Introduction to theoretical computer science

Moritz Müller

April 8, 2016

Contents

Chapter 1

Turing machines

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	
Instance:	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.

In 1928 Hilbert asked for an algorithm solving the so-called *Entscheidungsproblem*:

ENTSCHEIDUNG	
Instance:	a first-order sentence φ.
Problem:	is φ valid?

Recall that a first-order sentence is valid if it is true in all models interpreting its language. An interesting variant of the problem is

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

Here, being valid in the finite means to be true in all *finite* models (models with a finite universe).

At the time these questions have been informal. To understand the questions formally one has to define what "problems" and "algorithms" are and what it means for an algorithm to "decide" some problem.

1.2 What is a problem?

Definition 1.2.1 A *(decision) problem* is a subset Q of $\{0,1\}^*$. The set $\{0,1\}$ is called the *alphabet* and its elements *bits*.

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 also write

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

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

1.2.1 Example: encoding graphs

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

CONN	
Instance:	a (finite) graph G.
Problem:	is G connected ?

The objection is usually rebutted by saying that the definition captures such problems up to some encoding. For example, say the graph $G = (V, E)$ has vertices $V = [n]$, 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 string $\ulcorner G \urcorner = x_1 x_2 \cdots x_{n^2}$ where $x_{(i-1)n+j} = a_{i,j}$. Then we can understand CONN as the following problem in the sense of Definition 1.2.1 (about binary strings):

$$\{\ulcorner G \urcorner \mid G \text{ is a connected graph}\} \subseteq \{0,1\}^*.$$

1.2.2 Example: encoding numbers and pairs

As another example, consider the *Independent Set* problem

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

Recall that for a set of vertices X to be independent means that there is no edge between any two vertices in X. In this problem, 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 $k = \sum_{i=1}^{\lceil \log(k+1) \rceil} x_i \cdot 2^{i-1}$. Then IS can be viewed as the following problem in the sense of Definition 1.2.1:

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

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 $\langle x, y \rangle$.

Exercise 1.2.2 Give an encoding of n-tuples of binary strings by strings such that the code of $(x^1, \ldots, x^n) \in (\{0,1\}^*)^n$ has length at most $c \cdot \sum_{i=1}^n |x^i|$ for some suitable constant $c \in \mathbb{N}$. How long would the code be if one were to use $\langle x^1, \langle x^2, \langle x^3, \ldots \rangle \rangle \rangle \cdots \rangle$?

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

1.2.3 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 "construction" 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)	
Instance:	$x \in \{0,1\}^*$, 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?

It is true, albeit not so trivial to see, that an algorithm "efficiently" solving IS can be used to also "efficiently" solve the construction problem mentioned above.

1.3 What is an algorithm?

1.3.1 The Church-Turing Thesis

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

1.3.2 Turing machines

We define Turing machines and computations as paths in configuration graphs. Computations can be visualized in computation tables, a concept useful later on.

Definition 1.3.1 Let $k > 0$. A *Turing machine (TM) with k work-tapes* is a pair (S, δ) where S is a finite nonempty set of *states* containing an *initial state* $s_{\text{start}} \in S$ and a *halting state* $s_{\text{halt}} \in S$, and

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

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 $\{\S, \square, 0, 1\}^k$ and $m = m_1 \cdots m_k$ in $\{-1, 0, 1\}^k$, then for all $i \in [k]$

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

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

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

We now informally describe for how a Turing machine with one work-tape computes. The work-tape is what has been called above the scratch pad, 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 two contains x_2 and so on. Cell 0 contains a special symbol § that marks the end of the tape. It is never changed nor written in some other cell (condition (a)) and if some head scans § 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)).

We proceed with the formal definition.

1.3.3 Configuration graphs

A configuration (s, j, c) reports the state of the machine s, the positions $j = j_1 \cdots j_k$ of the heads and the contents $c = c_1 \cdots c_k$ of the work-tapes, namely, the cell contents of the ith tape read $c(0), c(1), c(2), \ldots$. We define computations to be sequences of configurations generated by δ in the obvious sense defined next.

Definition 1.3.2 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 $(s, j, c) \in S \times \mathbb{N}^k \times (\{0, 1, \square, \S\}^{\mathbb{N}})^k$. Notationally, we shall 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 $i \in [k], j \in \mathbb{N}$

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

That is, written as sequences, in the start configuration c_i for $i > 1$ reads $\S\ \square\ \square \cdots$ and c_1 reads $\S\ x_1\ x_2 \cdots x_n\ \square\ \square \cdots$. We write $(s, j, c) \vdash_1 (s', j', c')$ and call (s', j', c') a *successor configuration of* (s, j, c) if 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 $\ell \neq j_i$.

The binary relation $\vdash_1$ defines a directed graph on configurations of $\mathbb{A}$ – the *configuration graph of* $\mathbb{A}$. A *run of* $\mathbb{A}$ or a *computation of* $\mathbb{A}$ is a (directed) path in this directed graph. A run of $\mathbb{A}$ is *on* x if it starts with the start configuration of $\mathbb{A}$ on x, and *complete* if it ends in a halting configuration.

Exercise 1.3.3 Define a Turing machine whose configuration graph contains for every $x \in \{0,1\}^*$ a cycle of length at least $|x|$ containing the start configuration of $\mathbb{A}$ on x.

Definition 1.3.4 Let $k \geq 1$ 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.

Exercise 1.3.5 Verify the following claims. If a Turing machine decides a problem then it accepts it. 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 problem is computably enumerable if and only if it is the range of a computable total function.

1.3.4 Robustness

As said, the Church-Turing Thesis states that reasonable formal models of computation are pairwise equivalent. This is especially easy to see and important to know for certain variants for the Turing machine model. This short paragraph is intended to present some such variants used later on and convince ourselves that the variants are equivalent. We only sketch the proofs and some of the definitions. We recommend it as an exercise for the unexperienced reader to fill in the details.

• *Input and output tapes* As a first variant we consider machines with a special 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 1.3.6 For $k > 1$, a k-tape Turing machine *with (read-only) input tape* is one that in addition to (a)–(c) of Definition 1.3.1 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 1.3.1 also satisfies $m_k \neq -1$.

Having a usual k-tape machine is simulated by a $(k+2)$-tape machine with input and output tape as follows: copy the input from the input tape 1 to tape 2 and start running the given k-tape machine on tapes 2 to $k+1$; in the end copy the content of tape $k+1$ (up to the first blank cell) to the output tape; this can be done moving the two heads on tapes $k+1$ and $k+2$ stepwise from left to right, letting the second head write what the first head reads.

• *Larger alphabets* One may allow a Turing machine to not only work with bits $\{0,1\}$ but with a larger finite alphabet Σ. We leave it to the reader to formally define such Σ-*Turing machines*. A Σ-Turing machine can be simulated by a usual one as follows: instead of storing one symbol of $\Sigma \cup \{\square\}$, store a binary code of the symbol of length $\log|\Sigma|$. When the Σ-Turing machine writes one symbol and moves one cell left, the new machine writes the corresponding $\log|\Sigma|$ symbols and then moves $2\log|\Sigma|$ symbols to the left. Note that in this way, one step of the Σ-Turing machine is simulated by constantly (i.e. input independent) many steps of a usual machine.

• *Single-tape* machines are k-tape machines with $k = 1$. With a single-tape machine we can simulate a k-tape machine as follows. The idea, say for $k = 4$, is to use a larger alphabet including $\{0,1,\hat{0},\hat{1}\}^4$. A cell contains the letter $01\hat{0}1$ if the first tape has 0 in this cell, the second 1, the third 0 and the fourth 1 and the third head is currently scanning the cell. Using a marker for the currently right-most cell visited one step of the 4-tape machine is simulated by scanning the tape from left to right up to the marker collecting (and storing by moving to an appropriate state) the information which symbols the 4 heads are reading; it then moves back the tape and carries out the changes according to the transition function of the 4-tape machine. Note that in this way the ith step of the 4-tape machine is simulated by at most $2i$ many steps of the single-tape machine.

• *Unbounded tapes* 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 $\mathbb{N}$. It is straighforward to 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.

Exercise 1.3.7 Define Turing machines with 3-dimensional tapes and simulate them by usual k-tape machines. Proceed with Turing machines that operate with more than one head per worktape. Proceed with some other fancy variant.

1.3.5 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

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

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:

$(\S, s_{\text{start}})$	1	0	$\square$	$(\S, s_{\text{start}})$	$\square$	$\square$	$\square$	$\square$
$\S$	$(1, s_r)$	0	$\square$	$(\S, s_r)$	$\square$	$\square$	$\square$	$\square$
$\S$	1	$(0, s_r)$	$\square$	$(\S, s_r)$	$\square$	$\square$	$\square$	$\square$
$\S$	1	0	$(\square, s_r)$	$(\S, s_r)$	$\square$	$\square$	$\square$	$\square$
$\S$	1	$(0, s_\ell)$	$\square$	$\S$	$(\square, s_\ell)$	$\square$	$\square$	$\square$
$\S$	$(1, s_\ell)$	0	$\square$	$\S$	0	$(\square, s_\ell)$	$\square$	$\square$
$(\S, s_\ell)$	1	0	$\square$	$\S$	0	1	$(\square, s_\ell)$	$\square$
$(\S, s_{\text{halt}})$	1	0	$\square$	$\S$	0	1	$(\square, s_{\text{halt}})$	$\square$

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

Definition 1.3.8 Let $\mathbb{A} = (S, \delta)$ be a single-tape Turing machine, $x \in \{0,1\}^*$ and $t \geq 1$. The *computation table of* $\mathbb{A}$ *on* x *up to step* t is the following matrix $(T_{ij})_{ij}$ over the alphabet $\{0, 1, \square, \S\} \cup (\{0, 1, \square, \S\} \times S)$ with $i \in [t]$ and $j \leq \tilde{t} := \max\{|x|, t\} + 1$. 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 $\square$.

1.4 Hilbert's problems again

Once the intuitive notions in Hilbert's questions are formalized, the questions could be answered. After introducing their respective formal notion of computability, both Church and Turing answered Hilbert's Entscheidungsproblem in the negative:

Theorem 1.4.1 (Church, Turing 1936) ENTSCHEIDUNG *is not decidable.*

It is worthwhile to note that by Gödel's completeness theorem we have

Theorem 1.4.2 (Gödel 1928) ENTSCHEIDUNG *is computably enumerable.*

This contrasts with the following result of Trachtenbrot. It implies (intuitively) that there are no proof calculi that are sound and complete for first-order logic when one is to allow only finite models.

Theorem 1.4.3 (Trachtenbrot 1953) ENTSCHEIDUNG(fin) *is not computably enumerable.*

Building on earlier work of Julia Robinson, Martin Davis and Hilary Putnam, Hilbert's 10th problem has finally been solved by Yuri Matiyasevich:

Theorem 1.4.4 (Matiyasevich 1970) DIOPHANT *is not decidable.*

Chapter 2

Time

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*.

On the other hand, remember the *School Algorithm* – here is an example multiplying $k = 19$ with $\ell = 7$:

1	0	0	1	1	·	1	1	1
		1	0	0	1	1	0	0
			1	0	0	1	1	0
				1	0	0	1	1
	1	0	0	0	0	1	0	1

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 School Algorithm's number of steps. Algorithms that take n^{constant} steps on inputs of length n are what we are going to call polynomial time.

For sufficiently large input size the School Algorithm is faster than the Naïve Algorithm. The difference is drastic: assume you have at your disposal a computer performing a million steps per second. Using the Naïve Algorithm to compute the product of two 20 digit numbers, it will need time roughly the age of the universe. With the School Algorithm the computer will finish roughly in a 1/200000 fraction of a microsecond. Thus, when badly programmed, even a modern supercomputer is easily outrun by any school boy and is so already on numbers of moderate size. In short, whether or not a problem is "feasible" or "practically solvable" in the real world 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 the early 60s.

Ahead of their time, both Gödel and von Neumann troubled on questions that are foundational, even definitorial for complexity theory but predating its birth.

> *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 couldnt 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 1.4.1 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

Instance:	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. In his opinion

> *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

In modern terminology we may interpret Gödel's question as to whether the mentioned problem is solvable in polynomial time. It is known that an affirmative answer would imply P = NP (see the next section), for Gödel something "quite within the realm of possibility".

2.2 Time bounded computations

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 2.2.1 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 $t(|x|)$. By TIME(t)

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

we denote 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

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

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

Exercise 2.2.2 TIME(1) contains all finite problems (write 1 for the function constantly 1).

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

- $f \leq O(g)$ if there are $c, n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ we have $f(n) \leq c \cdot g(n)$,
- $f \geq \Omega(g)$ if $g \leq O(f)$,
- $f = \Theta(g)$ if $f \leq O(g)$ and $g \leq O(f)$,
- $f \leq o(g)$ if for all $c \in \mathbb{N}$ there is $n_0 \in \mathbb{N}$ such that for all $n \geq n_0 : c \cdot f(n) \leq g(n)$,
- $f \geq \omega(g)$ if and only if $g \leq o(f)$.

Exercise 2.2.4 (a) $f \leq o(g)$ if and only if $\limsup_n \frac{f(n)}{g(n)} = 0$.

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

Exercise 2.2.5 Let p be a polynomial with natural coefficients and degree d. Then $p = \Theta(n^d)$ and $p \leq o(2^{(\log n)^2})$.

Exercise 2.2.6 Define $f_0, f_1, \ldots$ such that $n^c \leq O(f_0)$ for all $c \in \mathbb{N}$ and $f_i \leq o(f_{i+1})$ and $f_i \leq o(2^n)$ for all $i \in \mathbb{N}$.

Exercise 2.2.7 Let $f : \mathbb{N} \to \mathbb{N}$ satisfy (a) $f(n+1) = f(n)+100$, (b) $f(n+1) = f(n)+n+100$, (c) $f(n+1) = f(\lfloor n/2 \rfloor) + 100$, (d) $f(n+1) = 2f(n)$, (e) $f(n+1) = 3f(\lfloor n/2 \rfloor)$. Then f is (a) $\Theta(n)$, (b) $\Theta(n^2)$, (c) $\Theta(\log n)$, (d) $\Theta(2^n)$, (e) $\Theta(n^{\log 3})$,

Robustness 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.

For example, the simulations sketched in Section 1.3 show:

Proposition 2.2.8 *Let $k > 0$, $f : \{0,1\}^* \to \{0,1\}^*$ and $t : \mathbb{N} \to \mathbb{N}$. If f can be computed by a t-time bounded k-tape Turing machine, then it can be computed by a t'-time bounded single-tape Turing machine such that $t' \leq O(t^2)$.*

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.

Example 2.2.9 The following problem is in P.

REACHABILITY	
Instance:	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** $Y \not\subseteq X$ **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"-condition in line 5. Lines 3 and 4 require at most $|E| < n$ edge checks and updates of Y. Lines 2-5 can be implemented in $O(n^2)$ time. These lines are executed once in the beginnning and then always when the algorithm (in line 5) jumps back to line 2. But this can happen 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 2.2.10 Think of some details of the implementation of the above algorithm. *Hint:* E.g. to check $Y \not\subseteq X$ one may use two worktapes, the first containing the string encoding Y and the second the one encoding 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. Thus the check can be carried out in $|V|$ many steps. How about the "if" condition in line 4?

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 1.3.8.

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

Theorem 2.3.1 (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 a $1^t = 11 \cdots 1$ 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 can be encoding by a binary string of length $O(\log s \cdot \max\{t, |x|\})$. Guven 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 2.3.2 A function $t : \mathbb{N} \to \mathbb{N}$ is *time-constructible* if and only if $t(n) \geq n$ for all $n \in \mathbb{N}$ and he function $x \mapsto bin(t(|x|))$ is computable in time $O(t(|x|))$.

Exercise 2.3.3 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. Show that the functions $\sqrt{n}, 2^{\log^2 n}$ are time-constructible.

Theorem 2.3.4 (Time hierarchy) $\mathrm{TIME}(t^6) \setminus \mathrm{TIME}(t) \neq \emptyset$ *for time-constructible* t.

Proof: Consider the problem

Q_t	
Instance:	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 $Q_t \in \mathrm{TIME}(t^6)$, 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 $O((|\ulcorner \mathbb{A} \urcorner| + s)^2) \leq O(t(|\ulcorner \mathbb{A} \urcorner|)^6)$ many steps (recall $t(n) \geq n$) using the universal machine.

We show that $Q_t \notin \mathrm{TIME}(t)$. Assume otherwise. Then by Proposition 2.2.8 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 without loss of generality that (c) $|\ulcorner \mathbb{B} \urcorner| > c$ – if this is not the case, add some dummy states to $\mathbb{B}$ that are never reached.

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

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

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

Remark 2.3.5 Something stronger is known: $\mathrm{TIME}(t') \setminus \mathrm{TIME}(t) \neq \emptyset$ for time-constructible t, t' with $t(n) \cdot \log t(n) \leq o(t'(n))$.

Corollary 2.3.6 $\mathrm{P} \subsetneq \mathrm{E} \subsetneq \mathrm{EXP}$.

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

Exercise 2.3.7 Find $t_0, t_1, \ldots$ such that $\mathrm{P} \subsetneq \mathrm{TIME}(t_0) \subsetneq \mathrm{TIME}(t_1) \subsetneq \cdots \subseteq \mathrm{E}$.

2.4 Circuit families

In this section we give a characterization of P by uniform circuit families. This is a lemma of central conceptual and technical importance.

Definition 2.4.1 Let *Var* be a set of *(Boolean) variables*. A *(Boolean) circuit* C is a tuple $(V, E, \lambda, <)$ such that

- (V, E) is a finite, directed, acyclic graph; vertices are called *gates*;
- the *labeling* $\lambda : V \to \{0, 1, \wedge, \vee, \neg\} \cup \mathit{Var}$ maps gates of fan-in 2 into $\{\wedge, \vee\}$, gates of fan-in 1 to $\neg$ and all other gates have fan-in 0 and are mapped into $\{0, 1\} \cup \mathit{Var}$.
- $<$ is a linear order on V.

The *size* of C is $|C| := |V|$. For a tuple $\bar{X} = X_1 \cdots X_n$ of variables we write $C(\bar{X})$ for C to indicate that it has n variable gates whose labels in order $<$ are $X_1, \ldots, X_n$. Assume $X_1, \ldots, X_n$ are pairwise distinct and $C = C(X_1, \ldots, X_n)$ has m output gates $o_1 < \cdots < o_m$. For every $x = x_1 \cdots x_n \in \{0, 1\}^n$ there is exactly one *computation of C on x*, that is, a function $\mathit{val}_x : V \to \{0, 1\}$ such that for all $g, g', g'' \in V, i \in [n]$

- $\mathit{val}_x(g) = x_i$ if $\lambda(g) = X_i$,
- $\mathit{val}_x(g) = \lambda(g)$ if $\lambda(g) \in \{0, 1\}$,
- $\mathit{val}_x(g) = \min\{\mathit{val}_x(g'), \mathit{val}_x(g'')\}$ if $\lambda(g) = \wedge$ and $(g', g), (g'', g) \in E$,
- $\mathit{val}_x(g) = \max\{\mathit{val}_x(g'), \mathit{val}_x(g'')\}$ if $\lambda(g) = \vee$ and $(g', g), (g'', g) \in E$,
- $\mathit{val}_x(g) = 1 - \mathit{val}_x(g')$ if $\lambda(g) = \neg$ and $(g', g) \in E$.

The *output of C on x* is $C(x) := \mathit{val}_x(o_1) \cdots \mathit{val}_x(o_m) \in \{0, 1\}^m$. The circuit C is said to *compute* the function $x \mapsto C(x)$ from $\{0, 1\}^n$ to $\{0, 1\}^m$. If C has only one output, then it is called *satisfiable* if there exists $x \in \{0, 1\}^n$ such that $C(x) = 1$.

Exercise 2.4.2 Prove that there is exactly one function val_x as claimed above. *Hint:* Induction on $|C|$.

Exercise 2.4.3 For every circuit $C(X_1, \ldots, X_n)$ with one output gate, there exists a Boolean formula α in the variables $X_1, \ldots, X_n$ such that for all $x \in \{0, 1\}^n$ the assignment $X_i \mapsto x_i$ satisfies α if and only if $C(x) = 1$. If every non-input gate in C has fan-out 1, then there is such an α with $|C|$ many symbols.

Exercise 2.4.4 The following problem is in P.

CIRCUIT-EVAL	
Instance:	a string $x \in \{0,1\}^*$ and a circuit $C = C(X_1, \ldots, X_{\lvert x\rvert})$ with one output gate.
Problem:	is $C(x) = 1$?

Exercise 2.4.5 Prove that every function $f : \{0,1\}^n \to \{0,1\}$ is computed by some circuit. *Hint:* Disjunctive normal form.

Solving the above exercise following the hint produces a circuit of size $O(n2^n)$. With some more effort, one can see that size $O(2^n/n)$ suffices:

Proposition 2.4.6 *For all sufficiently large n and all $m \geq 1$, every $f : \{0,1\}^n \to \{0,1\}^m$ is computable by a circuit of size $\leq 21 \cdot m \cdot 2^n/n$.*

Proof: It suffices to prove this for $m = 1$. For every $g : \{0,1\}^m \to \{0,1\}$ with $m \leq n$ 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 $s(f) \leq 5 + s(f^0) + s(f^1)$: 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 $f^{00}, f^{01}, f^{10}, f^{11}$. Observe that e.g. f^{00}, f^{01} could be equal and then we need only one circuit for them. 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, for every $k \leq n$

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

where g ranges over $\{f^y \mid y \in \{0,1\}^k\}$. For $k := n$ every $s(g)$ is 1, so f is computed by a circuit of size $\leq 6 \cdot 2^n$. To get the bound claimed, set $k := n - \lceil \log n \rceil + 2$ which is $< n$ 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 $\leq 6 \cdot 2^{\lceil \log n \rceil - 2} \leq 6 \cdot 2^{\log n + 1 - 2} = 3n$. Thus

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

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

Lemma 2.4.7 (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 the circuit $C_n = C_n(X_1, \ldots, X_n)$ is computable from 1^n in polynomial time and for all $x \in \{0,1\}^n$*

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

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) $C_{\lvert x\rvert}$ and then checks whether $C_{\lvert x\rvert}(x) = 1$. Computing $C_{\lvert x\rvert}$ needs only time $p(\lvert x\rvert)$ for some polynomial p, so the encoding of $C_{\lvert x\rvert}$ has length at most $p(\lvert x\rvert)$. By Exercise 2.4.4, the check can be done in time $q(\lvert x\rvert + p(\lvert x\rvert))$ for some polynomial q (here we assume that p, q are non-decreasing).

To see the converse direction, assume $Q \in \mathrm{P}$. By Proposition 2.2.8 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 $p(n) \geq n$ 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 $(T_{ij})_{i \in [p(n)], j \leq p(n)+1}$ of $\mathbb{A}$ on x up to step $p(n)$. It has entries in $\Sigma := \{0, 1, \S, \square\} \cup (\{0, 1, \S, \square\} \times S)$.

Entry T_{ij} of the table can be determined by only looking at $T_{(i-1)(j-1)}, T_{(i-1)j}, T_{(i-1)(j+1)}$ 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 using an arbitrary injection mapping $\sigma \in \Sigma$ to a (constant) length $s := \lceil \log(|\Sigma| + 1) \rceil$ binary string $\ulcorner \sigma \urcorner$. By Exercise 2.4.5 there exists a constant size 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 $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$. 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:

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

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 $b \in \{0,1\}$; the single input gate of this circuit is labeled X_i. Further, label the first block of s inputs with $0, 1$ according $\ulcorner (\S, s_{\mathrm{start}}) \urcorner$ and all other blocks with $0, 1$ according $\ulcorner \square \urcorner$.

The linear order of the circuit is chosen such that the input node labeled X_1 is smaller than the one labeled X_2 and so on, and further so that the output nodes of $C_{(p(n)-1)0}$ are smaller than those of $C_{(p(n)-1)1}$ and so on. This gives a a circuit that computes from x the encoding of the halting configuration of $\mathbb{A}$ on x. All what is left is to add a circuit (with $s \cdot (p(n) + 2)$ inputs) that maps this to 1 or 0 depending on whether it is accepting or not. We leave it to the reader to write down such a circuit of size $O(p(n))$. $\square$

Exercise 2.4.8 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 $f \restriction \{0,1\}^n$.

Chapter 3

Nondeterminism

3.1 NP

Definition 3.1.1 A relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is *polynomially bounded* if and only if there is a polynomial such that $|y| \leq p(|x|)$ 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 3.1.2 *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 3.1.3 Show that every $Q \in$ 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 3.1.4 The problems

Circuit-Sat	
Instance:	a Boolean circuit C with one output gate.
Problem:	is C satisfiable?

Sat	
Instance:	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 1.2) 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 "pure trial and error" takes exponential time in the worst case:

Proposition 3.1.5 P $\subseteq$ NP $\subseteq$ 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 $Q \in$ 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** $|y| \leq p(|x|)$ **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 $\leq p(|x|)$. 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 P $\neq$ EXP by Corollary 2.3.6.

> *It would be interesting to know [. . .] how significantly* in general *for finitist combinatorial problems the number of steps can be reduced when compared to pure trial and error.* Kurt Gödel, 1956

3.2 Nondeterministic time

We now give a machine characterization of NP.

Definition 3.2.1 A *nondeterministic Turing machine* is a triple (S, δ_0, δ_1) such that both (S, δ_0) and (S, δ_1) are Turing machines.

Let $x \in \{0,1\}^*$. The *configuration graph of* $\mathbb{A}$ is the directed graph $(V, E_0 \cup E_1)$ where (V, E_0) and (V, E_1) are the configuration graphs of (S, δ_0) and (S, δ_1) respectively. The *start configuration of* $\mathbb{A}$ *on* x is the the start configuration of (S, δ_0) on x (which equals that of (S, δ_1)). A *run* or *computation* of $\mathbb{A}$ is a path in the configuration graph of $\mathbb{A}$; a run is *on* x if it starts with the start configuration of $\mathbb{A}$ on x, and it is *complete* if it ends in a halting configuration of $\mathbb{A}$ (which is a halting configuration either of (S, δ_0) or, 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 this holds for some polynomial t.

Observe that 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 3.2.2 Let $t \in \mathbb{N}$ and let ρ be a run of length t of $\mathbb{A}$ on x. A string $y \in \{0,1\}^*$ of length at least t is said to *determine* ρ if the ith edge of ρ is in E_{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 3.2.3 *Let Q be a problem. The following are equivalent.*

1. $Q \in$ NP.
2. *There exists a polynomial p and a nondeterministic Turing machine $\mathbb{A}$ such that*

$$Q = \{x \in \{0,1\}^* \mid \text{ there is an accepting run of } \mathbb{A} \text{ on } x \text{ of length } p(|x|)\}.$$

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 $b \in \{0,1\}$

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

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 $|y| \leq q(|x|)$ such that $(x,y) \in R$. There exists a run of $\mathbb{A}$ that in $1+2|y|+1 \leq O(q(|x|))$ steps writes y on the tape and then moves to s_{check}. After that $(x,y) \in R$ is verified in time $p(|x|+|y|) \leq p(|x|+q(|x|))$ for a suitable polynomial p.

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

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

Remark 3.2.4 The above proof implies the claim in Exercise 3.1.3.

Definition 3.2.5 Let $t : \mathbb{N} \to \mathbb{N}$. By $\mathrm{NTIME}(t)$ 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

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

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

In this notation, the equivalence of (1) and (3) in Proposition 3.2.3 reads:

Corollary 3.2.6 $\mathrm{NP} = \bigcup_{c\in\mathbb{N}} \mathrm{NTIME}(n^c)$.

Exercise 3.2.7 Generalize Proposition 3.1.5 by showing the following. If $t : \mathbb{N} \to \mathbb{N}$ is time-constructible and $t(n) \geq n$ for all $n \in \mathbb{N}$, then $\mathrm{TIME}(t) \subseteq \mathrm{NTIME}(t) \subseteq \bigcup_{c\in\mathbb{N}} \mathrm{TIME}(2^{c\cdot t})$.

Proposition 3.2.8 *If* $\mathrm{P} = \mathrm{NP}$*, then* $\mathrm{E} = \mathrm{NE}$*.*

Proof: The proof is done by so-called "padding": make the input artificially long. Assume $\mathrm{P} = \mathrm{NP}$ and let $Q \in \mathrm{NE}$. 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 $Q \in \mathrm{E}$. Consider the problem

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

The following machine witnesses that $Q' \in \mathrm{NP}$: 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 simulates $\mathbb{A}$ on x - these are at most $c \cdot 2^{c\cdot|x|} + c \leq c \cdot |\langle x, 1^{2^{|x|}}\rangle|^c + c$ many steps.

By assumption $Q' \in \mathrm{P}$. To decide Q, proceed as follows. On input x, simluulate 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 3.2.9 Prove: if $\mathrm{E} = \mathrm{NE}$, then $\mathrm{EXP} = \mathrm{NEXP}$.

3.3 Nondeterministic time hierarchy

Theorem 3.3.1 (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*

$$\mathrm{NTIME}(t') \setminus \mathrm{NTIME}(t) \neq \emptyset.$$

Proof: Let $\mathbb{A}_0, \mathbb{A}_1, \ldots$ 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 $f(i)$ andf $f(i+1)$ tries to "diagonalize" against $\mathbb{A}_i$:

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

1. $i \leftarrow \min\{j \in \mathbb{N} \mid n \leq f(j+1)\}$.
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 $L(\mathbb{D}) \notin \text{NTIME}(t)$. 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 (see Proposition 2.2.8). We can assume i_0 is sufficiently large such that $f(i_0) \geq c$ (note $f(n) \geq n$ since f is increasing). For $n > f(i_0)$, the machine $\mathbb{A}_{i_0}$ on 1^n halts within $c \cdot t^2(n) + c < nt^2(n) - t^2(n) + n \leq t^3(n)$ steps.

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

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

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

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

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 $O(n^2)$ (see the exercise below). Lines 3-5 need time $O(t(n+1)^3)$ to compute $t(n+1)^3$, time $O(i) \leq O(n)$ to compute $\ulcorner \mathbb{A}_i \urcorner$ and then time $O((i + t(n+1)^3)^2)$ for the simulation: use a universal nondeterministic machine, which is similarly defined as the one in Theorem 2.3.1. Then also 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 is "fast growing wrt t" in the sense that $2^{t(f(i)+1)^3} \cdot (i + t(f(i)+1)^3)^2) \leq O(f(i+1))$. Then $\mathbb{D}$ halts in time $O(t(n+1)^6)$ as desired. We leave it as an exercise to show there exists a function f satisfying all requirements. □

Exercise 3.3.2 Let $t : \mathbb{N} \to \mathbb{N}$ be time-constructible. Show that there exists an increasing, time-constructible $f : \mathbb{N} \to \mathbb{N}$ which is "fast growing wrt t".

Solution: Define f by $f(0) := 2$ and $f(i+1) := 2^{t(f(i)+1)^4}$. We have to show f is time-constructible. To compute $f(i+1)$ given $i+1$, first compute $k := f(i)$ and then $t(k+1)^4$. The latter needs time $c \cdot t(k+1)^4 + c$ for some $c \in \mathbb{N}$. If T_i is the time needed to compute $f(i)$ from i, then $T_{i+1} \leq d \cdot (T_i + t(f(i)+1)^4) + d$ for some $d \in \mathbb{N}$. Thus, for large enough i,

$$T_i \leq d^i T_0 + \textstyle\sum_{j \in [i]} d^{1+i-j}(t(f(j-1)+1)^4 + d) \leq d^i T_0 \cdot i \cdot (t(f(i-1)+1)^4 + d) \leq f(i).$$

Exercise 3.3.3 Let $t : \mathbb{N} \to \mathbb{N}$ be time-constructible. Show that $n \mapsto \min\{j \in \mathbb{N} \mid n \leq t(j+1)\}$ 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 bigger than n; here c is a constant such that $t(j)$ can be computed in time $c \cdot t(j)$.

Remark 3.3.4 Theorem 3.3.1 is not optimal: it suffices to assume $t(n+1) \leq o(t'(n))$.

As in Corollary 2.3.6 it is now easy to derive the following.

Corollary 3.3.5 NP $\subsetneq$ NE $\subsetneq$ NEXP.

Exercise 3.3.6 Following the proof of Theorem 2.3.4, show that

Instance:	a nondeterministic Turing machine $\mathbb{A}$.
Problem:	is it true that $\mathbb{A}$ does not accept $\ulcorner \mathbb{A} \urcorner$ in at most $t(\lvert \ulcorner \mathbb{A} \urcorner \rvert)^3$ many steps?

is not in NTIME(t). What do you answer if somebody claims that one can show as in Theorem 2.3.4 that the problem is in NTIME(t^6)?

3.4 Polynomial 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 vertices precisely if G does not. So instead of solving IS on instance (G, k) we may solve CLIQUE on $(\overline{G}, k)$:

CLIQUE	
Instance:	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 3.4.1 Let Q, Q' be problems. A function $r : \{0,1\}^* \to \{0,1\}^*$ is a *reduction* from Q to Q' if and only if

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

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

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

Example 3.4.2 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 $\leq_p$ CLIQUE and CLIQUE $\leq_p$ IS. $\square$

Definition 3.4.3 A set $\mathcal{C} \subseteq P(\{0,1\}^*)$ is *closed under* $\leq_p$ if $Q \leq_p Q' \in \mathcal{C}$ implies $Q \in \mathcal{C}$ for all problems Q, Q'. A problem Q is *$\mathcal{C}$-hard (under $\leq_p$)* if $Q' \leq_p Q$ for every $Q' \in \mathcal{C}$; if additionally $Q \in \mathcal{C}$, then Q is *$\mathcal{C}$-complete (under $\leq_p$)*.

Exercise 3.4.4 Assume $\mathcal{C} \subseteq P(\{0,1\}^*)$ is closed under $\leq_p$ and show the following. Any two $\mathcal{C}$-complete problems are polynomial time equivalent. Given two polynomial time equivalent problems, one is $\mathcal{C}$-complete if and only if so is the other. Any problem to which some $\mathcal{C}$-hard problem is polynomial time reducible, is also $\mathcal{C}$-hard.

Lemma 3.4.5 $\mathrm{P}, \mathrm{NP}, \mathrm{EXP}, \mathrm{NEXP}$ *are closed under* $\leq_p$.

Proof: We show this only for NP, the arguments for the other classes are analogous. So assume $Q \leq_p Q' \in \mathrm{NP}$ and let $\mathbb{A}'$ be a p-time bounded nondeterministic Turing machine accepting Q' where p is some non-decreasing polynomial (Corollary 3.2.6). Let r be a polynomial time reduction from Q to Q'. There is a polynomial q such that $|r(x)| \leq q(|x|)$ for all $x \in \{0,1\}^*$. Now consider the following nondeterministic machine $\mathbb{A}$: on input x, it computes $r(x)$ and simulates $\mathbb{A}'$ on $r(x)$. Computing r can be done in polynomial time and we need to simulate at most $p(q(|x|))$ many steps of $\mathbb{A}$. 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 $Q \in \mathrm{NP}$. □

Corollary 3.4.6 *Let Q be a problem.*

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

Having an NP-complete problem we can thus reformulate the question whether $\mathrm{P} = \mathrm{NP}$ into a question whether this concrete problem can be solved in polynomial time. And NP-complete problems exist! The following exercise asks you to prove this. More surprisingly, we shall see that many natural problems are NP-complete.

Exercise 3.4.7 Show that the following problem is NP-complete.

Bounded Halting	
Instance:	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 3.4.8 Show that a problem is E-hard if and only if it is EXP-hard. *Hint:* Padding.

3.5 Cook's theorem

Theorem 3.5.1 (Cook 1971) 3Sat *is* NP*-complete.*

3Sat	
Instance:	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 3.5.2 CIRCUIT-SAT $\leq_p$ 3SAT.

Proof: Given an input x, the reduction checks in polynomial time that x encodes a circuit $C = (V, E, \lambda, <)$ with exactly one output. If this is not the case, it outputs some fixed "no"-instance of 3SAT. Otherwise, the reduction computes the following formula α_C: let $out \in V$ be the output gate of C; for each gate $g \in V$ let X_g be a propositional variable; then α_C is

$$X_{out} \wedge \bigwedge_{g \in V} \begin{cases} (X_g \leftrightarrow \lambda(g)) & \text{if } g \text{ is an input gate,} \\ (X_g \leftrightarrow (X_{g'} \lambda(g) X_{g''})) & \text{if } \lambda(g) \in \{\wedge, \vee\} \text{ and } (g', g), (g'', g) \in E, g' \neq g'' \text{ ,} \\ (X_g \leftrightarrow \neg X_{g'}) & \text{if } \lambda(g) = \neg \text{ and } (g', g) \in E. \end{cases}$$

To get a 3-CNF, let $(X \leftrightarrow (Y \wedge Z))$ stand for $(\neg X \vee Y) \wedge (\neg X \vee Z) \wedge (X \vee \neg Y \vee \neg Z)$, and $(X \leftrightarrow (Y \vee Z))$ for $(\neg X \vee Y \vee Z) \wedge (\neg Y \vee X) \wedge (\neg Z \vee X)$, and $(X \leftrightarrow \neg Y)$ for $(\neg X \vee \neg Y) \wedge (Y \vee X)$.

To see this gives a reduction, write $C = C(Y_1, \ldots, Y_\ell)$. If there is $y = y_1 \cdots y_\ell \in \{0,1\}^\ell$ such that $C(y) = 1$, then the assignment mapping Y_i to y_i and X_g to $val_y(X_g)$ satisfies α_C (cf. Section 2.4). Conversely, if A is an assignment satisfying α_C, then $X_g \mapsto A(X_g)$ is a computation of C on $A(Y_1) \cdots A(Y_\ell)$; hence, $C(A(Y_1) \cdots A(Y_\ell)) = A(X_{out}) = 1$. □

Corollary 3.5.3 CIRCUIT-SAT $\leq_p$ NAESAT.

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

Proof: by inspection of the previous proof. Given a circuit C compute α_C as there. 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 of the 3-clauses of α_C. □

Proof of Theorem 3.5.1: It is clear that 3SAT is in NP (see Example 3.1.4). To show it is NP-hard, by Lemma 3.5.2 (and Exercise 3.4.4) it suffices to show that CIRCUIT-SAT is NP-hard. That is, given $Q \in$ NP, we have to show $Q \leq_p$ 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 3.2.4). 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 $C(Z_1, \ldots, Z_{2n+2+2p(n)})$ such that

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

This can be done in polynomial time by Lemma 2.4.7. Consider the circuit

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

obtained from C by relabeling Z_is to constants x_is 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 3.5.4 NAESAT, CIRCUIT-SAT, SAT *and* 3SAT *are* NP*-complete.*

Proof: Note 3SAT $\leq_p$ SAT $\leq_p$ CIRCUIT-SAT $\leq_p$ NAESAT. The first two are trivial, the third is Corollary 3.5.3. By Theorem 3.5.1 (and Exercise 3.4.4) 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 3.5.5 2SAT $\in$ P.

2SAT	
Instance:	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

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

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 2.2.9). So we are left to prove the claim.

If there is a path from λ to λ' then $\alpha \models (\lambda \to \lambda')$. 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 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 $\leq n$ stages, A is defined on all $X_1, \ldots, X_n$ and A satisfies α by (a). □

Exercise 3.5.6 The above proof shows that the following construction 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. Given a 2-CNF, understand it 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. This can be done in polynomial time. Then Γ contains the empty clause if and only if the given 2-CNF is unsatisfiable. □

3.6 NP-completeness – examples

Example 3.6.1 IS *is* NP*-complete.*

Proof: We already noted that IS $\in$ NP in Examples 3.1.4. It thus suffices to show 3SAT $\leq_p$ IS (by Theorem 3.5.1). 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 contains a triangle with nodes corresponding to the three literals. Note that any cardinality m independent set X of G 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 satifies 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 3.4.2 we know CLIQUE $\equiv_p$ IS. Recalling Exercise 3.4.4 we get:

Example 3.6.2 CLIQUE *is* NP*-complete.*

Example 3.6.3 VC *is NP-complete.*

VC	
Instance:	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 $\leq_p$ 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 3.6.4 3COL *is* NP*-complete.*

3COL	
Instance:	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 3.5.4 to show NAESAT $\leq_p$ 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 new 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. It is not hard to see that this graph is 3-colourable if and only if α is a "yes"-instance of NAESAT. □

Exercise 3.6.5 For $k \in \mathbb{N}$ define kCOL similarly as 3COL but with k instead of 3 colours. Show that kCOL is NP-complete for $k \geq 3$ 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 girl in some home?

Example 3.6.6 TRIPARTITE MATCHING *is* NP*-complete.*

TRIPARTITE MATCHING	
Instance:	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 $\lvert R_0\rvert = \lvert B\rvert$ such that for all $(b,g,h),(b',g',h') \in R_0$ 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 each 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

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

The homes correspond to the possible occurrences of X and $\neg X$ in α. 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 3.6.7 3-Set Cover *is* NP*-complete.*

3-Set Cover	
Instance:	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 $|B| \leq |G|, |H|$. 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 $|B'| = |G'| = |H'|$. Making a copy if necessary, one 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 3.6.8 DS *is* NP*-complete.*

DS	
Instance:	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 we produce the graph $G(F)$ with 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 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 $|D'| \leq |\bigcup F|/3$, then $|D'| = |\bigcup F|/3$ and the sets in D' must be pairwise disjoint. Then $F \mapsto (G(F), |\bigcup F|/3)$ 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 3.6.9 *The following are* NP*-complete.*

Knapsack	
Instance:	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 $\sum_{i \in S} v_i \geq V$ and $\sum_{i \in S} w_i \leq W$?

Subset Sum	
Instance:	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 $\ell \leq |F|$ 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 $\ell \leq |F|$). 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 $|bin(G)| \leq O(\log((|F|+1)^{n+1})) \leq O(n \cdot |F|)$. □

Proposition 3.6.10 *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 $((v_1, \ldots, v_s), (w_1, \ldots, w_s), W, V)$, it suffices to compute $val(w, i)$ for each $w \leq W$ and $i \leq s$: the maximum v such that there exists $S \subseteq [i]$ with $\sum_{i \in S} v_i = v$ and $\sum_{i \in S} w_i \leq w$. This can easily be done iteratively: $val(w, 0) = 0$ for all $w \leq W$, and $val(w, i+1)$ is the maximum of $val(w, i)$ and $v_{i+1} + val(w - w_{i+1}, i)$. □

Remark 3.6.11 Note that the previous algorithm is not polynomial time because W is exponential in the length of its encoding $bin(W)$. Assuming P ≠ 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 weight bounds 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 such explosions into account. This is an important topic but outside the scope of this lecture.

By now 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 polyomial time algorithm is equivalent to P = 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 P ≠ NP.

3.7 NP-completeness – theory

We now take a more abstract view on NP-completeness. All the 13 NP-complete problems we saw are 'fat' in the sense of Definition 3.7.1. We show that

- 'fat' NP-hard problems are 'far from' being in P in the sense that any polynomial time algorithm makes superpolynomially many errors unless P = NP (Theorem 3.7.4);
- 'fat' NP-complete problems are pairwise *p-isomorphic* (Theorem 3.7.6);

- 'thin' problems (Definition 3.7.8) are not NP-hard unless P = NP (Theorem 3.7.9);
- if P ≠ NP, then there exist problems in NP of intermediate complexity, problems that are neither in P nor NP-complete (Theorem 3.7.11)

Definition 3.7.1 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) $|pad(x,y)| \geq |x| + |y|$;

(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 and only if there is a polynomial time bounded Turing machine that rejects every input x on which f is not defined and computes $f(x)$ on all other inputs.

Examples 3.7.2 Paddable problems are empty or infinite. SAT is paddable: map e.g. (α, y) to $(\alpha \wedge (1 \vee \bigwedge y_i))$. CLIQUE is paddable: given $((G,k), 0110)$, say with $k > 3$, map it 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.

Exercise 3.7.3 Choose a problem from Section 3.6 and show it is paddable. Show that *tally* problems $Q \subseteq \{1\}^*$ are not paddable. Show that there are polynomially equivalent problems, one paddable, the other not paddable.

3.7.1 Schöningh's theorem

Assuming P ≠ NP, we show that paddable NP-hard problems are 'far' from being in P. We use the notation $\{0,1\}^{\leq n} := \bigcup_{i \leq n} \{0,1\}^i$. 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

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

Here, $\mathbb{A}(x)$ denotes the output of $\mathbb{A}$ on x, and χ_Q the characteristic function of Q.

Theorem 3.7.4 (Schöningh) *Assume* P ≠ NP. *If Q is a paddable,* NP*-hard problem, then there does not exist a polynomial time bounded (deterministic) Turing machine with polynomially bounded error on Q.*

Proof: Let Q accord the assumption and assume $\mathbb{A}$ is a polynomially time bounded with error on Q bounded by n^c for $n \geq 2$. 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 $t > \sum_{\ell < |y_t|} 2^\ell = 2^{|y_t|} - 1$, so $|y_t| \leq \log t$. Running $\mathbb{B}$ on x and this t needs time polynomial in $|x|$. Hence $Q \in P$ and thus P = NP, contradicting our assumption. □

3.7.2 Berman and Hartmanis' theorem

Definition 3.7.5 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 p-invertible and bijective.

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

This statement is reminiscent to the Schröder-Bernstein theorem from set theory stating that two sets are bijective to each other in case there are injections in both directions. In fact, our proof mimicks the following particularly elegant proof of this result:

Proposition 3.7.7 *Let X, Y be arbitrary 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. Injectivity is easy to see. We 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 3.7.6: Let Q, Q' be paddable, and choose reductions $f : Q \leq_p Q'$ and $g : Q' \leq_p Q$. 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 if f, g are length-increasing, then the length of the strings in a pre-image sequence decreases. By p-invertibility, pre-image sequence 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 so-called *Berman-Hartmanis Conjecture* states all NP-complete problems are p-isomorphic to SAT. Currently many researchers tend to believe that this fails.

3.7.3 Mahaney's theorem

Definition 3.7.8 A problem Q is *sparse* if $|Q \cap \{0,1\}^n|$ is polynomially bounded in n.

Two p-isomorphic problems are either both sparse or both not. As obviously SAT is not sparse, none of our 13 NP-complete problems is sparse. To refute the Berman-Hartmanis conjecture it would thus be sufficient to find a sparse NP-complete problem. But,

Theorem 3.7.9 (Mahaney 1980) *If* P $\neq$ NP*, then no* NP*-hard problem is sparse.*

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 $y \models \alpha$ mean that $n \leq m$ 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.

LSAT	
Instance:	a propositional formula α and $x \in \{0,1\}^*$.
Problem:	does there exist $y \leq_{lex} x$ such that $y \models \alpha$?

Assume there is a sparse NP-hard problem Q, say $|Q \cap \{0,1\}^{\leq \ell}| \leq \ell^c$ with $c \in \mathbb{N}$. Choose a reduction r from LSAT to Q. We give a polynomial time algorithm solving SAT.

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

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

of strings in $\{0,1\}^i$ such that $s_i \leq N := \max_{x \in \{0,1\}^n} |r(\alpha, x)|^c$, and

> if there exists $z \in \{0,1\}^n$ such that $z \models \alpha$, 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. We are left to explain how to compute the sequence for $i+1$ given that for $i < n$ in polynomial time. 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 $2s_i \leq N$, we are done. Otherwise we delete some of the y_js such that $(*)$ is preserved, namely each y_j such that $r(\alpha, y_{j'} 1^{n-i-1}) = r(\alpha, y_j 1^{n-i-1})$ for some $j' < j$. This gives a subsequence $z_1 <_{lex} \cdots <_{lex} z_{s'}$ with $(*)$ and pairwise distinct values $r(\alpha, z_j 1^{n-i-1}), j \in [s']$. If $s' \leq N$, we are done. Otherwise we delete all but the last N many z_js. This preserves $(*)$ because any z_j with $j < s' - N$ has value $r(\alpha, z_j 1^{n-i-1}) \notin Q$. Indeed, otherwise all later values $r(\alpha, z_{j'} 1^{n-i-1}), j' \geq j$, would be in Q too – but there are at most N of them. $\square$

Remark 3.7.10 Berman had already shown in 1978 that tally problems cannot be NP-hard – assuming that P $\neq$ NP. Under a stronger assumption, Burhmann and Hitchcock showed in 2008 that even problems Q with $|Q \cap \{0,1\}^n| \leq 2^{n^{o(1)}}$ cannot be NP-hard. The assumption $\Sigma_3^{\mathrm{P}} \neq$ PH, that we shall introduce later, is sufficient for this purpose.

3.7.4 Ladner's theorem

Theorem 3.7.11 (Ladner 1975) *Assume* NP $\neq$ 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 NP $\neq$ P, then NP $\setminus$ P contains an infinite descending $\leq_p$-chain of pairwise non-equivalent problems:

Proposition 3.7.12 *If Q is a decidable problem outside* P*, then there exists a problem Q' such that $Q' \leq_p Q$ and $Q' \not\equiv_p Q$ and $Q' \notin$* P*.*

Proof: The proof is again by "slow diagonalization" (cf. Theorem 3.3.1). 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 from n in time $O(n)$, so $Q' \leq_p Q$ follows easily. 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 define $D(0) := 0$ and $D(n)$ for $n > 0$ as the value computed by the following recursive algorithm on input n.

Compute $D(0)$ and then compute recursively the values $D(1), D(2), \ldots$ as many as you can within $\leq n$ 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'. From this we see, that D's range is an initial segment of $\mathbb{N}$ and to prove our theorem it will be sufficient to show it is $\mathbb{N}$, i.e. D is unbounded.

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** $m \leftarrow D(|x|)$ // note $|x| < n$
4. **if** $\mathbb{A}$ rejects x and $\mathbb{Q}$ accepts x and m is even **then** accept
5. **if** $\mathbb{A}$ accepts x and ($\mathbb{Q}$ rejects x or m is odd) **then** accept
6. **if** k is odd **then**
7. $y \leftarrow$ the output of the simulated run of $\mathbb{A}$ on x
8. $m \leftarrow D(|y|)$ // note $|y| < n$
9. **if** $\mathbb{Q}$ rejects x and $\mathbb{Q}$ accepts y and m is even **then** accept
10. **if** $\mathbb{Q}$ accepts x and ($\mathbb{Q}$ rejects y or m is odd) **then** accept
11. reject

Our algorithm runs WITNESS TEST on $(c, \mathbb{A}, x, k)$ 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. It is easy to see that D is non-decreasing. We show it is unbounded. Otherwise, D takes its maximal value k on all sufficiently large 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. Indeed, for x_0 lexicographically minimal such that WITNESS TEST accepts $(\mathbb{A}, c, x_0, k)$, this complete accepting run is simulated by our algorithm for large enough n, and then our algorithm would output $k+1$.

It follows that $\mathbb{A}$ is $(n^c + c)$-time bounded: otherwise there is x_0 such that $\mathbb{A}$ on x_0 does not halt within $|x_0|^c + c$ steps and then WITNESS TEST would accept $(\mathbb{A}, c, x_0, k)$ in line 3. We arrive at the desired contradiction by showing $Q \in \mathrm{P}$. We have two cases:

If k is even, then Q and Q' have finite symmetric difference, so it suffices to show $Q' \in \mathrm{P}$. But this is true because $\mathbb{A}$ decides Q': otherwise there is x_0 such that either the condition in line 5 or the one in line 6 is satisfied, so WITNESS TEST would accept $(\mathbb{A}, c, x_0, k)$.

If k is odd, then Q' is finite, so it suffices to show $Q \leq_p Q'$. But this is true because $\mathbb{A}$ computes a reduction from Q to Q': otherwise there is x_0 such that either the condition in line 10 or the one in line 11 is satisfied, so WITNESS TEST would accept $(\mathbb{A}, c, x_0, k)$. □

3.8 SAT solvers

3.8.1 Self-reduciblity

If SAT ∈ P, can we find a satisfying assignments in polynomial time?

Definition 3.8.1 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 3.8.2 *If* SAT ∈ P*, then there exists a polynomial time bounded* SAT *solver.*

Proof: The proof 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 $\emptyset$. 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** run $\mathbb{A}^*$ on $(\alpha(X_1, \ldots, X_{n-1}, 0), A \cup \{(X_n, 0)\})$
3. 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 3.8.3 Recall the discussion at the end of Section 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 3.8.4 Assume P = NP. Show that for every polynomially bounded $R \subseteq (\{0,1\}^*)^2$ in P there is a polynomial time computable partial function $f \subseteq R$ with $\mathrm{dom}(f) = \mathrm{dom}(R)$. *Hint:* Recall the proof of Cook's Theorem.

3.8.2 Levin's theorem

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

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

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. ℓ ← 0
2. for all i ≤ ℓ do
3.     simulate ℓ steps of A_i on x
4.     if the simulated run halts and outputs y such that (x, y) ∈ R
5.         then halt and output y
6. ℓ ← ℓ + 1
7. goto 1
```

It is easy to check that $\mathbb{O}$ is a SAT solver. The time needed in lines 2-4 can be bounded by $c \cdot (\ell + |x|)^d$ for suitable $c, d \in \mathbb{N}$. 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 $\ell \geq t_{\mathbb{A}}(\alpha), i_{\mathbb{A}}$ – or earlier. Hence,

$$t_{\mathbb{O}}(\alpha) \leq O(\textstyle\sum_{\ell=0}^{t_{\mathbb{A}}(\alpha)+i_{\mathbb{A}}} (\ell + |\alpha|)^d) \leq c_{\mathbb{A}} \cdot (t_{\mathbb{A}}(\alpha) + |\alpha|)^{d+1}$$

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

Remark 3.8.6 The proof shows that the degree of $p_{\mathbb{A}}$ does not depend on $\mathbb{A}$.

Exercise 3.8.7 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 3.8.8 *Let $\mathbb{O}$ be the* SAT *solver from Theorem 3.8.5. Then* P = NP *if and only if there is a polynomial p such that $t_{\mathbb{O}}(\alpha) \leq p(|\alpha|)$ for every $\alpha \in$ SAT.*

Proof: By Theorem 3.8.2, P = NP implies there is a SAT solver $\mathbb{A}$ that is q-time bounded for some polynomial q. By Theorem 3.8.5, $t_{\mathbb{O}}(\alpha) \leq p_{\mathbb{A}}(q(|\alpha|) + |\alpha|)$ for $\alpha \in$ SAT.

Conversely, suppose a polynomial p as described exists. Then a formula α is satisfiable if and only if $\mathbb{O}$ on α outputs a satisfying assignment in time $p(|\alpha|)$. This can be checked in polynomial time, so SAT $\in$ P and P = NP follows. □

Chapter 4

Space

4.1 Space bounded computation

Recall that a configuration of a deterministic or nondeterministic Turing machine $\mathbb{A}$ with an 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 is the position of the input head, $\bar{j} = j_1 \cdots j_k$ are the positions of the heads on the k worktapes, c is the content of the input tape and $\bar{c} = c_1 \cdots c_k$ are the contents of the k worktapes.

Definition 4.1.1 Let $S \in \mathbb{N}$. A configuration $(s, i\bar{j}, c\bar{c})$ of $\mathbb{A}$ is *S-small* if $j_1, \ldots, j_k \leq S$ 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 \$ $x_1 \cdots x_n \,\square\, \square \cdots$.

Given x, an S-small configuration of $\mathbb{A}$ on x 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

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

Definition 4.1.2 Let $\mathbb{A}$ be a deterministic or nondeterministic Turing machine with input tape and k worktapes and $s : \mathbb{N} \to \mathbb{N}$. Then $\mathbb{A}$ is *s-space bounded* if for every $x \in \{0,1\}^*$ all configurations reachable (in the configuration graph) from the start configuration of $\mathbb{A}$ on x are $s(|x|)$-small. The set SPACE(s) (NSPACE(s)) contains the problems Q that are for some constant $c \in \mathbb{N}$ decided (accepted) by a (nondeterministic) $(c \cdot s + c)$-space bounded Turing machine. We define

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

Notation: Assume $\mathbb{A}$ is s-space bounded for some $s \geq \log$ and $x \in \{0,1\}^*$. Then every configuration reachable from the start configuration of $\mathbb{A}$ on x can be coded as explained above by a string of length *exactly* $c \cdot s(|x|)$ for c some suitable constant. Consider the graph

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

where $E_x^{\mathbb{A}}$ contains (c, d) if c and d code $s(|x|)$-small configurations and (the configuration coded by) d is a successor configuration of c.

Space complexity measures memory requirements to solve problems as time complexity measures time. Note that a time restriction f with $f(n) < n$ is boring in the sense that the time restriction does not allow us to even read the input. Not so for space restrictions, as they only apply to the work tape, and 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.

Definition 4.1.3 A function $s : \mathbb{N} \to \mathbb{N}$ is *space-constructible* if $s(n) \geq \log n$ 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 4.1.4 (Space Hierarchy) *Let s be space-constructible and $s' \leq o(s)$ arbitrary. Then* $\mathrm{SPACE}(s) \setminus \mathrm{SPACE}(s') \neq \emptyset$.

Proof: The proof mimicks the proof of the time hierarchy theorem. Recall $\ulcorner \mathbb{A} \urcorner$ denotes the binary encoding of a Turing machine $\mathbb{A}$ and consider the problem

P_s	
Instance:	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

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

many bits, say 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 $P_s \in \mathrm{SPACE}(s)$.

Let $s' \leq o(s)$ and assume for the sake of contradiction that $P_s \in \mathrm{SPACE}(s')$. 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 sufficiently large n. By adding, if necessary, dummy states to $\mathbb{B}$ we can assume that $\lvert \ulcorner \mathbb{B} \urcorner \rvert$ is sufficiently large.

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 $\lvert \ulcorner \mathbb{B} \urcorner \rvert \geq n_0$ 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 $\mathbb{B}$. □

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

Proposition 4.1.5 *Let $s : \mathbb{N} \to \mathbb{N}$ be space-constructible. Then*

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

Proof: Let Q in $\mathrm{NTIME}(s)$, 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 $Q \in \mathrm{SPACE}(s)$ 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 $bin(s(|x|))$ is 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 4.1.6 $\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP}$.

None of the displayed inclusions is known to be strict. From the time and space hierarchy theorems we know $\mathrm{P} \neq \mathrm{EXP}$ and $\mathrm{L} \neq \mathrm{PSPACE}$. We shall now see that even $\mathrm{NL} \neq \mathrm{PSPACE}$.

4.1.1 Savitch's theorem

Theorem 4.1.7 (Savitch 1970) REACHABILITY $\in \mathrm{SPACE}(\log^2)$.

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 $\leq 2^i$ 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:

$\mathbb{A}$ 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** $v \leq n$
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

If $s_{\mathbb{A}}(i)$ upper bounds the space needed by $\mathbb{A}$ on triples $V \times V \times \{i\}$ we have

$$s_{\mathbb{A}}(i) \leq O(\log n) + s_{\mathbb{A}}(i-1)$$

for $i \leq n$ and thus $s_{\mathbb{A}}(\log n) \leq O(\log^2 n)$. This proves the theorem because $\mathbb{A}$ on $(u, w, \lceil \log n \rceil)$ decides whether there is a path from u to w. □

Corollary 4.1.8 *If s is space-constructible, then* $\mathrm{NSPACE}(s) \subseteq \mathrm{SPACE}(s^2)$.

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 (e.g. equal to the starting configuration c_{start} with a 1 written on the first cell of the last worktape). 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 $E_x^{\mathbb{A}}$. □

This implies $\mathrm{NL} \subseteq \mathrm{SPACE}(\log^2) \neq \mathrm{PSPACE}$. It is open whether $\mathrm{NL} \subseteq \mathrm{SPACE}(o(\log^2))$.

Corollary 4.1.9 $\mathrm{NPSPACE} = \mathrm{PSPACE}$.

4.2 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.

Exercise 4.2.1 Show that for every quantified Boolean formula α with r quantifiers there exists a (quantifier free) propositional formula β that is equivalent to α of size $|\beta| \leq 2^r \cdot |\alpha|$ (here the size $|\alpha|$ of a formula α is the length of a reasonable encoding of α as a binary string).

Exercise 4.2.2 Given a Boolean circuit $C(\bar{X}) = C(X_1, \ldots, X_r)$ 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 $x \in \{0,1\}^r$

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

Hint: Recall the proof of Lemma 3.5.2.

Exercise 4.2.3 Show that there is a polynomial time algorithm that given a quantified Boolean formula computes an equivalent quantified Boolean formula 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.

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).

Theorem 4.2.4 QBF *is* PSPACE*-complete.*

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

Proof: Given a quantified Boolean sentence we rewrite it in prenex form (see Exercise 4.2.3) and then run the following recursive algorithm $\mathbb{A}$.

$\mathbb{A}$ on $q_1X_1 \cdots q_rX_r\beta(X_1, \ldots, X_r)$

1. **if** $r > 0$ and $q_1 = \forall$
2. **if** $\mathbb{A}$ accepts $q_2X_2 \cdots q_rX_r\beta(0, X_2 \ldots, X_r)$
3. clean tape
4. **if** $\mathbb{A}$ accepts $q_2X_2 \cdots q_rX_r\beta(1, X_2 \ldots, X_r)$, accept
5. reject
6. **if** $r > 0$ and $q_1 = \exists$
7. **if** $\mathbb{A}$ rejects $q_2X_2 \cdots q_rX_r\beta(0, X_2 \ldots, X_r)$
8. clean tape
9. **if** $\mathbb{A}$ rejects $q_2X_2 \cdots q_rX_r\beta(1, X_2 \ldots, X_r)$, reject
10. accept
11. check that β is true.

Note that the last line is reached only when the sentence does not contain quantifiers, so β is just a Boolean combination of Boolean constants. Truth then can be easily checked in polynomial time (cf. Exercise 2.4.4). 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 $s_{\mathbb{A}}(n, r) \leq s_{\mathbb{A}}(n, r-1) + O(n)$ and thus $s_{\mathbb{A}}(n, r) \leq O(r \cdot n)$. 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 graph $G_x^{\mathbb{A}} = (\{0,1\}^m, E_x^{\mathbb{A}})$ where $m \leq O(p(|x|))$. As in the proof of Corollary 4.1.9 we again assume that this graph contains at most one (code of an) accepting configuration c_{acc} reachable from the start configuration c_{start} of $\mathbb{A}$ on x. By Lemma 2.4.7 we find a circuit $C(X_1, \ldots, X_m, Y_1, \ldots, Y_m)$ (depending on x) such that for all $c, d \in \{0,1\}^m$

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

By Exercise 4.2.2 we find a quantified Boolean formula $\sigma(X_1, \ldots, X_m, Y_1, \ldots, Y_m)$ such that

$$\sigma(c, d) \text{ is true} \iff C(c, d) = 1.$$

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

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

If we manage to do this, 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 $\bigwedge_{i=1}^m (X_i \leftrightarrow Y_i)$. For $\sigma_{i+1}(\bar{X}, \bar{Y})$ we take, following the same idea as in Savitch's theorem, a formula equivalent to

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

We cannot use the displayed formula because then σ_m would have length $\geq 2^m$, so cannot be computed in polynomial time. Instead we take

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

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

4.3 Nondeterministic logarithmic space

We give a simple, but useful characterization of NL paralleling Proposition 3.2.3.

Definition 4.3.1 Consider a Turing machine $\mathbb{A}$ with an input tape and with a special work-tape called *guess tape*; the head of the guess tape is only allowed to move right; a computation of $\mathbb{A}$ on a pair of strings (x, y) is a path in the configuration graph of $\mathbb{A}$ starting with the configuration where all heads are on the first cell, x is written on the input tape and y is written on the guess tape. This computation can be accepting (rejecting), and we accordingly say that $\mathbb{A}$ *accepts (rejects)* (x, y). If there is a constant $c \in \mathbb{N}$ such that for all (x, y) the computation of $\mathbb{A}$ on (x, y) does not contain a configuration with head position $c \cdot \log |x|$ on some work tape except possibly the guess tape, then $\mathbb{A}$ is said to be a *logspace verifier*.

Proposition 4.3.2 *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 $x \in \{0,1\}^*$*

$$x \in Q \Longleftrightarrow \exists y \in \{0,1\}^{\leq p(|x|)} : \mathbb{A} \text{ accepts } (x, y).$$

Proof: To show that (1) implies (2) let $Q \in$ 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). An algorithm witnessing $Q \in$ NL simulates $\mathbb{A}$ and guesses at every step the symbol scanned by $\mathbb{A}$ on the guess tape. □

Remark 4.3.3 In (2) one can assume $\mathbb{A}$ to move the head one cell to the right in every step; furthermore, one can replace $y \in \{0,1\}^{\leq p(|x|)}$ by $y \in \{0,1\}^{p(|x|)}$.

Example 4.3.4 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 clear how to make $\mathbb{A}$ reject "proofs" y that do not even encode a sequence of vertices $\bar{u}$.

4.3.1 Implicit logarithmic space computability

Example 4.3.5 REACHABILITY $\in$ NL.

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

```
1. u ← v
2. while u ≠ v'
3.     guess w ∈ [n]
4.     if (u, w) ∈ E, then u ← w
5. accept
```

is a nondeterministic algorithm that runs in space roughly $2 \log n$. □

Reingold proved that REACHABILITY is in L when restricted to undirected graphs. But in general it is not known whether REACHABILITY $\in$ L. We now show that the answer is no unless L = 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:

Exercise 4.3.6 Show that NL is closed under $\leq_p$ if and only if NL = 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 1.2

Definition 4.3.7 $f : \{0,1\}^* \to \{0,1\}^*$ is *implicitly logspace computable* if it is polynomially bounded (i.e. $|f(x)| \leq |x|^{O(1)}$ for all $x \in \{0,1\}^*$) and BITGRAPH$(f) \in$ L. A problem Q is *logspace reducible* to another Q' if there is an implicitly logspace computable reduction from Q to Q'; we write $Q \leq_{log} Q'$ to indicate this. That $\mathcal{C} \subseteq P(\{0,1\}^*)$ is *closed under* $\leq_{log}$, that a problem is *hard* or *complete* for $\mathcal{C}$ under $\leq_{log}$ is explained analogously as for $\leq_p$.

Exercise 4.3.8 Every implicitly logspace computable function is polynomial time computable. Conclude $\leq_{log} \subseteq \leq_p$ and that P, NP, PSPACE, EXP are $\leq_{log}$-closed.

Lemma 4.3.9 *If $Q \leq_{log} Q'$ and $Q' \leq_{log} Q''$, then $Q \leq_{log} Q''$.*

Proof: It suffices to show that $f \circ g$ is implicitly logspace computable if both f and g are. First note that $f \circ g$ is polynomially bounded as so are f and g, and choose a polynomial p such that $|f(g(x))| < p(|x|)$. To decide the bit-graph of $f \circ g$, let $\mathbb{A}$ be an algorithm witnessing BITGRAPH$(f) \in$ L. Given an instance (x, i, b) of BITGRAPH$(f \circ g)$ we simulate $\mathbb{A}$ on $g(x)$ without computing $g(x)$: we maintain (the code of) a small configuration of $\mathbb{A}$ on $g(x)$ plus the current input symbol scanned. From this it is easy to compute the successor configuration of $\mathbb{A}$ on $g(x)$. Say in this configuration $\mathbb{A}$ scans cell i' of the input tape. To get the symbol scanned there we run a logspace algorithm for BITGRAPH(g) on $(x, i', 0)$ and $(x, i', 1)$. □

Exercise 4.3.10 Show that $Q \in$ L if and only if χ_Q is implicitly logspace computable, if and only if $Q \leq_{log} \{1\}$. Conclude that L is $\leq_{log}$-closed. Show, arguing similarly as in the previous lemma, that also NL is $\leq_{log}$-closed.

Theorem 4.3.11 REACHABILITY *is* NL*-complete under* $\leq_{log}$.

Proof: By Example 4.3.5 it suffices to show that REACHABILITY is NL-hard. Let $Q \in \mathrm{NL}$ and choose $c \in \mathbb{N}$ 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 $(G_x^{\mathbb{A}}, c_{\mathrm{start}}, c_{\mathrm{acc}})$ of REACHABILITY. □

4.3.2 Immerman and Szelepcsényi's theorem

Theorem 4.3.12 (Immerman, Szelepcsényi 1987) NONREACHABILITY $\in$ NL.

NONREACHABILITY	
Instance:	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 $n > 2$. Let

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

We define a logspace verifier $\mathbb{A}$ for NONREACHABILITY . It expects a "proof" y on the guess tape of the form $y = \rho_1 \cdots \rho_{n-2}$ and proceeds in such a way that after reading ρ_i it stores $|R_i|$ or abandons the computation and rejects. In the beginning it stores $|R_0| = 1$. Details follow.

The string ρ_{i+1} consists in n blocks

$$\rho_{i+1} = \pi_1 \cdots \pi_n,$$

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

$$\pi_u := \bar{u}_1 v_1, \ldots, \bar{u}_s v_s$$

such that the following conditions hold:

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

(b) $s = |R_i|$,

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

Reading $\pi_u = \bar{u}_1 v_1, \ldots, \bar{u}_s v_s$ from left to right, $\mathbb{A}$ checks these conditions in logspace: for (a) see the previous example, for (b) this is easy as $\mathbb{A}$ already stores $|R_i|$ when reading ρ_{i+1}, and for (c) $\mathbb{A}$ needs to remember only the last node v_j read. $\mathbb{A}$ increases a counter if $v_j = u$ or $(v_j, u) \in E$ for at least one $j \leq s$. After having read ρ_{i+1}, the counter stores $|R_{i+1}|$ as desired.

Further counters signal $\mathbb{A}$ when it is reading ρ_{n-2} and therein the block $\pi_{v'}$; then $\mathbb{A}$ accepts if during this block the counter is not increased. □

Definition 4.3.13 For $\mathcal{C} \subseteq P(\{0,1\}^*)$ let $\mathrm{co}\mathcal{C} := \{\{0,1\}^* \setminus Q \mid Q \in \mathcal{C}\}$.

Exercise 4.3.14 Show $\mathrm{coTIME}(t) = \mathrm{TIME}(t)$ and $\mathrm{coSPACE}(t) = \mathrm{SPACE}(t)$.

Concerning nondeterministic time classes, it is widely believed that NP $\neq$ coNP but this is not known – see next chapter. In contrast, for nondeterministic space we now know:

Corollary 4.3.15 NL = coNL.

Proof: Theorem 4.3.11 implies that $\{0,1\}^* \setminus$ Reachability is coNL-hard under $\leq_{log}$. This problem easily logspace reduces to NonReachability, so NonReachability is coNL-hard under $\leq_{log}$. By the previous theorem (and because NL is closed under $\leq_{log}$ by Exercise 4.3.10) we get coNL $\subseteq$ NL. But then also NL $=$ cocoNL $\subseteq$ coNL. □

More generally, the following can be inferred from Theorem 4.3.12, similarly as Corollary 4.1.9 is inferred from Theorem 4.1.7.

Corollary 4.3.16 *Let* $s \geq \log$ *be space-constructible. Then* NSPACE(s) = coNSPACE(s).

Corollary 4.3.17 2Sat *is* NL*-complete under* $\leq_{log}$.

Proof: Recall the claim in the proof of Theorem 3.5.5. By this claim a 2-CNF α is satisfiable if and only if there is no (directed) path from a variable to its negation and vice versa in a certain directed graph (V, E). This can be checked using the algorithm from Theorem 4.3.12 successively for each variable, reusing space. Whenever this algorithm wants to check whether $(\lambda, \lambda') \in E$, we check whether $(\overline{\lambda} \vee \lambda')$ or $(\lambda' \vee \overline{\lambda})$ is a clause in α. This shows 2Sat $\in$ NL.

To show hardness, it suffices to show NonReachability $\leq_{log}$ 2Sat (recall the proof above): map an instance $((V, E), v, v')$ of NonReachability to the 2-CNF

$$X_v \wedge \neg X_{v'} \wedge \bigwedge_{(u,w)\in E}(\neg X_u \vee X_w),$$

in the variables $X_u, u \in V$. This formula is implicitly logspace computable from $((V, E), v, v')$, and it is satisfiable if and only if $((V, E), v, v') \in$ NonReachability. □

Chapter 5

Alternation

5.1 Co-nondeterminism

Recall Definition 4.3.13. This introductory section treats the coNP versus NP question. As a first observation, we note that coNP problems are universal projections of polynomial time relations. This follows directly from the definitions.

Proposition 5.1.1 *Let Q be a problem. The following are equivalent.*

1. $Q \in$ coNP.
2. *There exists a polynomial time decidable relation $R \subseteq (\{0,1\}^*)^2$ and a polynomial p such that for all $x \in \{0,1\}^*$*

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

Remark 5.1.2 Again, one can equivalently replace $\leq p(|x|)$ in (2) by $p(|x|)$.

Proposition 5.1.3 Taut *is* coNP*-complete.*

Taut	
Instance:	a propositional formula α.
Problem:	is α valid?

Proof: Note α is *not* satisfiable if and only if $\neg\alpha$ is valid. Hence Taut $\equiv_p \{0,1\}^* \setminus$ Sat and the proposition follows from the NP-completeness of Sat. □

We see that similarly as NP is characterized by existential quantifiers, coNP is characterized by universal quantifiers. For NP we turned this into a machine characterization of NP by introducing nondeterministic Turing machines. We shall proceed for coNP similarly in the next section but in somewhat more general context. To get some feeling, let's play around a little with the definition of the nondeterministic acceptance condition.

Note a nondeterministic machine lies, it may reject "yes" instances on some paths, so it may give false negative answers. In contrast, if a nondeterministic machine answers "yes" on some path, we can trust that answer. It is nearlying to consider nondeterministic machines that never lie, so all paths lead to the correct answer. However, such machines accept precisely the problems in P, so modifying the definition in this way gives nothing new. So let's consider

machines that say "yes", "no" or "dunno" and never lie. Recall we agreed that accepting runs are those that halt with a 1 in the first cell on the worktape, and rejecting runs are those halting with a 0 in the first cell. We interpret a blank $\square$ in the first cell as the answer "I don't know". Recall χ_Q denotes the characteristic function of a problem Q.

Definition 5.1.4 A nondeterministic Turing machine $\mathbb{A}$ *strongly 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 5.1.5 *Let Q be a problem. The following are equivalent.*

1. $Q \in \mathrm{NP} \cap \mathrm{coNP}$.
2. *There exists a polynomial time bounded nondeterministic Turing machine that strongly accepts Q.*

Proof: Assume $Q \in \mathrm{NP} \cap \mathrm{coNP}$ and choose nondetermnistic polynomial time 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 (2) 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 (2) already witnesses $Q \in \mathrm{NP}$. The machine that simulates $\mathbb{A}$ and accepts precisely if $\mathbb{A}$ says "no", witnesses $Q \in \mathrm{coNP}$. $\square$

The inclusion $\mathrm{P} \subseteq \mathrm{NP} \cap \mathrm{coNP}$ is not known but generally believed to be strict.

5.2 Unambiguous nondeterminism

We continue to play around with the acceptance condition.

Definition 5.2.1 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 $\mathrm{P} \subseteq \mathrm{UP} \subseteq \mathrm{NP}$ are not known but generally believed to be strict.

Definition 5.2.2 A *worst-case one-way function* is a polynomial time computable, honest injection $f : \{0,1\}^* \to \{0,1\}^*$ such that the partial function f^{-1} is not polynomial time computable. Being *honest* means that there is a constant $c \in \mathbb{N}$ such that for all $x \in \{0,1\}^*$ we have $|x| \leq |f(x)|^c$.

Remark 5.2.3 *Cryptography* needs the assumption that worst-case one-way 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 5.2.4 *Worst-case one-way functions exist if and only if* $\mathrm{P} \neq \mathrm{UP}$.

Proof: For the backward direction assume $\mathbb{A}$ is a polynomial time unambiguous machine accepting $Q \notin \mathrm{P}$. As usual we consider runs of $\mathbb{A}$ as determined by strings $y \in \{0,1\}^{p(|x|)}$ where p is a polynomial bounding the running time of $\mathbb{A}$. Then the function given by

$$f_{\mathbb{A}}(x) := \begin{cases} 1z & \text{if } x = \langle y, z\rangle \text{ where } y \text{ determines an length } |y| \text{ accepting run of } \mathbb{A} \text{ on } z \\ 0x & \text{if } x \text{ is not of this form} \end{cases}$$

is clearly polynomial time computable. It is injective, because $\mathbb{A}$ is unambiguous, and it is honest. But $z \in Q$ if and only if $1z \in \mathrm{dom}(f_{\mathbb{A}}^{-1})$, so $f_{\mathbb{A}}^{-1}$ is not polynomial time computable.

Conversely, assume f is a worst-case one-way function and consider

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

where $\leq_{\mathrm{lex}}$ denotes the lexicographic order. The nondeterministic machine that on $\langle x, y\rangle$ guesses z and checks that $z \leq_{\mathrm{lex}} x$ and $f(z) = y$, witnesses that $Q_f \in \mathrm{UP}$.

We claim $Q_f \notin \mathrm{P}$. Otherwise f^{-1} could be computed in polynomial time by "binary search" as follows: given y, check $y \in \mathrm{dom}(f^{-1})$ by checking in polynomial time that $\langle 1^{|y|^c}, y\rangle \in Q_f$. Here $c \in \mathbb{N}$ witnesses that f is honest. If the check fails, then reject. Otherwise ask check successively $\langle 1^{|y|^c-1}, y\rangle \in Q_f, \langle 1^{|y|^c-2}, y\rangle \in Q_f, \ldots$ until you find $\langle 1^{\ell-1}, y\rangle \notin Q_f$. Then $|f^{-1}(y)| = \ell$. Check whether $\langle 01^{\ell-1}, y\rangle \in Q_f$. If so, $f^{-1}(y)$ starts with 0 and otherwise with 1; in other words $f^{-1}(y)$ starts with $b_1 := 1 - \chi_{Q_f}(\langle 01^{\ell-1}, y\rangle)$ Then ask whether $\langle b_1 01^{\ell-2}, y\rangle \in Q_f$. Then $f^{-1}(y)$ starts with $b_1 b_2$ where $b_2 := 1 - \chi_{Q_f}(\langle b_1 01^{\ell-2}, y\rangle)$. After ℓ steps one gets $b_1 \cdots b_\ell = f^{-1}(y)$. □

Exercise 5.2.5 Show that NP = coNP if and only if there exists a polynomial time computable, honest function with image TAUT.

Remark 5.2.6 Polynomial time computable functions P with image TAUT are called propositional proof systems – strings x with $P(x) = \alpha$ are called a P-proof of α. By the last exercise it is nearlying to believe that P-proofs cannot be polynomially bounded in length. *Cook's program* asks to prove this for natural proof systems P known from logic and the corresponding research area is known as *Proof Complexity*. This line to attack the coNP versus NP question hits at long standing open problems from mathematical logic concerning the independence of certain statements from weak theories of arithmetic.

5.3 The polynomial hierarchy

Recall that for a relation $R \subseteq (\{0,1\}^*)^t$ being in P means that $\{\langle x_1, \ldots, x_t\rangle \mid (x_1, \ldots, x_t) \in R\} \in \mathrm{P}$. We generalize the definition of NP.

Definition 5.3.1 Let $t \geq 1$. A problem Q is in Σ_t^{P} if and only if there are a relation $R \subseteq (\{0,1\}^*)^{t+1}$ in P and polynomials $p_1, \ldots, p_t$ such that for all $x \in \{0,1\}^*$:

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

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

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

Exercise 5.3.2 ne can equivalently write $y_i \in \{0,1\}^{p(|x|)}$ for a single polynomial p.

Exercise 5.3.3 Show that Σ_t^{P} and Π_t^{P} and PH are $\leq_p$-closed.

The following is easy to see:

Proposition 5.3.4 $\Sigma_t^{\mathrm{P}} \cup \Pi_t^{\mathrm{P}} \subseteq \Sigma_{t+1}^{\mathrm{P}} \cap \Pi_{t+1}^{\mathrm{P}}$.

It is not known whether any of Σ_t^{P} and Π_t^{P} are equal. But we shall see now that the hypotheses $\Sigma_t^{\mathrm{P}} \neq \Pi_t^{\mathrm{P}}$ grow in strength, so these hypotheses are stepwise strengthenings of the hypothesis that NP $\neq$ coNP. Later on we shall see some examples how 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 NP $\neq$ coNP.

Theorem 5.3.5 *Let $t \geq 1$. If $\Sigma_t^{\mathrm{P}} = \Pi_t^{\mathrm{P}}$, then $\mathrm{PH} = \Sigma_t^{\mathrm{P}}$. Further, if $\mathrm{P} = \mathrm{NP}$, then $\mathrm{P} = \mathrm{PH}$.*

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

To show the first statement assume $\Sigma_t^{\mathrm{P}} = \Pi_t^{\mathrm{P}}$. We prove that $\Sigma_{t+k}^{\mathrm{P}} \subseteq \Sigma_t^{\mathrm{P}}$ by induction on k. Assume the claim for k and let $Q \in \Sigma_{t+k+1}^{\mathrm{P}}$. Choose a relation R in P such that

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

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

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

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

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

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

Let R'' contain the tuples $(x, z, y_2', \ldots, y_{t+1}')$ such that there are y_1 with $|y_1| \leq p_1(|x|)$ and y_2' with $|y_2'| \leq p_2'(|x|)$ 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 $Q \in \Sigma_t^{\mathrm{P}}$. □

Corollary 5.3.6 *If there is a problem that is complete for* PH*, then* PH collapses*, i.e.* PH $= \Sigma_t^{\mathrm{P}}$ *for some $t \geq 1$.*

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

Remark 5.3.7 Philosophically speaking, recall we intend to interpret complexity classes as degrees of difficulties of certain problems. For example, we interpret NP as the degree of difficulty of solving SAT and NL as the degree of difficulty of solving REACHABILITY. It is not possible to understand PH in this sense unless it collapses.

Definition 5.3.8 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 5.3.9 *For all $t \geq 1$, Σ_tSAT is Σ_t^{P}-complete.*

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

Proof: It is easy to see that Σ_tSAT is contained in Σ_t^{P}. The hardness proof generalizes the proof of Theorem 3.5.1. Given $Q \in \Sigma_t^{\mathrm{P}}$ choose a relation R in P such that for all $x \in \{0,1\}^n$

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

with the y_is ranging over $\{0,1\}^{p(|x|)}$ for some polynomial p (Exercise 5.3.2). 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 $C_{|x|}$ computable in polynomial time from $|x|$ (by Lemma 2.4.7, see the proof of Theorem 3.5.1). By Exercise 4.2.2 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 5.3.10 $\mathrm{P} \subseteq \mathrm{NP} = \Sigma_1^{\mathrm{P}} \subseteq \Sigma_2^{\mathrm{P}} \subseteq \cdots \subseteq \mathrm{PH} \subseteq \mathrm{PSPACE}$ *and all inclusions are strict unless* PH *collapses.*

Proof: To see the last inclusion, note $\bigcup_{t \geq 1} \Sigma_t$SAT is PH-hard by the previous theorem. But the identity map reduces this problem to QBF which is contained in PSPACE. The other inclusions are trivial (Proposition 5.3.4).

Note that $\Sigma_t^{\mathrm{P}} = \Sigma_{t+1}^{\mathrm{P}}$ implies $\Pi_t^{\mathrm{P}} \subseteq \Sigma_t^{\mathrm{P}}$ (Proposition 5.3.4) and thus $\Sigma_t^{\mathrm{P}} = \mathrm{co}\Pi_t^{\mathrm{P}} \subseteq \mathrm{co}\Sigma_t^{\mathrm{P}} = \Pi_t^{\mathrm{P}}$, so $\Sigma_t^{\mathrm{P}} = \Pi_t^{\mathrm{P}}$. Assuming PH does not collapse, these inclusions are strict by Theorem 5.3.5. Strictness of the last inclusion PH $\subseteq$ PSPACE follows by Corollary 5.3.6 since PSPACE has complete problems. □

5.4 Alternating time

In this section we give a machine characterization of Σ_t^{P} analogously as we did for NP in Proposition 3.2.3.

Definition 5.4.1 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 *accepts* 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 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 a constant $t \geq 1$, an alternating Turing machine $\mathbb{A}$ is *t-alternating* if for all $x \in \{0,1\}^*$ every run of $\mathbb{A}$ moves for at most $t-1$ times from an existential to a universal state or vice versa; furthermore, the starting state s_{start} is existential.

For $f : \mathbb{N} \to \mathbb{N}$ the set $\text{ATIME}(f)$ (resp. $\Sigma_t\text{TIME}(f)$) contains the problems Q such that there are a constant $c \in \mathbb{N}$ and a $(c \cdot f + c)$-time bounded (resp. t-)alternating Turing machine $\mathbb{A}$ that accepts Q. We set

$$\begin{array}{rcl} \text{AP} & := & \bigcup_{c\in\mathbb{N}} \text{ATIME}(n^c) \\ \Sigma_t\text{P} & := & \bigcup_{c\in\mathbb{N}} \Sigma_t\text{TIME}(n^c) \end{array}$$

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

Proposition 5.4.2 For all $t \geq 1$, $\Sigma_t\text{P} = \Sigma_t^{\text{P}}$.

Proof: It is easy to solve $\Sigma_t\text{SAT}$ by a t-alternating machine in polynomial time. It is also easy to see that $\Sigma_t\text{P}$ is $\leq_p$-closed. This implies the inclusion from right to left (Theorems 5.3.9).

To see $\Sigma_t\text{P} \subseteq \Sigma_t^{\text{P}}$, let Q be a problem accepted by a t-alternating Turing machine $\mathbb{A}$ in polynomial time. Let $c \in \mathbb{N}$ be such that for long enough inputs x all runs of $\mathbb{A}$ on x have length at most $|x|^c$. Every run of $\mathbb{A}$ on x can be divided into $\leq t$ 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 tuples $(x, y_1, \ldots, y_t)$ such that the run of $\mathbb{A}$ on x determined by $(y_1, \ldots, y_t) \in (\{0,1\}^{|x|^c})^t$ is accepting; here we mean the run where the jth step in the ith block uses the transition function according to the jth bit of y_i. This relation can be decided in linear time (namely time $O(|x|^c)$) and witnesses that $Q \in \Sigma_t^{\text{P}}$. □

Proposition 5.4.3 $\text{AP} = \text{PSPACE}$

Proof: Note that AP is $\leq_p$-closed. By Theorem 4.2.4 it thus suffices to show that QBF is AP-complete. It is easy to see that $\text{QBF} \in \text{AP}$. For hardness, suppose Q is accepted by a p-time bounded alternating Turing machine $\mathbb{A}$ where p is some polynomial. Then $x \in Q$ if and only if

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

where y_i, z_i range over bits and respectively form the bits of the strings $y, z \in \{0,1\}^{p(|x|)}$. The relation R contains the triples (x, y, z) 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 5.3.9. □

Exercise 5.4.4 Define $\Pi_t\text{P}$ like $\Sigma_t\text{P}$ except that in the definition of t-alternating the starting state s_{start} is demanded to be universal. Show that $\Pi_t\text{P} = \text{co}\Sigma_t\text{P}$. In particular, $\Pi_1\text{P} = \text{coNP}$.

5.5 Oracles

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

Definition 5.5.1 Let O be a problem. An *oracle machine with oracle* O 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 $b = \chi_O(z)$ 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^O (NP^O) contains the problems accepted by a polynomial time bounded (nondeterministic) oracle machine with oracle O.

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

Exercise 5.5.2 $\mathrm{NP}^{\mathcal{C}}$ is $\leq_p$-closed for every $\mathcal{C} \subseteq P(\{0,1\}^*)$. If $Q \subseteq \{0,1\}^*$ is $\mathcal{C}$-hard, then $\mathrm{NP}^{\mathcal{C}} \subseteq \mathrm{NP}^Q$. Further show that $\mathrm{NP}^{\mathrm{P}} = \mathrm{NP} \subseteq \mathrm{NP} \cup \mathrm{coNP} \subseteq \mathrm{P}^{\mathrm{NP}} = \mathrm{P}^{\mathrm{SAT}}$.

Exercise 5.5.3 To address a maybe confusing point in the notation, think about whether $\mathrm{P} = \mathrm{NP}$ implies $\mathrm{P}^O = \mathrm{NP}^O$ for all $O \subseteq \{0,1\}^*$.

Theorem 5.5.4 For all $t \geq 1$, $\Sigma_{t+1}^{\mathrm{P}} = \mathrm{NP}^{\Sigma_t \mathrm{SAT}}$.

Proof: For the inclusion $\Sigma_{t+1}^{\mathrm{P}} \subseteq \mathrm{NP}^{\Sigma_t\mathrm{SAT}}$ it suffices to show $\Sigma_{t+1}\mathrm{SAT} \in \mathrm{NP}^{\Sigma_t\mathrm{SAT}}$ (by Theorem 5.3.9 and the above exercise). 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 (using nondeterministic steps) an assignment $\bar{b}$ to $\bar{X}$ 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 oracle 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 $O := \Sigma_t\mathrm{SAT}$. We can assume that $\mathbb{A}$ only makes queries that encode Σ_t-sentences. Say, an oracle step is *according to* a bit b if it ends in a configuration with 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 $y, a \in \{0,1\}^{\leq p(|x|)}$ 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 $a = \chi_O(\alpha_1)\cdots\chi_O(\alpha_{|a|})$ – this means the sentence

$$\beta := \bigwedge_{j=1}^{|a|} \begin{cases} \alpha_j & \text{, if } a_j = 1 \\ \neg\alpha_j & \text{, if } a_j = 0 \end{cases}$$

is true. Elementary formula manipulations transform β to an equivalent Σ_{t+1}-sentence β' in polynomial time. Thus $Q \in \Sigma_{t+1}\mathrm{P} = \Sigma_{t+1}^{\mathrm{P}}$ (Proposition 5.4.2) 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 this run is accepting and produces exactly $|a|$ many queries $\alpha_1,\ldots,\alpha_{|a|}$; then compute β' as above and check its truth. We already saw that this truth check can be done in $(t+1)$-alternating polynomial time (Proposition 5.4.2). □

Corollary 5.5.5 *For all* $t \geq 1$, $\Sigma_{t+1}^{\mathrm{P}} = \mathrm{NP}^{\Sigma_t^{\mathrm{P}}}$, *that is,* $\Sigma_2^{\mathrm{P}} = \mathrm{NP}^{\mathrm{NP}}, \Sigma_3^{\mathrm{P}} = \mathrm{NP}^{\mathrm{NP}^{\mathrm{NP}}}, \ldots$.

Proof: By Theorems 5.5.4 and 5.3.9, both sides of the equation equal $\mathrm{NP}^{\Sigma_t \mathrm{SAT}}$. □

Exercise 5.5.6 Use this corollary to show that $\Sigma_7^{\mathrm{P}} = \Sigma_2^{\mathrm{P}}$ if $\Sigma_3^{\mathrm{P}} = \Sigma_2^{\mathrm{P}}$.

Exercise 5.5.7 Define $(\mathrm{NP}^{\mathrm{NP}})^{\mathrm{NP}}$ and prove that it equals $\mathrm{NP}^{\mathrm{NP}}$.

Exercise 5.5.8 For $t \geq 1$ show $\Sigma_t^{\mathrm{P}} \cup \Pi_t^{\mathrm{P}} \subseteq \mathrm{P}^{\Sigma_t^{\mathrm{P}}} = \mathrm{P}^{\Pi_t^{\mathrm{P}}} \subseteq \Sigma_{t+1}^{\mathrm{P}} \cap \Pi_{t+1}^{\mathrm{P}}$.

5.6 Time-space trade-offs

Definition 5.6.1 Let $t, s : \mathbb{N} \to \mathbb{N}$. The set $\mathrm{TISP}(t, s)$ contains the problems decidable by a (deterministic) Turing machine with input tape that is $(c \cdot t + c)$-time bounded and $(c \cdot s + c)$-space bounded for some constant $c \in \mathbb{N}$.

Theorem 5.6.2 *If $a < \sqrt{2}$, then* $\mathrm{NTIME}(n) \not\subseteq \mathrm{TISP}(n^a, n^{o(1)})$.

We prove this via three lemmas. The first is proved by padding together with a similar trick as in Corollary 4.1.9 or in Lemma 4.3.9:

Lemma 5.6.3 *Let $t, s : \mathbb{N} \to \mathbb{N}$ be functions with $t(n) \geq n$ and $s(n) \geq \log n$ and let $c \geq 1$ be a natural. If* $\mathrm{NTIME}(n) \subseteq \mathrm{TISP}(t(n), s(n))$*, then* $\mathrm{NTIME}(n^c) \subseteq \mathrm{TISP}(n \cdot t(n^c), s(n^c))$.

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

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

Then Q' can be solved in linear nondeterministic time. By assumption Q' is decided by a machine $\mathbb{B}$ running in time $O(t)$ and space $O(s)$. We now decide Q as follows. Given x, run $\mathbb{B}$ on $xx \cdots x$ without actually computing $xx \cdots x$: instead simulate $\mathbb{B}$ on $xx \cdots x$ by storing its current head position on $xx \cdots x$. This allows to determine the bit scanned by $\mathbb{B}$ on $xx \cdots x$ in time $O(|x|)$. The additional space needed is only $O(\log |x|) \leq O(s(|x|))$. The simulation thus runs in time $O(|x| \cdot t(|x|^c))$ and space $O(s(|x|^c))$. □

Lemma 5.6.4 *Let $c, a \geq 1$ be rational. If* $\mathrm{NTIME}(n) \subseteq \mathrm{TIME}(n^a)$*, then* $\Sigma_2\mathrm{TIME}(n^c) \subseteq \mathrm{NTIME}(n^{ca})$.

Proof: Let $Q \in \Sigma_2\mathrm{TIME}(n^c)$. By the proof of Proposition 5.4.2 there is $R \subseteq (\{0,1\}^*)^3$ 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 $\{0,1\}^{|x|^c}$ and R is decidable in linear time, i.e. time $O(|x|^c)$. Then

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

can be accepted in nondeterministic linear time, i.e. time $O(|x|^c)$. By assumption, $\langle x, y_1 \rangle \stackrel{?}{\in} Q'$ can be decided in deterministic time $O(|x|^{ca})$. But

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

and $Q \in \mathrm{NTIME}(n^{ca})$ follows. □

The third and final lemma is the heart of the proof.

Lemma 5.6.5 *Let $c \in \mathbb{N}, c \geq 1$. Then* $\mathrm{TISP}(n^{2c}, n^{o(1)}) \subseteq \Sigma_2\mathrm{TIME}(n^{c+\epsilon})$ *for every real* $\epsilon > 0$.

Proof: Let Q be a problem decided by a machine $\mathbb{A}$ in time $O(n^{2c})$ and space $O(n^{o(1)})$. Let $e \geq 1$ be an arbitrary natural. Every configuration of $\mathbb{A}$ on $x \in \{0,1\}^n$ can be coded by a string of length $\leq n^{1/e}$ (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 $c_0 := c_{\mathrm{start}}$, 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 to run in time $O(n^{c+2/e})$. This needs a little care. First, to guess c_i the machine needs to know $\lfloor n^{1/e} \rfloor$: guess an verify this number in nondeterministic 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. This costs $O(n)$ time once. Then the whole simulation needs time $O(n + n^{2/e} \cdot n^c)$. □

Proof of Theorem 5.6.2: Assume $\mathrm{NTIME}(n) \subseteq \mathrm{TISP}(n^a, n^{o(1)})$ for some rational $\sqrt{2} > a \geq 1$. Let $c \in \mathbb{N}$ be "large enough" and such that $(2c-1)/a \geq 1$ is integer. Then

$$\begin{array}{rcll} \mathrm{NTIME}(n^{(2c-1)/a}) & \subseteq & \mathrm{TISP}(n^{2c}, n^{o(1)}) & \text{by Lemma 5.6.3} \\ & \subseteq & \Sigma_2\mathrm{TIME}(n^{c+(1/a)}) & \text{by Lemma 5.6.5} \\ & \subseteq & \mathrm{NTIME}(n^{ac+1}) & \text{by Lemma 5.6.4.} \end{array}$$

By the Nondeterministic Time Hierarchy Theorem in the strong form of Remark 3.3.4, 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 $\frac{2c-1}{c+1} \to_c 2 > a^2$. □

Remark 5.6.6 The bound $a < \sqrt{2} \approx 1.41$ in Theorem 5.6.2 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 $\mathrm{SAT} \notin \mathrm{L}$, and thus $\mathrm{NP} \neq \mathrm{L}$.

Chapter 6

Size

6.1 Non-uniform polynomial time

Definition 6.1.1 The set P/poly contains the problems Q such that there exists a polynomial p and a sequence of Boolean circuits $(C_n)_{n\in\mathbb{N}}$ such that each C_n has size $\leq p(n)$ and computes $\chi_Q \restriction \{0,1\}^*$; this means, $C_n = C_n(X_1, \ldots, X_n)$ has n variables, at most $p(n)$ many gates and for all $x \in \{0,1\}^*$

$$x \in Q \iff C_{|x|}(x) = 1.$$

Remark 6.1.2 Clearly P $\subseteq$ P/poly by Lemma 2.4.7. The converse is false since P/poly contains undecidable problems. In fact, if $Q \subseteq \{1\}^*$ (i.e. Q is tally), then $Q \in$ P/poly: if $1^n \in Q$, then C_n computes the conjunction of $X_1, \ldots, X_n$; otherwise it just outputs 0.

It is generally believed that NP $\not\subseteq$ P/poly but currently even NEXP $\not\subseteq$ P/poly is unknown. The class P/poly should be thought of as a non-uniform version of P (recall Lemma 2.4.7). We now give a machine characterization.

Definition 6.1.3 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 called *polynomially bounded* if there is a polynomial p such that $|a(n)| \leq p(n)$ for all $n \in \mathbb{N}$.

Recall Definitions 3.7.8 and 5.5.1.

Proposition 6.1.4 *Let Q be a problem. The following are equivalent.*

1. $Q \in$ P/poly.
2. *There is a polynomial time bounded Turing machine $\mathbb{A}$ with polynomially bounded advice $a : \mathbb{N} \to \{0,1\}^*$ that decides Q.*
3. *There are $R \subseteq (\{0,1\}^*)^2$ in* P *and a polynomially bounded $a : \mathbb{N} \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$*
$$x \in Q \iff (x, a(|x|)) \in R.$$
4. *There is a sparse $O \subseteq \{0,1\}^*$ such that $Q \in \mathrm{P}^O$.*

Proof: We first show that (1)-(3) are equivalent. To see (1) implies (2), define $a(n)$ to be the code of C_n and let $\mathbb{A}$ on x evaluate $C_{|x|}$ on x. To see (2) implies (3), set $R := \{(x, y) \mid \mathbb{A} \text{ accepts } \langle x, y\rangle\}$. To see (3) implies (1) use Lemma 2.4.7 to obtain for $n, m \in \mathbb{N}$ a circuit $D_{n,m}(\bar{X}, \bar{Y})$ of size $(n+m)^{O(1)}$ and variables $\bar{X} = X_1 \cdots X_n$ and $\bar{Y} = Y_1 \cdots Y_m$ such that for all $(x, y) \in \{0,1\}^n \times \{0,1\}^m$

$$(x, y) \in R \Longleftrightarrow D_{n,m}(x, y) = 1.$$

Then define $C_n(\bar{X}) := D_{n,|a(n)|}(\bar{X}, a(n))$.

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

(2) implies (1): choose $\mathbb{A}, a$ according (2). It suffices to find a sparse O such that $1^n \mapsto a(n)$ can be computed in polynomial time with oracle O. We can assume that $|a(n)| = n^c$ for some $c \in \mathbb{N}$ and large enough n. Then let O contain for $1 \leq i \leq n^c$ 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 6.1.5 Show that P/poly is $\leq_p$-closed.

6.1.1 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 3.8.2.

Exercise 6.1.6 If SAT $\in$ P/poly, then there is a polynomial time machine $\mathbb{A}$ with polynomially bounded advice that when given a satisfiable formula outputs a satisfying assignment.

Theorem 6.1.7 (Karp-Lipton 1980) *If* NP $\subseteq$ P/poly*, then* PH $= \Sigma_2^{\mathrm{P}}$.

Proof: Assume SAT $\in$ P/poly and choose $\mathbb{A}$ according to the above exercise, say, $\mathbb{A}$ uses advice $a : \mathbb{N} \to \{0,1\}^*$ satisfying $a(m) \leq m^c$ for all large enough m. View $\mathbb{A}$ as a usual polynomial time machine (run on $\langle x, a(|x|)\rangle$ instead x). Note that a formula α is satisfiable if and only if the output $\mathbb{A}(\langle \alpha, a(|\alpha|)\rangle)$ is a satisfying assignment of it.

By Theorem 5.3.5 it suffices to show $\Sigma_2^{\mathrm{P}} = \Pi_2^{\mathrm{P}}$, or equivalently (apply co-), $\Pi_2^{\mathrm{P}} \subseteq \Sigma_2^{\mathrm{P}}$. Theorem 5.3.9 implies that Π_2SAT is Π_2^{P}-complete. Hence, by Proposition 5.4.2, it suffices to give a polynomial time 2-alternating Turing machine accepting Π_2SAT.

Π_2SAT	
Instance:	a quantified Boolean sentence α of the form $\forall \bar{X} \exists \bar{Y} \beta$ with β quantifier free.
Problem:	is α true?

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

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

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 $a := a(m) \leq n^c$. □

6.2 Shannon's theorem and size hierarchy

Theorem 6.2.1 (Shannon 1949) *Let $Sh : \mathbb{N} \to \mathbb{N}$ be defined by*

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

Then $Sh(n) = \Theta(2^n/n)$. Moreover, for every large enough n there exists a function $f : \{0,1\}^n \to \{0,1\}$ such that every circuit computing f has size bigger than $2^n/(4n)$.

Proof: $Sh(n) \leq O(2^n/n)$ follows from Proposition 2.4.6 and $Sh(n) \geq \Omega(2^n/n)$ follows from the second statement. For the second statement we show for "small" s there are strictly less than 2^{2^n} many functions $f : \{0,1\}^n \to \{0,1\}$ which are computable by circuits of size $\leq s$. We use numbers 1 to s as gates, and reserve the first n gates as input gates; then a size $\leq s$ circuit is specified once we specify for each gate $n < i \leq s$ its label $\wedge, \vee, \neg$ or $*$ plus the at most 2 gates it receives inputs from; we use label $*$ to indicate that the gate does not occur in the circuit to be specified. This gives $\leq 4 \cdot s^2$ choices per gate, so $\leq (4s^2)^s \leq s^{3s}$ many circuits (for $1 < n \leq s$). But if $s \leq 2^n/(4n)$ then

$$\log s^{3s} = 3s \cdot \log s \leq \frac{3 \cdot 2^n \cdot (n - \log n - 2)}{4n} \leq \frac{3}{4} \cdot 2^n < 2^n,$$

and hence $s^{3s} < 2^{2^n}$. □

Remark 6.2.2 The vast majority of functions $f : \{0,1\}^n \to \{0,1\}$ is not computable by circuits of size $\leq s := \lfloor 2^n/(4n) \rfloor$. Indeed, if we choose $f : \{0,1\}^n \to \{0,1\}$ uniformly at random, then it is computed by such a circuit only with probability $\leq 2^{s^{3s}}/2^{2^n} \leq 2^{-\frac{1}{4} \cdot 2^n}$.

Definition 6.2.3 Let $s : \mathbb{N} \to \mathbb{N}$. A problem Q *has circuits of size s* if there is a family $(C_n)_n$ of Boolean circuits such that for large enouigh n, C_n has size $\leq s(n)$ and computes $\chi_Q \restriction \{0,1\}^n$. The class of these problems is denoted $\mathrm{SIZE}(s)$.

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

Theorem 6.2.4 (Size hierarchy) *For all $s, s' : \mathbb{N} \to \mathbb{N}$ nondecreasing and unbounded with*

$$336 \cdot s'(n) \leq s(n) \leq 2^n/n$$

we have $\mathrm{SIZE}(s) \setminus \mathrm{SIZE}(s') \neq \emptyset$.

Proof: By the proof of Proposition 2.4.6 for large enough ℓ, every function $f : \{0,1\}^\ell \to \{0,1\}$ is computable in size $21 \cdot 2^\ell/\ell$. Our constant is $336 = 2^4 \cdot 21$. Some $f_\ell : \{0,1\}^\ell \to \{0,1\}$ is not computable in size $2^\ell/4\ell$. For large enough n and suitable $\ell \leq n$ we show that the function $g_n : \{0,1\}^n \to \{0,1\}$ defined by $g_n(x_1 \cdots x_n) := f_\ell(x_1 \cdots x_\ell)$ is computable in size $s(n)$ but not in size $s'(n)$. Then $Q \in \mathrm{SIZE}(s) \setminus \mathrm{SIZE}(s')$ for the Q with $\chi_Q \restriction \{0,1\}^n = g_n$ for each n.

We are thus left to find a natural $\ell \leq n$ such that $2^\ell/(4\ell) \geq s'(n)$ and $21 \cdot 2^\ell/\ell \leq s(n)$. For large enough n, this holds if ℓ is in the real intervall $[a, b]$ for

$$\begin{aligned} a &:= \log s'(n) + \log\log s'(n) + 3; \\ b &:= \log s(n) + \log\log s(n) - \log 21. \end{aligned}$$

Indeed, $\ell \leq n$ by $\ell \leq b$ and assumption $s(n) \leq 2^n/n$. Note that as functions in ℓ both $2^\ell/(4\ell)$ and $21 \cdot 2^\ell/\ell$ are non-decreasing for large enough ℓ. Further note that $a \leq 2\log s'(n)$ and $b \geq \log s(n)$ for large enough n. Hence for $\ell \in [a, b]$

$$2^\ell/(4\ell) \geq 2^a/(4a) \geq \frac{s'(n) \cdot \log s'(n) \cdot 2^3}{4 \cdot 2\log s'(n)} = s'(n);$$
$$21 \cdot 2^\ell/\ell \leq 21 \cdot 2^b/b \leq \frac{21 \cdot s(n) \cdot \log s(n) \cdot \frac{1}{21}}{\log s(n)} = s(n).$$

Since $\log s(n) \geq \log s'(n) + \log 21 + 4$, so $b - a \geq 1$, so there exists a natural $\ell \in [a, b]$. □

6.2.1 Kannan's theorem

Theorem 6.2.5 (Kannan 1981) *For every $c \in \mathbb{N}$, $\Sigma_2^{\mathrm{P}} \not\subseteq \mathrm{SIZE}(n^c)$.*

Proof: Let $c \in \mathbb{N}$. It suffices to show $\mathrm{PH} \not\subseteq \mathrm{SIZE}(n^c)$. Then argue for the theorem by distinguishing two cases: if $\mathrm{NP} \subseteq \mathrm{P/poly}$, then $\mathrm{PH} = \Sigma_2^{\mathrm{P}}$ by Theorem 6.1.7 and we are done; otherwise, $\Sigma_2^{\mathrm{P}} \setminus \mathrm{SIZE}(n^c) \supseteq \mathrm{NP} \setminus \mathrm{P/poly} \neq \emptyset$ and we are done.

By the size hierarchy theorem there are, for large enough n, circuits of size n^{c+1} computing functions $f : \{0,1\}^n \to \{0,1\}$ which are not computable by circuits of size n^c. Viewed as a binary string (via some fixed encoding), one of these circuits is lexicographically minimal. Let $f_n : \{0,1\}^n \to \{0,1\}$ denote the function it computes. Define Q to contain those x with $f_{|x|}(x) = 1$. Clearly, $Q \notin \mathrm{SIZE}(n^c)$, so we are left to show $Q \in \mathrm{PH}$.

Consider the following alternating machine $\mathbb{A}$. On input x it existentially guesses a circuit of size $\leq n^{c+1}$ (with n inputs) and checks $C(x) = 1$. It then verifies that the function computed by C is not computed by a circuit C' of size $\leq n^c$; this verification is done by universally guessing such C', existentially guessing $y \in \{0,1\}^n$ and checking $C(y) \neq C'(y)$. Finally, $\mathbb{A}$ universally guesses a size n^{c+1} circuit D and does the following: if D is lexicographically smaller than C, then it verifies that there exists a size n^c circuit D' computing the same function as D; this verification is done by existentially guessing such D', universally guessing $y \in \{0,1\}^n$ and checking $D(y) = D'(y)$.

Then $\mathbb{A}$ accepts Q, runs in polynomial time and is 6-alternating. Hence $Q \in \Sigma_6\mathrm{P} \subseteq \mathrm{PH}$ by Proposition 5.4.2. □

6.3 Hastad's Switching Lemma

In this section we prove an exponential lower bound on the size of certain special circuits computing the parity function *par* mapping a binary string to the parity of its number of 1's; the restriction of *par* to $\{0,1\}^n$ is denoted par_n, i.e. $par_n(x_1 \cdots x_n) = \sum_{i \in [n]} x_i \mathrm{mod} 2$.

The circuits we have in mind are so-called *d-alternating* formulas, namely formulas of the form $\bigwedge_{i_1} \bigvee_{i_2} \cdots Q_{i_d} \lambda_{i_1 \cdots i_d}$ for literals $\lambda_{i_1 \cdots i_d}$ where $Q = \bigwedge$ if d is odd and $Q = \bigvee$ else. We say a Boolean formula $\alpha(X_1, \ldots, X_n)$ *computes* the function from $\{0,1\}^n$ to $\{0,1\}$ mapping $x_1 \cdots x_n$ to the truth value of α under the assignment $\{(X_i, x_i) \mid i \in [n]\}$.

Theorem 6.3.1 *Let $d \in \mathbb{N}$. There is $\delta > 0$ such that for all large enough n every d-alternating formula computing the parity function par_n has at least 2^{n^δ} many subformulas.*

If $\alpha = \alpha(X_1, \ldots, X_n)$ is a formula and A is a partial assigment to $\{X_1, \ldots, X_n\}$ write $\alpha \restriction A$ for the formula obtained by substituting the constant $A(X)$ for every $X \in dom(A)$. The proof of the above theorem is based on the so-called Switching Lemma which states, roughly, for a DNF α and a random A the formula $\alpha \restriction A$ is probably "far simpler" than α, simpler in the sense that it can be computed by a shallow decision tree:

Definition 6.3.2 A *decision tree* is a pair $\mathfrak{T} = (T, \lambda)$ such that

- $T \subseteq \{0,1\}^*$ is a finite set of *nodes*, closed under prefixes and whenever $\pi \in T$ then either both $\pi 0, \pi 1$ or none is in T; in the latter case π is called a *leaf*.
- λ maps each leaf into $\{0,1\}$ and each other $\pi \in T$ into $[n]$ for some $n \in \mathbb{N}$.

The *height* of $\mathfrak{T}$ is $\max_{\pi \in T} |\pi|$ and denoted *height*$(\mathfrak{T})$. For $n \in \mathbb{N}$ as above, each string $x = x_1 \cdots x_n \in \{0,1\}^n$ determines a leaf $\mathfrak{T}(x)$, namely the unique leaf $\pi = \pi_1 \cdots \pi_{|\pi|} \in T$ satisfying $\pi_i = x_{\lambda(\pi_1 \cdots \pi_{i-1})}$ for all $i \in [|t|]$. The decision tree *computes* the function $x \mapsto \lambda(\mathfrak{T}(x))$.

Each $\pi = \pi_1 \cdots \pi_{|\pi|} \in T$ *corresponds* to the partial assignment $\widehat{\pi}$ mapping $X_{\lambda(\pi_1 \cdots \pi_{i-1})}$ to π_i for each $0 < i < |\pi|$. Finally, we say $\mathfrak{T}$ is *for* α if it computes the same function as α.

Example 6.3.3 Every $\alpha(X_1, \ldots, X_n)$ has a decision tree, namely the *naive tree for* α of height n which has leafs corresponding to all assignments to the variables appearing in α. More precisely, if α does not contain variables then take the one node tree whose unique leaf is labeled by the truth value of α; otherwise $s \geq 1$ many variables appear in α, say $X_{i_0}, \ldots, X_{i_{s-1}}$ with $i_0 < \cdots i_{s-1}$; set $T := \{0,1\}^{\leq s}$ and let λ map each $\pi \in T$ with $|\pi| < s$ to $i_{|\pi|}$, and each leaf π to the truth value of α under A_π.

Example 6.3.4 Every decision tree computing par_n has height at least n.

Proof: Assume $\mathfrak{T}$ has height $< n$ and computes par_n. Let $x \in \{0,1\}^n$, set $\pi := \mathfrak{T}(x)$, choose $i \in [n]$ such that $X_i \notin dom(\widehat{\pi})$, i.e. i is not the label of a prefix of π, and consider x', the string obtained from x by flipping its ith bit. Then Then $\mathfrak{T}(x) = \mathfrak{T}(x')$, so the function computed by $\mathfrak{T}$ takes the same value on x and x' – but $par_n(x) \neq par_n(x')$. □

Remark 6.3.5 Assume a formula $\alpha(X_1, \ldots, X_n)$ *has* a decision tree of height $\leq k$ (i.e. the function computed by α is computed by a decision tree of height $\leq k$). Then α is equivalent both to a k-DNF and to a k-CNF. Indeed, letting π rangs over leafs labelled 1, and σ over the leafs labelled 0, α is equivalent to

$$\bigvee_\pi \bigwedge_{X \in dom(\widehat{\pi})} \neg^{1-\widehat{\pi}(X)} X \qquad \text{and} \qquad \bigwedge_\sigma \bigvee_{X \in dom(\widehat{\sigma})} \neg^{\widehat{\sigma}(X)} X,$$

where $\neg^0 X$ ist X and $\neg^1 X$ is $\neg X$.

Definition 6.3.6 Let $\alpha = \alpha(X_1, \ldots, X_n)$ be a DNF. Write $\alpha = t_1 \vee \ldots \vee t_\ell$ for terms t_i and $\ell \in \mathbb{N}$. The *canonical decision tree* $\mathfrak{T}(\alpha)$ *for* α is defined by induction on ℓ. For $\ell = 1$, take the naive tree for t_1. For $\ell > 1$ let $\mathfrak{T}(\alpha)$ be obtained from the naive tree for t_1 by *attaching* to every leaf π labelled 0 the tree $\mathfrak{T}(\alpha \restriction \widehat{\pi}) = \mathfrak{T}((t_2 \vee \cdots \vee t_\ell) \restriction \widehat{\pi})$. More precisely: for each such π add nodes $\pi\sigma$ where σ is a node in $\mathfrak{T}(\alpha \restriction \widehat{\pi})$; the label of such a node is the one of σ in $\mathfrak{T}(\alpha \restriction \widehat{\pi})$; all other nodes get the label they have in the naive tree for t_1.

Theorem 6.3.7 (Hastad's Switching Lemma) *Let $k, m, n, s \in \mathbb{N}$ with $m < n, 0 < s$ and $\alpha = \alpha(X_1, \ldots, X_n)$ be a k-DNF. Let R be a random variable uniformly distributed in the set R_n^m of all partial assigments ρ of $\{X_1, \ldots, X_n\}$ with $|dom(\rho)| = m$. Then*

$$\Pr\left[height(\mathfrak{T}(\alpha \restriction R)) \geq s\right] < \left(12 \cdot k \cdot \frac{n-m}{m}\right)^s$$

Proof: Write $\alpha = t_1 \vee t_2 \vee \ldots$. Call $\rho \in R_n^m$ *bad* if $height(\mathfrak{T}(\alpha \restriction \rho)) \geq s$. For bad ρ fix some leaf π in $\mathfrak{T}(\alpha \restriction \rho)$ of length $|\pi| > s$. Using π we give a succinct code of ρ:

(a) let i_1 be minimal such that $t_{i_1} \restriction \rho \not\equiv 0$; such i_1 exists as otherwise $\mathfrak{T}(\alpha \restriction \rho)$ would be the one node tree labelled 0.

(b) let π_1 be the initial segment of π such that $\widehat{\pi}_1$ is $\widehat{\pi}$, restricted to the variables in $t_{i_1} \restriction \rho$;

(c) let σ_1' be the unique satisfying assignment of $t_{i_1} \restriction \rho$ (defined on precisley its variables);

(c.1) if $\pi_1 \neq \pi$, set $\sigma_1 := \sigma_1'$; note in this case π_1 is defined on all variables of $t_{i_1} \restriction \rho$ and we have $t_{i_1} \restriction \rho\widehat{\pi}_1 \equiv 0$ and $t_{i_1} \restriction \rho\sigma_1 \equiv 1$; here e.g. $\rho\widehat{\pi}_1$ is shorthand for $\rho \cup \widehat{\pi}_1$.

(c.2) if $\pi_1 = \pi$, let σ_1 be the restriction of σ_1' to $dom(\widehat{\pi}_1)$; note in this case $t_{i_1} \restriction \rho\sigma_1 \not\equiv 0$;

(d) let $s^1 \in \{0,1\}^k$ have jth bit equal to 1 or 0 depending on whether or not the jth variable in t_{i_1} is in $dom(\sigma_1)$;

(e) let $r^1 \in \{0,1\}^{|dom(\sigma_1)|}$ have jth bit 1 or 0 depending on whether or not σ_1 and $\widehat{\pi}_1$ agree on the jth variable in $dom(\sigma_1)$.

We defined $(i_1, \pi_1, \sigma_1, s^1, r^1)$ from ρ, π. In case (c.1), $\pi \setminus \pi_1$ is a path in $\mathfrak{T}(\alpha \restriction \rho\pi_1)$ and we can define $(i_2, \pi_2, \sigma_2, s^2, r^2)$ from $\rho\pi_1, \pi \setminus \pi_1$. We continue like this until, say after ℓ rounds, case (c.2) happens. Then we code ρ by the tuple

$$(\rho\sigma_1 \cdots \sigma_\ell, s^1 \cdots s^\ell, r^1 \cdots r^\ell).$$

We show how to find ρ given its code. Assume we already found the indices $i_1, \ldots, i_{j-1}$, segments $\pi_1, \ldots \pi_{j-1}$, assignments $\sigma_1, \ldots, \sigma_{j-1}$, as well as $\rho\widehat{\pi}_1 \cdots \widehat{\pi}_{j-1}\sigma_j \cdots \sigma_\ell$. We show how to find these objects for j instead $j-1$. Namely, i_j is minimal such that $t_{i_j} \restriction \rho\widehat{\pi}_1 \cdots \widehat{\pi}_{j-1} \neq 0$, i.e. $t_{i_j} \restriction \rho\widehat{\pi}_1 \cdots \widehat{\pi}_{j-1}\sigma_j \cdots \sigma_\ell \neq 0$. But i_j and s^j (available from the code) determine σ_j. Using $r^1 \cdots r^\ell$ we then find π_j, and then $\rho\widehat{\pi}_1 \cdots \widehat{\pi}_j\sigma_{j+1} \cdots \sigma_\ell$

The number of bad ρ is bounded by the number of their codes. For such a code we have $\rho\sigma_1 \cdots \sigma_\ell \in R_n^{m+s}$, $s^1 \cdots s^\ell \in \{0,1\}^{k \cdot \ell}$ and $r^1 \cdots r^\ell \in \{0,1\}^s$. Note $\ell \leq s$ and the string $s^1 \cdots s^\ell$ contains exactly s many 1s, hence there are at most $\binom{ks}{s}$ such strings. So our probability is

$$\frac{|\{\rho \in R_n^m \mid \rho \text{ is bad}\}|}{|R_n^m|} \leq \frac{|R_n^{m+s}|}{|R_n^m|} \cdot \binom{ks}{s} \cdot 2^s.$$

First observe

$$\frac{|R_n^{m+s}|}{|R_n^m|} = \frac{\binom{n}{m+s} \cdot 2^{m+s}}{\binom{n}{m} \cdot 2^m} = 2^s \cdot \frac{n! \cdot (n-m)! \cdot m!}{(n-m-s)! \cdot (m+s)! \cdot n!} \leq 2^s \cdot \frac{(n-m)^s}{m^s}.$$

Second, recall $\binom{ks}{s} \leq (e \cdot \frac{ks}{s})^s = e^s \cdot k^s$. Thus, our probability is $\leq (4 \cdot e \cdot k \cdot \frac{(n-m)}{m})^s$. $\square$

Proof of Theorem 6.3.1: For contradiction, assume that for arbitrarily large n parity is computed by a d-alternating formula $\alpha(X_1, \dots, X_n)$ with $< S := 2^{n^\delta}$ many subformulas where $\delta < 2^{-d}$. Here and in the following, we mean $\lfloor n^\epsilon \rfloor$ when we write n^ϵ. Write

$$\alpha = \neg \bigvee_{i_d} \neg \bigvee_{i_{d-1}} \cdots \neg \bigvee_{i_1} \lambda_{i_1 \cdots i_d}$$

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 $t \leq d$ there exists $\rho_t \in R_n^{m_t}$ 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 $\beta_{i_d \cdots i_{d-t-1}} \restriction \rho_t$ has a decision tree of height $< \log S = n^\delta$.

This contradicts Example 6.3.4: both $\alpha \restriction \rho_d$ and $\neg\alpha \restriction \rho_d$ have a decision tree of height $< n^\delta$; but one of these formulas computes parity on $n^{2^{-d}} \geq n^\delta$ many variables ($\leq$ instead $<$ due to the rounding). We are left to prove the claim.

We assume to have found ρ_t and look for ρ_{t+1}. Note $\beta_{i_d \cdots i_{d-t-2}} \restriction \rho_t$ is the negation of $\bigvee_{i_{t+1}} \beta_{i_d \cdots i_{d-t-1}} \restriction \rho_t$. By Remark 6.3.5 the latter is equivalent to an n^δ-DNF. It has $n^{2^{-t}}$ many variables. By Theorem 6.3.7 the probability that a random $\rho \in R_{n'}^{m'}$ for $m' := n^{2^{-t}} - n^{2^{-t-1}}$ and $n' := n^{2^{-t}}$ does not simplify this formula is at most

$$(12 \cdot n^\delta \cdot (n' - m')/m')^{\log S} \leq (13 \cdot n^\delta \cdot n^{-2^{-t-1}})^{\log S}$$

where we estimate $(n' - m')/m' \leq \frac{13}{12} n^{-2^{-t-1}}$ for large enough n. For large enough n, this probability is $< 1/S$ because $n^\delta \cdot n^{-2^{-t-1}} \leq o(1)$. Then there exists a single ρ that simplifies all $\beta_{i_d \cdots i_{d-t-2}} \restriction \rho_t$ simultaneously. Set $\rho_{t+1} := \rho_t \cup \rho$ and note $\rho_{t+1} \in R_n^{m_{t+1}}$ since it evaluates exactly $m_t + (n^{2^{-t}} - n^{2^{-t-1}}) = m_{t+1}$ many variables. □

Chapter 7

Randomness

7.1 How to evaluate an arithmetical circuit

We need some notation for polynomials. We write polynomials $p \in \mathbb{Z}[X_1, \ldots, X_\ell]$ as formal sums

$$p = \sum_{(j_1,\ldots,j_\ell)\in\mathbb{N}^\ell} a_{(j_1,\ldots,j_\ell)} X_1^{j_1} \cdots X_\ell^{j_\ell},$$

with only finitely many of the $a_{(j_1,\ldots,j_\ell)} \in \mathbb{Z}$ not equal to 0. A polynomial is *nonzero* if not all of the $a_{(j_1,\ldots,j_\ell)}$s are 0. By $p^{|\cdot|}$ we denote the function computed by the formal expression

$$\sum_{(j_1,\ldots,j_\ell)\in\mathbb{N}^\ell} |a_{(j_1,\ldots,j_\ell)} X_1^{j_1} \cdots X_\ell^{j_\ell}|$$

in the obvious way. The *(total) degree* of p is

$$\deg(p) := \max\{j_1 + \cdots + j_\ell \mid a_{(j_1,\ldots,j_\ell)} \neq 0\}.$$

Definition 7.1.1 An *arithmetical circuit* is obtained from a Boolean circuit with one output gate by relabeling $\wedge, \vee, \neg$ by $\times, +, -$; and- and or-gates are now called *multiplication-* and *addition-gates*. Such a circuit, say with variables $\bar{X}$, computes in the obvious way a polynomial

$$p_C(\bar{X}) \in \mathbb{Z}[\bar{X}],$$

namely, multiplication- and addition-gates compute the product respectively the sum of their two inputs and gates labeled $-$ compute the unary additive inverse. The *depth* of a circuit is the length of the longest path in it (as a directed graph).

The proof of the following lemma is straightforward and left as an exercise.

Lemma 7.1.2 *Let $C(X_1, \ldots, X_\ell)$ be an arithmetical circuit of depth at most d with at most m multiplication gates on any path. Then $\deg(p_C) \leq 2^m$ and for all $a_1, \ldots, a_\ell \in \mathbb{Z}^\ell$*

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

Exercise 7.1.3 The polynomial $X^{2^k} \in \mathbb{Z}[X]$ is computed by a circuit with $k+1$ gates.

Hence, values computed by a circuit may have doubly exponential size, so their binary code may be of exponential size. It is therefore unclear whether the *arithmetical circuit evaluation problem*

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

is in P. In fact, this is an open question. The naive algorithm that computes bottom-up the value computed at each gate needs exponential time. Even ACE $\in$ NP is not obvious - but here is a clever idea: we have $b := |C(\bar{a})| \leq u^{d \cdot 2^m}$ where $u := \max\{2, \max_{i \in [\ell]} |a_i|\}$ and d, m are as in the lemma above. Then b has at most $\log b \leq d2^m \log u \leq 2^{2n}$ many prime divisors where we assume that n, the size of the input $(C, \bar{a})$, it large enough. By the Prime Number Theorem we can assume that there are at least $2^{10n}/(2 \cdot 10n)$ many primes below 2^{10n}. So let us guess a number $r < 2^{10n}$ by guessing $10n$ bits. Choosing the bits uniformly at random gives a prime with probability at least $1/(2 \cdot 10n)$, and in this case it is different from a prime divisor of b with probability at least

$$1 - \frac{2^{2n}}{2^{10n}/(2 \cdot 10n)} \geq \frac{1}{2}$$

for large enough n. It follows that there exists $r < 2^{10n}$ such that

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

But b mod r can be computed by evaluating the circuit C on $\bar{a}$ doing all operations modulo r. Each operation can be computed time polynomial in $\log r \leq O(n)$. This shows 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 $1/(2 \cdot 10n) \cdot (1/2) = 1/(40n)$ 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 may be exponentially small, if the formula has exactly one satisfying assignment it is 2^{-n}. So, if our algorithm for ACE does its guesses randomly, it answers correctly with considerable probability. This is the first idea. The second idea is, that we may run it independently for several times to boost this probability.

The next section makes these two ideas precise. It seems thus fair to consider ACE a "tractable" problem, albeit we do not know of 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 7.3.2).

7.2 Randomized computations

We define randomized computations following the first idea sketched in the previous section. 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 be as desired 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 $Q \subseteq \{0,1\}^*$. For $m \in \mathbb{N}$ let U_m denote a random variable that is uniformly distributed in $\{0,1\}^m$. Being uniformly distributed means $\Pr[U_m = y] = 2^{-m}$ for all $y \in \{0,1\}^m$. Here and in the following we shall always use Pr to refer to the probability measure underlying a random variable.

Definition 7.2.1 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 $y \in \{0,1\}^{t(|x|)}$. Define the random variable $\mathbb{A}(x)$ to be the output of $\mathbb{A}$ on x of the run determined by $U_{t(|x|)}$.

The set BPTIME(t) contains the problems Q such that there exist $c \in \mathbb{N}$ and a $c \cdot t$-time bounded nondeterministic Turing machine $\mathbb{A}$ such that for all $x \in \{0,1\}^*$

$$\Pr[\mathbb{A}(x) \neq \chi_Q(x)] \leq 1/4$$

The set 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 $x \notin Q$.

$$\begin{array}{lcl} \text{BPP} & := & \bigcup_{c\in\mathbb{N}} \text{BPTIME}(n^c), \\ \text{RP} & := & \bigcup_{c\in\mathbb{N}} \text{RTIME}(n^c). \end{array}$$

Remark 7.2.2 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 $U_{t(|x|)}$ or using $U_{t'(|x|)}$. E.g. $\Pr[\mathbb{A}(x) \neq \chi_Q(x)]$ is the same for both choices. This is why we supress t from the notation $\mathbb{A}(x)$.

Clearly, P $\subseteq$ RP $\subseteq$ NP. It is not known whether BPP $\subseteq$ NP or NP $\subseteq$ BPP.

Remark 7.2.3 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 the outcome of a single coin toss.

Exercise 7.2.4 BPP $\subseteq$ EXP (simulate a probabilistic machine for all its random choices).

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. Let $\mathbb{N}[X]$ denote the set of polynomials with natural coefficients.

Proposition 7.2.5 *Let Q be a problem. The following are equivalent.*

1. $Q \in$ RP.
2. *There are a polynomial time probabilistic Turing machine $\mathbb{A}$ with one-sided error and $p \in \mathbb{N}[X]$ such that $\Pr[\mathbb{A}(x) \neq \chi_Q(x)] \leq 1 - 1/p(|x|)$ for all $x \in \{0,1\}^*$.*
3. *For all $q \in \mathbb{N}[X]$ there is a polynomial time probabilistic Turing machine $\mathbb{B}$ with one-sided error such that $\Pr[\mathbb{B}(x) \neq \chi_Q(x)] \leq 2^{-q(|x|)}$ for all $x \in \{0,1\}^*$.*

Proof: Clearly, it suffices to show that (2) implies (3). Choose $\mathbb{A}$ and p according (2) and let $q \in \mathbb{N}[X]$ be given. We can assume that p is not constant. 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. Clearly, 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

$$(1 - 1/p(|x|))^{p(|x|)\cdot q(|x|)} \leq ((1 - 1/p(|x|))^{p(|x|)})^{q(|x|)} \leq (1/2)^{q(|x|)},$$

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 following probabilistic lemma, whose proof we omit. Recall, a *Bernoulli variable* is a random variable with values in $\{0, 1\}$.

Lemma 7.2.6 (Chernoff Bound) *Let $\ell \in \mathbb{N}$ and assume $X_1, \ldots, X_\ell$ are mutually independent, identically distributed Bernoulli variables. Then for every real $\epsilon > 0$*

$$\Pr\left[|\tfrac{1}{\ell}\textstyle\sum_{i=1}^{\ell} X_i - \mathbb{E}[X_1]| \geq \epsilon\right] < 2e^{-2\epsilon^2\cdot\ell}.$$

Proposition 7.2.7 *Let Q be a problem. The following are equivalent.*

1. *$Q \in$ BPP.*
2. *There is a polynomial time probabilistic Turing machine $\mathbb{A}$ and a positive $p \in \mathbb{N}[X]$ such that $\Pr[\mathbb{A}(x) \neq \chi_Q(x)] \leq 1/2 - 1/p(|x|)$ for all $x \in \{0, 1\}^*$.*
3. *For all $q \in \mathbb{N}[X]$ there is a polynomial time probabilistic Turing machine $\mathbb{B}$ such that $\Pr[\mathbb{B}(x) \neq \chi_Q(x)] \leq 2^{-q(|x|)}$. for all $x \in \{0, 1\}^*$.*

Proof: It suffices to show (2) implies (3). Choose $\mathbb{A}, p$ according to (2) and let $q \in \mathbb{N}[X]$ be given. We can assume $\mathbb{A}$ always outputs 0 or 1 (so $\mathbb{A}(x)$ is Bernoulli) and p is not constant. 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 $\hat{\mu} := \frac{1}{\ell}\sum_{i=1}^{\ell} b_i \geq 1/2$ 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, $\mu \geq 1/2 + 1/p(|x|)$ if $x \in Q$, and $\mu \leq 1/2 - 1/p(|x|)$ if $x \notin Q$. Thus, $\mathbb{B}$ errs only if $|\hat{\mu} - \mu| \geq 1/p(|x|)$. By the Chernoff bound this has probability $\leq 2e^{-\ell\cdot 2/p(|x|)^2}$. For suitable $r \in \mathbb{N}[X]$ and $\ell := r(|x|)$ this is $\leq 2^{-q(|x|)}$ and $\mathbb{B}$ runs in polynomial time. □

Exercise 7.2.8 Show that coBPP = BPP. Define BPP^{BPP} and show that it equals BPP.

The most famous problem in RP that is not known to be in P is *polynomial identity testing*, namely to decide whether two given arithmetical circuits compute the same polynomial. Clearly, this tantamount to decide whether a given circuit computes a nonzero polynomial:

PIT	
Instance:	an arithmetical circuit C.
Problem:	is p_C nonzero?

We need the following algebraic lemma, whose proof we omit.

Lemma 7.2.9 (Schwarz-Zippel) *Let $N, \ell, d \in \mathbb{N}$ and $p \in \mathbb{Z}[X_1, \ldots, X_\ell]$ nonzero of degree at most d. If $R_1, \ldots, R_\ell$ are mutually independent random variables uniformly distributed in $\{0, \ldots, N-1\}$, then*

$$\Pr[p(R_1, \ldots, R_\ell) = 0] \leq d/N.$$

Theorem 7.2.10 PIT $\in$ RP.

Proof: Given $C(X_1, \ldots, X_\ell)$ of size n and depth d, set $N := 10n$, and guess $\ell+1$ many strings in $\{0,1\}^N$; these encode numbers $a_1, \ldots, a_\ell, r < 2^N = 2^{10n}$. Compute $C(a_1, \ldots, a_\ell) \bmod r$ in time polynomial in $\log r \leq 10n$. Accept if you the outcome os $\neq 0$ and reject otherwise.

Clearly, if $p_C = 0$, then you reject for sure. Otherwise, $p(a_1, \ldots, a_\ell) \neq 0$ with probability at least $1 - 2^d/2^{N+1} \geq 1/2$ by the Schwarz-Zippel Lemma ($\deg(p_C) \leq 2^d$ by Lemma 7.1.2). Then $p_C(a_1, \ldots, a_\ell) \bmod r \neq 0$ with probablity at least $1/(40n)$ as seen in Section 7.1. In total, you accept with probability at least $1/(80n)$. Hence PIT $\in$ RP by Proposition 7.2.5. □

It is known that plausible circuit lower bounds imply that BPP = P, so against the probably first intuition, many researchers believe that randomization does not add computational power. But this is only a conjecture, so 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*, intuitively these are efficient functions that recover k truly random bits from $n >> k$ corrupted random bits that "contain" the randomness of k truly random bits.

7.3 Upper bounds on BPP

7.3.1 Adleman's theorem

Theorem 7.3.1 (Adleman 1978) BPP $\subseteq$ P/poly.

Proof: Let $Q \in$ BPP. By Proposition 7.2.7 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 $\Pr[\mathbb{A}(x) \neq \chi_Q(x)] \leq 2^{-(n+1)}$. 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 $Q \in$ P/poly by Proposition 6.1.4. □

Corollary 7.3.2 *If* NP $\subseteq$ BPP*, then* PH $= \Sigma_2^{\mathrm{P}}$.

Proof: Immediate by Theorems 7.3.1 and 6.1.7 . □

7.3.2 Sipser and Gács' theorem

Theorem 7.3.3 (Sipser-Gács 1983) $\mathrm{BPP} \subseteq \Sigma_2^{\mathrm{P}} \cap \Pi_2^{\mathrm{P}}$.

Proof: Let $Q \in \mathrm{BPP}$. By Proposition 7.2.7 there is a polynomial p and a p-time bounded probabilistic machine $\mathbb{A}$ such that $\Pr[\mathbb{A}(x) \neq \chi_Q(x)] \leq 2^{-n}$ for all $x \in \{0,1\}^n$. Let G_x be the set of those $y \in \{0,1\}^{p(n)}$ that determine a run of $\mathbb{A}$ on x with output $\chi_Q(x)$.

For some ℓ to be determined later, let $S_1, \ldots, S_\ell$ be mutually independent random variables that are uniformly distributed in $\{0,1\}^{p(n)}$. View $\{0,1\}^{p(n)}$ as a vectorspace over the 2-element field and let $\oplus$ denote its addition. Let $y \in \{0,1\}^{p(n)}$. For $X \subseteq \{0,1\}^{p(n)}, y \in \{0,1\}^{p(n)}$ write $X \oplus y := \{y' \oplus y \mid y' \in X\}$. Since $\oplus$ is a group operation, the map $y' \mapsto y' \oplus y$ is a bijection. It follows that the random variables $S_1 \oplus y, \ldots, S_\ell \oplus y$ are also mutually independent and uniformly distributed in $\{0,1\}^{p(n)}$. Finally, observe that the additive inverse of y is y itself.

Thus, letting y range over $\{0,1\}^{p(n)}$,

$$\begin{aligned} \Pr[\textstyle\bigcup_{i=1}^{\ell} G_x \oplus S_i = \{0,1\}^{p(n)}] &\geq 1 - \textstyle\sum_y \Pr[y \notin \bigcup_{i=1}^{\ell} G_x \oplus S_i] \\ &= 1 - \textstyle\sum_y \prod_{i=1}^{\ell} \Pr[S_i \oplus y \notin G_x] \\ &\geq 1 - 2^{p(n)} \cdot (2^{-n})^{\ell}. \end{aligned}$$

This is positive for $\ell := p(n)$. Thus there exist $s_1, \ldots, s_\ell \in \{0,1\}^{p(n)}$ such that

$$\textstyle\bigcup_{i=1}^{\ell} G_x \oplus s_i = \{0,1\}^{p(n)}.$$

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

$$x \in Q \iff \exists s_1, \ldots, s_\ell \in \{0,1\}^{p(n)} \forall y \in \{0,1\}^{p(n)} : \{y \oplus s_1, \ldots, y \oplus s_\ell\} \cap \mathrm{Acc}_x \neq \emptyset,$$

for all $x \in \{0,1\}^n$ provided n is large enough. Then it is immediate that $Q \in \Sigma_2^{\mathrm{P}}$, and $\mathrm{BPP} \subseteq \Sigma_2^{\mathrm{P}}$ follows. But then also $\mathrm{BPP} \subseteq \Pi_2^{\mathrm{P}}$ as $\mathrm{BPP} = \mathrm{coBPP} \subseteq \mathrm{co}\Sigma_2^{\mathrm{P}} = \Pi_2^{\mathrm{P}}$ (recall Exercise 7.2.8). So we are left to prove the claim.

If $x \in Q$, then $G_x = \mathrm{Acc}_x$. As seen above, there are $s_1, \ldots, s_\ell$ such that $\bigcup_{i=1}^{\ell} \mathrm{Acc}_x \oplus s_i = \{0,1\}^{p(n)}$. In other words, for all y there is i such that $y \oplus s_i \in \mathrm{Acc}_x$.

Conversely, assume $x \notin Q$ and let $s_1, \ldots, s_\ell$ be arbitrary. Then $|\mathrm{Acc}_x| \leq 2^{-n} \cdot 2^{p(n)}$, so $|\bigcup_{i=1}^{\ell} \mathrm{Acc}_x \oplus s_i| \leq \ell \cdot 2^{-n} \cdot 2^{p(n)} < 2^{p(n)}$ for n large enough (recall $\ell = p(n)$). Thus there exists $y \in \{0,1\}^{p(n)} \setminus \bigcup_{i=1}^{\ell} \mathrm{Acc}_x \oplus s_i$, in other words, $\{y \oplus s_1, \ldots, y \oplus s_\ell\} \cap \mathrm{Acc}_x = \emptyset$. □

Corollary 7.3.4 *If* $\mathrm{P} = \mathrm{NP}$*, then* $\mathrm{P} = \mathrm{BPP}$.

Proof: If $\mathrm{P} = \mathrm{NP}$, then $\mathrm{PH} = \mathrm{P}$ by Theorem 5.3.5. But $\mathrm{BPP} \subseteq \mathrm{PH}$ as we just saw. □