Computational Complexity Theory

Moritz Müller

February 7, 2024

Contents

Chapter 1

Time

1.1 Computation

1.1.1 Some problems of Hilbert

On the 8th of August in 1900, at the International Congress of Mathematicians in Paris, David Hilbert challenged the mathematical community with 23 open problems. These problems had great impact on the development of mathematics in the 20th century. Hilbert's 10th problem asks whether there exists an algorithm solving the problem

Diophant	
Input:	a diophantine equation.
Problem:	does the equation have an integer solution?

Recall that a diophantine equation is of the form $p(x_1, x_2, \ldots) = 0$ where $p \in \mathbb{Z}[x_1, x_2, \ldots]$ is a multivariate polynomial with integer coefficients.

Recall that a first-order sentence is *valid* if it is true in all models interpreting its language. In 1928 Hilbert asked for an algorithm solving

Entscheidung	
Input:	a first-order sentence φ.
Problem:	is φ valid?

At the time these questions have been informal. To understand the questions formally one has to define what "problems" are, what "algorithms" are, and what it means for an algorithm to "decide" some problem. This is what we do in this chapter, and in particular define *decidable* and *computably enumerable* problems. The answers to the above-mentioned problems read as follows.

Theorem 1.1.1 (Gödel 1928) Entscheidung *is computably enumerable.*

This follows from Gödel's completeness theorem. However, the notion of algorithm has been formalized only later by Church and Turing. This allowed them to prove

Theorem 1.1.2 (Church, Turing 1936) ENTSCHEIDUNG *is not decidable.*

This contrasts with the following result of Trachtenbrot. Being *valid in the finite* means to be true in all models with a finite universe.

Theorem 1.1.3 (Trachtenbrot 1953) *The following is not computably enumerable.*

ENTSCHEIDUNG(fin)	
Input:	a first-order sentence φ.
Problem:	is φ valid in the finite?

Building on earlier work of J. Robinson, Davis and Putnam, Hilbert's 10th problem has finally been solved by Matiyasevich:

Theorem 1.1.4 (Matiyasevich 1970) DIOPHANT *is not decidable.*

1.1.2 What is a problem?

Definition 1.1.5 A *(decision) problem* is a subset Q of $\{0,1\}^*$.

Here, $\{0,1\}^* = \bigcup_{n\in\mathbb{N}}\{0,1\}^n$ is the set of binary strings. We write a binary string $x \in \{0,1\}^n$ as $x_1\cdots x_n$ and say it has *length* $|x| = n$. Note there is a unique string λ of length 0. We write $\{0,1\}^{\leqslant n} := \bigcup_{i\leqslant n}\{0,1\}^i$ for the strings of length at most n. We also write

$$[n] := \{1,\ldots,n\}$$

for $n \in \mathbb{N}$ and understand $[0] = \varnothing$.

Example: encoding graphs One may object against this definition that many (intuitive) problems are not about finite strings, e.g.

ACYCLIC	
Input:	a (finite) directed graph G.
Problem:	is G acyclic?

A *directed graph* is a pair $G = (V,E)$ with a non-empty set V of *vertices* and an irreflexive set $E \subseteq V \times V$ of *edges*. A *path (in G)* is a nonempty finite sequence of pairwise distinct vertices $v_0\cdots v_\ell$ such that $(v_i, v_{i+1}) \in E$ for all $i < \ell$; it is a path *from* v_0 and *to* $v_{\ell-1}$; its *length* is ℓ, the number of edges. It is a *cycle (in G)* if $v_0 = v_\ell$ and $\ell > 0$. A directed graph is *acyclic* if there are no cycles in G.

The objection is usually rebutted by saying that the definition captures such problems up to some encoding. For example, say the directed graph $G = (V,E)$ has vertices $V = [n]$ for some $n \in \mathbb{N}$, and consider its *adjacency matrix* $(a_{ij})_{i,j\in[n]}$ given by

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{else} \end{cases}.$$

This matrix can be written as a binary string $\ulcorner G \urcorner = x_1x_2\cdots x_{n^2}$ where $x_{(i-1)n+j} = a_{ij}$. We identify ACYCLIC with $\{\ulcorner G \urcorner \mid G \text{ is an acyclic directed graph}\} \subseteq \{0,1\}^*$.

Example: encoding numbers and pairs Consider the *Independent Set* problem

IS	
Input:	a graph G and a natural $k \in \mathbb{N}$.
Problem:	does G contain an independent set of cardinality k?

A *graph* is a directed graph $G = (V, E)$ with symmetric E. That $X \subseteq V$ is *independent* means $E \cap X^2 = \emptyset$. The natural number k is encoded by its *binary representation*, that is, the binary string $bin(k) = x_1 \cdots x_{\lceil \log(k+1) \rceil}$ such that

$$k = \sum_{i=1}^{\lceil \log(k+1) \rceil} x_i \cdot 2^{\lceil \log(k+1) \rceil - i}.$$

Then IS can be viewed as the following problem in the sense of Definition 1.1.5:

$$\{ \langle \ulcorner G \urcorner, bin(k) \rangle \mid G \text{ is a graph containing an independent set of cardinality } k \},$$

where $\langle \cdot, \cdot \rangle : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ is a suitable pairing function, e.g.

$$\langle x_1 \cdots x_n, y_1 \cdots y_m \rangle := x_1 x_1 \cdots x_n x_n 01 y_1 y_1 \cdots y_m y_m.$$

This defines an injection and it is easy to 'read off' x and y from $\langle x, y \rangle$. Longer tuples $(x^1, \ldots, x^k) \in (\{0,1\}^*)^k$ for $k > 2$ are similarly coded by the string

$$\langle x^1, \ldots, x^k \rangle := x^1_1 x^1_1 \cdots x^1_{n_1} x^1_{n_1} 01 x^2_1 x^2_1 \cdots x^2_{n_2} x^2_{n_2} 01 \cdots 01 x^k_1 x^k_1 \cdots x^k_{n_k} x^k_{n_k}$$

where $n_i := |x_i|$ and $x^i = x^i_1 \cdots x^i_{n_i}$ for $i \in [k]$.

In general, we are not interested in the details of the encoding.

Example: bit-graphs Another and better objection against our definition of "problem" is that it formalizes only decision problems, namley yes/no-questions, whereas many natural problems ask, given $x \in \{0,1\}^*$, to compute $f(x)$, where $f : \{0,1\}^* \to \{0,1\}^*$ is some function of interest. For example, one might be interested not only in deciding whether a given graph has or not an independent set of a given cardinality, but one might want to compute such an independent set in case there exists one. This is a valid objection and we are going to consider such 'search' problems. But most phenomena we are interested in are already observable when restricting attention to decision problems. For example, computing f 'efficiently' is roughly the same as deciding "efficiently" the *bit-graph of f*:

BitGraph(f)	
Input:	$x \in \{0,1\}^*$, a natural $i \in \mathbb{N}$ and a bit $b \in \{0,1\}$.
Problem:	does the ith bit of $f(x)$ exist and equal b?

We shall see that an algorithm 'efficiently' solving IS can be used to also 'efficiently' solve the associated search problem.

1.1.3 What is an algorithm?

Let's start with an intuitive discussion: what are you doing when you are performing a computation? You have a scratch pad on which finitely many out of finitely many possible symbols are written. You read some symbol, change some symbol or add some symbol one at a time depending on what you are thinking in the moment. For thinking you have finitely many (relevant) states of consciousness. But, in fact, not much thinking is involved in doing a computation: you are manipulating the symbols according to some fixed "calculation rules" that are applicable in a purely syntactical manner, i.e. their "meaning" or what is irrelevant. By this is meant that your current state of consciousness (e.g. remembering a good looking calculation rule) and the current symbol read (or a blank place on your paper) determines how to change the symbol read, the next state of consciousness and the place where to read the next symbol.

It is this intuitive description that Alan Turing formalized in 1936 by the concept of a Turing machine. It seems unproblematic to say that everything computable in this formal sense is also intuitively computable. The converse is generally accepted, mainly on the grounds that nobody ever could come up with an (intuitive) counterexample. Another reason is that over time many different formalizations have been given and they all turned out to be equivalent. As an example, even a few months before Turing, Alonzo Church gave a formalization based on the so-called λ-calculus.

> *[Turing] has for the first time succeeded in giving an absolute definition of an interesting epistemological notion, i.e., one not depending on the formalism chosen.* Kurt Gödel, 1946

The *Church-Turing Thesis* claims that the intuitive and the formal concept coincide. Note this is a philosophical claim and cannot be subject to mathematical proof or refutation.

> *All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically.* Alan Turing, 1936

Definition 1.1.6 Let $k > 0$. A *(k-tape) Turing machine* is a pair $\mathbb{A} = (S, \delta)$ where S is a finite set of *states* containing an *initial state* $s_{\text{start}} \in S$ and a *halting state* $s_{\text{halt}} \in S$, and

$$\delta : S \times \{\triangleright, \square, 0, 1\}^k \to S \times \{\triangleright, \square, 0, 1\}^k \times \{1, 0, -1\}^k$$

is the *transition function* satisfying the following: if $\delta(s, a) = (s', b, m)$ where $a = a_1 \cdots a_k$ and $b = b_1 \cdots b_k$ are in $\{\triangleright, \square, 0, 1\}^k$ and $m = m_1 \cdots m_k$ in $\{-1, 0, 1\}^k$, then for all $i \in [k]$

(a) $a_i = \triangleright$ if and only if $b_i = \triangleright$,

(b) if $a_i = \triangleright$, then $m_i \neq -1$,

(c) if $s = s_{\text{halt}}$, then $s = s', a = b$ and $m_i = 0$.

A 1-tape Turing machine is called *single-tape*.

We informally describe for how a single-tape Turing machine computes. The tape is what we called a scratch pad above, and is an infinite array of *cells* each containing a symbol 0 or 1 or being blank, i.e. containing $\square$. The machine has a *head* moving on the cells, at each time *scanning* exactly one cell. At the start the machine is in its initial state, the head scans cell number 0 and the input $x = x_1 \cdots x_n \in \{0,1\}^n$ is written on the tape, namely, cell 1 contains x_1, cell 2 contains x_2 and so on. Cell 0 contains a special symbol $\triangleright$ that marks the end of the tape. It is never changed nor written in some other cell (condition (a)) and if some head scans $\triangleright$ it cannot move left (condition (b)) and fall of the tape. All other cells are blank. Assume the machine currently scans a cell containing a and is in state s. Then $\delta(s,a) = (s',b,m)$ means that it changes a to b, changes to state s' and moves the head on the input tape one cell to the right or one cell to the left or stays depending on whether m is 1, -1 or 0 respectively. If s_{halt} is reached, the computation stops in the sense that the current configuration is repeated forever (condition (c)).

Definition 1.1.7 Let $k,n \in \mathbb{N}, k > 0$, and $\mathbb{A} = (S,\delta)$ be a k-tape Turing machine and $x = x_1 \cdots x_n \in \{0,1\}^n$. A *configuration of* $\mathbb{A}$ is a tuple (s,j,c) where $s \in S, j \in \mathbb{N}^k$ and c is a k-tuple of functions from $\mathbb{N}$ into $\{0,1,\square,\triangleright\}$; we write $j = j_1 \cdots j_k$ and $c = c_1 \cdots c_k$. A configuration (s,j,c) is *halting* if $s = s_{\text{halt}}$.

Writing 0^k for the k-tuple $0 \cdots 0$, the *start configuration of* $\mathbb{A}$ *on* x is $(s_{\text{start}}, 0^k, c)$ where $c = c_1 \cdots c_k$ is defined as follows. For all $i \in [k], j \in \mathbb{N}$

$$c_i(j) = \begin{cases} \triangleright & \text{if } j = 0 \\ \square & \text{if } j > 0, i > 1 \text{ or } j > n, i = 1 \\ x_j & \text{if } i = 1, j \in [n] \end{cases}.$$

That is, written as sequences, in the start configuration c_i for $i > 1$ reads $\triangleright \; \square \; \square \cdots$ and c_1 reads $\triangleright \; x_1 \; x_2 \cdots x_n \; \square \; \square \cdots$. The *successor configuration of* (s,j,c) is the configuration (s',j',c') such that there exists $m = m_1 \cdots m_k \in \{-1,0,1\}^k$ such that for all $i \in [k]$:

(a) $\delta(s, c_1(j_1) \cdots c_k(j_k)) = (s', c'_1(j_1) \cdots c'_k(j_k), m)$,

(b) $j'_i = j_i + m_i$,

(c) $c_i(\ell) = c'_i(\ell)$ for all $\ell \neq j_i$.

A *run of* $\mathbb{A}$ or a *computation of* $\mathbb{A}$ is a (finite or infinite) sequence of configurations, each (besides the first) being a successor of the previous one. If the first configuration is the start configuration of $\mathbb{A}$ on x, the the run is *on* x. If it is finite and ends in a halting configuration, then it is *complete*.

Occasionally we consider Turing machines with *input tape*. On this tape the input is written in the start configuration and this content is never changed. Moreover, the machine is not allowed to move the input head far into the infinite array of blank cells after the cell containing the last input bit. On an *output tape* the machine only writes moving the head stepwise from left to right.

Definition 1.1.8 For $k > 1$, a k-tape Turing machine *with (read-only) input tape* is one that in addition to (a)–(c) of Definition 1.1.6 also satisfies

(d) $a_1 = b_1$;

(e) if $a_1 = \square$, then $m_1 \neq 1$.

For $k > 1$, a k-tape Turing machine *with (write-only) output tape* is one that in addition to (a)–(c) of Definition 1.1.6 also satisfies $m_k \neq -1$.

Computation tables Computation tables serve well for visualizing computations. As an example let's consider a 2-tape Turing machine $\mathbb{A} = (S, \delta)$ that reverses its input string: its states are $S = \{s_{\text{start}}, s_{\text{halt}}, s_r, s_\ell\}$ and its transition function δ satisfies

$$\begin{aligned}
&\delta(s_{\text{start}}, \triangleright\triangleright) = (s_r, \triangleright\triangleright, 10),\\
&\delta(s_r, b\triangleright) = (s_r, b\triangleright, 10) \text{ for } b \in \{0,1\},\\
&\delta(s_r, \square\triangleright) = (s_\ell, \square\triangleright, -11),\\
&\delta(s_\ell, b\square) = (s_\ell, bb, -11) \text{ for } b \in \{0,1\},\\
&\delta(s_\ell, \triangleright\square) = (s_{\text{halt}}, \triangleright\square, 00).
\end{aligned}$$

We are not interested where the remaining triples are mapped to, but we can explain this in a way making $\mathbb{A}$ a 2-tape machine with input tape and with output tape.

The following table pictures the computation of $\mathbb{A}$ on input 10. The ith row of the table shows the ith configuration in the sense that it lists the symbols from the input tape up to the first blank followed by the contents of the worktape; the head positions and the machines state are also indicated:

$$\begin{array}{cccc|ccccc}
(\triangleright, s_{\text{start}}) & 1 & 0 & \square & (\triangleright, s_{\text{start}}) & \square & \square & \square & \square\\
\triangleright & (1, s_r) & 0 & \square & (\triangleright, s_r) & \square & \square & \square & \square\\
\triangleright & 1 & (0, s_r) & \square & (\triangleright, s_r) & \square & \square & \square & \square\\
\triangleright & 1 & 0 & (\square, s_r) & (\triangleright, s_r) & \square & \square & \square & \square\\
\triangleright & 1 & (0, s_\ell) & \square & \triangleright & (\square, s_\ell) & \square & \square & \square\\
\triangleright & (1, s_\ell) & 0 & \square & \triangleright & 0 & (\square, s_\ell) & \square & \square\\
(\triangleright, s_\ell) & 1 & 0 & \square & \triangleright & 0 & 1 & (\square, s_\ell) & \square\\
(\triangleright, s_{\text{halt}}) & 1 & 0 & \square & \triangleright & 0 & 1 & (\square, s_{\text{halt}}) & \square
\end{array}$$

Here is the definition for the case of a single-tape machine:

Definition 1.1.9 Let $\mathbb{A} = (S, \delta)$ be a single-tape Turing machine, $x \in \{0,1\}^*$ and $t \geqslant 1$. The *computation table of* $\mathbb{A}$ *on* x *up to step* t is the following matrix $(T_{ij})_{ij}$ over the alphabet $\{0, 1, \square, \triangleright\} \cup (\{0, 1, \square, \triangleright\} \times S)$ with $i \in [t]$ and $j \leqslant \tilde{t} := \max\{|x|, t\} + 1$. For $i \in [t]$ let (s_i, j_i, c_i) be the ith configuration in the run of $\mathbb{A}$ on x. The ith row $T_{i0} \cdots T_{i\tilde{t}}$ equals $c_i(0)c_i(1)\cdots c_i(\tilde{t})$ except that the $(j_i + 1)$th symbol σ is replaced by (σ, s_i) respectively.

Note that the last column of a computation table contains only blanks $\square$.

1.1.4 What does it mean to decide a problem?

Definition 1.1.10 Let $k \geqslant 1$ and $\mathbb{A} = (S, \delta)$ be a k-tape Turing machine and $x \in \{0,1\}^*$. Assume there exists a complete run of $\mathbb{A}$ on x, say ending in a halting configuration with cell contents $c = c_1 \cdots c_k$; the *output* of the run is the binary string $c_k(1) \cdots c_k(j)$ where $j+1$ is the minimal cell number such that $c_k(j+1) = \square$; this is the empty string λ if $j = 0$. The run is *accepting* if $c_k(1) = 1$ and *rejecting* if $c_k(1) = 0$.

The machine $\mathbb{A}$ computes the partial function that maps x to the output of a complete run of $\mathbb{A}$ on x and is undefined if no complete run of $\mathbb{A}$ on x exists. The machine $\mathbb{A}$ is said to *accept* x or to *reject* x if there is an accepting or rejecting (complete) run of $\mathbb{A}$ on x respectively. It is said to *decide* a problem $Q \subseteq \{0,1\}^*$ if it accepts every $x \in Q$ and rejects every $x \notin Q$. Finally, it is said to *accept* the problem

$$L(\mathbb{A}) := \{x \mid \mathbb{A} \text{ accepts } x\}.$$

A problem is called *decidable* if there is a Turing machine deciding it. A partial function is *computable* if there is a Turing machine computing it. A problem is *computably enumerable* if there is a Turing machine accepting it.

In a certain sense, the number of tapes does not matter:

Lemma 1.1.11 *For every $k > 1$ and every k-tape Turing machine $\mathbb{A}$, there exists $c \in \mathbb{N}$ and a single-tape Turing machine $\mathbb{B}$ such that for all $x \in \{0,1\}^*, t \in \mathbb{N}$, if there is a halting (accepting, rejecting) computation of $\mathbb{A}$ on x of length t, then there is a halting (accepting, rejecting) computation of $\mathbb{B}$ on x of length $\leqslant c \cdot (t^2 + |x|) + c$.*

Proof: We first define a single-tape Turing machine $\mathbb{A}'$ that instead with bits $\{0,1\}$ works with a larger finite alphabet Σ. The definition machines is straightforward.

For concreteness assume $k = 4$. The alphabet Σ contains $\{0, 1, \hat{0}, \hat{1}\}^4$. The computation of $\mathbb{A}'$ on x produces for every configuration C in the run of $\mathbb{A}$ on x a configuration C' that codes C as follows. A cell in C' has letter $\hat{0}1\hat{0}1$ if C has 0 in this cell on the first tape, 1 on the second, 0 on the third, and 1 on the fourth; moreover, in C the first and third head currently scan the cell.

One step of $\mathbb{A}$ is simulated by scanning the tape from left to right (until the first blank) and back, collecting (and storing by moving to an appropriate state) the information which symbols the 4 heads are reading, say 0110. It then moves forth and back the tape and changes the first components of each symbol according to the action of the first head of $\mathbb{A}$. E.g. if writes 1 and moves right, the letter $\hat{0}1\hat{0}1$ above is changed to $01\hat{0}1$ and the next letter, say $100\hat{0}$ is changed to $\hat{1}00\hat{0}$. $\mathbb{A}'$ proceeds like this for all 4 heads in turn.

This way the ith step of $\mathbb{A}$ is simulated by $\leqslant 10 \cdot k \cdot \max\{i, |x|\} + 10$ many steps of $\mathbb{A}'$. In the beginning, $\mathbb{A}'$ replaces its input $x = (x_1, \ldots, x_n)$ by $(\hat{\triangleright}\hat{\triangleright}\hat{\triangleright}\hat{\triangleright}, x_1x_1x_1x_1, \ldots x_nx_nx_nx_n)$. This takes $\leqslant 10 \cdot n + 10$ steps. In total, $\mathbb{A}'$ takes $\leqslant d \cdot t^2 + d$ steps for a suitable *constant* $d \in \mathbb{N}$, i.e., independent of the input x.

We next change the alphabet Σ back to $\{0,1\}$. The machine $\mathbb{B}$ proceeds as $\mathbb{A}'$ but instead of a symbol in $\Sigma \cup \{\square\}$ stores a binary code of the symbol of length $s := \lceil \log(|\Sigma|+1) \rceil$.

E.g., when $\mathbb{A}'$ writes one symbol and moves one cell left, $\mathbb{A}$ writes the corresponding s bits and then moves $2s$ cells left. Note that one step is simulated by constantly many steps.

Since $\mathbb{A}'$ finishes in cell 0, $\mathbb{B}$ can finally change, in constantly many steps, the first cell to 1 or 0 according to whether $\mathbb{A}$s run simulated by $\mathbb{A}'$ is accepting or rejecting. □

Exercise 1.1.12 Define a *bidirectional Turing machine* to be one whose tapes are infinite also to the left, i.e. numbered by integers $\mathbb{Z}$ instead naturals $\mathbb{N}$. Simulate one such tape by two usual tapes: when the bidirectional machines want to moves the head to cell -1, the usual machine moves the head on the second tape to 1.

Define Turing machines with 3-dimensional tapes and simulate them by usual Turing machines. Proceed with Turing machines that operate with more than one head per worktape. Proceed with some other fancy variant.

Proposition 1.1.13 *A problem Q is decidable if and only if both Q and $\{0,1\}^* \smallsetminus Q$ are computably enumerable.*

Proof: Obviously, a machine $\mathbb{A}$ deciding Q also accepts Q, and a machine deciding $\{0,1\}^* \smallsetminus Q$ is obtained from $\mathbb{A}$ by running $\mathbb{A}$ and interchanging acceptance and rejection. Conversely, assume $Q = L(\mathbb{A}_1)$ for some k_1-tape machine $\mathbb{A}_1$, and $\{0,1\}^* \smallsetminus Q = L(\mathbb{A}_0)$ for some k_0-tape machine. Define a $k_1 + k_2 + 1$-tape machine $\mathbb{B}$ that behaves as follows: on input x, copy x to tape $k_0 + 1$ and then simultaneously run $\mathbb{A}_0$ on the first k_0 tapes and $\mathbb{A}_1$ on the last k_1 tapes. If $\mathbb{A}_0$ accepts, then accept, i.e., write 1 into cell 1 of tape $k_0 + k_1 + 1$ and halt. Similarly, if $\mathbb{A}_0$ accepts, then reject. □

Exercise 1.1.14 Verify the following claims. A problem Q is decidable if and only if its characteristic function $\chi_Q : \{0,1\}^* \to \{0,1\}$ that maps $x \in \{0,1\}^*$ to

$$\chi_Q(x) := \begin{cases} 1 & \text{if } x \in Q \\ 0 & \text{if } x \notin Q \end{cases}$$

is computable. A nonempty problem is computably enumerable if and only if it is the range of a computable total function.

Exercise 1.1.15 (Halting) Let $\ulcorner \mathbb{A} \urcorner$ be a 'reasonable' encoding of Turing machines by binary strings. Show the *Halting problem* is computably enumerable and undecidable:

HALTING	
Input:	a single-tape Turing machine $\mathbb{A}$.
Problem:	Does $\mathbb{A}$ halt on input λ?

Roughly, the following shows that besides halting on the empty input, also every other non-trivial property is undecidable.

Exercise 1.1.16 (Rice's theorem) Let $\mathcal{C}$ be a nonempty set of computably enumerable problems and assume $\varnothing \notin \mathcal{C}$. Show the following is undecidable.

RICE($\mathcal{C}$)	
Input:	a single-tape Turing machine $\mathbb{A}$.
Problem:	is $L(\mathbb{A}) \in \mathcal{C}$?

1.2 Time bounded computation

1.2.1 Some problems of Gödel and von Neumann

We consider the problem to compute the product of two given natural numbers k and ℓ. The *Naïve Algorithm* starts with 0 and adds repeatedly k and does so for ℓ times. Note that we agreed to consider natural numbers as given in binary representations $bin(k), bin(\ell)$, so $bin(k \cdot \ell)$ has length roughly $|bin(k)| + |bin(\ell)|$. Assuming that one addition can be done in roughly this many steps, the Naïve Algorithm performs roughly $\ell \cdot (|bin(k)| + |bin(\ell)|)$ many steps, which is roughly $2^{|bin(\ell)|} \cdot (|bin(k)| + |bin(\ell)|)$. Algorithms that take $2^{\text{constant}\cdot n}$ steps on inputs of length n are what we are going to call *simply exponential.*

Remember the *School Algorithm* – here is an example multiplying $k = 19$ with $\ell = 7$:

$$\begin{array}{ccccccccc}
1 & 0 & 0 & 1 & 1 & \cdot & 1 & 1 & 1 \\
\hline
 & & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
 & & & 1 & 0 & 0 & 1 & 1 & 0 \\
 & & & & 1 & 0 & 0 & 1 & 1 \\
\hline
 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1
\end{array}$$

The size of this table is roughly $(|bin(k)| + |bin(\ell)|)^2$. As the table is easy to produce, this gives a rough estimate of the number of steps. Algorithms that take n^{constant} steps on inputs of length n are what we are going to call polynomial time.

For large inputs the School Algorithm is much faster and the difference is drastic: to compute the product of two 22 digit numbers on a computer performing a million steps per second takes time more than twice the age of the universe with the Naïve Algorithm, and less than half a second with the School Algorithm. Thus, when badly programmed, even a modern supercomputer is easily outrun by any school boy, already on inputs of moderate size. Whether or not a problem is "efficiently solvable" or "feasible" or "practically solvable" is not so much a question of technological progress, i.e. of hardware, but more of the availability of fast algorithms, i.e., of software. It is thus sensible to formalize the notion of feasibility as a property of problems leaving aside any talk about computing technology. The most important formalization of feasibility, albeit contested by various rivals, is *polynomial time* due to Cobham and Edmonds in 1965.

An immediate philosophical objection against the formalization of the intuitive notion of feasibility by polynomial time is that the latter notion depends on the particular machine model chosen. However, this does not seem to be the case, in fact reasonable models of computation simulate on another with only polynomial overhead.

Another nearlying objection is that running times of n^{100} or $10^{10^{10}} \cdot n$ can hardly be considered feasible. It is a matter of experience that most natural problems in P already have a quadratic or cubic time algorithm with leading constants of tolerable size.

Ahead of their time, both Gödel and von Neumann troubled about complexity theoretic questions before the field was born.

Throughout all modern logic, the only thing that is important is whether a result can be achieved in a finite number of elementary steps or not. The size of the *number of steps which are required, on the other hand, is hardly ever a concern of formal logic. Any finite sequence of correct steps is, as a matter of principle, as good as any other. It is a matter of no consequence whether the number is small or large, or even so large that it couldn't possibly be carried out in a lifetime, or in the presumptive lifetime of the stellar universe as we know it. In dealing with automata, this statement must be significantly modified. In the case of an automaton the thing which matters is not only whether it can reach a certain result in a finite number of steps at all but also how many such steps are needed.* John von Neumann, 1948

A still lasting concern of the time has been the philosophical question as to what extent machines can be intelligent or conscious. In this respect Turing proposed 1950 what became famous as the *Turing Test*. A similar philosophical question is whether mathematicians could be replaced by machines. Church and Turing's Theorem 1.1.2 is generally taken to provide a negative answer, but in the so-called *Lost Letter* of Gödel to von Neumann from March 20, 1956, Gödel reveils that he has not been satisfied by this answer. He considers the following bounded version of ENTSCHEIDUNG

GÖDEL	
Input:	a first-order sentence φ and a natural n.
Problem:	does φ have a proof with at most n symbols

This is trivially decidable and Gödel asked whether it can be decided in $O(n)$ or $O(n^2)$ many steps for every fixed φ. It is known that an affirmative answer would imply $\mathsf{P} = \mathsf{NP}$ (see the next section). Gödel said:

that would have consequences of the greatest importance. Namely, this would clearly mean that the thinking of a mathematician in the case of yes-or-no questions could be completely[1] *replaced by machines, in spite of the unsolvability of the Entscheidungsproblem. [...] Now it seems to me to be quite within the realm of possibility* Kurt Gödel, 1956

1.2.2 Polynomial time

We now define our first complexity classes, the objects of study in complexity theory. These collect problems solvable using only some prescribed amount of a certain resource like time, space, randomness, nondeterminism, advice, oracle questions and what. A complexity class can be thought of as a degree of difficulty of solving a problem. Complexity theory then is the theory of degrees of difficulty.

Definition 1.2.1 Let $t : \mathbb{N} \to \mathbb{N}$. A Turing machine $\mathbb{A}$ is *t-time bounded* if for every $x \in \{0,1\}^*$ there exists a complete run of $\mathbb{A}$ on x that has length at most $t(|x|)$.

[1] Gödel remarks in a footnote "except for the formulation of axioms".

$\mathsf{TIME}(t)$ is the class of problems Q such that there is some $c \in \mathbb{N}$ and some $(c \cdot t + c)$-time bounded $\mathbb{A}$ that decides Q. The classes

$$\begin{array}{lcl} \mathsf{P} & := & \bigcup_{c\in\mathbb{N}} \mathsf{TIME}(n^c) \\ \mathsf{E} & := & \bigcup_{c\in\mathbb{N}} \mathsf{TIME}(2^{c\cdot n}) \\ \mathsf{EXP} & := & \bigcup_{c\in\mathbb{N}} \mathsf{TIME}(2^{n^c}) \end{array}$$

are called *polynomial time*, *simply exponential time* and *exponential time*.

Landau notation The above definition is such that constant factors are discarded as irrelevant. In fact, we are interested in the number of steps an algorithm takes in the sense of how fast it grows (as a function of the input length) asymptotically.

Definition 1.2.2 Let $f, g : \mathbb{N} \to \mathbb{N}$.

- $f \leqslant O(g)$ if there is $c \in \mathbb{N}$ such that $f(n) \leqslant c \cdot g(n) + c$ for all $n \in \mathbb{N}$,
- $f \geqslant \Omega(g)$ if $g \leqslant O(f)$,
- $f = \Theta(g)$ if $f \leqslant O(g)$ and $g \leqslant O(f)$,
- $f \leqslant o(g)$ if for all $c \in \mathbb{N}$ there is $n_c \in \mathbb{N}$ such that for all $n \geqslant n_c : c \cdot f(n) \leqslant g(n)$,
- $f \geqslant \omega(g)$ if $g \leqslant o(f)$.

Exercise 1.2.3 Assume $g(n) > 0$ for all $n \in \mathbb{N}$.

(a) $f \leqslant o(g)$ if and only if $\limsup_n \frac{f(n)}{g(n)} = 0$.

(b) $f \geqslant \omega(g)$ if and only if $\liminf_n \frac{f(n)}{g(n)} = \infty$.

Exercise 1.2.4 Let $p(x)$ be a polynomial with natural coefficients and degree d. Then $p = \Theta(n^d)$ and $p \leqslant o(2^{(\log n)^2})$.

Example 1.2.5 The following problem is in P.

REACHABILITY	
Input:	a directed graph $G = (V, E)$, $v, v' \in V$.
Problem:	is there a path from v to v' in G?

Proof: The following algorithm decides REACHABILITY: it first computes the smallest set that contains v and is closed under E-successors, and then checks whether it contains v'.

```
1. $X \leftarrow \{v\}, Y \leftarrow \emptyset$
2. $X \leftarrow X \cup Y$
3. **for all** $(u, w) \in E$
4.     **if** $u \in X$ **then** $Y \leftarrow Y \cup \{w\}$
5. **if** $Y \nsubseteq X$ **then goto** line 2
6. **if** $v' \in X$ **then** accept
7. reject
```

To estimate the running time, let n denote the input size, i.e. the length of the binary encoding of the input. Let's code $X, Y \subseteq V$ by binary strings of length $|V|$ - the ith bit encodes whether the ith vertex belongs or not to the subset. It is then easy to compute the unions in line 2 and 4 and it is also easy to check the "if"-conditions: e.g., $Y \nsubseteq X$ can be checked in $|V|$ steps with two worktapes, the first containing the string encoding Y and the second the one for X; then the machine moves their heads stepwise from left to right and checks whether at some point the first head reads 1 while the second reads 0.

The **for all** loop in lines 3 and 4 is run for $|E| \leqslant n$ times, so takes time $O(n^2)$. Hence, lines 2 to 5 take time $O(n^2)$. These lines are executed once in the beginnning and then always when the algorithm (in line 5) jumps back to line 2. But this happens at most $|V| - 1 < n$ times because X grows by at least 1 before jumping. In total, the algorithm runs in time $O(n^3)$. □

Exercise 1.2.6 ACYCLIC $\in$ P.

Exercise 1.2.7 This exercise exemplifies so-called *amortized time analysis* and shows that $x \mapsto bin(|x|)$ is computable in linear time. The machine scans its input x from left to right, each step updating a binary counter. Implement this in a way such that the k-th update takes $\leqslant 10 \cdot k$ steps. Then argue the total time is $O(|x|)$.

1.2.3 Time hierarchy

Fix some reasonable encoding of single-tape Turing machines $\mathbb{A}$ by binary strings $\ulcorner \mathbb{A} \urcorner$. For this we assume that $\mathbb{A}$'s set of states is $[s]$ for some $s \in \mathbb{N}$. The encoding should allow you to find $\delta(q, a)$ given (q, a) in time $O(|\ulcorner \mathbb{A} \urcorner|)$. Recall Definition 1.1.9.

Of course, an algorithm is said to run in *quadratic time* if it is $O(n^2)$-time bounded.

Theorem 1.2.8 (Polynomial time universal machine) *There exists a Turing machine that, given the code $\ulcorner \mathbb{A} \urcorner$ of a single-tape Turing machine $\mathbb{A}$, a string $x \in \{0,1\}^*$ and $1^t = 11\cdots1$ for some $t \in \mathbb{N}$, computes in quadratic time the tth row of the computation table of $\mathbb{A}$ on x up to step t.*

Proof: Note that a symbol in the table can be encoded with $O(\log s)$ many bits where s is the number of states of $\mathbb{A}$. A row of the table is encoded by a binary string of length $O(\log s \cdot \max\{t, |x|\})$. Given a row the next row can be computed in linear time. Thus the t-th row can be computed in time $O(|\ulcorner \mathbb{A} \urcorner| \cdot t^2 \cdot |x|)$, □

This theorem is hardly surprising in times where PCs executing whatever software belong to daily life (of the richer part of the world population). But historically, it is hard to underestimate the insight that instead of having different machines for different computational tasks one single machine suffices:

> *It is possible to invent a single machine which can be used to compute any computable sequence.* Alan Turing 1936

Definition 1.2.9 A function $t : \mathbb{N} \to \mathbb{N}$ is *time-constructible* if and only if $t(n) \geqslant n$ for all $n \in \mathbb{N}$ and the function $x \mapsto \mathit{bin}(t(|x|))$ is computable in time $O(t(|x|))$.

Exercise 1.2.10 Show the class of time constructible functions is closed under addition, multiplication and conclude that all polynomials are time-constructible. With f also 2^f is time-constructible. The functions $\sqrt{n}, \log n$ are time-constructible.

Exercise 1.2.11 Let $g : \mathbb{N}^2 \to \mathbb{N}$ be a computable. Show that there exists an increasing time-constructible function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n+1) > g(f(n), n)$ for all $n \in \mathbb{N}$.

Theorem 1.2.12 (Time hierarchy) $\mathsf{TIME}(t^6) \smallsetminus \mathsf{TIME}(t) \neq \varnothing$ *for time-constructible* t.

Proof: Consider the problem

Q_t	
Input:	a single-tape Turing machine $\mathbb{A}$.
Problem:	is it true that $\mathbb{A}$ does not accept $\ulcorner\mathbb{A}\urcorner$ in at most $t(|\ulcorner\mathbb{A}\urcorner|)^3$ many steps?

To show $Q_t \in \mathsf{TIME}(t^6)$, consider the following algorithm:

```
1. $s \leftarrow t(|\ulcorner\mathbb{A}\urcorner|)^3$
2. simulate $\mathbb{A}$ on $\ulcorner\mathbb{A}\urcorner$ for at most $s$ steps
3. if the simulation halts and accepts then reject
4. accept
```

With t also t^3 is time-constructible, so line 1 requires at most $O(t(|\ulcorner\mathbb{A}\urcorner|)^3)$ many steps. Line 2 requires at most $O((|\ulcorner\mathbb{A}\urcorner| + s)^2) \leqslant O(t(|\ulcorner\mathbb{A}\urcorner|)^6)$ many steps (recall $t(n) \geqslant n$) using the universal machine.

We show that $Q_t \notin \mathsf{TIME}(t)$. Assume otherwise. Then by Lemma 1.1.11 there is a single-tape machine $\mathbb{B}$ which (a) decides Q_t and (b) is $(c \cdot t^2 + c)$-time bounded for some $c \in \mathbb{N}$. We can assume (c) $|\ulcorner\mathbb{B}\urcorner| > c$ – otherwise add some dummy states to $\mathbb{B}$.

Assume $\mathbb{B}$ does not accept $\ulcorner\mathbb{B}\urcorner$. Then $\ulcorner\mathbb{B}\urcorner \in Q_t$ by definition of Q_t, so $\mathbb{B}$ does not decide Q_t – contradicting (a). Hence $\mathbb{B}$ accepts $\ulcorner\mathbb{B}\urcorner$, so $\ulcorner\mathbb{B}\urcorner \in Q_t$ by (a). By definition of Q_t we conclude that $\mathbb{B}$ needs strictly more than $t(|\ulcorner\mathbb{B}\urcorner|)^3$ steps on $\ulcorner\mathbb{B}\urcorner$. But

$$c \cdot t^2(|\ulcorner\mathbb{B}\urcorner|) + c < |\ulcorner\mathbb{B}\urcorner| \cdot t^2(|\ulcorner\mathbb{B}\urcorner|) - t^2(|\ulcorner\mathbb{B}\urcorner|) + |\ulcorner\mathbb{B}\urcorner| \leqslant |\ulcorner\mathbb{B}\urcorner| \cdot t^2(|\ulcorner\mathbb{B}\urcorner|) \leqslant t(|\ulcorner\mathbb{B}\urcorner|)^3$$

using (c) and $t(|\ulcorner\mathbb{B}\urcorner|) \geqslant |\ulcorner\mathbb{B}\urcorner|$ (since t is time-construcible). This contradicts (b). □

Remark 1.2.13 Something stronger is known: $\mathsf{TIME}(t') \smallsetminus \mathsf{TIME}(t) \neq \varnothing$ for time-constructible t, t' with $t(n) \cdot \log t(n) \leqslant o(t'(n))$.

Corollary 1.2.14 $\mathsf{P} \subsetneq \mathsf{E} \subsetneq \mathsf{EXP}$.

Proof: Note $n^c \leqslant O(2^n)$ for every $c \in \mathbb{N}$, so $\mathsf{P} \subseteq \mathsf{TIME}(2^n)$. By the Time Hierarchy Theorem $\mathsf{E} \smallsetminus \mathsf{P} \supseteq \mathsf{TIME}(2^{6n}) \smallsetminus \mathsf{TIME}(2^n) \neq \varnothing$. Similarly, $2^{cn} \leqslant O(2^{n^2})$ for every $c \in \mathbb{N}$, so by the Time Hierarchy Theorem $\mathsf{EXP} \smallsetminus \mathsf{E} \supseteq \mathsf{TIME}(2^{6n^2}) \smallsetminus \mathsf{TIME}(2^{n^2}) \neq \varnothing$. □

Exercise 1.2.15 Find $t_0, t_1, \ldots$ such that $\mathsf{P} \subsetneq \mathsf{TIME}(t_0) \subsetneq \mathsf{TIME}(t_1) \subsetneq \cdots \subseteq \mathsf{E}$.

1.3 Circuit families

In this section we give a characterization of P by families of Boolean circuits. This is a lemma of central conceptual and technical importance. To emphasize its fundamental character we spell out the definition abstractly for any first-order structure $\mathcal{A}$ interpreting a functional language: this consists of a set $A \neq \emptyset$, a family F of function symbols f, each having an *arity* $r_f \in \mathbb{N}$, and for each $f \in F$ an *interpretation* $f^A : A^{r_f} \to A$. For $r_f = 0$ we call f a *constant* and identify f^A witH its unique value in A.

Definition 1.3.1 Let $s, n \in \mathbb{N}$, and $\bar{X} = X_1 \cdots X_n$ and $\bar{Y} = Y_1 \cdots Y_s$ be tuples of variables. An $\mathcal{A}$*-circuit* $C = C(\bar{X})$ is a sequence of s equalities $E_1, \ldots, E_s$. Each E_i has one of the following forms:

(a) $Y_i = X_j$ for some $j \in [n]$, or,

(b) $Y_i = f(Y_{i_1}, \ldots, Y_{i_{r_f}})$ for some $f \in F$ and $i_1, \ldots, i_{r_f} \in [i-1]$.

The number s is the *size* of C and denoted $|C|$. The variables $\bar{X}$ are *input variables*, the variables $\bar{Y}$ are *gates*. A gate Y_i is an *input gate* if E_i is of type (a) or (b) for a constant f.

The *depth of* C is $d(s)$ where $d(i)$ is the *depth of gate* Y_i: if E_i is of type (b) with $r_f > 0$, then $d(i) := \max\{d(i_1), \ldots, d(i_{r_f})\}$; otherwise, i.e., if Y_i is an input gate, $d(i) := 0$.

Definition 1.3.2 For $\bar{a} = a_1 \cdots a_n \in A^n$ the *computation of* C *on* $\bar{a}$ is the tuple $\bar{b} = b_1 \cdots b_s \in A^s$ such that for all $i \in [s]$ we have $b_i = a_j$ if Y_i is of type (a), and $b_i = f^A(b_{i_1}, \ldots, b_{i_{r_f}})$ if Y_i is of type (b); for a constant f this means $b_i = f^A \in A$. The *output of* C *on* $\bar{a}$ is $C(\bar{a}) := b_s$. The circuit *computes* the function $\bar{a} \mapsto C(\bar{a})$ from A^n into A.

Let $m \in \mathbb{N}$. An $\mathcal{A}$-circuit C *with* m *output gates* is an $\mathcal{A}$-circuit together with an m-tuple $\bar{o} = o_1 \cdots o_m \in [s]^m$. The *output of* C *on* $\bar{a} \in A^n$ is $C(\bar{a}) := b_{o_1} \cdots b_{o_m} \in A^m$ where $\bar{b} = b_1 \cdots b_s \in A^s$ is the computation of C on $\bar{a}$. The circuit *computes* the function $\bar{a} \mapsto C(\bar{a})$ from A^n into A^m.

It should be clear that this is well-defined: for each $\bar{a}$ there exists exactly one computation of C on $\bar{a}$.

Graphical representation A circuit $C(\bar{X})$ of size $|C| = s$ is pictured as a directed acyclic graph (V, E) with a labeling λ: the nodes are $V := [s]$ and the edges E contain (j, i) if E_i is of type (b) and Y_j appears on the r.h.s. of E_i. The label $\lambda(i)$ is X_j for Y_i of type (a), and the $f \in F$ appearing the equation for E_i of type (b).

Note, the depth of $C(\bar{X})$ is the maximal length of some path in the above graph.

We are mainly interested in *Boolean circuits*:

Definition 1.3.3 A *Boolean circuit with unbounded fan-in* is an $\mathcal{A}$-circuit where $A = \{0,1\}$ and F contains constants 0 and 1, $\neg$ of arity 1 and $\wedge_n, \vee_n$ of arity n for each $n \geqslant 2$. The interpretations are $0^A := 0, 1^A := 1, \neg^A(a) := 1 - a, \wedge_n^A(a_1, \ldots, a_n) := \min\{a_1, \ldots, a_n\}$ and $\vee^A(a_1, \ldots, a_n) := \max\{a_1, \ldots, a_n\}$.

A *Boolean circuit* or just *circuit* allows $\wedge_n, \vee_n$ only for $n = 2$. We write $\wedge, \vee$ instead $\wedge_2, \vee_2$, and write $\neg Y, Y \wedge Z, Y \vee Z$ instead $\neg(Y), \wedge(Y, Z), \vee(Y, Z)$ for variables Y, Z.

Exercise 1.3.4 The following problem is in P.

CIRCUIT-EVAL	
Input:	a string $x \in \{0,1\}^*$ and a circuit $C = C(X_1, \dots, X_{|x|})$.
Problem:	is $C(x) = 1$?

For the following circuits it is unknown whether their evaluation problem is in P. We shall study the question in Chapter 6.

Definition 1.3.5 An *arithmetical circuit* is an $\mathcal{A}$-circuit where $A = \mathbb{Z}$ and F contains constants $1, -1$ and $+, \times$ of arity 2. The interpretations are $1^A := 1, -1^A := -1, +^A(a_1, a_2) := a_1 + a_2$ and $\times^A(a_1, a_2) := a_1 \cdot a_2$.

Boolean formulas We assume basic familiarity with *(Boolean) formulas.* Recall, these are those words α over the alphabet

$$(,), 0, 1, \neg, \wedge, \vee, X_1, X_2 \dots$$

that are obtained by finitely many applications of the rules:

$$\frac{}{X_i} \quad \frac{}{0} \quad \frac{}{1} \quad \frac{\alpha}{\neg\alpha} \quad \frac{\alpha \quad \beta}{(\alpha \wedge \beta)} \quad \frac{\alpha \quad \beta}{(\alpha \vee \beta)}.$$

An *assignment* A maps the *(Boolean) variables* $X_1, X_2, \dots$ into $\{0,1\}$. It uniquely extends to a function, again denoted A, on the set of all Boolean formulas with the properties

$$A(\neg\alpha) = 1 - A(\alpha),\ A((\alpha \wedge \beta)) = \min\{A(\alpha), A(\beta)\},\ A((\alpha \vee \beta)) = \max\{A(\alpha), A(\beta)\}.$$

Clearly, $A(\alpha)$ depends only on the values of A on the variables appearing in α. If $n \in \mathbb{N}$ is the maximal index of a variable in α (and $n = 0$ if there are none), then α *computes* the function that maps $\bar{a} = a_1 \cdots a_n \in \{0,1\}^n$ to $A(\alpha) \in \{0,1\}$ where A maps every X_i to a_i. We write $\alpha(\bar{a})$ for $A(\alpha)$. If the value is 1, we write $A \models \alpha$ or $\bar{a} \models \alpha$.

Definition 1.3.6 A (Boolean) circuit or formula C with n input variables is *satisfiable* if there is $\bar{a} \in \{0,1\}^n$ such that $C(\bar{a}) = 1$; C is *satisfied* by $\bar{a}$. If $C(\bar{a}) = 1$ for all $\bar{a} \in \{0,1\}^*$, then C is *tautological.* Two circuits or formulas are *equivalent* if they compute the same function.

Proposition 1.3.7 *Let $s \in \mathbb{N}$.*

(a) For every formula α of length s there is an equivalent circuit C_α of size $\leqslant s$.

(b) For every circuit C of size s there is an equivalent formula α_C of length $\leqslant 3^s$.

Proof: For (a), let α be a formula and let $\alpha_1, \ldots, \alpha_{s'}$ enumerate its subformulas so that: if α_i is a subformula of α_j, then $i \leqslant j$. Note $s' \leqslant s$ since at each position of α starts at most one subformula. The size s' circuit C_α has i-th equation depending on the main connective of α_i: e.g., if $\alpha_i = (\alpha_j \wedge \alpha_k)$, it is $Y_i = Y_j \wedge Y_j$ (note $j, k < i$); the other cases are similar.

For (b), given $C = (E_1, \cdots, E_s)$ we define α_i equivalent to $(E_1, \ldots, E_i)$ by recursion on $i \leqslant s$. E.g., if E_i is $Y_i = Y_j \wedge Y_k$ we set $\alpha_i := (\alpha_j \wedge \alpha_k)$. The other cases are similar. Let $\ell_i := \max_{j \leqslant i} |\alpha_j|$. Then $\ell_1 = 1$ and $\ell_{i+1} \leqslant 2\ell_i + 3$. Hence $\alpha_C := \alpha_s$ has length $\leqslant \ell_s \leqslant 3^s$. □

Every $f : \{0,1\}^n \to \{0,1\}$ is computed by a formula of size $2^{O(n)}$: the disjunction of

$$\neg^{1-a_1} X \wedge \ldots \wedge \neg^{1-a_n} X_n$$

for those $\bar{a} = a_1 \cdots a_n \in \{0,1\}^n$ with $f(\bar{a}) = 1$; we write $\neg^1 \alpha := \neg\alpha, \neg^0 \alpha := \alpha$.

For circuits we can do slightly better:

Proposition 1.3.8 *For all sufficiently large n and all $m \geqslant 1$, every $f : \{0,1\}^n \to \{0,1\}^m$ is computable by a circuit of size $\leqslant 21 \cdot m \cdot 2^n/n$.*

Proof: It suffices to prove this for $m = 1$. For every $g : \{0,1\}^m \to \{0,1\}$ with $m \leqslant n$ let $s(g)$ be the minimal size of a circuit computing g. For a bit $b \in \{0,1\}$ let f^b be the function that maps $x_1 \cdots x_{n-1} \in \{0,1\}^{n-1}$ to $f(x_1 \cdots x_{n-1} b)$. Write f^{010} for $((f^0)^1)^0$ and similarly f^y for any string y of length at most n.

Observe that $s(f) \leqslant 5 + s(f^0) + s(f^1)$: the circuit "$(C_1 \wedge X_n) \vee (C_0 \wedge \neg X_n)$" computes f; it is built from 5 gates plus a circuit C_0 computing f^0 and a circuit C_1 computing f^1. Replace the circuit C_0 by 5 gates plus circuits computing f^{00} and f^{01} and similarly for C_1. This results in a circuit for f with $5 + 2 \cdot 5$ gates plus circuits for $f^{00}, f^{01}, f^{10}, f^{11}$. So $s(f)$ is bounded by $5 + 2 \cdot 5$ plus the sum of $s(g)$ for $g \in \{f^{00}, f^{01}, f^{10}, f^{11}\}$. In general,

$$s(f) \leqslant \textstyle\sum_{i=0}^{k-1} 5 \cdot 2^i + \sum_g s(g) \leqslant 5 \cdot 2^k + \sum_g s(g).$$

where $k \leqslant n$ and g ranges over $\{f^y \mid y \in \{0,1\}^k\}$; note such g has arity $n - k$. For $k := n$ every $s(g)$ is 1, so f is computed by a circuit of size $\leqslant 6 \cdot 2^n$. To get the bound claimed, set

$$k := n - \lceil \log n \rceil + 2$$

which is $\leqslant n$ for large enough n. Then the sum ranges over at most $2^{2^{\lceil \log n \rceil - 2}}$ many functions $g : \{0,1\}^{\lceil \log n \rceil - 2} \to \{0,1\}$. Each g has a circuit of size $\leqslant 6 \cdot 2^{\lceil \log n \rceil - 2} \leqslant 6 \cdot 2^{\log n + 1 - 2} = 3n$. So,

$$s(f) \leqslant 5 \cdot 2^{(n - \log n + 2)} + 2^{2^{\log n + 1 - 2}} \cdot 3n \leqslant 20 \cdot 2^n/n + 2^{n/2} \cdot 3n.$$

Our claim follows, noting $2^{n/2} \cdot 3n \leqslant 2^n/n$ for sufficiently large n. □

1.3.1 The fundamental lemma

Lemma 1.3.9 (Fundamental) *A problem Q is in P if and only if there exists a family of circuits $(C_n)_{n\in\mathbb{N}}$ such that for every $n \in \mathbb{N}$ the circuit $C_n = C_n(X_1, \dots, X_n)$ is computable from 1^n in polynomial time and* decides Q on $\{0,1\}^n$, *i.e., for all $x \in \{0,1\}^n$*

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

Proof: Assume a family $(C_n)_n$ as stated exists. Then a polynomial time algorithm for Q proceeds as follows. On input $x \in \{0,1\}^*$ it computes (an encoding of) $C_{|x|}$ and then checks whether $C_{|x|}(x) = 1$. Computing $C_{|x|}$ needs time $p(|x|)$ for some polynomial p, so the encoding of $C_{|x|}$ has length at most $p(|x|)$. By Exercise 1.3.4, the check can be done in time $q(|x| + p(|x|))$ for some polynomial q.

To see the converse direction, assume $Q \in \mathsf{P}$. By Lemma 1.1.11 there is a single-tape Turing machine $\mathbb{A} = (S, \delta)$ that decides Q and is p-time bounded for some polynomial p. We can assume that $p(n) \geqslant n$ for all $n \in \mathbb{N}$. Given $n \in \mathbb{N}$, we describe the circuit C_n. It will be obvious that it can be computed from 1^n in polynomial time.

Fix some $x \in \{0,1\}^n$ and consider the computation table $(T_{ij})_{i\in[p(n)], j\leqslant p(n)+1}$ of $\mathbb{A}$ on x up to step $p(n)$. It has entries in $\Sigma := \{0, 1, \triangleright, \square\} \cup (\{0, 1, \triangleright, \square\} \times S)$.

Entry T_{ij} of the table can be determined by $T_{(i-1)(j-1)}, T_{(i-1)j}, T_{(i-1)(j+1)}$ and the transition function of $\mathbb{A}$. In other words, there exists a function $f : \Sigma^3 \to \Sigma$ such that

$$f(T_{(i-1)(j-1)}, T_{(i-1)j}, T_{(i-1)(j+1)}) = T_{ij},$$

for all ij with $i \neq 1$ and $j \notin \{p(n)+1, 0\}$. This is referred to as "locality of computation" and constitutes the key insight behind the proof.

We write the table in binary, coding every $\sigma \in \Sigma$ by $\ulcorner\sigma\urcorner \in \{0,1\}^s$ for (the constant) $s := \lceil \log(|\Sigma| + 1) \rceil$. Choose a circuit C with $3s$ variable-gates and s output gates such that

$$C(\ulcorner\sigma_1\urcorner\ulcorner\sigma_2\urcorner\ulcorner\sigma_3\urcorner) = \ulcorner f(\sigma_1, \sigma_2, \sigma_3)\urcorner,$$

for all $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$. The size of C is constant in the sense that it does not depend on x. For each ij make a copy C_{ij} of C and identify

– its first s inputs with the outputs of $C_{(i-1)(j-1)}$,

– its second s inputs with the outputs of $C_{(i-1)j}$,

– its third s inputs with the outputs of $C_{(i-1)(j+1)}$.

For the marginal value $j = 0$ use a similar circuit with $2s$ many inputs: observe that an entry in the first column T_{i0} is determined by the two entries $T_{(i-1)0}, T_{(i-1)1}$ one row above. For the marginal value $j = p(n)+1$ you can use a circuit that constantly outputs the s bits of $\ulcorner\square\urcorner$ (recall that the rightmost column of a computation table contains only $\square$).

The circuit sofar has $p(n) + 2$ many blocks of s inputs corresponding to the encoding of the first row of the table:

$$T_{i0} \cdots T_{i(p(n)+1)} = (\triangleright, s_{\text{start}})\ x_1 \cdots x_n\ \square \cdots \square\,.$$

Identify the $(i+1)$th block of s inputs (i.e. the one corresponding to x_i) with the outputs of some fixed circuit computing $b \mapsto \ulcorner b \urcorner$ for bits $b \in \{0,1\}$; its single input variable is X_i. The first block of s inputs is assigned $0, 1$ according $\ulcorner(\triangleright, s_{\text{start}})\urcorner$ and all other blocks are assigned $0, 1$ according $\ulcorner\square\urcorner$.

This gives a a circuit that computes from x the encoding of the halting configuration of $\mathbb{A}$ on x. Finally, add a constant size circuit that maps the second block of s gates to 1 or 0 depending on whether the configuration is accepting or not. $\square$

Exercise 1.3.10 Let $f : \{0,1\}^* \to \{0,1\}^*$ be a function computable in polynomial time such that there exists a function $\ell : \mathbb{N} \to \mathbb{N}$ such that $|f(x)| = \ell(|x|)$ for all $x \in \{0,1\}^*$. Then there is a polynomial time function mapping 1^n to a circuit C_n computing $f \upharpoonright \{0,1\}^n$.

Chapter 2

Nondeterminism

2.1 NP

Definition 2.1.1 A relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is *polynomially bounded* if and only if there is a polynomial such that $|y| \leqslant p(|x|)$ for all $(x,y) \in R$.

Of course, that R is in P means that $\{\langle x,y\rangle \mid (x,y) \in R\}$ is in P. The *domain* of R is

$$dom(R) := \{x \mid \exists y \in \{0,1\}^* : (x,y) \in R\}.$$

Definition 2.1.2 *Nondeterministic polynomial time* NP is the set of problems Q such that $Q = dom(R)$ for some polynomially bounded $R \subseteq \{0,1\}^* \times \{0,1\}^*$ in P.

Exercise 2.1.3 Show that every $Q \in \mathsf{NP}$ is the domain of a binary relation R in P such that for some polynomial p we have $|y| = p(|x|)$ for all $(x,y) \in R$.

Examples 2.1.4 The problems

Circuit-Sat	
Input:	a Boolean circuit C.
Problem:	is C satisfiable?

Sat	
Input:	a propositional formula α.
Problem:	is α satisfiable?

are in NP: for the first problem let R contain the pairs (C,x) such that C is a circuit with one output and $|x|$ variable-gates and $C(x) = 1$. For the second let R contain the pairs (α, A) such that A is an assignment to the variables appearing in α that satisfies α.

As a further example, the independent set problem IS (cf. section 1.1.2) is in NP as can be seen by letting R relate pairs (G,k) (i.e. their encodings $\langle \ulcorner G \urcorner, bin(k)\rangle$) to (encodings of) X where X is an independent set of cardinality k in the graph G.

Intuitively, $(x, y) \in R$ means that y is a solution to problem instance x. To check if a given y is indeed a solution to instance x is easy (polynomial time). But, of course, it may be more difficult to find a solution y. Note there are exponentially many candidate solutions for instance x, so exhaustive search takes exponential time in the worst case:

Proposition 2.1.5 $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXP}$.

Proof: Any problem Q in P is the domain of $R = Q \times \{0\}$ and hence in NP. To see the second inclusion let $Q \in \mathsf{NP}$ and choose R in P such that $Q = dom(R)$. Further choose a polynomial p witnessing that R is polynomially bounded. Observe that given any string y it is easy to compute its successor y^+ in the lexicographical order $\lambda, 0, 1, 00, 01, \ldots$ (where λ denotes the empty string). Consider the algorithm that on input x proceeds as follows:

```
1. $y \leftarrow \lambda$
2. if $(x, y) \in R$ then accept
3. $y \leftarrow y^+$
4. if $|y| \leqslant p(|x|)$ goto line 2
5. reject
```

Lines 2-4 can be executed in polynomial time, say time $q(|x|)$. The number of times they are executed is bounded by the number of strings y of length $\leqslant p(|x|)$. In total, the running time is $O(q(|x|) \cdot 2^{p(|x|)+1})$ which is $O(2^{|x|^c})$ for some suitable $c \in \mathbb{N}$. □

It is not known whether any of these inclusions is strict. But recall that we know $\mathsf{P} \neq \mathsf{EXP}$ by Corollary 1.2.14.

2.2 Nondeterministic time

We now give a machine characterization of NP.

Definition 2.2.1 Let $k \in \mathbb{N}, k > 0$. A *k-tape nondeterministic Turing machine* is a triple (S, δ_0, δ_1) such that both (S, δ_0) and (S, δ_1) are k-tape Turing machines. A *configuration of* $\mathbb{A}$ is a configuration of (S, δ_0) (which is also one of (S, δ_1)), and a *successor configuration of* $\mathbb{A}$ of another configuration of $\mathbb{A}$ is one of (S, δ_0) or (S, δ_1). A *run* or *computation* of $\mathbb{A}$ is a sequence of configurations of $\mathbb{A}$ each besides the first being a successor of the previous one. It is *on* $x \in \{0,1\}^*$ if the first is the *start configuration of* $\mathbb{A}$ *on* x, namely the start configuration of (S, δ_0) on x (which equals that of (S, δ_1)). It is it is *complete* if it ends in a *halting configuration of* $\mathbb{A}$ which is a halting configuration of (S, δ_0) (equivalently, of (S, δ_1)). It is *accepting (rejecting)* if cell 1 contains 1 (contains 0) in the halting configuration. $\mathbb{A}$ *accepts* x if there is an accepting run of $\mathbb{A}$ on x. $\mathbb{A}$ *accepts* the problem

$$L(\mathbb{A}) := \{x \mid \mathbb{A} \text{ accepts } x\}.$$

For $t : \mathbb{N} \to \mathbb{N}$ we say $\mathbb{A}$ is *t-time bounded* if all complete runs of $\mathbb{A}$ on x with the halting configuration not repeated have length at most $t(|x|)$. Being *polynomial time bounded* means that this holds for some polynomial t.

In general there are many complete runs of $\mathbb{A}$ on x. The intuition is that at every configuration $\mathbb{A}$ nondeterministically chooses which one of the two transition functions to apply. An input is accepted if there exist choices causing it to accept. The idea is that these choices correspond to guessing a solution that is then deterministically checked. If there exists a solution, then there exists an accepting run, otherwise the check will fail no matter what has been guessed. This is sometimes referred to as the "guess and check" paradigm. We introduce the following handy mode of speech, continuing the above definition:

Definition 2.2.2 Let $t \in \mathbb{N}$ and let ρ be a run of length $t > 0$ of $\mathbb{A} = (S, \delta_0, \delta_1)$ on x. A string $y = y_1 \cdots y_{|y|} \in \{0,1\}^*$ of length $|y| \geqslant t - 1$ is said to *determine* ρ if the $(i+1)$-th configuration is a successor configuration of (S, δ_{y_i}).

Obviously, any string determines at most one complete run. As said above, the intuitive idea is that strings encoding accepting runs "are" solutions.

Proposition 2.2.3 *Let Q be a problem. The following are equivalent.*

1. *$Q \in \mathsf{NP}$.*
2. *There exists a polynomial p and a nondeterministic Turing machine $\mathbb{A}$ such that*
$$Q = \{x \in \{0,1\}^* \mid \text{ there is an accepting run of } \mathbb{A} \text{ on } x \text{ of length} \leqslant p(|x|)\}.$$
3. *There exists a polynomial time bounded nondeterministic Turing machine that accepts Q.*

Proof: (1) implies (2): this follows the mentioned "guess and check" paradigm. Choose a polynomially bounded relation R such that $Q = dom(R)$ and consider the following nondeterministic machine $\mathbb{A}$. On input x it guesses a string y and checks whether $(x,y) \in R$. More precisely, take $\mathbb{A}$ as a nondeterministic machine with input tape and one work tape. It has states $s_{\text{guess}}, s_?, s_{\text{check}}$ and two transition functions δ_0, δ_1 such that for $b \in \{0,1\}$

$$\begin{aligned} \delta_b(s_{\text{start}}, \triangleright\triangleright) &= (s_?, \triangleright\triangleright, 01), \\ \delta_0(s_?, \triangleright\Box) &= (s_{\text{guess}}, \triangleright\Box, 00), \\ \delta_1(s_?, \triangleright\Box) &= (s_{\text{check}}, \triangleright\Box, 00), \\ \delta_b(s_{\text{guess}}, \triangleright\Box) &= (s_?, \triangleright b, 01). \end{aligned}$$

Upon reaching state s_{check} some binary string y has been written on the worktape. From s_{check} both transition functions move to the initial state of a polynomial time Turing machine that checks whether the pair (x,y) is in R.

It is clear, that $\mathbb{A}$ accepts only inputs that are in Q. Conversely, let $x \in Q$ and choose a nondecreasing polynomial q witnessing that R is polynomially bounded. Then there is y of length $|y| \leqslant q(|x|)$ such that $(x, y) \in R$. There exists a run of $\mathbb{A}$ that in $1+2|y|+1 \leqslant O(q(|x|))$ steps writes y on the tape and then moves to s_{check}. After that $(x, y) \in R$ is verified in time $p(|x| + |y|) \leqslant p(|x| + q(|x|))$ for a suitable polynomial p.

(2) implies (3): given $\mathbb{A}$ and p as in (2) let the machine $\mathbb{B}$ on x simulate $\mathbb{A}$ for $p(|x|)$ many steps. If the simulation halts and accepts, $\mathbb{B}$ accepts. Otherwise $\mathbb{B}$ rejects.

(3) implies (1): let p be a polynomial and $\mathbb{A}$ be a p-time bounded nondeterministic machine accepting Q. Let R contain the pairs (x, y) such that $|y| = p(|x|)$ and y determines an accepting run of $\mathbb{A}$ on x. Then R is in P, polynomially bounded and has domain Q. □

Remark 2.2.4 The above proof implies the claim in Exercise 2.1.3.

Definition 2.2.5 Let $t : \mathbb{N} \to \mathbb{N}$. By $\mathsf{NTIME}(t)$ we denote the class of problems Q such that there is some $c \in \mathbb{N}$ and some $c \cdot t + c$-time bounded nondeterministic Turing machine $\mathbb{A}$ that accepts Q. The classes

$$\begin{array}{lcl} \mathsf{NE} & := & \bigcup_{c\in\mathbb{N}} \mathsf{NTIME}(2^{c\cdot n}) \\ \mathsf{NEXP} & := & \bigcup_{c\in\mathbb{N}} \mathsf{NTIME}(2^{n^c}) \end{array}$$

are called *nondeterministic simply exponential time* and *nondeterministic exponential time* respectively.

In this notation, the equivalence of (1) and (3) in Proposition 2.2.3 reads:

Corollary 2.2.6 $\mathsf{NP} = \bigcup_{c\in\mathbb{N}} \mathsf{NTIME}(n^c)$.

Exercise 2.2.7 Generalize Proposition 2.1.5 by showing the following. If $t : \mathbb{N} \to \mathbb{N}$ is time-constructible, then $\mathsf{NTIME}(t) \subseteq \bigcup_{c\in\mathbb{N}} \mathsf{TIME}(2^{c\cdot t})$.

Proposition 2.2.8 *If* $\mathsf{P} = \mathsf{NP}$*, then* $\mathsf{E} = \mathsf{NE}$*.*

Proof: The proof is done by so-called "padding": make the input artificially long. Assume $\mathsf{P} = \mathsf{NP}$ and let $Q \in \mathsf{NE}$. Choose a nondeterministic machine $\mathbb{A}$ that accepts Q and is $(c \cdot 2^{c\cdot n} + c)$-time bounded for some $c \in \mathbb{N}$. We claim $Q \in \mathsf{E}$. Consider the problem

$$Q' := \left\{\langle x, 1^{2^{|x|}}\rangle \mid x \in Q\right\}.$$

The following machine witnesses that $Q' \in \mathsf{NP}$: it first checks in polynomial time that the input is of the form $\langle x, 1^{2^{|x|}}\rangle$. If this is not the case it rejects. Otherwise it runs $\mathbb{A}$ on x - these are at most $c \cdot 2^{c\cdot|x|} + c \leqslant c \cdot |\langle x, 1^{2^{|x|}}\rangle|^c + c$ many steps.

By assumption $Q' \in \mathsf{P}$. To decide Q, proceed as follows. On input x, simulate a polynomial time machine for Q' on $y := \langle x, 1^{2^{|x|}}\rangle$. The computation of y takes time $2^{O(|x|)}$ and the simulation takes time polynomial in $|\langle x, 1^{2^{|x|}}\rangle|$, so at most $2^{O(|x|)}$. □

Exercise 2.2.9 Prove: if $\mathsf{E} = \mathsf{NE}$, then $\mathsf{EXP} = \mathsf{NEXP}$.

2.2.1 Nondeterministic time hierarchy

Theorem 2.2.10 (Nondeterministic time hierarchy) *Assume $t : \mathbb{N} \to \mathbb{N}$ is time-constructible and increasing and let $t' : \mathbb{N} \to \mathbb{N}$ be given by $t'(n) := t(n+1)^6$. Then*

$$\mathsf{NTIME}(t') \smallsetminus \mathsf{NTIME}(t) \neq \varnothing.$$

Proof: Let $\mathbb{A}_0, \mathbb{A}_1, \dots$ be an enumeration of all nondeterministic single-tape Turing machines such that for every such machine $\mathbb{A}$ there are infinitely many i such that $\mathbb{A}_i = \mathbb{A}$ and such that the map $1^i \mapsto \ulcorner \mathbb{A}_i \urcorner$ is computable in time $O(i)$.

The proof is by so-called "slow" or "lazy" diagonalization. Let $f : \mathbb{N} \to \mathbb{N}$ be an increasing function to be determined later. The "diagonalizing" machine $\mathbb{D}$ on input 1^n with n in the intervall between $f(i)$ and $f(i+1)$ tries to "diagonalize" against $\mathbb{A}_i$:

```
$\mathbb{D}(1^n)$

1. $i \leftarrow \min\{j \in \mathbb{N} \mid n \leqslant f(j+1)\}$.
2. **if** $n < f(i+1)$ **then**
3.    simulate $t(n+1)^3$ steps of $\mathbb{A}_i$ on $1^{n+1}$
4.    **if** the simulated run is complete and accepts **then** accept.
5.    reject
6. **if** $n = f(i+1)$ **then for all** $y \in \{0,1\}^{t(f(i)+1)^3}$
7.    **if** $y$ determines an accepting run of $\mathbb{A}_i$ on $1^{f(i)+1}$ **then** reject
8. accept
```

Whatever f is, we claim $L(\mathbb{D}) \notin \mathsf{NTIME}(t)$. Otherwise there are $c, i_0 \in \mathbb{N}$ such that $L(\mathbb{A}_{i_0}) = L(\mathbb{D})$ and $\mathbb{A}_{i_0}$ is $(c \cdot t^2 + c)$-time bounded (Lemma 1.1.11 holds for nondeterministic machines). We can assume i_0 is sufficiently large such that $f(i_0) \geqslant c$ (note $f(n) \geqslant n$ since f is increasing). For $n > f(i_0)$, the machine $\mathbb{A}_{i_0}$ on 1^n halts within $c \cdot t^2(n) + c < nt^2(n) - t^2(n) + n \leqslant t^3(n)$ steps.

We claim that for all $f(i_0) < n < f(i_0+1)$:

$$\mathbb{A}_{i_0} \text{ accepts } 1^n \quad \Longleftrightarrow \quad \mathbb{A}_{i_0} \text{ accepts } 1^{n+1}.$$

Indeed, both sides are equivalent to $\mathbb{D}$ accepting 1^n: the l.h.s. by assumption $L(\mathbb{A}_{i_0}) = L(\mathbb{D})$, and the r.h.s. because the run simulated in line 3 is complete. Thus

$$\mathbb{A}_{i_0} \text{ accepts } 1^{f(i_0)+1} \quad \Longleftrightarrow \quad \mathbb{A}_{i_0} \text{ accepts } 1^{f(i_0+1)}.$$

But lines 7 and 8 ensure that the l.h.s. is equivalent to $\mathbb{D}$ rejecting $1^{f(i_0+1)}$. Thus, $\mathbb{D}$ and $\mathbb{A}_{i_0}$ answer differently on $1^{f(i_0+1)}$, a contradiction.

It remains to choose f such that $\mathbb{D}$ on 1^n halts in time $O(t(n+1)^6)$. If we choose f time-constructible, then lines 1 and 2 and can be executed in time $O(n^2)$ (Exercise below).

Lines 3-5 need time $O(t(n+1)^3)$ to compute $t(n+1)^3$, time $O(i) \leqslant O(n)$ to compute $\ulcorner \mathbb{A}_i \urcorner$ and then time $O((i+t(n+1)^3)^2)$ for the simulation: guess a binary string of length $t(n+1)^3$ and simulate the (possibly partial) run of $\mathbb{A}$ on 1^{n+1} determined by it using the universal machine from Theorem 1.2.8. Similarly, line 7 needs time $O((i+t(f(i)+1)^3)^2)$ and is executed up to $2^{t(f(i)+1)^3}$ many times. Note this is done only if $n = f(i+1)$. This time is $O(n)$ if f satisfies $2^{t(f(i)+1)^3} \cdot (i + t(f(i)+1)^3)^2) \leqslant O(f(i+1))$. Then $\mathbb{D}$ halts in time $O(t(n+1)^6)$ as desired. By Exercise 1.2.11, a function f as required exists. □

Exercise 2.2.11 Let $t : \mathbb{N} \to \mathbb{N}$ be time-constructible and increasing. Show that the function $1^n \mapsto \min\{j \in \mathbb{N} \mid n \leqslant t(j+1)\}$ can be computed in time $O(n^2)$.

Hint: compute $t(0), t(1), \ldots$ until the computation of $t(i)$ does not halt in $c \cdot n$ steps or halts with output $\geqslant n$; here, c is a suitable constant.

Remark 2.2.12 Theorem 2.2.10 is not optimal: it suffices to assume $t(n+1) \leqslant o(t'(n))$.

As in Corollary 1.2.14 it is now easy to derive the following.

Corollary 2.2.13 $\mathsf{NP} \subsetneq \mathsf{NE} \subsetneq \mathsf{NEXP}$.

2.3 NP-completeness

2.3.1 Polynomial time reductions

Intuitively, finding in a given graph G a set of pairwise non-adjacent vertices of a given size k (i.e. an independent set) is not harder than the problem of finding a set of pairwise adjacent vertices of size k, i.e. a *clique*. An independent set in G is the same as a clique in the dual graph $\overline{G}$ which has the same vertices as G and an edge between two distinct vertices precisely if G does not. So instead of deciding IS on instance (G, k) we may decide Clique on $(\overline{G}, k)$:

Clique	
Input:	a graph G and $k \in \mathbb{N}$.
Problem:	does G contain a clique of size k?

Clearly, this 'refomulation' of the question can be done in polynomial time. We compare the difficulty of problems according to the availability of such 'refomulations':

Definition 2.3.1 Let Q, Q' be problems. A function $r : \{0,1\}^* \to \{0,1\}^*$ is a *reduction from Q to Q'*, symbolically $r : Q \leqslant_p Q'$, if for all $x \in \{0,1\}^*$

$$x \in Q \iff r(x) \in Q'.$$

It is *polynomial time* if it is so computable. If such a reduction exists, we write $Q \leqslant_p Q'$ and say Q is *polynomial time reducible* to Q'. If both $Q \leqslant_p Q'$ and $Q' \leqslant_p Q$, we write $Q \equiv_p Q'$ and say Q and Q' are *polynomial time equivalent.*

It is easy to see that $\leqslant_p$ is transitive and that $\equiv_p$ is an equivalence relation.

Example 2.3.2 IS $\equiv_p$ Clique.

Proof: Mapping a string encoding a pair (G, k) of a graph G and $k \in \mathbb{N}$ to an encoding of $(\overline{G}, k)$, and other strings to themselves witnesses both IS $\leqslant_p$ Clique and Clique $\leqslant_p$ IS.□

Definition 2.3.3 A set $\mathsf{C} \subseteq P(\{0,1\}^*)$ is *closed under* $\leqslant_p$ if $Q \leqslant_p Q' \in \mathsf{C}$ implies $Q \in \mathsf{C}$ for all problems Q, Q'. A problem Q is C-*hard (under* $\leqslant_p$*)* if $Q' \leqslant_p Q$ for every $Q' \in \mathsf{C}$; if additionally $Q \in \mathsf{C}$, then Q is C-*complete (under* $\leqslant_p$*)*.

Exercise 2.3.4 Assume $\mathsf{C} \subseteq P(\{0,1\}^*)$ is closed under $\leqslant_p$ and show the following. Any two C-complete problems are polynomial time equivalent. Given two polynomial time equivalent problems, one is C-complete if and only if so is the other. Any problem to which some C-hard problem is polynomial time reducible, is also C-hard.

Lemma 2.3.5 P, NP, EXP, NEXP *are closed under* $\leqslant_p$.

Proof: We treat NP, the other classes are treated similarly. Assume $Q \leqslant_p Q' \in \mathsf{NP}$ and let $\mathbb{A}'$ be a p-time bounded nondeterministic Turing machine accepting Q' where p is some non-decreasing polynomial (Corollary 2.2.6). Let r be a polynomial time reduction from Q to Q'. There is a polynomial q such that $|r(x)| \leqslant q(|x|)$ for all $x \in \{0,1\}^*$. Now consider the following nondeterministic machine $\mathbb{A}$: on input x, it computes $r(x)$ and then runs $\mathbb{A}'$ on $r(x)$. Computing r can be done in polynomial time and then $\mathbb{A}'$ takes at most $p(q(|x|))$ many steps. Hence, $\mathbb{A}$ is polynomial time bounded. Further, $\mathbb{A}$ accepts x if and only if $\mathbb{A}'$ accepts $r(x)$, if and only if $r(x) \in Q'$ (since $\mathbb{A}'$ accepts Q'), if and only if $x \in Q$ (since r is a reduction). Thus, $\mathbb{A}$ witnesses that $Q \in \mathsf{NP}$. □

Corollary 2.3.6 *Let Q be a problem.*

1. *If Q is* NP*-hard, then $Q \in$* P *only if* P = NP.
2. *If Q is* NP*-complete, then $Q \in$* P *if and only if* P = NP.

Having an NP-complete problem we can thus reformulate the question whether P = NP as a question whether this concrete problem can be solved in polynomial time. The following exercise asks to prove that NP-complete problems exist. More surprisingly, we shall see that many *natural* problems are NP-complete.

Exercise 2.3.7 Show that the following problem is NP-complete.

Bounded Halting	
Input:	a nondeterministic Turing machine $\mathbb{A}$, $x \in \{0,1\}^*$ and 1^t for some $t \in \mathbb{N}$.
Problem:	is there an accepting run of $\mathbb{A}$ on x of length t?

Exercise 2.3.8 Use padding to show that a problem is E-hard if and only if it is EXP-hard. Show that Exp Halt is in E and EXP-complete under $\leqslant_p$. Further, for every problem in E there is a linear time computable reduction to Exp Halt.

Exp Halt	
Input:	a single-tape Turing machine $\mathbb{A}$, $x \in \{0,1\}^*$, $t \in \mathbb{N}$ (in binary).
Problem:	does $\mathbb{A}$ accept x in $\leqslant t$ steps?

2.3.2 The Cook-Levin theorem

Theorem 2.3.9 (Cook 1971, Levin 1973) 3Sat *is* NP*-complete.*

3Sat	
Input:	a 3-CNF α.
Problem:	is α satisfiable?

A k*-CNF* is a conjunction of k*-clauses*, i.e. disjunctions of at most k literals.

Lemma 2.3.10 Circuit-Sat $\leqslant_p$ 3Sat.

Proof: Let $C = (E_1, \dots, E_s)$ be a circuit, say, with input variables $X_1, \dots, X_n$. For $i \in [s]$ let the formula α_i express E_i, that is, if E_i is $Y_i = Y_j \wedge Y_k, Y_i = Y_j \vee Y_k, Y_i = \neg Y_j, Y_i = X_j, Y_i = 0$ or $Y_i = 1$, then α_i is $(Y_i \leftrightarrow Y_j \wedge Y_k), (Y_i \leftrightarrow Y_j \vee Y_k), (Y_i \leftrightarrow \neg Y_j), (Y_i \leftrightarrow X_j), (Y_i \leftrightarrow 0)$ or $(Y_i \leftrightarrow 1)$, respectively. Clearly, an assignment A satisfies these formulas if and only if $y := A(Y_1) \cdots A(Y_s) \in \{0,1\}^s$ is a computation of C on $x := A(X_1) \cdots A(X_n) \in \{0,1\}^n$. Hence their conjunction together with Y_s is satisfiable if and only if so is C.

To get to 3Sat replace $(Y_i \leftrightarrow Y_j \wedge Y_k)$ by the three clauses $(\neg Y_i \vee Y_j), (\neg Y_i \vee Y_k)$ and $(\neg Y_j \vee \neg Y_k \vee Y_i)$, and similarly for the other equivalences. □

Corollary 2.3.11 Circuit-Sat $\leqslant_p$ NaeSat.

NaeSat	
Input:	a 3-CNF α.
Problem:	is there a satisfying assignment of α that does not satisfy all literals in any clause of α?

Proof: Given a circuit C compute α_C as in the previous proof. Let X_0 be a new variable. The 3-CNF β_C is obtained from α_C by adding X_0 to every clause of α_C with < 3 literals. We claim β_C is a "yes" instance of NaeSat if and only if C is satisfiable.

Assuming the former, let A be an assignment witnessing this. We can assume $A(X_0) = 0$ (otherwise switch to the assignment $1 - A$). But then A satisfies α_C, so C is satisfiable. Conversely, assume C is satisfiable. Then so is α_C, and extending this assignment by mapping X_0 to 0, we get an assignment A satisfying β_C. We have to check that all 3-clauses of α_C contain a literal that is false under A. This follows by inspection. □

Proof of Theorem 2.3.9: It is clear that 3Sat is in NP (see Example 2.1.4). To show it is NP-hard, by Lemma 2.3.10 (and Exercise 2.3.4) it suffices to show that Circuit-Sat is NP-hard. That is, given $Q \in$ NP, we have to show $Q \leqslant_p$ Circuit-Sat.

Choose a relation R in P such that $Q = dom(R)$ and $(x, y) \in R$ only if $|y| = p(|x|)$ for some polynomial p (Remark 2.2.4). That R is in P means $\{\langle x, y\rangle \mid (x, y) \in R\} \in$ P. Recall that $\langle x, y\rangle$ is the string $x_1 x_1 \cdots x_{|x|} x_{|x|} 01 y_1 y_1 \cdots y_{|y|} y_{|y|}$ of length $2|x| + 2 + 2|y|$.

We describe a polynomial time reduction from Q to Circuit-Sat. On input $x \in \{0,1\}^n$ compute a circuit that decides R on $\{0,1\}^{2n+2+2p(n)}$, that is, a circuit $C(Z_1,\ldots,Z_{2n+2+2p(n)})$ such that for all $x \in \{0,1\}^n$ and $y \in \{0,1\}^{p(n)}$

$$C(\langle x,y\rangle) = 1 \iff (x,y) \in R.$$

This can be done in polynomial time by the Fundamental Lemma 1.3.9. The circuit

$$D_x(Y_1,\ldots,Y_n) := C(x_1,x_1,\ldots,x_n,x_n,0,1,Y_1,Y_1,\ldots,Y_{p(n)},Y_{p(n)}),$$

is obtained from C by replacing Z_is by x_is, constants and variables Y_is as indicated. Then

$$D_x(y) = 1 \iff C(\langle x,y\rangle) = 1,$$

for all $y \in \{0,1\}^{p(n)}$. It follows that D_x is satisfiable if and only if $(x,y) \in R$ for some $y \in \{0,1\}^{p(n)}$, if and only if $x \in dom(R) = Q$. Thus, $x \mapsto D_x$ is a reduction as desired. □

Corollary 2.3.12 NaeSat, Circuit-Sat, Sat *and* 3Sat *are* NP*-complete.*

Proof: Note 3Sat $\leqslant_p$ Sat $\leqslant_p$ Circuit-Sat $\leqslant_p$ NaeSat. The first two are trivial, the third is Corollary 2.3.11. By Theorem 2.3.9 (and Exercise 2.3.4) all these problems are NP-hard. It is easy to see that they belong to NP. □

The result that 3Sat is NP-complete is complemented by the following

Theorem 2.3.13 2Sat $\in$ P.

2Sat	
Input:	a 2-CNF α.
Problem:	is α satisfiable?

Proof: Let an input, a 2-CNF α be given, say it has variables $X_1,\ldots,X_n$. We can assume that every clause in α contains two (not necessarily distinct) literals. Consider the folowing directed graph $G(\alpha) := (V,E)$, where

$$\begin{aligned} V &:= \{X_1,\ldots,X_n\} \cup \{\neg X_1,\ldots,\neg X_n\},\\ E &:= \{(\lambda,\lambda') \in V^2 \mid \alpha \text{ contains } (\overline{\lambda} \vee \lambda') \text{ or } (\lambda' \vee \overline{\lambda})\}. \end{aligned}$$

Here, $\overline{\lambda}$ denotes the complementary literal of λ. Note this graph has the following curious symmetry: if there is an edge from λ to λ' then also from $\overline{\lambda'}$ to $\overline{\lambda}$.

Claim: α is satisfiable if and only if there is no vertex $\lambda \in V$ such that $G(\alpha)$ contains paths from λ to $\overline{\lambda}$ and from $\overline{\lambda}$ to λ.

The r.h.s. can be checked in polyomial time using a polynomial time algorithm for Reachability (Example 1.2.5). So we are left to prove the claim.

If there is a path from λ to λ' then α logically implies $(\lambda \to \lambda')$. This implies the forward direction. Conversely, assume the r.h.s.. We construct a satisfying assignment in stages. At every stage we have a partial assignment A that

(a) does not falsify any clause from α,

(b) for all $\lambda \in V$ and all $\lambda' \in V$ reachable from λ: if $A \models \lambda$, then $A \models \lambda'$.

Here, $A \models \lambda$ means that A is defined on the variable occuring in λ and satisfies λ. We start with $A := \emptyset$. If $dom(A) \neq \{X_1, \ldots, X_n\}$, choose $i \in [n]$ such that A is undefined on X_i. Set

$$\lambda_0 := \begin{cases} \neg X_i & \text{if there is a path from } X_i \text{ to } \neg X_i, \\ X_i & \text{else.} \end{cases}$$

If λ is reachable from λ_0, then $A \not\models \overline{\lambda}$: otherwise $\overline{\lambda_0}$ would be evaluated by A because it is reachable from $\overline{\lambda}$ by the curious symmetry. Furthermore, not both λ and $\overline{\lambda}$ can be reachable from λ_0. Otherwise there is a path from λ_0 to $\overline{\lambda_0}$ (by the curious symmetry). Now, if $\lambda_0 = X_i$ this would be a path from X_i to $\neg X_i$, contradicting the definition of λ_0. Hence $\lambda_0 = \neg X_i$ and there is a path from X_i to $\neg X_i$ (definition of λ_0). But then the path from $\lambda_0 = \neg X_i$ to $\overline{\lambda_0} = X_i$ contradicts our assumption (the r.h.s. of the claim).

It follows that there exists a minimal assignment A' that extends A and satisfies λ_0 and all λ reachable from λ_0. Then, A' has property (b). To see (a), let $(\lambda \vee \lambda')$ be a clause from α and assume $A' \models \overline{\lambda}$. But there is an edge from $\overline{\lambda}$ to λ', so $A' \models \lambda'$ by (b).

After $\leqslant n$ stages, A is defined on all $X_1, \ldots, X_n$ and A satisfies α by (a). □

Exercise 2.3.14 The above proof shows that the following search problem associated with 2SAT is solvable in polynomial time (cf. Section 1): given a 2CNF α, compute a satisfying assignment of α in case there is one, and reject otherwise.

Sketch of a second proof: Observe that applying the resolution rule to two 2-clauses again gives a 2-clause. In n variables there are at most $(2n)^2$ many 2-clauses. View a given 2-CNF as a set Γ of 2-clauses and, by subsequently applying the Resolution rule, compute the set Γ^* of all 2-clauses that can be derived from Γ in Resolution. Then Γ^* contains the empty clause if and only if the given 2-CNF is unsatisfiable. □

2.4 NP-completeness – examples

Example 2.4.1 IS *is* NP*-complete.*

Proof: We already noted that IS $\in$ NP in Examples 2.1.4. It thus suffices to show 3SAT $\leqslant_p$ IS (by Theorem 2.3.9). Let α be a 3-CNF, say with m many clauses. We can assume that each clause has exactly 3 (not necessarily distinct) literals. Consider the following graph $G(\alpha)$. For each clause $(\lambda_1 \vee \lambda_2 \vee \lambda_3)$ of α the graph $G(\alpha)$ contains a triangle with nodes corresponding to the three literals. Note that any cardinality m independent set X of $G(\alpha)$ has to contain exactly one vertex in each triangle. Add an edge between any two vertices corresponding to complementary literals. Then there exists an assignment for the variables in α that satisfies those λ that correspond to vertices in X. Thus if $(G(\alpha), m)$

is a "yes"-instance of IS, then α is satisfiable. Conversely, if A is a satisfying assignment of α, then choose from each triangle a node corresponding to a satisfied literal – this gives an independent set of cardinality m. □

By Example 2.3.2 we know CLIQUE $\equiv_p$ IS. Recalling Exercise 2.3.4 we get:

Example 2.4.2 CLIQUE *is* NP-*complete.*

Example 2.4.3 VC *is* NP-*complete.*

VC	
Input:	a graph $G = (V, E)$ and $k \in \mathbb{N}$.
Problem:	does G contain a cardinality k *vertex cover*, i.e., a set of vertices containing at least one endpoint of every edge ?

Proof: It is easy to see that VC $\in$ NP. To show NP-hardness it suffices to show IS $\leqslant_p$ VC. But note that X is an independent set in a graph $G = (V, E)$ if and only if $V \setminus X$ is a vertex cover of G. Hence $((V, E), k) \mapsto ((V, E), |V| - k)$ is a reduction as desired. □

Example 2.4.4 3COL *is* NP-*complete.*

3COL	
Input:	a graph G.
Problem:	is G *3-colourable*, that is, is there a function $f : V \to \{0, 1, 2\}$ such that $f(v) \neq f(w)$ for all $(v, w) \in E$?

Proof: It is easy to see that 3COL $\in$ NP. To show NP-hardness it suffices by Corollary 2.3.12 to show NAESAT $\leqslant_p$ 3COL. Given a 3-CNF α we compute the following graph $G = (V, E)$. Take as "starting" vertices the literals over variables in α plus an additional vertex v. For each variable X in α form a triangle on $\{v, X, \neg X\}$. Then add a triangle for each clause $(\lambda_1 \vee \lambda_2 \vee \lambda_3)$ of α with vertices corresponding to $\lambda_1, \lambda_2, \lambda_3$. Here we assume that every clause of α contains exactly three (not necessarily distinct) literals. Add an edge from a vertex corresponding to λ_i to its complementary literal $\overline{\lambda_i}$ among the "starting" vertices.

Assume this graph is 3-colorable, say, witnessed by f. Permuting colours, we can assume $f(v) = 2$. Then f (restricted to the variables) is an assignment. Consider a triangle corresponding to a 3-clause $(\lambda_1 \vee \lambda_2 \vee \lambda_3)$; the colour of the vertex for λ_i is either its truth value or 2; as all colours appear, the clause contains a verified and a falsified literal. Hence α is a "yes"-instance of NAESAT.

Conversely, let α be a "yes"-instance of NAESAT, and let the assignment A witness this. In the graph, colour variables according A and v by 2. Every 3-clause contains a true and a false literal and we colour the corresponding vertices in the triangle by this truth value; the third vertex gets colour 2. This shows the graph is 3-colourable □

Exercise 2.4.5 For $k \in \mathbb{N}$ define kCOL similarly as 3COL but with k instead of 3 colours. Show that kCOL is NP-complete for $k \geqslant 3$ and in P for $k < 3$.

Given boys and girls and homes plus restrictions of the form (boy, girl, home) – can you locate every boy with a unique girl in some unique home?

Example 2.4.6 Tripartite Matching *is* NP*-complete.*

Tripartite Matching	
Input:	Sets B, G, H and a relation $R \subseteq B \times G \times H$.
Problem:	is there a *perfect trimatching* of R, i.e. a set $R_0 \subseteq R$ with $\|R_0\| = \|B\|$ such that for distinct $(b, g, h), (b', g', h') \in R_0$ we have $b \neq b', g \neq g', h \neq h'$?

Proof: It is easy to see that Tripartite Matching $\in$ NP. To show NP-hardness we reduce from 3Sat. Let a 3-CNF α be given. We can assume that every literal occurs at most 2 times in α. To ensure this, replace every occurrence of a literal λ by a new variable Y and add clauses expressing $(Y \leftrightarrow \lambda)$.

For every variable X in α, introduce four homes $h_0^X, h_1^X, h_0^{\neg X}, h_1^{\neg X}$ (corresponding to the occurrences of literals with variable X), boys b_0^X, b_1^X and girls g_0^X, g_1^X with triples

$$(b_0^X, g_0^X, h_0^X), (b_1^X, g_0^X, h_0^{\neg X}), (b_1^X, g_1^X, h_1^X), (b_0^X, g_1^X, h_1^{\neg X}).$$

For boy b_0^X one may choose girl g_0^X with home h_0^X or girl g_1^X with home $h_1^{\neg X}$. For boy b_1^X one may choose girl g_0^X with home $h_0^{\neg X}$ or girl g_1^X with home h_1^X. As the two boys want different girls either a X-home or a $\neg X$-home is chosen but not both. This way the choice for b_0^X, b_1^X determines an assignment for X: map X to 0 if a X-home is chosen and to 1 if a $\neg X$ home is chosen. Thereby, unoccupied homes correspond to satisfied occurrences of literals. To ensure that the assignment resulting from a perfect trimatching satisfies α, add for every clause C in α a boy b_C, a girl g_C and as possible homes for the two the ones corresponding to the occurrences of literals in C.

The relation constructed has a perfect trimatching if and only if α is satisfiable. □

Example 2.4.7 3-Set Cover *is* NP*-complete.*

3-Set Cover	
Input:	A family F of 3-element sets.
Problem:	is there a disjoint subfamily $F_0 \subseteq F$ with $\bigcup F_0 = \bigcup F$?

Proof: It is easy to see that 3-Set Cover $\in$ NP. To show NP-hardness we reduce from Tripartite Matching. Given an instance $R \subseteq B \times G \times H$ output some fixed "no" instance of 3-Set Cover in case $|B| > |G|$ or $|B| > |H|$. Otherwise $|B| \leqslant |G|, |H|$. Then add new boys B_0 to B and equally many new homes H_0 to H and $B_0 \times G \times H_0$ to R to get an equivalent instance with equal number m of boys and girls and a possibly larger number ℓ of homes. Then add new boys $b_1, \ldots, b_{\ell-m}$ and girls $g_1, \ldots, g_{\ell-m}$ and triples (b_i, g_i, h) for every home h. We arrive at an equivalent instance $R' \subseteq B' \times G' \times H'$ of Tripartite Matching where $|B'| = |G'| = |H'|$. Clearly, we can assume that B', G', H' are pairwise disjoint. Then, mapping R' to $F := \{\{b, g, h\} \mid (b, g, h) \in R'\}$ gives the desired reduction. □

Example 2.4.8 DS *is* NP*-complete.*

DS	
Input:	a graph G and $k \in \mathbb{N}$.
Problem:	does G contain a cardinality k *dominating set*, i.e. a subset X of vertices such that each vertex outside X has a neighbor in X?

Proof: It is easy to see that DS $\in$ NP. To show NP-hardness we reduce from 3-Set Cover. Given an instance F of 3-Set Cover let the graph $G(F)$ have vertices $F \cup \bigcup F$ and two kinds of edges: an edge between any two vertices in F plus edges $(X, u) \in F \times \bigcup F$ if $u \in X$. We can assume that $|\bigcup F|$ is divisible by 3 (otherwise F is a "no"-instance) and $\bigcup F$ is disjoint from F. If $G(F)$ has a dominating set $D \subseteq F \cup \bigcup F$ of cardinality k, then replace every $u \in \bigcup F$ by an element $X \in F$ containing u, to obtain a dominating set $D' \subseteq F$ of size at most k. Note $\bigcup D' = \bigcup F$. If $|D'| \leqslant |\bigcup F|/3$, then $|D'| = |\bigcup F|/3$ and the sets in D' must be pairwise disjoint. Thus, mapping F to $(G(F), |\bigcup F|/3)$ is a reduction as desired. □

You have a knapsack allowing you to carry things up to some given weight and you want to load it with items summing up to at least some given value.

Example 2.4.9 *The following are* NP*-complete.*

Knapsack	
Input:	sequences $(v_1, \ldots, v_s), (w_1, \ldots, w_s) \in \mathbb{N}^s$ for some $s \in \mathbb{N}$, and $V, W \in \mathbb{N}$.
Problem:	is there $S \subseteq [s]$ with $\sum_{i \in S} v_i \geqslant V$ and $\sum_{i \in S} w_i \leqslant W$?

Subset Sum	
Input:	a sequence $(m_1, \ldots, m_s) \in \mathbb{N}^s$ for some s, and $G \in \mathbb{N}$.
Problem:	is there $S \subseteq [s]$ with $\sum_{i \in S} m_i = G$?

Proof: It is easy to see that Knapsack $\in$ NP. It suffices to show that Subset Sum, a "special case" of Knapsack, is NP-hard. We reduce from 3-Set Cover.

Let an instance F of 3-Set Cover be given, and assume $\bigcup F = [n]$ for some $n \in \mathbb{N}$. View a subset X of $[n]$ as an n bit string (the ith bit being 1 if $i \in X$ and 0 otherwise); think of this n bit string as the $(|F|+1)$-adic representation of $m_X := \sum_{i \in X} 1 \cdot (|F|+1)^{i-1}$. Observe that if $\ell \leqslant |F|$ and $X_1, \ldots, X_\ell \subseteq [n]$ then

$$\begin{aligned} X_1, \ldots, X_\ell \text{ are pairwise disjoint} &\iff m_{X_1 \cup \cdots \cup X_\ell} = m_{X_1} + \cdots + m_{X_\ell} \\ &\iff \exists Z \subseteq [n] : m_Z = m_{X_1} + \cdots + m_{X_\ell}. \end{aligned}$$

This observation follows noting that when adding up the m_{X_j}s then no carries can occur (due to $\ell \leqslant |F|$). The reduction maps F to the following instance: as sequence take $(m_X)_{X \in F}$ and as "goal" G take $m_{[n]}$ which has $(|F|+1)$-adic representation 1^n. To see this can be computed in polynomial time, note $|bin(G)| \leqslant O(\log((|F|+1)^{n+1})) \leqslant O(n \cdot \log |F|)$. □

Proposition 2.4.10 *There exists a Turing machine that decides* KNAPSACK *and runs in time polynomial in $W \cdot n$ on instances of size n with weight bound W.*

Proof: On an instance $((v_1, \dots, v_s), (w_1, \dots, w_s), W, V)$, it suffices to compute $val(w, i)$ for each $w \leqslant W$ and $i \leqslant s$: the maximum v such that there exists $S \subseteq [i]$ with $\sum_{i \in S} v_i = v$ and $\sum_{i \in S} w_i \leqslant w$. This can easily be done iteratively: $val(w, 0) = 0$ for all $w \leqslant W$, and $val(w, i+1)$ is the maximum of $val(w, i)$ and $v_{i+1} + val(w - w_{i+1}, i)$. □

Remark 2.4.11 Note that the previous algorithm is not polynomial time because W is exponential in the length of its encoding $bin(W)$. Assuming $\mathsf{P} \neq \mathsf{NP}$, the last two propositions imply that KNAPSACK is *not* NP-hard when W is encoded in unary. Under the same assumption, it also follows that every polynomial time reduction from some NP-hard Q to KNAPSACK produces instances with superpolynomially large W.

Such explosions of "parameters" block the interpretation of an NP-completeness result as a non-tractability result if one is mainly interested in instances with small "parameter". *Parameterized Complexity Theory* is a more fine-grained complexity analysis that takes this into account. This is an important topic but outside the scope of this lecture.

We saw 13 examples of NP-complete problems and end the list here. It has been Karp in 1972 who famously proved the NP-completeness of 21 problems. After that a flood of NP-completeness results have been proven by countless researchers. The flood petered out and culminated in Garey and Johnsson's monograph listing hundreds of such problems.

For all NP-complete problems the existence of a polynomial time algorithm is equivalent to $\mathsf{P} = \mathsf{NP}$. Many of them are important in computational practice, so programmers all over the world are trying to find as fast as possible algorithms for them every day. This is somewhat of "empirical" evidence for the hypothesis that $\mathsf{P} \neq \mathsf{NP}$.

2.5 NP-completeness – theory

We now take a more abstract view on NP-completeness. Definition 2.5.1 distinguishes 'thin' and 'fat' problems. All our 13 NP-complete problems are 'fat'. We shall show:

1. Note that finitely many errors can be corrected: $Q \in \mathsf{P}$ if there is a polynomial time algorithm that decides Q except for finitely many inputs. We show that for 'fat' problems even polynomially many errors can be corrected. Hence, 'fat' NP-hard problems are 'far from' being in P: polynomial time algorithms make superpolynomially many errors unless $\mathsf{P} = \mathsf{NP}$ (Theorem 2.5.4).
2. 'Fat' NP-complete problems are pairwise *p-isomorphic* (Theorem 2.5.6). Intuitively, this means they are the same up to an effective re-writing of strings. So, despite their apparent wealth, we know only one NP-problem up to p-isomorphism.
3. 'Thin' NP-hard problems do not exist unless $\mathsf{P} = \mathsf{NP}$ (Theorem 2.5.8).

4. The maybe most interesting insight about NP-completeness in this section is that, unless $\mathsf{P} = \mathsf{NP}$, there exist problems of 'intermediate' complexity: problems in NP that are neither in P nor NP-complete (Theorem 2.5.10).

Definition 2.5.1 A problem Q is *sparse* if $|Q \cap \{0,1\}^n|$ is polynomially bounded in n. A problem Q is *paddable* if there is a polynomial time computable injection $pad : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ such that for all $x, y \in \{0,1\}^*$

(a) $pad(x,y) \in Q \iff x \in Q$;

(b) $|pad(x,y)| \geqslant |x| + |y|$;

(c) the partial functions $pad(x,y) \mapsto y$ and $pad(x,y) \mapsto x$ are polynomial time computable.

Here, a partial function f from binary strings to binary strings is said to be polynomial time computable if there is a polynomial time bounded Turing machine that rejects inputs x on which f is not defined and outputs $f(x)$ otherwise.

Exercise 2.5.2 Paddable problems are empty or infinite. Non-empty paddable problems are not sparse. Every problem Q is polynomial time equivalent to a paddable one.

Examples 2.5.3 Every problem from Section 2.4 is paddable. E.g., for SAT map (α, y) to $(\alpha \wedge (1 \vee \bigwedge y_i))$. For CLIQUE map, say, $((G,k), 0110)$ with $k > 3$ to (G', k) where G' results from $G = (V, E)$ by first disjointly adding a path of length $4 + |V|$ (where $4 = |0110|$), and then adding a vertex with edges to the 2nd and 3rd vertex in the path.

2.5.1 Schöningh's theorem

For a time-bounded algorithm $\mathbb{A}$ and a problem Q, the *error of* $\mathbb{A}$ *on* Q is the function from $\mathbb{N}$ to $\mathbb{N}$ given by

$$n \mapsto |\{x \in \{0,1\}^{\leqslant n} \mid \mathbb{A}(x) \neq \chi_Q(x)\}|.$$

Here, $\mathbb{A}(x)$ denotes the output of $\mathbb{A}$ on x, and χ_Q the characteristic function of Q: it maps $x \in \{0,1\}^*$ to 1 or 0 depending on whether $x \in Q$ or not.

Theorem 2.5.4 (Schöningh) *A paddable problem Q is in* P *if and only if there exists a polynomial time bounded (deterministic) Turing machine with polynomially bounded error on Q.*

Proof: Assume Q is paddable and $\mathbb{A}$ is polynomially time bounded with error on Q bounded by n^c for $n \geqslant 2$ and some constant $c \in \mathbb{N}$. We show that $Q \in \mathsf{P}$.

Let $y_1, y_2, \ldots$ enumerate all strings in lexicographic order. We define an algorithm $\mathbb{B}$: on $x \in \{0,1\}^*$ and $t \in \mathbb{N}$ it outputs the majority value of

$$\mathbb{A}(pad(x, y_1)), \ldots, \mathbb{A}(pad(x, y_t)).$$

Here, *pad* witnesses that Q is paddable. Let $d \in \mathbb{N}$ be a constant such that every $pad(x, y_i)$ has length at most $(|x| + |y_t|)^d$. Among the inputs of at most this length there are at most $(|x| + |y_t|)^{d \cdot c}$ many where $\mathbb{A}$ has output different from χ_Q. Thus $\mathbb{B}$ outputs $\chi_Q(x)$ if

$$t > 2 \cdot (|x| + |y_t|)^{d \cdot c}.$$

This holds true for $t := 2 \cdot (2|x|)^{d \cdot c}$ and long enough x: note $t > \sum_{\ell < |y_t|} 2^\ell = 2^{|y_t|}$, so $|y_t| \leqslant \log t$. Running $\mathbb{B}$ on x and this t needs time polynomial in $|x|$. Hence $Q \in \mathsf{P}$. □

2.5.2 Berman and Hartmanis' theorem

Definition 2.5.5 An injection $f : \{0,1\}^* \to \{0,1\}^*$ is *p-invertible* if the partial function f^{-1} is polynomial time computable. Two problems Q, Q' are *p-isomorphic* if there is a polynomial time reduction from Q to Q' (or, equivalently, vice-versa) which is bijective and p-invertible.

Note a reduction witnessing that Q and Q' are p-isomorphic is an isomorphism from the structure $(\{0,1\}^*, Q)$ onto the structure $(\{0,1\}^*, Q')$.

Theorem 2.5.6 (Berman, Hartmanis 1977) *Two paddable problems are polynomial time equivalent if and only if they are p-isomorphic.*

Our proof mimicks a particularly elegant proof of the following historical set-theoretic

Proposition 2.5.7 (Schröder-Bernstein) *Let X, Y be sets and assume there are injections f from X into Y and g from Y into X. Then there is a bijection from X onto Y.*

Proof: For x in X define the *preimage sequence*

$$g^{-1}(x),\ f^{-1}(g^{-1}(x)),\ g^{-1}(f^{-1}(g^{-1}(x))), \ldots$$

as long as it is defined. E.g., this is the empty sequence if x is not in the image of g. Then

$$h(x) := \begin{cases} g^{-1}(x) & \text{if the preimage sequence of } x \text{ has odd length,} \\ f(x) & \text{if the preimage sequence of } x \text{ has even or infinite length.} \end{cases}$$

defines a bijection from X onto Y. We only verify verify surjectivity: given $y \in Y$, set $x := g(y)$. Then the preimage sequence of x is not empty. If its length is odd, then $h(x) = g^{-1}(x) = y$. If it is even or infinite, then it has length at least 2, so $x' := f^{-1}(y)$ exists and the length of its preimage sequence is also even or infinite; hence $h(x') = f(x') = y$. □

Proof of Theorem 2.5.6: Let Q, Q' be paddable, and let $f : Q \leqslant_p Q'$ and $g : Q' \leqslant_p Q$. Without loss of generality, f, g are p-invertible and length-increasing (i.e. they map each string to some longer string). Indeed, if, say, f would not have these properties, replace it by $x \mapsto pad(f(x), 1x)$ where *pad* witnesses paddability of Q'.

But then the lengths of the strings in a pre-image sequence decrease. By p-invertibility, pre-image sequences are computable in polynomial time. This implies that h, defined as in the previous proposition, is polynomial time computable. It is clearly a reduction. That h^{-1} is polynomial time computable follows from the proof of surjectivity of h. □

The *Berman-Hartmanis Conjecture* states that in fact all NP-complete problems are pairwise p-isomorphic. This is open but many researchers tend to believe that it fails.

2.5.3 Mahaney's theorem

Theorem 2.5.8 (Mahaney 1980) *If* $\mathsf{P} \neq \mathsf{NP}$*, then sparse* NP*-hard problems do not exist.*

Proof: Let $<_{lex}$ be the lexicographic order on strings. For a string $y = y_1 \cdots y_m$ and a propositional formula $\alpha = \alpha(X_1, \ldots, X_n)$ let $y \vDash \alpha$ mean that $n \leqslant m$ and α is satisfied by the assignment $X_i \mapsto y_i$. The following problem is NP-complete: it is obviously in NP and it is NP-hard, since $\alpha \mapsto (\alpha, 1^{|\alpha|})$ is a (polynomial time) reduction from SAT.

LEFT SAT	
Input:	a propositional formula α and $x \in \{0,1\}^*$.
Problem:	does there exist $y \leqslant_{lex} x$ such that $y \vDash \alpha$?

Assume there is a sparse NP-hard problem Q, say $|Q \cap \{0,1\}^{\leqslant \ell}| \leqslant \ell^c$ for some $c \in \mathbb{N}$. Let r: LEFT SAT $\leqslant_p Q$. We give a polynomial time algorithm solving SAT.

Given $\alpha = \alpha(X_1, \ldots, X_n)$, the algorithm computes for every $1 \leqslant i \leqslant n$ a sequence

$$x_1^i <_{lex} \cdots <_{lex} x_{s_i}^i$$

of strings in $\{0,1\}^i$ such that $s_i \leqslant N := \left(\max_{x \in \{0,1\}^n} |r(\langle \alpha, x \rangle)|\right)^c + 2$, and

if there exists $z \in \{0,1\}^n$ such that $z \vDash \alpha$, then there is $j \in [s_i]$ such that x_j^i is an initial segment of the $<_{lex}$-minimal such z. $(*)$

Once the sequence for $i = n$ has been computed, the algorithm simply checks whether an element of it satisfies α. For $i = 1$ take $0 <_{lex} 1$ (note $2 \leqslant N$). Assume we computed the sequence for $i < n$. We explain how to compute the sequence for $i + 1$.

Write $y_{2j-1} := x_j^i 0$ and $y_{2j} := x_j^i 1$ and note $y_1 <_{lex} \cdots <_{lex} y_{2s_i}$. If $2s_i \leqslant N$, we are done. Otherwise we delete some of the y_js such that $(*)$ is preserved, namely each y_j such that $r(\langle \alpha, y_{j'} 1^{n-i-1} \rangle) = r(\langle \alpha, y_j 1^{n-i-1} \rangle)$ for some $j' < j$. This gives a subsequence $z_1 <_{lex} \cdots <_{lex} z_{s'}$ with $(*)$ and pairwise distinct values $r(\langle \alpha, z_j 1^{n-i-1} \rangle), j \in [s']$. If $s' \leqslant N$, we are done. Otherwise we delete all but the last N many z_js. This preserves $(*)$: if a deleted z_j would be initial segment of some $x \vDash \alpha$, then $x \leqslant_{lex} z_j 1^{n-i-1}$, so $r(\langle \alpha, z_j 1^{n-i-1} \rangle) \in Q$ and all later values $r(\langle \alpha, z_{j'} 1^{n-i-1} \rangle), j' \geqslant j$, would be in Q too – but there are at most N of them. □

Remark 2.5.9 Under a stronger assumption, Burhmann and Hitchcock showed in 2008 that even problems Q with $|Q \cap \{0,1\}^n| \leqslant 2^{n^{o(1)}}$ cannot be NP-hard. The assumption $\Sigma_3^{\mathsf{P}} \neq \mathsf{PH}$, that we shall introduce later, is sufficient for this.

2.5.4 Ladner's theorem

Theorem 2.5.10 (Ladner 1975) *Assume* $\mathsf{NP} \neq \mathsf{P}$. *Then* NP *contains problems which are neither* NP*-complete nor in* P.

It suffices to apply the following proposition to an NP-complete Q. Indeed, if $\mathsf{NP} \neq \mathsf{P}$, then $\mathsf{NP} \setminus \mathsf{P}$ contains an infinite descending $\leqslant_p$-chain of pairwise non-equivalent problems:

Proposition 2.5.11 *If* Q *is a decidable problem outside* P, *then there exists a problem* Q' *such that* $Q' \leqslant_p Q$ *and* $Q' \not\equiv_p Q$ *and* $Q' \notin \mathsf{P}$.

Proof: The proof is again by "slow diagonalization" (cf. Theorem 2.2.10). We define

$$Q' := \{x \in Q \mid D(|x|) \text{ is even}\}$$

for a certain "diagonalizing" function $D : \mathbb{N} \to \mathbb{N}$. The value $D(n)$ will be computable in time polynomial in n, so $Q' \leqslant_p Q$ follows.

Let $\mathbb{Q}$ decide Q. Let $(\mathbb{A}_0, c_0), (\mathbb{A}_1, c_1), \ldots$ enumerate all pairs $(\mathbb{A}, c)$ of (deterministic) single-tape Turing machines $\mathbb{A}$ and $c \in \mathbb{N}$ such that the ith pair can be computed in time $O(i)$. We set $D(0) := 0$ and compute $D(n)$ for $n > 0$ as follows: compute $D(0) = 0$ and then recursively the values $D(1), D(2), \ldots$ as many as you can within $\leqslant n$ steps. Let k be the last value computed, and let $(\mathbb{A}, c)$ be the $\lfloor k/2 \rfloor$th pair. What happens next depends on the parity of k. In any case the algorithm outputs k or $k + 1$.

If k is even, the algorithm will output $k + 1$ only if $\mathbb{A}$ is not a $(n^c + c)$-time bounded machine deciding Q'. If k is odd, the algorithm will output $k+1$ only if $\mathbb{A}$ is not a (n^c+c)-time bounded reduction from Q to Q'.

To do so, the algorithm runs Witness Test on $(\mathbb{A}, c, x, k)$ successively for all strings x in lexicographic order. It does so for at most n steps in total, and it outputs $k + 1$ in case Witness Test accepted at least once; otherwise it outputs k. In the pseudo-code $\mathbb{A}(x)$ denotes the output of $\mathbb{A}$ on x; note $|x|, |y| < n$, so D is already defined.

```
Witness Test on input $(\mathbb{A}, c, x, k)$

1. simulate $\mathbb{A}$ on $x$ for at most $|x|^c + c$ steps
2. **if** the simulation does not halt **then** accept
3. **if** $k$ is even **then** $m \leftarrow D(|x|)$
4.    **if** $\mathbb{A}$ rejects $x$ and $\mathbb{Q}$ accepts $x$ and $m$ is even **then** accept
5.    **if** $\mathbb{A}$ accepts $x$ and $\big(\mathbb{Q}$ rejects $x$ or $m$ is odd$\big)$ **then** accept
6. **if** $k$ is odd **then** $y \leftarrow \mathbb{A}(x)$; $m \leftarrow D(|y|)$
7.    **if** $\mathbb{Q}$ rejects $x$ and $\mathbb{Q}$ accepts $y$ and $m$ is even **then** accept
8.    **if** $\mathbb{Q}$ accepts $x$ and $\big(\mathbb{Q}$ rejects $y$ or $m$ is odd$\big)$ **then** accept
9. reject
```

Obviously, D assumes a value $k+1$ only if it assumes value k, so its range is an initial segment of $\mathbb{N}$. The reader easily verifies that D is nondecreasing. We are thus left to show that D is unbounded.

Assume otherwise, so there are $k \in \mathbb{N}$ such that $D(n) = k$ for all large enough n. Write $\mathbb{A} := \mathbb{A}_{\lfloor k/2 \rfloor}$ and $c := c_{\lfloor k/2 \rfloor}$. Then Witness Test rejects $(\mathbb{A}, c, x, k)$ for every $x \in \{0,1\}^*$. Indeed, if Witness Test accepts $(\mathbb{A}, c, x, k)$, its run is simulated by our algorithm for large enough n, and then our algorithm would output $k+1$. It follows that $\mathbb{A}$ halts in time $(n^c + c)$: otherwise Witness Test would accept $(\mathbb{A}, c, x, k)$ for suitable x in line 2. We arrive at the desired contradiction by showing $Q \in \mathsf{P}$. We have two cases.

If k is even, then Q and Q' agree on sufficiently long inputs. Hence, it suffices to show $Q' \in \mathsf{P}$. But this is true because $\mathbb{A}$ decides Q': otherwise Witness Test would accept $(\mathbb{A}, c, x, k)$ for suitable x in line 4 or 5.

If k is odd, then Q' is finite, so it suffices to show $Q \leqslant_p Q'$. But this is true because $\mathbb{A}$ computes a reduction from Q to Q': otherwise Witness Test would accept $(\mathbb{A}, c, x, k)$ for suitable x in line 7 or 8. □

2.6 Sat solvers

2.6.1 Self-reduciblity

We present the "search-to-decision reduction" for Sat: from an algorithm deciding Sat we get an algorithm that solves the search version of Sat, namely to compute satisfying assignments on satisfiable inputs. A similar reduction exists for all NP-complete problems.

Definition 2.6.1 A Sat *solver* is an algorithm that given a satisfiable formula α computes a satisfying assignment of α.

Note a Sat solver is allowed to do whatever it wants on inputs $x \notin$ Sat.

Theorem 2.6.2 Sat $\in \mathsf{P}$ *if and only if there exists a polynomial time bounded* Sat *solver.*

Proof: If p is a polynomial and $\mathbb{A}$ a p-time bounded Sat solver, then Sat $\in \mathsf{P}$: given α, run $\mathbb{A}$ on α for $p(|\alpha|)$ steps and check it outputs a satisfying assignment of α.

The converse exploits the so-called "self-reducibility" of Sat. Let $\mathbb{B}$ decide Sat in polynomial time. The algorithm $\mathbb{A}$ on an input formula α runs the following algorithm $\mathbb{A}^*$ on α together with the empty partial assignment. The algorithm $\mathbb{A}^*$ given a Boolean formula $\alpha(X_1, \ldots, X_n)$ and a partial assignment A proceeds as follows.

```
1. if $n = 0$ then output $A$
2. if $\mathbb{B}$ accepts $\alpha(X_1, \ldots, X_{n-1}, 0)$ then
3.    run $\mathbb{A}^*$ on $(\alpha(X_1, \ldots, X_{n-1}, 0), A \cup \{(X_n, 0)\})$
4. run $\mathbb{A}^*$ on $(\alpha(X_1, \ldots, X_{n-1}, 1), A \cup \{(X_n, 1)\})$
```

It is easy to see that $\mathbb{A}$ has the claimed properties. □

Exercise 2.6.3 Recall the discussion at the end of Section 1.1.2. Observe that a graph G has an independent set of size $k > 0$ which contains vertex v if and only if G' has an independent set of size $k-1$ where G' is obtained from G by deleting v and all its neighbors from G. Use this to prove an analogue of the above theorem for IS instead of SAT.

Exercise 2.6.4 Assume $\mathsf{P} = \mathsf{NP}$. Show that for every polynomially bounded $R \subseteq (\{0,1\}^*)^2$ in P with NP-complete $Q := \mathrm{dom}(R)$ there is a polynomial time computable partial function $f \subseteq R$ with $\mathrm{dom}(f) = Q$. *Hint:* Recall the proof of the Cook-Levin Theorem.

2.6.2 Levin-optimal SAT solvers

For a (deterministic) algorithm $\mathbb{A}$ we let $t_{\mathbb{A}}(x)$ be the length of the run of $\mathbb{A}$ on x; the function $t_{\mathbb{A}}$ takes values in $\mathbb{N} \cup \{\infty\}$.

Theorem 2.6.5 (Levin 1973) *There exists a* SAT *solver* $\mathbb{O}$ *which is* Levin-optimal*: for every* SAT *solver* $\mathbb{A}$ *there exists a polynomial* $p_{\mathbb{A}}$ *such that for all* $\alpha \in$ SAT*:*

$$t_{\mathbb{O}}(\alpha) \leqslant p_{\mathbb{A}}(t_{\mathbb{A}}(\alpha) + |\alpha|).$$

Proof: Let $R \subseteq (\{0,1\}^*)^2$ in P contain the pairs (x, y) such that x codes a Boolean formula and y codes a satisfying assignment of it. Let $\mathbb{A}_0, \mathbb{A}_1, \ldots$ be an enumeration of all algorithms such that $\mathbb{A}_i$ can be computed in time $O(i)$. We define $\mathbb{O}$ on input x as follows.

```
1. $\ell \leftarrow 0$
2. for all $i \leqslant \ell$ do
3.    simulate $\ell$ steps of $\mathbb{A}_i$ on $x$
4.    if the simulated run halts and outputs $y$ such that $(x, y) \in R$
5.       then halt and output $y$
6. $\ell \leftarrow \ell + 1$
7. goto 2
```

It is easy to check that $\mathbb{O}$ is a SAT solver. The time needed in lines 2-4 can be bounded by $(\ell + |x| + 2)^d$ for suitable $d \in \mathbb{N}$. Let $\mathbb{A}$ be a SAT solver and $\alpha \in dom(R) =$ SAT. Choose $i_{\mathbb{A}} \in \mathbb{N}$ such that $\mathbb{A} = \mathbb{A}_{i_{\mathbb{A}}}$. Then $\mathbb{O}$ on α halts in line 4 if $\ell \geqslant t_{\mathbb{A}}(\alpha), i_{\mathbb{A}}$ – or earlier. Hence,

$$t_{\mathbb{O}}(\alpha) \leqslant O(\textstyle\sum_{\ell=0}^{t_{\mathbb{A}}(\alpha)+i_{\mathbb{A}}}(\ell + |\alpha| + 2)^d) \leqslant c_{\mathbb{A}} \cdot (t_{\mathbb{A}}(\alpha) + |\alpha|)^d$$

for a suitable constant $c_{\mathbb{A}} \in \mathbb{N}$. □

Remark 2.6.6 The proof shows that the degree of $p_{\mathbb{A}}$ does not depend on $\mathbb{A}$. It is open whether there is a similarly optimal decision algorithm for SAT.

Exercise 2.6.7 Make precise and prove: let R be a binary relation in P. There exists an "optimal" algorithm that given x finds y such that $(x, y) \in R$ provided such y exist.

Cook's theorem casts the P versus NP question as one about one particular problem. Levin's theorem allows to cast the question as one about one particular algorithm.

Corollary 2.6.8 *Let $\mathbb{O}$ be an optimal* Sat *solver. Then $\mathsf{P} = \mathsf{NP}$ if and only if there is a polynomial p such that $t_{\mathbb{O}}(\alpha) \leqslant p(|\alpha|)$ for every $\alpha \in$* Sat.

Proof: By Theorem 2.6.2, $\mathsf{P} = \mathsf{NP}$ implies there is a Sat solver $\mathbb{A}$ that is q-time bounded for some polynomial q. By Theorem 2.6.5, $t_{\mathbb{O}}(\alpha) \leqslant p_{\mathbb{A}}(q(|\alpha|) + |\alpha|))$ for $\alpha \in$ Sat.

The converse follows from Theorem 2.6.2 and the NP-completeness of Sat. □

2.7 History and significance of P versus NP

Usually, the P versus NP is dated 1971, the year when Cook published the NP-completeness of Sat. The likely first formal statement dates 1967 in lecture notes of Cook.

Recall from Section 1.2.1 that Gödel asked 1956 whether an NP-complete problem is decidable in quadratic time and states this is "quite within the realm of possibility". Is this the first formulation of the P versus NP problem and did he conjecture that $\mathsf{P} = \mathsf{NP}$? It is the author's understanding that he rather intended to stress that a refutation is unknown. Also, Gödel asks for quadratic time while polynomial time as a formalization of 'efficient computability' has been put forward only later in 1965 by Cobham and Edmonds. Remarkably, Gödel aks more generally: "It would be interesting to know [. . .] how significantly *in general* for finitist combinatorial problems the number of steps can be reduced when compared to mere trying." E.g., $\mathsf{NP} \subseteq \mathsf{TIME}(2^{\sqrt{n}})$ would clearly constitute a general and significant reduction of "mere trying". It seems fair to say that, within his context, Gödel addressed questions that later condensed as the P versus NP problem.

Following Gödel's letter, NP-completeness bears on the philosophy of mind (e.g., Turing's Test or Searle's Room) and, further, cognitive science (Marr's computational level of explanation). These are only examples, few scientific concepts spread as broadly through science. Papadimitriou said in 1995 that "about 6000 papers each year have the term 'NP-complete' on their title, abstract, or list of keywords." Why? Wigderson states that "NP actually captures far more than computational tasks. After appropriately formalizing the recognition process, all mathematical tasks, many engineering tasks, and central scientific tasks as well fall into NP." Going further, he claims NP is "a mathematical definition of all interesting problems" – why start looking for something if you don't know when you found it? Informally, then $\mathsf{P} = \mathsf{NP}$ states that all interesting problems are efficiently solvable.

The vast majority of experts believes that $\mathsf{P} \neq \mathsf{NP}$. As Impagliazzo puts it, "as soon as a feasible algorithm for an NP-complete problem is found, the capacity of computers will become that currently depicted in science fiction." Indeed, "almost any optimization problem would be easy and automatic [. . .] Programming languages would not need to specify *how* the computation should be performed. Instead, one would just specify the properties that a desired output should have in relation to the input.[...] One could [...] automatically train a computer to perform any task that humans can [...] computers could find proofs for any theorem in time roughly the length of the proof."

As Wigderson puts it, "while elusive to define, people feel that creativity, ingenuity or leap-of-thought which lead to discoveries are the domain of very singular, talented and well-trained individuals, and that the process leading to discovery is anything but the churning of a prespecified procedure or recipe. These few stand in sharp contrast to the multitudes who can appreciate the discoveries after they are made."

As Aaronson puts it, "if $\mathsf{P} = \mathsf{NP}$, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in 'creative leaps,' no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss".

Chapter 3

Space

3.1 Space bounded computation

Recall that a configuration of a deterministic or nondeterministic Turing machine $\mathbb{A}$ with input tape and k worktapes is a tuple $(s, i\bar{j}, c\bar{c})$ where s is the current state of the machine, i and $\bar{j} = j_1 \cdots j_k$ are the positions of the input head and the heads on the k worktapes, c and $\bar{c} = c_1 \cdots c_k$ are the contents of the input tape and the k worktapes.

Definition 3.1.1 Let $\mathbb{A}$ be a deterministic or nondeterministic Turing machine with input tape and k worktapes. Let $S \in \mathbb{N}$. A configuration $(s, i\bar{j}, c\bar{c})$ of $\mathbb{A}$ is *S-small* if $j_1, \ldots, j_k \leqslant S$ and on all worktapes all cells with number bigger than S are blank; it is *on* $x = x_1 \cdots x_n$ if $c(0)c(1)\cdots$ reads $\triangleright\, x_1 \cdots x_n \;\square\; \square\cdots$.

Let $s : \mathbb{N} \to \mathbb{N}$. We say $\mathbb{A}$ is *s-space bounded* if for every $x \in \{0,1\}^*$ all configurations in all computations of $\mathbb{A}$ on x are $s(|x|)$-small. The set $\mathsf{SPACE}(s)$ ($\mathsf{NSPACE}(s)$) contains the problems Q that are for some constant $c \in \mathbb{N}$ decided (accepted) by a $(c \cdot s + c)$-space bounded (nondeterministic) Turing machine. We define

$$\begin{array}{lcl} \mathsf{L} & := & \mathsf{SPACE}(\log), \\ \mathsf{NL} & := & \mathsf{NSPACE}(\log), \\ \mathsf{PSPACE} & := & \bigcup_{c \in \mathbb{N}} \mathsf{SPACE}(n^c), \\ \mathsf{NPSPACE} & := & \bigcup_{c \in \mathbb{N}} \mathsf{NSPACE}(n^c). \end{array}$$

Space complexity measures memory requirements to solve problems. While sublinear time restrictions are boring in the sense that they prohibit to read the input, this is not the case for space restrictions because they only apply to the work tapes. In fact, the most interesting classes are L and NL above. We are however only interested in space bounds that are at least log. Otherwise the space restriction does not even allow us to remember the fact that we have read a certain bit at a certain position in the input.

An S-small configuration of $\mathbb{A}$ on $x \in \{0,1\}^*$ can be specified giving the positions $i, \bar{j}$ plus the contents of the worktapes up to cell S. It thus can be coded by a string of length

$$O(\log |\ulcorner \mathbb{A} \urcorner| + \log |x| + k \log S + kS).$$

Definition 3.1.2 Assume $\mathbb{A}$ is s-space bounded for some $s \geqslant \log$ and $x \in \{0,1\}^*$. Then $s(|x|)$-small configurations of $\mathbb{A}$ on x can be coded as explained above by a string of length *exactly* $e \cdot s(|x|)$ for $e \in \mathbb{N}$ some suitable constant. The *configuration graph of* $\mathbb{A}$ *on* x is the directed graph

$$G_x^{\mathbb{A}} := \left(\{0,1\}^{e \cdot s(|x|)}, E_x^{\mathbb{A}}\right)$$

where $E_x^{\mathbb{A}}$ contains (c,d) if c codes a non-halting configuration and d a successor of it.

The next proposition concerns how time and space relate as complexity measures.

Proposition 3.1.3 *Let $s : \mathbb{N} \to \mathbb{N}$ be space-constructible. Then*

$$\mathsf{NTIME}(s) \subseteq \mathsf{SPACE}(s) \subseteq \mathsf{NSPACE}(s) \subseteq \textstyle\bigcup_{c \in \mathbb{N}} \mathsf{TIME}(2^{c \cdot s}).$$

Proof: Let Q in $\mathsf{NTIME}(s)$, say $\mathbb{A}$ is a nondeterministic $(c \cdot s + c)$-time bounded Turing machine that accepts Q where $c \in \mathbb{N}$ is a suitable constant. Every run of $\mathbb{A}$ on an input x is determined by a binary string of length $c \cdot s(|x|) + c$. To see $Q \in \mathsf{SPACE}(s)$ consider the machine that on x goes lexicographically through all strings y of length $c \cdot s(|x|) + c$ and simulates the run of $\mathbb{A}$ on x determined by y. It accepts if one of the simulations is accepting. The space needed is the space to store the current y plus the space needed by a run of $\mathbb{A}$ on x. But this latter space is $O(s(|x|))$ because $\mathbb{A}$ is $O(s)$ time-bounded.

The second inclusion is trivial. For the third, suppose Q is accepted by an $O(s)$-space bounded nondeterministic Turing machine $\mathbb{A}$. Given x, compute $bin(s(|x|))$ running the machine witnessing space constructibility. This machine stops in time $2^{O(s(|x|))}$: otherwise the machine would enter some configuration twice and thus never halt. Given $bin(s(|x|))$ it is easy to compute the graph $G_x^{\mathbb{A}}$ in time $2^{O(s(|x|))}$. Then apply a polynomial time algorithm for REACHABILITY to check whether there exists a path from the starting configuration of $\mathbb{A}$ on x to some accepting configuration. This needs time (polynomial in) $2^{O(s(|x|))}$. □

Corollary 3.1.4 $\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}$.

It is commonly conjectured that all inclusions are strict. We know $\mathsf{P} \neq \mathsf{EXP}$; in Section 3.1.2 we shall see that $\mathsf{NL} \neq \mathsf{PSPACE} = \mathsf{NPSPACE}$; no other inequalities are known.

3.1.1 Space hierarchy

We next prove a hierarchy theorem for space as we did for time. The proof is similar but gives a tight result.

Definition 3.1.5 A function $s : \mathbb{N} \to \mathbb{N}$ is *space-constructible* if $s(n) \geqslant \log n$ for all $n \in \mathbb{N}$ and the function $x \mapsto bin(s(|x|))$ is computable by a $O(s)$-space bounded Turing machine.

Theorem 3.1.6 (Space Hierarchy) *Let s be space-constructible and $s' \leqslant o(s)$ arbitrary. Then* $\mathsf{SPACE}(s) \setminus \mathsf{SPACE}(s') \neq \emptyset$.

Proof: Recall $\ulcorner\mathbb{A}\urcorner$ denotes the binary encoding of a Turing machine $\mathbb{A}$ and consider

P_s	
Input:	a Turing machine $\mathbb{A}$ with input tape.
Problem:	for k the number of worktapes of $\mathbb{A}$, is it true that $\mathbb{A}$ on $\ulcorner\mathbb{A}\urcorner$ visits cell $\lfloor s(\lvert\ulcorner\mathbb{A}\urcorner\rvert)/k\rfloor$ on some of its worktapes or does not accept?

To decide P_s one simulates $\mathbb{A}$ on $\ulcorner\mathbb{A}\urcorner$ but stops rejecting once the simulation wants to visit cell $\lfloor s(\lvert\ulcorner\mathbb{A}\urcorner\rvert)/k\rfloor$. The simulation computes the successor configuration from the current configuration and then deletes the current configuration. This way only two configurations need to be stored. Each configuration can be written using

$$O(\log\lvert\ulcorner\mathbb{A}\urcorner\rvert + \log\lvert\ulcorner\mathbb{A}\urcorner\rvert + k\cdot s(\lvert\ulcorner\mathbb{A}\urcorner\rvert)/k)$$

many bits. These are at most $c\cdot s(\lvert\ulcorner\mathbb{A}\urcorner\rvert)$ many where $c\in\mathbb{N}$ is a suitable constant. Thus the simulation can be aborted after $2^{c\cdot s(\lvert\ulcorner\mathbb{A}\urcorner\rvert)}$ many steps as then $\mathbb{A}$ must have entered a loop. To signal abortion we increase a binary counter for each simulated step. The counter needs space roughly $\log 2^{c\cdot s(\lvert\ulcorner\mathbb{A}\urcorner\rvert)} = c\cdot s(\lvert\ulcorner\mathbb{A}\urcorner\rvert)$. This shows that $P_s\in\mathsf{SPACE}(s)$.

Let $s'\leqslant o(s)$ and assume for the sake of contradiction that $P_s\in\mathsf{SPACE}(s')$. Choose an algorithm $\mathbb{B}$ that decides P_s and is $(c\cdot s'+c)$-space-bounded for some constant $c\in\mathbb{N}$. Let k be its number of worktapes. We have $c\cdot s'(n)+c<\lfloor s(n)/k\rfloor$ for all n bigger than some suitable $n_0\in\mathbb{N}$. By adding, if necessary, dummy states to $\mathbb{B}$ we can assume that $\lvert\ulcorner\mathbb{B}\urcorner\rvert>n_0$.

If $\mathbb{B}$ would not accept $\ulcorner\mathbb{B}\urcorner$, then $\ulcorner\mathbb{B}\urcorner\in P_s$ by definition, so $\mathbb{B}$ wouldn't decide P_s. Hence $\mathbb{B}$ accepts $\ulcorner\mathbb{B}\urcorner$. Hence $\ulcorner\mathbb{B}\urcorner\in P_s$. Hence $\mathbb{B}$ on $\ulcorner\mathbb{B}\urcorner$ visits cell $\lfloor s(\lvert\ulcorner\mathbb{B}\urcorner\rvert)/k\rfloor$. As $\lvert\ulcorner\mathbb{B}\urcorner\rvert>n_0$ we get $c\cdot s'(\lvert\ulcorner\mathbb{B}\urcorner\rvert)+c<\lfloor s(\lvert\ulcorner\mathbb{B}\urcorner\rvert)/k\rfloor$, contradicting the assumed space bound of $\mathbb{B}$. $\square$

3.1.2 Savitch's theorem

Theorem 3.1.7 (Savitch 1970) REACHABILITY $\in\mathsf{SPACE}(\log^2)$.

Proof: Let (V,E) be a directed graph and assume $V=[n]$ for some $n\in\mathbb{N}$. Call a triple (u,w,i) *good* if there is a path of length $\leqslant 2^i$ from u to w. Observe that (u,w,i) is good if and only if there is a $v\in V$ such that both $(u,v,i-1)$ and $(v,w,i-1)$ are good.

Algorithm $\mathbb{A}$ decides whether (u,w,i) is good. Clearly, $\mathbb{A}$ on $(u,w,\lceil\log n\rceil)$ decides whether there is a path from u to w. On inputs from $V\times V\times\{i\}$ the space required is $O(m_i\cdot\log n$ where m_i is the maximal number of triples stored. Clearly, $m_0=0$ and $m_{i+1}=m_i+1$. For $i\leqslant\lceil\log n\rceil)$ we have $m_i\leqslant\log n$. Hence, $\mathbb{A}$ is $O(\log^2 n)$ space bounded. $\square$

```
$\mathbb{A}$ on $(u, w, i)$ and $(V, E)$

1.  if $i = 0$ check whether $u = w$ or $(u, w) \in E$
2.  $v \leftarrow 1$
3.  while $v \leqslant n$
4.      write $(u, v, i-1)$ on tape
5.      if $\mathbb{A}$ accepts $(u, v, i-1)$
6.          replace $(u, v, i-1)$ by $(v, w, i-1)$
7.          if $\mathbb{A}$ accepts $(v, w, i-1)$, erase $(v, w, i-1)$ and accept
8.      erase last triple on tape
9.      $v \leftarrow v + 1$
10. reject
```

Corollary 3.1.8 *If s is space-constructible, then* $\mathsf{NSPACE}(s) \subseteq \mathsf{SPACE}(s^2)$.

Proof: Suppose Q is accepted by a $(c \cdot s + c)$-space bounded nondeterministic Turing machine $\mathbb{A}$. We can assume that $\mathbb{A}$ has at most one accepting configuration c_{acc} reachable from x. To decide whether $x \in Q$ it suffices to decide whether there is a path from c_{start} to c_{acc} in $G_x^{\mathbb{A}}$. But this graph is too large to be written down if one is to respect the desired space bound. Instead one runs Savitch's algorithm and whenever this algorithm wants to check $(c, d) \in E_x^{\mathbb{A}}$ (in line 1) one runs a logarithmic space subroutine deciding $E_x^{\mathbb{A}}$. □

The above proof showcases an important idea, that we are going to formalize in Section 3.3.1: a small space algorithm works with the huge object $G_x^{\mathbb{A}}$ given implicitly.

Corollary 3.1.8 together with the space hierarchy theorem yields:

Corollary 3.1.9 $\mathsf{NL} \subseteq \mathsf{SPACE}(\log^2) \neq \mathsf{PSPACE} = \mathsf{NPSPACE}$.

It is open whether $\mathsf{NL} \subseteq \mathsf{SPACE}(o(\log^2))$. It is also open whether REACHABILITY can be decided by a machine that runs simultaneously in space $O(\log^2)$ and polynomial time.

3.2 Polynomial space

Quantified Boolean logic is obtained from propositional logic by adding the syntactical rules declaring $\exists X\alpha$ and $\forall X\alpha$ to be quantified Boolean formulas whenever α is one and X is a propositional variable. The semantics are explained such that an assignment A satisfies $\exists X\alpha$ (resp. $\forall X\alpha$) if $A\frac{0}{X}$ or (resp. and) $A\frac{1}{X}$ satisfies α. Here, $A\frac{b}{X}$ maps X to b and agrees with A on all other variables. A *quantified Boolean sentence* is a quantified Boolean formula without free variables. Such a sentence is either *true* (satisfied by some (equivalently all) assignments) or *false* (otherwise).

Exercise 3.2.1 For every quantified Boolean formula α with r quantifiers there exists a (quantifier free) propositional formula β that is equivalent to α of length $|\beta| \leqslant 2^r \cdot |\alpha|$.

Exercise 3.2.2 Given a Boolean circuit $C(\bar{X}) = C(X_1, \dots, X_r)$ with one output gate and $q \in \{\exists, \forall\}$ one can compute in polynomial time a quantified Boolean formula $q\bar{Z}\ \alpha(\bar{X}, \bar{Z})$ with α quantifier free, such that for all $x \in \{0,1\}^r$

$$q\bar{Z}\ \alpha(x, \bar{Z}) \text{ is true } \iff C(x) = 1.$$

Exercise 3.2.3 There is a polynomial time algorithm that given a quantified Boolean formula computes an equivalent one in *prenex form*, that is, of the form

$$q_1 X_1 \cdots q_r X_r \beta$$

where the q_is are quantifiers $\exists$ or $\forall$ and β is quantifier free.

Theorem 3.2.4 QBF *is* PSPACE-*complete.*

QBF	
Input:	a quantified Boolean sentence α.
Problem:	is α true?

Proof: By Exercise 3.2.3 it is sufficient to design an algorithm that works as desired on inputs in prenex form $q_1 X_1 \cdots q_r X_r \beta(X_1, \dots, X_r)$.

If $r = 0$, the sentence is a Boolean combination of 0 and 1 and its truth value can be computed in polynomial time. If $r > 0$, run $\mathbb{A}$ on $q_2 X_2 \cdots q_r X_r \beta(0, X_2 \dots, X_r)$ and store the truth value $b \in \{0,1\}$ obtained. Then erase all other workspace used in this computation. Then run $\mathbb{A}$ on $q_2 X_2 \cdots q_r X_r \beta(1, X_2 \dots, X_r)$ and store the truth value $b' \in \{0,1\}$ obtained. Then again erase all other workspace used in this computation. Finally, accept in case $\max\{b, b'\} = 1$ and $q_1 = \exists$, or $\min\{b, b'\} = 1$ and $q_1 = \forall$; otherwise reject.

Let $s_{\mathbb{A}}(n, r)$ denote the space required by this algorithm on sentences of size at most n with at most r variables. Then $s_{\mathbb{A}}(n, r) \leqslant s_{\mathbb{A}}(n, r-1) + O(n)$ and thus $s_{\mathbb{A}}(n, r) \leqslant O(r \cdot n)$. This implies that $\mathbb{A}$ runs in polynomial space.

To see that QBF is PSPACE-hard let $Q \in$ PSPACE and choose a polynomial p and a p-space bounded Turing machine $\mathbb{A}$ that decides Q. Let $x \in \{0,1\}^*$ and consider the configuration graph $G_x^{\mathbb{A}} = (\{0,1\}^m, E_x^{\mathbb{A}})$ of $\mathbb{A}$ on x; here, $m \leqslant O(p(|x|))$. We assume there is at most one accepting configuration c_{acc} reachable from the start configuration c_{start} of $\mathbb{A}$ on x. By Lemma 1.3.9 we find a circuit $C(X_1, \dots, X_m, Y_1, \dots, Y_m)$ (depending on x) such that for all $c, d \in \{0,1\}^m$

$$C(c, d) = 1 \iff (c, d) \in E_x^{\mathbb{A}}.$$

We want to compute a formula $\sigma_i(\bar{X}, \bar{Y})$ such that for all $c, d \in \{0,1\}^m$

$$\sigma_i(c, d) \text{ is true } \iff \text{ there is a length } \leqslant 2^i \text{ path from } c \text{ to } d \text{ in } G_x^{\mathbb{A}}$$

Then $x \mapsto \sigma_m(c_{\text{start}}, c_{\text{acc}})$ is a reduction as desired. For $\sigma_0(\bar{X}, \bar{Y})$ we take $\sigma(\bar{X}, \bar{Y}) \vee \bar{X} = \bar{Y}$ where $\bar{X} = \bar{Y}$ stands for $\bigwedge_{i=1}^m (X_i \leftrightarrow Y_i)$, and $\sigma(\bar{X}, \bar{Y})$ expresses $C(\bar{X}, \bar{Y}) = 1$ according to Exercise 3.2.2. For $\sigma_{i+1}(\bar{X}, \bar{Y})$ we follow Savitch, an take a formula equivalent to

$$\exists \bar{Z}(\sigma_i(\bar{X}, \bar{Z}) \wedge \sigma_i(\bar{Z}, \bar{Y})).$$

We cannot use this formula because then σ_m would have length $\geqslant 2^m$. Instead we take

$$\sigma_{i+1}(\bar{X}, \bar{Y}) := \exists \bar{Z} \forall \bar{U} \forall \bar{V} \big((\bar{U}\bar{V} = \bar{X}\bar{Z} \vee \bar{U}\bar{V} = \bar{Z}\bar{Y}) \rightarrow \sigma_i(\bar{U}, \bar{V}) \big).$$

Note that then $|\sigma_{i+1}| \leqslant O(m) + |\sigma_i|$, so $|\sigma_m| \leqslant O(m^2 + |\sigma|)$. It follows that σ_m can be computed from x in polynomial time. □

3.3 Nondeterministic logarithmic space

The following improves Example 1.2.5.

Example 3.3.1 REACHABILITY $\in$ NL.

Proof: Given a directed graph (V, E), say with $V = [n]$, and $v, v' \in V$ then

```
1. $u \leftarrow v$
2. while $u \neq v'$
3.    guess $w \in [n]$
4.    if $(u, w) \in E$, then $u \leftarrow w$
5. accept
```

is a nondeterministic algorithm that runs in logarithmic space. □

We give a simple, but useful characterization of NL paralleling Proposition 2.2.3.

Definition 3.3.2 Consider a (deterministic) Turing machine $\mathbb{A}$ with an input tape and with a special work-tape called *guess tape*; the head of the guess tape is not allowed to move left. A computation of $\mathbb{A}$ is *on* (x, y) if it starts in the configuration with all heads on cell 0, x written on the input tape, y written on the guess tape, and all other cells on any tape blank. $\mathbb{A}$ *accepts (rejects)* (x, y) if there is an accepting (rejecting) computation of $\mathbb{A}$ on (x, y). If there is $c \in \mathbb{N}$ such that for all (x, y) the computation of $\mathbb{A}$ on (x, y) does not contain a configuration with head position $c \cdot \log|x| + c$ on some work tape except possibly the guess tape, then $\mathbb{A}$ is said to be a *logspace verifier*.

Proposition 3.3.3 *Let Q be a problem. The following are equivalent.*

1. $Q \in$ NL.

2. *There is a logspace verifier* for Q*, i.e., there is a logspace verifier* $\mathbb{A}$ *and a polynomial* p *such that for all* $x \in \{0,1\}^*$

$$x \in Q \iff \exists y \in \{0,1\}^{\leqslant p(|x|)} : \mathbb{A} \textit{ accepts } (x,y).$$

Proof: To see (1) implies (2) let $Q \in \mathsf{NL}$ and choose a nondeterministic logspace bounded machine $\mathbb{B}$ that accepts Q. We can assume that $\mathbb{B}$ is p-time bounded for some polynomial p. Every $y \in \{0,1\}^{p(|x|)}$ determines a run of $\mathbb{B}$ on x. A logspace verifier $\mathbb{A}$ on (x,y) simulates $\mathbb{B}$ on x applying the transition function corresponding to the bit read on the guess tape. It moves the head on the guess tape one cell to the right for every simulated step.

Conversely, let $\mathbb{A}$ accord (2). A machine witnessing $Q \in \mathsf{NL}$ simulates $\mathbb{A}$ without the guess tape: it guesses the symbol read whenever $\mathbb{A}$ moves right on the guess tape. □

Remark 3.3.4 In (2) one can assume $\mathbb{A}$ to move the head one cell to the right in every step; furthermore, one can replace $y \in \{0,1\}^{\leqslant p(|x|)}$ by $y \in \{0,1\}^{p(|x|)}$ or $y \in \{0,1\}^*$.

Example 3.3.5 The following is a logspace verifier for Reachability. Given an x encoding an instance $((V,E),v,v')$ of Reachability the logspace verifier $\mathbb{A}$ expects 'the proof' y to encode a path $\bar{u}$ from v to v'.

Say, y encodes the sequence $\bar{u} = u_0 \cdots u_{\ell-1}$ of vertices in V. Then $\mathbb{A}$ copies the first vertex u_0 to the work tape and checks $u_0 = v$; it then replaces the worktape content by (u_0,u_1) and checks $(u_0,u_1) \in E$; it replaces the worktape content by (u_1,u_2) and checks $(u_1,u_2) \in E$; and so on. Finally it checks $u_{\ell-1} = v'$. If any of the checks fails, $\mathbb{A}$ rejects. It is clear that this can be done moving the head on the guess tape (containing y) from left to right. It is easy to make $\mathbb{A}$ also reject every y that does not even encode a sequence of vertices $\bar{u}$.

Some words on the significance of NL versus NP: the latter contains problems whose "yes" instances have proofs that can be checked by a polynomial time verifier, e.g., a proof for $\alpha \in$ Sat is a satisfying assignment. Problems in NL have "yes" instances with proofs that can be *streamed* to a verifier with logarithmic memory. Is it possible to design a notion of 'proof' of satisfiability that can be similarly verified?

3.3.1 Implicit logarithmic space computability

Reingold proved that Reachability is in L when restricted to undirected graphs. But in general it is not known whether Reachability $\in \mathsf{L}$. We now show that the answer is no unless $\mathsf{L} = \mathsf{NL}$. We do so by showing that the problem is NL-complete under a suitable notion of reduction. The notion used sofar does not work: NL is not closed under $\leqslant_p$ unless it equals P. To come up with the right notion we face the problem that we want allow the reduction to output strings that are longer than $\log n$, so in logspace we are not even able to write down the string to be produced. The right notion is that logspace allows to answer all questions about this string. Recall the bit-graph of a function from Section 1.1.2.

Definition 3.3.6 $f : \{0,1\}^* \to \{0,1\}^*$ is *implicitly logspace computable* if it is polynomially bounded (i.e. $|f(x)| \leqslant |x|^{O(1)}$ for all $x \in \{0,1\}^*$) and $\textsc{BitGraph}(f) \in \mathsf{L}$. A problem Q is *logspace reducible* to another Q' if there is an implicitly logspace computable reduction from Q to Q'; we write $Q \leqslant_{log} Q'$ to indicate this. That $\mathcal{C} \subseteq P(\{0,1\}^*)$ is *closed under* $\leqslant_{log}$, that a problem is *hard* or *complete* for $\mathcal{C}$ under $\leqslant_{log}$ is explained analogously as for $\leqslant_p$.

Exercise 3.3.7 Every implicitly logspace computable function is polynomial time computable. Hence $\leqslant_p$-closed classes are $\leqslant_{log}$-closed.

Lemma 3.3.8 *If $Q \leqslant_{log} Q'$ and $Q' \leqslant_{log} Q''$, then $Q \leqslant_{log} Q''$.*

Proof: It suffices to show that $f \circ g$ is implicitly logspace computable if both f and g are. Clearly, $f \circ g$ is polynomially bounded since f and g are. To decide the bit graph of $f \circ g$, let $\mathbb{A}$ be an algorithm witnessing $\textsc{BitGraph}(f) \in \mathsf{L}$. Given an instance (x, i, b) of $\textsc{BitGraph}(f \circ g)$ simulate $\mathbb{A}$ on $g(x)$ without computing $g(x)$: maintain (the code of) a configuration of $\mathbb{A}$ on $g(x)$ plus the current input symbol scanned: this requires space $O(\log |g(x)|) \leqslant O(\log |x|)$. From this it is easy to compute the successor configuration of $\mathbb{A}$ on $g(x)$. If $\mathbb{A}$ scans, say, cell j of the input tape, determine the symbol scanned by running a logspace algorithm for $\textsc{BitGraph}(g)$ on $(x, j, 0)$ and $(x, j, 1)$. □

Exercise 3.3.9 $Q \in \mathsf{L}$ if and only if $Q \leqslant_{log} \{1\}$; hence, L is $\leqslant_{log}$-closed. Also NL is $\leqslant_{log}$-closed.

Theorem 3.3.10 $\textsc{Reachability}$ *is* NL*-complete under* $\leqslant_{log}$.

Proof: By Example 3.3.1 it suffices to show that $\textsc{Reachability}$ is NL-hard. Let $Q \in \mathsf{NL}$ and a logarithmic space bounded nondeterministic machine $\mathbb{A}$ accepting Q. Again we assume that there is at most one accepting configuration c_{acc} reachable from the start configuration c_{start} of $\mathbb{A}$ on x. It is easy to see that $G_x^{\mathbb{A}}$ is implicitly logspace computable from x. The reduction maps x to the instance $(G_x^{\mathbb{A}}, c_{\text{start}}, c_{\text{acc}})$ of $\textsc{Reachability}$. □

3.3.2 Immerman and Szelepcsényi's theorem

Theorem 3.3.11 (Immerman, Szelepcsényi 1987) $\textsc{NonReachability} \in \mathsf{NL}$.

$\textsc{NonReachability}$	
Input:	a directed graph (V, E) and $v, v' \in V$.
Problem:	there is *no* path from v to v' in (V, E)?

Proof: Let $((V, E), v, v')$ be given and assume $V = [n]$ with $n > 1$. For $i < n$ let

$$R_i := \{u \in V \mid \text{there is a length} \leqslant i \text{ path from } v \text{ to } u\}.$$

We define a logspace verifier $\mathbb{A}$ for $\textsc{NonReachability}$. It expects a 'proof' y on the guess tape of the form $y = (\rho_1 \ldots, \rho_{n-1})$ and proceeds in such a way that after reading ρ_i it stores the number $|R_i|$ or rejects. In the beginning it stores $|R_0| = 1$. Details follow.

For $i < n-1$, the string ρ_{i+1} is an n-tuple

$$\rho_{i+1} = (\pi_1, \dots, \pi_n)$$

where every $\pi_u, u \in V = [n]$, has the form

$$\pi_u := (\bar{u}_1 v_1, \dots, \bar{u}_s v_s)$$

where the $\bar{u}_j v_j$ are sequences in V such that the following conditions hold:

(a) every $\bar{u}_j v_j$ is a length $\leqslant i$ path from v (to v_j);

(b) $s = |R_i|$;

(c) $v_1 < \cdots < v_s$.

Reading $\pi_u = (\bar{u}_1 v_1, \dots, \bar{u}_s v_s)$ from left to right, $\mathbb{A}$ checks these conditions in logspace: for (a) see Example 3.3.5, for (b) this is easy as $\mathbb{A}$ stores $|R_i|$ when reading ρ_{i+1}, and for (c) $\mathbb{A}$ needs to remember only the last node v_j read.

The conditions imply that $v_1, \dots, v_s$ lists R_i. $\mathbb{A}$ increases a counter if $v_j = u$ or $(v_j, u) \in E$ for at least one $j \in [s]$; this happens if and only if $u \in R_{i+1}$. Thus, having read ρ_{i+1}, the counter stores $|R_{i+1}|$.

Further counters signal $\mathbb{A}$ when it is reading ρ_{n-1} and therein $\pi_{v'}$; then $\mathbb{A}$ accepts if during reading $\pi_{v'}$ the counter is not increased. $\square$

Definition 3.3.12 For $\mathcal{C} \subseteq P(\{0,1\}^*)$ let $\mathsf{co}\mathcal{C} := \big\{\{0,1\}^* \setminus Q \mid Q \in \mathcal{C}\big\}$.

Remark 3.3.13 If $\mathsf{co}\mathcal{C} \subseteq \mathcal{C}$, then $\mathcal{C} = \mathsf{coco}\mathcal{C} \subseteq \mathsf{co}\mathcal{C}$, so $\mathcal{C} = \mathsf{co}\mathcal{C}$

It is easy to see that $\mathsf{coTIME}(t) = \mathsf{TIME}(t)$ and $\mathsf{coSPACE}(t) = \mathsf{SPACE}(t)$. Concerning nondeterministic time classes, it is widely believed that $\mathsf{NP} \neq \mathsf{coNP}$ but this is not known – see next chapter. In contrast, for nondeterministic space we get:

Corollary 3.3.14 *If s be space-constructible, then* $\mathsf{NSPACE}(s) = \mathsf{coNSPACE}(s)$. *In particular,* $\mathsf{NL} = \mathsf{coNL}$

Proof: By the remark, it suffices to show $\mathsf{coNSPACE}(s) \subseteq \mathsf{NSPACE}(s)$. Suppose $\{0,1\}^* \setminus Q$ is accepted by a $(c{\cdot}s{+}c)$-space bounded nondeterministic Turing machine $\mathbb{A}$. We can assume that $\mathbb{A}$ on x has at most one reachable accepting configuration c_{acc}. To decide whether $x \in Q$ we check that there is *no* path from c_{start} to c_{acc} in $G_x^{\mathbb{A}}$. We run the logspace verifier from Theorem 3.3.11 without computing $G_x^{\mathbb{A}}$: whenever it wants to check $(c, d) \in E_x^{\mathbb{A}}$, run a logarithmic space subroutine deciding $E_x^{\mathbb{A}}$.

Corollary 3.3.15 2SAT *is* NL*-complete under* $\leqslant_{log}$.

Chapter 4

Alternation

4.1 Co-nondeterminism

Intuitively, a 'proof' of a statement is an object that allows an easy check of its truth. Statements of concern are "$x \in Q$" for some problem Q. The following abstract notion takes this property as definitorial:

Definition 4.1.1 Let Q be a nonempty problem. A *proof system for Q* is a polynomial time computable function $P : \{0,1\}^* \to \{0,1\}^*$ with image Q; pre-images of $x \in Q$ are *P-proofs* of x. If for some polynomial p every $x \in Q$ has a P-proof y with $|y| \leqslant p(|x|)$, then P is *polynomially bounded.*

Proposition 4.1.2 *A nonempty problem Q is in* NP *if and only if there is a polynomially bounded proof system for Q.*

Proof: Let $Q = dom(R)$ for R in P polynomially bounded. A polynomially bounded proof system P for Q is obtained by mapping $\langle x, y\rangle$ with $(x,y) \in R$ to x and all other strings to some fixed element of Q.

Conversely, if P is a proof system for Q that is polynomially bounded, say via p, then Q is the domain of $R := \{(P(y), y) \mid |y| \leqslant p(|x|)\}$. □

Mathematical logic studies many proof systems for

TAUT	
Input:	a propositional formula α.
Problem:	is α tautological?

and it is open whether there is one that is polynomially bounded. This is equivalent to $\mathsf{NP} = \mathsf{coNP}$ (recall Definition 3.3.12):

Proposition 4.1.3 TAUT *is* coNP*-complete.*

TAUT	
Input:	a propositional formula α.
Problem:	is α valid?

Proof: Since SAT is NP-complete, its complement is coNP-complete. But TAUT $\equiv_p \{0,1\}^* \setminus$ SAT: note a formula α is *not* satisfiable if and only if $\neg\alpha$ is valid. □

Proposition 4.1.4 *Let $Q \neq \{0,1\}^*$ be a problem. The following are equivalent.*

1. *$Q \in$ coNP.*
2. *There is a polynomially bounded proof system for $\{0,1\}^* \setminus Q$.*
3. *There is $R \subseteq (\{0,1\}^*)^2$ in P and a polynomial p such that for all $x \in \{0,1\}^*$*

$$x \in Q \iff \forall y \in \{0,1\}^{\leqslant p(|x|)} : (x,y) \in R.$$

The proof is easy. (2) characterizes coNP as the problems Q where true statements "$x \notin Q$" have short proofs. (3) characterizes coNP with a universal quantifier similarly as NP is defined using an existential one. For NP we turned this into a machine characterization of NP by introducing nondeterministic Turing machines. In Section 4.3.1 we shall proceed similarly for coNP – in a more general context.

By previous propositions NP ∩ coNP contains the problems Q where true statements of the form "$x \in Q$" and "$x \notin Q$" have short proofs. Recall our convention that a run answers "yes" or "no" depending on the bit stored finally in cell 1; we now interpret a blank in this cell as "I don't know". Recall also that χ_Q denotes the characteristic function of Q.

Definition 4.1.5 A nondeterministic Turing machine $\mathbb{A}$ *honestly accepts* a problem Q if for all inputs x every complete run of $\mathbb{A}$ on x reaches a halting configuration with the first cell containing either $\chi_Q(x)$ or $\square$; moreover, at least one run reaches a halting configuration containing $\chi_Q(x)$ in the first cell.

Proposition 4.1.6 *Let Q be a nonempty problem. The following are equivalent.*

1. *$Q \in$ NP ∩ coNP.*
2. *There are polynomially bounded proof systems both for Q and for $\{0,1\}^* \setminus Q$.*
3. *There exists a polynomial time bounded nondeterministic Turing machine $\mathbb{A}$ that honestly accepts Q.*

Proof: By Propositions 4.1.2 and 4.1.4, (1) and (2) are equivalent. To see (1) implies (3), assume $Q \in$ NP ∩ coNP and choose polynomial time nondeterministic Turing machines $\mathbb{A}_1$ and $\mathbb{A}_0$ such that $\mathbb{A}_1$ accepts Q and $\mathbb{A}_0$ accepts its complement. A machine as in (3) is obtained by simulating both $\mathbb{A}_1$ and $\mathbb{A}_0$ and saying "yes" in case $\mathbb{A}_1$ accepts and "no" in case $\mathbb{A}_0$ accepts and "I don't know" otherwise.

Conversely, a machine $\mathbb{A}$ according (3) already witnesses $Q \in$ NP. The machine that simulates $\mathbb{A}$ and accepts precisely if $\mathbb{A}$ says "no", witnesses $Q \in$ coNP. □

Remark 4.1.7 *Cook's program* asks to prove that *natural* proof systems for TAUT known from mathematical logic are not polynomially bounded. The corresponding research area is known as *Proof Complexity*. This line to attack the coNP versus NP question hits at open problems from mathematical logic concerning the independence of certain statements from weak theories of arithmetic.

4.2 Unambiguous nondeterminism

We continue to play around with the acceptance condition for nondeterministic machines.

Definition 4.2.1 The set UP contains precisely those problems Q that are accepted by polynomial time bounded *unambiguous* Turing machines; these are nondeterministic Turing machines which on every input have at most one accepting run.

The inclusions $\mathsf{P} \subseteq \mathsf{UP} \subseteq \mathsf{NP}$ are generally conjectured to be strict.

Definition 4.2.2 A *worst-case one-way function* is a polynomial time computable, honest injection $f : \{0,1\}^* \to \{0,1\}^*$ that is not p-invertible. Being *honest* means that there is a polynomial $p \in \mathbb{N}$ such that for all $x \in \{0,1\}^*$ we have $|x| \leqslant p(|f(x)|)$.

Remark 4.2.3 Cryptography needs that such functions exist. It needs more: for secure encryption functions that are easy to invert on "many" inputs albeit maybe difficult to invert on all inputs are not useful. Instead one needs "average-case" one-way functions.

Proposition 4.2.4 *The following are equivalent.*

1. $\mathsf{P} \neq \mathsf{UP}$.
2. *There is a honest polynomial time computable injection whose image is not in* P.
3. *Worst-case one-way functions exist.*

Proof: To see (1) implies (2), let $\mathbb{A}$ be a polynomial time unambiguous machine accepting a problem $Q \notin \mathsf{P}$. Runs of $\mathbb{A}$ on x are determined by strings $y \in \{0,1\}^{p(|x|)}$ where p is a polynomial. Define the function f to map $\langle x, y\rangle$ to $1x$ if $y \in \{0,1\}^{p(|x|)}$ determines an accepting run of $\mathbb{A}$ on x; other strings z are mapped to $0z$.

Then f is polynomial time computable and honest. It is injective because $\mathbb{A}$ is unambiguous. Its image is not in P because it contains $1x$ if and only if $x \in Q$.

That (2) implies (3) is trivial. To see (3) implies (1), let f be a worst-case one-way function. The following decision problem encapsulates the problem to invert f:

$$Q_f := \{\langle x, y\rangle \mid \exists z \leqslant_{lex} x : f(z) = y\},$$

where $\leqslant_{lex}$ denotes the lexicographic order. The nondeterministic machine that on $\langle x, y\rangle$ guesses z and checks that $z \leqslant_{lex} x$ and $f(z) = y$, witnesses that $Q_f \in \mathsf{UP}$; note it is unambiguous because f is injective. We are left to show $Q_f \notin \mathsf{P}$.

Otherwise f would be p-invertible by 'binary search': given y, check $y \in \mathrm{im}(f)$ by checking in polynomial time that $\langle 1^{p(|y|)}, y\rangle \in Q_f$. Here, p witnesses that f is honest. If the check fails, then reject. Otherwise check successively $\langle 1^{p(|y|)-1}, y\rangle \in Q_f, \langle 1^{|y|^c-2}, y\rangle \in Q_f, \ldots$ until $\langle 1^{\ell-1}, y\rangle \notin Q_f$. Then $|f^{-1}(y)| = \ell$. Check $\langle 01^{\ell-1}, y\rangle \in Q_f$. If so, $f^{-1}(y)$ starts with 0 and otherwise with 1; in other words, $f^{-1}(y)$ has first bit $b_1 := 1 - \chi_{Q_f}(\langle 01^{\ell-1}, y\rangle)$. Similarly, $f^{-1}(y)$ has second bit $b_2 := 1 - \chi_{Q_f}(\langle b_1 01^{\ell-2}, y\rangle)$, and so on. □

Exercise 4.2.5 (2) with 'injection' replaced by 'function' is equivalent to $\mathsf{P} \neq \mathsf{NP}$.

We intend to interpret complexity classes as degrees of difficulties of natural problems, e.g., NP as the degree of difficulty of SAT and NL as the degree of difficulty of REACHABILITY. It is unknown whether $\mathsf{NP} \cap \mathsf{coNP}$ or UP can be understood in this way: complete problems, natural or not, are not known to exist. Propositions 4.1.6 and 4.2.4, however, evidence a certain naturality.

Exercise 4.2.6 *If* $\mathsf{P} \neq \mathsf{UP}$, *then* $\mathsf{P} \neq \mathsf{NP} \cap \mathsf{coNP}$.

4.3 The polynomial hierarchy

That a relation $R \subseteq (\{0,1\}^*)^t$ is in P means that $\{\langle x_1, \ldots, x_t\rangle \mid (x_1, \ldots, x_t) \in R\} \in \mathsf{P}$. The following generalizes the definition of NP.

Definition 4.3.1 Let $t \geqslant 1$. A problem Q is in Σ_t^{P} if there are a relation $R \subseteq (\{0,1\}^*)^{t+1}$ in P and polynomials $p_1, \ldots, p_t$ such that for all $x \in \{0,1\}^*$:

$$x \in Q \iff \exists y_1 \in \{0,1\}^{\leqslant p_1(|x|)} \forall y_2 \in \{0,1\}^{\leqslant p_2(|x|)} \cdots q y_t \in \{0,1\}^{\leqslant p_t(|x|)} : (x, y_1, \ldots, y_t) \in R.$$

Here, q stands for $\forall$ or $\exists$ depending on whether t is even or odd, respectively. Further, we set $\Pi_t^{\mathsf{P}} := \mathsf{co}\Sigma_t^{\mathsf{P}}$, and write PH for the *polynomial hierarchy* $\bigcup_{t\geqslant 1} \Sigma_t^{\mathsf{P}}$.

Observe that $\mathsf{NP} = \Sigma_1^{\mathsf{P}}$ and $\mathsf{coNP} = \Pi_1^{\mathsf{P}}$.

Exercise 4.3.2 One can equivalently write $y_i \in \{0,1\}^{p(|x|)}$ for a single polynomial p.

Exercise 4.3.3 Show that Σ_t^{P} and Π_t^{P} and PH are $\leqslant_p$-closed and contained in $\Sigma_{t+1}^{\mathsf{P}} \cap \Pi_{t+1}^{\mathsf{P}}$.

It is not known whether any of Σ_t^{P} and Π_t^{P} are equal. The hypotheses $\Sigma_t^{\mathsf{P}} \neq \Pi_t^{\mathsf{P}}$ are stepwise strengthenings of the hypothesis $\mathsf{P} \neq \mathsf{NP}$. We shall see that these strengthenings are 'useful' hypotheses, in the sense that they tell us interesting things we do not know how to infer from the weaker hypothesis that $\mathsf{P} \neq \mathsf{NP}$.

Theorem 4.3.4 *Let* $t \geqslant 1$. *If* $\Sigma_t^{\mathsf{P}} = \Pi_t^{\mathsf{P}}$, *then* $\mathsf{PH} = \Sigma_t^{\mathsf{P}}$. *Further, if* $\mathsf{P} = \mathsf{NP}$, *then* $\mathsf{P} = \mathsf{PH}$.

Proof: The second statement follows from the first: assume $\mathsf{P} = \mathsf{NP}$. Then $\mathsf{NP} = \mathsf{coNP}$, so $\mathsf{PH} = \mathsf{NP}$ by the first statement, and hence $\mathsf{PH} = \mathsf{P}$ using the assumption again.

Assuming $\Sigma_t^{\mathsf{P}} = \Pi_t^{\mathsf{P}}$, we show $\Sigma_{t+k}^{\mathsf{P}} \subseteq \Sigma_t^{\mathsf{P}}$ by induction on k. Assume the claim for k and let $Q \in \Sigma_{t+k+1}^{\mathsf{P}}$. Choose a relation R in P such that

$$x \in Q \iff \exists y_1 \forall y_2 \cdots q y_{t+k+1} : (x, y_1, \ldots, y_{t+k+1}) \in R.$$

Here, the y_is range over $\{0,1\}^{\leqslant p_i(|x|)}$ for suitable polynomials p_i. Observe

$$x \in Q \iff \exists y_1 : \langle x, y_1 \rangle \in Q'$$

for suitable $Q' \in \Pi_{t+k}^{\mathsf{P}}$. But $\Pi_{t+k}^{\mathsf{P}} = \mathsf{co}\Sigma_{t+k}^{\mathsf{P}} \subseteq \mathsf{co}\Sigma_t^{\mathsf{P}} = \Pi_t^{\mathsf{P}} = \Sigma_t^{\mathsf{P}}$, where the inclusion follows from the induction hypothesis. Thus $Q' \in \Sigma_t^{\mathsf{P}}$ and we can write

$$\langle x, y_1 \rangle \in Q' \iff \exists y_2' \forall y_3' \cdots q y_{t+1}' : (\langle x, y_1 \rangle, y_2', \ldots, y_{t+1}') \in R'$$

for some R' in P. Here, the y_i's range over $\{0,1\}^{\leqslant p_i'(|x|)}$ for suitable polynomials p_i'.

Let R'' contain the tuples $(x, z, y_3', \ldots, y_{t+1}')$ such that there are y_1 with $|y_1| \leqslant p_1(|x|)$ and y_2' with $|y_2'| \leqslant p_2'(|x|)$ such that $z = \langle y_1, y_2' \rangle$ and $(\langle x, y_1 \rangle, y_2', y_3' \ldots, y_{t+1}')$ is in R'. Then R'' witnesses that $Q \in \Sigma_t^{\mathsf{P}}$. □

Corollary 4.3.5 *If there is a problem that is complete for* PH*, then* PH *collapses, i.e.* $\mathsf{PH} = \Sigma_t^{\mathsf{P}}$ *for some* $t \geqslant 1$.

Proof: If Q is complete for PH, then $Q \in \Sigma_t^{\mathsf{P}}$ for some t. As Σ_t^{P} is $\leqslant_p$-closed, $\mathsf{PH} \subseteq \Sigma_t^{\mathsf{P}}$. □

Thus, PH cannot be interpreted as the degree of difficulty of some problem unless it collapses. We now observe that the levels Σ_t^{P} do have natural complete problems.

Definition 4.3.6 A quantified Boolean formula is a Σ_t-formula if it has the form

$$\exists \bar{X}_1 \forall \bar{X}_2 \cdots q \bar{X}_t \beta$$

where β is quantifier free, and q is $\forall$ for even t and $\exists$ for odd t.

Theorem 4.3.7 *For every* $t \geqslant 1$, Σ_tSAT *is* Σ_t^{P}*-complete.*

Σ_tSAT	
Input:	a Σ_t-sentence α.
Problem:	is α true?

Proof: It is easy to see that $\Sigma_t\text{SAT} \in \Sigma_t^{\mathsf{P}}$. The hardness proof generalizes the proof of the Cook-Levin Theorem 2.3.9. Given $Q \in \Sigma_t^{\mathsf{P}}$ choose a relation R in P such that

$$x \in Q \iff \exists y_1 \forall y_2 \cdots q y_t : (x, y_1, \ldots, y_t) \in R,$$

for all $x \in \{0,1\}^n$, with the y_is ranging over $\{0,1\}^{p(|x|)}$ for some polynomial p (Exercise 4.3.2). As R is in P we can write the condition $(x, y_1, \ldots, y_t) \in R$ as $C(x, y_1, \ldots, y_t) = 1$

for some suitable circuit C computable in polynomial time from $|x|$ (by the Fundamental Lemma 1.3.9). By Exercise 3.2.2 the latter is equivalent to the truth of $q\bar{Z}\,\alpha(\bar{Z}, x, y_1, \ldots, y_t)$ for some quantifier free formula $\alpha(\bar{Z}, \bar{X}, \bar{Y}_1, \ldots, \bar{Y}_t)$ which is computable from C in polynomial time. Then

$$x \mapsto \exists\bar{Y}_1\forall\bar{Y}_2\cdots q\bar{Y}_t\bar{Z}\ \alpha(\bar{Z}, x, \bar{Y}_1, \ldots, \bar{Y}_t)$$

defines the desired reduction. □

Corollary 4.3.8 $\mathsf{P} \subseteq \mathsf{NP} = \Sigma_1^{\mathsf{P}} \subseteq \Sigma_2^{\mathsf{P}} \subseteq \cdots \subseteq \mathsf{PH} \subseteq \mathsf{PSPACE}$ *and all inclusions are strict unless* PH *collapses.*

Proof: To see that $\mathsf{PH} \subseteq \mathsf{PSPACE}$ note $\Sigma_t\textsc{Sat} \leqslant_p \text{QBF} \in \mathsf{PSPACE}$ for every $t \geqslant 1$. The inclusion is strict by Corollary 4.3.5 since PSPACE has complete problems. The other inclusions are trivial and strict by Theorem 4.3.4: $\Sigma_t^{\mathsf{P}} = \Sigma_{t+1}^{\mathsf{P}}$ implies $\Pi_t^{\mathsf{P}} \subseteq \Sigma_t^{\mathsf{P}}$ and thus $\Sigma_t^{\mathsf{P}} = \mathsf{co}\Pi_t^{\mathsf{P}} \subseteq \mathsf{co}\Sigma_t^{\mathsf{P}} = \Pi_t^{\mathsf{P}}$, so $\Sigma_t^{\mathsf{P}} = \Pi_t^{\mathsf{P}}$. □

4.3.1 Alternating time

We give a machine characterization of Σ_t^{P} analogously as we did for NP in Proposition 2.2.3.

Definition 4.3.9 An *alternating* Turing machine is a nondeterministic Turing machine $\mathbb{A}$ whose set of states is partitioned into a set of *existential* and a set of *universal* states. It is said to *accept* a problem Q if it accepts precisely the strings x which are in Q. That it *accepts* x means that the starting configuration of $\mathbb{A}$ on x is good, where the set of *good* configurations is the minimal set satisfying

- an accepting halting configuration is good,
- a configuration with an existential state is good, if it has some good successor,
- a configuration with a universal state is good, if all its successors are good.

For $t \geqslant 1$, $\mathbb{A}$ is *t-alternating* if starting state is existential and every run of $\mathbb{A}$ on any input moves less than t times from an existential to a universal state or vice versa.

For $f : \mathbb{N} \to \mathbb{N}$ the set $\mathsf{ATIME}(f)$ (resp. $\Sigma_t\mathsf{TIME}(f)$) contains the problems Q such that there are $c \in \mathbb{N}$ and a $(c \cdot f + c)$-time bounded (resp. t-)alternating machine that accepts Q. *Alternating polynomial time* and *t-alternating polynomial time* are, respectively,

$$\begin{aligned} \mathsf{AP} &:= \textstyle\bigcup_{c\in\mathbb{N}} \mathsf{ATIME}(n^c), \\ \Sigma_t\mathsf{P} &:= \textstyle\bigcup_{c\in\mathbb{N}} \Sigma_t\mathsf{TIME}(n^c). \end{aligned}$$

Note $\Sigma_1\mathsf{TIME}(f) = \mathsf{NTIME}(f)$ and especially $\Sigma_1^{\mathsf{P}} = \mathsf{NP} = \Sigma_1\mathsf{P}$. More generally:

Proposition 4.3.10 For all $t \geqslant 1$, $\Sigma_t\mathsf{P} = \Sigma_t^{\mathsf{P}}$.

Proof: It is easy to solve $\Sigma_t\textsc{Sat}$ by a t-alternating machine in polynomial time. It is also easy to see that $\Sigma_t\mathsf{P}$ is $\leqslant_p$-closed. This implies $\supseteq$ by Theorem 4.3.7.

To see $\subseteq$, let Q be accepted by a p-time bounded t-alternating machine $\mathbb{A}$, where p is some polynomial. Every run of $\mathbb{A}$ on x can be divided into $\leqslant t$ subruns such that the first block contains only configurations with existential states, the second only universal states, and so on. Let $R \subseteq (\{0,1\}^*)^{t+1}$ contain those $(x, y_1, \ldots, y_t)$ such that the run of $\mathbb{A}$ on x 'determined' by $(y_1, \ldots, y_t) \in (\{0,1\}^{p(|x|)})^t$ is accepting: namely, the run taking the jth step in the ith block according to the jth bit of y_i. Then R witnesses $Q \in \Sigma_t^{\mathrm{P}}$. □

Proposition 4.3.11 $\mathsf{AP} = \mathsf{PSPACE}$.

Proof: Note that AP is $\leqslant_p$-closed. By Theorem 3.2.4 it thus suffices to show that $\textsc{Qbf}$ is AP-complete. It is easy to see that $\textsc{Qbf} \in \mathsf{AP}$. For hardness, suppose Q is accepted by a p-time bounded alternating Turing machine $\mathbb{A}$ where p is some polynomial. Then

$$x \in Q \iff \exists y_1 \forall z_1 \cdots \exists y_{p(|x|)} \forall z_{p(|x|)}\ (x, y, z) \in R$$

where y_i, z_i range over bits $\{0,1\}$. Let R contain the triples $(x, y_1 \cdots y_{p(|x|)}, z_1 \cdots z_{p(|x|)})$ such that the following run of $\mathbb{A}$ on x is accepting: the ith step is detemined by either y_i or z_i depending on whether the state of the current configuration is existential of universal. Now continue as in the proof of Theorem 4.3.7. □

Exercise 4.3.12 Define $\Pi_t\mathsf{P}$ like $\Sigma_t\mathsf{P}$ except that the starting state s_{start} is demanded to be universal. Show that $\Pi_t\mathsf{P} = \mathsf{co}\Sigma_t\mathsf{P}$, in particular, $\Pi_1\mathsf{P} = \mathsf{coNP}$.

4.3.2 Oracles

In actual programming one uses instructions to call a previously defined program for another problem X. This allows to study the complexity of problems relative to X. Theoretically, this makes sense for every X, e.g., undecidable X.

Recall, χ_X denotes the characteristic function of $X \subseteq \{0,1\}^*$.

Definition 4.3.13 Let X be a problem. An *oracle machine with oracle* X is a (deterministic or nondeterministic) Turing machine with a special worktape, called *oracle tape*, and special states $s_?, s_0, s_1$; the successor of a configuration with state $s_?$ is the same configuration but with $s_?$ changed to s_b where $b = \chi_X(z)$ and z is the *query* of the configuration: the string of bits stored on the oracle tape in cell number one up to exclusively the first blank cell; such a step in the computation is a *query step*.

The class P^X (NP^X) contains the problems accepted by a polynomial time bounded (nondeterministic) oracle machine with oracle X.

For $\mathcal{C} \subseteq P(\{0,1\}^*)$ we set $\mathsf{P}^{\mathcal{C}} := \bigcup_{X \in \mathcal{C}} \mathsf{P}^X$ and $\mathsf{NP}^{\mathcal{C}} := \bigcup_{X \in \mathcal{C}} \mathsf{NP}^X$.

Exercise 4.3.14 $\mathsf{NP}^{\mathcal{C}}$ is $\leqslant_p$-closed for every $\mathcal{C} \subseteq P(\{0,1\}^*)$. If $Q \subseteq \{0,1\}^*$ is $\mathcal{C}$-hard, then $\mathsf{NP}^{\mathcal{C}} \subseteq \mathsf{NP}^Q$.

Exercise 4.3.15 To address a maybe confusing point in the notation, think about whether $\mathsf{P} = \mathsf{NP}$ implies $\mathsf{P}^X = \mathsf{NP}^X$ for all $X \subseteq \{0,1\}^*$.

Theorem 4.3.16 *For every* $t \geqslant 1$, $\Sigma^{\mathsf{P}}_{t+1} = \mathsf{NP}^{\Sigma_t\text{SAT}}$.

Proof: For $\subseteq$ it suffices to show $\Sigma_{t+1}\text{SAT} \in \mathsf{NP}^{\Sigma_t\text{SAT}}$ by Theorem 4.3.7. Take the machine that given as input a Σ_{t+1}-sentence $\exists\bar{X}\forall\bar{Y}\exists\bar{Z}\cdots\alpha(\bar{X},\bar{Y},\bar{Z},\ldots)$ first guesses values $\bar{b}$ for $\bar{X}$ and asks the oracle for the truth of the Σ_t-sentence $\exists\bar{Y}\forall\bar{Z}\cdots\neg\alpha(\bar{b},\bar{Y},\bar{Z},\ldots)$, i.e. it writes this sentence on the oracle tape and enters state $s_?$. If the oracle answers "no", i.e. the query step ends in s_0, then the machine accepts; otherwise it rejects.

Conversely, suppose p is a polynomial and Q is accepted by a p-time bounded nondeterministic oracle machine $\mathbb{A}$ with oracle $X := \Sigma_t\text{SAT}$. We can assume that $\mathbb{A}$ only makes queries that encode Σ_t-sentences. Say, a query step is *according to* a bit b if it ends in state s_b. As usual, we say a non query step is according to 0 or 1, if the first or the second transition function is used. Then $\mathbb{A}$ accepts an input x if and only if the following holds: there exists strings $y, a \in \{0,1\}^{\leqslant p(|x|)}$ the run of $\mathbb{A}$ with ith query step according a_i and ith non query step according y_i is accepting and produces a sequence of queries $\alpha_1,\ldots,\alpha_{|a|}$ such that $a = \chi_X(\alpha_1)\cdots\chi_X(\alpha_{|a|})$ – this means that the sentence

$$\beta := \bigwedge_{i=1}^{|a|} \neg^{1-a_i}\alpha_i,$$

is true; recall the notation $\neg^0\alpha = \alpha$ and $\neg^1\alpha = \neg\alpha$. Elementary formula manipulations allow to transform β to an equivalent Σ_{t+1}-sentence β' in polynomial time. Thus $Q \in \Sigma_{t+1}\mathsf{P} = \Sigma^{\mathsf{P}}_{t+1}$ (Proposition 4.3.10) is witnessed by the following $(t+1)$-alternating Turing machine: first existentially guess (via nondeterministic steps in existential states) the strings y, a and then simulate the run of $\mathbb{A}$ determined by y and a; check it accepts and β' is true – this can be done in $(t+1)$-alternating polynomial time (Proposition 4.3.10). $\square$

Corollary 4.3.17 $\Sigma^{\mathsf{P}}_1 = \mathsf{NP}$, $\Sigma^{\mathsf{P}}_2 = \mathsf{NP}^{\mathsf{NP}}$, $\Sigma^{\mathsf{P}}_3 = \mathsf{NP}^{\mathsf{NP}^{\mathsf{NP}}}$, $\Sigma^{\mathsf{P}}_4 = \mathsf{NP}^{\mathsf{NP}^{\mathsf{NP}^{\mathsf{NP}}}}$ $\ldots$.

Proof: $\Sigma^{\mathsf{P}}_{t+1} = \mathsf{NP}^{\Sigma_t\text{SAT}} = \mathsf{NP}^{\Sigma^{\mathsf{P}}_t}$ by Theorems 4.3.16 and 4.3.7. $\square$

Exercise 4.3.18 Define $(\mathsf{NP}^{\mathsf{NP}})^{\mathsf{NP}}$ and prove that it equals $\mathsf{NP}^{\mathsf{NP}}$.

Exercise 4.3.19 For $t \geqslant 1$ show $\Sigma^{\mathsf{P}}_t \cup \Pi^{\mathsf{P}}_t \subseteq \mathsf{P}^{\Sigma^{\mathsf{P}}_t} = \mathsf{P}^{\Pi^{\mathsf{P}}_t} \subseteq \Sigma^{\mathsf{P}}_{t+1} \cap \Pi^{\mathsf{P}}_{t+1}$.

4.4 Time-space trade-offs

Definition 4.4.1 Let $t, s : \mathbb{N} \to \mathbb{N}$. The class $\mathsf{TISP}(t,s)$ contains the problems decidable by a (deterministic) Turing machine with input tape that is both $(c \cdot t + c)$-time bounded and $(c \cdot s + c)$-space bounded for some constant $c \in \mathbb{N}$.

Theorem 4.4.2 (Lipton, Viglas 1999) *If* $a < \sqrt{2}$, *then* $\mathsf{NTIME}(n) \not\subseteq \mathsf{TISP}(n^a, n^{o(1)})$.

We prove this via three lemmas. The first is proved by padding:

Lemma 4.4.3 *Let $t, s : \mathbb{N} \to \mathbb{N}$ be functions and $c \in \mathbb{N}$. If $\mathsf{NTIME}(n) \subseteq \mathsf{TISP}(t(n), s(n))$, then $\mathsf{NTIME}(n^c) \subseteq \mathsf{TISP}(n \cdot t(n^c), s(n^c))$.*

Proof: We can assume $c > 1$. Let $Q \in \mathsf{NTIME}(n^c)$ and choose a nondeterministic Turing machine $\mathbb{A}$ accepting Q in time $O(n^c)$. Further, let

$$Q' := \{ \underbrace{xx\cdots x}_{|x|^{c-1} \text{ times}} \mid x \in Q\}.$$

To see that Q' is in linear nondeterministic time it suffices to check, given an input y of length m, that y has the form $x\cdots x$ above and, in case, compute x. First, guess an initial segment x of y and compute $n := |x|$. This takes time $O(m)$ using Exercise 1.2.7. Alternatively, use a lavish time $O(n^2)$ algorithm and reject if it does not halt in time m: time m suffices if m is large enough and the guess is right. Second, compute n^{c-1} (in binary): this takes time polynomial in $\log n$, so at most m if m is large enough. Third, check $y = x\cdots x$ by scanning y from left to right; during reading each copy of x increase a binary counter (time $10 \log n < n$ if n is large enough) and finally check it equals n^{c-1}.

By assumption Q' is decided by a machine $\mathbb{B}$ running in time $O(t)$ and space $O(s)$. Now decide Q as follows. On x, run $\mathbb{B}$ on $xx\cdots x$ without computing $xx\cdots x$: instead, make sure the input head always scans the same bit as $\mathbb{B}$'s input head; e.g., if $\mathbb{B}$ moves right from one x-copy into the next, then move to cell 1. Clearly, $O(|x|)$ steps suffice for one step of $\mathbb{B}$. Hence, the simulation runs in time $O(|x| \cdot t(|x|^c))$ and space $O(s(|x|^c))$. □

Lemma 4.4.4 *Let $a, b \geqslant 1$ be rational. If $\mathsf{NTIME}(n) \subseteq \mathsf{TIME}(n^a)$, then $\Sigma_2\mathsf{TIME}(n^b) \subseteq \mathsf{NTIME}(n^{ba})$.*

Proof: Let $Q \in \Sigma_2\mathsf{TIME}(n^b)$. By the proof of Proposition 4.3.10 there is $R \subseteq (\{0,1\}^*)^3$ decidable in linear time such that for all long enough $x \in \{0,1\}^*$

$$x \in Q \iff \exists y_1 \forall y_2 : (x, y_1, y_2) \in R,$$

where y_1, y_2 range over $\{0,1\}^{|x|^b}$; here and below we write $|x|^b$ instead $\lfloor |x|^b \rfloor$. We claim

$$Q' := \{\langle x, y_1\rangle \mid \exists y_2 : (x, y_1, y_2) \notin R\}$$

can be accepted in time $O(|x|^b)$: guess and and verify the binary representation of $|x|^b$; compute $1^{|x|^b}$; guess y_2 of length $\leqslant |x|^b$; check $(x, y_1, y_2) \notin R$.

By assumption, whether $\langle x, y_1\rangle \in Q'$ can be decided in deterministic time $O(|x|^{ba})$. But

$$x \in Q \iff \exists y_1 : \langle x, y_1\rangle \notin Q',$$

and $Q \in \mathsf{NTIME}(n^{ba})$ follows. □

The third and final lemma is the heart of the proof. It states that, in certain contexts, time can be traded for alternation.

Lemma 4.4.5 *Let $c \in \mathbb{N}, c \geqslant 1$. Then $\mathsf{TISP}(n^{2c}, n^{o(1)}) \subseteq \Sigma_2\mathsf{TIME}(n^{c+\epsilon})$ for every real $\epsilon > 0$.*

Proof: Let Q be a problem decided by a machine $\mathbb{A}$ in time $O(n^{2c})$ and space $n^{o(1)}$. Let $e \geqslant 1$ be an arbitrary natural. Every configuration of $\mathbb{A}$ on $x \in \{0,1\}^n$ can be coded by a string of length $\leqslant n^{1/e}$ (for sufficiently large n). The 2-alternating machine existentially guesses n^c such codes $c_1, \ldots, c_{n^c}$ and verifies that the last one is accepting. This can be done in time $O(n^c \cdot n^{1/e})$. Understanding c_0 as the start configuration, it then universally guesses an index $i < n^c$ and verifies that $\mathbb{A}$ reaches c_{i+1} from c_i in at most n^c many steps.

Since e was arbitrary it suffices to show that this algorithm can be implemented in time $O(n^{c+2/e})$. First, to guess c_i the machine needs to know $\lfloor n^{1/e} \rfloor$: guess an verify this number in time $O(n)$. Second, for the simulation of $\mathbb{A}$, start with c_i and successively compute (codes of) successor configurations. Given a configuration c plus the input bit scanned its successor can be computed in quadratic time $O(|c|^2)$ (for some reasonable encoding; besides 2, also any other constant would be good enough). In order to have the input bit available we move our input head according to the simulation; in the beginning we move it to the cell scanned by c_i. Thus the simulation runs in time $O(n^{2/e} \cdot n^c)$. $\square$

Proof of Theorem 4.4.2: Assume $\mathsf{NTIME}(n) \subseteq \mathsf{TISP}(n^a, n^{o(1)})$ for some rational $\sqrt{2} > a \geqslant 1$. Let $c \in \mathbb{N}$ be 'large enough' and such that $(2c-1)/a \geqslant 1$ is integer. Then

$$\begin{array}{rcll} \mathsf{NTIME}(n^{(2c-1)/a}) & \subseteq & \mathsf{TISP}(n^{2c}, n^{o(1)}) & \text{by Lemma 4.4.3} \\ & \subseteq & \Sigma_2\mathsf{TIME}(n^{c+(1/a)}) & \text{by Lemma 4.4.5} \\ & \subseteq & \mathsf{NTIME}(n^{ac+1}) & \text{by Lemma 4.4.4.} \end{array}$$

By the Nondeterministic Time Hierarchy Theorem in the strong form of Remark 2.2.12, we arrive at a contradiction if $(2c-1)/a > ac+1$, and thus if $\frac{2c-1}{c+1} > a^2$. This is true for 'large enough' c, because $\frac{2c-1}{c+1} \to_c 2 > a^2$. $\square$

Remark 4.4.6 The bound $a < \sqrt{2} \approx 1.41$ in Theorem 4.4.2 has been improved to $a < 2\cos(\pi/7) \approx 1.80$. This marks the current record due to Williams, 2007. Getting to arbitrarily large a would entail $\textsc{Sat} \notin \mathsf{L}$, and thus $\mathsf{NP} \neq \mathsf{L}$.

Chapter 5

Size

5.1 Non-uniform polynomial time

The class $\mathsf{P/poly}$ contains the problems that can be solved by small circuits. It is generally believed that $\mathsf{NP} \notin \mathsf{P/poly}$ but currently even $\mathsf{NEXP} \notin \mathsf{P/poly}$ is unknown.

Definition 5.1.1 The set $\mathsf{P/poly}$ contains the problems Q such that there exists a polynomial p such that for all $n \in \mathbb{N}$ there exists a size $\leqslant p(n)$ circuit that decides Q on $\{0,1\}^n$

Exercise 5.1.2 $\mathsf{P/poly}$ is $\leqslant_p$-closed.

By the Fundamental Lemma 1.3.9 this is a 'non-uniform version' of P in that there are no computability conditions imposed on the map $n \mapsto C$. It is thus clear that $\mathsf{P} \subseteq \mathsf{P/poly}$. The converse is fails because $\mathsf{P/poly}$ contains undecidable problems.

Definition 5.1.3 Let $a : \mathbb{N} \to \{0,1\}^*$. A *Turing machine with advice* a is just a Turing machine $\mathbb{A}$ but its starting configuration on x is redefined to be the starting configuration of $\mathbb{A}$ on $\langle x, a(|x|)\rangle$. The advice a is *polynomially bounded* if there is a polynomial p such that $|a(n)| \leqslant p(n)$ for all $n \in \mathbb{N}$.

Recall Definition 2.5.1 of sparsity.

Proposition 5.1.4 *Let Q be a problem. The following are equivalent.*

1. *$Q \in \mathsf{P/poly}$.*
2. *There is a polynomial time bounded Turing machine with polynomially bounded advice that decides Q.*
3. *There are $Q' \in \mathsf{P}$ and $a : \mathbb{N} \to \{0,1\}^*$ polynomially bounded such that for all $x \in \{0,1\}^*$*

$$x \in Q \iff \langle x, a(|x|)\rangle \in Q'.$$

4. *There is a sparse $X \subseteq \{0,1\}^*$ such that $Q \in \mathsf{P}^X$.*

Proof: (1) implies (2): let the advice $a(n)$ be the code of C_n and on input $x \in \{0,1\}^n$ evaluate C_n on x.

(2) implies (3): let Q' contain the strings $\langle x, y\rangle$ that are accepted by the machine in (2).

(3) implies (1): use the Fundamental Lemma 1.3.9 to obtain for $n, m \in \mathbb{N}$ a circuit $D_{n,m}(X_1 \cdots X_n, Y_1 \cdots Y_m)$ of size polynomial in $(n+m)$ such that for all $(x, y) \in \{0,1\}^n \times \{0,1\}^m$

$$\langle x, y\rangle \in Q' \Longleftrightarrow D_{n,m}(x, y) = 1.$$

Then $C_n(\bar{X}) := D_{n,|a(n)|}(\bar{X}, a(n))$ decides Q on $\{0,1\}^n$.

(4) implies (2): assume a p-time bounded $\mathbb{A}$ with sparse oracle X decides Q, where p is a polynomial. Simulate $\mathbb{A}$ in polynomial time using as advice $a(n)$ a list of all elements of $X \cap \{0,1\}^{\leqslant p(n)}$. Note $|a(n)| \leqslant n^{O(1)}$ since X is sparse.

(2) implies (4): choose $\mathbb{A}, a$ according (2). It suffices to find a sparse X such that $1^n \mapsto a(n)$ can be computed in polynomial time with oracle X. We can assume that $|a(n)| = n^c$ for some $c \in \mathbb{N}$ and large enough n. Then let X contain for $1 \leqslant i \leqslant n^c$ such that the ith bit of $a(n)$ is 1 the string $0 \cdots 010 \cdots 0$ of length n^c with a 1 at the ith position. □

Exercise 5.1.5 Formulate and prove a variant of this proposition for NP instead of P.

Exercise 5.1.6 Define $\mathsf{NEXP/poly}$ to contain the problems that are accepted by exponential time bounded nondeterministic Turing machines with polynomially bounded advice. Show $\mathsf{NEXP/poly}$ is closed under complementation.

Hint: the advice for $\{0,1\}^n \setminus Q$ contains the number $|Q \cap \{0,1\}^n|$ (in binary).

5.1.1 Karp and Lipton's theorem

Assuming the polynomial hierarchy does not collapse we show that polynomial advice does not help to solve SAT. We need the following non-uniform analogue of Theorem 2.6.2.

Exercise 5.1.7 If SAT $\in \mathsf{P/poly}$, then there is a polynomial time machine with polynomially bounded advice that when given a satisfiable formula outputs a satisfying assignment.

Theorem 5.1.8 (Karp-Lipton 1980) *If* $\mathsf{NP} \subseteq \mathsf{P/poly}$, *then* $\mathsf{PH} = \Sigma_2^{\mathsf{P}}$.

Proof: Assume SAT $\in \mathsf{P/poly}$ and choose $\mathbb{A}$ according to the above exercise, say, $\mathbb{A}$ uses advice $a : \mathbb{N} \to \{0,1\}^*$ satisfying $|a(n)| \leqslant n^c$ for $n > 1$. A formula α is satisfiable if and only if the output $\mathbb{A}(\langle \alpha, a(|\alpha|)\rangle)$ is a satisfying assignment of α.

By Theorem 4.3.4 (and Remark 3.3.13) it suffices to show $\Pi_2^{\mathsf{P}} \subseteq \Sigma_2^{\mathsf{P}}$. Theorem 4.3.7 implies that Π_2SAT is Π_2^{P}-complete. Hence, by Proposition 4.3.10, it suffices to give a polynomial time 2-alternating Turing machine accepting Π_2SAT.

Π_2SAT	
Input:	$\forall \bar{X} \exists \bar{Y} \beta$ where $\beta = \beta(\bar{X}, \bar{Y})$ is a quantifier free Boolean formula.
Problem:	is $\forall \bar{X} \exists \bar{Y} \beta$ true?

On input $\forall\bar{X}\exists\bar{Y}\beta(\bar{X},\bar{Y})$ of length n, such a machine checks

$$\exists a\in\{0,1\}^{\leqslant n^c}\ \forall x\in\{0,1\}^{|\bar{X}|}:\quad \mathbb{A}(\langle\beta(x,\bar{Y}),a\rangle)\text{ satisfies }\beta(x,\bar{Y}).$$

Clearly, this condition implies that $\forall\bar{X}\exists\bar{Y}\beta(\bar{X},\bar{Y})$ is true. To see the converse, we assume that for all x the encoding length of $\beta(x,\bar{Y})$ is at most n and depends only on β and not on x. If this length is m we can choose $a:=a(m)$. □

The following is an analogue for EXP instead NP. It is proved by different means, namely by a similar trick as in Lemma 4.4.5. We sketch the proof and leave it as an exercise to fill in the details. Interestingly, we see that $\mathsf{P}\neq\mathsf{NP}$ could be proved by establishing circuit *upper bounds* – Exercise 5.2.4 and Corollary 5.2.7 improve on this.

Theorem 5.1.9 (Meyer) *If* $\mathsf{EXP}\subseteq\mathsf{P/poly}$, *then* $\mathsf{EXP}\subseteq\Sigma_2^{\mathsf{P}}$ *and* $\mathsf{P}\neq\mathsf{NP}$.

Proof: (Sketch) Let $Q\in\mathsf{EXP}$, say, witnessed by $\mathbb{A}$. Let $f_{\mathbb{A}}(x,i)$ output the i-th bit of the computation table of $\mathbb{A}$ on x. By assumption $f_{\mathbb{A}}$ is computed by a small circuit. Decide Q in 2-alternating polynomial time as follows. Given x, existentially guess a small circuit and check it computes $f_{\mathbb{A}}(x,\cdot)$. This can be done in polynomial time after universally guessing some entries of the computation table. Hence $Q\in\Sigma_2^{\mathsf{P}}$. But $\mathsf{EXP}\subseteq\Sigma_2^{\mathsf{P}}$ implies $\mathsf{P}\neq\mathsf{NP}$ by Theorem 4.3.4 and Corollary 1.2.14. □

5.2 Shannon's theorem and size hierarchy

Theorem 5.2.1 (Shannon 1949) *Let* $Sh:\mathbb{N}\to\mathbb{N}$ *be defined by*

$$Sh(n):=\max_{f:\{0,1\}^n\to\{0,1\}}\min\big\{s\mid\text{there is a circuit of size}\leqslant s\text{ computing }f\big\}.$$

Then $Sh(n)=\Theta(2^n/n)$.

Proof: $Sh(n)\leqslant O(2^n/n)$ follows from Proposition 1.3.8. To show $Sh(n)\geqslant\Omega(2^n/n)$ we show there are less circuits of size $\leqslant s:=2^n/(10n)$ than Boolean functions on $\{0,1\}^n$. Each equation of a size $\leqslant s$ circuit can be encoded by, say, $5\cdot\log(s)$ bits, and the circuit by, say, $10\cdot s\cdot\log(s)$ bits. Hence there are at most $2^{10\cdot s\cdot\log(s)}$ functions computed by such circuits. This is $<2^{2^n}$: note $10\cdot s\cdot\log(s)=10\cdot(2^n/10n)\cdot(n-\log n-\log 10)<2^n$. □

Remark 5.2.2 The vast majority of functions $f:\{0,1\}^n\to\{0,1\}$ is not computable by circuits of size $\leqslant s:=2^n/(11n)$. Indeed, if we choose $f:\{0,1\}^n\to\{0,1\}$ uniformly at random, then it is computed by such a circuit with probability $<2^{\frac{10}{11}\cdot 2^n}/2^{2^n}=2^{-2^n/11}$.

Definition 5.2.3 Let $s:\mathbb{N}\to\mathbb{N}$. The set $\mathsf{SIZE}(s)$ contains the problems Q such that for all large enough $n\in\mathbb{N}$ there exists a size $\leqslant s(n)$ circuit that decides Q on $\{0,1\}^n$.

Note $\mathsf{P/poly}=\bigcup_{c\in\mathbb{N}}\mathsf{SIZE}(n^c)$.

Exercise 5.2.4 $\mathsf{EXP} \subseteq \mathsf{P/poly}$ if and only if $\mathsf{E} \subseteq \mathsf{SIZE}(n^c)$ for some $c \in \mathbb{N}$ (*Hint:* recall Exercise 2.3.8).

Theorem 5.2.5 (Size hierarchy) *$\mathsf{SIZE}(s) \setminus \mathsf{SIZE}(s') \neq \varnothing$ for all $s, s' : \mathbb{N} \to \mathbb{N}$ nondecreasing and unbounded with*

$$840 \cdot s'(n) \leqslant s(n) \leqslant 2^n/n.$$

Proof: By the proofs of Proposition 1.3.8 and Theorem 5.2.1, for large enough ℓ, there is a function $f_\ell : \{0,1\}^\ell \to \{0,1\}$ which is computable by circuits of size $\leqslant 21 \cdot 2^\ell/\ell$ but not by circuits of size $\leqslant 2^\ell/(10\ell)$. For large enough n we claim there is a natural $\ell \leqslant n$ such that $s'(n) \leqslant 2^\ell/(10\ell)$ and $21 \cdot 2^\ell/\ell \leqslant s(n)$. This suffices: define $Q \in \mathsf{SIZE}(s) \setminus \mathsf{SIZE}(s')$ by declaring $x_1 \cdots x_n \in Q$ if and only if $f_\ell(x_1 \cdots x_\ell) = 1$.

For the claim, it suffices to find a natural ℓ in the real interval $[a, b]$ where

$$\begin{aligned} a &:= \log s'(n) + \log\log s'(n) + \log 20; \\ b &:= \log s(n) + \log\log s(n) - \log 21. \end{aligned}$$

Indeed, $\ell \leqslant n$ by $\ell \leqslant b$ and $s(n) \leqslant 2^n/n$. Note that, as functions on $\mathbb{R}$, both $2^\ell/(10\ell)$ and $21 \cdot 2^\ell/\ell$ are non-decreasing for large enough ℓ. Further, $a \leqslant 2\log s'(n)$ and $b \geqslant \log s(n)$ for large enough n. Then $\ell \in [a, b]$ implies

$$\begin{aligned} 2^\ell/(10\ell) \geqslant 2^a/(10a) &\geqslant \frac{s'(n) \cdot \log s'(n) \cdot 20}{10 \cdot 2\log s'(n)} = s'(n); \\ 21 \cdot 2^\ell/\ell \leqslant 21 \cdot 2^b/b &\leqslant \frac{21 \cdot s(n) \cdot \log s(n) \cdot \frac{1}{21}}{\log s(n)} = s(n). \end{aligned}$$

Since $s(n) \geqslant 20 \cdot 21 \cdot 2 \cdot s'(n)$ we have $b \geqslant a + 1$, so there exists a natural $\ell \in [a, b]$. □

5.2.1 Kannan's theorem

Theorem 5.2.6 (Kannan 1981) *$\Sigma_2^{\mathsf{P}} \nsubseteq \mathsf{SIZE}(n^c)$ for every $c \in \mathbb{N}$.*

Proof: Let $c \in \mathbb{N}$. It suffices to show $\mathsf{PH} \nsubseteq \mathsf{SIZE}(n^c)$. Then argue for the theorem by distinguishing two cases: if $\mathsf{NP} \subseteq \mathsf{P/poly}$, then $\mathsf{PH} = \Sigma_2^{\mathsf{P}}$ by Theorem 5.1.8 and we are done; otherwise, $\Sigma_2^{\mathsf{P}} \setminus \mathsf{SIZE}(n^c) \supseteq \mathsf{NP} \setminus \mathsf{P/poly} \neq \varnothing$ and we are done.

By the size hierarchy theorem there exists, for large enough n, a size $\leqslant n^{c+1}$ circuit which is not equivalent to any circuit of size $\leqslant n^c$. Let C_n be the lexicographically minimal such circuit (with respect to some fixed encoding). Clearly,

$$Q := \{x \in \{0,1\}^* \mid C_{|x|}(x) = 1\},$$

is not in $\mathsf{SIZE}(n^c)$, so we are left to show $Q \in \mathsf{PH}$. By Proposition 4.3.10 is suffices to define a 6-alternating machine accepting Q.

Given x, existentially guess a circuit C of size $\leqslant n^{c+1}$ (with n inputs) and check $C(x) = 1$. Then verify that C is not equivalent to any size $\leqslant n^c$ circuit C': universally guess such C', existentially guess $y \in \{0,1\}^n$ and check $C(y) \neq C'(y)$. Finally, universally guess a size $\leqslant n^{c+1}$ circuit D and do the following: if D is lexicographically smaller than C, then verify that it is equivalent to some size $\leqslant n^c$ circuit D': existentially guess such D', universally guess $y \in \{0,1\}^n$ and check $D(y) = D'(y)$. $\square$

It is open whether the above holds for P or NP instead Σ_2^{P}. Theorem 4.3.4 implies

Corollary 5.2.7 *If there is $c \in \mathbb{N}$ such that $\mathsf{P} \subseteq \mathsf{SIZE}(n^c)$, then $\mathsf{P} \neq \mathsf{NP}$.*

5.3 Lower bounds for bounded depth circuits

In this section we prove:

Theorem 5.3.1 *Let $d \in \mathbb{N}$. There is $\delta > 0$ such that for all $n \in \mathbb{N}$ every circuit with unbounded fan-in and depth $\leqslant d$ that decides* PARITY *on $\{0,1\}^n$ has size $\geqslant 2^{n^\delta}$.*

PARITY	
Input:	$x \in \{0,1\}^n$.
Problem:	does x contain an odd number of 1s?

5.3.1 Decision trees

A decision tree repeats querying input bits, each query depending on the answers received so far and at some time point outputs a bit.

Definition 5.3.2 Let $\bar{X} = X_1 \cdots X_n$ be a tuple of variables. A *decision tree (with variables $\bar{X}$)* is a pair (T, ℓ) such that

- $T \subseteq \{0,1\}^*$ is a non-empty finite set of *nodes*, closed under prefixes and whenever $t \in T$ then either both $t0, t1$ or none is in T; in the latter case, t is a *leaf*.
- ℓ maps each leaf into $\{0,1\}$ and each other $t \in T$ to some variable in $\bar{X}$. We say a leaf t *outputs* $\ell(t)$ and a non-leaf t *queries* $\ell(t)$. We require that no proper prefix of a non-leaf t queries the same variable as t.

The *height* of (T, ℓ) is $\max_{t \in T} |t|$. Every $x = x_1 \cdots x_n \in \{0,1\}^n$ determines a leaf $t(x) \in T$, namely the unique leaf $t = t_1 \cdots t_{|t|} \in T$ such that for all $i \in [|t|]$, $t_i = x_j$ where $j \in [n]$ is such that $\ell(t_1 \cdots t_{i-1}) = X_j$. The decision tree *computes* the function $x \mapsto \ell(t(x))$ from $\{0,1\}^n$ into $\{0,1\}$. A decision tree is equivalent to another or a Boolean formula or a circuit C if it computes the same function as C.

Example 5.3.3 Every $f : \{0,1\}^n \to \{0,1\}$ is computed by a decision tree of height n: the tree queries all bits of its input x and outputs $f(x)$. Formally, $T := \{0,1\}^{\leqslant n}$ with $\ell(t) := X_{|t|+1}$ for $|t| < n$, and $\ell(t) := f(t)$ for leafs $t \in \{0,1\}^n$– note $t(x) = x$ for all $x \in \{0,1\}^n$.

Example 5.3.4 Every decision tree computing (the characteristic function of) PARITY on $\{0,1\}^n$ has height n.

Proof: Assume (T,ℓ) has height $< n$. For $x \in \{0,1\}^n$ choose $i \in [n]$ such that $\ell(t) \neq X_i$ for all prefixes t of $t(x)$; this exists because $|t(x)| < n$. Let x' be x with i-th bit flipped. Then $t(x) = t(x')$, so $\ell(t(x)) = \ell(t(x'))$, but exactly one of x, x' is in PARITY. □

We are interested in height as a measure of complexity. We show it is closely related to another natural measure: how many bits of an input x determine $f(x)$?

Notation: If $\alpha = \alpha(X_1,\dots,X_n)$ is a formula and A is a partial assignment write $\alpha\restriction A$ for the formula obtained by substituting the constant $A(X)$ for every $X \in dom(A)$.

Proposition 5.3.5 *Let $k \leqslant n$ and $f : \{0,1\}^n \to \{0,1\}$.*

1. *If f is computed by a decision tree of height k, then f is computed by both a k-DNF and a k-CNF.*
2. *If f is computed by both a k-DNF and a k-CNF, then f is computed y a decision tree of height $\leqslant k^2$.*

Proof: (1): Let (T,ℓ) be a decision tree of height k computing f. Let $t = t_1\cdots t_{|t|}$ range over the leafs that output 1 and let $s = s_1\cdots s_{|s|}$ range over the leafs that output 0. Then

$$\textstyle\bigvee_t \bigwedge_{i<|t|} \neg^{1-t_{i+1}}\ell(t_1\cdots t_i) \quad \text{and} \quad \bigwedge_s \bigvee_{i<|s|} \neg^{s_{i+1}}\ell(s_1\cdots s_i).$$

both compute f.

(2): Let $\alpha = \tau_1 \vee \tau_2 \vee \cdots$ and $\beta = \zeta_1 \wedge \zeta_2 \wedge \cdots$ compute f where the τ_i are k-terms (conjunctions of $\leqslant k$ literals) and the ζ_j are k-clauses (disjunctions of $\leqslant k$ literals).

Key observation: if τ_i is satisfiable and ζ_j is falsifiable, then they share a variable: otherwise there exists an assignment satisfying τ_i and falsifying ζ_j, so $\alpha \not\equiv \beta$ – contradiction.

We specify a decision tree by informally describing what variables to query or stop with an output, given answers $b_1,\dots,b_{r-1}$ to previous queries $X_{i_1},\dots,X_{i_{r-1}}$, in other words, given the partial assignment that maps X_{i_j} to b_j. Formally, this is a node $t = t_1\cdots t_{r-1}$ with $b_j = t_{j+1}, \ell(t_1\cdots t_j) = X_{i_j}$ for all $j < r$.

The tree proceeds in rounds. In each round a partial assignment A is given; in the first round $A = \varnothing$. The tree outputs 1 if some $\tau_i\restriction A$ (like e.g. $\neg 0 \wedge 1$) or all $\zeta_j\restriction A$ are tautological; it outputs 0 if all $\tau_i\restriction A$ or some $\zeta_j\restriction A$ are unsatisfiable. Otherwise it queries all variables of $\tau_{i_0}\restriction A$ where i_0 is minimal with $\tau_{i_0}\restriction A$ satisfiable.

Note that if the round does not halt with an output, $\tau_{i_0}\restriction A$ is satisfiable, so shares a variable with every falsifiable $\zeta_j\restriction A$. Thus, after $\leqslant k$ rounds, the assignment A evaluates all variables of all falsifiable ζ_j. Then the tree halts with an output. Since every round sets $\leqslant k$ variables, the height of the tree is $\leqslant k^2$. □

5.3.2 Håstad's Switching Lemma

Theorem 5.3.6 (Switching Lemma) *Let $k, m, n, h \in \mathbb{N}$ with $m < n, 0 < h$ and $\alpha = \alpha(X_1, \ldots, X_n)$ be a k-DNF. Let $\mathcal{A}_n^m$ be the set of all partial assignments A defined on m many variables from $\{X_1, \ldots, X_n\}$. Call $A \in \mathcal{A}_n^m$* bad *if $\alpha \restriction A$ is* not *equivalent to a decision tree of height $< h$. Then the number of bad A is*

$$< \left(12 \cdot k \cdot \frac{n-m}{m}\right)^h \cdot |\mathcal{A}_n^m|.$$

Proof: Write $\alpha = \tau_1 \vee \tau_2 \vee \ldots$ for k-terms τ_i. Say a partial assignment *falsifies* a term if it falsifies a literal in it. Let $A \in \mathcal{A}_n^m$. As in Proposition 5.3.5 we specify a decision tree for $\alpha \restriction A$ by informally describing what variables to query given a partial assignment collecting previous answers. Again, the tree proceeds in rounds.

Round 1 proceeds as follows, given the empty assignment. If A satisfies some τ_i then output 1; if A falsifies all τ_i, then output 0; otherwise, let i_1 be minimal such that A does not falsify τ_{i_1}; query all *critical* variables in τ_{i_1} outside the domain of A. Notation: let V_1 denote the set of critical variables and note $V_1 \neq \emptyset$; let $C_1 : V_1 \to \{0,1\}$ denote the unique partial assignment such that $A \cup C_1$ satisfies τ_{i_1}.

Round 2 has given assignment $B_1 : V_1 \to \{0,1\}$ collecting the answers obtained in round 1. It proceeds as round 1 but with A replaced by $A \cup B_1$. To wit: if $A \cup B_1$ satisfies some τ_i then output 1; if $A \cup B_1$ falsifies all τ_i, then output 0; otherwise, let i_2 be minimal such that $A \cup B_1$ does not falsify τ_{i_2}; query all *critical* variables in τ_{i_2} outside the domain of $A \cup B_1$. Notation: let V_2 denote the set of critical variables and note $V_2 \neq \emptyset$; let $C_2 : V_2 \to \{0,1\}$ denote the unique partial assignment such that $A \cup B_1 \cup C_2$ satisfies τ_{i_2}.

And so on.

Assume A is bad, so this tree has height $\geqslant h$. Choose a node of length h, say reached in round r of the tree. The idea of the proof, due to Razborov, is to use this node to define a short code of A. Then the number of short codes upper bounds the number of bad As.

According to our notation, we have indices $i_1, \ldots, i_r$, sets of critical variables $V_1, \ldots, V_r$ and assignments $B_1, \ldots, B_r$ and $C_1, \ldots, C_r$ to them. Our node of length h in round r obtained answers $B'_r \subseteq B_r$ to critical variables $V'_r \subseteq V_r$; set $C'_r := C_r \restriction V'_r$.

Since $|V_1| + \cdots + |V_{r-1}| + |V'_r| = h$ and the sets $V_1, \ldots, V_r$ and $dom(A)$ are pairwise disjoint, we get an assignment to $m + h$ variables:

$$A^* \; := \; A \cup C_1 \cup \cdots \cup C_{r-1} \cup C'_r \; \in \; \mathcal{A}_n^{m+h}.$$

We code A by A^* plus some auxiliary information. The auxiliary infomation allows to recover A from A^* as follows.

Observe i_1 is the minimal index such that A^* does not falsify τ_{i_1}. Note V_1 is a subset of the $\leqslant k$ many variables in τ_{i_1}, so is determined by k bits $v_1 \in \{0,1\}^k$. This determines C_1. Additional $|V_1|$ bits $b_1 \in \{0,1\}^{|V_1|}$ determine B_1 and thus $A_1^* := A \cup B_1 \cup C_2 \cup \cdots \cup C_{r-1} \cup C'_r$.

Observe i_2 is the minimal index such that A_1^* does not falsify τ_{i_2}. Note V_2 is a subset of the $\leqslant k$ many variables in τ_{i_2}, so is determined by k bits $v_2 \in \{0,1\}^k$. This determines C_2. Additional $|V_2|$ bits $b_2 \in \{0,1\}^{|V_2|}$ determine B_2 and $A_2^* := A \cup B_1 \cup B_2 \cup C_3 \cup \cdots \cup C'_r$.

Repeating this r times recovers $C_1, \ldots, C_{r-1}, C'_r$ and thus A. We code A by A^* and the binary strings $v_1 \cdots v_r$ and $b_1 \cdots b_r$. The first has length $kr \leqslant kh$ and exactly h many 1s, so there are $\leqslant \binom{kh}{h} \leqslant (e \cdot \frac{kh}{h})^h < (3k)^h$ many. The second has length $|V_1| + \cdots |V_{r-1}| + |V'_r| = h$, so there are 2^h many. Hence the number of bad As is $< |\mathcal{A}_n^{m+h}| \cdot (6k)^h$. But

$$\frac{|\mathcal{A}_n^{m+h}|}{|\mathcal{A}_n^m|} = \frac{\binom{n}{m+h} \cdot 2^{m+h}}{\binom{n}{m} \cdot 2^m} = 2^h \cdot \frac{n! \cdot (n-m)! \cdot m!}{(n-m-h)! \cdot (m+h)! \cdot n!} \leqslant 2^h \cdot \frac{(n-m)^h}{m^h}.$$

The lemma follows. □

5.3.3 The lower bound

Lemma 5.3.7 *Let $d \in \mathbb{N}$. There is a polynomial time algorithm that maps every depth $\leqslant d$ size s circuit with unbounded fan-in to an equivalent formula of the following form or a negation thereof:*

$$\bigwedge_{i_1 < s} \bigvee_{i_2 < s} \cdots C_{i_d < s} \lambda_{i_1 \cdots i_d}$$

where $\lambda_{i_1 \cdots i_d}$ are literals, and C is $\bigwedge$ if d is odd, and C is $\bigvee$ if d is even.

Proof: Given a circuit of depth $\leqslant d$ and size s push all $\neg$ to the bottom to get an equivalent circuit where only input gates are wired into $\neg$-gates. For an $\bigwedge$gate $Y_i = \bigwedge_{j<r} Y_{i_j}$ we can ensure all Y_{i_j} are $\neg$- or $\bigvee$-gates: if otherwise some Y_{i_j} is a conjunction, replace it by its conjuncts in the equation for Y_i (increasing r); and repeat. Similarly, we can ensure only $\neg$- and $\bigwedge$-gates are wired into $\bigvee$-gates. Deleting or adding duplicates we can ensure all arities r equal s. Then every sequence $i_1 \cdots i_d$ of numbers $< s$ determines an input gate or a negation thereof, so a literal $\lambda_{i_1 \cdots i_d}$: take the i_1-th gate Z_1 wired into the output gate, then the i_2-th gate Z_2 wired into Z_1, and so on; possibly the literal is reached by a proper prefix of $i_1 \cdots i_d$. Note the gates $Z_1, Z_2, \ldots,$ on this path alternate $\bigwedge, \bigvee$. If the output gate is $\bigwedge$ we have a formula of the displayed form, otherwise a similar formula starting with $\bigvee$ - so equivalent to the negation of a formula of the displayed form. □

Proof of Theorem 5.3.1: It suffices to show that every formula α of the form displayed in the previous lemma that computes PARITY or its complement on $\{0,1\}^n$ has $s > 2^{n^\delta}$ for $\delta < 2^{-d}$; here and in the following, we mean $\lfloor n^\epsilon \rfloor$ writing n^ϵ for $0 < \epsilon < 1$.

Assume otherwise, so $\log s \leqslant n^\delta =: k$. Note α is equivalent to

$$\neg \bigvee_{i_d < s} \neg \bigvee_{i_{d-1} < s} \cdots \neg \bigvee_{i_1 < s} \lambda_{i_1 \cdots i_d}$$

for literals $\lambda_{i_1 \cdots i_d}$. Set $m_t := n - n^{2^{-t}}$. We claim that for all large enough n and all $t \leqslant d$ there exists $A_t \in \mathcal{A}_n^{m_t}$ that *simplifies* all formulas $\beta_{i_d \cdots i_{d-t-1}} := \neg \bigvee_{i_t} \cdots \neg \bigvee_{i_1} \lambda_{i_1 \cdots i_d}$ (for all $i_d \cdots i_{d-t-1}$) in the sense that $\beta_{i_d \cdots i_{d-t-1}} \upharpoonright A_t$ has a decision tree of height $< k$.

This contradicts Example 5.3.4: both $\alpha \upharpoonright A_d$ and $\neg\alpha \upharpoonright A_d$ are computed by decision trees of height $< k = n^\delta$; but one of these formulas computes parity on $n^{2^{-d}} \geqslant n^\delta$ many variables ($\geqslant$ instead $>$ due to rounding).

We are left to prove the claim. For $t = 0$, $A_0 = \varnothing \in \mathcal{A}_n^0$ satisfies the claim. We assume to have found A_t for $t < d$ and look for A_{t+1}. Note $\beta_{i_d \cdots i_{d-t-2}} \upharpoonright A_t$ is the negation of $\bigvee_{i_{t+1}} \beta_{i_d \cdots i_{d-t-1}} \upharpoonright A_t$. By Proposition 5.3.5 (1), this is equivalent to a k-DNF. It has $n^{2^{-t}}$ many variables. Write $m' := n^{2^{-t}} - n^{2^{-t-1}}$ and $n' := n^{2^{-t}}$. By the Switching Lemma the fraction of $A \in \mathcal{A}_{n'}^{m'}$ that do not simplify this formula is

$$< (12 \cdot k \cdot (n' - m')/m')^k \leqslant (13 \cdot k \cdot n^{-2^{-t-1}})^k$$

where we estimate $(n' - m')/m' \leqslant \frac{13}{12} n^{-2^{-t-1}}$ for large enough n. For large enough n, this is $< 2^{-dk} \leqslant s^{-d}$ because $k \cdot n^{-2^{-t-1}} = n^{\delta} \cdot n^{-2^{-t-1}} \leqslant o(1)$. Thus there exists a single $A \in R_{n'}^{m'}$ that simplifies all $\beta_{i_d \cdots i_{d-t-2}} \upharpoonright A_t$ simultaneously. Set $A_{t+1} := A_t \cup A$ and note $A_{t+1} \in \mathcal{A}_n^{m_{t+1}}$ since it evaluates exactly $m_t + (n^{2^{-t}} - n^{2^{-t-1}}) = m_{t+1}$ many variables. □

Chapter 6

Randomness

6.1 How to evaluate an arithmetical circuit

Recall Definition 1.3.5 of an *arithmetical circuit*, and recall the *depth* of such a circuit is the maximum length of a path in it (viewed as a directed graph). Such a circuit $C(\bar{X})$ computes a polynomial in $\mathbb{Z}[\bar{X}]$ in the obvious sense.

Exercise 6.1.1 Let $C(X_1, \dots, X_\ell)$ be an arithmetical circuit of depth at most d with at most m multiplication gates on any path. Then C computes a polynomial of degree at most 2^m, and for all $a_1, \dots, a_\ell \in \mathbb{Z}^\ell$ ($|\cdot|$ denotes absolute value)

$$|C(a_1, \dots, a_\ell)| \leqslant \max\{2, \max_{i\in[\ell]} |a_i|\}^{d\cdot 2^m}.$$

An arithmetical circuit can compute doubly exponential values: e.g., the polynomial $X^{2^k} \in \mathbb{Z}[X]$ is computed by a circuit with $k+1$ gates. It is therefore unclear whether the *arithmetical circuit evaluation problem*

ACE	
Input:	an arithmetical circuit $C(\bar{X})$ and $\bar{a} \in \mathbb{Z}^{|\bar{X}|}$.
Problem:	is $C(\bar{a}) \neq 0$?

is in P. In fact, this is an open question. Even ACE $\in$ NP is not obvious - but here is a clever idea: we have $b := |C(\bar{a})| \leqslant u^{d\cdot 2^m}$ where $u := \max\{2, \max_{i\in[\ell]} |a_i|\}$ and d, m are as in the exercise above. Then b has at most $\log b \leqslant d2^m \log u \leqslant 2^{2n}$ many prime divisors where we assume that n, the size of the input $(C, \bar{a})$, is large enough. By the Prime Number Theorem we can assume that there are at least $2^{10n}/(20n)$ many primes below 2^{10n} and (much) less than half of them divide of b. Guess $10n$ bits determining a number $r < 2^{10n}$. Guessing uniformly at random gives a prime r that does not divide b with probability at least $1/(40n)$, in particular such r exist. Then

$$b \neq 0 \iff b \bmod r \neq 0.$$

But b mod r can be computed by evaluating the circuit C on $\bar{a}$ bottom-up doing all operations modulo r. Each operation takes time polynomial in $\log r < 10n$.

Hence, ACE $\in$ NP. Here is the crucial observation: this algorithm is better than a usual NP-algorithm in that it gives us the correct answer for a considerable fraction of its guesses, namely at least $1/(40n)$ for large enough n. This may not look impressing at first sight but it should be compared to the canonical nondeterministic algorithm for Sat, that guesses on a formula with n variables an assignment – the fraction of accepting runs is 2^{-n} if the formula has exactly one satisfying assignment.

This is the first idea. The second idea is, that we may run the algorithm several times to boost its success probability. It seems thus fair to consider ACE a 'tractable' problem albeit we don't know a polynomial time algorithm for it. At the very least, ACE seems to be 'easier' than e.g. Sat which is generally believed not to admit such 'probably correct' polynomial time algorithms. Indeed, we can rule out the existence of such algorithms under the hypothesis that PH does not collapse (Corollary 6.3.14).

6.2 Primer on probability theory

A *probability space* $(\Omega, \mathcal{A}, \Pr)$ consists of a nonempty set Ω, a σ-algebra $\mathcal{A}$ on Ω (i.e., a collection of *events* $A \subseteq \Omega$ that contains $\varnothing$ and is closed under complementation and countable intersections) and a probability measure Pr on $\mathcal{A}$. The probability of an event B *conditioned on* an event A of positive probability is $\Pr[B \mid A] := \Pr[A \cap B]/\Pr[A]$.

A *random variable* is a function $\mathbf{x}$ from Ω into a set V such that

$$\{\mathbf{x} \in U\} := \{\omega \in \Omega \mid \mathbf{x}(\omega) \in U\} \in \mathcal{A}$$

for every $U \subseteq V$. Its *distribution* is the probability measure $U \mapsto \Pr[\mathbf{x} \in U]$ on $P(V)$. If, for finite V, this maps U to $|U|/|V|$, then $\mathbf{x}$ is *uniformly distributed in* V. A *Bernoulli variable* $\mathbf{x}$ is a random variable with values in $\{0,1\}$. Note $\mathbf{x} = \chi_A$ for the event $A := \{\mathbf{x} = 1\}$. The characteristic function χ_A of an event A is the *indicator variable for* A.

Events $A_1, \ldots, A_t \in \mathcal{A}$ are *independent* if $\Pr[A_1 \cap \cdots \cap A_t] = \Pr[A_1] \cdot \cdots \cdot \Pr[A_t]$. Random variables $\mathbf{x}_1, \ldots, \mathbf{x}_t$ are *independent* if so are the events $\{\mathbf{x}_1 \in U_1\}, \ldots, \{\mathbf{x}_t \in U_t\}$ for all $U_1, \ldots, U_t \subseteq V$, and *pairwise independent* if any two of them are independent.

Assume $V \subseteq \mathbb{R}$ is countable. Then $\mathbf{x}$ has *expectation* $\mathbb{E}[\mathbf{x}] := \sum_{v \in V} v \cdot \Pr[\mathbf{x} = v]$, and *variance* $\mathrm{Var}[\mathbf{x}] := \mathbb{E}[(\mathbf{x} - \mathbb{E}[\mathbf{x}])^2] = \mathbb{E}[\mathbf{x}^2] - \mathbb{E}[\mathbf{x}]^2$.

Note $\mathbb{E}[\mathbf{x} \cdot \mathbf{y}] = \mathbb{E}[\mathbf{x}] \cdot \mathbb{E}[\mathbf{y}]$ for independent $\mathbf{x}, \mathbf{y}$ if these expectations exist.

Lemma 6.2.1 (Bienaymé) $\mathrm{Var}[\mathbf{x}_1 + \cdots + \mathbf{x}_t] = \mathrm{Var}[\mathbf{x}_1] + \cdots + \mathrm{Var}[\mathbf{x}_t]$ *for pairwise independent random variables* $\mathbf{x}_1, \ldots, \mathbf{x}_t$*, provided these variances exist.*

This can be verified by a simple calculation.

Lemma 6.2.2 *Let* $c \in \mathbb{R}_{>0}$ *and* $\mathbf{x}$ *be a nonnegative random variable whose expectation and variance exist.*

1. *(Markov) If* $\mathbb{E}[\mathbf{x}] > 0$, *then* $\Pr[\mathbf{x} \geqslant c \cdot \mathbb{E}[\mathbf{x}]] \leqslant 1/c$.
2. *(Chebychev)* $\Pr[|\mathbf{x} - \mathbb{E}[\mathbf{x}]| \geqslant c] \leqslant \mathrm{Var}[\mathbf{x}]/c^2$.

Proof: Let $f : \mathbb{R}_{\geqslant 0} \to \mathbb{R}_{\geqslant 0}$ be nondecreasing and $f(c) > 0$. Letting v range over V, we have

$$\Pr[\mathbf{x} \geqslant c] \leqslant \textstyle\sum_{v \geqslant c} \Pr[\mathbf{x} = v] \cdot f(v)/f(c) \leqslant \mathbb{E}[f \circ \mathbf{x}]/f(c).$$

Markov's inequality follows plugging $c \cdot \mathbb{E}[\mathbf{x}]$ for c and the identity for f. Chebychev's inequality follows plugging $|\mathbf{x} - \mathbb{E}[\mathbf{x}]|$ for $\mathbf{x}$ and x^2 for $f(x)$. □

Lemma 6.2.3 (Chernoff bound) *Let* $\ell \in \mathbb{N}$ *and assume* $\mathbf{x}_1, \ldots, \mathbf{x}_\ell$ *are independent Bernoulli variables, each with expectation* μ. *Then for every real* $\epsilon > 0$

$$\Pr\left[\left|\tfrac{1}{\ell}\textstyle\sum_{i=1}^{\ell} \mathbf{x}_i - \mu\right| \geqslant \epsilon\right] < 2e^{-2\epsilon^2 \cdot \ell}.$$

We omit the proof.

6.3 Randomized computations

We define randomized computations following the first idea sketched in Section 6.1. Probabilistic Turing machines are just nondeterministic ones, different however is the definition of what it means to accept an input: nondeterministic choices are interpreted to be done randomly according to the outcome of a coin toss. The output of such a Turing machine is then a random variable and we want it to take desired values with good probability. We then proceed elaborating the second idea on how to efficiently amplify the success probabilities of such algorithms.

Recall that χ_Q denotes the characteristic function of $Q \subseteq \{0,1\}^*$.

Definition 6.3.1 A *probabilistic Turing machine* is a nondeterministic Turing machine $\mathbb{A}$. Assume $\mathbb{A}$ is t-time bounded for some function $t : \mathbb{N} \to \mathbb{N}$ and recall complete runs of $\mathbb{A}$ on $x \in \{0,1\}^*$ are determined by strings $y \in \{0,1\}^{t(|x|)}$. Let $\mathbf{y}$ be uniformly distributed in $\{0,1\}^{t(|x|)}$. The random variable $\mathbb{A}(x)$ is the output of $\mathbb{A}$ on x of the run determined by $\mathbf{y}$. The random variable $t_{\mathbb{A}}(x)$ is the length of the run determined by $\mathbf{y}$.

The set $\mathsf{BPTIME}(t)$ contains the problems Q such that there exist $c \in \mathbb{N}$ and a $(c \cdot t + c)$-time bounded nondeterministic Turing machine $\mathbb{A}$ such that for all $x \in \{0,1\}^*$

$$\Pr[\mathbb{A}(x) \neq \chi_Q(x)] \leqslant 1/4.$$

The set $\mathsf{RTIME}(t)$ is similarly defined but additionally one requires that the machine $\mathbb{A}$ has *one sided error*, that is, $\Pr[\mathbb{A}(x) = 1] = 0$ for all $x \notin Q$. We set

$$\begin{array}{lcl} \mathsf{BPP} & := & \bigcup_{c \in \mathbb{N}} \mathsf{BPTIME}(n^c), \\ \mathsf{RP} & := & \bigcup_{c \in \mathbb{N}} \mathsf{RTIME}(n^c). \end{array}$$

Clearly, $\mathsf{P} \subseteq \mathsf{RP} \subseteq \mathsf{NP}$. It is unknown whether $\mathsf{BPP} \subseteq \mathsf{NP}$ or $\mathsf{NP} \subseteq \mathsf{BPP}$. It is easy to see that $\mathsf{BPP} \subseteq \mathsf{EXP}$ and we shall get much better upper bounds (Theorem 6.4.5).

Remark 6.3.2 The distribution of $\mathbb{A}(x)$ does not depend on the choice of the time bound. More precisely, if $\mathbb{A}$ is both t- and t'-time bounded then the distribution of the random variable $\mathbb{A}(x)$ is the same whether we define it using $\mathbf{y}$ uniformly distributed in $\{0,1\}^{t(|x|)}$ or in $\{0,1\}^{t'(|x|)}$. We therefore suppress the choice of t from the notation $\mathbb{A}(x)$.

Remark 6.3.3 BPP-algorithms are sometimes said to have *two-sided error*. RTIME stands for *randomized time* and BP stands for *bounded probability*: the error probability is 'bounded away' from $1/2$. Note that every problem can be decided in constant time with two-sided error $1/2$ by simply answering according to a coin toss.

Exercise 6.3.4 (Simulate a biased coin) Let $r = \sum_{i=1}^{\infty} b_i \cdot 2^{-i}$ and assume $1^n \mapsto b_n$ is polynomial time computable. Show that there exists a probabilistic polynomial time algorithm $\mathbb{A}$ such that $r - 2^{-n} \leqslant \Pr[\mathbb{A}(1^n) = 1] \leqslant r + 2^{-n}$ for all $n \in \mathbb{N}$

6.3.1 Probability amplification

The bound $1/4$ on the error probability is a somewhat arbitrary choice. Our definitions are justified by showing that BPP or RP are not very sensible to this choice.

Proposition 6.3.5 *Let Q be a problem. The following are equivalent.*

1. *$Q \in \mathsf{RP}$.*
2. *There are a polynomial time probabilistic machine $\mathbb{A}$ with one-sided error and a positive polynomial p such that $\Pr[\mathbb{A}(x) \neq \chi_Q(x)] \leqslant 1 - 1/p(|x|)$ for all $x \in \{0,1\}^*$.*
3. *For every polynomial q there is a polynomial time probabilistic machine $\mathbb{B}$ with one-sided error such that $\Pr[\mathbb{B}(x) \neq \chi_Q(x)] \leqslant 2^{-q(|x|)}$ for all $x \in \{0,1\}^*$.*

Proof: It suffices to show that (2) implies (3). Let $\mathbb{A}, p$ accord (2) and let q be given. We can assume that p is increasing. Define $\mathbb{B}$ on x to run $\mathbb{A}$ for $q(|x|) \cdot p(|x|)$ many times; $\mathbb{B}$ accepts if at least one of the simulated runs is accepting and rejects otherwise.

If $x \notin Q$, then $\mathbb{B}$ rejects for sure. If $x \in Q$, then it errs only if $\mathbb{A}$ errs on all $q(|x|) \cdot p(|x|)$ many runs. This happens with probability at most $(1 - 1/p(|x|))^{p(|x|) \cdot q(|x|)} \leqslant (1/2)^{q(|x|)}$, for x sufficiently long because $(1 - 1/p(n))^{p(n)} \to_n 1/e < 1/2$. □

Probability amplification for two-sided error rests on the Chernoff bound (Lemma 6.2.3):

Proposition 6.3.6 *Let Q be a problem. The following are equivalent.*

1. *$Q \in \mathsf{BPP}$.*
2. *There is a polynomial time probabilistic machine $\mathbb{A}$ and a positive polynomial p such that $\Pr[\mathbb{A}(x) \neq \chi_Q(x)] \leqslant 1/2 - 1/p(|x|)$ for all $x \in \{0,1\}^*$.*

3. *For every polynomial q there is a polynomial time probabilistic machine $\mathbb{B}$ such that* $\Pr[\mathbb{B}(x) \neq \chi_Q(x)] \leqslant 2^{-q(|x|)}$ *for all* $x \in \{0,1\}^*$.

Proof: It suffices to show that (2) implies (3). Let $\mathbb{A}, p$ accord (2) and let q be given. We can assume p is increasing $\mathbb{A}$ always outputs 0 or 1, so $\mathbb{A}(x)$ is Bernoulli. For a certain ℓ, define $\mathbb{B}$ on x to simulate ℓ runs of $\mathbb{A}$ on x and answer according to the majority of the answers obtained; more precisely, if $b_1, \ldots, b_\ell \in \{0,1\}$ are the outputs of the ℓ runs of $\mathbb{A}$, then $\mathbb{B}$ accepts if $\hat{\mu} := \frac{1}{\ell}\sum_{i=1}^{\ell} b_i \geqslant 1/2$ and rejects otherwise.

Intuitively, $\mathbb{B}$ computes an estimation $\hat{\mu}$ of the acceptance probability $\mu := \Pr[\mathbb{A}(x) = 1] = \mathbb{E}[\mathbb{A}(x)]$ of $\mathbb{A}$ on x. Now, $\mu \geqslant 1/2 + 1/p(|x|)$ if $x \in Q$, and $\mu \leqslant 1/2 - 1/p(|x|)$ if $x \notin Q$. Thus, $\mathbb{B}$ errs only if $|\hat{\mu} - \mu| \geqslant 1/p(|x|)$. By the Chernoff bound this has probability $\leqslant 2e^{-\ell \cdot 2/p(|x|)^2}$. This is $\leqslant 2^{-q(|x|)}$ for suitable ℓ polynomial in $|x|$. □

Exercise 6.3.7 Define $\mathsf{BPP}^{\mathsf{BPP}}$ and show that it equals BPP. Using self-reducibility, show that $\mathsf{NP} \subseteq \mathsf{BPP}$ implies $\mathsf{NP} = \mathsf{RP}$.

Remark 6.3.8 It is known that certain circuit lower bounds imply that $\mathsf{BPP} = \mathsf{P}$, so against the probably first intuition, many researchers believe that randomization does not add computational power. But this is only a conjecture, and randomness should be considered a computational resource like time or space. It is usually measured by the number of *random bits*, i.e. the number of nondeterministic (random) choices made by a randomized algorithm. A field of interest is to prove probability amplification lemmas where the new algorithms do something more clever than just repeating a given algorithm for a lot of times, more clever in the sense that they do not use much more random bits. The combinatorics needed here is provided by *expander graphs*. A second field of interest is whether randomized algorithms really need fair coins or if biased coins or otherwise corrupted random sources are good enough. This line of research led to the theory of *extractors*: roughly, these are functions that recover k truly random bits from $n >> k$ corrupted random bits that "contain" the randomness of k truly random bits.

6.3.2 Polynomial identity testing

The most famous problem in RP not known to be in P is *polynomial identity testing*, namely to decide whether two arithmetical circuits compute the same polynomial. Clearly, this tantamount to decide whether a given circuit computes a nonzero polynomial:

PIT	
Input:	an arithmetical circuit C.
Problem:	does C compute a nonzero polynomial?

We need the following algebraic lemma:

Lemma 6.3.9 (Schwarz-Zippel) *Let $N, \ell, d \in \mathbb{N}$ and $p \in \mathbb{Z}[X_1, \ldots, X_\ell]$ be nonzero of degree at most d. If $\mathbf{r}_1, \ldots, \mathbf{r}_\ell$ are independent and uniformly distributed in $\{0, \ldots, N-1\}$, then*

$$\Pr[p(\mathbf{r}_1, \ldots, \mathbf{r}_\ell) = 0] \leqslant d/N.$$

Proof: By induction on ℓ. For $\ell = 1$, p has $\leqslant d$ roots. For $\ell > 1$ write $\bar{X} = X_1 \cdots X_{\ell-1}, \bar{\mathbf{r}} = \mathbf{r}_1 \cdots \mathbf{r}_{\ell-1}$ and $p = \sum_{i \leqslant d} q_i(\bar{X}) \cdot X_\ell^i$. Let i be maximal such that $q_i(\bar{X})$ is non-zero. Clearly,

$$\Pr[p(\bar{\mathbf{r}}, \mathbf{r}_\ell) = 0] \leqslant \Pr[p(\bar{\mathbf{r}}, \mathbf{r}_\ell) = 0 \mid q_i(\bar{\mathbf{r}}) \neq 0] + \Pr[q_i(\bar{\mathbf{r}}) = 0].$$

Since q_i has degree $\leqslant d - i$, by induction $q_i(\bar{\mathbf{r}}) = 0$ has probability $\leqslant (d-i)/N$. But the other probability above is at most i/N: for every realization $\bar{r} \in \mathbb{Z}^{\ell-1}$ of $\bar{\mathbf{r}}$ with $q_i(\bar{r}) \neq 0$, $p(\bar{r}, X_\ell)$ has degree i, so $\leqslant i$ roots. □

Example 6.3.10 PIT $\in$ RP.

Proof: Given $C(X_1, \ldots, X_\ell)$ of size n and depth d, and guess $\ell+1$ strings in $\{0,1\}^{10n}$; these encode numbers $a_1, \ldots, a_\ell, r < 2^{10n}$. Compute $C(a_1, \ldots, a_\ell) \bmod r$ in time polynomial in $\log r \leqslant 10n$. Accept if the outcome is $\neq 0$ and reject otherwise.

Clearly, if C computes the zero polynomial, then the algorithm rejects for sure. Otherwise, let the random variables $\mathbf{a}_i, \mathbf{r}$ denote its guesses. Then $C(\mathbf{a}_1, \ldots, \mathbf{a}_\ell) \neq 0$ with probability $\geqslant 1 - 2^d/2^{10n} > 1/2$ by Lemma 6.3.9 and Exercise 6.1.1. For each realization $a_1 \cdots a_\ell$ of $\mathbf{a}_1 \cdots \mathbf{a}_\ell$ such that that $C(a_1, \ldots, a_\ell) \neq 0$ we have $C(a_1, \ldots, a_\ell) \bmod \mathbf{r} \neq 0$ with probability $\geqslant 1/(40n)$ as seen in Section 6.1. In total, the algorithm accepts with probability $> 1/(80n)$. Hence PIT $\in$ RP by Proposition 6.3.5. □

6.3.3 Zero error

Definition 6.3.11 Zero error polynomial time *is the class* ZPP := RP $\cap$ coRP.

Recall, honest acceptance means always outputting either a correct answer bit or □, meaning 'I don't know' (Definition 4.1.5). Paralleling Proposition 4.1.6, ZPP is characterized requiring a bound on the probability of output □. Additionally, ZPP is characterized as 'expected polynomial time' – recall the random variable $t_{\mathbb{A}}(x)$ from Definition 6.3.1.

Proposition 6.3.12 *Let Q be a problem. The following are equivalent.*

1. *$Q \in$ ZPP.*
2. *There is a positive polynomial p and a probabilistic polynomial time machine $\mathbb{A}$ that honestly accepts Q such that $\Pr[\mathbb{A}(x) = \square] \leqslant 1 - 1/p(|x|)$ for all $x \in \{0,1\}^*$.*
3. *For all polynomials q there is a probabilistic polynomial time machine $\mathbb{A}$ that honestly accepts Q such that $\Pr[\mathbb{A}(x) = \square] \leqslant 2^{-q(|x|)}$ for all $x \in \{0,1\}^*$.*
4. *There are a polynomial p and a time bounded probabilistic machine $\mathbb{B}$ such that $\Pr[\mathbb{B}(x) = \chi_Q(x)] = 1$ and $\mathbb{E}[t_{\mathbb{B}}(x)] \leqslant p(|x|)$ for all $x \in \{0,1\}^*$.*

Proof: (3) trivially implies (2) and that (2) implies (1) follows as in Proposition 6.3.5.

(1) implies (3): let q be a polynomial and apply Proposition 6.3.5 to get $\mathbb{A}_0$ and $\mathbb{A}_1$ that decide $\{0,1\} \setminus Q$ and Q with one-sided error $2^{-q(|x|)}$. A machine as in (3) runs $\mathbb{A}_0$ and

$\mathbb{A}_1$ and rejects if $\mathbb{A}_0$ accepts, accepts if $\mathbb{A}_1$ accepts and answers 'I don't know' otherwise. On every input $x \in \{0,1\}^*$, exactly one of $\mathbb{A}_0$ and $\mathbb{A}_1$ can possibly accept and does so with probability $\geqslant 1 - 2^{-q(|x|)}$. Hence, the output 'I don't know' has probability $\leqslant 2^{-q(|x|)}$.

(3) implies (4): let $\mathbb{A}$ witness (3) for $q(n) := 1$. Given x run $\mathbb{A}$ repeatedly until it outputs $\neq \square$, i.e., $\chi_Q(x)$; in parallel run a machine deciding Q. This is gives a time bounded machine $\mathbb{B}$ deciding Q, so the random variable $t_{\mathbb{B}}(x)$ is defined. The expected number of repetitions is $\leqslant \sum_{t=1}^{\infty} \cdot t \cdot 2^{-t}$. This converges by the ratio test: $(t+1)2^{-(t+1)}/(t2^{-t}) \to_t 1/2 < 1$.

(4) implies (2): let $p, \mathbb{B}$ witness (4). On $x \in \{0,1\}^*$ run $\mathbb{B}$ for at most $2p(|x|)$ steps. If it halts, answer accordingly, otherwise answer 'I don't know'. By Markov's inequality the latter happens with probability $\leqslant 1/2$. This witnesses (2) with $p(n) := 2$. □

6.3.4 Adleman's trick

Theorem 6.3.13 (Adleman 1978) $\mathsf{BPP} \subseteq \mathsf{P/poly}$.

Proof: Let $Q \in \mathsf{BPP}$. By Proposition 6.3.6 there exists a polynomial p and a p-time bounded probabilistic Turing machine $\mathbb{A}$ such that for all $x \in \{0,1\}^n$ we have $\Pr[\mathbb{A}(x) \neq \chi_Q(x)] \leqslant 2^{-(n+1)}$. This means that at most a $2^{-(n+1)}$ fraction of $y \in \{0,1\}^{p(n)}$ are *bad for* x in the sense that they determine a run of $\mathbb{A}$ on x with a wrong answer. There are at most $2^n \cdot 2^{-(n+1)} \cdot 2^{p(n)}$ many ys that are bad for some x. Hence there exists a y that is not bad for any $x \in \{0,1\}^n$. Given such a y as advice $a(n)$ on instances of length n allows to decide Q in polynomial time. Then $Q \in \mathsf{P/poly}$ by Proposition 5.1.4. □

Recalling the Karp-Lipton Theorem 5.1.8, this yields:

Corollary 6.3.14 *If* $\mathsf{NP} \subseteq \mathsf{BPP}$, *then* $\mathsf{PH} = \Sigma_2^{\mathsf{P}}$.

6.4 Hashing

We view $\{0,1\}^n$ as the n-dimensional vector space $\mathbb{F}_2^n$ over the 2 element field $\mathbb{F}_2 = \{0,1\}$. Note 0^n is the zero vector and vector addition $x + y$ equals vector subtraction $x - y$. The dot product is $\langle x_1 \cdots x_n, y_1 \cdots y_n \rangle := \sum_{i \in [n]} x_i y_i \bmod 2$. Note e.g. $\langle 110, 110 \rangle = 0$ in $\mathbb{F}_2^3$.

For $m, n > 0$ the set of $m \times n$ matrices over $\mathbb{F}_2$ is denoted $\mathbb{F}_2^{m \times n}$.

Definition 6.4.1 For positive $n, m \in \mathbb{N}$ let $H_{m,n}$ be the set of affine maps from $\mathbb{F}_2^n$ to $\mathbb{F}_2^m$, i.e., maps of the form $x \mapsto Ax + b$ where $A \in \mathbb{F}^{m \times n}$ and $b \in \mathbb{F}_2^m$.

In this section we progressively prove three lemmas about the *hash family* $H_{m,n}$, each one triggering an algorithmic application.

Remark 6.4.2 In the terminology of algorithmics, a *hash family* is a set of *hash functions* from a large set of *keys* (above $\{0,1\}^n$) to small set of *hashes* (above $\{0,1\}^m$ for $m < n$). Clearly, every hash functions has *collisions*: distinct keys with the same hash. Choosing a random hash function, any pair of distinct keys has collision probability 2^{-m}. A *2-universal* hash family achieves the same collision probability but is typically much smaller and can be efficiently sampled and evaluated. We avoid all this general terminology.

Exercise 6.4.3 Let $\mathbb{F}$ be a finite field. Let H be the set of functions $x \mapsto a \cdot x + b$ for $a, b \in \mathbb{F}$. Let $\mathbf{h}$ be uniformly distributed in H. Let $x, x', y, y' \in \mathbb{F}$ with $x \neq x'$. Show

$$\Pr[\mathbf{h}(x) = y, \mathbf{h}(x') = y'] = |\mathbb{F}|^{-2}.$$

Conclude that the random variables $\mathbf{h}(x), x \in \mathbb{F}$, are pairwise independent, uniformly distributed in $\mathbb{F}$, and have collision probability $\Pr[\mathbf{h}(x) = \mathbf{h}(x')] = |\mathbb{F}|^{-1}$.

6.4.1 Trading randomness for alternation

Our first insight about $H_{m,n}$ is that it is 2-*universal*:

Lemma 6.4.4 *Let $n, m \in \mathbb{N}$ be positive and let $\mathbf{h}$ be uniformly distributed in $H_{m,n}$. Then $\mathbf{h}(x)$ is uniformly distributed in $\mathbb{F}_2^m$ for every $x \in \mathbb{F}_2^n$. Further, for all distinct $y, z \in \mathbb{F}_2^n$*

$$\Pr[\mathbf{h}(y) = \mathbf{h}(z)] = 2^{-m}.$$

Proof: If $x \in \mathbb{F}_2^n \setminus \{0^n\}$ and $\mathbf{a}$ is uniformly distributed in $\mathbb{F}_2^n$, then $\Pr[\langle \mathbf{a}, x\rangle = 0] = 1/2$. Indeed, the set of $a \in \mathbb{F}_2^n$ with $\langle a, x\rangle = 0$ has the same size as the set of $a \in \mathbb{F}_2^n$ with $\langle a, x\rangle = 1$: if, say, the i-th bit of x is 1, then flipping the i-th bit of a defines a bijection between the sets. A uniformly distributed $\mathbf{A}$ in $\mathbb{F}_2^{m\times n}$ has independent uniformly distributed rows, so $\Pr[\mathbf{A}x = z] = (1/2)^m$ for every $z \in \mathbb{F}_2^m$ and $\mathbf{A}x$ is uniformly distributed in $\mathbb{F}_2^m$.

Note $\mathbf{h}(x) = \mathbf{A}x + \mathbf{b}$ for $\mathbf{A}$ as above and uniformly distributed $\mathbf{b}$ in $\mathbb{F}_2^m$. For $x = 0^m$, $\mathbf{h}(x) = \mathbf{b}$ is clearly uniformly distributed. For $x \neq 0^m$ and $y \in \mathbb{F}_2^m$ we have

$$\Pr[\mathbf{h}(x) = y] = \textstyle\sum_{z\in\mathbb{F}_2^m} \Pr[\mathbf{A}x = (y - z)] \cdot \Pr[\mathbf{b} = z] = \sum_{z\in\mathbb{F}_2^m} 2^{-2m} = 2^{-m},$$

where the first equality follows from the independence of $\mathbf{A}, \mathbf{b}$. The second statement follows because $\mathbf{h}(y) = \mathbf{h}(z)$ if and only if $\mathbf{h}(x) = 0^m$ for $x := y - z \neq 0^n$. □

Theorem 6.4.5 (Sipser-Gács 1983) $\mathsf{BPP} \subseteq \Sigma_2^{\mathsf{P}} \cap \Pi_2^{\mathsf{P}}$.

Proof: Let $Q \in \mathsf{BPP}$. By Proposition 6.3.6 there is a polynomial p and a p-time bounded probabilistic machine $\mathbb{A}$ such that $\Pr[\mathbb{A}(x) \neq \chi_Q(x)] \leqslant 2^{-n}$ for all $x \in \{0,1\}^n$. Let Acc_x be the set of $y \in \{0,1\}^{p(n)}$ that determine an accepting run of $\mathbb{A}$ on x. Deciding whether $x \in Q$ or not means distinguishing the cases that Acc_x is *huge* of size $\geqslant 2^{p(n)}(1 - 2^{-n})$ or *tiny* of size $\leqslant 2^{p(n)-n}$. Set $m := p(n) - n + 1$.

Assume Acc_x is tiny and let $y \in Acc_x$. By the previous lemma, $\mathbf{h}(y) \neq \mathbf{h}(z)$ for all $z \in Acc_x \setminus \{y\}$ with probability $\geqslant 1 - 2^{p(n)-n}2^{-m} = 1/2$; we say $\mathbf{h}$ *separates* y. The expected number of $y \in Acc_x$ that $\mathbf{h}$ separates is $\geqslant |Acc_x|/2$. Hence there exists a realization $h_1 \in H_{m,n}$ of $\mathbf{h}$ that separates at least half of the $y \in Acc_x$. For the rest of Acc_x we similarly find h_2 separating half of its members. Continuing for $m > \log |Acc_x|$ steps we find $h_1, \ldots, h_m \in H_{m,n}$ that *isolates* Acc_x: every $y \in Acc_x$ is separated by some h_j.

The existence of isolating $h_1, \ldots, h_m$ implies that $Acc(x)$ is not huge: every h_j is injective on the ys it separates, so there is an injection from $Acc(x)$ into m copies of $\mathbb{F}_2^m$, a set of size $m2^m < 2^{p(n)}(1 - 2^{-n})$ for large enough n.

Thus, $x \notin Q$ if and only if there exist isolating $h_1, \ldots, h_m$. This existence is straightforwardly checked by a 2-alternating polynomial time machine. By Proposition 4.3.10, $\{0,1\}^* \setminus Q \in \Sigma_2^{\mathsf{P}}$, so $Q \in \Pi_2^{\mathsf{P}}$. Thus, $\mathsf{BPP} \subseteq \Pi_2^{\mathsf{P}}$. But then also $\mathsf{BPP} = \mathsf{coBPP} \subseteq \mathsf{co}\Pi_2^{\mathsf{P}} = \Sigma_2^{\mathsf{P}}$. □

Remark 6.4.6 In this proof we could have worked with $\mathbf{A} \in \mathbb{F}_2^{m\times n}$ instead $\mathbf{h} \in H_{m,n}$.

Second proof (Lautemann): Let $Q \in \mathsf{BPP}$. Similarly as before is suffices to show $\mathsf{BPP} \subseteq \Sigma_2\mathsf{P}$. By Proposition 6.3.6 there is a polynomial p and a p-time bounded probabilistic machine $\mathbb{A}$ such that $\Pr[\mathbb{A}(x) \neq \chi_Q(x)] \leqslant 2^{-n}$ for all $x \in \{0,1\}^n$. View $\{0,1\}^{p(n)}$ as $\mathbb{F}_2^{p(n)}$ and let G_x be the set of those $y \in \mathbb{F}_2^{p(n)}$ that determine a run of $\mathbb{A}$ on x with output $\chi_Q(x)$.

For some $\ell \in \mathbb{N}$ to be determined later, let $\mathbf{y}_1, \ldots, \mathbf{y}_\ell$ be independent and uniformly distributed in $\mathbb{F}_2^{p(n)}$. For $X \subseteq \mathbb{F}_2^{p(n)}$ and $y \in \mathbb{F}_2^{p(n)}$ write $X + y := \{y' + y \mid y' \in X\}$. Since the map $y' \mapsto y' + y$ is a bijection, the random variables $\mathbf{y}_1 + y, \ldots, \mathbf{y}_\ell + y$ are also independent and uniformly distributed in $\mathbb{F}_2^{p(n)}$. Thus, letting y range over $\mathbb{F}_2^{p(n)}$, (and recall + is −)

$$\begin{array}{rcl} \Pr[\bigcup_{i=1}^{\ell} G_x + \mathbf{y}_i = \mathbb{F}_2^{p(n)}] & \geqslant & 1 - \sum_y \Pr[y \notin \bigcup_{i=1}^{\ell} G_x + \mathbf{y}_i] \\ & = & 1 - \sum_y \prod_{i=1}^{\ell} \Pr[\mathbf{y}_i + y \notin G_x] \\ & \geqslant & 1 - 2^{p(n)} \cdot (2^{-n})^{\ell}. \end{array}$$

This is positive for $\ell := p(n)$. Thus there exist $y_1, \ldots, y_\ell \in \mathbb{F}_2^{p(n)}$ such that

$$\bigcup_{i=1}^{\ell} G_x + y_i = \mathbb{F}_2^{p(n)}.$$

This is the core of the argument – roughly said: a few random shifts of the set of good runs cover all runs.

Write Acc_x for the ys that determine accepting runs of $\mathbb{A}$ on x. We claim that

$$x \in Q \quad \Longleftrightarrow \quad \exists y_1, \ldots, y_\ell \in \mathbb{F}_2^{p(n)}\ \forall y \in \mathbb{F}_2^{p(n)} : \{y + y_1, \ldots, y + y_\ell\} \cap Acc_x \neq \emptyset,$$

for all $x \in \{0,1\}^n$ provided n is large enough. Then $Q \in \Sigma_2\mathsf{P}$ and we are done.

If $x \in Q$, then $G_x = Acc_x$. As seen above, there exist $y_1, \ldots, y_\ell$ such that $\bigcup_{i=1}^{\ell} Acc_x + y_i = \mathbb{F}^{p(n)}$. In other words, for all y there is i such that $y + y_i \in Acc_x$.

If $x \notin Q$, then $|Acc_x| \leqslant 2^{-n} \cdot 2^{p(n)}$. Let $y_1, \ldots, y_\ell \in \mathbb{F}_2^{p(n)}$ be arbitrary. Note $\bigcup_{i=1}^{\ell} Acc_x + y_i$ has size $\leqslant \ell \cdot 2^{-n} \cdot 2^{p(n)} < 2^{p(n)}$ for n large enough (recall $\ell = p(n)$). Hence, there exists $y \in \{0,1\}^{p(n)} \setminus \bigcup_{i=1}^{\ell} Acc_x + y_i$, in other words, $\{y + y_1, \ldots, y + y_\ell\} \cap Acc_x = \emptyset$. □

6.4.2 Witness isolation

Our second insight about $H_{m,n}$ strengthens 2-universality:

Lemma 6.4.7 *Let $m, n \in \mathbb{N}$ be positive. For $\mathbf{h}$ uniformly distributed in $H_{m,n}$, the variables $\mathbf{h}(x), x \in \mathbb{F}_2^n$, are pairwise independent.*

Proof: Let $\mathbf{A}$ and $\mathbf{b}$ be uniformly distributed in $\mathbb{F}_2^{m\times n}$ and $\mathbb{F}_2^m$ and let $x, x' \in \mathbb{F}_2^n$ be distinct. Further, let $y, y' \in \mathbb{F}_2^m$ and let E and E' denote the events that $\mathbf{A}x + \mathbf{b} = y$ and $\mathbf{A}x' + \mathbf{b} = y'$. We have to show that E, E' are independent or, equivalently, $\Pr[E' \mid E] = \Pr[E']$.

By Lemma 6.4.4, $\Pr[E] = \Pr[E'] = 2^{-m}$. Now,

$$\Pr[E' \mid E] = \Pr[\mathbf{A}x' + (y - \mathbf{A}x) = y' \mid E] = \Pr[\mathbf{A}(x - x') = y' - y \mid E].$$

If we can omit the condition on E, this equals $2^{-m} = \Pr[E']$ since $x - x' \neq 0^n$ (see the proof of Lemma 6.4.4) and we are done. And indeed for every $F \subseteq \mathbb{F}_2^{m\times n}$

$$\Pr[\mathbf{A} \in F \mid E] = \textstyle\sum_{A\in F} \Pr[\mathbf{A} = A \mid E] = \sum_{A\in F} \Pr[E \mid \mathbf{A} = A] \cdot \frac{\Pr[\mathbf{A}=A]}{\Pr[E]} = \Pr[\mathbf{A} \in F]$$

because $\Pr[E \mid \mathbf{A} = A] = \Pr[\mathbf{b} = y - Ax] = 2^{-m} = \Pr[E]$. □

For a circuit $C = C(X_1, \dots, X_n)$ we write $C^{-1}(1) := \{x \in \{0,1\}^n \mid C(x) = 1\}$.

Theorem 6.4.8 (Valiant-Vazirani 1985) *There is a probabilistic polynomial time algorithm that given a positive $n \in \mathbb{N}$ in unary outputs a circuit C with n input variables such that for every nonempty $S \subseteq \{0,1\}^n$ with probability $\geqslant 1/(8n)$*

$$|S \cap C^{-1}(1)| = 1.$$

Proof: $\mathbb{A}$ on 1^n chooses uniformly at random some $2 \leqslant m \leqslant n+1$ and $h \in H_{m,n}$, and outputs a circuit with n input variables that accepts $x \in \{0,1\}^n$ if and only if $h(x) = 0^m$.

Let $\varnothing \neq S \subseteq \{0,1\}^n$ and $s := |S|$. With probability $\geqslant 1/n$ the algorithm chooses some m with $2^{m-2} \leqslant s \leqslant 2^{m-1}$. We are left to show that in this case its random choice $\mathbf{h} \in H_{m,n}$ maps exactly one $x \in S$ to 0^m with probability $\geqslant 1/8$.

By inclusion-exclusion and the previous lemma, $\mathbf{h}$ maps at least one $x \in S$ to 0^m with probability at least

$$\textstyle\sum_{x\in S} \Pr[\mathbf{h}(x) = 0^m] - \sum_{\substack{y,z\in S\\ y\neq z}} \Pr[\mathbf{h}(y) = \mathbf{h}(z) = 0^m] \geqslant s \cdot 2^{-m} - s^2/2 \cdot 2^{-2m}.$$

Note $s^2/2 \cdot 2^{-2m}$ upper bounds the probability that $\mathbf{h}$ maps at least 2 elements of S to 0^m. It follows that $\mathbf{h}$ maps exactly one element of S to 0^m with probability $\geqslant s \cdot 2^{-m} - s^2 \cdot 2^{-2m} = s2^{-m}(1 - s2^{-m})$. This is $\geqslant 1/8$ because $1/4 \leqslant s2^{-m} \leqslant 1/2$. □

An interesting corollary is that it is apparently hard to decide 3SAT on CNFs 'promised' to have at most one satisfying assignment. As for notation, given circuits C, C' of sizes s, s', we let $C \wedge C'$ be the straightforwardly defined circuit of size $s + s' + 1$ that computes the conjunction, i.e., the minimum of the output bits of C, C'.

Corollary 6.4.9 *Assume* NP $\nsubseteq$ RP. *Then there is no polynomial time algorithm that rejects unsatisfiable 3CNFs and accepts 3CNFs with exactly one satisfying assignment.*

Proof: Assume there is such an algorithm $\mathbb{B}$ and let r be the polynomial time reduction from CIRCUIT SAT to 3SAT from Lemma 2.3.10. Given a circuit C with n input variables, run $\mathbb{B}$ on $r(C \wedge \mathbb{A}(1^n))$ where $\mathbb{A}$ is from the previous theorem. If C is unsatisfiable, then, with probability 1, so is $r(C \wedge \mathbb{A}(1^n))$ and $\mathbb{B}$ rejects. If C is satisfiable, then, with probability $1/(8n)$, $C \wedge \mathbb{A}(1^n)$ has exactly one satisfying assignment. Inspecting the proof of Lemma 2.3.10, then also $r(C \wedge \mathbb{A}(C))$ has exactly one satisfying assignment, so $\mathbb{B}$ accepts.

By Proposition 6.3.5, CIRCUIT SAT is in RP. □

6.4.3 Approximate counting

We analyzed the probability that the random variable $|S \cap \mathbf{h}^{-1}(0^m)|$ has value 1. The following is a tail inequality for this random variable - our third insight about $H_{m,n}$:

Lemma 6.4.10 (Hashing Lemma) *Let $n, m > 0$ be natural, $\epsilon > 0$ real, $\mathbf{h}$ uniformly distributed in $H_{m,n}$ and $S \subseteq \mathbb{F}_2^n$. Set $\mu := |S|/2^m$. Then*

$$\Pr\Big[\big||S \cap \mathbf{h}^{-1}(0^m)| - \mu\big| \geqslant \epsilon\mu\Big] < 1/(\epsilon^2\mu).$$

Proof: By Lemma 6.4.7, the indicator variable $\mathbf{I}_x$ of the event $\mathbf{h}(x) = 0^m$ has expectation 2^{-m}. The variable $|S \cap \mathbf{h}^{-1}(0^m)|$ equals $\sum_{x \in S} \mathbf{I}_x$, so has expectation μ.

Its variance is the sum of $\mathrm{Var}[\mathbf{I}_x]$ by Bienaymé because the $\mathbf{I}_x$ are pairwise independent by Lemma 6.4.7. Since $\mathbf{I}_x^2 = \mathbf{I}_x$, we have $\mathrm{Var}[\mathbf{I}_x] = \mathbb{E}[\mathbf{I}_x^2] - \mathbb{E}[\mathbf{I}_x]^2 = 2^{-m} - 2^{-2m} < 2^{-m}$. Hence $|S \cap \mathbf{h}^{-1}(0^m)|$ has variance $< \mu$. Now apply Chebychev's inequality. □

Theorem 6.4.11 (Stockmeyer 1985) *There is a probabilistic polynomial time algorithm with an* NP *oracle that given a circuit C a natural $a > 0$ in unary and a natural $e > 1$ in binary with probability $> 1 - 1/e$ ouputs a rational $\hat{s}$ such that*

$$s \cdot (1 - 1/a) \leqslant \hat{s} \leqslant s \cdot (1 + 1/a),$$

where s is the number of satisfying assignments of C.

Proof: The proof proceeds in 3 steps: first we give a weak approximation, then we boost the approximation, and then the success probability.

Weak approximation: assume C has n input variables, and set $S := C^{-1}(1) \subseteq \{0,1\}^n$. Note $s = |S|$ and set $\ell := \lfloor \log s \rfloor$. The algorithm uses an NP-oracle to check whether $\ell < 4$ and in case computes $\hat{s} := s$. Otherwise, the algorithm proceeds in rounds for $m = 1, \ldots, n$. In round m it chooses uniformly at random some $h_m \in H_{m,n}$ and queries an NP-oracle whether $S \cap h_m^{-1}(0^m) = \varnothing$. If the oracle answers "yes", it halts with output $\hat{s} := 2^m$.

The crucial insight is that with high probability this algorithm halts in a round close to $m = \ell$. More precisely, with probability $> 3/4$ the algorithm halts in a round m between $\ell - 3$ and $\ell + 4$. Indeed, let $\mathbf{h}_m \in H_{m,n}$ be the random choice in round m and let $\mu_m := s/2^m$.

– For $m \leqslant \ell - 4$, the event $S \cap \mathbf{h}_m^{-1}(0^m) = \varnothing$ implies $|S \cap \mathbf{h}_m^{-1}(0^m) - \mu_m| = \mu_m$ and hence has probability $< 1/\mu_m = 2^m/s \leqslant 2^{m-\ell}$ by the Hashing Lemma (with $\epsilon := 1$); hence, the probability that the algorithm halts in any such round m is at most $\sum_{m=0}^{\ell-4} 2^{m-\ell} < 1/8$.

– The probability that the algorithm does not halt in round $m := \ell + 4$ is the probability of $S \cap \mathbf{h}_m^{-1}(0^m) \neq \varnothing$ and this is $\leqslant \mathbb{E}[|S \cap \mathbf{h}_m^{-1}(0^m)|] = s/2^m < 2^{\ell+1}/2^m = 1/8$.

Thus, with probability $> 3/4$ the output $\hat{s} = 2^m$ satisfies $s/16 \leqslant 2^{\ell-3} \leqslant \hat{s} \leqslant 2^{\ell+4} \leqslant 16s$.

Boosting approximation: for $c \in \mathbb{N}$ define C_c as the conjunction of c copies of C with disjoint input variables. Then C_c has s^c many satisfying assignments. On C_c, the weak approximation returns $s^c/16 \leqslant \hat{s} \leqslant 16s^c$ with probability $> 3/4$. Then $s/16^{1/c} \leqslant \hat{s}^{1/c} \leqslant 16^{1/c}s$. For a given a choose $c \leqslant O(a)$ so that $1/16^{1/c} \geqslant 1 - 1/(2a)$ and $16^{1/c} \leqslant 1 + 1/a$. Output a rational below $\hat{s}^{1/c}$ and above $\hat{s}^{1/c} - 1/(2a)$.

Boosting success: repeat this algorithm r times and output a median of the outputs. If this fails to produce an approximation as desired, then $\geqslant r/2$ repetitions failed while we expect $< r/4$ to fail. By Chernoff, this has probability $< 1/e$ for suitable $r \leqslant O(\log e)$. □

6.4.4 Approximate sampling

Self-reducibility shows that an NP-oracle allows to compute a satisfying assignment for a given satisfiable circuit (Theorem 2.6.2) in polynomial time. We now show that, adding randomness, we can produce an almost uniformly distributed satisfying assignment.

Definition 6.4.12 The *total variation distance* of two random variables $\mathbf{x}, \mathbf{y}$ with values in a finite set Z is

$$d(\mathbf{x}, \mathbf{y}) := \frac{1}{2} \sum_{z \in Z} \big| \Pr[\mathbf{x} = z] - \Pr[\mathbf{y} = z] \big|.$$

For $\epsilon > 0$, $\mathbf{x}$ is *ϵ-uniform in Z* if $d(\mathbf{x}, \mathbf{y}) < \epsilon$ for $\mathbf{y}$ uniformly distributed in Z.

Exercise 6.4.13 Show that

$$d(\mathbf{x}, \mathbf{y}) = \max_{U \subseteq Z} \big(\Pr[\mathbf{x} \in U] - \Pr[\mathbf{y} \in U] \big).$$

Let A be an event with positive probability, and $\overline{A}$ its complement. Let $\mathbf{x}, \mathbf{y}$ be random variables such that $\Pr[\mathbf{x} = z \mid A] = \Pr[\mathbf{y} = z]$ for all $z \in Z$. Show that $d(\mathbf{x}, \mathbf{y}) \leqslant \Pr[\overline{A}]$.

Theorem 6.4.14 (Jerrum, Valiant, Vazirani 1986) *There is a probabilistic polynomial time algorithm $\mathbb{A}$ with an* NP *oracle that given a satisfiable circuit C and a natural $e > 0$ in binary, has output $1/e$-uniform in $C^{-1}(1)$.*

Proof: Assume C has $n > 0$ input variables and $C^{-1}(1) = \{x \in \{0,1\}^n \mid C(x) = 1\} \neq \varnothing$. For $y \in \{0,1\}^{\leqslant n}$ let C_y be C with the bits of y substituted for the first $|y|$ variables of C; note $C_y(z) = C(yz)$ for all $z \in \{0,1\}^{n-|y|}$ and $C_\lambda = C$ for the empty string λ.

Our sampler computes a sample $y_1 \cdots y_n$ bit by bit for $i = 1, \ldots, n$. Having computed $y := y_1 \cdots y_{i-1}$, it computes an approximation $a(y0)$ of $C_{y0}^{-1}(1)$, and similarly $a(y1)$. It sets $y_i = 0$ with probability $p(y0) := a(y0)/a(y)$ and $y_i = 1$ with probability $p(y1) := a(y1)/a(y)$ and stops with output *none* with probability $1 - p(y0) - p(y1)$ – if any of these so-called probabilities is < 0 or > 1, the sampler outputs *fail*. Here, $a(y)$ is positive and has been computed earlier; for $i = 1$ we compute $a(\lambda)$ and output *fail* if it is not positive.

Suitable approximations $a(y)$ for y of length n are 1 or 0 depending on whether $C(y) = 1$. This ensures that the sampler outputs only strings in $C^{-1}(1)$ or *none* or *fail*.

For shorter y we compute $a(y)$ as follows. Choose $b \leqslant O(2^n e)$ and $a \leqslant O(n)$ such that

$$(|bin(e)| + 1) \cdot 2^n/b < 1/(4e) \;\text{ and }\; 1 - (1 + 1/a)^{-(3n+1)} < 1/2.$$

Let t, polynomial in $|C|$, bound the running time of the algorithm $\mathbb{A}$ of Theorem 6.4.11 on $(C_y, 1^a, bin(b))$ for all $y \in \{0,1\}^{<n}$. The sampler guesses $z \in \{0,1\}^t$ uniformly at random in the beginning and computes all required $a(y)$ as the product of $(1+1/a)^{3(n-|y|)}$ with the output of the run of $\mathbb{A}$ on $(C_y, 1^a, bin(b))$ determined by z.

If z is *good* in the sense that all these runs of $\mathbb{A}$ produce approximations as desired, then the sampler does not output *fail*. Indeed, then $a(y0) + a(y1) \leqslant a(y)$ because:

$$\begin{aligned} a(y0) + a(y1) &\leqslant (1+1/a)^{3(n-|y|-1)} \cdot (1+1/a) \cdot (|C_{y0}^{-1}(1)| + |C_{y1}^{-1}(1)|), \\ a(y) &\geqslant (1+1/a)^{3(n-|y|)} \cdot (1-1/a) \cdot |C_y^{-1}(1)|. \end{aligned}$$

Further, for good z, the sampler outputs every $y_1 \cdots y_n \in C^{-1}(1)$ with probability

$$\frac{a(y_1)}{a(\lambda)} \cdot \frac{a(y_1 y_2)}{a(y_1)} \cdot \dots \cdot \frac{1}{a(y_1 \cdots y_{n-1})} = \frac{1}{a(\lambda)}.$$

As it does not output *fail*, it outputs *none* with probability (recall the choice of a)

$$1 - \frac{|C^{-1}(1)|}{a(\lambda)} \leqslant 1 - \frac{1}{(1+1/a)^{3n+1}} < 1/2.$$

A random $\mathbf{z} \in \{0,1\}^t$ is good with probability $\geqslant 1 - 2^n/b$. Conditioned on this, the sampler outputs all $y \in C^{-1}(1)$ with the same probability and *none* with probability $< 1/2$.

Now, run this sampler $|bin(e)| + 1$ times and output *none* if all runs output *none*; otherwise output the first output distinct from *none*. The event A that all the $|bin(e)|+1$ many $\mathbf{z}$'s chosen in these runs are good has probability $\geqslant 1 - (|bin(e)|+1)2^n/b$. By choice of b this is $> 1 - 1/(4e)$. Conditioned on A, output *none* now has probability $< 1/2^{|bin(e)|+1} < 1/(2e)$ while all $y \in C^{-1}(1)$ are still output with the same probability.

Change output *none* to an element of $C^{-1}(1)$, found with the help of the NP-oracle. This element of $C^{-1}(1)$ is then oversampled by an additive factor $< 1/(2e)$, so we sample $1/(4e)$-close to uniform conditioned on A. Thus, without the condition, we sample $1/(2e)$-close to uniform by Exercise 6.4.13.

We chose the numbers to end up with $1/(2e)$ instead $1/e$ in order to address a final subtle point: our sampler tosses loaded coins that tail with some probability $q \neq 1/2$, e.g., $q = p(y1)$. This can be simulated with k fair coins if q is of the form $k'/2^k$ for $k, k' \in \mathbb{N}$. Otherwise compute such a number c with $q \leqslant c \leqslant (1+2^{-\ell})q$ for suitable $\ell \in \mathbb{N}$; then also $1 - c \leqslant (1+2^{-\ell})(1-q)$. This way, the simulation of a run with $\leqslant O(n)$ loaded coins has probability larger by a multiplicative factor $\leqslant (1+2^{-\ell})^{O(n)}$. For suitable $\ell \leqslant O(\log n + \log e)$ this is $\leqslant 1 + 1/(2e)$. Thus the sampler with fair coins is $1/e$-uniform. □

6.5 Learning and antichecker

The following implies the existence of *anticheckers* for a function requiring large circuits: a small number of input-output pairs of f that cannot be computed by any circuit of somewhat smaller size. Recall a size $\leqslant s$ circuit can be coded by $10s\lceil \log s \rceil$ bits.

Proposition 6.5.1 *For every $f : \{0,1\}^n \to \{0,1\}$ and $n, s \in \mathbb{N}$ one of the following is true:*

(a) f can be computed by a circuit of size $O(sn)$;

(b) there are $m \leqslant O(s \log s)$ many $x_1, \dots, x_m$ such that for every circuit C of size $\leqslant s$ there is $i \in [m]$ such that $C(x_i) \neq f(x_i)$.

Proof: We construct finite sequences $x_1, \dots, x_m$ of length n binary strings and $\mathcal{C}^0, \dots, \mathcal{C}^m$ of sets of circuits of size $\leqslant s$ with n input variables. $\mathcal{C}^0$ is the set of all such circuits. Given $\mathcal{C}^i$ we either construct $\mathcal{C}^{i+1}, x_{i+1}$ or stop (with $m := i$). The former happens in case $\mathcal{C}^i \neq \varnothing$ and there exists $x_{i+1} \in \{0,1\}^n$ such that $\mathcal{C}^{i+1} := \{C \in \mathcal{C}^i \mid C(x) = f(x)\}$ has size $\leqslant 3/4|\mathcal{C}^i|$.

Then $|\mathcal{C}^i| \leqslant (3/4)^i \cdot |\mathcal{C}^0| \leqslant (3/4)^i \cdot 2^{10s\lceil \log s \rceil}$. Thus, $m \leqslant O(s \log s)$. The construction stops either because $\mathcal{C}^m = \varnothing$ or because no x_{m+1} as required exists.

In the former case, $x_1, \dots, x_m$ witness (b). In the latter case, we verify (a) as follows. Choose ℓ circuits independently and uniformly at random from $\mathcal{C}^m$. We determine ℓ below. For $i \in [\ell]$ and $x \in \{0,1\}^n$, let $\mathbf{I}_i^x$ be the indicator variable for the event that the ith chosen circuit computes $f(x)$ on x. Then $\mathbf{I}_i^x$ has expectation $\mu \geqslant 3/4$. Let $\mathbf{C}$ compute the majority output of the ℓ random circuits. The event $\mathbf{C}(x) \neq f(x)$ implies $\mu - \frac{1}{\ell}\sum_{i=1}^{\ell} \mathbf{I}_i^x \geqslant 1/4$, so has probability $< 2 \cdot e^{-\ell/8}$ by Chernoff. For suitable $\ell \leqslant O(n)$ this is $< 2^{-n}$. Then the probability that $\mathbf{C}(x) \neq f(x)$ for at least one $x \in \{0,1\}^n$ is < 1. Thus, there exists a realization of $\mathbf{C}$ that computes f. It has a circuit of size $O(s \cdot \ell)$ by the next lemma. □

For $n \in \mathbb{N}$ the *majority function maj_n* maps $x_1 \cdots x_n \in \{0,1\}^n$ to 1 if $\sum_{i=1}^n x_i \geqslant n/2$, and otherwise to 0.

Lemma 6.5.2 *There is a polynomial time algorithm that maps every 1^n for $n \in \mathbb{N}$ to a size $O(n)$ circuit computing maj_n.*

Proof: It suffices to consider the case $n = 2^k$ for some k. There is a constant size circuit that maps 3 bits x, y, z to two bits u, v such that $x + y + z = 2u + v$. Compute bits u_1, v_1 and u_2, v_2 and ... and $u_{2^{k-1}}, v_{2^{k-1}}$ such that $0 + x_1 + x_2 = 2u_1 + v_1$, and $v_1 + x_3 + x_4 = 2u_2 + v_2$ and ... and $v_{2^{k-1}-1} + x_{2^k-1} + x_{2^k} = 2u_{2^{k-1}} + v_{2^{k-1}}$. This can be done in size $O(2^k)$. Then

$$\textstyle\sum_{i=1}^{2^k} x_i = 2\sum_{j=1}^{2^{k-1}} u_j + v_{2^k}.$$

Then $maj_{2^k}(x_1 \cdots x_{2^k}) = maj_{2^{k-1}}(u_1 \cdots u_{2^{k-1}})$. Thus, if s_k denotes the size of a circuit for maj_{2^k}, then $s_k \leqslant s_{k-1} + O(2^k)$. Hence $s_k \leqslant O(2^k)$, as claimed.

The polynomial time computability of such a circuit is easy to check. □

Proposition 6.5.1 is proved by the probabilistic method, so is non-constructive. We now prove a constructive variant. Applied to a problem Q with small circuits, it implies that somewhat larger circuits can be *learned* in probabilistic polynomial time with the help of a *teacher*: the teacher can answer queries to SAT, and answer whether a candidate circuit has false positives or false negatives. More formally, the latter two are oracles for the problems:

FP$_Q$
Input: a circuit C with n input variables.
Problem: is there a *false positive*, i.e., some $x \in \{0,1\}^n$ with $C(x) > \chi_Q(x)$?

FN$_Q$	
Input:	a circuit C with n input variables.
Problem:	is there a *false negative*, i.e., some $x \in \{0,1\}^n$ with $C(x) < \chi_Q(x)$?

Formally, an algorithm with three oracles $X_1, X_2, X_3 \subseteq \{0,1\}^n$ is an algorithm with oracle that contains the strings $00x$ for $x \in X_1$, $01x$ for $x \in X_1$ and $11x$ for $x \in X_3$.

Theorem 6.5.3 *Let Q be a problem. There is a probabilistic polynomial time algorithm with oracles* SAT*,* FP$_Q$*,* FN$_Q$ *that given $n, s \in \mathbb{N}$ in unary and positive $b \in \mathbb{N}$ in binary outputs* fail *with probability $< 1/b$ and otherwise outputs*

(a) a size $O(sn)$ circuit with n input variables that computes χ_Q on $\{0,1\}^n$; or,

(b) a set $A \subseteq \{0,1\}^n$ such that for every circuit C of size at most s with n input variables there is $x \in A$ such that $C(x) \neq \chi_Q(x)$.

Proof: Code circuits of size at most s with n variables by strings of length exactly $10s\lceil\log s\rceil$. For $A \subseteq \{0,1\}^n$ let $\mathcal{C}_A \subseteq \{0,1\}^{10s\lceil\log s\rceil}$ denote the set of codes of such circuits C such that $C(x) = \chi_Q(x)$ for all $x \in A$.

A high level description of the algorithm looks as follows. We determine ℓ later.

```
1. $A \leftarrow \emptyset$
2. **if** $\mathcal{C}_A = \emptyset$ **then** output $A$
3. sample $C_1, \ldots, C_\ell$ from $\mathcal{C}_A$
4. compute a circuit $C$ computing the majority of $C_1, \ldots, C_\ell$
5. **if** $C$ computes $\chi_Q$ on $\{0,1\}^n$ **then** output $C$
6. compute $x \in \{0,1\}^n$ with $C(x) \neq \chi_Q(x)$
7. $A \leftarrow A \cup \{x\}$
8. **goto** 2
```

To implement line 3 the algorithm computes not only A but also the answer bits $\chi_Q(x), x \in A$. Then it is easy to check whether a given circuit is in $\mathcal{C}_A$. The Fundamental Lemma allows to compute a circuit D with $10s\lceil\log s\rceil$ variables that accepts precisely the strings in $\mathcal{C}_A$. This takes time polynomial in $|A| \cdot n \cdot s$. Then run the sampler from Theorem 6.4.14 independently for ℓ times on $(D, \mathit{bin}(8))$.

Line 4 is implemented using Lemma 6.5.2 in time polynomial in $\ell \cdot s$ and the **if**-condition in line 5 is checked with two oracle queries, one to FP$_Q$ and one to FN$_Q$.

Line 6 is entered only if one of these two queries is answered 'yes', say FP$_Q$. In this case line 6 computes a false positive x with answer bit $\chi_Q(x) = 0$. This is done by binary search with $O(\log n)$ queries to FP$_Q$. These queries determine for various $z \in \{0,1\}^n$ whether there exists a false positive $x \leqslant_{lex} z$. Such a query is an easily constructed circuit that accepts exactly those $x \in \{0,1\}^n$ with $C(x) = 1$ and $x \leqslant_{lex} z$.

Clearly, the outputs in lines 2 and 5 accord to (b) and (a). We show that the algorithm can be stopped after polynomial time with output *fail* such that this output is unlikely.

Call a jump in line 8 *good* if $\mathcal{C}_A$ shrinks at least by a factor of 3/4. We claim a jump is good with probability $> 1 - 1/(2b)$. After $m \leqslant O(s \log s)$ good jumps, the algorithm halts in line 2. We stop the algorithm after $2m$ jumps with output *fail*. In expectation $< 2m/(2b)$ of them are not good, so m of them are not good with probability $< 1/b$ by Markov.

We are left to prove the claim. Call $x \in \{0,1\}^n$ *good (for A)* if $|\mathcal{C}_{A \cup \{x\}}| \leqslant 3|\mathcal{C}_A|/4$. Let $\mathbf{I}_i^x$ be the indicator variable of the event that the ith circuit sampled in line 3 maps x to $\chi_Q(x)$. This circuit is 1/8-uniform in $\mathcal{C}_A$. If x is not good, then $\mathbf{I}_i^x$ has expectation $\mu \geqslant 3/4 - 1/8$. Let $\mathbf{C}$ be the random circuit computed in line 4. Then $\mathbf{C}(x) \neq f(x)$ implies $\mu - 1/\ell \sum_{i=1}^{\ell} \mathbf{I}_i^x \geqslant 5/8 - 1/2$. By Chernoff, this has probability $< 2^{-n}/(2b)$ for suitable $\ell \leqslant O(n \cdot |bin(b)|)$. Thus, with probability $> 1 - 1/(2b)$ all $x \in \{0,1\}^n$ are good, and, in particular, the x computed in line 6. □

Remark 6.5.4 The learner constructed in the previous proof needs only *equivalence* queries instead $\mathrm{FP}_Q, \mathrm{FN}_Q$: a type of oracle that answers whether a given C computes χ_Q and provides a counterexample, if not. Learning theory considers still weaker oracles, prominently *membership queries* (i.e., oracle Q) and might additionally require the learner to draw its queries according to some distribution. Often the output circuit is desired to compute χ_Q only on *most* inputs (according to some distribution). Such *probably almost correct (PAC) learning* contrasts with the *exact learning* considered here.

For the following we assume that formulas are encoded in a way ensuring the following: there is a polynomial time algorithm that maps a bit $b \in \{0,1\}$ and formula α with at least one variable to a formula α_b of the same encoding length as α that is equisatisfiable to and has the same variables as the formula α with its first variable substituted by b.

Theorem 6.5.5 *For $Q = \textsc{Sat}$ there exists an algorithm as in Theorem 6.5.3 but without the oracles* $\textsc{Fp}_{\textsc{Sat}}, \textsc{Fn}_{\textsc{Sat}}$.

Proof: We show how to implement line 5 and 6 in polynomial time using an NP oracle. An NP oracle can decide whether C has a false negative. If so, it can construct such an x with n queries to the NP oracle determining whether C has a false negative extending a given string y of length $< n$.

Similarly, an NP oracle allows to decide and witness whether C is *stupid*: it accepts some $x \in \{0,1\}^n$ that does not encode a formula or encodes a false Boolean sentence.

So assume C does not have false negatives and is not stupid. We claim that C does not compute $\textsc{Sat}$ on $\{0,1\}^n$ if and only if there exists a length n formula with at least one variable such that C accepts α but rejects both α_0 and α_1.

This suffices: with an NP oracle we can decide whether there exists such an α, and in case construct one in polynomial time.

We are left to prove the claim. The direction from right to left is trivial. Conversely, assume the l.h.s.. Since C does not have false negatives, there is an unsatisfiable length n formula accepted by C. Choose such a formula α with a minimum number of variables. Since C is not stupid, this number is positive. Then C rejects both α_0 and α_1. □

6.5.1 An improvement of the Karp-Lipton theorem

It is straightforward to relativize RP to an oracle $X \subseteq \{0,1\}^*$: the class RP^X contains those $Q \subseteq \{0,1\}^*$ such that there exists a probabilistic polynomial time machine $\mathbb{A}$ with oracle X such that $\Pr[\mathbb{A}(x) = 1]$ is $\geqslant 3/4$ if $x \in Q$, and $= 0$ if $x \notin Q$. Then ZPP^X is $\mathsf{RP}^X \cap \mathsf{coRP}^X$, and it is easy to verify an analogue of Proposition 6.3.12.

The following improves Theorem 5.1.8.

Corollary 6.5.6 *If* $\mathsf{NP} \subseteq \mathsf{P/poly}$*, then* $\mathsf{PH} \subseteq \mathsf{ZPP}^{\textsc{Sat}}$.

Proof: By Theorem 4.3.7 it suffices to show that $\Sigma_t\textsc{Sat} \in \mathsf{ZPP}^{\mathsf{NP}}$ for every positive $t \in \mathbb{N}$. We give the proof for $t = 3$; the generalization to arbitrary t will be apparent.

Let a length n instance

$$\exists \bar{X}_1 \forall \bar{X}_2 \exists \bar{X}_3\ \alpha(\bar{X}_1, \bar{X}_2, \bar{X}_3)$$

with quantifier free α be given. For notational simplicity assume all $\bar{X}_i$ have length $\ell \in \mathbb{N}$. Use an encoding of formulas such that for all $x_1, x_2 \in \{0,1\}^\ell$ the formulas $\alpha(x_1, x_2, \bar{X}_3)$ are coded by strings of the same length m_1, polynomial in n.

Choose s_1 polynomial in (m_1, hence in) n, such that $\textsc{Sat}$ has a size s_1 circuit on input length m_1. Run the learner from Theorem 6.5.5 on $(1^{m_1}, 1^{s_1}, \mathit{bin}(n))$ (for $t = 3$, $\mathit{bin}(4)$ would work too). If it outputs *fail*, halt and output 'I don't know'. This happens with probability $< 1/n$. Otherwise it outputs a circuit D_1 of size polynomial in n that computes $\chi_{\textsc{Sat}}$ on $\{0,1\}^{m_1}$. By the Fundamental Lemma, construct a circuit $E_1(\bar{X}_1, \bar{X}_2)$ of size polynomial in n that computes the code of $\alpha(x_1, x_2, \bar{X}_3)$ from $x_1, x_2 \in \{0,1\}^\ell$. Let C_1 be the circuit that feeds the m_1 outputs of E_1 to the input variables of D_1.

Then our instance is a 'yes' instance if and only if

$$\exists x_1 \in \{0,1\}^\ell \neg \exists x_2 \in \{0,1\}^\ell : C_1(x_1, x_2) = 0.$$

By the Cook-Levin Theorem and the Fundamental Lemma, compute a circuit that given $x_1 \in \{0,1\}^\ell$ computes a formula β_{x_1} that is satisfiable if and only if $\exists x_2 \in \{0,1\}^\ell : C_1(x_1, x_2) = 0$. We assume all these formulas have the same length m_2, polynomial in n. As before, the learner either outputs *fail* with probability $< 1/n$, and then answer 'I don't know', or a circuit D_2 of size polynomial in n that computes $\chi_{\textsc{Sat}}$ on $\{0,1\}^{m_2}$. As before, construct a circuit $C_2(\bar{X}_1)$ of size polynomial in n that on $x_1 \in \{0,1\}^\ell$ runs D_2 on β_{x_1}.

Then our instance is a 'yes' instance if and only if

$$\exists x_1 \in \{0,1\}^\ell : C_2(x_1) = 0.$$

This can be answered using the NP oracle. This algorithm either outputs the right answer or 'I don't know'. The latter output has probability $< 2/n$. As in Proposition 6.3.12, $\Sigma_3\textsc{Sat} \in \mathsf{ZPP}^{\mathsf{NP}}$ follows. □

Exercise 6.5.7 If $\textsc{Sat} \in \mathsf{BPP}$, then $\mathsf{PH} \subseteq \mathsf{BPP}$.

6.6 Exact counting

Roughly speaking, we saw in Section 6.4.3 that approximately counting the number of satisfying assignments is at most as hard as NP. In this section we show that exact counting is at least as hard as PH.

6.6.1 Counting problems

For a polynomially bounded relation $R \subseteq (\{0,1\}^*)^2$ in P and $x \in \{0,1\}*$ write

$$R(x) := \{y \in \{0,1\}^* \mid (x,y) \in R\}.$$

We considered the *decision problem* to decide whether $R(x) = \varnothing$, and the *search problem* to compute a member of $R(x)$ if nonempty. We now consider the *counting problem* to compute $|R(x)|$; note the length of this number is polynomially bounded in $|x|$. E.g.,

#Sat	
Input:	a Boolean formula α.
Problem:	compute the number of satisfying assignments of α.

More precisely, $\#\textsc{Sat} : \{0,1\}^* \to \mathbb{N}$ maps $x \in \{0,1\}^*$ to 0 if x does not encode a Boolean formula and otherwise to the number of satisfying assignments to its variables. We define #CircSat and #3Sat similarly requiring that α is a circuit or, respectively, a 3CNF.

Definition 6.6.1 The class #P is the set of all functions $x \mapsto |R(x)|$ for some polynomially bounded $R \subseteq (\{0,1\}^*)^2$ in P. A function $f : \{0,1\}^* \to \mathbb{N}$ is #P-*complete* if it is in #P and #P-*hard*: every function $g \in \#\mathsf{P}$ is in P^f, i.e., g can be computed in polynomial time with an oracle for (the bitgraph of) f.

Proposition 6.6.2 *#CircSat, #Sat and #3Sat are #P-complete.*

Proof: Let R be a polynomially bounded relation in P. We can assume $p(|x|) = |y|$ for some polynomial p and all $(x,y) \in R$. The proof of the Cook-Levin Theorem 2.3.9 constructed from x a circuit C_x that accepts precisely the strings in $R(x)$, so #CircSat is #P-complete. Lemma 2.3.10 gives *parsimonious* reductions: these map a circuit to a Boolean formula with the same number of satisfying assignments, and a Boolean formula to a 3CNF also with the same number of satisfying assignments. This implies $\#\textsc{CircSat} \in \mathsf{P}^{\#3\textsc{Sat}}$. □

We showed in Section 2.6.1 that for NP-complete problems search reduces to decision. Counting is clearly at least as hard as decision, and can be apparently harder: let R contain (α, y) for a 3DNF α and y a satisfying assignment to its variables. The decision problem $dom(R)$ is trivial but #3SAT easily reduces to the counting problem $x \mapsto |R(x)|$.

Remark 6.6.3 A more impressing such example is to count the number of perfect matchings in a given bipartite graph, or equivalently, to compute the permanent of a given square matrix over $\mathbb{F}_2$. In 1979 Valiant showed these problems are #P-complete. In 2004 Jerrum, Sinclair and Vigoda found *fully polynomial randomized approximation schemes* for them, i.e., algorithms as in Stockmeyer's Theorem 6.4.11 but without oracle.

6.6.2 Toda's theorem

It is unknown whether the success probability $1/(8n)$ in the Valiant-Vazirani-Lemma can be improved to a constant. We shall now observe that the probability can be boosted when asking for an odd number of solutions instead of a unique one. This then allows to eliminate quantifiers by parity quantifiers – we need some notation:

Extend Boolean logic by a parity quantifier and define $\oplus$-*formulas*: declare $\oplus X\alpha$ a $\oplus$-formula whenever α is one. An assignment A *satisfies* $\oplus X\alpha$ if exactly one of $A\frac{0}{X}$ and $A\frac{1}{X}$ satisfies α; here, $A\frac{b}{X}$ is the assignment that maps X to b and agrees with A on other variables. Clearly, that an $\oplus$-formula is *equivalent* to another $\oplus$-formula or a quantified Boolean formulas or a circuit means that they are satisfied by the same assignments. For a tuple $\bar{X} = X_1 \cdots X_\ell$ we write $\oplus\bar{X}\alpha$ for $\oplus X_1 \cdots \oplus X_\ell\alpha$. A formula of this form with α a Boolean formula is a *strict* $\oplus$-formula. The following is straightforward:

Exercise 6.6.4 Let $\bar{X}$ be a tuple of pairwise distinct variables. Show that an assignment A satisfies $\oplus\bar{X}\alpha$ if and only if there is an odd number of assignments that satisfy α and agree with A on variables outside $\bar{X}$.

Lemma 6.6.5 *For every $t \in \mathbb{N}$ there is a probabilistic polynomial time algorithm that given $m \in \mathbb{N}$ in unary and a Σ_t-formula outputs with probability $\geqslant 1 - 2^{-m}$ an equivalent strict $\oplus$-formula.*

Proof: The claim is trivial for $t = 0$. For $t > 0$ we can write our input formula equivalently as $\exists\bar{Y}\neg\alpha(\bar{X}, \bar{Y})$ for α a Σ_{t-1}-formula. Inductively, we have an algorithm as desired for Σ_{t-1}-formulas. Run it on 1^{m+1} and $\alpha(\bar{X}, \bar{Y})$. With probability $\geqslant 1 - 2^{-m}/2$ it outputs a strict $\oplus$-formula $\oplus\bar{Z}\beta(\bar{X}, \bar{Y}, \bar{Z})$ equivalent to α. Assuming this we produce a formula as desired with probability $\geqslant 1 - 2^{-m}/2$ as follows.

For ℓ to be determined, run Valiant and Vazirani's algorithm (Theorem 6.4.8) on $1^{|\bar{Y}|}$ for ℓ times and get circuits $C_1(\bar{Y}), \ldots, C_\ell(\bar{Y})$. Convert each $C_i(\bar{Y})$ to a formula $\gamma_i(\bar{Y}, \bar{Y}')$ with the same number of satisfying assignments. For every $x \in \{0,1\}^{|\bar{X}|}$ we have with probability $\geqslant 1 - (1 - 1/(8|\bar{Y}|))^\ell$ that the sentence $\exists\bar{Y}\neg\alpha(x, \bar{Y})$ is equivalent to

$$\bigvee_{i=1}^{\ell} \oplus\bar{Y}\bar{Y}' \left(\gamma_i(\bar{Y}, \bar{Y}') \wedge \neg\oplus\bar{Z}\beta(x, \bar{Y}, \bar{Z})\right).$$

Choose $\ell \leqslant O(m \cdot |\bar{X}| \cdot |\bar{Y}|)$ so that this probability is $\geqslant 1 - 2^{-|\bar{X}|} \cdot 2^{-m}/2$. Then, with probability $\geqslant 1 - 2^{-m}/2$ the equivalence holds simultaneously for all $x \in \{0,1\}^{|\bar{X}|}$, so the formula $\exists \bar{Y} \neg\alpha(\bar{X}, \bar{Y})$ is equivalent to the formula above with x replaced by $\bar{X}$.

We are left to explain how to compute an equivalent strict $\oplus$-formula. For a formula $\delta(\bar{X}, \bar{Y})$ let $\#_{\bar{Y}}\delta$ be the function that maps $x \in \{0,1\}^{|\bar{X}|}$ to the number of satisfying assignments (for $\bar{Y}$) of $\delta(x, \bar{Y})$. For $\delta^{\bar{Y}+}(\bar{X}, \bar{Y}, U) := (U \wedge \delta) \vee (\neg U \wedge \bigwedge_i Y_i)$ we have $\#_{\bar{Y}U}\delta^{\bar{Y}+} = \#_{\bar{Y}}\delta + 1$. Then $\neg \oplus \bar{Y}\delta$ is equivalent to $\oplus \bar{Y} U \delta^{\bar{Y}+}$. Further note that $\delta \wedge \oplus \bar{Y}\delta'$ is equivalent to $\oplus \bar{Y}(\delta \wedge \delta')$ if the variables $\bar{Y}$ do not appear in δ. Hence, $\exists \bar{Y} \neg\alpha(\bar{X}, \bar{Y})$ is equivalent to

$$\bigvee_{i=1}^{\ell} \oplus \bar{Y}\bar{Y}'\bar{Z}U \left(\gamma_i(\bar{Y}, \bar{Y}') \wedge \beta^{\bar{Z}+}(\bar{X}, \bar{Y}, \bar{Z}, U)\right).$$

Since we know how to handle $\neg$, we are left to transform a conjunction $\bigwedge_{i=1}^{\ell} \oplus \bar{Y}_i \delta_i(\bar{X}, \bar{Y}_i)$ into a strict $\oplus$-formula. But, assuming the tuples $\bar{Y}_i$ are pairwise disjoint,

$$\oplus \bar{Y}_1 \cdots \bar{Y}_\ell \bigwedge_{i=1}^{\ell} \delta_i(\bar{X}, \bar{Y}_i)$$

is equivalent because

$$\#_{\bar{Y}_1 \cdots \bar{Y}_\ell} \bigwedge_{i=1}^{\ell} \delta_i = \prod_{i=1}^{\ell} \#_{\bar{Y}_i}\delta_i$$

and a product of natural numbers is odd if and only if all of them are odd. □

This lemma implies $\mathsf{PH} \subseteq \mathsf{BPP}^{\#\text{SAT}}$: the truth value of $\oplus \bar{X}\alpha(\bar{X})$ is the least significant bit of the value of #SAT on α. The following is a clever derandomization of this algorithm.

Theorem 6.6.6 (Toda 1991) $\mathsf{PH} \subseteq \mathsf{P}^{\#\text{SAT}}$.

Proof: The polynomial $p(x) := 3x^4 + 4x^3$ is a *modulus amplifier* in the sense that

- if $a = -1 \bmod b$, then $p(a) = -1 \bmod b^2$,
- if $a = 0 \bmod b$, then $p(a) = 0 \bmod b^2$,

for all $a, b \in \mathbb{N}$. Given a formula $\alpha = \alpha(\bar{X})$, we want a formula $p[\alpha]$ such that

$$\#p[\alpha] = p(\#\alpha),$$

where $\#\alpha$ is the number of satisfying assignments of α. Note the conjunction β of 3 copies of α with disjoint copies of its variables satisfies $\#\beta = (\#\alpha)^3$. We are left to construct sums: for $\alpha(X_1, \ldots, X_\ell), \beta(Y_1, \ldots, Y_k)$, say $k \geqslant \ell$, the formula $\gamma := (Z_0 \wedge \alpha(Z_1, \ldots Z_\ell) \wedge \bigwedge_{\ell < j \leqslant k} Z_j) \vee (\neg Z_0 \wedge \alpha'(Z_1, \ldots, Z_k))$ satisfies $\#\gamma = \#\alpha + \#\beta$.

Note $|p[\alpha]| \leqslant O(|\alpha|)$. Consider the sequence

$$\alpha_1 := p[\alpha],\ \alpha_2 := p[\alpha_1],\ \alpha_3 := p[\alpha_2], \ldots$$

Then α_ℓ has size polynomial in $2^\ell \cdot |\alpha|$ and satisfies:

- if $\#\alpha = -1 \bmod 2$, then $\#\alpha_\ell = -1 \bmod 2^{2^\ell}$,
- if $\#\alpha = 0 \bmod 2$, then $\#\alpha_\ell = 0 \bmod 2^{2^\ell}$.

Let $t \in \mathbb{N}$. We show $\Sigma_t\text{SAT} \in \mathsf{P}^{\#\text{SAT}}$. Choose an algorithm $\mathbb{A}$ according to the previous lemma. Let the polynomial $p(n)$ bound its running time on inputs $(1^2, \alpha)$ for Σ_t-sentences α of length n. On such $(1^2, \alpha)$, its output yields with probability $\geqslant 3/4$ a Boolean formula whose number of satisfying assignments $\bmod 2$ is the truth value of α. For $y \in \{0,1\}^{p(n)}$ let α^y denote this formula for the run determined by y. Consider the relation R that contains $(\alpha, \langle y, z\rangle)$ if and only if α is a Σ_t-sentence, say of length n, and $y \in \{0,1\}^{p(n)}$ and z codes a satisfying assignment of α^y_ℓ where $\ell := \lceil \log(p(n)+1)\rceil$.

Note α^y_ℓ has length polynomial in $2^\ell \cdot |\alpha^y| \leqslant O(p(n)^2)$. Hence, R is polynomially bounded and in P. Further, for α of length n,

$$|R(\alpha)| = \sum_{y \in \{0,1\}^{p(n)}} \#\alpha^y_\ell.$$

Assume first that α is true. If $y \in \{0,1\}^{p(n)}$ causes $\mathbb{A}$ to output a formula α^y as desired, it has $1 = -1 \bmod 2$ satisfying assignments. Otherwise it has $0 \bmod 2$ satisfying assignments. Hence, at least a $3/4$ fraction of the terms $\#\alpha^y_\ell$ equal $-1 \bmod 2^{2^\ell}$ and the rest $0 \bmod 2^{2^\ell}$. Hence, $-|R(\alpha)| \bmod 2^{2^\ell}$ is $\geqslant 3/4 \cdot 2^{p(n)}$.

Now assume that α is false. Then at least a $3/4$ fraction of the terms equal $0 \bmod 2^{2^\ell}$ and the rest equals $-1 \bmod 2^{2^\ell}$. Hence, $-|R(\alpha)| \bmod 2^{2^\ell}$ is $\leqslant 1/4 \cdot 2^{p(n)}$.

Knowing $|R(\alpha)|$ thus allows to decide whether α is true or false. By Proposition 6.6.2 the theorem follows. □