All of section 2.2,

(ii) All of section 2.3.

Exercise 4.4.11. Show that K^A is complete for NP^A under unrelativized p-m-reductions.

Exercise 4.4.12. Show that

(i) $\text{P}^{\text{P}} = \text{P}$,

(ii) $\text{EXP}^{\text{EXP}} \neq \text{EXP}$.

(iii) $\text{PSPACE}^{\text{PSPACE}} = \text{PSPACE}$.

(iv) $\exists\text{PSPACE} = \text{PSPACE}$.

Exercise 4.4.13. Let A be an NP-complete set. Show that

(i) $\Sigma^p_1 = \{B \mid B \leqslant^p_m A\}$,

(ii) $\Delta^p_2 = \{B \mid B \leqslant^p_T A\}$.

Chapter 5

Diagonalization

Diagonalization is a technique for constructing sets in infinitely many stages. The name derives from Cantors famous diagonal argument showing that the reals are uncountable. Since Cantor the method has been extended and refined in numerous ways in the field of mathematical logic. In this chapter we will see some incarnations of this method in the resource bounded setting. Note that we have already made use of the method in the proofs of the hierarchy theorems in section 2.3.

5.1 The Halting Problem

As a warm-up, let us consider the proof that the halting problem is undecidable. Let $\{M_e\}_{e\in\mathbb{N}}$ be an effective enumeration of all Turing machines. The halting set is defined as

$$H = \big\{ \langle e, x\rangle : M_e(x)\downarrow \big\}.$$

Now suppose for a contradiction that H is computable. Then we can define a machine M such that for all e, $M(e)\downarrow$ if and only if $M_e(e)\uparrow$. Now let d be a code of M, i.e. $M = M_d$. Then we have $M_d(d)\downarrow$ if and only if $M_d(d)\uparrow$. From this contradiction we conclude that H must be noncomputable.

5.2 Intermediate sets

Are there any sets in NP that are neither in P nor NP-complete? Call such sets of *intermediate* degree. Of course, in case P = NP then the class of NP-complete sets coincides with P, so then there are no intermediate sets. A priori it could be possible that P $\neq$ NP and that still there are no intermediate sets. In that case NP would split into the sets in P and the NP-complete sets. The next result shows that this cannot happen. The proof uses a form

of the diagonalization method introduced by Ladner, called *delayed diagonalization*, also known as the *looking back technique* or *wait-and-see arguments*. The result itself is the direct analogue of the Friedberg-Muchnik Theorem in Computability Theory, which states that there exist c.e. sets of intermediate Turing-degree [28].

Theorem 5.2.1. (Ladner [25]) *If* P $\neq$ NP *then there is a set* $B \in$ NP *that is neither in* P *nor* NP*-complete.*

Proof. We will in fact ensure that B is not p-T-complete, so that in particular it is not p-m-complete. Fix a set A that is p-m-complete for NP, and suppose that $A \notin$ P (i.e. that P $\neq$ NP). Fix a polynomial time enumeration $\{M_e\}_{e\in\mathbb{N}}$ of all polynomial time machines. We construct B in infinitely many stages, while trying to satisfy for every e the requirements

$$R_{2e}: \qquad \exists y\big(A(y) \neq M_e^B(y)\big),$$

$$R_{2e+1}: \qquad \exists y\big(B(y) \neq M_e(y)\big).$$

Note that the even requirements ensure that B is not complete, and the odd requirements that B is not in P. Call y as in R_d a *witness* for the requirement. The problem is that to see if y is a witness for R_{2e} we need to compute A, which is expensive. (Even computing $M_e(y)$ is expensive, since the polynomial varies with e.) We know however that we will succeed if we let B look like a finite set for long enough. If we wait until we are dealing with larger inputs (and hence we have more computing time) we can simply look back and see a point where we succeeded. We deal with R_{2e+1} in a similar way, now letting B look like A to keep it out of P.

At stage x we have defined B up to input x. We define $B(x)$ as follows. First, let s be the largest number such that, by recursively running the construction, we can compute $B\restriction s$ in $|x|$ steps. Next we use $|x|$ computation steps to compute for as many $y = 0, 1, 2, \ldots$ as we can whether y is a witness for some R_d with $d < x$ at stage s. (We say that y is a witness for R_{2e} at stage s if $A(y) \neq M_e^{B\restriction s}(y)$, and the computation uses only numbers smaller than s.) We do this by dovetailing all pairs $\langle y, d\rangle$ in their natural order. Note that since A is hard to compute, this may not get us very far, but that is o.k.[1] Let d be the least number such that no witness for R_d is found. If $d = 2e$ is even that means we are still working on R_{2e}, so we let $B(x) = 0$. If $d = 2e+1$ is odd that means we are still working on R_{2e+1}, so we let $B(x) = A(x)$. This ends the construction of B.

Clearly $B \in$ NP by construction. We verify that the construction succeeds in satisfying all requirements. Suppose for a contradiction that some

[1] This is reminiscent of Sterne's Tristram Shandy, who needs a full year to describe just one day of his life [35, Vol. 2, Chapter XIII]. Slow as this may be, every day would eventually be described, if he were to live forever.

requirement is not satisfied, and let R_d be the least such. Note that if a witness for R_d is never found, it does not exist. If $d = 2e$ is even then $B(x) = 0$ for almost every x, and hence, because $A(y) = M_e^B(y)$ for every y, if follows that $A \in \mathrm{P}$, contrary to assumption. If $d = 2e + 1$ is odd then $B(x) = A(x)$ for almost every x, and $B(y) = M_e(y)$ for every y, hence we have again that $A \in \mathrm{P}$, contrary to assumption. □

Since we do not know that $\mathrm{P} \neq \mathrm{NP}$, of course we cannot prove at the moment of any particular set A in NP that it has intermediate degree. However, there are natural candidates for such sets. The most famous one is the problem GRAPH ISOMORPHISM defined by:
INPUT: two graphs G_0 and G_1,
QUESTION: are G_0 and G_1 isomorphic?
Obviously, this problem is in NP (guess an isomorphism). Up to now it has resisted any attempt to prove that it is in P or that it is NP-complete. Further evidence for the fact that this problem is not NP-complete is the fact (cf. Schöning [33]) that the set A representing GRAPH ISOMORPHISM is low_2, meaning that $\Sigma_2^{p,A} = \Sigma_2^p$. Hence if A were NP-complete we would have $\Sigma_3^p = \Sigma_2^p$, hence the polynomial time hierarchy would collapse to the second level (cf. Exercise 4.4.9).

According to Theorem 5.2.1 there are at least three p-m-degrees inside NP, provided of course that $\mathrm{P} \neq \mathrm{NP}$. From Exercise 5.7.1 we may conclude that in this case there are in fact infinitely many p-m-degrees inside NP. Moreover, by Exercise 5.7.3 these degrees are not linearly ordered. Much more is known about this world of intermediate degrees that falls outside of the scope of these notes. In a precise sense, the structure of intermediate degrees is as complicated as possible.

5.3 Oracle separations

A central theme in computability theory is the study of relativized computation, that is, the way sets behave in the presence of oracles. We will use the following basic fact about oracle computations.

Proposition 5.3.1. (Use Principle) *Suppose that B is such that for every query q in the oracle computation $M^A(x)$ we have $B(q) = A(q)$. Then $M^A(x) = M^B(x)$.*

Proof. The oracle B gives the same answers as oracle A, hence the computations are the same. □

Theorem 5.3.2. (Baker, Gill, and Solovay [3]) *There exists a computable set A such that $\mathrm{P}^A = \mathrm{NP}^A$.*

Proof. Let A be any set that is p-m-complete for PSPACE, such as QBF. Then we have

$$\mathrm{P}^A \subseteq \mathrm{NP}^A \subseteq \mathrm{PSPACE}^A \subseteq \mathrm{PSPACE} \subseteq \mathrm{P}^A.$$

□

Theorem 5.3.3. (Baker, Gill, and Solovay [3]) *There exists a computable set A such that $\mathrm{P}^A \neq \mathrm{NP}^A$.*

Proof. For any set A the set

$$B = \left\{0^n : \exists x \in \{0,1\}^n (x \in A)\right\}$$

is in NP^A. We want to construct A such that $B \notin \mathrm{P}^A$. As in a typical diagonalization construction, we split this global requirement into infinitely many subrequirements R_e. We construct A in stages s. At stage s we have defined a finite part A_s of the set $A = \bigcup_{s\in\mathbb{N}} A_s$. Let M_e be an enumeration of all polynomial time p-T-reductions, and suppose that M_e works in time p_e. We want to satisfy for every e the requirement

$$R_e : \qquad M_e^A \neq B.$$

Requirement R_e guarantees that the eth p-T-reduction to A does not compute B, hence all the requirements together guarantee that B is not in P^A. We satisfy R_e by ensuring that for some n, $M_e^A(0^n) \neq B(0^n)$. We use the fact that the polynomial time computation M_e^A can only make a polynomial number of queries to the oracle A. Changing the oracle outside the set of queries does not change the computation, and this gives us an opportunity to diagonalize. At stage $s = e$, pick a fresh number n, that is, a number larger than any number used so far in the construction, and also ensure that $2^n > p_e(n)$.

Case 1. Suppose that $M_e^{A_s}(0^n) = 1$. Then we can satisfy R_e by keeping $A \cap \{0,1\}^n = \emptyset$, so that $0^n \notin B$.

Case 2. Otherwise we have $M_e^{A_s}(0^n) = 0$. (W.l.o.g. M_e^X is total and 0-1-valued for all X.) Now we want that $0^n \in B$, that is, we have to put something of length n into A. There are 2^n strings of length n, and the computation $M_e^{A_s}(0^n)$ can make at most $p_e(n)$ queries to A_s, so there must be a string $x \in \{0,1\}^n$ that is not queried. Hence, adding x to A_s does not change the computation. Also, since n was fresh, adding x does not change any computation from a previous stage by the Use Principle 5.3.1. So in this case, setting

$$A_{s+1} = A_s \cup \{x\}$$

satisfies R_e, without destroying the satisfaction of previous requirements R_d with $d < e$. □

As pointed out in section 4.3, Theorems 5.3.2 and 5.3.3 put methodological limits on the possible ways we can prove P = NP or P ≠ NP.

5.4 Many-one versus Turing reductions

The definition of p-T-reduction seems much more liberal than that of p-m-reduction. Let us prove that they are indeed different.

Theorem 5.4.1. (Ladner, Lynch, and Selman [26]) *The relations* $\leqslant^p_m$ *and* $\leqslant^p_T$ *differ on* EXP.

Proof. This is in fact an easier diagonalization, exploiting the fact that a p-m-reduction is only allowed a single query. We want to construct sets A and B such that $A \leqslant^p_T B$ but $A \not\leqslant^p_m B$. Clearly the former is satisfied if we let $n \in A$ if and only if either $2n \in B$ or $2n+1 \in B$. Let f_e be an efficient enumeration of all polynomial time p-m-reductions. We want to satisfy for every e the requirement

$$R_e : \qquad \exists n\ A(n) \neq B(f_e(n)),$$

that is, A does not p-m-reduce to B via f_e, as witnessed by the counterexample n. We construct $A = \bigcup_s A_s$ and $B = \bigcup_s B_s$ in stages. At stage $s = e$ of the construction we are given finite sets A_s and B_s, and we have already satisfied R_d for all $d < e$. To satisfy R_e we pick a fresh n, i.e. an n bigger than any number used so far in the construction, and we compute $f_e(n)$. If $f_e(n) \in B_s$ then we let $n \notin A$. In this case we should also have $2n \notin B$ and $2n+1 \notin B$, which is indeed the case since n is fresh, so B_s does not contain any elements bigger than n. If $f_e(n) \notin B_s$ then we let $n \in A_{s+1}$, and we also choose one of $2n$ and $2n+1$ that is different from $f_e(n)$ and put it into B_{s+1}. This clearly satisfies R_e, without affecting the satisfaction of earlier requirements since n was picked fresh.

To ensure that A and B are in EXP, we can easily modify the construction by considering only n such that $f_e(n) < 2^n$. Since f_e is a polynomial, such n always exist. This extra condition ensures that A and B are in TIME(2^n). □

Do these two fundamental reduction notions differ on NP? To prove this, it seems one needs an assumption stronger than just P $\neq$ NP. E.g. the answer is yes if there is a sparse set in NP $-$ P. (Cf. for example [28, p223,254].)

Definition 5.4.2. A set A is *sparse* if there is a polynomial p such that for all n, $|A \cap \{0,1\}^{\leqslant n}| \leqslant p(n)$. Here $\{0,1\}^{\leqslant n}$ denotes the set of all strings of length at most n. A set A is *tally* if $A \subseteq \{0\}^*$.

Definition 5.4.3. A polynomial time computable function $f : \{0,1\}^* \to \{0,1\}^*$ is a *p-isomorphism* if f is bijective and has a polynomial time computable inverse. Two sets A and B are *p-isomorphic* if there is a p-isomorphism f such that $f(A) = B$.

Berman-Hartmanis Conjecture *All sets that are p-m-complete for* NP *are p-isomorphic.*

This conjecture implies that all NP-complete sets have the same density. Indeed, the following is known:

Theorem 5.4.4. (Mahaney) *If there is a sparse p-m-complete set for* NP*, then* P = NP*.*

For a proof of this theorem see e.g. Odifreddi [28, p206]. We will return to sparse complete sets in Exercise 5.7.6 and section 7.5.

5.5 The Gap Theorem

By the time hierarchy theorem, for time constructible functions f we have $\mathrm{TIME}(f) \neq \mathrm{TIME}(2^f)$. The next result shows that the assumption that f is time constructible is essential for this result.

Theorem 5.5.1. (Gap Theorem, Trakhtenbrot, Borodin) *There exists a computable function f such that* $\mathrm{TIME}(f(n)) = \mathrm{TIME}(2^{f(n)})$.

Proof. We construct f in stages, satisfying for every e the requirement

$$R_e: \qquad \forall x\big(|x| = n \geqslant e \wedge M_e(x)\!\downarrow \text{ in } t \text{ steps } \implies t \notin [f(n), 2^{f(n)}]\big).$$

This suffices to prove the theorem: Suppose that $A \in \mathrm{TIME}(2^{f(n)})$ as witnessed by M_e. Then for every x, $M_e(x)\!\downarrow$ within $2^{f(n)}$ steps, hence by R_e, for every x of length $\geqslant e$, $M_e(x)\!\downarrow$ within $f(n)$ steps. Since the finitely many exceptions do not matter, we have $A \in \mathrm{TIME}(f(n))$.

At stage n we define $f(n)$ as follows. We consider all computations $M_e(x)$ with $e \leqslant n$ and $|x| = n$. Define the sequence $k_0 = 0$, $k_{l+1} = 2^{k_l}$. Call k_l *wrong* for a computation $M_e(x)$ if $M_e(x)\!\downarrow$ in t steps and $t \in [k_l, 2^{k_l}]$. Note that we can compute whether k_l is wrong for $M_e(x)$ by running $M_e(x)$ for 2^{k_l} steps. Note further that there can be at most one k_l wrong for $M_e(x)$. Since at stage n we consider only finitely many computations $M_e(x)$ it follows that there is a k_l that is not wrong for any of these. Define $f(n) = k_l$ for the least such k_l.

Now suppose that $|x| \geqslant e$. Then at stage $n = |x|$ the computation $M_e(x)$ is taken into consideration, and hence R_e is satisfied. □

5.6 The Speed-Up Theorem

We often talk about "the" complexity of a computational problem. We already know that this use of language is not entirely appropriate because of the linear speed-up theorem: Any program computing a function can be sped

up by any linear factor on almost all of its arguments. We can welcome this situation by interpreting it as that we can be sloppy about constants, as we in fact did. But the situation is in fact much worse than this. In this section we show that there are computable problems A such that any program for A can be sped up exponentially. This result challenges the idea that every problem has a well-defined level of complexity.

The result in fact holds for any measure of complexity, but we will prove it here for time complexity. Also, the result holds in more generality: Instead of an exponential speed-up one can use *any* fast growing computable function, cf. Exercise 5.7.12. The idea of speed-up goes back to Gödel [16], who proved a speed-up theorem about the length of proofs in arithmetic.

As before in Chapter 2, fix an enumeration $\{M_e\}_{e\in\mathbb{N}}$ of all deterministic Turing machines. In the following we will identify M_e with the language computed by it. We call e an *index* for the language M_e. Let $\Phi_e(x)$ denote the number of computation steps of $M_e(x)$ if this computation converges, and let $\Phi_e(x)$ be undefined otherwise. Note that in general we cannot decide whether $M_e(x)$ converges, hence neither whether $\Phi_e(x)$ is defined, but given t we can decide whether $\Phi_e(x) < t$ by running M_e for t steps. We will use this fact in the proof.

Theorem 5.6.1. (Speed-Up Theorem, M. Blum [6]) *There exists a computable set A such that for every index e for A there is another index i for A such that*

$$\forall^\infty x\big(\Phi_i(x) \leqslant \log \Phi_e(x)\big). \tag{5.1}$$

That is, the program M_i computes A exponentially faster than M_e.

Proof. Let $g(x) = 2^x$, and define $g^{(1)}(x) = g(x)$ and $g^{(n+1)}(x) = g(g^{(n)}(x))$. So $g^{(n)}(x)$ is an iterated stack of exponentials of height n. Also, let $g^{(n)}(x) = 0$ if $n \leqslant 0$. First we define a *scale* of functions as follows. Let

$$h_e(x) = g^{(x-e)}(0).$$

Then $g(h_{e+1}(x)) = h_e(x)$ for all $x > e{+}1$, so the functions h_e form a decreasing family of functions, with an exponential gap between every two of them.

We construct a set A such that

(I) $\forall e\big(M_e = A \Longrightarrow \forall^\infty x \;\; h_e(x) \leqslant \Phi_e(x)\big)$,

(II) $\forall e \exists i\big(M_i = A \wedge \forall^\infty x \;\; \Phi_i(x) \leqslant h_{e+1}(x)\big)$.

Note that (I) and (II) together imply (5.1):

$$\Phi_i(x) \leqslant h_{e+1}(x) \leqslant \log h_e(x) \leqslant \log \Phi_e(x).$$

Construction. To obtain (I), we satisfy for every e the requirement

$$R_e: \qquad \exists^\infty x\big(\Phi_e(x) < h_e(x)\big) \Longrightarrow M_e \neq A.$$

At stage x of the construction we define $A(x)$ as follows. Say that e *requires attention* at stage x if $e \leqslant x$, e is not yet cancelled, and $\Phi_e(x) < h_e(x)$. At stage x, choose the least e that requires attention, define $A(x) = 1 - M_e(x)$, and declare e *cancelled*. If there is no such e we give $A(x)$ some default value, say $A(x) = 0$. This ends the construction of A.

We verify that the construction satisfies all requirements. Suppose that the premiss of R_e holds, that is, there are infinitely many x such that $\Phi_e(x) < h_e(x)$. Since there are only finitely many $d < e$ that can require attention at a stage x, and every time this happens some $d < e$ is cancelled, there are only finitely many stages at which e requires attention but is not the least such number. At the first stage following this where e requires attention (which exists by assumption), e is the least such number and it is ensured that $A \neq M_e$. Hence R_e is satisfied.

It remains to verify (II). Clearly the construction above gives an algorithm for computing A. We want to compute A in a more efficient way. Fix $u \in \mathbb{N}$. Suppose we know the finite set

$$F_u = \big\{(e, x, A(x)) : e < u \wedge e \text{ cancelled at stage } x\big\}.$$

Then we can compute the set A as follows. We run the above construction, but instead of considering all $e \leqslant x$ at stage x we only consider e with $u \leqslant e \leqslant x$. The full information of what happened at stage x in the original construction can be recovered from the finite set F_u. The point of this is that we save having to run the functions $h_e(x)$ for $e < u$. Instead, we directly find out which, if any, of these e required attention by consulting F_u. To see which e with $u \leqslant e \leqslant x$ requires attention we only have to compute $h_e(x)$ for $e \geqslant u$. Hence the time to compute $A(x)$ is bounded by

$$x \cdot h_u(x) + O(1) \leqslant h_{u-1}(x) \text{ a.e. } x.$$

This proves (II), because we may take u as large as we want. □

5.7 Exercises

Exercise 5.7.1.* Use delayed diagonalization to prove that the p-m-degrees of the computable sets are *dense*, that is, if A and C are computable and $C <_m^p A$ then there is a computable B such that $C <_m^p B <_m^p A$.

Exercise 5.7.2.* Same as Exercise 5.7.1, but now for $\leqslant_T^p$ instead of $\leqslant_m^p$.

Exercise 5.7.3.* Assuming that NP $\neq$ P, show that there are *incomparable* p-m-degrees in NP, that is, show that there are sets A and B in NP such that neither $A \leqslant_T^p B$ nor $B \leqslant_T^p A$. Note that it automatically follows that

A and B are intermediate sets. This result is the analogue of the Friedberg-Muchnik theorem in computability theory that states that there are Turing-incomparable c.e. sets.

Exercise 5.7.4. For every sparse set S there is a tally set T such that $S \leqslant_T^p T$. Hint: consider

$$T = \left\{0^{\langle n,j,i\rangle} : \text{the } i\text{-th bit of the } j\text{-th string in } S^{=n} \text{ is } 1\right\}$$

Exercise 5.7.5. Show that the Berman-Hartmanis conjecture implies that P $\neq$ NP.

Exercise 5.7.6. Let $\{f_i\}_{i\in\mathbb{N}}$ be an effective enumeration of all p-m-reductions. Call f *almost injective* if $f(x) = f(y)$ for only finitely many pairs x, y with $x \neq y$.

(a) Show that there is a set $B \in$ EXP such that for *any* set A and any i, if f_i is not almost injective then B does not p-m-reduce to A via f_i.

(b) Show that there exists B as in (a) such that $|B \cap \{0,1\}^{\leqslant n}| \geqslant 2^n$.

(c) Show that there are no sparse p-m-complete sets for EXP. (See also Theorem 7.5.2.)

Exercise 5.7.7. Use diagonalization to explicitly construct a computable function that is not time constructible.

Exercise 5.7.8. Show that the function f from Theorem 5.5.1 is bounded by a time constructible function.

Exercise 5.7.9. Prove the analog of the Gap Theorem for space complexity.

Exercise 5.7.10. Prove that the gap in Theorem 5.5.1 can be made arbitrarily large by using, instead of 2^n, an arbitrary computable function.

Exercise 5.7.11. Given any nondecreasing computable function g, with at least $g(x) \geqslant x$, construct a *g-scale*, that is, a descending family of functions $\{h_e\}_{e\in\mathbb{N}}$ such that

$$g(h_{e+1}(x)) \leqslant h_e(x)$$

for every x.

Exercise 5.7.12. Let g be an arbitrary fast growing computable function with $g(x) \geqslant x$. Formulate and prove a general version of the speed-up theorem, using g instead of 2^x.

Chapter 6

Randomized computation

6.1 Probabilistic classes

In this section we consider variants of NP where the acceptance criterion is given by a probabilistic condition over all paths. Consider nondeterministic Turing machines M whose computation trees are full binary trees, and in which every computation path has the same length. Given a set L and a machine M that is supposed to compute L, and an input x, the *error probability* is the ratio of the number of paths giving the wrong answer and the total number of paths. When thinking of Turing machines in this way we speak also of *probabilistic* Turing machines. All the probabilistic classes considered in this chapter were introduced by Gill [15].

Definition 6.1.1. The class PP, for *probabilistic polynomial time*, is the class of sets L that are computed in polynomial time by a probabilistic machine such that

$$x \in L \Leftrightarrow \text{the fraction of accepting paths is } > \tfrac{1}{2},$$
$$x \notin L \Leftrightarrow \text{the fraction of rejecting paths is } \geqslant \tfrac{1}{2}.$$

By Exercise 6.5.1, we can replace the second clause in the definition by

$$x \notin L \Rightarrow \text{the fraction of rejecting paths is } > \tfrac{1}{2}.$$

Example 6.1.2. Let MAJ be the subset of SAT consisting of the Boolean formulas that are satisfied by more than $\frac{1}{2}$ of all assignments. Then MAJ is in PP.

Proposition 6.1.3. NP $\subseteq$ PP $\subseteq$ PSPACE.

Proof. PP $\subseteq$ PSPACE: Try all paths, reusing the same space and keep a counter for the paths used.

NP $\subseteq$ PP: We can convert any NP-computation into a PP-computation as in Figure 6.1, where on the right is the original NP-computation and on the left we add a tree of the same size where all paths accept. □

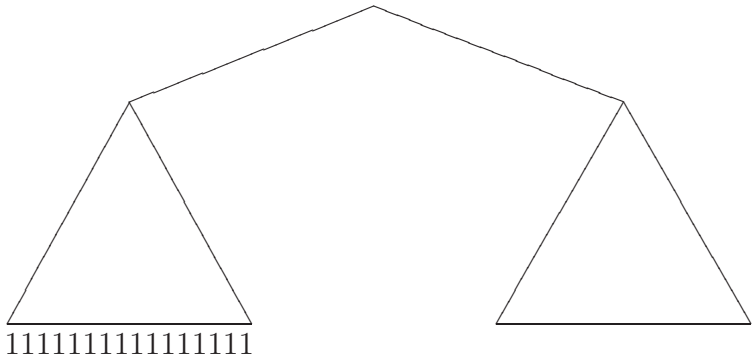


Figure 6.1: Converting an NP into a PP computation

It follows from Exercise 6.5.1 that PP is closed under complementation, hence also co-NP $\subseteq$ PP. The closure of PP under unions and intersections was open for a long time before it was finally solved in [5].[1] By Exercise 6.5.2, PP is closed under p-m-reductions.

Like NP, the class PP has sets that are complete under p-m-reductions. Consider the set

$$\mathrm{SA} = \big\{\langle \varphi, i\rangle : \varphi \text{ is satisfied by } > i \text{ assignments}\big\}.$$

Theorem 6.1.4. *The sets* MAJ *and* SA *are* PP*-complete.*

Proof. We already noticed in Example 6.1.2 that MAJ is in PP. We first show that SA $\leqslant^p_m$ MAJ. Then SA $\in$ PP follows from the fact that PP is closed under p-m-reductions (Exercise 6.5.2). Let $\langle \varphi, i\rangle$ be an instance of SA, where φ has m variables. Note that we must have that $2^m \geqslant i$. Let ψ be a formula on the same m variables as φ with exactly $2^m - i$ satisfying assignments. (One can effectively construct such a formula, cf. Exercise 6.5.3.) Now define

$$f(\langle \varphi, i\rangle) = (y \wedge \varphi) \vee (\neg y \wedge \psi),$$

where y is a new variable. Then φ has $> i$ satisfying assignments if and only if $f(\langle \varphi, i\rangle)$ has $> i + 2^m - i = 2^m$ satisfying assignments. Note that $2^m = \frac{1}{2}2^{m+1}$, and that $m+1$ is the number of variables in $f(\langle \varphi, i\rangle)$. Hence f is a p-m-reduction from SA to MAJ.

To prove completeness of SA, use the same reduction $x \mapsto \varphi_x$ as for the NP-completeness of SAT (Theorem 3.2.2). Given $A \in$ PP computed in time $p(n)$, x is accepted if and only if $\geqslant 2^{p(n)-1}+1$ paths accept x. Hence $A \leqslant^p_m$ SA via $x \mapsto \langle \varphi_x, 2^{p(n)-1}\rangle$. □

[1] Key to the proof is to approximate the sign of the function #acc − #rej, the difference of the number of accepting and rejecting paths, by suitable polynomials. This can only be done for each length separately.

Definition 6.1.5. The class BPP, for *bounded probabilistic polynomial time*, is the class of sets L that are recognized in polynomial time by probabilistic machines with error probability bounded away from $\frac{1}{2}$, i.e. such that for some $\varepsilon > 0$ and every x,

$$x \in L \Leftrightarrow \text{the fraction of accepting paths is } > \tfrac{1}{2} + \varepsilon,$$
$$x \notin L \Leftrightarrow \text{the fraction of rejecting paths is } > \tfrac{1}{2} + \varepsilon.$$

Obviously P $\subseteq$ BPP $\subseteq$ PP. Note that the relation between BPP and NP is not clear. The interest in the class BPP comes from the following result, which implies that the answers given by BPP-machines can be boosted to any precision arbitrarily close to complete correctness.

Theorem 6.1.6. *$A \in$ BPP if and only if for all polynomials p there is a probabilistic Turing machine M recognizing A in polynomial time with error probability $\leqslant \frac{1}{2^{p(n)}}$.*

Proof. We simply iterate the computations of M a polynomial number of times and take the majority of the answers. A straightforward computation shows that this will suffice to obtain correct answers with a high probability. Suppose M recognizes A with error probability $\varepsilon < \frac{1}{2}$. Let $\delta = 1 - \varepsilon$, and fix a polynomial p. Let $q(n) = cp(n)$, where c is a constant such that

$$\big(4\varepsilon\delta\big)^c < \frac{1}{2}. \tag{6.1}$$

(Note that $\varepsilon\delta < \frac{1}{4}$ because $(\frac{1}{2} + \rho)(\frac{1}{2} - \rho) < \frac{1}{4}$ for every $\rho > 0$.) Now, given an input x of length n, iterate the nondeterministic computation $M(x)$ $m = 2q(n) + 1$ times, and accept only when at least $q(n) + 1$ of the computations accept. The probability that this procedure gives a wrong answer is the probability that at most $q(n)$ computations are correct, which is

$$\sum_{j=0}^{q(n)} \binom{m}{j} \delta^j \varepsilon^{m-j}. \tag{6.2}$$

Since $\delta > \varepsilon$ and $j \leqslant \frac{m}{2}$, we have $\delta^j \varepsilon^{m-j} \leqslant \delta^{\frac{m}{2}} \varepsilon^{\frac{m}{2}}$ so the expression (6.2) is less or equal to

$$\begin{aligned}
\delta^{\frac{m}{2}} \varepsilon^{\frac{m}{2}} \sum_{j=0}^{q(n)} \binom{m}{j} &< \delta^{\frac{m}{2}} \varepsilon^{\frac{m}{2}} 2^m && \text{(by the binomial formula } 2^m = \textstyle\sum_{j=0}^{m} \binom{m}{j}\text{)} \\
&= \big(4\varepsilon\delta\big)^{\frac{m}{2}} && \\
&\leqslant \big(4\varepsilon\delta\big)^{cp(n)} && (\tfrac{m}{2} = cp(n) + \tfrac{1}{2}) \\
&< \frac{1}{2^{p(n)}} && \text{(by choice of } c\text{)}
\end{aligned}$$

□

Proposition 6.1.7. BPP *is closed under complementation, unions, and intersections.*

Proof. Exercise 6.5.4. □

6.2 More about BPP

In this section we prove two fundamental results about BPP. We first prove that BPP has short advice strings, a notion that is related to *circuit complexity*, cf. Chapter 7. Then we prove that BPP is included in Σ_2^p, the second level of the polynomial time hierarchy.

Definition 6.2.1. Fix $B \in \mathrm{P}$. Given a set A, y is *correct advice* for A at length n if

$$x \in A \iff \langle x, y\rangle \in B \tag{6.3}$$

for every x of length n.

One can think of the advice string y as coding a circuit to determine membership in A for all x of a given length. We now prove that the sets in BPP have (many) short advice strings for each length. Using the circuit terminology one can phrase this by saying that BPP has small circuits, cf. section 7.3.

Theorem 6.2.2. (Adleman) *The following are equivalent.*

(i) $A \in \mathrm{BPP}$,

(ii) *For any polynomial q there exist $B \in \mathrm{P}$ and a polynomial p such that for all n, among all strings of length $p(n)$ there are at least*

$$2^{p(n)}\left(1 - \frac{1}{2^{q(n)}}\right).$$

correct advice strings y for A at length n, as in (6.3).

Proof. (ii)⇒(i) is straightforward: Given B and p and n, randomly pick an advice string y of length $p(n)$. Then with high probability, $A(x) = B(\langle x, y\rangle)$.

(i)⇒(ii). Fix A and q. By Theorem 6.1.6 we may assume that there is a machine M accepting A with error probability at most $\left(\frac{1}{2}\right)^{q(n)+n}$. Let p be the running time of M. Given x of length n, at most $2^{p(n)-q(n)-n}$ paths are incorrect. Therefore at most $2^n \cdot 2^{p(n)-q(n)-n}$ paths are incorrect for *some* x, and hence $2^{p(n)} - 2^{p(n)-q(n)}$ paths are correct for all x, and we can use any of these as a correct advice string. □

Sipser proved that BPP $\subseteq$ PH, which was improved by Gács to the following:

Theorem 6.2.3. BPP $\subseteq \Sigma_2^p$.

Proof. Let $L \in$ BPP. By Theorem 6.1.6 we may assume that there is a machine M accepting L with error probability at most $\frac{1}{2^n}$. Let l be the length of the paths in M. Given a path $r \in \{0,1\}^l$, denote the outcome of M along that path by $M(x,r)$, and define

$$A(x) = \big\{r \in \{0,1\}^l : M(x,r) \text{ accepts}\big\}.$$

We can view $\{0,1\}^l$ as a vector space of dimension l over the finite field $\mathbb{F}_2$, with addition of vectors modulo 2. For $t \in \{0,1\}^l$ define $A(x)+t = \big\{r+t : r \in A(x)\big\}$. Thus every t defines a 1-1 map, that we can think of as a translation. Note that $(r+t)+t = r$ for every r and t. If $A(x)$ is large then there is a small set of translations of it covering the whole space $\{0,1\}^l$:

Claim 1. If $|A(x)| \geqslant 2^l\big(1 - \frac{1}{2^{|x|}}\big)$ then there exist $t_1 \dots t_l$, all of length l, such that $\bigcup_i A(x) + t_i = \{0,1\}^l$.

To prove Claim 1, pick $t_1 \dots t_l$ at random. (That is, we use the probabilistic method to prove the existence of the t's.) Let $S = \bigcup_i A(x) + t_i$. For every $r \in \{0,1\}^l$ we have

$$\begin{aligned} \Pr\big[r \notin S\big] &= \prod_i \Pr\big[r \notin A(x) + t_i\big] \\ &\leqslant \left(\frac{1}{2^{|x|}}\right)^l. \end{aligned}$$

(Here Pr refers to the choice of the t_i's.) Hence

$$\Pr\big[\exists r\ r \notin S\big] \leqslant \sum_r 2^{-|x|l} = 2^{l-|x|l} \leqslant 2^{-|x|},$$

since $l - |x|l \leqslant -|x|$ when $l, |x| \geqslant 2$. Hence $\Pr\big[S = \{0,1\}^l\big] \geqslant 1 - \frac{1}{2^{|x|}}$. In particular there exist $t_1 \dots t_l$ such that $S = \{0,1\}^l$. This proves Claim 1.

Claim 2. If $|A(x)| \leqslant 2^l \frac{1}{2^{|x|}}$ then there do not exist $t_1 \dots t_l$ of length l such that $\bigcup_i A(x) + t_i = \{0,1\}^l$.

To prove Claim 2, note that $|A(x)+t_i| \leqslant 2^{l-|x|}$ for every i and that $l \cdot 2^{l-|x|} < 2^l$ because l is polynomial in $|x|$. This proves Claim 2.

Now to finish the proof of the theorem, note that it follows from these claims that we can write

$$x \in L \Leftrightarrow \exists t_1 \dots t_l \in \{0,1\}^l \forall r \in \{0,1\}^l \bigvee_{1 \leqslant i \leqslant l} M(x, r + t_i) \text{ accepts}$$

and that this is a Σ_2^p-formula, with polynomially bounded quantifiers. □

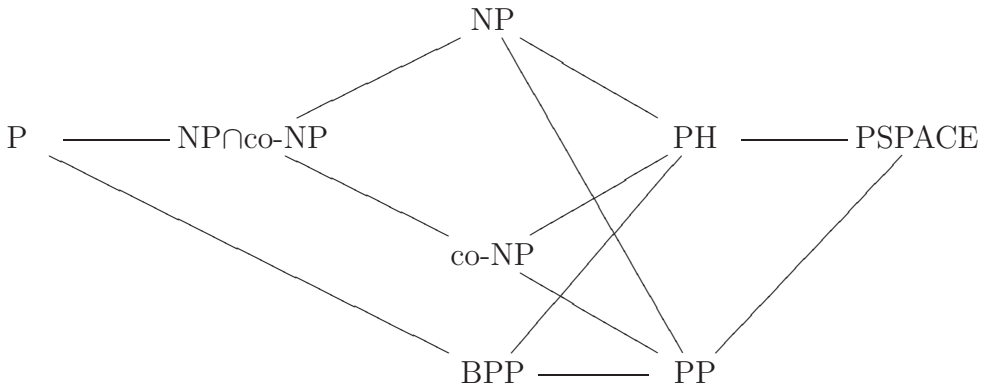


Figure 6.2: Relations between complexity classes

Note that it follows immediately from Theorem 6.2.3 that BPP $\subseteq \Sigma_2^p \cap \Pi_2^p$ because BPP is closed under complementation (Proposition 6.1.7). In contrast with PP, there are no problems known to be complete for BPP. It has been conjectured that P = BPP, and of course this statement is equivalent to the statement that all the problems in P are complete for BPP.

Figure 6.2 summarizes some of the relations between complexity classes. The relation between PP and PH is not clear, but the following is known. In Section 6.1 we discussed closure properties of PP. The ultimate closure property would be $\mathrm{P}^{\mathrm{PP}} = \mathrm{PP}$, i.e. the closure of PP under $\leqslant_T^p$. If this is the case then PP in fact contains PH by the following result revealing some of the strength of randomization.

Theorem 6.2.4. (Toda) $\mathrm{PH} \subseteq \mathrm{P}^{\mathrm{PP}}$.

Proof. This important theorem can be proved using *counting classes*, which is a very interesting topic in itself. A proof can be found in [31]. □

6.3 The classes RP and ZPP

Define the following probabilistic classes:

Definition 6.3.1. The class RP, for *randomized polynomial time*, is the class of sets L that are recognized in polynomial time by probabilistic machines with one-sided error probability $\frac{1}{2}$, i.e. such that

$$x \in L \Leftrightarrow \text{the fraction of accepting paths is } \geqslant \tfrac{1}{2},$$
$$x \notin L \Leftrightarrow \text{the fraction of rejecting paths is 1.}$$

By Exercise 6.5.7 the class RP remains the same if we replace the $\geqslant \frac{1}{2}$ in its definition by $\geqslant \frac{2}{3}$. It follows from this that RP $\subseteq$ BPP. RP-algorithms are

also known as *Monte Carlo algorithms*. Note that we have P $\subseteq$ RP $\subseteq$ NP (Exercise 6.5.6).

The class ZPP, for *zero-error probabilistic polynomial time*, is the class of sets L that are recognized in polynomial time by probabilistic machines with *three* possible outcomes: accept, reject, and ? for "don't know". The requirement is that no path gives a wrong answer, and that the fraction of paths that outputs ? is at most $\frac{1}{3}$, so that at least $\frac{2}{3}$ in fact give the correct answer. ZPP-algorithms are also known as *Las Vegas algorithms*. By Exercise 6.5.8 we have ZPP = RP $\cap$ co-RP. Note that this is a probabilistic analog of the question whether P = NP $\cap$ co-NP

6.4 Primes again

We saw in section 2.5 that PRIMES is in NP $\cap$ co-NP. In this section we show that PRIMES is in co-RP, that is, there is a Monte Carlo algorithm for testing compositeness. Given a number n, if n is prime this algorithm will with certainty output the verdict "n is prime", and if n is composite it will with high probability output the verdict "n is composite".

We start with some number-theoretic preliminaries. From now on, p is a prime larger than 2.

Definition 6.4.1. A number $0 < a < p$ is called a *quadratic residue* modulo p if the equation $x^2 \equiv a \bmod p$ has a solution. The *Legendre symbol* is defined as

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{if } a \text{ is a quadratic residue modulo } p, \\ -1 & \text{otherwise.} \end{cases}$$

Theorem 6.4.2. (Euler's criterion) $\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \bmod p$.

Proof. Note that since $(\mathbb{Z}/p\mathbb{Z})$ is a field, the equation $x^2 \equiv a$ can have at most two solutions modulo p. We claim that there are either two or zero solutions. By Theorem 1.6.4, pick a primitive root r modulo p. In particular there is $i < p-1$ such that $r^i \equiv a \bmod p$. If $i = 2j$ is even then $a^{\frac{p-1}{2}} = r^{j(p-1)} \equiv 1 \bmod p$, and $x^2 \equiv a$ has the solutions r^j and $r^{j+\frac{p-1}{2}}$. Note that if i varies over the even numbers, we obtain half of all possible residues, and since each has two square roots, no such roots are left for the odd i. So if $i = 2j+1$ is odd, r^i has no square roots, and $a^{\frac{p-1}{2}} \equiv r^{j(p-1)} r^{\frac{p-1}{2}} \equiv r^{\frac{p-1}{2}} \bmod p$. Since the latter is a square root of 1 unequal to 1, it must be -1. □

Theorem 6.4.2 allows us to compute the Legendre symbol efficiently. Note that we can efficiently compute exponentials by repeated squaring, cf. [8] for more details.

Theorem 6.4.3. (Law of Quadratic Reciprocity) $\left(\frac{p}{q}\right) \cdot \left(\frac{q}{p}\right) = (-1)^{\frac{p-1}{2}\frac{q-1}{2}}$.

Definition 6.4.4. The *Jacobi symbol* generalizes the Legendre symbol as follows. If n is odd and has prime factorization $n = \prod_i q_i$ then

$$\left(\frac{m}{n}\right) = \prod_i \left(\frac{m}{q_i}\right).$$

Lemma 6.4.5. (i) $\left(\frac{m_1 m_2}{n}\right) = \left(\frac{m_1}{n}\right)\left(\frac{m_2}{n}\right)$.

(ii) $\left(\frac{m+n}{n}\right) = \left(\frac{m}{n}\right)$.

(iii) $\left(\frac{n}{m}\right) \cdot \left(\frac{m}{n}\right) = (-1)^{\frac{m-1}{2}\frac{n-1}{2}}$.

Proof. The first two items easily follow from the definitions and the third follows from Theorem 6.4.3. □

It follows from Lemma 6.4.5 that also the Jacobi symbol is effectively computable, in a manner similar to that of Euclid's algorithm for computing the gcd.

Lemma 6.4.6. *If n is odd and $\left(\frac{m}{n}\right) \equiv m^{\frac{n-1}{2}} \bmod n$ for all $m \in (\mathbb{Z}/n\mathbb{Z})^*$ then n is prime.*

Proof. Suppose that the premiss holds but that n is composite. Suppose first that $n = p_1 \dots p_k$ is the product of distinct primes. Let $r \in (\mathbb{Z}/p_1\mathbb{Z})^*$ be such that $\left(\frac{r}{p_1}\right) = -1$. By the Chinese remainder theorem let $m \in (\mathbb{Z}/n\mathbb{Z})^*$ be such that $m \equiv r \bmod p_1$ and $m \equiv 1 \bmod p_j$ for $j = 2 \dots k$. Then $m^{\frac{n-1}{2}} \equiv \left(\frac{m}{n}\right) \equiv -1 \bmod n$, hence $m^{\frac{n-1}{2}} \equiv -1 \bmod p_2$, contradicting that $m \equiv 1 \bmod p_2$.

From the previous contradiction we conclude that $n = p^2 m$ for some $p > 2$ prime. By Exercise 1.7.3, let r be a primitive root modulo p^2. By assumption we have $r^{\frac{n-1}{2}} \equiv \left(\frac{r}{n}\right) \in \{1, -1\} \bmod n$, hence $r^{n-1} \equiv 1 \bmod n$. The latter implies that $n - 1$ is a multiple of $\varphi(p^2) = p(p-1)$, hence p divides $n - 1$. But p also divides n, a contradiction. □

Lemma 6.4.7. *If n is odd and composite, then at least half of the elements $m \in (\mathbb{Z}/n\mathbb{Z})^*$ satisfy $\left(\frac{m}{n}\right) \not\equiv m^{\frac{n-1}{2}} \bmod n$.*

Proof. Consider the set

$$X = \left\{x \in (\mathbb{Z}/n\mathbb{Z})^* : \left(\tfrac{x}{n}\right) \equiv x^{\frac{n-1}{2}} \bmod n\right\}.$$

By Lemma 6.4.5 the elements of X form a subgroup of $(\mathbb{Z}/n\mathbb{Z})^*$, and by Lemma 6.4.6 it is not the whole group. If $a \in (\mathbb{Z}/n\mathbb{Z})^* - X$ then the left coset aX has the same cardinality as X, hence X contains at most half the elements of $(\mathbb{Z}/n\mathbb{Z})^*$. □

Theorem 6.4.8. (Solovay and Strassen [34]) PRIMES *is in* co-RP.

Proof. Given n odd, the algorithm runs as follows: Randomly pick a number m with $2 \leqslant m \leqslant n-1$. If $\gcd(m,n) \neq 1$ then n is composite. (Note that we can effectively compute the gcd using Euclid's algorithm.) Otherwise, $m \in (\mathbb{Z}/n\mathbb{Z})^*$. See if $\left(\frac{m}{n}\right) \equiv m^{\frac{n-1}{2}} \bmod n$. If not, declare n to be composite. If yes, declare n to be probably prime. Note that if p is prime then by Theorem 6.4.2, $\left(\frac{m}{p}\right) \equiv m^{\frac{p-1}{2}} \bmod p$ so in the first case we are sure the answer is correct. If n is composite then by Lemma 6.4.7 the answer is correct with probability at least $\frac{1}{2}$. □

6.5 Exercises

Exercise 6.5.1. Show that if $L \in \mathrm{PP}$ then there is a probabilistic machine M such that $x \notin L \Rightarrow$ the fraction of accepting paths is $< \frac{1}{2}$.

Exercise 6.5.2. PP is closed downwards under $\leqslant^p_m$. That is, if $B \in \mathrm{PP}$ and $A \leqslant^p_m B$, then also $A \in \mathrm{PP}$.

Exercise 6.5.3. Show that, given a set of n Boolean variables $\{x_1, \ldots, x_n\}$ and $i \leqslant 2^n$, there is a formula in these variables having exactly i satisfying assignments. Furthermore, the formula is of size a polynomial in n. (Hint: First do this when i is a power of 2. Then for the general case write i in binary as a sum of powers of 2 and combine the formulas of the components by disjunctions, making sure that the assignments do not overlap.)

Exercise 6.5.4. Prove Proposition 6.1.7.

Exercise 6.5.5. Use Theorem 6.1.6 to prove that in Definition 6.1.5 we can allow ε to be a function of the form $1/p(n)$, where p is a polynomial and n is the size of the input.

Exercise 6.5.6. Prove that $\mathrm{P} \subseteq \mathrm{RP} \subseteq \mathrm{NP}$.

Exercise 6.5.7. Prove that we obtain an equivalent definition of RP if we replace the $\geqslant \frac{1}{2}$ in its definition by $\geqslant \frac{2}{3}$. (Hint: Because the error is one-sided, we can achieve this by iteration. We cannot use the same argument to prove that $\mathrm{PP} \subseteq \mathrm{BPP}$.) Conclude from this that $\mathrm{RP} \subseteq \mathrm{BPP}$.

Exercise 6.5.8. Prove that $\mathrm{ZPP} = \mathrm{RP} \cap \text{co-RP}$.

Exercise 6.5.9. Show that $\mathrm{P} \subseteq \mathrm{ZPP} \subseteq \mathrm{NP} \cap \text{co-NP}$.

Exercise 6.5.10. Prove that $\mathrm{NP} \subseteq \mathrm{BPP}$ if and only if $\mathrm{NP} = \mathrm{RP}$. (Hint: One direction is easy from Exercise 6.5.7. For the nontrivial direction, first try to find, bit by bit, a certificate for the NP-computation, using the BPP-machine. Second, use the NP-machine to *check* whether the certificate is correct.) See also Exercise 7.6.8.

Chapter 7

Circuit complexity

7.1 Boolean functions

We already discussed in Exercise 1.7.2 that every propositional formula can be put into conjunctive normal form. As mentioned before (on page 19), putting a formula in CNF may result in an exponential blow-up. In this chapter we will consider various ways in which Boolean functions and formulas may be represented succinctly.

Definition 7.1.1. An n-ary *Boolean function* is a function $f : \{0,1\}^n \to \{0,1\}$.

First note that every propositional formula $\varphi(x_1, \ldots, x_n)$ of n variables defines an n-ary Boolean function. Conversely, every Boolean function can be represented by a propositional formula:

Proposition 7.1.2. *For every n-ary Boolean function f there is a propositional formula φ such that the function defined by φ is equal to f.*

Proof. Define

$$F = \big\{\vec{z} \in \{0,1\}^n : f(\vec{z}) = 1\big\}.$$

For every vector $\vec{z} = (z_1, \ldots, z_n) \in \{0,1\}^n$ define the formula $\varphi_{\vec{z}}$ as the conjunction of all variables x_i such that $z_i = 1$ and all negated variables $\neg x_i$ such that $z_i = 0$. Finally define the formula φ by

$$\varphi = \bigvee_{\vec{z} \in F} \varphi_{\vec{z}}.$$

Then clearly φ is true under an assignment $\vec{z}$ of its variables precisely when $f(\vec{z}) = 1$. □

Note that the *size* of the formula φ in Proposition 7.1.2 can be exponential in n.

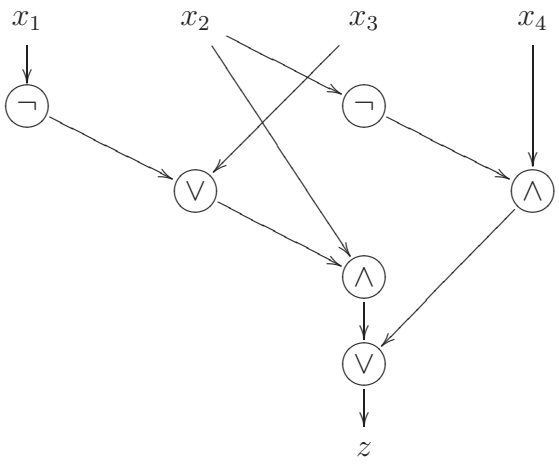


Figure 7.1

7.2 Circuits

A *circuit* is a directed acyclic graph (*dag* for short), the nodes of which are called *gates*. The in-degree, or *fan-in*, of a node is the number of arrows leading to it, and the out-degree, or *fan-out*, the number of arrows leaving it. There are input gates $x_1, \ldots, x_n$, output gates $z_1, \ldots, z_m$, and Boolean gates of type not, and, and or. The and-gates and or-gates have in-degree 2 and the not-gates in-degree 1. The fan-out of any gate can be arbitrary. The input gates act as function inputs that can be 0 or 1. Thus a circuit computes a function from $\{0,1\}^n$ to $\{0,1\}^m$. In principle n and m can be arbitrary, but we will mostly consider circuits with one output gate, i.e. with $m = 1$. Figure 7.1 pictures an example circuit. The Boolean formula corresponding to this circuit is $\big((\neg x_1 \vee x_3) \wedge x_2\big) \vee \big(\neg x_2 \wedge x_4\big)$.

The *size* of a circuit is the total number of and-gates, or-gates, and not-gates.[1] Thus we can take as a measure of complexity of a Boolean function $f : \{0,1\}^n \to \{0,1\}$ the minimal size of a circuit computing it.

To extend the notion of circuit complexity to arbitrary languages $L \subseteq \{0,1\}^*$ we consider circuit *families*, with one circuit for every input length, as follows.

Definition 7.2.1. Suppose that $\{C_n\}_{n\in\mathbb{N}}$ is a family of circuits such that for every n, C_n computes a Boolean function of n inputs. We say that such a family *computes* a language $L \subseteq \{0,1\}^*$ if for every length n and every $x \in \{0,1\}^n$,

$$x \in L \iff C_n(x) = 1.$$

For a given function $s : \mathbb{N} \to \mathbb{N}$, $\mathrm{SIZE}(s(n))$ denotes the class of languages

[1] Some authors do not count the not-gates, which makes no essential difference.

L that are computed by a circuit family such that the size of C_n is bounded by $s(n)$.

It is important to note that circuit size complexity as defined here is a *nonuniform* measure of complexity, in the sense that in the definition of $\mathrm{SIZE}(s(n))$ there is no computational restriction on the sequence of circuits $\{C_n\}_{n\in\mathbb{N}}$. (See also Exercise 7.6.3.) In some contexts it makes sense to impose extra computational restrictions, e.g. by requiring that the sequence $\{C_n\}_{n\in\mathbb{N}}$ is uniformly computable in polynomial time. As for now, we stick to the nonuniform definition.

The following result shows that we do not need to consider circuits of more than exponential size.

Proposition 7.2.2. *Any language $L \subseteq \{0,1\}^*$ is in $\mathrm{SIZE}(O(2^n))$.*

Proof. Given L and a length n, we can consider the characteristic function of $L \cap \{0,1\}^n$. We can describe this Boolean function with a formula in DNF of size $O(2^n)$ (cf. the proof of Proposition 7.1.2). Finally, we can easily translate this formula into a circuit of roughly the same size. □

Proposition 7.2.3. *There are languages that do not have subexponential circuit size: For any s such that $s(n) = 2^{o(n)}$ there is a language $L \notin \mathrm{SIZE}(s(n))$.*

Proof. This follows by counting the circuits of size s. Assuming that $s \geqslant n$, we arrive at an upper bound as follows. There are s internal gates and n input gates. To describe the circuit, we need to indicate for every gate the type (one of three) and where its two inputs are coming from. In total this costs no more than

$$s \cdot (2\log(n+s)+3) \leqslant s \cdot (2\log 2s + 3) = s \cdot (2\log s + 5)$$

bits. Thus the total number of circuits of size s is bounded by $2^{s(2\log s+5)}$. On the other hand, there are 2^{2^n} Boolean functions $f : \{0,1\}^n \to \{0,1\}$. Now if $s(n) = 2^{o(n)}$ then $s(n)(2\log s(n) + 5) \in o(2^n)$, because $2^{o(n)} \cdot 2^{o(n)} = 2^{o(n)} \in o(2^n)$. Since $2^{o(2^n)}$ is asymptotically smaller than 2^{2^n}, there are not enough circuits of size $s(n)$ to compute all Boolean functions, if n is large. □

7.3 The relation with advice classes

Recall the definition of advice strings (Definition 6.2.1).

Definition 7.3.1. Let $\mathcal{C}$ be a complexity class and let $\mathcal{F}$ be a class of *advice functions* of the form $s : \mathbb{N} \to \{0,1\}^*$. We think of s as giving an advice string for every length n. Then $\mathcal{C}/\mathcal{F}$ is the class of all sets of the form

$$\{x : \langle x, s(|x|)\rangle \in B\}$$

for some $B \in \mathcal{C}$ and some $s \in \mathcal{F}$. Let poly be the class of polynomially bounded advice functions, i.e. those s such that $|s(n)| \leqslant n^c$ for some constant c. In particular P/poly is the class of sets that have polynomial advice.

Theorem 7.3.2. $\mathrm{P/poly} = \mathrm{SIZE}(n^{O(1)})$.

Proof. The direction $\mathrm{SIZE}(n^{O(1)}) \subseteq \mathrm{P/poly}$ follows because we can evaluate a circuit in polynomial time. This means that the set

$$B = \big\{\langle x, C\rangle : C \text{ circuit } \wedge\ C(x) = 1\big\}$$

is in P. So if $A \in \mathrm{SIZE}(n^{O(1)})$ and for every length n, C_n is a polynomial size circuit for $A \cap \{0,1\}^n$, then $x \in A \Leftrightarrow \langle x, C_{|x|}\rangle \in B$, hence $A \in \mathrm{P/poly}$.

For the converse direction $\mathrm{P/poly} \subseteq \mathrm{SIZE}(n^{O(1)})$ we have to do a bit more work. We have to show that computations of a polynomial time Turing machine on inputs of a fixed length n can be converted into a polynomial size circuit. We leave this as Exercise 7.6.5. □

Note that by Theorem 6.2.2 we have $\mathrm{BPP} \subseteq \mathrm{P/poly}$, so from the previous theorem we obtain:

Theorem 7.3.3. $\mathrm{BPP} \subseteq \mathrm{SIZE}(n^{O(1)})$.

7.4 Small circuits for NP?

In this section we prove that not all sets in NP have polynomial size circuits, unless the polynomial hierarchy collapses to the second level, that is, if $\Pi_2^p \subseteq \Sigma_2^p$. For this we need to prove that polynomially bounded $\forall\exists$-statements can be written as $\exists\forall$-statements. This uses an old proof-theoretic trick called *Skolemization*. The idea is that if the formula

$$\forall y \exists z\ \varphi(y, z)$$

holds in $\mathbb{N}$ (or in $\{0,1\}^*$) then there is an implicit *Skolem function* $f : \mathbb{N} \to \mathbb{N}$ such that $\forall y\, \varphi(y, f(y))$. That is, f picks for every y a witness $z = f(y)$ such that $\varphi(y, f(y))$. Now the statement that such an f exists, i.e.

$$\exists f \forall y\ \varphi(y, f(y))$$

has the required Σ_2^p-form, provided that we have a polynomially bounded way to describe f. Here we use the assumption that SAT has small circuits. The next lemma shows that in this case we actually have small circuits that *produce* satisfying assignments, instead of just deciding satisfiability.

Lemma 7.4.1. *If* $\mathrm{NP} \subseteq \mathrm{SIZE}(n^{O(1)})$ *then there is a family of polynomial size circuits that, given (the binary encoding of) a formula as input, compute a satisfying assignment for that formula, if such an assignment exists.*

Proof. Suppose that $\{C_n\}_{n\in\mathbb{N}}$ is a family of polysize circuits computing SAT, such that for every formula φ of size n, $C_n(\varphi) = 1$ if and only if $\varphi \in$ SAT. Given a formula $\varphi = \varphi(x_1, \ldots, x_k)$ of size n, compute a satisfying assignment as follows. First see if there is a satisfying assignment starting with 1 by checking if $C_n(\varphi(1, x_2, \ldots, x_k)) = 1$. Continue inductively as follows: Given the first $i-1$ bits $b_1, \ldots, b_{i-1}$ of a satisfying assignment, see if we can extend it with $b_i = 1$ by checking if

$$C_n(\varphi(b_1, \ldots, b_{i-1}, 1, x_{i+1}, \ldots, x_k)) = 1.$$

By putting these k circuit computations together in series we obtain one big (but still polynomial) circuit C such that $\varphi \in$ SAT if and only if $C(\varphi)$ satisfies φ, for every φ of size n. □

Theorem 7.4.2. (Karp-Lipton-Sipser [24]) *If* NP $\subseteq$ SIZE($n^{O(1)}$) *then* $\Pi_2^p \subseteq \Sigma_2^p$.

Proof. Suppose that $L \in \Pi_2^p$. By the NP-completeness of SAT there is a polynomial time computable function f such that for all x,

$$x \in L \iff \forall^p y\ f(x, y) \in \text{SAT}.$$

Without loss of generality we may assume that, for fixed x, the formulas $f(x, y)$ all have the same polynomial size. By Lemma 7.4.1 we have a polysize circuit C such that $f(x, y) \in$ SAT if and only if the assignment $C(f(x, y))$ satisfies $f(x, y)$. Hence there is a polynomial q such that

$$x \in L \iff \exists^q C \forall^p y \big(C(f(x, y)) \text{ satisfies } f(x, y)\big)$$

which shows that $L \in \Sigma_2^p$. □

In particular, if NP has small circuits then PH collapses, cf. Exercise 7.6.7. It also follows that *it is unlikely that randomization will help in solving* NP*-complete problems*, cf. Exercise 7.6.8.

7.5 Sparse T-complete sets

Theorem 7.5.1. $\text{P/poly} = \bigcup \big\{P(S) : S \text{ sparse}\big\}$.

Proof. Suppose $A \in$ P/poly, say $B \in$ P and f is a polynomially bounded advice function such that

$$x \in A \iff \langle x, f(|x|)\rangle \in B. \tag{7.1}$$

We code the advice function into a sparse set S as follows:

$$S = \big\{\langle 0^n, x\rangle : x \sqsubseteq f(n)\big\}.$$

S is indeed sparse: the strings of a given length m are of the form $\langle 0^n, x\rangle$ with $n \leqslant m$ and $|x| \leqslant m$, and for a given n there are at most m such x, so in total there are no more than m^2 strings of length m in S. Clearly $A \leqslant_T^p S$: Given x, first use S to find $f(|x|)$, and then apply (7.1). This takes a polynomial number of steps since f is polynomially bounded.

For the converse, suppose that S is sparse and $A \leqslant_T^p S$. Suppose that $A = M^S$ and that M runs in time p. Then on input x, M can only query elements up to length $p(|x|)$. We can code the polynomial many elements in $S \cap \{0,1\}^{\leqslant p(|x|)}$ into an advice string $f(|x|)$ of polynomial length. In the computation $M^S(x)$, instead of using the oracle S we can equivalently use the advice string $f(|x|)$. □

We saw in Exercise 5.7.6 that there are no sparse p-m-complete sets for EXP. As a consequence of the previous theorem we have:

Theorem 7.5.2. *If there is a sparse p-T-hard set for* EXP *then*

(i) PH *collapses, and*

(ii) P $\neq$ NP.

Proof. (i) From the assumption and Theorem 7.5.1 we have EXP $\subseteq$ P/poly. By Exercise 7.6.11 we then have EXP $\subseteq \Sigma_2^p$. In particular, since PH $\subseteq$ EXP, we have PH $= \Sigma_2^p$.
(ii) If P = NP then P $= \Sigma_2^p$, hence by the proof of (i) we have EXP = P, contradiction. □

7.6 Exercises

Exercise 7.6.1. The Boolean XOR operator is defined as follows: xXORy is 1 when exactly one of x and y is 1, and it is 0 otherwise.
(a) Show that xXORy is equal to $x + y$ mod 2.
(b) Define XOR in terms of the usual propositional connectives.
(c) Define a circuit computing XOR.
(d) Show that adding XOR-gates to the definition of circuits only changes the notion of circuit size by a constant factor.

Exercise 7.6.2. The n-ary *parity function* $\text{parity}_n : \{0,1\}^n \to \{0,1\}$ outputs 1 if the input string contains an odd number of 1's, and 0 otherwise. Note that the 2-ary parity function is equal to the XOR operator from Exercise 7.6.1. Show that there is a circuit of size $O(n)$ that computes parity_n.

Exercise 7.6.3. As noted above, circuit size is a nonuniform measure of complexity. Show that indeed the class SIZE($s(n)$) contains noncomputable sets, even if s is constant.

Exercise 7.6.4. Improve on Exercise 7.6.3 by showing that SIZE(1) is uncountable.

Exercise 7.6.5.* Finish the proof of Theorem 7.3.2. (For a complete description see Vol. I of [4]. Though conceptually straightforward, the full proof is a significant amount of work.)

Exercise 7.6.6. Draw a picture of the circuit from Lemma 7.4.1.

Exercise 7.6.7. Show that if $\Pi_2^p \subseteq \Sigma_2^p$ then $\mathrm{PH} = \Sigma_2^p$.

Exercise 7.6.8. Show that if NP $\subseteq$ BPP then PH collapses. (This shows that it is unlikely that randomized algorithms can solve all problems in NP. See also Exercise 6.5.10.)

Exercise 7.6.9. Show that P/poly is closed downwards under $\leqslant_T^p$.

Exercise 7.6.10. Prove that $\mathrm{P/poly} = \bigcup \{P(S) : S \text{ tally}\}$.

Exercise 7.6.11.* (Meyer, cf. [24]) Prove that if EXP $\subseteq$ P/poly then EXP $\subseteq \Sigma_2^p$.

Exercise 7.6.12. There is a sparse p-T-hard set for EXP if and only if EXP $\subseteq$ P/poly.

Chapter 8

Cryptography

We tend to think of complexity as something *negative* entailing that we cannot do something. In this chapter we will see that sometimes complexity can be something *positive* that we can use to our advantage. Cryptography is the prime example of such a topic. This is in fact one of the oldest fields of applied mathematics, that has stirred the imagination since ancient history. The main problem is to devise a secure and efficient method to transfer secret messages. To complicate the task, we require that the method should also work in case the parties that want to exchange messages have never met before, so that they have not had the chance of agreeing in private on some key to be used. We will see that such a method indeed exists, based on a complexity-theoretic assumption.

8.1 Public-key cryptography

In public-key cryptography, every participant has a public key known to everybody and a private key only known to himself. If participant Alice wants to send participant Bob a secret message over a, presumably unsafe, channel, she encrypts the message using Bob's public key. The protocol is devised in such a way that the only way to recover the message is to use Bob's private key. This ensures that only Bob can decrypt the message, and that a possible eavesdropper cannot decipher it.

RSA (which stands for the names of its inventors Rivest, Shamir, and Adleman) is such a public-key encryption protocol. In 1977 the authors of RSA published a challenge to decipher a message that was encrypted using their protocol. It took until 1994 until the message was deciphered, in a large effort using massive parallel computing, coordinated by Arjen Lenstra and others. The message turned out to be "The Magic Words are Squeamish Ossifrage".

While it is easy to multiply numbers, there does not seem to be an efficient way of factoring a large number into its prime factors. RSA is based on the

assumption that there is no polynomial time algorithm for factoring. (In contrast, note that we can decide in polynomial time whether a number is prime, see p 11.) The RSA protocol works as follows. As before, φ denotes Euler's function.

- Choose two large prime numbers p and q.
- Choose an integer e coprime with $\varphi(pq)$.
- Let d be a multiplicative inverse of e modulo $\varphi(pq)$ (which exist by the previous item), that is, $ed \equiv 1 \bmod \varphi(pq)$. Note that d can be easily computed using the Euclidean algorithm, provided we know p and q, and hence $\varphi(pq) = (p-1)(q-1)$.
- Public key: pq and e
- Private key: p, q, and d
- Encryption algorithm: $E(e, x) = x^e \bmod pq$
- Decryption algorithm: $D(d, y) = y^d \bmod pq$

Proposition 8.1.1. *For any integer x,*

$$x^{ed} \equiv x \ mod\ pq. \tag{8.1}$$

Proof. If $\gcd(x, p) = 1$ then by Fermat's little theorem (cf. section 1.6), for any k we have

$$x^{1+k\varphi(pq)} \equiv x^{1+k(p-1)(q-1)} \equiv x \bmod p. \tag{8.2}$$

On the other hand, (8.2) clearly also holds if $x \equiv 0 \bmod p$, hence it holds for *any* x. Similarly, $x^{1+k\varphi(pq)} \equiv x \bmod q$ for any x. It follows by the Chinese remainder theorem that

$$x^{1+k\varphi(pq)} \equiv x \bmod pq \tag{8.3}$$

for any integer x. Since $ed \equiv 1 \bmod \varphi(pq)$, (8.1) follows from this. □

From Proposition 8.1.1 we see that the decryption algorithm D is indeed the inverse of the encryption algorithm E. In particular, since E has an inverse, it is a bijection on $\mathbb{Z}/pq\mathbb{Z}$.

Note further that anybody can encrypt a message, since pq and e are public, but that one needs d for decryption, where d is obtained from p and q. Since pq is public, it is essential that there is no efficient way to obtain p and q from their product pq. It is currently widely believed, but unproven, that such a method indeed does not exist.

Note using the above protocol we can encrypt messages of length at most pq. Longer messages need to be chopped up in pieces of this length. In the original RSA challenge, the primes p and q were 64 and 65 digits long.

8.2 Signed messages

Suppose Alice wants to send Bob a secret message, and that in addition she wants to *prove* to Bob that she is the sender. She can use her own pair of public and private keys e_A and d_A to do so in the following way. Let e_B and d_B denote Bob's pair of keys. Instead of sending the encrypted message x^{e_B} as before, Alice now sends the message $x^{d_A \cdot e_B}$ to Bob. Bob decodes the message to x^{d_A} using his private key d_B. Then he uses Alice's public key e_A to further decode the message to x. At this point Bob not only knows the message x, but also that the message *had to come from Alice*, since Alice is the only person that knows d_A.

The dynamics of the interaction between Alice and Bob in this protocol reveals some of the power of interactive cryptographic protocols that we will encounter again later in Chapter 9. Note that the introduction of interaction between various parties that perform computations adds a whole new layer of complexity to the theory of computation. The computational as well as the logical aspects of this are highly interesting.

8.3 One-way functions

Informally, a *one-way* (or *trapdoor*) function is a function that is easy to compute, but hard to invert. Such functions may serve as the basis for a cryptographic scheme.

Note first that some easily computable functions may be hard to invert for trivial reasons. For example, if the function is not injective then there is no inverse at all. We may counter this by requiring that an inverse should compute *some* inverse value, that need not be unique. A second trivial reason that a function may not be easily invertible is that it might map large strings to small ones. Consider for example $f(x) = \log x$. Since we have $f(2^n) = n$, to write down the inverse $f^{-1}(n)$ we need to write 2^n, which is of length n, which is exponential in $\log n$, the input size of $f^{-1}(n)$. So f^{-1} is not polynomial time computable. Functions that do not map large strings to small ones, i.e. such that the length of inputs and outputs are polynomially related, are called *honest*. So in studying invertability, we should restrict our attention to such honest functions.

We have the following formal definition. A function $f : \{0,1\}^* \to \{0,1\}^*$ is a *one-way function* if

(i) f is injective. (As indicated above, this requirement is not always imposed.)

(ii) f is honest: There is a constant k such that for all x,

$$|x| \leqslant |f(x)|^k.$$

(iii) f is polynomial time computable.

(iv) f^{-1} is not polynomial time computable.

Note that the existence of one-way functions implies that P $\neq$ NP. (Because of the honesty, computing the – unique – inverse of $f(x)$ is a typical NP -task: we can guess an inverse and use f to check its correctness.) Even if we assume that P $\neq$ NP, however, the existence of one-way functions is not known. Their existence is tied to a special complexity class that we discuss in the next section

The RSA protocol is based on the assumption that multiplication, which has factoring as an inverse, is a trapdoor function. There are many other such functions. An example is the discrete exponential function, and its inverse the discrete logarithm. Given a prime p and an integer x, the discrete exponential is 2^x mod p. This function can be computed in time $O(n^3)$, where n is the length of p. Currently no polynomial time algorithm is known for computing the inverse of this function.

8.4 The class UP

Definition 8.4.1. Call a nondeterministic Turing machine *unambiguous* if for every input there is at most one accepting computation. UP is the class of sets that are accepted by some unambiguous machine in polynomial time.

Obviously we have the inclusions

$$\mathrm{P} \subseteq \mathrm{UP} \subseteq \mathrm{NP}.$$

Nothing more than this is known. The following result ties the class UP to the existence of one-way functions.

Theorem 8.4.2. *One-way functions exist precisely when* P $\neq$ UP*.*

Proof. Suppose that f is a one-way function. Define the set

$$L = \big\{(x, y) : \exists z \leqslant x(f(z) = y)\big\}.$$

Since f is injective, $L \in$ UP. Also $L \notin$ P: If $L \in$ P then we could find an inverse of f in polynomial time by binary search.

Conversely, if L is a set in UP $-$ P then we can define a one-way function f as follows. Suppose M is an unambiguous machine accepting L. Given a computation path y of M on input x, we let f map y to $1\widehat{\ }x$ if y is accepting, and to $0\widehat{\ }y$ if y is not accepting. Note that x can be effectively retrieved from y. Since accepting paths of M are unique, f is injective. Also, f is honest since the computations of M are of polynomial length. So to prove

that f is one-way it suffices to show that its inverse cannot be computed in polynomial time. (Note that f is not surjective, so the inverse is only defined on the range of f.) Suppose we could do this. Then we could decide L in polynomial time as follows: Given x, compute $f^{-1}(1\widehat{\ }x)$. If this yields a y we know that $x \in L$, and if not we know that $x \notin L$. This contradicts the assumption that $L \notin \mathrm{P}$, hence f is one-way. □

8.5 Exercises

Exercise 8.5.1. Strictly speaking, in the RSA protocol above we only need to know $\varphi(pq)$ and not p and q to compute d. Show however that if we know both pq and $\varphi(pq)$ then we also know p and q, so obtaining $\varphi(pq)$ from pq is just as hard as to obtain p and q.

Exercise 8.5.2. Show that UP is downwards closed under p-m-reductions.

Chapter 9

Interactive proofs

9.1 Interactive protocols and the class IP

We can view NP as the class of languages with short certificates. Think of this as an unbounded *Prover* P convincing a polynomial time bounded *Verifier* V. It is not clear how to describe e.g. co-NP in this way. However, the idea still works if we add rounds between P and V and make V probabilistic.

Example: Graph Nonisomorphism. Recall the problem GRAPH ISOMORPHISM of deciding whether two given graphs are isomorphic. This is a famous example of an NP-problem not known to be in P or NP-complete. We describe an interactive protocol for its complement Graph Nonisomorphism. Given G_0 and G_1, how can Prover convince Verifier that $G_0 \ncong G_1$? Note that Prover cannot send a short certificate, since Graph Nonisomorphism is not known to be in NP. Instead, Verifier probabilistically tests Prover as follows. Verifier randomly selects G_0 or G_1 and randomly permutes its nodes to obtain a graph H, which he sends to Prover with the question which of G_0, G_1 was the source of H. This is repeated for a polynomial, or constant, number of rounds, say 100 times. If indeed $G_0 \ncong G_1$ then Prover can always reconstruct the answer, using his unbounded computing power. If on the other hand $G_0 \cong G_1$ he has no better than a chance of $\frac{1}{2}$ to answer correctly. So after 100 rounds, if all of the answers of Prover are still correct, $G_0 \ncong G_1$ with probability at least $1 - \frac{1}{2^{100}}$. In summary, if P does his best to convince V that $G_0 \ncong G_1$ he can always succeed in doing so if indeed $G_0 \ncong G_1$, whereas he will fail almost surely if $G_0 \cong G_1$.

The complexity class IP consists of all languages that allow for a protocol of the type just described. In the following we make this more precise. The protocol will consist of several rounds of sending messages between Prover P and Verifier V, and at every round there will be a history string $\langle m_1, \ldots, m_i \rangle$ of messages of previous rounds.

- Verifier V is a polynomial time computable function that is given the input w of length n, as well as a random string r of length $p(n)$, where p is a fixed polynomial. Given message history $\langle m_1, \ldots, m_i \rangle$, V either accepts, rejects, or sends a next message m_{i+1}.
- Prover P is an unrestricted function that is given the input, and that outputs a message m_{i+1} when given history $\langle m_1, \ldots, m_i \rangle$.
- Define the interaction $V \leftrightarrow P$ by setting $(V \leftrightarrow P)(w, r) = 1$ if there is a legal sequence of messages $\langle m_1, \ldots, m_k \rangle$ between V and P for some k such that m_k is "accept".

We also require that all messages have length at most $p(n)$, and that the number of rounds k is bounded by $p(n)$. Define

$$\Pr\big[V \leftrightarrow P \text{ accepts } w\big] = \Pr_{|r|=p(n)}\big[(V \leftrightarrow P)(w, r) = 1\big].$$

Definition 9.1.1. We define $A \in \mathrm{IP}$ if there exist a polynomial time computable verifier V such that for all inputs w,

$$w \in A \implies \exists P \Pr\big[V \leftrightarrow P \text{ accepts } w\big] \geqslant \tfrac{2}{3},$$
$$w \notin A \implies \forall P \Pr\big[V \leftrightarrow P \text{ accepts } w\big] \leqslant \tfrac{1}{3}.$$

As for BPP, we can boost the probability by polynomial iterations to make the error exponentially small. Note that we have $\mathrm{NP} \subseteq \mathrm{IP}$ (P supplies V with the certificate in just one round) and $\mathrm{BPP} \subseteq \mathrm{IP}$. For the latter no interaction is needed; given w and r, V simply performs a polytime computation and accepts or rejects.

Theorem 9.1.2. $\mathrm{IP} \subseteq \mathrm{PSPACE}$.

Proof. Though it is tedious to work out the details, this is a straightforward derandomization of all possibilities in polyspace. We can get rid of the Prover P by considering

$$\Pr\big[V \text{ accepts } w\big] = \max_P \Pr\big[V \leftrightarrow P \text{ accepts } w\big].$$

Note that this value is $\geqslant \frac{2}{3}$ if $w \in A$ and $\leqslant \frac{1}{3}$ if $w \notin A$. We can define a similar expression starting with an arbitrary message history. □

9.2 IP = PSPACE

Towards the converse inclusion of Theorem 9.1.2, consider the following variant of the problem SA from page 43:

$$\#\mathrm{SAT} = \big\{\langle \varphi, k \rangle : \varphi \text{ has exactly } k \text{ satisfying assignments}\big\}.$$

Theorem 9.2.1. #SAT ∈ IP.

Proof. (Sketch.) Suppose $\varphi = \varphi(x_1, \ldots, x_m)$ is a Boolean formula and let $f_i(a_1, \ldots, a_i)$ be the number of satisfying assignments of φ where $x_j = a_j$ for $j \leqslant i$. Hence $f_0()$ is the number of satisfying assignments of φ. Then we have

$$f_i(a_1, \ldots, a_i) = f_{i+1}(a_1, \ldots, a_i, 0) + f_{i+1}(a_1, \ldots, a_i, 1).$$

Consider the following (not yet polynomial) protocol for #SAT. The input is a pair $\langle \varphi, k \rangle$, where $k \in \mathbb{N}$ is supposed to be the number of assignments of φ.

Round 0. P sends $f_0()$ to V, and V checks whether $f_0() = k$, and rejects if not.

Round 1. P sends $f_1(0)$ and $f_1(1)$ to V. V checks $f_0() = f_1(0) + f_1(1)$ and rejects if not.

Round 2. P sends $f_2(0,0)$, $f_2(0,1)$, $f_2(1,0)$, $f_2(1,1)$ to V. V checks whether

$$\begin{aligned} f_1(0) &= f_2(0,0) + f_2(0,1), \\ f_1(1) &= f_2(1,0) + f_2(1,1) \end{aligned}$$

and rejects if not.

Round m. P sends $f_m(a_1, \ldots, a_m)$ for all $a_1, \ldots, a_m \in \{0,1\}$. V checks 2^m equations and rejects if any of them fail.

Round m + 1. If all $f_m(a_1, \ldots, a_m)$ are correct V accepts, and V rejects otherwise.

This protocol for #SAT is obviously correct, but the problem is that it is not polynomial: the rounds are exponential because at every next round the number of equations and the size of the messages doubles. To remedy this, write the f_i's as polynomials over a finite field $\mathbb{F}_q$, $q \geqslant 2^n$ prime, where n is the size of φ. Such polynomials are easily defined from φ by writing $\wedge$ as multiplication and $x \vee y$ as

$$x * y = 1 - (1 - x)(1 - y). \tag{9.1}$$

(We could use + instead of $*$, but $*$ has the advantage that it behaves the same as $\vee$ on Boolean values.) Now, in Round 1 of the protocol, instead of $f_1(0)$ and $f_1(1)$, choose a random $r \in \mathbb{F}_q$ and consider $f_1(r)$. If adversary P lies about $f_0()$ then it also has to lie about at least one of $f_1(0)$ and $f_1(1)$, hence about the polynomial $f_1(z)$. Say it sends $\tilde{f}_1(z) \neq f_1(z)$. The key point of the proof then is that for random r, $\tilde{f}_1(r) \neq f_1(r)$ with high probability, because two different polynomials of degree $\leqslant n$ can agree on at most n points, for a nonzero polynomial of degree n has at most n roots.[1] So working with random inputs for the f_i's prevents the blow-up in the above protocol, and brings it down to polytime. □

[1] The precise formulation of this fact for multivariate polynomials is called the Schwartz-Zippel theorem, cf. [38, p45], [31, p29].

Theorem 9.2.2. (Shamir) IP = PSPACE.

Proof. One inclusion was already proven in Theorem 9.1.2, so we only have to prove PSPACE $\subseteq$ IP. We try to repeat the proof of #SAT $\in$ IP with the PSPACE-complete problem QBF instead of #SAT. Without loss of generality we can work with closed q.b.f.'s (no free variables) since this subset of QBF is still PSPACE-complete. Given a closed q.b.f. $\varphi = Q_1 x_1 \dots Q_m x_m \psi$, where $Q_i \in \{\forall, \exists\}$, define

$$f_i(a_1, \dots, a_i) = \begin{cases} 1 & \text{if } Q_{i+1} x_{i+1} \dots Q_m x_m \psi \text{ is true,} \\ 0 & \text{otherwise.} \end{cases}$$

$f_0()$ is the truth value of φ. Again we can find polynomial expressions for the f_i's. We now have the identities

$$Q_{i+1} = \forall : \quad f_i(a_1, \dots, a_i) = f_{i+1}(a_1, \dots, a_i, 0) \cdot f_{i+1}(a_1, \dots, a_i, 1), \quad (9.2)$$
$$Q_{i+1} = \exists : \quad f_i(a_1, \dots, a_i) = f_{i+1}(a_1, \dots, a_i, 0) * f_{i+1}(a_1, \dots, a_i, 1), \quad (9.3)$$

where $*$ was defined in (9.1). Now we run the same protocol as in the proof of Theorem 9.2.1. A new problem is that in the identities (9.2) and (9.3) every Q_i potentially doubles the degree of the polynomial (so that they may grow exponential), whereas it is essential in the proof of Theorem 9.2.1 that we work with polynomials of a degree bounded by a fixed $n \approx \text{size}(\varphi)$. To keep the degree small we perform the following trick. Given a polynomial $f(x)$ in variable x, note that

$$\hat{f}(x) = (1 - x) f(0) + x f(1)$$

is linear in x and coincides with f on Boolean values of x. If every time (9.2) or (9.3) is applied we first perform this transformation for every x_j, $j \leqslant i$ (i.e. those variables whose degree might go up) then the degree is kept bounded by 2, without changing the result on Boolean values. The obstacle thus being removed, the rest of the proof is the same as before. □

A full account of the proof of Theorem 9.2.2 can be found in Sipser [32].

9.3 Zero-knowledge proofs

Using public-key cryptography and interactive proofs, it is possible to convince someone that you are in possession of a proof *without actually giving anything away about it*. Such proofs (which are actually interactive procedures) are called *zero-knowledge proofs*.

The following example (due to M. Blum [7]) may serve to illustrate the idea. Consider the NP-complete problem 3COLORING. Suppose Alice wants

to convince Bob that a given graph $G = (V, E)$ is 3-colorable. She could of course do this by giving Bob a 3-coloring after which Bob could efficiently check that it is correct, but Alice wants to keep her 3-coloring secret and still convince Bob that she has it. This seemingly impossible task can be achieved by the following interactive protocol.

In every round Alice chooses a random permutation π of the 3 colors, and for every node i chooses an RSA tuple (p_i, q_i, e_i, d_i). For every i she announces to Bob $(p_i q_i, e_i)$ and the encoded version of $\pi(\chi(i))$, the color of i after applying π. At this point Bob cannot do anything with this information since he does not have the decoding key d_i, but Alice has made a commitment, albeit encrypted, as to what the colors are. Next, it is Bob's turn to test Alice by picking at random an edge (i, j) of the graph, after which Alice reveals to Bob the secret keys d_i and d_j that allow Bob to decode $\pi(\chi(i))$ and $\pi(\chi(j))$ and see if they are different, as they should be. This ends one round of the protocol.

Now if Alice uses indeed a 3-coloring of the graph, Bob will always be satisfied. If not, in every round there is an edge (i, j) where $\pi(\chi(i)) = \pi(\chi(j))$. There are $|E|$ edges, so in every round Bob has a chance at least $\frac{1}{|E|}$ of discovering a mistake. By iterating a polynomial number of times this chance becomes exponentially close to 1.

Let us check that Bob has indeed learned nothing about Alice's 3-coloring in the course of this procedure. In every round Bob receives a number of keys and a pair of colors. But these are randomly permuted versions of the original colors, hence nothing more than a random pair of different colors. Since in the next round Alice chooses another random permutation, Bob might just as well flip a coin himself to generate such random pairs, and he learns nothing from this.

The NP-completeness of 3COLORING can be used to show that all problems in NP have zero-knowledge proofs, cf. [17].

9.4 Exercises

Exercise 9.4.1. Show that the class IP does not change if we require that the prover P is in EXP instead of unbounded.

Exercise 9.4.2. The original proof of Theorem 9.2.2 employed the notion of a simple formula. A first-order formula φ is *simple* if no occurrence of a variable is separated by more than one universal quantifier from its point of quantification. Show that by introducing new variables every φ can be effectively transformed to an equivalent simple formula. How many new variables are needed at most?

Chapter 10

Approximations

Since there is an abundance of computational problems that we do not know how to handle directly, it makes sense to ask for approximate solutions. In this chapter we study the extent to which hard computational problems can be effectively approximated.

10.1 Approximating the traveling salesman

Recall the problem TSP from section 2.5. In general this problem is NP-complete (Corollary 3.4.2), so we have currently no feasible way to deal with the problem in general. Does it become easier when we ask for approximate solutions? In particular, say we only ask for a polynomial time algorithm M that, given an instance x of TSP, always returns a solution $M(x)$ that is within a factor r of an optimal solution $S(x)$, i.e. such that

$$c(M(x)) \leqslant r \cdot c(S(x)), \tag{10.1}$$

where c is the length of the tour and $r \in [1, \infty)$ is fixed. One can think of many situations where having such an algorithm for some not too large factor r would be quite helpful. The bad news is that no matter how large we choose r, such an algorithm does not exist, unless we could already solve the original problem in polynomial time!

Theorem 10.1.1. (Sahni and Gonzalez) *Let $r \geqslant 1$ be a constant. There is no polynomial time algorithm M that approximates* TSP *to within a factor r, as in* (10.1)*, unless* P = NP*.*

Proof. Suppose that such an M exists. We show how to solve the NP-complete problem HAMILTON CIRCUIT in polynomial time. The idea is very similar to that of Corollary 3.4.2. Given a graph $G = (V, E)$ we use M to decide whether G has a Hamilton circuit. Define an instance I of TSP

as follows. The cities are the points in $V = \{1, \ldots, n\}$, and we define their distances by

$$d_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E, \\ rn & \text{otherwise.} \end{cases}$$

Note that if G has a Hamilton circuit then I has a tour of length n, and that if G does not have a Hamilton circuit then every tour in I has length at least $rn + (n-1)$, which is at least $rn + 1$ assuming that $n \geqslant 2$, which we may assume since the case $n = 1$ is trivial. Now apply M to the instance I. Then G has a Hamilton circuit if and only if the length of the tour $M(I)$ is at most rn. □

10.2 Approximation algorithms

In general, for every instance x of an optimization problem A we have a possible set of solutions $F(x)$ that we try to minimize or maximize. This means that every possible solution $s \in F(x)$ has a certain cost $c(s)$. In case of a minimization problem, the optimum cost is then defined as

$$\mathrm{OPT}(x) = \min_{s \in F(x)} c(s).$$

(In case of a maximization problem of course we take the maximum.)

Definition 10.2.1. An ε-approximation algorithm M for a minimization problem A is an algorithm that for each instance x always returns a solution $M(x) \in F(x)$ such that

$$\frac{c(M(x)) - \mathrm{OPT}(x)}{c(M(x))} \leqslant \varepsilon. \tag{10.2}$$

The ratio on the left is called the *relative error* of $M(x)$. The *approximation threshold* of A is the greatest lower bound of all $\varepsilon > 0$ such that there exists a polynomial time ε-approximation algorithm for A. Note that this threshold is always a number in the interval $[0, 1]$.

With the above definitions, we can reformulate Theorem 10.1.1 as follows:

Theorem 10.2.2. *The approximation threshold of* TSP *is 1, unless* P = NP.

Proof. Suppose that M is a polynomial time ε-approximation algorithm for TSP. Then by (10.2) we have $c(M(x)) - \mathrm{OPT}(x) \leqslant \varepsilon \cdot c(M(x))$, hence

$$c(M(x)) \leqslant \frac{1}{1-\varepsilon}\mathrm{OPT}(x).$$

By Theorem 10.1.1 this cannot hold for any $\varepsilon < 1$. □

Definition 10.2.3. A *polynomial time approximation scheme* (PTAS) for an optimization problem is an algorithm that for each $\varepsilon > 0$ and each instance x outputs a solution with relative error at most ε, as in (10.2), and that runs in time polynomial in $|x|$. If the running time is also polynomial in $\frac{1}{\varepsilon}$ then the scheme is called *fully polynomial*.

Definition 10.2.3 points to an interesting feature, namely that computational problems may have several parameters, and that algorithms to solve them may be polynomial in some of parameters but not in others. The field in which this phenomenon is systematically explored is called *parameterized complexity theory*, and the standard textbook on this subject is Downey and Fellows [14].

The reader should note that approximability is *not* preserved by p-m-reductions: Some NP-complete problems, such as TSP, do not have a PTAS (unless P = NP, cf. Theorem 10.2.2), whereas others, such as KNAPSACK, do have one. It is possible to devise a type of reduction that preserves approximability, and to develop the theory of approximation problems analogous to the theory of NP-completeness.

10.3 Probabilistically checkable proofs

In this section we discuss the notion of probabilistically checkable proof, and the main theorem about this notion, the PCP Theorem. Although we will not prove this theorem, we discuss some of its consequences for the theory of approximations, where the theorem has had a profound influence.

Suppose that L is a language in NP. By Proposition 4.2.4 there is a polynomial time predicate R such that

$$L = \big\{x : \exists^{p(|x|)} y\ (x, y) \in R\big\}.$$

This implies the existence of a *verifier* V for L, that, given any input x and any certificate y of length at most $p(|x|)$, checks whether y is indeed proof of the fact that $x \in L$ or not. The verification can be done in polynomial time, and in principle uses all of the bits of y. In the following we will see that we can greatly improve on the number of bits examined if we are willing to make V probabilistic.

Definition 10.3.1. A *probabilistic verifier* is a probabilistic algorithm (in the sense of section 6.1) that given an input x and a *proof* π as oracle, examines the proof and accepts or rejects it with a certain probability. Given functions $r(n)$ and $q(n)$, a language L is in the class PCP$[r(n), q(n)]$ if there is a polynomial time probabilistic verifier V that on an input x of length n uses $r(n)$ random bits and at most $q(n)$ nonadaptive queries to the oracle π such that

$$x \in L \Leftrightarrow \exists\pi \Pr\big[V^\pi(x) = 1\big] = 1,$$
$$x \notin L \Leftrightarrow \forall\pi \Pr\big[V^\pi(x) = 1\big] \leqslant \tfrac{1}{2}.$$

Here the probabilities are taken over all choices of the $r(n)$ random bits.

The following theorem is one of the landmarks of the subject. It was the culmination of a series of papers by a large number of authors, with Sanjeev Arora as one of the main contributors. For more information about the history of the result we refer to [30].

Theorem 10.3.2. (PCP Theorem) *There exists a constant $q \in \mathbb{N}$ such that* $\mathrm{NP} = \bigcup_{c \geqslant 1} \mathrm{PCP}[c \log n, q]$.

Hence in order to verify with high probability that a given instance x is in some NP-language, we only need to examine a *constant* number of bits of the proof, regardless the length of x. Furthermore, the constant is the same for all languages in NP. The proof of Theorem 10.3.2 is too long and complicated to discuss here. By now there is an somewhat easier proof by Irit Dinur available. We recommend the exposition of Dinur's proof in [30]. Despite the various improvements in presentation, the proof remains beyond the scope of these course notes.

10.4 PCP theory and nonapproximability

Consider the following optimization problem MAX3SAT:
INPUT: a 3CNF formula φ (with three literals per clause),
QUESTION: find an assignment satisfying the largest number of clauses of φ.
Since 3SAT is NP-complete by Theorem 3.3.4, MAX3SAT is not solvable in polynomial time, unless P = NP.

Proposition 10.4.1. *The approximation threshold of* MAX3SAT *is at most* $1/2$.

Proof. Given a formula in CNF, consider the all-zero and all-one assignments. Since every clause is satisfied by at least one of these, one of them satisfies at least half of the clauses, so pick one that does this. □

In fact, it is possible to improve Proposition 10.4.1 from $\frac{1}{2}$ to $\frac{1}{8}$, cf. Exercise 10.5.5.

By the following theorem, the approximation threshold of MAX3SAT is bigger than 0. In particular there is no PTAS for MAX3SAT, unless P = NP.

Definition 10.4.2. We say that a CNF formula φ is *ε-far* from satisfiable if for every assignment, at most a fraction $1 - \varepsilon$ of the clauses of φ is satisfied.

Theorem 10.4.3. *There exists $\varepsilon > 0$ such that for every $L \in \mathrm{NP}$ there is a p-m-reduction $x \mapsto \varphi_x$ of L to* 3SAT *such that*

$$\begin{aligned} &x \in L \Leftrightarrow \varphi_x \text{ satisfiable}, \\ &x \notin L \Leftrightarrow \varphi_x \ \varepsilon\text{-far from satisfiable}. \end{aligned} \tag{10.3}$$

Proof. By the PCP theorem, let V be a probabilistic verifier witnessing that $L \in \mathrm{PCP}[O(\log n), O(1)]$. Let $r \in \{0,1\}^{O(\log |x|)}$ range over all the random strings for V. For every r, V makes a constant number of queries to the proof π. Without loss of generality we may assume that the queries are nonadaptive (since an exponential of a constant number of queries is still constant). Suppose that the number of queries is ℓ. For each choice of r, $V^\pi(x)$ depends on a Boolean function $f_r : \{0,1\}^\ell \to \{0,1\}$ describing all possible answers to the ℓ queries. For the translation into a 3CNF-formula, introduce variables $x_1, \ldots, x_k$ that stand for all positions of π that can be queried, i.e. x_i stands for $\pi(i)$. The number k equals the length of the proof π, which is polynomial in $|x|$. We can represent f_r by a CNF-formula ψ_r with 2^ℓ clauses (cf. Proposition 7.1.2) using the variables $x_{q_1}, \ldots, x_{q_\ell}$ corresponding to the queries V makes with r. To convert every clause of length ℓ to a clause with three literals (as in Theorem 3.3.4) we need for every clause ℓ auxiliary clauses, so in total ψ_r has at most $\ell 2^\ell$ clauses. For every r we use a different set of auxiliary variables. Finally, we let $\varphi_x = \bigwedge_r \psi_r$. Now

$$\begin{aligned} x \in L &\Rightarrow \exists \pi \Pr\big[V^\pi(x) = 1\big] = 1 \\ &\Rightarrow \pi \text{ satisfies all clauses of } \varphi_x. \end{aligned}$$

$$\begin{aligned} x \notin L &\Rightarrow \forall \pi \Pr\big[V^\pi(x) = 1\big] \leqslant \tfrac{1}{2} \\ &\Rightarrow \text{for at least half of the } r, \pi \text{ does not satisfy } \psi_r, \text{ i.e.} \\ &\quad \text{at least one clause of } \psi_r \text{ is false,} \\ &\Rightarrow \varphi_x \text{ is } \varepsilon\text{-far from satisfiable, where } \varepsilon = \frac{1}{2}\frac{1}{\ell 2^\ell}. \end{aligned}$$

□

Theorem 10.4.3 is a consequence of the PCP theorem. Interestingly, the converse also holds:

Theorem 10.4.4. *If $L \in$ NP and L has a reduction as in* (10.3) *then $L \in \mathrm{PCP}[O(\log n), O(1)]$.*

Proof. We define a probabilistic verifier V for L as follows. Given x, compute φ_x as in (10.3), and interpret the proof π as an assignment for φ_x. V randomly picks $c = \frac{1}{\varepsilon} \ln 2$ clauses of φ_x and checks whether π satisfies all of them. If so, V accepts x, and V rejects x otherwise. To specify an item in a list of n objects takes $\log n$ bits, and since the number of clauses in φ_x is polynomial in $|x|$ we need $c \log |x|^{O(1)} = O(\log |x|)$ random bits. Checking these clauses takes $O(1)$ bits of π. Now

$$\begin{aligned} x \in L &\Rightarrow \varphi_x \text{ satisfiable} \\ &\Rightarrow \exists \pi \Pr\big[V^\pi(x) = 1\big] = 1 \end{aligned}$$

$$\begin{aligned} x \notin L &\Rightarrow \forall \pi \text{ at least a fraction } \varepsilon \text{ of the clauses of } \varphi_x \text{ is false} \\ &\Rightarrow \forall \pi \Pr\bigl[V^\pi(x) = 1\bigr] \leqslant (1-\varepsilon)^c \end{aligned}$$

To finish the proof, we just have to see that $(1-\varepsilon)^c \leqslant \frac{1}{2}$ for $c = \frac{1}{\varepsilon}\ln 2$, which easily follows from the inequality $1 - x \leqslant e^{-x}$. □

10.5 Exercises

Exercise 10.5.1. $\mathrm{PCP}[0, O(\log n)] = \mathrm{P}$.

Exercise 10.5.2. $\mathrm{PCP}[0, \operatorname{poly} n] = \mathrm{NP}$.

Exercise 10.5.3. Prove the easy direction of Theorem 10.3.2, namely that $\mathrm{PCP}[O(\log n), O(1)] \subseteq \mathrm{NP}$.

Exercise 10.5.4. $\mathrm{PCP}[O(\log n), 1] = \mathrm{P}$.

Exercise 10.5.5. Show that the approximation threshold of MAX3SAT is at most $\frac{1}{8}$. (Hint: Use the *probabilistic method*: Assign the value 1 to every variable x with probability $\frac{1}{2}$ and compute the expected number of satisfied clauses.)

Chapter 11

Proof complexity

In mathematical logic, the complexity of a theorem can sometimes be measured by considering the strength of the axioms that are needed to prove it. There are interesting complexity theoretic analogues of this theme, that we discuss in this chapter. Gödel's classic incompleteness theorem tells us that, no matter how strong our axioms, not every true arithmetical formula is provable. Here we consider the *length* of proofs, and show that it is unlikely that every true propositional formula has a short proof. "Short" here of course means of polynomial size in the length of the formula.

11.1 Propositional proof systems and Cook's program

Recall that SAT is the set of all satisfiable propositional formulas. Consider its companion VALID consisting of all *valid* propositional formulas, that is, those formulas that are valid under all assignments.

By Exercise 4.4.7, VALID is co-NP-complete. Hence, if VALID $\in$ NP then NP = co-NP, something that is believed to be unlikely. This means that, if is indeed true that NP $\neq$ co-NP, *not every valid formula has a short certificate that proves its validity.* This can be seen as a bounded version of Gödel's incompleteness theorem: Not every true propositional formula has a short proof.

Definition 11.1.1. A *propositional proof system* is any polynomial time computable function $F : \{0,1\}^* \to$ VALID. (We can without loss of generality require that F is also surjective, cf. Exercise 11.4.2.) If $F(x) = \varphi$, then x is a proof that φ is valid. The restriction that F be polynomial time computable is natural: given a proof we certainly want to be able to check efficiently that it indeed is a proof.

If NP $\neq$ co-NP then for every propositional proof system F and every constant c there are infinitely many valid formulas φ such that for $|\varphi| = n$, for no $x \in \{0,1\}^{n^c}$ we have $F(x) = \varphi$. So if we ever want to prove that NP $\neq$ co-NP,

we should at least be able to prove this for any given F. *Cook's program*[1] is to come up with stronger and stronger proof systems, and with formulas that have superpolynomial proof complexity in those systems. One nice aspect of this program is that we can measure our progress on such a difficult question as NP $\neq$ co-NP.

Let us start simple. The most naive proof system for propositional logic is the method of truth tables (Truth tables apparently first appeared, in their modern form, in Wittgenstein [40, 4.31 ff].) Since the truth table of a formula with n propositional variables has size $O(2^n)$, *every* proof of validity using truth tables has exponential size, regardless which formula we are proving. So we have made our first step in Cook's program, namely we have proven an exponential lower bound for the proof system of truth tables.

Of course the problem becomes more difficult for more sophisticated proof systems F. The problem is open for several of the standard proof systems for classical propositional logic. In section 11.3 we consider proofs based on resolution. One possibility for making some progress is to consider proof systems that are *weaker* than those for classical logic. An obvious candidate is to consider *intuitionistic logic*. A breakthrough result for this logic was obtained by Hrubeš [20]. He showed that in a proof system for intuitionistic propositional logic, exponential lower bounds for certain tautologies can indeed be proven. (N.B. The set of intuitionistic validities is PSPACE-complete, and it is a subset of VALID.) Hrubeš uses a tautology similar to the pigeonhole principle discussed below, namely k-colorability: a complete graph on $k+1$ nodes is not k-colorable. This has a short classical proof, but it does not have a short proof in an intuitionistic Frege system.

11.2 The pigeonhole principle

As discussed above, part of the task in Cook's program is, given a proof system F, to come up with candidate formulas that are supposed to only have superpolynomial proofs. For some time it was hoped that the combinatorial statement called the *pigeonhole principle* would provide such formulas. This statement says that if we have m pigeons that we have to put into n holes, and $m > n$, then there is a pigeonhole containing more than one pigeon. By Exercise 11.4.3, the pigeonhole principle can be formulated as a Boolean formula PHP_n^m. The hope that the formulas PHP_n^m would only have superpolynomial proofs was crushed when Buss [9] showed that in fact they do have polynomial size proofs, in a certain Hilbert style proof system for classical logic. Such systems, based on modus ponens, are called *Frege systems* in this context. The formulas PHP_n^m can be used, however, to show an

[1] The connection between proof systems and NP $\neq$ co-NP was first made in Cook and Reckhow [13]. For a recent survey see Buss [10].

exponential lower bound for resolution, see the proof of Theorem 11.3.4.

11.3 Resolution

In this section we discuss a simple propositional proof system, namely resolution. In resolution there is only one rule, which speeds up the search for proofs, but the method only works for formulas in CNF. (Recall that we have seen in section 3.3 that converting formulas to CNF may be expensive. Recall also that the CNF fragment of SAT is still NP-complete.) Resolution is also the method underlying the declarative programming language Prolog.

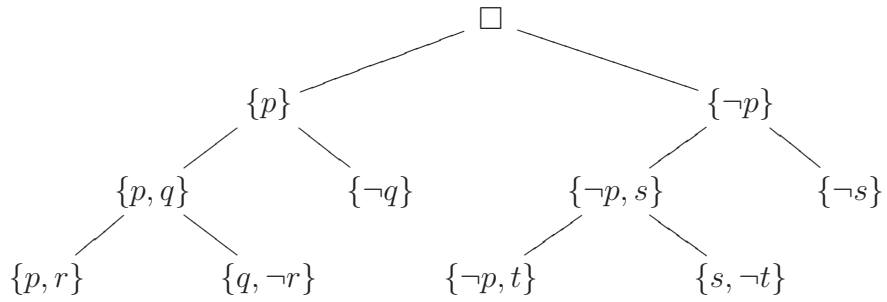
In the following we use the terminology from section 1.5.

Definition 11.3.1. If $C_1 = \{l\} \sqcup C_1'$ and $C_2 = \{\bar{l}\} \sqcup C_2'$ are clauses, where $\sqcup$ denotes that we are taking a union of disjoint sets, then $C_1' \cup C_2'$ is called a *resolvent* of C_1 and C_2.

Note that resolution is sound, i.e. preserves satisfiability. That is, if both of the parent clauses are satisfiable then also their resolvent is satisfiable. A resolution *proof* of a clause C from a formula S is a finite sequence $C_1, C_2, \ldots, C_n = C$ of clauses such that each C_i is an element of S or a resolvent of clauses earlier in the sequence. If such a proof exists we write $S \vdash_{\mathcal{R}} C$. If $S \vdash_{\mathcal{R}} \square$ we say that S is *refutable*.

We can in principle picture resolution proofs as binary trees. For example, the following is a refutation proof from the set

$$S = \big\{\{p, r\}, \{q, \neg r\}, \{\neg q\}, \{\neg p, t\}, \{\neg s\}, \{s, \neg t\}\big\}.$$



Note however that the representation of a resolution proof as a tree may be costly in case a clause is repeated several times in the proof, which may result in a repetition of parts of the tree.

Theorem 11.3.2. (Soundness of propositional resolution) $S \vdash_{\mathcal{R}} \square \implies S$ *unsatisfiable.*

Proof. Obviously, if V is an assignment satisfying C_1 and C_2, then also $V \models C$ for any resolvent C of C_1 and C_2. Hence if S is satisfiable then $S \nvdash_{\mathcal{R}} \square$. □

Resolution is not complete in the sense that for every set of clauses S and every clause C, whenever $S \models C$ then $S \vdash_{\mathcal{R}} C$. For example, $\models p \vee \neg p$ but $\nvdash_{\mathcal{R}} \{p, \neg p\}$. However, resolution is complete in the sense that any inconsistent S is refutable. This fact is called *refutation completeness*.

Theorem 11.3.3. (Refutation completeness of propositional resolution) *S unsatisfiable $\Longrightarrow S \vdash_{\mathcal{R}} \square$.*

Proof. Cf. [27]. □

The first superpolynomial lower bounds for resolution were proven by Tseitin [39], and the first exponential ones by Haken [18]. The proof of the latter result uses the following representation of $\neg\mathrm{PHP}_n^m$ as a CNF-formula: For every $i \leqslant m$ and $j \leqslant n$ there is a variable p_{ij} with the intended meaning "pigeon i is in hole j". For every i we have a clause $p_{i1} \vee p_{i2} \vee \ldots \vee p_{in}$ expressing that i is in some hole, and for every k and $i \neq j$ we have a clause $\bar{p}_{ik} \vee \bar{p}_{jk}$ expressing that no hole k gets two pigeons i and j. Taken together these clauses express the formula $\neg\mathrm{PHP}_n^m$, which is of course false if $m > n$.

Theorem 11.3.4. (Haken [18]) *There are unsatisfiable formulas that only have exponential size resolution refutations. More specific, for every $n \geqslant 2$, any resolution refutation of $\neg\mathrm{PHP}_{n-1}^n$ has size at least $2^{\frac{n}{20}}$.*

Proof. We follow the presentation in [2]. We view a resolution refutation as an analysis showing that there is no assignment from the set of all possible assignments that satisfies all the given clauses. The analysis remains valid if we restrict attention to a *subset* of all possible assignments. We will restrict attention to the following set of assignments: An assignment for $\neg\mathrm{PHP}_{n-1}^n$ is *i-critical* if it assigns $n-1$ pigeons in an injective way to $n-1$ holes and leaves pigeon i unassigned.

Restricting attention to the above set of critical assignments has the advantage that we can make all clauses in the refutation *monotone*, that is, containing only positive literals. (Strictly speaking, changing the clauses makes the refutation no longer a resolution proof, but it is equivalent to it as far as the assignments under consideration are concerned.) We make the proof monotone by replacing in every clause every negative literal $\bar{p}_{ij}$ by

$$\bigvee_{k \neq i} p_{kj}.$$

Note that for critical assignments these are equivalent, since such an assignment assigns a pigeon to every hole.

Call a clause in the proof *large* if it has at least $\frac{n^2}{10}$ variables, and let L be the number of large clauses. All large clauses together have at least $\frac{n^2}{10}L$ occurrences of variables, and there are n^2 variables p_{ij}, so there must be a

variable p_{ij} that occurs at least $\frac{1}{10}L$ times in the L large clauses. Setting $p_{ij} = 1$ and $p_{ij'} = 0$ for every $j' \neq j$ and $p_{i'j} = 0$ for every $i' \neq i$ leaves at most $\frac{9}{10}L$ large clauses. Since this removes one pigeon and one hole, we now have a monotone refutation of $\neg\mathrm{PHP}^{n-1}_{n-2}$.

We repeat this t times, where t is so large that $(\frac{9}{10})^t L < 1$, i.e. $(\frac{10}{9})^t > L$, so we let $t = \log_{\frac{10}{9}} L$ (rounded up, plus 1 if necessary). This removes all the large clauses, and leaves us with a monotone refutation of $\neg\mathrm{PHP}^{n-t}_{n-t-1}$.

Now if $L < 2^{\frac{n}{20}}$ (which would be the case if the original refutation had size less than $2^{\frac{n}{20}}$) then $t = \log_{\frac{10}{9}} L < \frac{n}{20}\log_{\frac{10}{9}} 2$, from which it follows that

$$\frac{2}{9}(n-t)^2 > \frac{1}{10}n^2.$$

So we have a monotone refutation of $\neg\mathrm{PHP}^{n-t}_{n-t-1}$ with no clauses with $\frac{2}{9}(n-t)^2$ variables, contradicting Lemma 11.3.5. □

Lemma 11.3.5. *Every monotone refutation of $\neg\mathrm{PHP}^{n}_{n-1}$ contains a clause with at least $\frac{2}{9}n^2$ variables.*

Proof. For every clause C define

$$\mathrm{witness}(C) = \big\{i : \text{there is an } i\text{-critical assignment falsifying } C\big\},$$

and define the complexity $\mathrm{comp}(C)$ to be $|\mathrm{witness}(C)|$. If C is a resolvent of C' and C'' then $\mathrm{comp}(C) \leqslant \mathrm{comp}(C') + \mathrm{comp}(C'')$ because any assignment falsifying C must falsify at least one of C' and C''. Note that if $\mathrm{comp}(C) \geqslant \frac{2}{3}n$ then $\mathrm{comp}(C') + \mathrm{comp}(C'') \geqslant \frac{2}{3}n$, hence $\mathrm{comp}(C') \geqslant \frac{1}{3}n$ or $\mathrm{comp}(C'') \geqslant \frac{1}{3}n$. So if C is the *first* clause in the proof with $\mathrm{comp}(C) \geqslant \frac{1}{3}n$ (which exists since $\mathrm{comp}(\Box) = n$) then

$$\frac{1}{3}n \leqslant \mathrm{comp}(C) \leqslant \frac{2}{3}n.$$

We show that C has at least $\frac{2}{9}n^2$ variables, namely, that if $\mathrm{comp}(C) = x$ then it contains at least $x(n-x)$ variables. This suffices since $x(n-x) \geqslant \frac{2}{9}n^2$ for $\frac{1}{3}n \leqslant x \leqslant \frac{2}{3}n$.

Suppose $i \in \mathrm{witness}(C)$ and σ is an i-critical assignment falsifying C. For each $j \notin \mathrm{witness}(C)$, consider the assignment σ' obtained by replacing j by i: if σ maps j to hole k, σ' maps i to hole k and leaves j unassigned. Then σ' is j-critical, and since $j \notin \mathrm{witness}(C)$ it satisfies C. Since σ' satisfies C and σ does not, this can only be because σ' makes p_{ik} true, and in particular C must contain p_{ik}. Repeating the argument for every $j \notin \mathrm{witness}(C)$, and noting that for every j the assignment σ maps j to a different hole k, we see that C contains $(n-x)$ distinct variables p_{ik}. Since there are x different i in $\mathrm{witness}(C)$, this gives $x(n-x)$ different variables p_{ik} in total. □

11.4 Exercises

Exercise 11.4.1. Show that if we drop the requirement in Definition 11.1.1 that F is polynomial time computable then there is a proof system in which every valid formula has a short proof.

Exercise 11.4.2. Show that in Definition 11.1.1 we can require without loss of generality that F is surjective.

Exercise 11.4.3. Formulate the pigeonhole principle for m pigeons and n holes as a formula PHP_n^m of propositional logic. Introduce variables $x_{i,j}$ with the meaning "pigeon i is in hole j".

Exercise 11.4.4. Let PHP_n^m the formula from Exercise 11.4.3. Prove PHP_2^3 using resolution, i.e. write $\neg\mathrm{PHP}_2^3$ in clausal form, and refute it.

Further reading

By now there is a large number of textbooks available for the subject, and several texts and notes are electronically available. The following is a very incomplete list:

- M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.
- J. L. Balcázar, J. Díaz, and J. Gabarró, *Structural Complexity*, two volumes, Springer, 1988 (Vol. I) and 1990 (Vol. II).
- C. H. Papadimitriou, *Computational Complexity*, Addison Wesley, 1994.
- M. Sipser, *Introduction to the Theory of Computation*, Course Technology, 2006.
- S. Arora and B. Barak, *Computational Complexity: A Modern Approach*, Cambridge University Press, 2009.
- L. Trevisan, *Lecture Notes on Computational Complexity*, available at web pages of the author.
- H. S. Wilf, *Algorithms and complexity*, available at web pages of the author.

Bibliography

[1] M. Agrawal, N. Kayal, and N. Saxena, PRIMES *is in* P, Annals of Mathematics 160 (2004) 781–793.

[2] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*, Cambridge University Press, 2009.

[3] T. Baker, J. Gill, and R. M. Solovay, *Relativizations of the* P = NP *question*, SIAM Journal on Computing 4 (1975) 431–442.

[4] J. L. Balcázar, J. Díaz, and J. Gabarró, *Structural Complexity*, two volumes, Springer, 1988 (Vol. I) and 1990 (Vol. II).

[5] R. Beigel, N. Reingold and D. Spielman, PP *is closed under intersection*, Journal of Computer and System Sciences 50(2) (1995) 191–202.

[6] M. Blum, *A machine-independent theory of the complexity of recursive functions*, Journal of the ACM 14 (1967) 322–336.

[7] M. Blum, *How to prove a theorem so no one else can claim it.*, Proc. International Congress of Mathematicians (1987) 1444–1451.

[8] J. Buhler and S. Wagon, *Basic algorithms in number theory*, in: Algorithmic number theory, MSRI Publications 44 (2008) 25–68.

[9] S. R. Buss, *Polynomial size proofs of the propositional pigeonhole principle*, Journal of Symboloc Logic 52 (1987) 916–927.

[10] S. R. Buss, *Towards* NP-P *via proof complexity and search*, Annals of Pure and Applied Logic 163 (7) (2012) 906–917.

[11] S. Cook, *The complexity of theorem proving procedures*, Proc. 3rd ACM Symposium on the Theory of Computing (1971) 151–158.

[12] S. A. Cook, *A hierarchy for nondeterministic time complexity*, Proc. 4th ACM Symposium on the Theory of Computing (1972) 187–192.

[13] S. A. Cook and R. A. Reckhow, *On the lengths of proofs in the propositional calculus*, Proc. 6th ACM Symposium on the Theory of Computing, (1974) 135–148.

[14] R. Downey and M. R. Fellows, *Parameterized complexity*, Springer, 1999.

[15] J. Gill, *Computational complexity of probabilistic Turing machines*, SIAM Journal on Computing 6 (1977) 675–695.

[16] K. Gödel, *Über die Länge von Beweisen*, Ergebnisse eines mathematischen Kolloquiums 7 (1936) 23–24.

[17] O. Goldreich, S. Micali, and A. Wigderson, *Proofs that yield nothing but their validity, and a methodology of cryptographic protocol design*, Proc. 27th IEEE Symposium on the Foundations of Computer Science (1986) 174–187.

[18] A. Haken, *The intractability of resolution*, Theoretical Computer Science 39 (1985) 297–308.

[19] J. E. Hopcroft and J. D. Ullman, *Introduction to automata theory, languages, and computation*, Addison-Wesley, 1979.

[20] P. Hrubeš, *A lower bound for intuitionistic logic*, Annals of Pure and Applied Logic 146 (2007) 72–90.

[21] N. Immerman, *Nondeterministic space is closed under complementation*, SIAM Journal on Computing 17 (1988) 935–938.

[22] A. R. Meyer and L. J. Stockmeyer, *The equivalence problem for regular expressions with squaring requires exponential space*, Proc. 13th IEEE Symposium on Switching and Automata Theory (1972) 125-129.

[23] R. M. Karp, *Reducibility among combinatorial problems*, In: R. E. Miller and J. W. Thatcher (eds), Complexity of Computer Computations, 1972, 85–103.

[24] R. Karp and R. Lipton, *Turing machines that take advice*, L'ensignement mathématique 28 (1982) 191–210.

[25] R. Ladner, *On the structure of polynomial time reducibility*, Journal of the ACM 22 (1975) 155–171.

[26] R. Ladner, N. Lynch, and A. Selman, *Comparison of polynomial-time reducibilities*, Proc. 6th ACM Symposium on the Theory of Computing (1974) 110–121.

[27] A. Nerode and R. A. Shore, *Logic for applications*, 2nd edition, Springer 1997.

[28] P. G. Odifreddi, *Classical recursion theory*, Vol. 2, Studies in logic and the foundations of mathematics Vol. 143, North-Holland, 1999.

[29] C. H. Papadimitriou, *Computational complexity*, Addison Wesley, 1994.

[30] J. Radhakrishnan and M. Sudan, *On Dinur's proof of the* PCP *theorem*, Bulletin of the AMS 44 (2007) 19–61.

[31] S. Rudich and A. Wigderson (eds.), *Computational complexity theory*, American Mathematical Society, 2004.

[32] M. Sipser, *Introduction to the theory of computation*, 2nd edition, Brooks Cole, 2005.

[33] U. Schöning, *Graph Isomorphism is in the low hierarchy*, Journal of Computer and System Sciences 37 (1988) 312–323.

[34] R. Solovay and V. Strassen, *A fast Monte-Carlo test for primality*, SIAM Journal on Computing 6(1) (1977) 84–85.

[35] L. Sterne, *The Life and Opinions of Tristram Shandy, Gentleman*, 1759. Everyman's Library, 1991.

[36] R. Szelepcsényi, *The method of forcing for nondeterministic automata*, Bulletin of the EATCS 33 (1987) 96–100.

[37] S. A. Terwijn, *Syllabus computability theory*, Technical University of Vienna, 2004. Available at web pages author.

[38] L. Trevisan, *Lecture Notes on Computational Complexity*, available at web pages of the author.

[39] G. S. Tseitin, *On the complexity of derivation in propositional calculus*, In: A. O. Slisenko (ed.), *Studies in constructive mathematics and mathematical logic* (1970) 115–125.

[40] L. Wittgenstein, *Tractatus Logico-Philosophicus*, 1922. Routledge 1981.

Index