Computational complexity

Lecture notes

Damian Niwiński

May 21, 2013

Contents

Credits and acknowledgments

These notes accompany a course actually held by Damian Niwiński at the University of Warsaw. In bibliography, we credit the textbooks on which we rely but, especially for the exercises, the source is sometimes hard to identify. The author thanks his collaborators chairing the tutorial sessions: Vince Bárány, Marcin Benke, Tomasz Kazana, Bartek Klin, Eryk Kopczyński, Henryk Michalewski, Filip Murlak and Paweł Parys for contributing exercises, hints and comments. Last but not least, thanks go to the students following the course for a valuable feedback, with hope for more !

1 Introduction

The terms *complex, complexity* are used in natural sciences, art, politics, as well as in everyday life. They sometimes hide our ignorance, e.g., when we say: the situation is complex...

Interestingly, when qualifying something as complex, we often have in mind one of the following opposite features:

- the thing is too chaotic to find a structure in it (e.g., atmospheric turbulence, financial market in crisis), or
- it exhibits a particularly fine structure or behaviour (e.g., a cell in a live organism, a masterpiece of polyphonic music, or a mathematical theory).

We will encounter these features and much more when studying computational complexity.

Overview of basic ideas

Universal machine Computation can be viewed as a process of decision making. In Turing's model, a single agent makes its decisions sequentially, each time choosing one of finitely many possibilities. This extremely simple setting leads to the concept of an *universal Turing machine* which comprises the whole computational power possible in ideal world. According to the *Church-Turing Thesis*, any partial function $f : \mathbb{N} \to \mathbb{N}$ computable in some "intuitive sense" can be also computed by the universal machine if we provide the code ("program") of the function f.

In real world however, when we are interested in getting the values of f for some concrete arguments, we have to take into account the physical parameters like the *time* of computation and the amount of *memory* needed.

Other parameters may be also of importance like, e.g., the *salary* of programmers. The last will not be covered by our theory, although it is certainly related to it. Indeed, complexity theory helps us to identify problems for which "routine" algorithms supposedly fail, and hence creativity of programmers is needed in order to handle at least some special cases.

Polynomial time It is believed that a computation which performs $n^{\mathcal{O}(1)}$ steps for an input of length n is still practically feasible (*polynomial time*). An analogous restriction on the memory (*polynomial space*) is apparently weaker (why ?) but **we don't know** whether it is really the case. Indeed, a great deal of the theory tries to understand what happens between these two restrictions (P vs. $PSPACE$).

The source of difficulty In spite of an apparent diversity of computational problems, we realize that the difficulty is usually the same. The bottleneck occurs when we need to find an object in a huge search space and we have no better method than brute-force (exhaustive) search.

This idea leads to formal concepts of *reduction* between problems and a problem *complete* in a class. Replacing exhaustive search by an oracle which finds an answer in one step gives rise to the complexity class NP. The question $P \stackrel{?}{=} NP$ is perhaps the most famous open problem of computer science.

The reader may wonder what does it mean that "we have no better method than brute-force". Maybe we are just not intelligent enough to find a better method ? How to prove that such a method does not exist whatsoever ? Is this related to the chaotic nature of the problem ? These are the questions beyond the $P \stackrel{?}{=} NP$ problem.

Randomness Using a *random* choice instead of exhaustive search may sometimes speed up the computation considerably. This is often (but not always) related to some weakening of the computation paradigm: we tolerate errors if they are rare. A recent development leads however to the conjecture that randomness, perhaps surprisingly, can be always eliminated (derandomization).

Approximation We can also tolerate errors if they are not too big. For example, while searching for an optimal traversal over a weighted graph, we might be satisfied with a solution which is 10% worse than the optimal one. A related idea is that of *parametrized* complexity: we try to optimize the time w.r.t. one parameter, and disregard another one. It turns out that, in spite of a common "source of difficulty", the problems behave quite differently in this respect.

Interaction Rather than a solitary run of a single agent, we can design computation as interaction of multiple agents. Various scenarios are possible. The agents may compete (game), or cooperate (interactive proof). Competing with an agent that behaves randomly is known as *game against nature*.

Interaction may speed up computation considerably, but it gives rise to new complexity parameters like, e.g., the number of messages to be exchanged.

Positive use of computational difficulty Computational complexity is at the basis of modern cryptography. The crucial fact is that some functions are easy to compute, but hard to invert. An eminent example is *multiplication* of integers. This phenomenon allows us to construct systems where encryption is easy while decryption is hard.

Computational hardness is also exploited in construction of pseudo-random generators, which can be used to derandomize random algorithms. Here, like in mechanics, the "energy" concentrated in one problem is used to solve another problem.

The last examples suggest a yet another feature of complexity, in addition to chaos and structure mentioned at the beginning. This is a kind of *one-wayness* of some mathematical functions, which can be compared with one-way phenomena in physical world and everyday life.

2 Turing machine

In this lecture we review the concept of the Turing machine which you may remember from the course on Automata, languages, and computations. This mathematical concept introduced by Alan Turing in the 1930s has several important consequences: (1) gives rise to the modern programmable computer, (2) reveals the absolute limits of computablility, (3) allows for definition of the basic measures of complexity: computation time and space.

2.1 Understand computation

The story of Turing machine shows how an abstract question on the border of mathematics and philosophy can give rise to a new branch of technology. The question, asked by David Hilbert was:

> *Is there an algorithm to decide any mathematical conjecture ?*

Hilbert expected a positive answer to this question (known as *Entscheidungsproblem*), but the answer given by Alonzo Church (1936) and, independently Alan Turing (1937), revealed that this is not possible.

To this end, Turing gave the first mathematical definition of computation; he just formalized the actions performed by a person performing computation. Note that the concept of *Turing machine*, today learned by computer science students all over the world, was not conceived as a model of a computer ! The other way around: a modern computer came out of it. Nevertheless, it we think of a simplest model of how information is processed on elementary level, we realize that Turing's machine is in fact *the* model of computation.

Before proceeding further, we give an example to illustrate how a (relatively simple) question may fail to have algorithmic solution.

Digression — not everything is computable

We jump to our time. Fix your preferred programming language and consider a program without input dedicated to computing a natural number M. Of course any number can be computed by sort of *write* (M) program, but some numbers may have much shorter programs (e.g., $10^{10^{10}}$). For each M, let $K(M)$ be the length of a shortest program[1] computing M. **Suppose there is an algorithm to compute $K(M)$.** Then we can implement it and use to construct a program P for a mapping

$$n \mapsto \text{ least } M, \text{ such that } K(M) \geq n; \text{ let us call it } M_n.$$

If we fix n and build it into the program P, we obtain a program P_n dedicated to computing M_n. What is the length of this new program ? The number n can be represented in binary, so its length is at most $\lfloor \log_2 n \rfloor + 1$. The program P has had some length $|P|$, and possibly some constant Δ should be added responsible for building-in the input n into the program P; this Δ does not depend on n. We thus obtain

$$|P_n| \leq \lfloor \log_2 n \rfloor + |P| + \Delta + 1 < n,$$

where the last inequality holds true for all sufficiently large n. But, by definition, the number M_n generated by P_n can be only generated by programs of length $\geq n$, a **contradiction !**

This shows that the hypothesis of computability of the function K was wrong.

The above argument uses an idea of the so-called *Berry paradox*: Let n be the least number that cannot be defined in English with less than 1000 symbols. (But we just defined this n.)

[1] It is so-called *Kolmogorov complexity* of M, see [5] and, e.g., [4].

2.2 Machine at work

The basic model

The basic model can be viewed as a *single* right-infinite tape divided into cells. A cell may contain one symbol or be empty (blank). Initially, the tape contains an input word followed by blanks. At each moment, the machine is in some state and scans one cell. It can then re-write the symbol in the cell, change the state, and move one step to the left or to the right (or not at all).

We represent a machine M as a tuple

$$\langle \Sigma, \Sigma_{i/o}, B, \rhd, Q, q_I, q_A, q_R, \delta \rangle \tag{1}$$

Here Σ is a finite *alphabet* with a special symbol $\Sigma \ni B$ (*blank*) representing the empty cell, and a special symbol $\Sigma \ni \rhd$, marking the left-most cell.

We distinguish a subset $\Sigma_{i/o} \subseteq \Sigma$ as an *input/output* alphabet.

Unless stated otherwise, we assume that $\Sigma_{i/o} = \{0, 1\}$.

Q a (finite) set of states containing

- *initial* state q_I
- *accepting* state q_A
- *rejecting* state q_R.

δ is a *transition function* $\delta : (Q - \{q_A, q_R\}) \times \Sigma \to Q \times \Sigma \times \{L, R, Z\}$.

A transition $\delta(q, a) = (p, b, D)$ is usually written

$$q, a \to p, b, D.$$

Here $D \in \{L, R, Z\}$ represents *direction* of a movement: left, right, or *zero*.

Note that $\delta(q, a)$ is undefined if $q \in \{q_A, q_R\}$; we call these two states *final*. We additionally assume that the symbol $\rhd$ cannot be neither removed nor written in a cell where it was not present originally, and that when scanning this symbol, the machine cannot move left. Formally, if $q, a \to p, b, D$ is a transition then $a = \rhd$ iff $b = \rhd$, and if it is the case then $D \in \{R, Z\}$.

To define formally the process of computation, we need the concept of *configuration* (sometimes called *instantaneous description*). A configuration gathers

- current state,
- content of the tape,
- location actually scanned.

For example, suppose the tape contains

$$\rhd\, 1\, \flat\, \flat\, 0\, B\, 1\, \sharp\, 1\, B\, B \ldots$$

where "..." means prolongation by infinite sequence of blanks. Suppose the machine scans the 6th cell (the second 1) in state q. Then the configuration can be represented by

$$(q, 6, \rhd 1 \flat\flat 0 B\, 1\, \sharp 1 B B)\,. \tag{2}$$

Slightly more economically, we can omit the second component and replace the actually scanned symbol x by a composed symbol $(@x)$ (indicating that machine's head is *at* x):

$$(q, \rhd 1 \flat\flat 0 B\, (@1)\, \sharp 1 B B)\,. \tag{3}$$

Formally, in this second representation, we view a configuration C as a pair (q, α), where $q \in Q$, and $\alpha \in \Sigma^*\ (\{@\} \times \Sigma)\ \Sigma^*$.

For simplicity, we sometimes omit parentheses and write, e.g., $a\,a\,@\,b\,c$ instead of $a\,a\,(@b)\,c$.

We will *identify* configurations (q, α) and $(q, \alpha\Delta)$ if $\Delta \in B^*$. So, for example, (3) could be also written $(q, \triangleright 1 \flat\flat 0 B\, @1\, \sharp 1)$ or $(q, \triangleright 1 \flat\flat 0 B\, @1\, \sharp 1 B B B)$.

The relation $C \to_M C'$ meaning that a configuration C' follows from C in one step of computation is defined depending on a transition applicable to C, if any. (Recall that we write $q, y \to p, \xi, D$ to mean $\delta(q, y) = (p, \xi, D)$.)

$$\begin{array}{lll} (q, \beta\, x\, (@y)\, z\, \gamma) & \to_M & (p, \beta\, x\, \xi\, (@z)\, \gamma) \quad \text{if} \quad q, y \to p, \xi, R \\ & \to_M & (p, \beta\, (@x)\, \xi\, z\, \gamma) \quad \text{if} \quad q, y \to p, \xi, L \\ & \to_M & (p, \beta\, x\, (@\xi)\, z\, \gamma) \quad \text{if} \quad q, y \to p, \xi, Z. \end{array} \tag{4}$$

Note that if $(q, \alpha) \to_M (p, \alpha')$ then the state q and three consecutive symbols $x_{i-1}x_ix_{i+1}$ of α completely determine the i-th symbol of α'. We leave the proof of the following as an exercise.

Lemma 1. *Let $C = (q, \alpha)$ be a configuration with $\alpha = \alpha_1\alpha_2 \ldots \alpha_m$. Let $E = (q, \beta)$ be another configuration and suppose that $\alpha_{i-1}\alpha_i\alpha_{i+1} = \beta_{j-1}\beta_j\beta_{j+1}$, for some i and j. Suppose further that $C \to_M (p, \alpha')$ and $E \to_M (s, \beta')$. Then the i-th symbol in α' and the j-th symbol in β' are the same.*

If additionally $\alpha_1\alpha_2 = \beta_1\beta_2$ then the first symbol in α' and the first symbol in β' are the same.

Computations and computability

An *initial* configuration of a machine M (see (1)) on an *input word* $w \in \{0,1\}^*$ is of the form $(q_I, \triangleright @w)$. Any configuration of the form (q_A, α) is *accepting*, and of the form (q_R, α) is *rejecting*.

A computation of M on w is a sequence of configurations

$$C_0 \to_M C_1 \to_M C_2 \to_M \ldots \tag{5}$$

starting from $C_0 = (q_I, \triangleright @w)$.

We say that M *accepts* w if the above computation is finite and ends in an accepting configuration. In other words, $(q_I, \triangleright @w) \to_M^* (q_A, \alpha)$, for some α. (Here $\to_M^*$ denotes the transitive closure of $\to_M$.)

We say that M *rejects* w if $(q_I, \triangleright @w) \to_M^* (q_R, \alpha)$, for some α.

We write $M(w)\downarrow$ if M accepts or rejects w.

The last possibility is that the computation (5) never ends; in this case we write $M(w)\uparrow$. We call a machine M total if this eventuality never happens, i.e., $M(w)\downarrow$, for any input word w. Although totality is a desired property, we shall see in the next section that it cannot be, in general, guaranteed.

The language *recognized* (or accepted) by M is the set

$$L(M) \quad = \quad \{w \in \{0,1\}^* : M \text{ accepts } w\}.$$

We call a language $L \subseteq \{0,1\}^*$ *partially computable* (or *recursively enumerable*) if $L = L(M)$, for some Turing machine M. We call L *computable* (or *recursive*) if moreover M is total.

Comments. Compared to a (deterministic) finite automaton, Turing machine has three additional features:

1. head moves in both directions,
2. tape is infinite,
3. the machine can write in a cell.

It is well-known that adding just (1) we accept only regular languages. An exercise shows that it remains the same for (1) plus (2), although the situation changes if we allow more tapes (see next section). Clearly (2) and (3) brings no new power to finite automata. An interesting case is (1) plus (3), i.e., a Turing machine restricted to the space of the input. This model defines the class of *deterministic context-sensitive* languages that we will revisit later.

Question 1. In a deterministic finite automaton we usually need more than one accepting state and cannot assume that the accepting states are final. Explain why it is possible in Turing machine.

Computability of functions

Let $f : dom\, f \to \{0,1\}^*$ be a function, where $dom\, f \subseteq \{0,1\}^*$. We refer to f as to a *partial* function from $\{0,1\}^*$ to $\{0,1\}^*$. A Turing machine M *computes* the function f if $L(M) = dom\, f$ and, for any $w \in dom\, f$, the accepting configuration for the input w is $(q_A, \triangleright @ f(w))$.

We call a function $f : dom\, f \to \{0,1\}^*$ *partial computable* if it is computed by some Turing machine in the above sense. We call a function *computable* if it is partial computable and total, i.e., defined for every argument.

The multi-tape model

In this model we allow a Turing machine to use k right-infinite tapes; this number is fixed for the machine. Initially, the input word is placed in the leftmost segment of the first tape; all other tapes are empty, except for the leftmost cell, which permanently holds marker $\triangleright$. We can think that the machine uses k heads which operate similarly as the unique head in the previous case; however the state of the machine is global.

Formally, the difference with respect to (1) is only in the type of transition function which is now

$$\delta : (Q - \{q_A, q_R\}) \times \Sigma^k \to Q \times \Sigma^k \times \{L, R, Z\}^k.$$

Configuration of a k-tape machine is of the form $(q, \alpha_1, \ldots, \alpha_k)$, with $\alpha_i \in \Sigma^*(\{@\} \times \Sigma)\Sigma^*$. The next step relation $C \to_M C'$ is defined analogously to (4) in an obvious manner. The initial configuration of M on a word w is

$$(q_I, \underbrace{\triangleright @ w, \triangleright @ B, \ldots, \triangleright @ B}_{k}).$$

The concept of acceptance is completely analogous.

The off-line model

This model differs from the previous one only in that the tape containing the input word is finite and *read-only*. That is, the corresponding head can read symbols and move in both directions, but cannot write. Additionally, we assume that an input word $w \in \{0,1\}^*$ is placed between two symbols $\triangleright$, $\triangleleft$, serving as *markers*. We refer to such machine as a *k-tape off-line machine* if additionally it has k working tapes. Hence, an initial configuration is of the form

$$(q_I, \triangleright @ w \triangleleft, \underbrace{\triangleright @ B, \ldots, \triangleright @ B}_{k}).$$

The transition function is of the type

$$\delta : (Q - \{q_A, q_R\}) \times (\Sigma \cup \{\triangleright, \triangleleft\}) \times \Sigma^k \to Q \times \Sigma^k \times \{L, R, Z\}^{k+1}.$$

The remaining definitions are updated accordingly.

Comments. Simulation of a k-tape machine or a k-tape off-line machine by a single tape machine is an easy exercise (see 2.5.2). In exercises we also consider the cost of this simulation in terms of the time overhead.

2.3 Universal machine

An ingenious idea of Alan Turing was that a machine can "read" another machine (or itself !) and use it as a program. To this end, we need some encoding of machines as words. This can be done in many ways; here we follow closely the textbook [3].

The construction

We may identify states with numbers $1, 2, 3, \ldots$, and assume that

the *initial* state is 1
the *accepting* state is 2
the *rejecting* state is 3.

Without loss of generality, we may assume that the alphabet is $\{0, 1, B\}$.

We first fix an encoding of ingredients of the machine:

States		Symbols		Directions	
1	0	0	0	L	0
2	00	1	00	R	00
3	000	B	000	Z	000
4	0000				
...	...				

A transition $q, a \to p, b, D$ is encoded as 5 blocks of 0's encoding q, a, p, b, D, respectively, separated by 1's. For example

5		1	$\to$	4		0		R
00000	1	00	1	0000	1	0	1	00

Finally, if the set of all transitions of M is $\delta = \{tr_1, \ldots, tr_m\}$, and $\langle tr \rangle$ denotes the encoding of transition tr, we let

$$\langle M \rangle = 111\, tr_1\, 11\, tr_2\, 11 \ldots 11 tr_m\, 111. \tag{6}$$

To make the above definition unambiguous, we may require that the encodings of the transitions appear in some order, e.g., lexicographic.

Note that a word $w \in \{0, 1\}^*$ may have at most one prefix of form $\langle M \rangle$, for some M.

We define an *universal Turing machine* U as a 2-tape off-line machine. The machine first verifies if an input word is of the form $\langle M \rangle\, w$, for some M; if it is not the case, U stops and rejects.

Otherwise, U copies w on the first working tape, and places the head at the leftmost cell; it also prints one 0 on the second working tape—this is an encoding of the initial state of M.

The configuration of U is now

$$(q, \triangleright \langle M \rangle\, w \triangleleft, \triangleright @ w, \triangleright @ 0)$$

(for some q).

The computation of U can be now seen as a sequence of "big steps" simulating the subsequent steps of the computation of M on w. At the beginning of each big step, the situation on the first working tape corresponds exactly to the situation in some configuration (i, α) of M: the content of the tape is the same and the head is in the same place. The second working tape contains i zeros—the encoding of the state i. s Suppose, for concreteness, that $i = 5$ and the symbol scanned at the first working tape is 1.Then U examines the encoding of $\langle M \rangle$ on the input tape in order to find the unique transition suitable for $5, 1$. Suppose M has a transition $5, 1 \to 4, 0, R$. Then U finds its encoding 00000 1 00 1 0000 1 0 1 00, and realizes the encoded

transition. That is, U replaces the scanned symbol 1 on the first working tape by 0, and moves the head to the right. Finally, it updates the (encoding of the) state of M on the second working tape; in our case it replaces 00000 (state 5) by 0000 (state 4). This ends a big step.

If, at the end of a big step, the content of the second working tape is 00 or 000 (corresponding to the final states of M) then U stops and accepts or rejects, respectively.

It follows easily from the above description that

$$L(U) = \{\langle M\rangle w : w \in L(M)\}. \tag{7}$$

We call any machine satisfying (7) *universal*. The above construction shows the following.

Theorem 1 (Turing)**.** *Universal machines exists.*

Intuitively, U acts as a computer capable to execute any program $\langle M\rangle$. It is believed that this model achieves the maximal computation power, sometimes called *Turing's power.*

What is the relation between Turing machines and algorithms ? Intuitively, an algorithm solves a decision problem if it always gives a correct answer: *yes* or *no* if finite time, given an instance of the problem. On the other hand, a Turing machine may loop in an infinite computation, unless it is total (cf. page 6).

Church-Turing Thesis. A set $L \subseteq \{0,1\}^*$ is recognized by some algorithm in intuitive sense, if and only if it can be recognized by some total Turing machine M, i.e., for all $w \in \{0,1\}^*$

$$\begin{aligned} w \in L &\Rightarrow U \text{ accepts } \langle M\rangle u \\ w \notin L &\Rightarrow U \text{ rejects } \langle M\rangle u. \end{aligned}$$

An analogous statment can be made for a concept of a *partial algorithm*, i.e., one that can fail to give a *negative* answer in finite time.

Note that the above is not a mathematical theorem, because the concept of *intuitive sense* is not defined precisely. However, no evidence has been found against this thesis, and it is generally admitted to be a *fact*.

Undecidability of the halting problem

At first glance, the Church-Turing Thesis might appear as an evidence for a positive answer to the Hilbert question mentioned in Section 2.1 (*Entscheidungsproblem*). If a universal machine can realize any algorithm, then it should be able to simulate any techniques to solve mathematical conjectures, as soon as we can make them "algorithmic". Unfortunately—or, maybe, fortunately—it is not the case. The difficulty comes from the issue of halting in finite time.

Theorem 2 (Turing)**.** *No machine can be simultaneously universal and total.*

Proof. Suppose U is such a machine. We can construct a (single tape) machine D ("diagonal") such that

$$L(D) = \{\langle M\rangle : M \text{ accepts } \langle M\rangle\}. \tag{8}$$

The machine D first checks if its input x is of the form $x = \langle M\rangle$, for some M; if so, it simulates U on $\langle M\rangle\,\langle M\rangle$. If U halted for every input then D would be total as well. Therefore we could construct a total machine D' acting as a *negative* of D, i.e.,

$$(\forall w)\, w \in L(D) \Longleftrightarrow w \notin L(D'). \tag{9}$$

Now a contradiction will arise when we try to decide if D' accepts its own encoding $\langle D'\rangle$.

Suppose it does. Then, by (8), D should accept $\langle D' \rangle$, contradicting (9).

Suppose it doesn't. Then, by (8), D should not accept $\langle D' \rangle$, again contradicting (9).

This shows that the hypothesis that a universal Turing machine can be designed in such a way that it halts for all inputs was wrong. □

Let us sketch an idea of Turing's negative answer to Hilbert's question. Suppose there is an algorithm to decide any mathematical conjecture. Then we could use it to decide the conjectures of the form "$w \in L(U)$?", but we have just seen that it is not possible.

Remark. If we view an universal Turing machine as a model of a computer then an intuitive interpretation of Theorem 2 above is: *computer must sometimes loop.* But maybe this is because some programs are *intentionally* written to loop ?

Let us call a machine U_0 *semi-universal* if it is universal for total machines, i.e., if M is total then, for any w,

$$\langle M \rangle w \in L(U_0) \iff w \in L(M)$$

(if M is not total, U_0 may behave in an arbitrary way).

Question. Does there exist a **total** semi-universal machine ?

Beyond partial computability

Theorem 2 shows that, in the terminology introduced on page 6, the class of partially computable sets strictly extends the class of computable sets. A useful relation between the two concepts is the following.

Theorem 3 (Turing–Post)**.** *If a set $L \subseteq \{0,1\}^*$ and its complement $\overline{L} = \{0,1\}^* - L$ are both partially computable then they are also both computable.*

Proof. If $L = L(M_1)$ and $\overline{L} = L(M_2)$, construct a machine which simulates both M_1 and M_2 in parallel, by assumption one of them will always stop. □

This implies that, in particular, the complements of $L(U)$ and $L(D)$ are not even partially computable. We will now see an example of a meaningful language, such that neither L nor $\overline{L}$ is partially computable.

Assuming the coding of standard Turing machines defined at the beginning of this section, let

$$\begin{aligned} \mathit{Total} &= \{\langle M \rangle : M \text{ is total } \} \\ \mathit{Partial} &= \overline{\mathit{Total}}. \end{aligned}$$

Proposition 1. *Neither* Total *nor* Partial *is partially computable.*

Proof. Let us first see that the following set

$$\mathit{Loop} = \{\langle M \rangle w : M(w) \uparrow\}$$

is not partially computable. Indeed, we can present

$$\overline{L(U)} = \{v : v \neq \langle M \rangle w, \text{ for any } M, w\} \cup \{\langle M \rangle w : M \text{ rejects } w\} \cup \mathit{Loop}$$

where the first two sets on the right-hand side are clearly partially computable. This implies that *Loop* is not, since the class of partially computable sets is closed under finite union.

Suppose $\mathit{Total} = L(M_0)$, for some machine M_0. We will arrive at a contradiction by constructing a machine recognizing Loop. First, for a fixed M and w, let $\mathit{Try}_{M,w}$ be a machine which, for an input v, tries to perform $|v|$ steps of computation of M on w. If within this computation the machine M stops then $\mathit{Try}_{M,w}$ loops forever. Otherwise $\mathit{Try}_{M,w}$ stops and accepts. By definition

$$\mathit{Try}_{M,w} \in \mathit{Total} \quad \Leftrightarrow \quad \langle M \rangle w \in \mathit{Loop}\,.$$

Now, our machine, for an input $\langle M \rangle w$, computes the code of the machine $\mathit{Try}_{M,w}$ and verifies if it is accepted by the hypothetical machine M_0. By above, this machine recognizes Loop, a contradiction.

Now suppose $\mathit{Partial} = L(M_1)$, for some machine M_1. Again, we will construct a machine recognizing Loop. First, for a fixed M and w, let $\mathit{Prove}_{M,w}$ be a machine which, for an input v, simulates the computation of M on w (ignoring v). Then clearly

$$\mathit{Prove}_{M,w} \in \mathit{Partial} \quad \Leftrightarrow \quad \langle M \rangle w \in \mathit{Loop}\,.$$

Similarly as in the previous case, our machine, for an input $\langle M \rangle w$, computes the code of the machine $\mathit{Prove}_{M,w}$ and verifies if it is accepted by the hypothetical machine M_1, thus recognizing Loop. This contradiction completes the proof. □

2.4 Bounding resources

Computation time

Let $T : \mathbb{N} \to \mathbb{N}$ be any function. We say that a Turing machine M *works in time* $T(n)$ if, for any input word w of length n, M makes *at most*[2] $T(n)$ steps. In other words, the computation (5) has length at most $T(n)$. Note that the above definition does not depend on the number of tapes (transition of a multi-tape machine counts as one step), and it applies to an off-line machine in the same manner.

We say that a language L is *recognized in time* $T(n)$ if $L = L(M)$ for some multi-tape Turing machine M working in time $T(n)$. We let $\mathit{DTIME}(T(n))$ denote the class of all languages recognized in time $T(n)$.

Comment. A mathematical model of a computer is an universal machine U rather than a particular machine M. If we view M as a program for a universal machine U, our main concern should be how much time U spends on simulating M for an input w. Roughly, this indicates how much we may loose by using a universal computer rather than constructing M directly. If M works in time $T(n)$ then the time of the simulation described in subsection 2.3 is $\mathcal{O}(|\langle M \rangle| \cdot T(|w|))$, which indicates that the overhead is "only" linear. However, we simulate there a 1-tape machine by a 2-tape off-line machine (or, more generally, a k-tape machine by $k+1$-tape off-line machine). To make our estimation fair, we should compare a machine M with a universal machine "of the same class", i.e., with the same number of working tapes. The straightforward construction of Exercise 2.5.4.1 yields the time of simulation $\mathcal{O}(T(n)^2)$; however a better bound $\mathcal{O}(T(n) \log T(n))$ can be achieved (see [1], sect. 1.7).

Computation space

Let $S : \mathbb{N} \to \mathbb{N}$ be any function. We say that an off-line Turing machine M *works in space* $S(n)$ if, for any input word w of length n, M visits *at most* $S(n)$ cells on the working tapes. Here, we consider a cell as *visited* if it is scanned at least once by the machine's head. Note that the cells of the input tape are *not* counted.

[2] We do not require that the bound $T(n)$ is reached.

We say that a language L is *recognized in space* $S(n)$ if $L = L(M)$ for some multi-tape off-line Turing machine M working in space $S(n)$. We let $DSPACE(S(n))$ denote the class of all languages recognized in space $S(n)$.
(We occasionally estimate the space complexity of an ordinary machine, not off-line; in this case the cells of all tapes will be counted.)

Example 1. Consider the language of palindromes. It can be recognized by an off-line machine with one working tape which copies the input word on the working tape and then examines the agreement of the corresponding symbols by moving the two heads in opposite directions. This machine uses the space n. Alternatively, we can use the working tape as a counter holding a binary representation of a number $i \leq n$. We can use this counter to verify that the i-th symbols from the beginning and from the end are the same. This machine uses only $\lfloor \log n \rfloor + 1$ space. (Although it is less efficient as far as time is concerned.)

There is however an important difference between limiting computation time and space. It may happen that a machine works in a bounded space, but the computation is *infinite*. We may therefore wonder if a language recognized by such a space-bounded is always computable. The above problem can be easily fixed by controlling repetitions: the machine records all subsequent configurations on a supplementary working tape and halts–rejects as soon as some configuration is repeated. This solution however uses much bigger space than the original machine.

A better solution is based on an idea of a *clock*. This is a "sub-routine" of a Turing machine that counts to c^K, for some c and K. It can be implemented by listing in lexicographic order all words of length K over some alphabet of size c. Consider an off-line machine M working in space $S(n)$. The *number* of all configurations that M can reach from an initial configuration on an input of length n is bounded by $C^{S(n)} \cdot n$, for some constant C. Note that the n factor is needed to account on the location of the head on the input tape. However, if $\log n \leq S(n)$, we can eliminate it by increasing the constant, thus obtaining the bound of $D^{S(n)}$, for some D. Therefore it is enough to use a clock of length $S(n)$ over an auxiliary alphabet of size D. In this solution we do not increase the space (c.f. Exercise 4), but we need two assumptions about the function $S(n)$: that it is at least $\log n$, and that it is space constructible (see Exercise 2.5.4.2). We leave the details of this construction as Exercise 2.5.4.5.

A much more elegant and general solution is given by the following.

Theorem 4 (Sipser)**.** *Suppose an off-line Turing machine M works in space $S(n)$. Then we can construct an off-line machine M', such that*

- *$L(M') = L(M)$,*
- *M' works in space $S(n)$,*
- *M' halts for every input.*

Proof. The machine M' will simulate computations of M, however not forward, but backward. We first outline an algorithm informally, and then show that it can indeed by implemented by a machine M' satisfying the requirements of the theorem. The following structure is useful.

Configuration graph. We consider a directed graph G_m whose nodes are configurations of M, and edges correspond to the relation $\rightarrow_M$ defined in (4), i.e., there is an edge $C \rightarrow C'$ if C' follows from C in one step. Note that the out-degree of each node in this graph is at most 1. Let $\rightarrow^*$ stand for the reflexive-transitive closure of $\rightarrow$. For a configuration D, let

$$Back(D) \quad = \quad \{E : E \rightarrow^* D\}.$$

We identify $Back(D)$ with the induced subgraph of G_m.

For a given node D, we will be interested in a subgraph $Back(D)$ induced by the backward search of the configuration graph, i.e., the set of nodes E, such that $E \rightarrow^* D$. Note that this set is either a tree with the

root D (where the edges go backward, i.e., from child to parent) or a cycle, each node on which is a root of a tree (possibly trivial). Note however that these trees may have infinite branches.

We first outline an algorithm to be implemented by M'; the space requirement is expressed in the *Claim*, which will be verified later. For simplicity, we assume that M uses one working tape (in addition to the input tape); the proof for k tapes is analogous. The machine M' will use an alphabet and a set of states which extend the corresponding items of the machine M.

We say that a configuration $C = (q, \triangleright\beta\triangleleft, \alpha)$, is a *$K$-configuration* if it uses no more than K cells on the working tape, i.e., $|\alpha| \leq K$. We first define a procedure $Search(C, K, w)$ for a natural number $K \geq 2$, a K-configuration C of the machine M, and an input word w. We assume that there is no K-configuration C', such that $C \to C'$. (That is, either C is final, or the next step should increase the space to $K + 1$.) The procedure will verify if the configuration C can be reached by M from the initial configuration on input w, going only through K-configurations. The idea of procedure *Search* is as follows.

Consider the subgraph $Back_K(C)$ of $Back(C)$ obtained by restriction to K-configurations. That is, E is a node in $Back_K(C)$ if there is a computation from E to C going through K-configurations. Note that, by our assumption, C cannot lie on a cycle of K-configurations, therefore the subgraph in consideration is a tree. Moreover, this tree cannot have infinite branches, because the number of K-configurations is finite (for a fixed input word). The machine implementing procedure $Search(C, K, w)$ will operate on a working tape where K cells are *marked*. (Technically, a marked symbol x can be represented by a new symbol $(\heartsuit, x)$.) The machine performs the backward search in DFS manner; the search is restricted to K-configurations, which is easy to ensure thanks to the marking. If the initial configuration is found, the procedure answers *Yes*; otherwise the DFS algorithms returns to C, and the procedure answers *No*.

Claim. Procedure $Search(C, K, w)$ can be implemented by an off-time Turing machine in space K.

We postpone verification of the claim to the end of the proof, and describe now the algorithm for M'. We call a K-configuration C *augmenting* if $C \to C'$, for some $K + 1$-configuration C'. Note that, in an augmenting configuration, the head of the working tape scans the rightmost symbol and the transition requires move right. In particular, it is easy to verify if a given configuration is augmenting.

Let w be an input word of length n. Note that, by definition, the initial configuration $(q_I, \triangleright w\triangleleft, \triangleright(@B))$ uses the space 2. We mark the cell scanned by the head in the initial configuration. (The marker $\triangleright$ will be always considered as marked.) The algorithm will use an integer variable K initialized by $K = 2$. For a given K, it performs the following steps. We assume that K cells of the working tape are already marked.

1. Check if some augmenting K-configuration can be reached from the initial configuration. If so than let $K := K + 1$, mark a new cell, and go back to 1.

2. Otherwise, check if some accepting K-configuration can be reached from the initial configuration. If so then stop and **accept**; otherwise stop and **reject**.

Note that the algorithm will necessarily stop for some $K \leq S(n)$. It accepts only if some accepting configuration is reached by M from the initial configuration, i.e., M accepts the input w. It rejects if, for some K, M does not accept w in space $\leq K$, but, at the same time, M does not want to augment this space. This means that M loops in the space K and hence w should be rejected.

To realize (1) and (2) we use the procedure *Search* defined above. We assume some pre-defined lexicographic order on all configurations. Hence, in (1), we can generate in space K all augmenting K-configurations and, in (2), all accepting K-configurations. We leave to the reader to verify that the whole algorithm can indeed be realized by a Turing machine in space $S(n)$, *modulo* the *Claim*.

Proof of the Claim. A machine M' realizing $Search(C, K, w)$ starts with the configuration C of M, where K cells are marked on the working tape. As usual, the computation of M' can be viewed as a sequence of *big* steps, which usually consists of several *small* steps. In big steps, M' performs the backward DFS over the graph $Back_K(C)$. At the beginning of each big step, the configuration of M' coincides with some

K-configuration of M. Then M' has to simulate a step forward of M, or a step backward, according to DFS. The actual mode is remembered in the state. Hence, when arriving in a node D, the machine knows the direction from which it has come. A step forward consists of simulation of just one move of M, say $D \to D'$. In a step backward, we have to find a configuration E, such that $E \to D$. Note that there can be several such configurations, or none. It follows however from Lemma 1 that each such configuration E can be obtained from D by modifying at most three symbols on the working tape, along with the locations of the heads, and the state. Therefore the information needed to perform a step back can be encoded in the states of M' along with some ordering on the predecessor configurations. This is enough to realize DFS in the required space.

If no step backward is possible from configuration D, the machine M' checks if this configuration is actually initial. Again, it is easy because of the marking. If it is the case, M' suspends the DFS procedure and instead moves only forward until it reaches C; thus it follows the computation of M on input w as long as it is restricted to K-configurations. (By assumption about C, it must stop there.) The answer is *Yes* in this case. If an initial configuration is not found, M' completes the DFS procedure and terminates in C as well, but in this case the answer is *No*. This remark ends the proof.

□

Remark. Theorem 4 shows that if a Turing machine M cannot be improved to a machine that halts on every input—we have seen examples of such machines in Section 2.3—then machine M must use infinite space for some inputs.

2.5 Exercises

2.5.1 Coding objects as words and problems as languages

1. Try to find an efficient way to encode a **pair** of binary words as a single binary word.

   Hint. Let $C(a_1a_2\ldots a_k) = a_10a_20\ldots a_k011$. The first approximation codes (u, v) by $C(u)v$, but it can be improved to $C(\alpha)uv$, where α is the binary representation of the *length* of u. Now this construction can be iterated...

2. The *incidence matrix* of a directed graph with n vertices identified with $\{1, \ldots, n\}$, is a binary word of length n^2, which has 1 on position $(i-1)n + j$ if there is an arrow $i \to j$, and 0 otherwise. Note that no infinite set of graphs is encoded by a context-free language (why ?).

   But with some care we can find some (not completely trivial) graph-theoretic properties encoded by *complements* of context-free languages. Give an example.

   Hint. For example, graphs with only self-loops $i \to i$.

3. Propose an alternative encoding of symmetric graphs using the concept of *adjacency list*.

4. Show that if $\alpha : \mathbb{N} \to \{0,1\}^*$ is a 1:1 function then, for infinitely many n's $|\alpha(n)| > \lfloor \log_2 n \rfloor$. Show that this bound is tight in the sense that there is a 1:1 function $alpha : \mathbb{N} - \{0\} \to \{0,1\}^*$, satisfying $|\alpha(n)| \leq \lfloor \log_2 n \rfloor$, for all argument.

   Remark. This means in particular that no notation for natural numbers can be more efficient than the usual binary notation.

   (It is the content of *Corollary 1* in [4].)

2.5.2 Turing machines as computation models

1. **Robustness.** Show the equivalences between various models of Turing machines: two-way infinite tape *vs* one-way infinite tape; k tapes *vs* 1 tape. Estimate the cost of simulation. Show that the alphabet of auxiliary symbols can be reduced to $\{0, 1, B\}$ (in fact, $\{1, B\}$ would suffice as well).

2. The transitions of Turing machine could be defined in a slightly different way. Namely, we could require that the machine first moves the head (left or right) and then writes a symbol in a "new" cell. Show that this type of machines is equivalent to the original one.

3. **Write–once Turing machine.** A Turing which writes a symbol in a cell only if it is empty (i.e., contains a blank) can simulate an arbitrary Turing machine. We do not put restriction on the number of tapes.

4. If we additionally require that a machine of (3) has only **one tape** then it can recognize (only) a regular language.

5. ([3]) Each Turing machine can be simulated by a machine which, in addition to the final states q_A, q_R uses only **2** working states (one of them initial).

   Remark. That some *fixed* number of states suffices, follows easily from the construction of a universal machine. Note however that auxiliary symbols are needed to produce the encoding of the simulated machine. The construction of a (universal) machine with 2 states was first given by Claude Shannon, who also showed that 1 working state would not suffice.

   Hint. The states of an original machine can be encoded by auxiliary tape symbols which most conveniently can be viewed as sequences of bits. Moving of such a "symbol" from one cell to another is performed in many steps, "bit by bit".

6. **Reversible computation.** We call a configuration c of a machine M reversible if there is at most one configuration d, such that c is reached from d in one step. The machine M is *weakly reversible* if any configuration *reachable from some initial configuration* is reversible. Note that if it is the case then we can trace the computation back.

   For a given Turing machine M, construct a weakly reversible machine recognizing the same language; estimate the time overhead in your construction.

   Remark. A machine is (strongly) reversible if *all* its configurations are reversible. Note that this implies that all maximal computation paths are disjoint. The analogous claim for reversible machines is problematic, see discussion on `http://mathoverflow.net/questions/24083/reversible-turing-machines`.

2.5.3 Computability

1. Recall that a language $L \subseteq \{0,1\}^*$ is *partially computable* (or *recursively enumerable*) if $L = L(M)$, for some Turing machine M. It is *computable* (or *recursive*) if this machine halts for all inputs (see *Lecture notes*). A partial function $f : \{0,1\}^* \supseteq dom\, f \to \{0,1\}^*$ is computable if there is a Turing machine M which halts precisely for $w \in dom$ with the value of $f(w)$ on the tape.

   Prove that the following conditions are equivalent:

   (a) L is partially computable,

   (b) L is the set of values of some computable partial function,

   (c) L is empty or is the set of values of some computable total function.

2. Prove the so-called Turing-Post theorem: if a set and its complements are partially computable then they are also computable.

3. (*) If a machine M computes a partial function f, we say that a machine M' *friendly corrects* M is it computes a total function g such that $g \supseteq f$. Show that there is no computable transformation $M \mapsto M'$, such that M' friendly corrects M.

2.5.4 Turing machines — basic complexity

1. **Reduction in number of tapes** (2.5.2.1 revisited). Show that a Turing machine with k tapes working in time $T(n)$ and space $S(n)$ can be simulated by a machine with one tape working in time $\mathcal{O}(T(n)^2)$ and (the same) space $S(n)$.

   Remark. A better bound for computation time can be showed for two tapes: a machine with $k > 2$ tapes can be simulated by a machine with two tapes working in time $\mathcal{O}(T(n) \log T(n))$ (see [3], Theorem 12.6).

2. **Constructible functions.** A function $f : \mathbb{N} \to \mathbb{N}$ is *space constructible* if there is an *off-line* Turing machine which, for an input of length $n \geq 1$, writes in exactly $f(n)$ cells of the auxiliary tapes. Show that the following functions are space constructible: $n, 2n, n^2, n^2 + n, 2^n, \lceil \log n \rceil$.

   A function f is *time constructible* if there is a Turing machine which, for an input of length $n \geq 1$, makes exactly $f(n)$ steps and halts. Show that the following functions are time constructible: $n + 1, 2n, n^2, 2^n, 2^{2^n}$.

3. **Linear speed-up.** Let M be a k-tape Turing machine working in time $T(n)$. Show that, for any $c > 0$, there is an equivalent machine M' using $k + \lfloor \frac{1}{k} \rfloor$ tapes and working in time $c \cdot T(n) + \mathcal{O}(n)$.

   Hint. Use new symbols to represent the blocks of symbols of suitable length.

4. **Linear space compression.** Let M be an off-line Turing machine with k working tapes working in space $S(n)$. Show that, for any $c > 0$, there is an equivalent off-line machine M' using $c \cdot S(n)$ space.

5. Assuming that $S(n)$ is a space constructible function satisfying $\log n \leq S(n)$, show that an off-line machine working in space $S(n)$ can be simulated by a machine that halts for every input, using the clock construction described on page 12.

6. **Arithmetic.** Assuming that natural numbers k, m, n are given in binary representation, estimate (from above) the time to compute $m + n$, $m \bmod n$, $m \cdot n$, $m^n \bmod k$.

7. Estimate the computation time of the **Euclid algorithm** implemented on Turing machine.

8. **Complexity vs. computability.**

   (a) For a Turing machine M, consider a mapping $S_M : \{0,1\}^* \to \mathbb{N} \cup \{\infty\}$, where $S_M(w)$ is the exact amount of space used by M for an input w (possibly infinity). Show that this function is, in general, not computable. On the other hand, if $S_M(w)$ never takes the value ∞ then it is computable.

   Hint. Use Theorem 4.

   (b) If $S : \mathbb{N} \to \mathbb{N}$ is any computable function then the set

   $$\{w : S_M(w) \leq S(w)\}$$

   is computable.

   (c) Formulate and prove analogous properties for time complexity.

2.5.5 One-tape Turing machines

1. Construct a 1-tape DTM, which recognizes the language $\{0^n 1^n : n \in \mathbb{N}\}$ in time $\mathcal{O}(n \log n)$.

   Hint. Implement a counter moving along with the head.

2. A one-tape Turing machine which operates in **linear time** can only recognize a regular language[3].

   Hint. Use a concept of a *history* of a cell similar to that of *crossing sequence* in two-way finite automata. A history gathers the information how many times the head has visited the cell, in which states, in what direction it moved, etc. Note that the sum of the heights of all histories gives the time of computation which, by assumption, is $K \cdot n$, for some K. Show that there is a common bound K' on the height of all histories. (From this it is routine to construct a finite automaton.) To show the claim suppose that there is an accepting computation with some history of a huge height $\geq M$. Choose a shortest word w_M which witnesses this fact. As the "average" height is K, M must be balanced by a sufficient number of histories shorter than K. But there is only a finite number of them. Thus, with an appropriate relation between K and M, we can find two identical histories in two different points lying on the same side of the huge history. Then, by anti-pumping we can make w_M shorter, contradicting its minimality.

3. Show that a one-tape Turing machine recognizing the language $\{www : w \in \{0,1\}^*\}$ requires $\Omega(n^2)$ steps. More precisely, for each such machine M, there is some constant ε_M, such that M makes at least $\varepsilon_M n^2$ steps, for some input of length n, if n is sufficiently large.

   Hint. Fix $n = 3m$ and consider a point in the middle third. Note that different words w of length m must have different histories there, as otherwise we could cheat the machine. So, in each point, 2^m histories must occur. Find a constant ε, such that if we collect all words whose history goes below $\varepsilon \cdot m$ at some point in the middle third, then there will be less than 2^m of them. Hence, some word will require "long" histories in the whole middle third, and consequently the quadratic time.

4. Show that a one-tape Turing machine recognizing **palindromes** requires $\Omega(n^2)$ steps.

   Hint. Use a technique of the previous exercise. Alternatively, you may introduce and use Kolmogorov complexity (see footnote 1 on page 4). See Exercise 2.8.5 in Papadimitriou [6] and the hint there.

3 Non-uniform computation models

3.1 Turing machines with advice

As we have noted in the previous lecture, a universal Turing machine U can be seen as a mathematical model of a *computer*. A typical input for U has the form $\langle M \rangle w$, where $\langle M \rangle$ is an encoding of some Turing machine M. Let $L = L(M)$; we have

$$w \in L \iff \langle M \rangle w \in L(U)$$

(see (7)). Hence we can view $\langle M \rangle$ as a *program* implementing an *algorithm* designed to solve the *problem* L. Typically L is infinite, but M has only a finite number of states and transitions. It is not surprising, since an algorithm usually does not depend on the size of data. In some situations, however, it is reasonable to change this assumption.

For example, recent implementations of the cryptographic algorithm RSA have used the keys $1024--2048$ bits long. Since an attack to the 768 bit RSA was presented[4] in 2010, it is now recommended to use keys of length 2048 or longer. But it is thinkable that in few years another attack will be found, which will force us to increase the security parameters further, and so on. Even if the computation power of an attacker is not unlimited, we should not assume that she always uses the *same* algorithm. These considerations lead to the following definitions.

For a language $L \subseteq \{0,1\}^*$, let

$$L_n = L \cap \{0,1\}^n = \{w \in L : |w| = n\}. \tag{10}$$

[3] I mean a *deterministic* machine here, although the proof should work for non-deterministic machines as well.
[4] See `http://eprint.iacr.org/2010/006`.

Let, for $n \in \mathbb{N}$, K_n be a Turing machine, such that $L(K_n) = L_n$. We now have

$$w \in L \iff \langle K_{|w|} \rangle w \in L(U).$$

How to evaluate the efficiency of this new model ? Note that, since each L_n is finite, we could make K_n a finite automaton, and thus recognize $L(K_n)$ in time $n+1$. Does it mean that L can be "locally" solved in linear time ? Not quite so, because the time that our "computer" U will use on the input $\langle K_{|w|} \rangle |w|$ may also depend on the size of $K_{|w|}$. *A priori* however, it is possible that we can achieve a better computation time by taking a sequence of machines K_n recognizing L_n rather than a single machine recognizing L.

We now define the non-uniform mode of recognition generally, not exclusively for universal machines.

Let M be a Turing machine, and let $(k_n)_{n\in\mathbb{N}}$ be a sequence of words in $\{0,1\}^*$. We assume that this sequence is prefix–free, i.e., no k_n is a prefix of k_m, unless $m = n$. Note that this implies that each word v has at most one decomposition $v = k_{|w|}w$. We say that the pair $\left(M, (k_n)_{n\in\mathbb{N}}\right)$ *recognizes a language* L if, for all $w \in \{0,1\}^*$,

$$w \in L \iff k_{|w|}w \in L(M). \tag{11}$$

We then write

$$L = L\left(M, (k_n)_{n\in\mathbb{N}}\right). \tag{12}$$

In this context, the sequence $(k_n)_{n\in\mathbb{N}}$ is called an *advice*.

We further say that $\left(M, (k_n)_{n\in\mathbb{N}}\right)$ *recognizes* L *in time* $T(n)$ if, for each $w \in \{0,1\}^*$, the computation of M on $k_{|w|}w$ takes no more than $T(|w|)$ steps. Note that we measure the computation time only with respect to $|w|$, not to $|k_{|w|}|$. The concept of recognizing a language *in space* $S(n)$ is defined similarly.

3.2 Boolean circuits

We now introduce a new model of computation, based on a different architecture than Turing machine. A close connection between the two models will appear later, maybe with a little surprise.

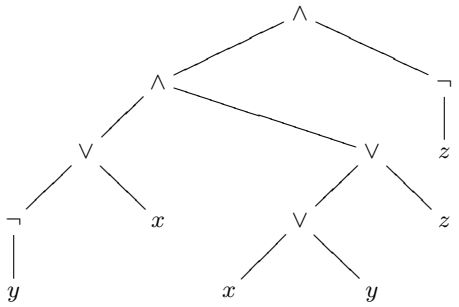
Motivating example. It is a common practice that to find the semantics of a formal expression: arithmetical term, formula, program code, or even sentence of natural language, we represent it as a tree. Usually, there are many ways of doing it.

For example a propositional formula

$$(\neg y \vee x) \wedge (x \vee y \vee z) \wedge (\neg z) \tag{13}$$

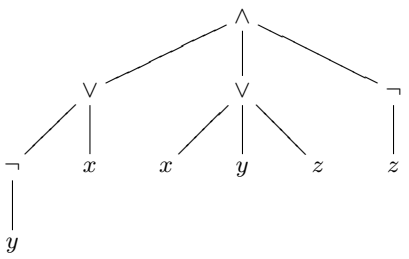
can be represented

(14)



but also

(15)

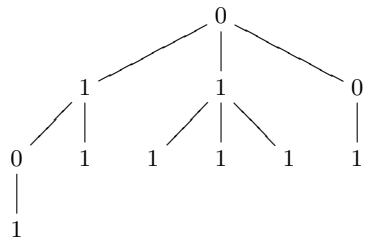


Note that (14) is just a graphical representation (parsing) of the formula (13) with implicitly added parentheses,

$$((\neg y) \vee x) \wedge (((x \vee y) \vee z) \wedge (\neg z)),$$

the nodes labeled by $\vee$ or $\wedge$ have always fan-in 2. In contrast, (15) treats disjunction and conjunction as operations of unspecified arity (thanks to their associativity), which allows us to reduce the depth of the tree. We leave the precise explanation of the transformation (13) $\rightarrow$ (15) to the reader. In any case, given a valuation of the variables x, y, z, we can easily evaluate the formula in a bottom-up manner. We use 0 and 1 also as logical values, with $0 = \mathit{false}$ and $1 = \mathit{true}$. Then, a node labeled $\vee$ gets value 1 if at least one of its children has value 1, and a node labeled $\wedge$ gets value 1 if all its children have value 1. A node labeled $\neg$ changes the value of its (unique) child. For example, given an evaluation $v(x) = v(y) = v(z) = 1$, the representation (15) yields

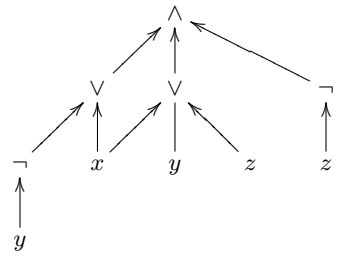
(16)



hence the formula is false in this case.

Note that the evaluation procedure will work equally well if we merge some nodes inducing the same subtrees. We thus obtain a graph which is not necessarily a tree. (17) represents such a graph obtained from (15). Although, in our previous figures, we have omitted the orientation of arrows for simplicity, it is clear that it has been implicitly assumed. We now make it explicit:

(17)



The above examples suggest a generalization of formulas to more compact expressions that we will now introduce.

Basic definition. A *Boolean circuit*, also called *logical circuit*, or simply a *circuit*, is a directed acyclic graph C with labels on the nodes, satisfying the requirements below. The nodes of a circuit are usually called *gates*, and the edges are often called *wires*. We fix some countable set of symbols $\{x_1, x_2, \ldots\}$.

The requirements of circuit follow.

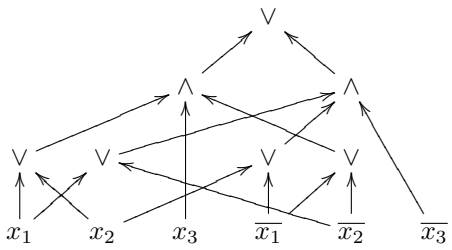


Figure 1: A circuit with 3 input gates.

1. There is a number $n \in \mathbb{N}$, such that, for each $i = 1, \ldots, n$, C has exactly one node labeled x_i, and exactly one node labeled $\overline{x_i}$. These gates have fan-in 0 (no predecessors). The gates labeled by x_i's are called *input gates*, and those labeled by $\overline{x_i}$'s *negated input gates*.
2. The nodes labeled by $\vee$ (called Or-*gates*), and by $\wedge$ (And-*gates*) have arbitrary fan-in and fan-out.
3. One gate is distinguished as an *output gate*; it has fan-out 0 (no successors).
4. There are no other labels.

We sometimes write $C = C(x_1, \ldots, x_n)$ to make explicit the number of input gates of C.

Semantics. Given a valuation of variables $v : \{x_1, \ldots x_n\} \to \{0, 1\}$, we define the value of each gate by induction on the structure of the graph. The gate labeled x_i gets the value $v(x_i)$, and the gate labeled $\overline{x_i}$ gets the value $\neg v(x_i)$. The value of an Or-gate (And-gate) is obtained by taking disjunction (respectively, conjunction) of the values of its children. Note that we allow that an Or-gate (And-gate) may have fan-in 0 (no children). In this case, the value of the gate is 0 (respectively, 1). Note that the above definition is correct, because the graph is finite and acyclic. Finally, the *value of the circuit C under the valuation v* is defined as the value of the output gate; we denote it by $C(v)$. We also write $C(a_1, \ldots, a_n)$, if $v(x_i) = a_i$, for $i = 1, \ldots, n$. Therefore, a circuit $C(x_1, \ldots, x_n)$ defines a Boolean function $\{0, 1\}^n \ni (a_1, \ldots, a_n) \mapsto C(a_1, \ldots, a_n) \in \{0, 1\}$.

For example, the circuit of Figure 1 computes the function *parity*, which outputs 1 iff the number of the input gates valued by 1 is even.

Negation. Note that in our definition we admit the negated input gates, but we do not allow gates labeled by negation. In fact, we could extend circuits in this way, provided that the fan-in of a negation gate is always 1. However, the negation gates could be easily eliminated by De Morgan laws without increase of the size (Exercise 3.5.1.1).

Encoding. For further considerations, it is useful to represent circuits by words. We do not make it as formal as we have done for Turing machines in Section 2.3; instead we propose an alternative "high-level" representation of a circuit.

For each gate of a circuit C, we choose some *identifier* $p \in \{0, 1\}^*$ (like a variable in a program). For example, for the circuit of Figure 1, we fix some words $p_1, p_2, p_3, \ldots, p_{13}$, and a correspondence between the gates and identifiers as in Figure 2.

Then, for each gate, we form an equation relating this gate with its children. In our example, the system of equations will be

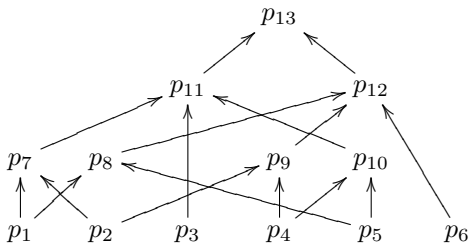


Figure 2: Identifiers of the gates of circuit from Figure 1.

$$\begin{array}{llllllllll} p_1 & = & x_1 & p_7 & = & \text{Or } \{p_1, p_2\} & p_{11} & = & \text{And } \{p_3, p_7, p_{10}\} \\ p_2 & = & x_2 & p_8 & = & \text{Or } \{p_1, p_5\} & p_{12} & = & \text{And } \{p_6, p_8, p_9\} \\ p_3 & = & x_3 & p_9 & = & \text{Or } \{p_2, p_4\} & p_{13} & = & \text{Or } \{p_{11}, p_{12}\} \\ p_4 & = & \text{Not } (x_1) & p_{10} & = & \text{Or } \{x_4, x_5\} & \text{Output} & = & p_{13}. \\ p_5 & = & \text{Not } (x_2) & & & & & & \\ p_6 & = & \text{Not } (x_3) & & & & & & \end{array} \tag{18}$$

In general, if p is an identifier of an Or-gate and $q_1, \ldots, q_k$, are identifiers of its children, we form an equation $p = \text{Or } \{q_1, \ldots, q_k\}$; similarly for And-gates. Additionally, an equation Output $= p$ is formed if p identifies the output gate. It is routine to encode such a systems of equations as by a binary word (c.f. Exercise 2.5.1.1); we examine the size of this encoding in the next paragraph.

The above representation yields an elegant characterization of the semantics of the circuit. Indeed, if we substitute the value $v(x_i)$ for x_i then there is a unique valuation of all the identifiers which makes the system true. The value of an identifier p coincides with the value of the gate identified by p. Then $C(v)$ coincides the value of the identifier Output.

In accordance with the semantics, we will abbreviate

$$\begin{array}{rcl} \text{Or } \emptyset & = & \mathit{false} \\ \text{And } \emptyset & = & \mathit{true}. \end{array}$$

Complexity parameters. With a circuit C, we associate the following parameters:

- the number of input gates, n,
- the number of all gates, denoted $\mathit{Gate}(C)$,
- the *depth* of C, denoted $\mathit{Depth}(C)$, which is the length of a longest path from an input gate to the output gate (equals 3 in our example),
- the number of all wires (i.e., edges), denoted $\mathit{Wire}(C)$.

Note that if a circuit with k gates and m wires is represented by a system of equations as above, we have $k + 1$ equations with $k + m + 2$ occurrences of identifiers. Therefore, we define the *size* of a circuit by

$$\mathit{Size}(C) = (\mathit{Gate}(C) + \mathit{Wire}(C)) \cdot \log \mathit{Gate}(C). \tag{19}$$

The logarithmic factor is considered because of the size of identifiers. It is routine to define an encoding of circuits by binary words in which the encoding of a circuit C has length $\mathcal{O}(\mathit{Size}(C))$.

Game. For a circuit C and a valuation v, we define a game, $Game(C, v)$, played by two players: Ms. Or, and Mr. And. Intuitively, Or wants to show that the value $C(v)$ is 1, while And wants to show that it is 0. The players start in the output gate and then move the token down the circuit, i.e., in the direction opposite to the wires. The move is selected by Or in Or-gates, and by And in And-gates. A player who cannot move, looses. (It may happen if a node has no predecessors.) If the token arrives in an input gate labeled x_i then Or wins if $v(x_i) = 1$, otherwise And is the winner. Similarly, in a gate labeled $\overline{x_i}$, Or wins if $v(x_i) = 0$, and And otherwise.

We leave the proof of the following characterization to the reader.

Proposition 2. *The player who*[5] *has a winning strategy in the game $Game(C, v)$ is Ms. Or if $C(v) = 1$, and Mr. And if $C(v) = 0$.*

Circuits as acceptors of languages. If the value $C(w_1, \ldots, w_n)$ is 1, for a sequence of bits $w_1, \ldots, w_n$, we say that the circuit $C(x_1, \ldots, x_n)$ *accepts the word* $w = w_1 \ldots w_n$. We abbreviate $C(w_1, \ldots, w_n) = C(w)$ if the length of w is known from the context. Hence, a circuit $C(x_1, \ldots, x_n)$ can be viewed as an "automaton" recognizing the (finite[6]) language

$$L(C) \quad = \quad \{w \in \{0,1\}^n : C(w) = 1\}. \tag{20}$$

To recognize infinite sets, we use sequences of circuits.

Let $(C_n)_{n\in\mathbb{N}}$ be a sequence of circuits, where $C_n = C_n(x_1, \ldots, x_n)$. The language recognized by this sequence is

$$L\left((C_n)_{n\in\mathbb{N}}\right) \quad = \quad \bigcup_{n\in\mathbb{N}} L(C_n) \tag{21}$$

$$= \quad \{w : C_{|w|}(w) = 1\}. \tag{22}$$

At first sight, this mode of acceptance may appear too powerful. Indeed, it is well-known (and easy to see) that any Boolean function $\{0,1\}^n \to \{0,1\}$ can be represented by a propositional formula in disjunctive normal form. Transforming formulas to circuits like in our motivating example, we easily obtain the following (see Exercise 3.5.1.3).

Proposition 3. *Any set $L \subseteq \{0,1\}^*$ is recognized by some sequence of circuits $(C_n)_{n\in\mathbb{N}}$, where moreover $Depth(C_n) = 2$, for all $n \in \mathbb{N}$.*

However, the model becomes interesting if we restrict the size of the circuits.

3.3 Simulating machine by circuits

We show that a Turing machine working in time $T(n)$ can be simulated by a sequence of circuits polynomially related to $T(n)$.

Theorem 5. *Let M be a Turing machine working in time $T(n)$. There is an algorithm which, for $n \in \mathbb{N}$, constructs a circuit $D_n(x_1, \ldots, x_n)$, such that the sequence $(D_n)_{n\in\mathbb{N}}$ recognizes the language $L(M)$. Moreover, we can guarantee that*

$$\begin{aligned} Gate(D_n) &= \mathcal{O}\left(T(n)^2\right) \\ Depth(D_n) &= \mathcal{O}\left(T(n)\right). \end{aligned}$$

[5] As $Game(C, v)$ is a finite perfect information game, it is easy to see that one of the players has always a winning strategy. (This fact was first formally proved for chess by E. Zermelo.) Note that some *infinite* games with perfect information are known, where a winning strategy fails to exist for any of the players.

[6] This definition makes sense also for $n = 0$. A circuit $C()$ without input gates has a constant value 1 or 0, and hence accepts or not the empty word ε. But we will not particularly care about this case.

Proof. Let $M = \langle \Sigma, \Sigma_{i/o}, B, \triangleright, Q, q_I, q_A, q_R, \delta \rangle$ be a Turing machine with $\Sigma_{i/o} = \{0,1\}$ (c.f. (1), page 5). For simplicity, we consider a single-tape model here; the argument for other models is similar. It is convenient to extend the concept of computation (5) so that it has length *exactly* $T(n)$ if an input has length n. Note that, according to our definition, the machine may stop only in a *final* state (q_A or q_R). If it happens for some $t < T(n)$, we let the last configuration be repeated. That is, an *extended computation* of M on input w with $|w| = n$ is a sequence of configurations

$$(q_I, \triangleright @ w) = C_0 \to_M C_1 \to_M C_2 \to_M \ldots \to_M C_t \to_M C_{t+1} \to_M C_{T(n)} \tag{23}$$

for some $t \leq T(n)$, where $C_0 \to_M \ldots \to_M C_t$ is a computation in the sense of (5), C_t is final, and $C_t = C_{t+1} = \ldots = C_{T(n)}$. Note that each C_i can be presented in the form (q, α), for some $q \in Q$ and $\alpha \in \Sigma^* \, (\{@\} \times \Sigma) \, \Sigma^*$, where moreover $|\alpha| \leq T(n) + 1$, because M cannot scan more cells in time is $T(n)$. For convenience, we will assume that $|\alpha| = T(n) + 1$.

We now define a ternary relation

$$Hist_w \subseteq \{0, 1, \ldots, T(n)\} \times \{0, 1, \ldots, T(n) + 2\} \times (\Sigma \cup (Q \times \Sigma))$$

with the intention to completely describe the *history* of computation of M on w. That is, the relation $Hist_w$ will hold for a triplet (i, j, y) if a symbol y is in the location j at the moment i. If moreover the machine is scanning this location in state q, $Hist_w$ will hold for a triplet $(i, j, (q, y))$. More specifically:

1. If $C_i = (p, \alpha y \beta)$, for some p, α, β with $|\alpha| = j$, then $Hist_w(i, j, y)$ holds.
2. If $C_i = (q, \alpha(@y)\beta)$, for some α, β, with $|\alpha| = j$, then $Hist_w(i, j, (q, y))$ holds.
3. $Hist_w(i, T(n) + 2, B)$ holds, for all i.
4. Otherwise, $Hist_w$ does not hold.

An essential property is that the content of location j in moment $i + 1$ is determined by the content of at most 3 neighbour locations in moment i. The following lemma refines Lemma 1; it can be proved by a careful analysis of all the cases in the next-step relation (c.f. (4)). Like before, the leftmost cell requires a special clause.

Lemma 2. *For any $z_{-1}, z_0, z_1 \in \Sigma \cup (Q \times \Sigma)$, there is at most one $y \in \Sigma \cup (Q \times \Sigma)$, such that, for any $w \in \{0,1\}^*$, whenever $Hist_w(i, j-1, z_{-1})$, $Hist_w(i, j, z_0)$, and $Hist_w(i, j+1, z_1)$ hold for some $i < T(|w|)$ and $1 \leq j \leq T(|w|) + 1$, then $Hist_w(i+1, j, y)$ also holds. If it is the case, we write*

$$z_{-1}, z_0, z_1 \quad \vdash_M \quad y.$$

Moreover, for any $z_0 \in \{\triangleright\} \cup (Q \times \{\triangleright\})$ and $z_1 \in \Sigma \cup (Q \times \Sigma)$, there is at most one $y \in \{\triangleright\} \cup (Q \times \{\triangleright\})$, such that, whenever $Hist_w(i, 0, z_0)$, and $Hist_w(i, 1, z_1)$ hold then $Hist_w(i+1, 0, y)$ also holds. If it is the case, we write $z_0 z_1 \vdash_M y$.

Let $n \in \mathbb{N}$. We will construct a circuit D_n satisfying the requirements of the theorem. We will use expressions "$Hist(i, j, y)$" as identifiers of the gates in the circuit. The intention is that, for an input word $w \in \{0,1\}^n$, the gate "$Hist(i, j, y)$" gets value *true* iff the relation $Hist_w(i, j, y)$ holds. We will also use auxiliary identifiers of the form "$Story(i, j, z_{-1}, z_0, z_1)$" and "$Story(i, 0, z_0, z_1)$", with $z_{-1}, z_0, z_1 \in \Sigma \cup (Q \times \Sigma)$. Note that all these identifiers can be encoded as binary words of size $\mathcal{O}(\log T(n))$ if we use binary representation of i and j.

We now give the equational presentation of our circuit (like in (18)).

OR-gates. For $i \in \{1, \ldots, T(n)\}$, $j \in \{0, 1, \ldots, T(n)+1\}$, and $y \in \Sigma \cup (Q \times \Sigma)$, we have an OR-gate with identifier $Hist(i, j, y)$. The equation of this gate is

$$Hist(i, j, y) \quad = \quad \text{Or } \{Story(i-1, j, z_{-1}, z_0, z_1) : z_{-1}z_0z_1 \vdash_M y\}, \tag{24}$$

for $j \geq 1$, and

$$Hist(i, 0, y) \quad = \quad \text{Or } \{Story(i-1, 0, z_0, z_1) : z_0z_1 \vdash_M y\}. \tag{25}$$

Additionally, we have the output gate Win, with the equation

$$Win \quad = \quad \text{Or } \{Hist(T(n), j, (q_A, y)) : 0 \leq j \leq T(n)+1,\ y \in \Sigma\}. \tag{26}$$

AND-gates. For $i \in \{0, 1, \ldots, T(n)\}$, $j \in \{1, \ldots, T(n)+1\}$, and $z_{-1}, z_0, z_1 \in \Sigma \cup (Q \times \Sigma)$, we have an AND-gate with identifier $Story(i, j, z_{-1}, z_0, z_1)$. The equation of this gate is

$$Story(i, j, z_{-1}, z_0, z_1) \quad = \quad \text{And } \{Hist(i, j-1, z_{-1}),\ Hist(i, j, z_0),\ Hist(i, j+1, z_1)\}. \tag{27}$$

Moreover, for $i \in \{0, 1, \ldots, T(n)\}$, and $z_0, z_1 \in \Sigma \cup (Q \times \Sigma)$, we have an AND-gate with identifier $Story(i, 0, z_0, z_1)$. The equation is

$$Story(i, 0, z_0, z_1) \quad = \quad \text{And } \{Hist(i, 0, z_0),\ Hist(i, 1, z_1)\}. \tag{28}$$

Input gates. We have the equations

$$Hist(0, 1, (q_I, 1)) \quad = \quad x_1 \tag{29}$$

$$Hist(0, 1, (q_I, 0)) \quad = \quad \overline{x_1} \tag{30}$$

and, for $j = 2, \ldots, n$,

$$Hist(0, j, 1) \quad = \quad x_j \tag{31}$$

$$Hist(0, j, 0) \quad = \quad \overline{x_j}. \tag{32}$$

Constant gates. We have the equations

$$Hist(0, 0, \triangleright) \quad = \quad true \tag{33}$$

and, for $j = n+1, \ldots, T(n)+2$,

$$Hist(0, j, B) \quad = \quad true. \tag{34}$$

For each σ other than in cases (29 – 34) above, and $j \in \{0, 1, \ldots, T(n)+2\}$, we have an equation

$$Hist(0, j, \sigma) \quad = \quad false. \tag{35}$$

Finally, for $i \in \{0, 1, \ldots, T(n)\}$, we have

$$Hist(i, T(n)+2, B) \quad = \quad true \tag{36}$$

and

$$Hist(i, T(n)+2, \sigma) \quad = \quad false \tag{37}$$

for $\sigma \neq B$.

This completes the description of the circuit D_n. Figure 3 represents the most important fragment of the circuit.

The estimations for $Gate(D_n)$ and $Depth(D_n)$ follow directly from the construction. It remains to verify that, for $w \in \{0,1\}^n$, the value $D_n(w)$ is 1 if and only if M accepts w. Note that the last is equivalent to the statement that the relation $Hist_w$ holds for a triplet $(T(n), j, (q_A, y))$, for some $0 \leq j \leq T(n)+1$ and $y \in \Sigma$. By the equation for the gate Win, $D_n(w) = 1$ if and only if one of the gates $Hist(T(n), j, (q_A, y))$ has value 1.

The argument stems from the following.

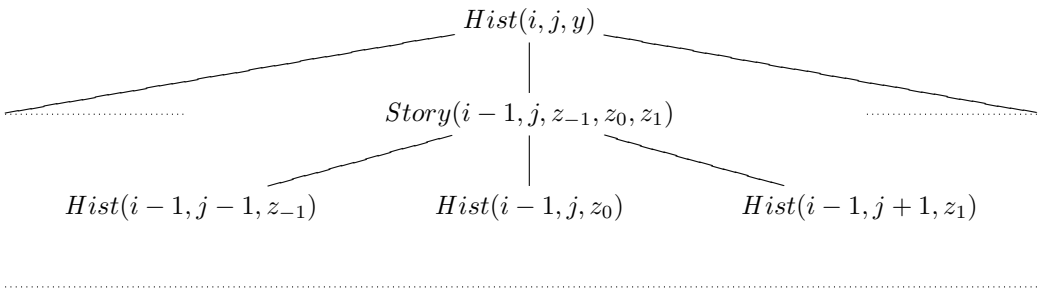


Figure 3: Fragment of circuit D_n.

Claim. Let $\{0,1\}^n \ni w = w_1 \dots w_n$. For a gate identified by p, let $p(w)$ denote the value of this gate under the valuation $x_k \mapsto w_k$, for $k = 1, \dots, n$. Then $Hist(i,j,\sigma)(w) = 1$ if and only if the relation $Hist_w(i,j,\sigma)$ holds.

The proof is by induction on $i = 0, 1, \dots, T(n)$.

For $i = 0$, let us consider $2 \leq j \leq n$. If $w_j = 1$ then the gate $Hist(0,j,1)$ receives value 1, and the gate $Hist(0,j,0)$ receives value 0. Moreover, the constant gates $Hist(0,j,\sigma)$ with $\sigma \neq 0,1$ have also value 0 by the clause (35).

Similarly, if $w_j = 0$ then the gate $Hist(0,j,0)$ receives value 1, and all other gates $Hist(0,j,\sigma)$ receives value 0. The argument for $i = 1$ is similar, and the claim for the constant gates with $i = 0$ follows directly from definition.

To show the induction step, suppose that the claim holds for some $i < T(n)$. Let us consider $1 \leq j \leq T(n)+1$. If a gate $Hist(i,j,y)(w)$ has value 1 (where $y \in \Sigma \cup (Q \times \Sigma)$) then there must be some z_{-1}, z_0, z_1, such that $z_{-1}z_0z_1 \vdash_M y$, and the gates $Hist(i-1,j-1,z_{-1})$, $Hist(i-1,j,z_0)$, and $Hist(i-1,j+1,z_1)$, have also value 1 (see Figure 3). By induction hypothesis, the relations $Hist_w(i-1,j-1,z_{-1})$, $Hist_w(i-1,j,z_0)$, and $Hist_w(i-1,j+1,z_1)$ hold. Since $z_{-1}z_0z_1 \vdash_M y$, then $Hist_w(i,j,y)$ holds as well.

Conversely, suppose $Hist_w(i,j,y)$ holds. By definition of computation, there are some z_{-1}, z_0, z_1, such that the relations $Hist_w(i-1,j-1,z_{-1})$, $Hist_w(i-1,j,z_0)$, and $Hist_w(i-1,j+1,z_1)$ hold. By inductive hypothesis, the corresponding gates have value 1. By Lemma 2 (uniqueness of y), $z_{-1}z_0z_1 \vdash_M y$. Hence $Hist(i,j,\sigma)(w) = 1$ (again, see Figure 3). The case of $j = 0$ can be treated similarly, and the case of $j = T(n)+2$ follows immediately from the equations (36-37). This completes the proof of the theorem. □

Remark. The construction described above has also a natural game interpretation. Ms. Or claims that w is accepted by M, while Mr. And claims the opposite. To support her claim, Ms. Or makes it more specific: in the last moment of computation, $T(n)$, the machine is in the accepting state q_A and scans the cell j which contains symbol y. Mr. And challenges this claim and then Ms. Or provides z_{-1}, z_0, z_1 such that $z_{-1}z_0z_1 \vdash_M (q_A, y)$, and claims that these (generalized) symbols have occupied the locations $j-1, j$, and $j+1$ in the moment $T(n)-1$. Then Mr. And wants to see the proof for one of these three cases, and situation repeats. When eventually the players come back to the moment 0, i.e., the initial configuration, the truth of the claim of Ms. Or is verified directly.

Now, if w is indeed accepted by M, Ms. Or has an obvious strategy: always tell the truth. In contrast, if w is not accepted by M then the first claim of Ms. Or is false. Mr. And has then a strategy: maintain the false. In this way, Ms. Or will loose in the final checking.

3.4 Simulating circuits by machines with advice

A natural question is whether Theorem 5 can be reversed, i.e., whether a sequence of circuits of bounded size can be always simulated by a Turing machine. A simple example shows that this may not be true.

Example. Let $A \subseteq \mathbb{N}$. Consider a sequence of circuits $(C_n)_{n\in\mathbb{N}}$, where C_n is given by an equation

$$\text{Output} = \begin{cases} x_1 & \text{if } n \in A \\ \overline{x_1} & \text{otherwise .} \end{cases}$$

That is, the only gates of C_n are input gates and negated input gates, and (consequently) there are no wires. The Output gate is x_1 or $\overline{x_1}$ depending on whether $n \in A$. Thus the size of the circuit $C(x_1, \ldots, x_n)$ is as small as possible.

The language recognized by this sequence is

$$\{1w : |w| + 1 \in A\} \cup \{0w : |w| + 1 \notin A\}.$$

Clearly, by varying A, we can obtain uncountably many languages, hence also non-computable ones.

However, Theorem 5 can be reversed in terms of the non-uniform recognizability (12).

Theorem 6. *Suppose $L \subseteq \{0,1\}^*$ is recognized by a sequence of circuits $(C_n)_{n\in\mathbb{N}}$ with*

$$\mathit{Size}(C_n) = \mathcal{O}(T(n)).$$

Then L can be recognized by a Turing machine with advice $\left(M, (k_n)_{n\in\mathbb{N}}\right)$ in time $T(n)^2$.

Proof. We let k_n be an encoding of the circuit C_n (c.f. (18)). The machine M evaluates the circuit C_n for an input $w \in \{0,1\}^n$. The evaluation of a Boolean circuit is a well-known algorithmic problem. With the equational representation, the problem boils down to solving a system of equations like (18) (without cycles). We start by substituting the values $w_1, \ldots, w_n$, for $x_1, \ldots, x_n$, and $\overline{w_1}, \ldots, \overline{w_n}$ for $\overline{x_1}, \ldots, \overline{x_n}$, respectively. Then we run over the system several times, replacing identifiers by their value, whenever possible. (For example, if an equation is $p_7 = \text{Or } (p_3, 1)$, we replace p_7 by 1 throughout the system.) This naive algorithm takes $2 \cdot \mathit{Gate}(C_n) \cdot \mathit{Size}(C_n)$ steps, which already yields the bound required in the theorem.

Some better algorithms can be found (see Exercise 3.5.1.14). □

3.5 Exercises

3.5.1 Circuits — basics

1. **Negation.** Extend the definition of circuits by allowing gates with fan-in 1 labeled by $\neg$. The value of such gate is negation of a value of its unique child. Show that an extended circuit can be transformed to a semantically equivalent circuit in our form, such that the number of gates and the depth will not increase.

   How the rules of the game $\mathit{Game}(C, v)$ should change in the extended case ?

2. **Restricted fan-in.** In a *restricted circuit*, the fan-in of the Or-gates and And-gates is at most 2. Describe a transformation of a circuit into a semantically equivalent restricted circuit; estimate the change of parameters.

3. Prove Proposition 3 by giving an explicit representation in terms of equation system (like in (18)) of a circuit computing an arbitrary Boolean function $f : \{0,1\}^n \to \{0,1\}$.

4. Suppose there is an algorithm which, for each $n \in \mathbb{N}$, computes a circuit $C_n(x_1, \ldots, x_n)$. Show that in this case, the set recognized by the sequence $C_n(x_1, \ldots, x_n)$ is computable (in the sense of Section 2.2).

5. Describe all functions $f : \{0,1\}^n \to \{0,1\}$ with the property that if α and β differ precisely by one bit then $f(\alpha) \neq f(\beta)$.

6. *The parity function* $P_n : \{0,1\}^n \to \{0,1\}$ is defined as $P(x_0 \ldots x_{n-1}) = \sum_{i=0}^{n-1} x_i \mod 2$. Construct a circuit computing P_3 and do the computation for the input 011.

7. Let $A_n : \{0,1\}^{2n} \to \{0,1\}^{n+1}$ be the mapping which adds two binary sequences $x_0 \dots x_{n-1}$ and $x_n \dots x_{2n-1}$. Construct a circuit of size $O(n)$ with n outputs computing A_n.

8. Define $M_n : \{0,1\}^n \to \{0,1\}$ as 1 if $\sum_{i=1}^n x_i \geq n/2$ and as 0 otherwise. Construct a circuit of size $O(n \log n)$ computing M_n. *Hint.* One can apply Exercise 7.

9. Show that any regular language $L \subseteq \{0,1\}^*$ can be recognized by a sequence of circuits of polynomial size and depth $\mathcal{O}(\log n)$, even if we require that fan-in of each gate is at most 2.

   Hint. The proof is easiest if we apply the monoid recognizability of regular languages.

10. A regular language is *star free* if it can be represented by a star-free regular expression

$$R := a|\emptyset|(\neg R)|(RR)|(R+R)$$

    Show that a star-free regular language can be recognized by a sequence of circuits of polynomial size and constant depth. (Here we assume arbitrary fan-in of the gates.)

11. Consider circuits where, instead of Or and And gates, we use only one type of the gate, namely *Majority* gate, along with a constant gate *false*. The Majority gate with n inputs is like M_n from Exercise 8. Show that a circuit with K gates can be transformed to an equivalent circuit of this new type with $\mathcal{O}(K)$ gates.

12. Show that the functions P_n of Exercise 6 can be computed by circuits with Majority gate (Exercise 11) of polynomial size and constant depth.

    Remark. A difficult result by Furst, Saxe, and Sipser says that it is not possible for And-Or circuits.

13. Prove that the function M_n is computable using a circuit of size $O(n)$.

    Remark. A circuit is *monotone* if it uses only OR and AND gates. The following strengthening of this exercise was proved by Hoory, Magen and Pitassi: M_n is computable using a monotone circuit of size $O(n)$ and depth $O(\log(n))$.

14. Try to optimize an algorithm in the proof of Theorem 6. Find a representation of a circuit, for which a linear (in the sense of model RAM) algorithm is possible.

    Hint. The algorithm should propagate the value in bottom-up manner.

3.5.2 Size of circuits

1. Show that, for any $\epsilon < 1$, for sufficiently large n, there is a Boolean function $f : \{0,1\}^n \to \{0,1\}$ that cannot be computed by a circuit with 2^{n^ϵ} gates.

   Hint. Estimate the size of binary word encoding a circuit with 2^{n^ϵ} gates.

2. ([6]) Show that, for sufficiently large n, there is a Boolean function $f : \{0,1\}^n \to \{0,1\}$ that cannot be computed by a *restricted circuit* (see Exercise 3.5.1.2) with $\frac{2^n}{2n}$ gates.

3. Show that any Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a (unrestricted) circuit with $\frac{2^{n+1}}{n} + 4n$ gates.

   Hint. Present f by

$$f(x_1, \dots, x_n) = \text{Or } \{\text{And } \{x_{k+1}^{a_{k+1}}, \dots, x_n^{a_n}, f(x_1, \dots, x_k, a_{k+1}, \dots, a_n)\} : (a_{k+1}, \dots, a_n) \in \{0,1\}^{n-k}\},$$

   where $x^a = (x \wedge a) \vee (\bar{x} \wedge \bar{a})$. Design a circuit which computes f using the functions $f(x_1, \dots, x_k, a_{k+1}, \dots, a_n)$ as *black boxes* (bb). Then design a circuit (with variables $x_1, \dots, x_k$) computing the bb's. The first level is common for all bb's, and consists of the gates And $(x_1^{a_1} \wedge \dots \wedge x_k^{a_k})$. The actual function $f(x_1, \dots, x_k, a_{k+1}, \dots, a_n)$ is computed by an Or-gate wired with an appropriate subset of the first level.

4. Prove that any function $f : \{0,1\}^n \to \{0,1\}$ is computable by a restricted circuit of size $1000\frac{2^n}{n}$.

5. Improve the bound of Exercise 3 above, by showing that any Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a (unrestricted) circuit with $\mathcal{O}(2^{\frac{n}{2}})$ gates.

4 Polynomial time

4.1 Problems vs. languages

In complexity theory, like in algorithmics, we are interested in solving real problems, which may involve various kind of data: natural numbers, graphs, finite sets, formulas. It is usually straightforward to represent these data as words over a finite alphabet. Hence we can identify a decision problem with recognition of a language.

For example, consider a problem of *connectivity* of graphs:

> Given an undirected graph, is any path of vertices connected by a path ?

One possible representation of a graph (usually not most efficient, but convenient) is by the *incidence matrix*. We have introduced it in Exercise 2.5.1.2 for directed graphs, but an undirected graph can be identified with a directed *symmetric* graph (i.e., there is an edge from i to j iff there is an edge from j to i).

So, the problem above can be represented by the sets of words $w \in \{0,1\}^*$, such that w is an incidence matrix of some undirected (i.e., symmetric) connected graph. For example, the word 0100101101010110 is in the language, whereas $000001010, 010001100, 1111111$, are not. Note that a machine recognizing this language should in particular verify that the word is of length n^2, for some n, and that it represents a symmetric graph.

In most cases the properties related to the representation are easy to verify, and we may abstract from them while discussing examples of problems on abstract level. So we may think that an algorithm takes as an input, e.g., a graph, or a formula of propositional calculus. One important exception concerns the number theoretic problems: it is essential if a natural number n is given in unary, or in binary representation (in the latter case, the size of the representation is $\lfloor \log_2 n \rfloor + 1$). The former representation is seldom used, and we will clearly distinguish it by writing n in unary as 1^n. On the other hand, the choice of a k-ary representation, for $k \geq 2$, is not essential for complexity (see Exercise 4.7.1.1).

Similar remarks apply to function problems. It should be noted, however, that we mean *discrete* problems here, which use data with unambiguous finite representation. Problems occurring in continuous mathematics may require *approximation*, e.g., of real numbers. It is a subject of *numerical analysis*, which will not be considered in this lecture, although several theories of complexity over reals have been also proposed.

4.2 Uniform case: P

If a language $L \subseteq \Sigma^*$ is recognized by a Turing machine working in time n^k, for some $k \geq 1$, we say that it is recognized *in polynomial time*. We denote the class of all such languages by P. In other words,

$$P \quad = \quad \bigcup_{k \in \mathbb{N}} DTIME(n^k). \tag{38}$$

Note that, because of the linear speed-up property of Turing machines (Exercise 2.5.4.3) we need not use the $\mathcal{O}$ notation here. We sometimes write $f(n) = n^{\mathcal{O}(1)}$ to mean that $f(n) \leq n^k$, for some $k \geq 1$.

The definition of the polynomial time is very robust, it does not depend on whether we use a single tape, multiple tape, or off-line Turing machine. In fact, we could consider models closer to real computers (like the Random Access Machine, RAM), and the class of languages recognized in polynomial time would remain the same. Also, restriction to alphabet $\{0,1\}$ is not essential here, as a language over a larger alphabet Σ can be always encoded over $\{0,1\}$ preserving the polynomial-time recognizability.

4.2.1 Functions

Any Turing machine can be viewed not only as accepting/rejecting, but also computing a (partial) function. For complexity considerations, it will be convenient to extend the off-line model (section 2.2) to an *input-output* Turing machine. This machine, in addition to a read-only input tape, has also a *write-only output tape* (initially empty). We assume that the head of the output tape can only move right or wait. We refer to such a machine as a k-tape input-output machine if additionally it has k working tapes. Formally, we present the input tape as the first, and the output tape as the last tape. Hence, an initial configuration is of the form

$$(q_I, \triangleright @w \triangleleft, \underbrace{\triangleright @B, \ldots, \triangleright @B}_{k}, @B).$$

The transition function is of the type

$$\delta : (Q - \{q_A, q_R\}) \times (\Sigma_{i/o} \cup \{\triangleright, \triangleleft\}) \times \Sigma^k \to Q \times \Sigma^k \times \Sigma_{i/o}\{L, R, Z\}^{k+1} \times \{R, Z\}.$$

We say that a machine M as above computes a function $f : dom\, f \to \Sigma^*_{i/o}$, (where $dom\, f \subseteq \Sigma^*_{i/o}$), if $L(M) = dom\, f$ and, for any input $w \in dom\, f$, the last component of the accepting configuration, representing the content of the output tape, is precisely $f(w)$.

We let FP denote the class of all functions $f : \Sigma^*_{i/o} \to \Sigma^*_{i/o}$ computed by a machine as above in polynomial time.

4.3 Non-uniform case: *P/poly*

We let *P/poly* denote the class of languages L recognized in polynomial time by a Turing machine with advice $\left(M, (k_n)_{n\in\mathbb{N}}\right)$ (see Section 3.1), where moreover k_n is of polynomial size, i.e., $|k_n| \leq n^c$, for some constant $c \geq 1$.

Remark. The second assumption could be omitted as the assumption about polynomial time implies that only polynomial-length advices are useful (Exercise 4.7.2.1).

The considerations of the previous section allows us to characterize this class in terms of circuits.

Theorem 7. *For a language $L \subseteq \{0,1\}^*$, the following conditions are equivalent.*

1. *$L \in P/poly$,*
2. *L is recognized by a sequence of circuits C_n of polynomial size, i.e.,*

$$Size(C_n) \leq a \cdot n^k,$$

for some constants $c, k \in \mathbb{N}$.

Proof. (2) $\Rightarrow$ (1) follows directly from Theorem 6.

To show (1) $\Rightarrow$ (2), suppose L is recognized by a pair $\left(M, (k_n)_{n\in\mathbb{N}}\right)$ in time $T(n)$. Without loss of generality, we may assume that $|k_n| = p(n)$, where $1^n \mapsto 1^{p(n)}$ is computable in polynomial time. We apply the construction of Theorem 5 to the machine M. Consider the circuit $D_{p(n)+n}$ constructed in the proof of this theorem. By construction it satisfies, for $w \in \{0,1\}^n$,

$$D_{n+p(n)}(w) = 1 \quad \Leftrightarrow \quad k_n w \in L(M). \tag{39}$$

Note that the size of this circuit is bounded by $T^2(p(n)+n)$, hence polynomial. We obtain C_n from $D_{n+p(n)}$ by replacing the first $p(n)$ input gates by constant gates representing the bits of k_n. (Specifically, x_i is replaced by *true* or *false* depending on whether the i-th bit of k_n is 1 or 0, respectively.) Next, we rename the (not

replaced) input gates $x_{p(n)+1}, \ldots, x_{p(n)+n}$ by $x_1, \ldots, x_n$. By construction and (39), we have $C_n(w) = 1$ iff $k_n w \in L(M)$, and the size of C_n is polynomial in n.

A similar reasoning gives us a characterization of P.

Theorem 8. *For a language $L \subseteq \{0,1\}^*$, the following conditions are equivalent.*

1. *$L \in P$,*
2. *L is recognized by a sequence of circuits $(C_n)_{n\in\mathbb{N}}$ of polynomial size, and moreover the function that, for an argument 1^n outputs an encoding of circuit C_n is computed in polynomial time.*

Proof. (2) $\Rightarrow$ (1) Given w of length n, we compute the circuit C_n in polynomial time; hence C_n has polynomial size. We have to evaluate $C_n(w)$. Recall from the proof of Theorem 6 that evaluating a circuit of size $S(n)$ can be made in time $S(n)^2$. Thus we obtain a polynomial-time algorithm for L.

(1) $\Rightarrow$ (2) follows directly from Theorem 5 if we realize that the construction $1^n \mapsto D_n$ described in the proof of this theorem works in time polynomial in $T(n)$, hence in polynomial time in our case. □

4.4 Time vs. space

We let L denote the class of languages K recognized in *logarithmic space*, i.e., in space $c \cdot \log n$, for some c. Note that, by Exercise 2.5.4.4, we can omit the constant factor c, so

$$L \quad = \quad DSPACE(\log n). \tag{40}$$

We let *PSPACE* denote the class of languages K recognized in *polynomial space*, i.e., in space n^k, for some $k \geq 1$. In other words,

$$PSPACE \quad = \quad \bigcup_{k\in\mathbb{N}} DSPACE(n^k). \tag{41}$$

Clearly, a k-tape Turing machine working in time $T(n)$ can visit at most $k \cdot T(n)$ cells on the working tapes. Hence a machine working in polynomial time, works in polynomial space as well.

On the other hand, if a machine works in space $\log n$, we can assume by Theorem 4 that it does not loop, i.e., no configuration is repeated. As a configuration is described by the content of the working tapes, the location of a head on the input tape, and the state, there are no more than $n \cdot c^{\log n} \leq d \cdot n^k$ configurations (for some constants c, d, k).

Hence we have

$$L \subseteq P \subseteq PSPACE. \tag{42}$$

We currently do not know if the inclusions $L \subseteq P$ and $P \subseteq PSPACE$ are proper. We are able, however, to show that the outermost inclusion $L \subseteq PSPACE$ is proper. This will be obtained as a corollary from a more general fact. Recall that, for a function $S : \mathbb{N} \to \mathbb{N}$, $DSPACE(S(n))$ denoted the class of all languages recognized by in space $S(n)$ (c.f. Section 2.4).

Theorem 9 (Space hierarchy)**.** *Suppose that a function $S_2(n)$ is space constructible (Exercise 2.5.4.2) and satisfies $S_2(n) \geq \log n$, and a function $S_1(n)$ satisfies*

$$\limsup_{n\to\infty} \frac{S_2(n)}{S_1(n)} \quad = \quad \infty.$$

Then there exists a language

$$L \in DSPACE(S_2(n)) - DSPACE(S_1(n)).$$

Proof. For the sake of this proof, a *standard machine* is an off-line machine with one working tape, over the input/output alphabet $\{0,1\}$, using as auxiliary symbols only $B, \rhd, \lhd$. We define an encoding of standard machines by words in $\{0,1\}^*$ in a similar manner as we have done it for single-tape machines (Section 2.3); we omit the details. Let $\langle M \rangle$ denote the encoding of a standard machine M.

Claim. There exists an off-line Turing machine H (over input/output alphabet $\{0,1\}$) which satisfies the following properties, for an input word of the form $1^k\langle M \rangle$ (with $n = |1^k\langle M \rangle|$):

$$1^k\langle M \rangle \in L(H) \iff 1^k\langle M \rangle \in L(M) \tag{43}$$

$$\text{if } M \text{ uses space } f(n) \text{ then } H \text{ uses space } f(n) + 2\log n. \tag{44}$$

We define a machine H similarly to the universal machine (Section 2.3), with some necessary modifications. H has two working tapes in addition to the input tape. Suppose the input tape contains the word $1^k\langle M \rangle$. The working tape 1 simulates the working tape of M in its computation for the input $1^k\langle M \rangle$. The working tape 2 keeps an information about the actual state of M, encoded in binary, and the actual location of the input head of M, also encoded in binary. With these two new ingredients, the simulation of M by H is very similar as in the case of an universal Turing machine, yielding the equivalence (43); we omit the details. The space restriction (44) follows immediately from construction.

We will define our language *via* a machine D similar to the diagonal machine considered in Section 2.3. Like H, D is an off-line machine with two working tapes. Its program can be described by the following instructions.

1. If an input is not in the form $1^k\langle M \rangle$, for some standard machine M and $k \geq 1$ then **stop reject**.
2. Otherwise, mark $S_2(n)$ cells on the working tape 1. (It is possible, because the function $S_2(n)$ is constructible.)
3. Then act in the same way as H for the input $1^k\langle M \rangle$, but if H tries to leave the marked cells on the tape 1 then **stop reject**.
4. If H terminates its computation then
   (a) if H accepts then **stop reject**,
   (b) if H rejects then **stop accept**.

Let $L = L(D)$. We first verify that $L \in DSPACE(S_2(n))$. By construction of H and D, the space used by D, for an input of length n, is $S_2(n) + 2\log n$. By the assumption that $\log n \leq S_2(n)$, this space can be bounded by $3 \cdot S_2(n)$. By the linear space compression (Exercise 2.5.4.4), we can reduce it to $S_2(n)$, as required.

To show that $L \notin DSPACE(S_1(n))$, suppose to the contrary that $L = L(M)$, for some machine M working in space $S_1(n)$. Of course, M need not be standard: it may use a huge number of tapes, and a huge auxiliary alphabet. However, it can be easily transformed to an equivalent standard machine M_D, working in space $c \cdot S_1(n)$, for some constant c (c.f. Exercise 2.5.2.1). Moreover, by Theorem 4, we can assume that M_D halts for every input.

Now, by the assumption (43), we can find k, such that $n = |1^k\langle M_D \rangle|$ satisfies

$$c \cdot S_1(n) < S_2(n).$$

Consider the computation of D for the input $1^k\langle M_D \rangle$. By the above inequality, the exception mentioned in the point 3 will not happen: the space $S_2(n)$ is sufficient to carry on the simulation of H to the end. Now, by the rule 4, we have

$$\begin{aligned} D \text{ accepts } 1^k\langle M_D\rangle &\Leftrightarrow H \text{ rejects } 1^k\langle M_D\rangle \\ &\Leftrightarrow M_D \text{ rejects } 1^k\langle M_D\rangle \quad \text{(by (43))}. \end{aligned}$$

This contradicts the assumption that M_D is equivalent to D. Hence L cannot be recognized in space $S_1(n)$, as required. □

Corollary 1. *$L \neq PSPACE$.*

Proof. It is enough to take $S_2(n) = n$ and $S_1(n) = \lceil \log n \rceil$. □

We will note one more interesting consequence of the Hierarchy Theorem. We already know that the sum in (41) is indeed infinite, as any $DSPACE(n^k)$ is a proper subclass of $DSPACE\left(n^{k+1}\right)$. How is P related to these classes ? For example, is it possible that $P \subseteq DSPACE(n)$, or $DSPACE(n) \subseteq P$? At present, we cannot exclude any of these two possibilities, but nevertheless we can show the following.

Corollary 2. *$P \neq DSPACE(n)$.*

Proof. Suppose, to the contrary, that the equality holds. Let

$$L \in DSPACE(n^2) - DSPACE(n). \tag{45}$$

Such a language exists by Theorem 9, and we can assume $L \subseteq \{0,1\}^*$. Let $\perp$ be a fresh symbol, and let

$$L^\perp = \{w\perp^{|w|^2-|w|} : w \in L\}.$$

Note that a word in $L^\perp$ consists of a prefix in L of some length n prolonged by a sequence of $\perp$'s, so that the whole word has the length n^2. To recognize $L^\perp$, we can use a $DSPACE(n^2)$ machine recognizing L. The new machine first verifies that the input is in $\{0,1\}^*\perp^*$ and the sequence of $\perp$'s has an appropriate length (this is possible in $DSPACE(\log n)$). Then it simulates the machine for L on the maximal $\{0,1\}^*$ prefix of the input. Note that we calculate the complexity with respect to the length of the whole input (which is n^2). Hence we can conclude that

$$L^\perp \in DSPACE(n).$$

By assumption ($DSPACE(n) \subseteq P$), there is a machine M recognizing $L^\perp$ in time n^k, for some $k \geq 1$. Then we can recognize L in the following way.

Given an input $w \in \{0,1\}^*$ of length n, prolong it by $\perp^{n^2-n}$ and then apply M.

The new machine works in time $\mathcal{O}(n^{2k})$, hence $L \in P$. By assumption ($P \subseteq DSPACE(n)$), $L \in DSPACE(n)$, contradicting (45). □

Remark. It is easy to show an analogous result for any $DSPACE(n^k)$ instead of $DSPACE(n)$ along the same lines. Hence, we can conclude

$$(\forall k)\ P \neq DSPACE(n^k). \tag{46}$$

4.5 Alternation

We will now consider a yet another model of computation, somewhat in between Turing machines and Boolean circuits. It is organized as a machine, but allows for multiple computation paths, which may co-exist simultaneously.

We present here a simple single-tape variant; an extension to multi-tape and off-line variants is straightforward.

An *alternating Turing machine* can be presented as a tuple

$$M = \langle \Sigma, \Sigma_{i/o}, B, \triangleright, Q, Q_\exists, Q_\forall, q_I, q_A, q_R, \delta \rangle, \tag{47}$$

where the items are as in the definition of deterministic Turing machine (1) with two modifications.

1. The set of states Q is partitioned into sets $Q_\exists$ and $Q_\forall$, i.e., $Q_\exists \cup Q_\forall = Q$, and $Q_\exists \cap Q_\forall = \emptyset$ ($Q_\exists$ or $Q_\forall$ can be empty). We call the states in $Q_\exists$ *existential*, and the states in $Q_\forall$, *universal*.

2. $\delta \subseteq (Q - \{q_A, q_R\}) \times \Sigma \times Q \times \Sigma \times \{L, R, Z\}$ is a relation, not necessarily a function. The notation $q, a \to p, b, D$ means that $(q, a, p, b, D) \in \delta$. We assume[7] that, for each $q \in Q - \{q_A, q_R\}$, and $a \in \Sigma$, there is always some triple p, b, D, such that $q, a \to p, b, D$.

The concept of (initial/accepting/rejecting) configuration, and the next-step relation is defined in the same way as for deterministic machine (c.f. (4)); the distinction between existential and universal states does not matter here. Note however that there can be more than one configuration C', such that $C \to_M C'$; we call any such configuration a *successor* of C. We call a configuration (q, α) *existential* (*universal*) if so is the state q.

The acceptance is defined in terms of the *configurations winning for* M (or *winning configurations*, for short).

- An accepting configuration is winning.
- An existential configuration is winning if at least one of its successors is winning.
- An universal configuration is winning if all its successors are winning.
- No other configuration is winning.

A word $w \in \Sigma^*_{i/o}$ is *accepted* if the initial configuration $(q_I, \triangleright w)$ is winning. As usual, the language $L(M)$ recognized by an alternating Turing machine M is the set of words accepted by M.

Remark. Alternatively, the acceptance by an alternating machine can be defined in terms of a game played by Ms. Or, and Mr. And, similar to the game we have considered for circuits on the page 22. Ms. Or wants to show that an input w is accepted, while Mr. And claims the opposite. The arena is the configuration graph of M defined as in the proof of Theorem 4. The players start in the initial configuration for w and then move the token down the graph. The move is selected by Or in the existential configurations and by And in the universal ones. It is not difficult to show that M accepts w if and only if Ms. Or has a winning strategy in this game.

A *computation path* is any sequence

$$(q_I, \triangleright @ w) = C_0 \to_M C_1 \to_M C_2 \to_M \dots$$

We say that an alternating Turing machine M *works in time* $T(n)$ if any computation path has length at most $T(n)$, for an input of length n. We let $ATIME(T(n))$ denote the class of all languages recognized by an alternating Turing machine working in time $T(n)$.

Similarly, we say that an alternating Turing machine M *works in space* $S(n)$ if the number of cells of the working tapes visited on a computation path is bounded by $S(n)$, for an input of length n. We let $ASPACE(S(n))$ denote the class of all languages recognized by an alternating Turing machine working in space $S(n)$.

We are not aware of an analogue of Theorem 4 for alternating Turing machines. However, if an alternating machine M works in space bounded by a constructible function $S(n) \geq \log n$, we may assume without loss of generality that M halts for every input (c.f. Exercise 2.5.4.5).

Now we define the alternating analogues of the classes L and P.

$$AL = \bigcup_{c \geq 1} ASPACE(c \cdot \log n) \tag{48}$$

$$AP = \bigcup_{k \in \mathbb{N}} ATIME(n^k). \tag{49}$$

[7] This is for simplicity only.

Theorem 10 (Chandra, Kozen, Stockmeyer)**.**

$$AL = P \tag{50}$$
$$AP = PSPACE. \tag{51}$$

Proof. We first show a simulation of deterministic machines by alternating ones.

$\underline{P \subseteq AL}$ Let $L = L(M)$, for some deterministic machine M working in time n^k, for some k. The idea is similar to simulation of machines by circuits presented in section 3.3, and we will refer to the notation introduced there. In some sense, an alternating machine M' will play a game of Ms. Or and Mr. And associated with the circuit constructed for the machine M in the proof of Theorem 5. We may assume that M is a single-tape machine (like in (1)); our alternating machine M' will be an of–line machine with one working tape. We describe the computation of M' in terms of "big" steps which usually consist of several elementary transitions.

For an input w of length n, M' will have existential configurations containing (encoding of) expressions $Hist(i, j, y)$, where $0 \leq i \leq n^k$, $0 \leq j \leq n^k + 2$, and $y \in \Sigma \cup (Q \times \Sigma)$ is a generalized symbol. Similarly, M' will have universal configurations containing (encoding of) expressions $Story(i, j, z_{-1}, z_0, z_1)$, where i and j are as above and z_{-1}, z_0, z_1 are generalized symbols (see page 23). We will refer to them as to "configurations of type $Hist$" or "configurations of type $Story$", respectively. Note that the length of expressions of both kinds is $\mathcal{O}(k \cdot \log n)$.

From the initial configuration, M' uses existential states to generate an expression $Hist(i, j, (q_A, \sigma))$, where $0 \leq i \leq n^k$, $0 \leq j \leq n^k+1$, q_A is the accepting state of M, and $\sigma \in \Sigma$. (The pair (q_A, σ) is a generalized symbol.) Intuitively, by choosing this expression, player Ms. Or "declares" that the input w is accepted by M because, at the moment i, the machine enters the accepting state q_A, while the head is scanning the cell number j containing symbol σ. In other words, the configuration containing $Hist(i, j, (q_A, \sigma))$ will be winning for M' iff the predicate $Hist_w(i, j, (q_A, \sigma))$ holds true for M (see the proof of Theorem 5, page 23, for definition of $Hist_w$). The construction of M' will guarantee that, in general, a configuration of M' with an expression $Hist(i, j, y)$ will be winning iff the predicate $Hist_w(i, j, y)$ holds true.

If $i > 0$ then, from a configuration with expression $Hist(i, j, y)$, M' goes in existential states to a configuration of type $Story$, containing an expression $Story(i-1, j, z_{-1}, z_0, z_1)$, for some generalized symbols z_{-1}, z_0, z_1, such that $z_{-1}z_0z_1 \vdash_M y$. From this configuration, M' goes in universal states to a configuration of type $Hist$ with one of the expressions $Hist(i, j-1, z_{-1})$, $Hist(i-1, j, z_0)$, or $Hist(i-1, j+1, z_1)$. Recall that the resulting configuration is again existential.

Similarly as for the circuit, some extra rules apply to a configuration $Hist(i, j, y)$, for extremal values of j. If $j = 0$ then the respective universal configuration has the form $Story(i-1, j, z_0, z_1)$ (with $z_0z_1 \vdash_M y$). If $j = n^k + 2$ then the machine does not generate the expression of type $Story$, but stops and accepts whenever $y = B$ (blank) and rejects otherwise[8].

Finally, for a configuration of type $Hist(i, j, y)$ with $i = 0$, the machine M' verifies deterministically if $Hist_w(0, j, y)$ holds true. That is, the configurations containing

$Hist(0, 0, \triangleright)$, $Hist(0, 1, (q_I, w_1))$, $Hist(0, j, w_j)$, for $2 \leq j \leq n$

$Hist(0, j, B)$, for $n + 1 \leq j \leq n^k + 2$,

are winning, and all others are not.

The correctness of the construction stems from the following.

Claim. A configuration of type $Hist$ with an expression $Hist(i, j, y)$ is winning for M' if and only if the predicate $Hist_w(i, j, y)$ holds true for M.

The proof goes by induction on $i = 0, 1, \ldots, n^k$, similarly as the proof of an analogous Claim in the proof of Theorem 5; we leave the details to the reader.

[8]Because the cell number $n^k + 2$ will never be reached in the computation of M on w.

Also, it is clear from the construction that the machine M' uses space $\mathcal{O}(k \cdot \log n)$.

PSPACE ⊆ AP Let $L = L(M)$, for some deterministic Turing machine M working in space n^k. Without loss of generality, we can assume that M has only one tape. We also assume that M halts on all inputs (Theorem 4). Let us call a configuration of M *n-admissible* if it uses no more than n^k cells[9]. Clearly there is a constant d, such that the number of n-admissible configurations is bounded by $2^{n^{dk}}$ (for $n \geq 2$). Consequently, any computation for an input of length n takes at most $2^{n^{dk}}$ steps.

We now describe an alternating machine M' simulating M. Given an input w of length n, M' starts by generating some n-admissible accepting configuration C_{acc}. This is done in existential states. Intuitively, Ms. Or declares that M can reach the configuration C_{acc} from the initial configuration C_{init}. By remark above, this should happen in no more than $2^{n^{dk}}$ steps. The checking of this claim can be best described as a call of a procedure:

call procedure $\mathit{Reach}(C_{init}, C_{acc}, n^{dk})$.

Here $\mathit{Reach}(C_1, C_2, m)$ is a recursive procedure which takes as parameters two n-admissible configurations C_1 and C_2, and a number $m \leq n^{dk}$, given in binary[10]. Its goal is to verify that M can reach C_2 from C_1 in no more than 2^m steps. To this end, the machine first checks deterministically if $C_1 = C_2$ or $C_1 \to_M C_2$ (i.e., if M can reach C_2 from C_1 in no more than *one* step); if it is the case, M' accepts. Otherwise, if $m = 0$ then M' rejects. If $m > 0$ then M' generates in non-deterministic states an admissible configuration C. Then, in universal state, it chooses one bit of information, say *left* or *right*. Then, according to the result, it calls $\mathit{Reach}(C_1, C, m-1)$ or $\mathit{Reach}(C, C_2, m-1)$, respectively.

It can be easily shown by induction on m that whenever, in some configuration, M' calls the procedure $\mathit{Reach}(C_1, C_2, m)$, then this configuration is winning for M' if and only if there is a computation of M, which goes from C_1 to C_2 in no more than 2^m steps. This proves the correctness of M'.

To estimate the computation time, note that, along one computation path, $\mathit{Reach}(C_1, C_2, m)$ is called for strictly descendant values of m, so there are at most n^{dk} calls. Execution of the body of *Reach* requires reading of C_1 and C_2, and possibly generating C. As these configurations are n-admissible, this can be done in $\mathcal{O}(n^k)$ steps. Therefore, the length of any computation path can be estimated by $\mathcal{O}(n^{(d+1)k})$, i.e., by a polynomial, as required.

We now move to a simulation of alternating machines by deterministic ones.

AL ⊆ P A deterministic machine M', simulating an alternating machine M for an input w, performs a DFS through the graph of configurations of M reachable from the initial configuration. At the same time, M' finds the qualification of a node (winning or loosing), and accepts if the initial configuration turns out to be winning[11]. As the size of the graph of reachable configurations is polynomial with respect to $|w|$, M' works in polynomial time, as required.

AP ⊆ PSPACE Let M be an alternating machine working in time n^k. We cannot use the method from the previous paragraph (simple DFS), because the graph of reachable configurations can be too big. Instead we explore the fact that the *depth* of this graph is polynomial, because the length from an initial configuration to a terminal one is bounded by the computation time of M. A deterministic machine M' simulating M, given an input w of length n, performs a DFS algorithm in the *tree* of configurations of M reachable from the initial configuration (rather than the configuration graph). That is, M' does not memorize the whole configuration graph at the expense of loosing track of which configurations have been already visited in course of DFS.

[9] That is, it can be represented by $(q, \triangleright\alpha)$, where $|\alpha| \leq n^k$, c.f. (3).
[10] This assumption is not essential, as anyway m is polynomially related to $|C_i|$.
[11] A similar algorithm has been used for evaluation of Boolean circuits in Theorem 6.

In the simple version of the algorithm, M' remembers the whole computation path leading to the actually visited configuration of M. As this path has length at most n^k, and each configuration has size $\mathcal{O}(n^k)$, this gives the bound of $\mathcal{O}(n^{2k})$ for the space used by M', which is sufficient for our purpose.

In fact, we can obtain a better bound of $\mathcal{O}(n^k)$ if, instead of the whole computation path, M' remembers only three generalized symbols of each configuration (plus the whole actual configuration), see Exercise 4.7.5.2. □

4.6 Non-deterministic case: *NP*

4.6.1 Existential decision problems and search problems

Many algorithmic problems have the pattern: *find whenever exist*. Let us see few examples.

Compositeness. Given a natural number $n \geq 2$ (in binary), find a natural divisor k of n different from 1 and n itself, or answer that n is prime.

Hamiltonicity. Given an undirected graph[12], find a cycle which visits each node exactly once (Hamiltonian cycle), or answer that the graph is not Hamiltonian.

Satisfiability. Given a formula of propositional calculus, find a valuation which makes the formula true, or answer that the formula is unsatisfiable.

In the above, divisor, cycle, or valuation can be viewed as a *solution* to the problem, or a *witness* that a particular instance satisfies a required property. Sometimes, the mere existence of a witness is uninteresting, but we search for a maximal or minimal one (in appropriate sense).

Max Clique. Given an undirected graph, find a largest set of vertices X, such that each pair of distinct vertices in X is connected by an edge (i.e. a largest *clique*).

Min Vertex Cover. Given an undirected graph, find a set of vertices X of minimal size, such that each edge is incident with at least one vertex in X.

Observe that the optimization problems above can be reduced to the search problems of the first kind. For example, consider the problem

k **Clique.** Given an undirected graph with n vertices and a number $k \leq n$, find a clique of size k, or answer that there is no such clique[13].

Now the Max Clique problem can be reduced to solving k Clique, for $k = n, n-1, \ldots$, and stopping at the first k, where the answer is positive. The Min Vertex Cover problem can be solved similarly. We will study reduction between problems more precisely in the next section.

An apparently more restricted kind of problems consists of the *existential problems*, where we only ask if a solution (witness) exists. For example

Is a number composite ?
Is a graph Hamiltonian ?
Is a formula satisfiable ?

The relation between existential and search problem is, in general, not completely understood. On positive side, the Satisfiability problem can be reduced to its existential version in the following sense.

[12] We present problems informally, according the convention of Section 4.1.

[13] In this problem, it is inessential if k is given in binary or unary, as its size is in any case dominated by the representation of the graph.

Lemma 3. *Suppose an algorithm A decides in time $T(n)$, whether a given formula φ is satisfiable (where n is the size of φ). Then there is an algorithm A', which moreover finds an evaluation satisfying φ whenever it exists, acting in time $\mathcal{O}(n \cdot T(n))$.*

Proof. We present A' by a pseudo-code.

Given:	$\varphi(x_1, \ldots, x_k)$		
If	$A(\varphi) = No$	then	answer: **unsatisfiable**
else	$\psi := \varphi$		
For	$i = 1, \ldots, k$	do	
	If $A(\psi[\mathbf{0}/x_i]) = Yes$	then	$\psi := \psi[\mathbf{0}/x_i]$
			$x_i := \mathbf{0}$
	else		$\psi := \psi[\mathbf{1}/x_i]$
			$x_i := \mathbf{1}$

If $A(\varphi) = Yes$ then the valuation of $x_1, \ldots, x_k$ computed by this algorithm satisfies the formula φ. □

The situation is quite different for the Compositeness problem, where we do know a polynomial-time algorithm for the existential problem: this is the celebrated Agrawal–Kayal–Saxena primality test (first published in 2002). However, no efficient algorithm is known to find a non-trivial divisor whenever it exists. Indeed, a great deal of the public-key cryptography, including the RSA cryptosystem, is based on the assumption that no such algorithm exists.

Hence, we may verify that a number is composite without finding a witness. But what exactly does it mean: *to be a witness* ?

Indeed, much before the AKS test, it was known that there can be other witnesses of compositeness than divisors. For example:

1. If there is $a \not\equiv \pm 1 \bmod n$ such that $a^2 \equiv 1 \bmod n$ then n is composite (a is a non-trivial square root of 1).
2. If there is a such that $a^n \not\equiv a \bmod n$ then n is composite (a is a *Fermat witness*).

There is however a difference between the two kind of witnesses. Given a non-trivial square root a of 1, we can easily find a divisor as well. Indeed, if p is a prime factor of n, we have $a^2 \equiv 1 \bmod p$, but then $a \equiv 1 \bmod p$ or $a \equiv -1 \bmod p$. Hence we can find p by the Euclid algorithm (c.f. Exercise 2.5.4.7) applied to the pair $(n, a-1)$ or $(n, a+1)$.

On the other hand, no polynomial-time algorithm is known to find a divisor from a Fermat witness.

We will now give a precise definition of the concept of witness.

4.6.2 Polynomial relations and their projections

Let Σ be an alphabet. For a relation $R \subseteq \Sigma^* \times \Sigma^*$, let $\exists R$ be its projection to the first component, i.e.,

$$\exists R \quad = \quad \{x : (\exists y)\, R(x, y)\}. \tag{52}$$

For simplicity, assume $|\Sigma| \geq 2$; we can then fix an encoding of a pair of words (x, y) by a single word $\alpha(x, y)$, such that $|\alpha(x, y)| \leq 2|x| + |y|$ (c.f. Exercise 2.5.1.1). We say that a relation R is in the class P if so is the corresponding language $\{\alpha(x, y) : R(x, y)\}$. We call a relation R *polynomial* if it satisfies the following two conditions:

$$R \quad \text{is in} \quad P \tag{53}$$

$$(\forall x, y)\, R(x, y) \quad \Rightarrow \quad |y| \leq p_R(|x|), \tag{54}$$

for some polynomial p_R.

The class NP consists of those languages L, which can be presented as $L = \exists R$, for some polynomial relation R.

In this context, we say that the relation R *verifies* language L, and we call y a *witness* for x, whenever $R(x, y)$. Not that, by definition, a witness, if any, must be "short" (of the length polynomially related to $|x|$), and the property of being a witness must be efficiently checkable.

It is easy to see that the problems considered at the beginning of this section are in the class NP (when encoded as languages). For example, let R consist of the pairs of words (x, y), such that $x \in \{0,1\}^{n^2}$ is an incidence matrix of a symmetric graph over the set of vertices $\{1, \ldots, n\}$, and y is a word of the form $y = y_1 \ldots y_n$, satisfying the following conditions. Each y_i is a binary word of length $\lfloor \log n \rfloor + 1$ representing some number $a_i \in \{1, \ldots, n\}$, such that $a_i \neq a_j$, whenever $i \neq j$, and there is an edge from a_i to a_{i+1}, for $i = 1, \ldots, n-1$, and from a_n to a_1. Clearly, the relation R is polynomial and the language $\exists R$ represents the set of (undirected) graphs with Hamiltonian cycles.

Remark Both conditions in the definition of polynomial relation are essential; omitting any of them would yield a trivial extension of NP. Indeed, for any computable language $L = L(M)$ (c.f. Section 2.2), we can view an accepting computation of M on x as a witness for x. This relation is computable in polynomial time, but the witnesses are, in general, very long. On the other hand, any language L can be presented by $\exists R$, for $R = \{(x, \varepsilon) : x \in L\}$. Here, any witness has length 0, but clearly the complexity of R is the same as that of L.

The last observation yields an easy inclusion of the complexity classes:

$$P \quad \subseteq \quad NP. \tag{55}$$

The question whether this inclusion is strict ($P \stackrel{?}{=} NP$) is perhaps the most famous open question of computer science.

An alternative characterization of the class NP can be given in terms of non-deterministic Turing machines.

A *non-deterministic Turing machine* is an alternating machine (c.f. (47)), where all states are existential, i.e., $Q = Q_\exists$ and $Q_\forall = \emptyset$. Thus, a non-deterministic machine accepts an input word if there is a path (at least one) from the initial configuration to an accepting one.

Proposition 4. *A language L is in* NP *iff $L = L(M)$, for some non-deterministic Turing machine working in polynomial time.*

Proof. ($\Rightarrow$) Let $L = \exists R$, for a polynomial relation R. A non-deterministic machine recognizing L acts as follows. Given an input x, the machine uses existential states to generate a word y, $|y| \leq p_R(|x|)$ (c.f. (54)). Then it checks deterministically if $R(x, y)$ holds and accepts if it is the case.

($\Leftarrow$) Let $L = L(M)$, for a non-deterministic machine M working in time $p(n)$, for some polynomial p. Let R consist of the pairs (x, y), where y encodes a sequence of $\leq p(|x|)$ transitions of M, such that if M follows these transitions then it accepts x. By construction, R is polynomial and $L = \exists R$. □

4.7 Exercises

4.7.1 *P*

1. For $k, n \in \mathbb{N}$, $k \geq 2$, let $rep_k(n)$ be a k-ary representation of n, i.e., a unique word $a_m a_{m-1} \ldots a_1 a_0$ over alphabet $\{0, 1, \ldots, k-1\}$, such that $n = a_m k^m + a_{m-1} k^{m-1} + \ldots + a_1 k + a_0$ and $a_m \neq 0$ or $n = m = 0$. For a set $A \subseteq \mathbb{N}$, let

$$rep_k(A) \quad = \quad \{rep_k(n) : n \in A\}.$$

Estimate the complexity of transformation

$$k, \ell, rep_k(n) \mapsto rep_\ell(n).$$

Deduce that, for any $A \subseteq \mathbb{N}$, and $k, \ell \geq 2$, $rep_k(A)$ is in P if and only if $rep_\ell(A)$ is in P.

4.7.2 *P/poly*

1. Suppose $\left(M, (k_n)_{n\in\mathbb{N}}\right)$ recognizes L in polynomial time without any restriction on the length of k_n. Show that $L \in P/poly$.

4.7.3 Easy cases of SAT

1. (Horn Clauses) Let X be a finite set of variables and ϕ be a CNF Boolean expression with variables from the set X such that each clause contains only one positive literal, that is all literals, except possible for one, are negations of variables, e. g. $\phi = (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$. Prove, that the SAT problem for these Boolean expressions is solvable in P. *Hint.* Consider a Boolean expression ϕ such that each clause contains only one positive literal and notice, that clauses with at least one non–negative variable one can re-write as implications. In order to find a truth assignment for ϕ start from an empty set T (all assignments are *false*) and for every implication add one variable to T.

2. Let X be a finite set of variables and ϕ be a CNF Boolean expression with variables from the set X such that each clause involves two variables, e. g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_1)$. Let G be a graph with the set of vertices $\{x, \neg x : x \in X\}$ and an edge from x_1 to x_2 if there is a clause in ϕ of the form $\neg x_1 \wedge x_2$ or $x_2 \wedge \neg x_1$. Prove that ϕ is unsatisfiable if and only if there are paths from x to $\neg x$ and from $\neg x$ to x.

3. Prove that 2SAT problem, that is the problem whether a Boolean expression in the form described in the previous exercise is satisfiable, is in P.

4. Prove that the (CNF) SAT problem, where each variable occurs at most twice is in P.

5. (The Timetable Problem) Let H be a finite set of hours, $T_1, \ldots, T_n \subset H$ be the set of hours during which ith teacher is available for teaching and $C_1, \ldots, C_m \subset H$ be the set of hours during which jth class is available for studying and $R_{ij} \in \mathbb{N}$ $(1 \leq i \leq n,\ 1 \leq j \leq m)$ be the number of hours ith teacher is required to teach jth class. The TT problem is to determine whether there exists a *meeting function* that is a mapping

$$f : \{1, \ldots, n\} \times \{1, \ldots, m\} \times H \to \{0, 1\}$$

such that

   (a) $\forall_{1\leq i\leq n, 1\leq j\leq m}\ f(i, j, h) = 1 \to h \in C_i \cap T_j$,
   (b) $\forall_{1\leq i\leq n, 1\leq j\leq m}\ \sum_{h\in H} f(i, j, h) = R_{ij}$,
   (c) $\forall_{1\leq i\leq n, h\in H} \sum_{i=1}^{n} f(i, j, h) \leq 1$,
   (d) $\forall_{1\leq j\leq m, h\in H} \sum_{j=1}^{m} f(i, j, h) \leq 1$.

   Assume, that for every $1 \leq i \leq n$ we have $|T_i| = 2$ and show that this 2–TT problem is solvable in P. *Remark.* This fact was noticed by S. Even, A. Itai and A. Shamir (1976) alongside a similar observation that 2SAT problem is solvable in $O(n)$ time.

4.7.4 Logarithmic space

1. Show that the set of well-formed bracket expressions is recognizable in L (logarithmic space).

2. Show that the evaluation of Boolean formulas (not circuits!) without variables can be done in L.

   Remarks. You may assume that the parentheses are obligatory. You may first consider the case where negation is applied only to variables.

3. Suppose we have to compute a term τ formed from constants and binary operations. We may use only instructions of the form $x := c$, and $x := f(y, z)$, where x, y, x are variables, c is constant and f a binary operation. How many variables we need w.r.t. the size of τ ?

   Hint. Think of the number of pebbles we need in a pebble game on a binary tree.

4. Let A be a finite algebra over a set of binary operations; we view all elements of A as constants. Considering A as fixed, show that evaluation of any term over A can be done in L. (This generalizes Exercise 1.)

   Hint. Show first that it can be done in $DSPACE\ (\log^2 n)$, using Exercise 3. Reduction to the single logarithm is quite tricky.

5. Suppose there is an algorithm which, for a number n (in unary) produces a circuit C_n of polynomial size and logarithmic depth, with the fan-in of Or-gates and And-gates at most 2. Show that the language M recognized by $(C_n)_{n\in\mathbb{N}}$ is in the class L.

   Hint. Note that you cannot store the circuit C_n in the memory of your machine. However, you can call the algorithm generating C_n again and again, whenever needed. Therefore, you can evaluate the circuit keeping in the memory only an identifier of the actual gate (of size $\log n$) and the path from the output gate to the actual gate (as a binary string).

4.7.5 Alternation

1. Define alternating finite automata over finite words in one-way, and two-way versions.

   (a) Show that the non-emptiness problem for these automata is decidable.

   (b) Show that these automata recognize only regular languages.

2. Assuming that a function $f(n) \geq n$ is space constructible show

$$ATIME(f(n)) \subseteq DSPACE(f(n)).$$

5 Reduction between problems

We often speak about reduction of one problem to another. For example, analytic or Cartesian geometry enables us to reduce problems of elementary geometry to questions of arithmetics and algebra. In section 4.6.1, we have seen that the problem of finding a satisfying assignment of propositional formula can be reduced to the question whether a satisfying assignment exists. On the other hand, we have remarked that no reduction is known of the problem of finding a prime factorization of an integer to the question whether a number is prime or composite.

Reductions between problems play a crucial role both in algorithmics and in complexity theory. Suppose we are able to reduce problem A to problem B. Then, if we find an algorithm for B, we can use it to solve A. On the other hand, if we don't know algorithm for B, we know at least that this problem is "as hard as A".

As usual, a precise definition of the intuitive concept of reduction presents some subtleties, and there are several reasonable variants of this notion.

5.1 Case study – last bit of RSA

We show here one concrete example of a reduction between problems. The algorithm RSA[14] is commonly used in public–key cryptography. Let us recall the basic assumptions.

The *public key* is given by the numbers n and e, where $n = p \cdot q$ is a product of two odd primes p and q, and $e \perp \varphi(n)$. Here $x \perp y$ means that x and y are coprime (relatively prime), and $\varphi(n)$ is the *Euler function* of n

$$\begin{aligned} \varphi(n) &= |\{a : a \perp n\}| \\ &= (p-1) \cdot (q-1). \end{aligned}$$

The *plaintext* ranges over $\{0, 1, \ldots, n-1\}$, and the *encryption* is given by a function

$$\{0, 1, \ldots, n-1\} \ni x \quad \mapsto \quad x^e \bmod n.$$

The (legal) *decryption* uses a *private key* d, where $e \cdot d \equiv 1 \pmod{\varphi(n)}$,

$$\{0, 1, \ldots, n-1\} \ni y \quad \mapsto \quad y^d \bmod n,$$

it explores the fact that $x^{e \cdot d} \bmod n = x$. An *attacker* wishes to compute the function

$$n, e, x^e \bmod n \quad \mapsto \quad x \tag{56}$$

without knowing d. It is of course possible *via* factorization of n and then computing the inverse of e modulo $\varphi(n)$; but it is broadly believed that this problem is computationally hard and consequently an efficient attacker does not exist.

Now consider a *weak attacker*, who only wishes to find the last bit of the plaintext

$$n, e, x^e \bmod n \quad \mapsto \quad x \bmod 2. \tag{57}$$

We will show that if a weak attacker succeeds then a ("strong") attacker succeeds as well. More precisely,

Proposition 5. *If an algoritm A computes the function (57) in time $T(\log n)$ then there is an algorithm A', which computes (56) in time polynomial of* $\max(T(\log n), \log n)$.

Proof. We start with the following observation, which holds for $x < n$, whenever n is odd.

$$(2x \bmod n) \bmod 2 = 1 \quad \Longleftrightarrow \quad \frac{n}{2} < x < n. \tag{58}$$

Thus, knowledge of the last bit of $2x \bmod n$ helps us to locate x in the lower or upper "half" of the interval $\{0, 1, \ldots, n-1\}$.

Similarly, it is easy to verify that

$$(4x \bmod n) \bmod 2 = 1 \quad \Longleftrightarrow \quad \left(\frac{1}{4}n < x < \frac{1}{2}n\right) \vee \left(\frac{3}{4}n < x < n\right).$$

More generally, the following observation will be useful.

[14] From the names of Ron Rivest, Adi Shamir, and Leonard Adleman. See [7] for more background on RSA.

Claim. For n odd and $i < n$, the i-th bit of the binary expansion of $\frac{x}{n}$ equals $\big(2^i x \bmod n\big) \bmod 2$.
For, we analyse the school algorithm of division x by n, which can be represented by a recursive system of equations[15]

$$\begin{aligned} p_1 &= x \\ 2p_i &= s_i \cdot n + p_{i+1}, \quad \text{with } p_{i+1} < n. \end{aligned}$$

As a result,

$$\frac{x}{n} = 0,\, s_1 s_2 \ldots$$

The claim follows from the observation that

$$\begin{aligned} p_i &= 2^{i-1} x \bmod n \\ s_i &= \big(2^i x \bmod n\big) \bmod 2, \end{aligned}$$

for $i = 1, 2, \ldots$, which follows recursively from the identity, which holds for odd n and any α,

$$2 \cdot (\alpha \bmod n) = (2\alpha \bmod n) \bmod 2 \cdot n + 2\alpha \bmod n.$$

Now, clearly x is completely determined if we know the subsequent bits $s_1\, s_2 \ldots s_i$, for $i \leq \lceil \log n \rceil$. By Claim, these digits can be obtained by computing $\big(2^i x \bmod n\big) \bmod 2$, for $i = 1, \ldots, \lceil \log n \rceil$.

Suppose we are given $n, e, x^e \bmod n$, as in (56) above, and we wish to compute x. The crucial fact is that, even without knowing x, we can, for given i, compute efficiently $(2^i x)^e \bmod n$, using the fact that

$$(2^i x)^e \bmod n = \big(2^{ie} \bmod n\big) \cdot (x^e \bmod n) \bmod n.$$

By assumption, the algorithm A, given

$$n, e, (2^i x)^e \bmod n = (2^i x \bmod n)^e \bmod n$$

computes $\big(2^i x \bmod n\big) \bmod 2$ in time $T(\log n)$. Hence, by calling it for $i = 1, 2, \ldots, \lceil \log n \rceil$, we can retrieve x as required. Considering the time $\mathcal{O}(\log^3 n)$ needed to compute $(2^i x)^e \bmod n$ (c.f. Exercise 2.5.4.6), we can estimate the time of the whole algoritm by $\mathcal{O}\big(\log n \cdot (\log^3 n + T(\log n))\big)$.

5.2 Turing reduction

The kind of reduction that we have seen in the example above has a natural interpretation in terms of programming: we make a program to solve problem A, having at our disposal a program solving problem B, although we may have no access to the source code of the latter. For Turing machines, this is realized in the concept of machines with oracles.

An *oracle Turing machine* $M^{\square}$ is defined similarly as a deterministic machine in Section 2.2 with one modification. We assume the machine has an additional tape, called *question tape*, and a special *question state* $q_?$. On the question tape, the machine can only write symbols[16] 0, 1, or B (blank). Originally, the question tape is empty, i.e., contains the marker $\triangleright$ followed by an infinite sequence of blanks. The *proper content* of the question tape is the $\{0,1\}$-word which occupies the leftmost cells of the tape until the first blank (we don't count $\triangleright$).

The machine $M^{\square}$ *per se* does not recognize any language; it has meaning only together with a language $K \subseteq \{0,1\}^*$, called *oracle*. The pair $(M^{\square}, K)$ is usually denoted by M^K.

[15] We can view this process as an infinite run of a finite automaton with states in $\{0, 1, \ldots, n-1\}$ and transitions $p \overset{2p \div n}{\to} 2p \bmod n$.

[16] To be precise, it also has to **re**-write the symbol $\triangleright$, whenever visiting the leftmost cell.

The computation of M^K on an input w is defined similarly as for deterministic Turing machine (Section 2.2) with one modification. Suppose the machine enters the state $q_?$ and the proper content of the question tape is $v \in \{0,1\}^*$. Then, before the machine makes the next step of the computation, the first blank cell of the question tape is rewritten by 1 if $v \in K$, and by 0 otherwise.

Note that is it reasonable (although formally not required) that the head ot the question tape scans the first blank cell while entering the state $q_?$; then the machine can immediately read the "answer" to its "question".

The acceptance is defined similarly as for ordinary Turing machines; we let $L(M^K)$ denote the language recognized by the machine $M^\square$ with oracle K.

The concept of time and space used by a machine with oracle is defined similarly as for ordinary Turing machines (the space of the question tape is counted as working space).

We say that a language A is *Turing reducible* to a language B, in symbols

$$A \quad \leq_T \quad B, \tag{59}$$

if there is an oracle Turing machine $M^\square$, such that M^B works in polynomial time, and $A = L(M^B)$.

5.3 Karp reduction

Let $A, B \subseteq \Sigma^*$, and let $f : \Sigma^* \to \Sigma^*$ be a function computable in polynomial time (see Section 4.2.1). We say that f *reduces* A to B in the sense of *Karp* if, for all $w \in \Sigma^*$

$$w \in A \quad \Leftrightarrow \quad f(w) \in B, \tag{60}$$

or, in other words,

$$f^{-1}(B) \quad = \quad A. \tag{61}$$

In this context, we call f a *polynomial* (or *Karp*) *reduction* (of A to B).

We say that a language A is *Karp reducible* to a language B, in symbols

$$A \quad \leq_K \quad B \tag{62}$$

if there is a polynomial reduction of A to B.

If $A \leq_K B$ then also $A \leq_T B$. Indeed, an oracle machine M^B recognizing A computes the reduction $f(w)$ and then asks only one question, whether $f(w) \in B$. It immediately accepts or rejects, following the answer received to the question.

Remark. Note that a polynomially computable function $f : \Sigma^* \to \Sigma^*$ may serve as reduction for many pairs of sets. Indeed, it follows from (61) that, for any $B \subseteq \Sigma^*$, $f^{-1}(B) \leq_K B$. In particular, if $A \leq_K B$ then also $\overline{A} \leq_K \overline{B}$ (where $\overline{X}$ is the complement of X), by the same reduction.

Note. In the literature, Turing reducibility $A \leq_T B$ is often understood as a reduction by a computable function (page 7), without any complexity bound. In this context, a Turing reduction in polynomial time is called *Cook reduction*, after Stephen Cook. (Karp reduction is called so after Richard M. Karp.) In modern literature (e.g., [6]), often a more restricted complexity bounds are considered. For example, it is required that a reduction $f : \Sigma^* \to \Sigma^*$ is computable in logarithmic space (thus *a fortiori* in polynomial time, but a converse implication is not known). In this lecture, we usually present classical polynomial-time (Karp) reductions, but in most cases they can be improved to logarithmic-space reductions. We leave this improvement as an exercise.

The Turing and Karp reductions allow us to reduce one existential problem (*does there exist ? ...*) to another. But in the case of *NP*-problems, we are usually interested in finding a solution, or witness to a question. Can a witness found for one problem be used to construct a witness for another problem ? This idea is captured by the concept of Levin reduction (called so after Leonid Levin).

5.4 Levin reduction

It is defined for relations, rather than languages. Let $R_1, R_2 \subseteq \Sigma^* \times \Sigma^*$. A *Levin reduction* consists of three functions: $f : \Sigma^* \to \Sigma^*$ and $g, h : \Sigma^* \times \Sigma^* \to \Sigma^*$; each of them computable in polynomial time, satisfying the following conditions[17].

1. $(\forall x, y)\, R_1(x, y) \Rightarrow R_2(f(x), g(x, y))$,
2. $(\forall x, z)\, R_2(f(x), z) \Rightarrow R_1(x, h(x, z))$.

Note that the above conditions imply that

$$(\forall x)\,(\exists y) R_1(x, y) \Leftrightarrow (\exists z) R_2(f(x), z)$$

hence f is a Karp reduction of $\exists R_1$ to $\exists R_2$.

We write $R_1 \leq_{Le} R_2$ if there exist f, g, h, satisfying the conditions above.

5.5 *NP*-completeness

A language L is *NP-hard in the sense of Karp* (or simply: *NP-hard*, for short) if, for any L' in *NP*, $L' \leq_K L$. It is *NP-complete in the sense of Karp* (or simply: *NP-complete*) if it is NP-hard and moreover is itself in the class *NP*.

We say that a language L is *NP-complete in the sense of Levin* if L is verified by a polynomial relation R (i.e., $L = \exists R$, see page 37), such that, for any polynomial relation R', $R' \leq_{Le} R$. Note that if L is NP-complete in the sense of Levin then it is also NP-complete in the sense of Karp.

We will now focus on the problem *Boole-Sat* of the satisfiability of Boolean circuits. More precisely, fix some encoding of Boolean circuits $C \mapsto \hat{C}$ (see (18)), and let R_{Boole} be the relation

$$R_{Boole} \quad = \quad \{(\hat{C}, w) : C \text{ is a circuit with } n \text{ variables, } w \in \{0,1\}^n, \text{ and } C(w) = \mathit{true}\}. \tag{63}$$

We let

$$\mathit{Boole\text{-}Sat} = \exists R_{Boole}$$

Theorem 11. *The problem Boole-Sat is NP-complete in the sense of Levin.*

Proof. Let $L = \exists R$, for some polynomial relation R (c.f. definitions on page 37). Without loss of generality, we can strengthen (54) by

$$(\forall x, y)\, R(x, y) \quad \Rightarrow \quad |y| = p_R(|x|). \tag{64}$$

We first adapt to polynomial relations the construction *machine* $\mapsto$ *circuit* of Theorems 5 and 8.

Claim 1. *There is a P-time computable function* $1^n \mapsto D_n(x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+p_R(n)})$*, such that, for* $v \in \{0,1\}^n$ *and* $w \in \{0,1\}^{p_R(n)}$*,*

$$D_n(v, w) = 1 \quad \Longleftrightarrow \quad R(v, w). \tag{65}$$

To show the claim, consider the language $\{\alpha(x, y) : R(x, y)\}$ defined in Section 4.6.2. As this language is in P, we have by Theorem 8 a polynomial mapping $1^{2n+p_R(n)} \mapsto E_{2n+p_R(n)}$, where the circuits E satisfy $E_{2n+p_R(n)}(u) = 1$ iff $u = \alpha(v, w)$ and $R(v, w)$. Recall that $\alpha(v, w) = v_1 0 v_2 0 \ldots v_{n-1} 0 v_n 1 w$. We transform the circuit $E_{2n+p_R(n)}$ into the circuit D_n be replacing the gates $x_2, x_4, \ldots, x_{2n-2}$ by 0, and the gate x_{2n} by 1[18]. We replace the negated gates $\overline{x_{2i}}, i \leq n$ accordingly, and rename the remaining gates in the obvious way. Clearly

[17] In full generality, we could admit different alphabets here, say $R_1 \subseteq \Sigma_1^* \times \Gamma_1^*$, $R_2 \subseteq \Sigma_2^* \times \Gamma_2^*$. In this case, the types of the functions would be: $f : \Sigma_1^* \to \Sigma_2^*$, $g : \Sigma_1^* \times \Gamma_1^* \to \Gamma_2^*$, and $h : \Sigma_1^* \times \Gamma_2^* \to \Gamma_1^*$. We can easily restrict ourselves to a single alphabet, e.g., by taking a superset of these alphabets (and slightly modifying the functions if necessary).

[18] Here we identify 0 and 1 with the constant gates *false* and *true* defined in (19) and (19), respectively.

$$D_n(v,w) \quad \text{iff} \quad E_{2n+p_R(n)}(\alpha(v,w)) \quad \text{iff} \quad R(v,w).$$

We are ready to define a Levin reduction of R to R_{Boole}. For $w \in \{0,1\}^n$, we let

$$f : v \mapsto H_n\left(x_1, \ldots, x_{p_R(n)}\right) =_{def} D_n\left(v, x_1, \ldots, x_{p_R(n)}\right).$$

That is, H_n is obtained from D_n by first replacing in D_n the gates $x_1, \ldots, x_n$ by the subsequent bits $w_1, \ldots, w_n$ and the negated gates $\overline{x_1}, \ldots, \overline{x_n}$ by $\overline{w_1}, \ldots, \overline{w_n}$ (c.f. footnote 18), and then renaming the remaining gates $x_{n+1}, \ldots, x_{n+p_R(n)}$ by $x_1, \ldots, x_{p_R(n)}$. The witnesses in both directions remain the same, i.e., we let

$$g(v,w) = h(v,w) = w.$$

It follows from (65) that, for all v and w,

$$R(v,w) \Leftrightarrow D_n(v,w) = 1 \Leftrightarrow R_{Boole}(f(v), w),$$

hence f, g, h constitute indeed a Levin reduction. □

Analogously, we define the satisfiability problem for propositional formulas and subclasses of formulas.

We fix a countable set of variables *Var*.
The set of *propositional formulas* is defined by the following rules.

1. *true* and *false* are formulas,
2. a variable $x \in Var$ is a formula,
3. if φ is a formula then so is $(\neg\varphi)$,
4. if $\varphi_1, \ldots, \varphi_k$, are formulas, for $k \geq 1$, then so are
   (a) $(\varphi_1 \vee \ldots \vee \varphi_k)$, and
   (b) $(\varphi_1 \wedge \ldots \wedge \varphi_k)$.

A *literal* is a variable or negation of a variable, $(\neg x)$. A *clause* is a disjunction

$$(\ell_1 \vee \ldots \vee \ell_k),$$

where $\ell_1, \ldots, \ell_k$ are literals. If $k = 0$, we assume by convention that the clause equals *false*.

A formula is in *conjunctive normal form* (*CNF*) if it is a conjunction

$$(\psi_1 \wedge \ldots \wedge \psi_m),$$

where $\psi_1, \ldots, \psi_m$ are clauses. If $m = 0$, we assume by convention that the formula equals *true*.

A formula is in 3-*CNF* if it is in *CNF* and moreover each clause consists of at most 3 literals.

The formuals are evaluated in the Boolean algebra $\{0,1\}$. Given a formula φ and a valuation v of all the variables of φ, the value $\varphi[v]$ is defined as expected. If the variables of φ are $x_1, \ldots, x_n$, we will identify a valuation $x_i \mapsto w_i$ with a word $w = w_1 \ldots w_n$.

Like for circuits, we assume some standard encoding of formulas over some fixed alphabet; we will not distinguish between a formula and its encoding. Let the relation R_{prop} be defined by

$$R_{prop} \quad = \quad \{(\varphi, w) : \varphi[w] = 1\} \tag{66}$$

and let the relations $R_{CNF-prop}$ and $R_{3-CNF-prop}$ be subsets of R_{prop} obtained by restricting the first component to formulas in *CNF* and in 3-*CNF*, respectively.

Let

$$\begin{aligned} \mathit{Sat} &= \exists R_{\mathit{CNF-prop}} && (67)\\ \mathit{CNF\text{-}Sat} &= \exists R_{\mathit{CNF-prop}} && (68)\\ \mathit{3\text{-}CNF\text{-}Sat} &= \exists R_{\mathit{3-CNF-prop}}. && (69) \end{aligned}$$

Theorem 11 infers the following.

Corollary 3 (Cook-Levin Theorem)**.** *The problems Sat, CNF-Sat, and 3-CNF-Sat are NP-complete in the sense of Levin.*

Proof. We first show a Levin reduction of Boole-Sat to Sat. Clearly, a Boolean circuit is always equivalent to some formula, but a natural construction which unravels a circuit, may yield a formula which is exponentially bigger[19]. Instead, for a circuit C, we will construct a formula φ_C with a bigger number of variables, such that satisfiability of C is equivalent to satisfiability of φ_C.

We first demonstrate the construction on example. Consider the circuit of Figure 1 and its labelling by identifiers represented on Figure 2. We consider the identifiers as propositional variables and form a formula, which describes the circuit

$$\begin{aligned} &(p_1 \Leftrightarrow x_1) \wedge (p_2 \Leftrightarrow x_2) \wedge (p_3 \Leftrightarrow x_3) \wedge (p_4 \Leftrightarrow \neg x_1) \wedge (p_5 \Leftrightarrow \neg x_2) \wedge (p_6 \Leftrightarrow \neg x_3) \wedge \\ &\quad (p_7 \Leftrightarrow (p_1 \vee p_2)) \wedge (p_8 \Leftrightarrow (p_1 \vee p_5)) \wedge (p_9 \Leftrightarrow (p_2 \vee p_4)) \wedge (p_{10} \Leftrightarrow (p_4 \vee p_5)) \wedge \\ &\quad\quad (p_{11} \Leftrightarrow (p_7 \wedge p_3 \wedge p_{10})) \wedge (p_{12} \Leftrightarrow (p_8 \wedge p_9 \wedge p_6)) \wedge (p_{13} \Leftrightarrow (p_{11} \vee p_{12})) \wedge p_{13} \end{aligned}$$

Here, for convenience, we have used the connective of *equivalence*, where $\alpha \Leftrightarrow \beta$ is an abbreviation of $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$, and $\alpha \Rightarrow \beta$ is an abbreviation of $\neg\alpha \vee \beta$.

In general, for a circuit $C(x_1, \ldots, x_n)$ with M gates, our formula φ_C has variables $x_1, \ldots, x_n, p_1, \ldots, p_M$, and is a conjunction of the identifier of the output gate (p_M), and the equivalence formulas ($p_i \Leftrightarrow \ldots$), defining all the gates of the circuit. A crucial property is that, for a valuation w of the variables $x_1, \ldots, x_n$, there is a unique extension

$$\tilde{w} : \{x_1, \ldots, x_n, p_1, \ldots, p_M\} \to \{0, 1\}$$

which makes all the equivalence formulas true, and the value $\tilde{w}(p_i)$ equals to the value of the gate (identified by) p_i under the valuation w. Hence, the whole formula φ_C is true iff the value of p_M is true iff $C[w] = 1$.

Hence, the function f of the Levin reduction is realized by the transformation $C \mapsto \varphi_C$ described above, the function g by $w \mapsto \tilde{w}$, and the function h restricts a valuation v of the variables in $\{x_1, \ldots, x_n, p_1, \ldots, p_M\}$ to the variables in $\{x_1, \ldots, x_n\}$. We leave to the reader verification that all these functions are computable in polynomial time.

We will now show the NP-completeness of CNF-Sat.

A closer analysis shows that the formula φ_C constructed above is almost in a CNF form. By definition, it can be rewritten as a conjunction of implications

Indeed, the implications

$w : \{x_1, \ldots, x_n\} \to \{0, 1\}$, there is a unique extension $\tilde{w} : \{x_1, \ldots, x_n\} \to \{0, 1\}$

The above construction $C \mapsto \varphi_C$ realizes

[19] It is however an open problem if it *must* be the case; see [6], Problem 15.5.4.

6 Randomized polynomial time

In this section we will only consider polynomial relations R (see Section 4.6), where the condition (54) is strengthened in the following way: there is a polynomial r, such that

$$(\forall x, y)\, R(x, y) \quad \Rightarrow \quad |y| = r(|x|). \tag{70}$$

(We adopt the convention that the parameter of a relation is denoted by the same letter in the lower case.) The above restriction is for convenience only; note that the class *NP* can be defined using polynomial relations satisfying (70). We make an additional proviso that the alphabet in consideration is $\{0, 1\}$.

The class *NP* has been defined in terms of the existence of a *witness*: an input x is accepted if there exists a witness y, such that the pair (x, y) satisfies a polynomial relation R. The probabilistic class *BPP* will be defined in terms of the *number* of witnesses. Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$ be a polynomial relation. For a fixed n we consider a random variable $U_{r(n)}$ taking the values in the finite set $\{0,1\}^{r(n)}$ with uniform probability, i.e.,

$$(\forall u \in \{0,1\}^{r(n)})\ \Pr(U_{r(n)} = y) = \frac{1}{2^{r(n)}}. \tag{71}$$

Note that, for any property α,

$$\Pr(\alpha(U_{r(n)})) = \sum_{\alpha(y)} \Pr(U_{r(n)} = y) = \frac{|\{y : \alpha(y)\}|}{2^{r(n)}}. \tag{72}$$

We say that that a polynomial relation R *verifies* a language $L \subseteq \{0,1\}^*$ *with bounded error* if

$$(\forall x)\ x \in L \ \Rightarrow\ \Pr(R(x, U_{r(|x|)})) \geq \frac{3}{4} \tag{73}$$

$$(\forall x)\ x \notin L \ \Rightarrow\ \Pr(R(x, U_{r(|x|)})) < \frac{1}{4}. \tag{74}$$

The above two conditions can be summarized in one:

$$(\forall x)\ \Pr\big(x \in L \iff R(x, U_{r(|x|)})\big) \geq \frac{3}{4}. \tag{75}$$

If (75) holds, we write $L = \exists^{bpp} R$. The class *BPP* consists of those languages L, which can be presented as $L = \exists^{bpp} R$, for some polynomial relation R.

Note that, in general, a polynomial relation need not define any language in the above sense; for this to be the case, it is necessary that, for any x, the proportion of the y's satisfying $R(x, y)$ to all y's of length $r(|x|)$ is either $\geq \frac{3}{4}$ or $< \frac{1}{4}$.

Remark A deterministic Turing machine M recognizing R can be viewed as an implementation of a *probabilistic polynomial time algorithm* recognizing L. Note that we are only interested in the inputs for M consisting of a (suitable encoded) pair (x, y), where $|y| = r(|x|)$. (The other inputs will surely be rejected.) For convenience, we may assume that M has two input tapes: the "main" input tape containing x, and a *random input tape* containing y. During the execution, the machine occasionally "tosses a coin", and this is implemented by reading of a subsequent bit of y. We are interested in the *probability* of an event that M has eventually accepts x, and this is precisely $\Pr R(x, U_{r(|x|)})$, whenever M and L satisfy (75).

The use of $\frac{3}{4}$ in (75) is not essential. To show this, the following technique will be useful. Suppose we repeat the choice of a random input y say $2m + 1$ times, thus selecting a sequence $y_1, \ldots, y_{2m+1}$. Possibly $R(x, y_i)$ holds for some i's, and fails for others, but we are interested in the result for the *majority* of i's.

Lemma 4. *Let $U^1_{r(n)}, U^2_{r(n)}, \ldots, U^{2m+1}_{r(n)}$ be independent*[20] *random variables with the same distribution as $U_{r(n)}$ (see (71)). Let $x \in \{0,1\}^n$, and suppose $\Pr\left(x \in L \iff R(x, U_{r(|x|)})\right) = \alpha$, for some $\alpha > \frac{1}{2}$. Then*

$$\Pr\left(\frac{|\{i : x \in L \iff R(x, U^i_{r(|x|)})\}|}{2m+1} > \frac{1}{2}\right) \geq 1 - (\alpha \cdot (1-\alpha) \cdot 4)^m. \tag{76}$$

Proof. We will estimate the probability of the opposite event, i.e., that only *less* than a half of i's satisfy $x \in L \iff R(x, U^i_{r(|x|)})$. By the independence of the $U^i_{r(n)}$'s, this amounts to

$$\begin{aligned}\sum_{j=1}^{m} \binom{2m+1}{j} \alpha^j \cdot (1-\alpha)^{2m+1-j} &\leq \underbrace{\sum_{j=1}^{m} \binom{2m+1}{j}}_{2^{2m}} \alpha^m \cdot (1-\alpha)^m \\ &= (\alpha \cdot (1-\alpha) \cdot 4)^m\end{aligned}$$

□

Corollary 4. *Let $\frac{1}{2} < \alpha < \beta$. Suppose L and R satisfy (75) with the constant α replacing $\frac{3}{4}$, i.e.,*

$$(\forall x)\ \Pr\left(x \in L \iff R(x, U_{r(|x|)})\right) \geq \alpha.$$

Then there is another polynomial relation R', such that L and R' satisfy

$$(\forall x)\ \Pr\left(x \in L \iff R'(x, U_{r'(|x|)})\right) \geq \beta.$$

Proof. Note that the mapping $z(1-z)$ achieves its maximum $\frac{1}{4}$ for $z = \frac{1}{2}$ and then decreases, hence, in particular, $(\alpha \cdot (1-\alpha) \cdot 4) < 1$. Therefore, we can find m, such that $(\alpha \cdot (1-\alpha) \cdot 4)^m < 1 - \beta$. We let $r'(n) = (2m+1) \cdot r(n)$, and, for any $x \in \{0,1\}^n$,

$$R'(x, y) \iff \frac{|\{i : R(x, y_i)\}|}{2m+1} > \frac{1}{2}, \tag{77}$$

where $y = y_1 y_2 \ldots y_{2m+1} \in \{0,1\}^{r'(n)}$, with $|y_1| = |y_2| = \ldots = |y_n|$. Using Lemma 4 (along with the monotonicity of the mapping $z(1-z)$), we obtain that $\Pr\left(x \in L \iff R(x, U_{r'(|x|)})\right) \geq \beta$, as required. □

Remark No *NP*-complete problem is known to be in *BPP*. Some researchers go even further, conjecturing that $BPP = P$.

One evidence for this conjecture is a non-uniform derandomization.

Theorem 12. BPP $\subseteq$ P/poly.

[20] Independence means that $\Pr\left(U^1_{r(n)} = y_1 \wedge U^2_{r(n)} = y_2 \wedge \ldots \wedge U^{2m+1}_{r(n)} = y_{2m+1}\right) = \Pr\left(U^1_{r(n)} = y_1\right) \cdot \Pr\left(U^2_{r(n)} = y_2\right) \cdot \ldots \cdot \Pr\left(U^{2m+1}_{r(n)} = y_{2m+1}\right)$.

Proof. Let $BPP \ni L = \exists^{bpp} R$, for a polynomial relation R (satisfying (75). Let R' be defined by (77), for some m. By Lemma 4 (along with the monotonicity of $z(1-z)$), we obtain that, for all $x \in \{0,1\}^n$,

$$\Pr\left(x \in L \iff R'(x, U_{r'(|x|)})\right) \quad \leq \quad \left(\frac{1}{4} \cdot \frac{3}{4} \cdot 4\right)^m = \left(\frac{3}{4}\right)^m.$$

This estimates the probability that a sequence $y = y_1 y_2 \dots y_{2m+1}$ is *not* a good witness for x. Note that a sequence $y \in \{0,1\}^{r'(n)}$ can be a good witness for some x, and bad for some other x'. Is there a sequence good for all $x \in \{0,1\}^n$?

Let us estimate that y is not good for *some* x, where $m = 3n$.

$$\begin{aligned}
\Pr\left((\exists x \in \{0,1\}^n)\, x \in L \iff R'(x, U_{r'(|x|)})\right) &\leq \sum_{x \in \{0,1\}^n} \Pr\left(x \in L \iff R'(x, U_{r'(|x|)})\right) \\
&\leq 2^n \cdot \left(\frac{3}{4}\right)^{3n} \\
&\leq \left(2 \cdot \frac{27}{64}\right)^n \\
&= \left(\frac{27}{32}\right)^n \\
&< 1.
\end{aligned}$$

Therefore, there exists some sequence $k_n = \tilde{y_1}, \tilde{y_2}, \dots, \tilde{y_{6n+1}}$, which is good for all $x \in \{0,1\}^n$, i.e., for all $x \in \{0,1\}^n$,

$$x \in L \iff R'(x, k_n).$$

As this holds for all $n \in \mathbb{N}$ and R' is recognized in polynomial time, we obtain a presentation (11) of L, proving that $L \in P/poly$. □

7 Polynomial space

Recall we have defined the class *PSPACE* in (41). A language L is *complete* in this class *PSPACE* w.r.t. polynomial (Karp) reductions (*PSPACE-complete*) if L is in *PSPACE* and, for any $M \in PSPACE$, $M \leq_K L$ (c.f. (62)). We will present in this section a problem which naturally extends the problem *Boole-Sat* discussed in Section 5.5. To this end, we first characterize the class *PSPACE* in terms of polynomial relations. Let R be a polynomial relation. We let

$$\widetilde{\exists\forall} R \quad = \quad \{x : \exists y_1 \forall y_2 \exists y_3 \dots Q y_{r(|x|)}\, R\left(x, y_1 y_2 \dots y_{p_R(|x|)}\right)\}, \tag{78}$$

where the variables y_i range over bits in $\{0,1\}$, the quantifiers alternate, and Q is $\exists$ of $\forall$ depending on whether $p_R(|x|)$ is odd or even, respectively.

Intuitively, $\widetilde{\exists\forall} R$ is a game version of $\exists R$. It comprises those x with the property that a witness y (satisfying $R(x,y)$) not only exists, but it can be constructed by a player who only chooses odd bits (1,3,5,...) against an adversary who chooses even bits (2,4,6,...).

Proposition 6. *A language L is in* PSPACE *iff $L = \widetilde{\exists\forall} R$, for some polynomial relation R.*

Proof. Not difficult using the characterization of *PSPACE* in terms of alternating Turing machines (Theorem 10, (51)). □

Recall the definition of the polynomial relation R_{Boole} and let

$$Q\text{-}Boole\text{-}Sat \quad = \quad \widetilde{\exists\forall} R_{Boole}. \tag{79}$$

Theorem 13. *The language Q-Boole-Sat is complete in PSPACE.*

Proof. The proof is analogous to the proof of Theorem 11, indeed the same reduction will work. Let $L \in PSPACE$; by Proposition 6, we have $L\widetilde{\exists\forall}R$, for some polynomial relation R. Without loss of generality we can assume that the witnesses have length exactly $p_R(|x|)$ (c.f. (64)). Consider the mapping $1^n \mapsto D_n(x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+p_R(n)})$ of Claim 1, satisfying (65), i.e.,

$$D_n(v, w) = 1 \iff R(v, w).$$

The reduction is

$$f : v \mapsto H_n\left(x_1, \ldots, x_{p_R(n)}\right) =_{def} D_n\left(w, x_1, \ldots, x_{p_R(n)}\right).$$

We have

$$\begin{aligned} v \in L &\iff \exists y_1 \forall y_2 \ldots Q y_{p_R(|x|)} R(v, y_1 y_2 \ldots y_{p_R(|x|)}) \\ &\iff \exists y_1 \forall y_2 \ldots Q y_{p_R(|x|)} D_n(v, y) = 1 \\ &\iff H_n\left(x_1, \ldots, x_{p_R(n)}\right) \in Q - Boole - Sat, \end{aligned}$$

as required. □

As for *Sat*, we have an analogous problem for propositional formulas. We extend the formation rules of page 5.5, by: if φ is a formula and x a variable then $\exists x\varphi$, $\forall x\varphi$ are formulas.

Let *QBF* be the set of all true quantified propositional sentences.

Corollary 5. *The language QBF is complete in PSPACE.*

Lemma 5 (Savitch)**.** *If f is fully space constructible and $f(n) \log n$ then*

$$NSPACE(f(n)) \quad \subseteq \quad DSPACE\left((f(n))^2\right)$$

Corollary 6. NPSPACE = PSPACE.

7.1 Interactive proofs

We start with a very informal example. Consider the following problem.

Given two graphs G_0 and G_1. Decide if they are **non-isomorphic**.

Here, we think of a directed graph presented by $\langle\{1, \ldots, n\}, E\rangle$, where $E \subseteq \{1, \ldots, n\}^2$. It is easy to see that the dual problem to decide if two graphs are isomorphic, is in *NP*, but we do not know if it is in *P*. We do not know if the non-isomorphism problem is in *NP*, and no probabilistic algorithm is known for this problem.

We will show a probabilistic *Algorithm*, which solves the problem fast at the expense of interacting with an entity called *Mathematician*. Mathematician has an insight and can solve any problem disregarding its computational difficulty. However, Algorithm must be cautious, as it is not completely sure to really interact with Mathematician, or maybe with a dishonest party who masquerades him.

Assume Mathematician claims: *G_0 or G_1 are not isomorphic.*

Here is a protocol to verify the claim.

Algorithm tosses a coin and thus chooses a graph G_i (with $i = 0$ or 1).

Algorithm further selects a *random copy* of G_i. More specifically, by tossing a coin an appropriate number of times, Algorithm selects a permutation $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$, and computes $H = \langle \{1, \ldots, n\}, E_\sigma \rangle$, where

$$E_\sigma(\sigma(i), \sigma(j)) \iff E(i, j).$$

Algorithm then presents H to Mathematician, asking **what is it?** (i.e., G_0 or G_1 ?)

Mathematician, thanks to the insight, gives the correct answer.

However, if the graphs *are* isomorphic, a dishonest party is in trouble. Indeed, the correct answer to the question is: **both**, but it is not what Algorithm expects ! A dishonest party has still a chance $\frac{1}{2}$ to guess the result of Algorithm's coin–tossing, and to give an expected answer. But by repeating the protocol m times, the probability that Algorithm is cheated is reduced to $\left(\frac{1}{2}\right)^2$.

We now proceed to the precise definitions.

We consider an alphabet $\{0, 1, ?\}$. By definition, a *strategy* is any partial mapping

$$S : (\{0,1\}^*?)^+ \to \{0,1\}.$$

We tacitly assume that a strategy is defined, whenever we use it. Intuitively, we view an argument of S as a sequence

$$\mathit{Question}_1,\ \mathit{Question}_2,\ \ldots,\ \mathit{Question}_k$$

The strategy gives an answer *yes* (1) or *no* (0) to the last question ($\mathit{Question}_k$). However, the answer may depend on the previous questions as well (thus the strategy is *adaptive*).

Verifier is a randomized Turing machine which, in addition to the working tapes, the input tape, and the random tape, has an *interactive tape*. This tape is initially empty. In course of computation, the machine can fill it with a word, say $w_1 \in \{0,1\}^*$, followed by '?'. Once the symbol '?' is put on the tape, the machine receives an answer a_1 written next to '?'. Thus the content becomes $w_1?a_1$. The machine makes a usual transition to read the answer, and then the interactive tape is cleared up. In course of the further computation, the machine can write down a second question w_2, it receives an answer a_2, and so on.

We consider a verifier which, for an input x of length n, works in time bounded by a polynomial $p(n)$, and uses a random string r of length equal to some polynomial function $r(n)$. Given x and r, the computation is determined if we additionally fix some strategy S. That is, if the machine asks questions $w_1, w_2, \ldots$ as described above, then the answer to the i-th question is

$$a_i \quad = \quad S\left(w_1?w_2? \ldots w_i?\right).$$

Note that, for an input of length n, the machine can ask at most $p(n)$ questions; therefore a strategy S can be viewed as a (huge but) finite object.

We write $V(x, r, S) = 1$ if the unique computation determined by x, r, S is accepting, and $V(x, r, S) = 0$, otherwise.

An interactive polynomial proof system is a pair (V, P), where V (given along with $p(n)$ and $r(n)$) is a verifier as above, and P, called *prover*, is a family of strategies $P = (P_x)_{x \in \{0,1\}^*}$. We say that (V, P), *recognizes* a language $L \subseteq \{0,1\}^*$ if, for any $x \in \{0,1\}^*$,

- if $x \in L$ then $\Pr\left(V(x, U_{r(|x|)}, P_x) = 1\right) \geq \frac{3}{4}$,
- if $x \notin L$ then, for **any** strategy S, $\Pr\left(V(x, U_{r(|x|)}, S) = 1\right) < \frac{1}{4}$.

Let *IP* be the class of languages recognizable by interactive polynomial proof systems.

Remark. Interactive proof systems can be viewed as a common generalization of both polynomial time non-deterministic Turing machines (which define *NP*) and polynomial-time bounded-error randomized Turing machines (which define *BPP*). Indeed, a randomized machine can be viewed as a system which makes no interaction, and a non-deterministic machine can be easily simulated by a system which makes no use of random bits, and whose strategy just tells the machine which non-deterministic move to select.

Consequently $NP \cup BPP \subseteq IP$. Indeed, the power of interactive proofs goes much (?) further.

Theorem 14 (Shamir)**.** *$IP = PSPACE$.*

We sketch the proof of an easier direction $IP \subseteq PSPACE$. Suppose a language L is recognized by an interactive proof (V, P). Observe first that this implies that

$$x \in L \iff \text{there } \textbf{exists } S, \text{ s.t. } \Pr\big(V(x, U_{r(|x|)}, S) = 1\big) \geq \frac{3}{4}.$$

The idea is to compute

$$\max_S \Pr\big(V(x, U_{r(|x|)}, S) = 1\big)$$

(where S ranges over all possible strategies) and compare it to $\frac{3}{4}$. Technically, we need not to compute the probability exactly, as a fraction; it is enough to find its numerator, which is the maximal cardinality

$$\max_S |\{r \in \{0,1\}^{r(|x|)} : V(x, r, S) = 1\}| \tag{80}$$

and compare it to $\frac{3}{4} \cdot 2^{r(|x|)}$; note that these numbers can be represented in polynomial space.

For simplicity of presentation, let us make a *proviso* that all questions asked by the verifier V for a given input of length n have the same length $q(n)$, for some polynomial q. (Clearly we can ensure this condition by a slight modification of (V, P) if necessary.)

We are going to describe a deterministic *PSPACE* algorithm recognizing L. Let us fix an input x of length n. We consider a tree T of all possible question–answer scenarios for the input x. More specifically, T has two kinds of nodes:

- **answer nodes** of the form $(q_1, a_1, q_2, a_2, \ldots, q_k, a_k)$,
- **question nodes** of the form $(q_1, a_1, q_2, a_2, \ldots, q_k, a_k, q_{k+1})$.

In this writing, q_i range over $\{0,1\}^{q(n)}$, a_i range over $\{0,1\}$, and k ranges over $\{1, 2, \ldots, p(n)\}$. Note that each question node v has exactly two successors $v0$ and $v1$, whereas each answer node is either a leaf or has $2^{q(n)}$ successors. By definition, the *root* ε is an answer node.

For each node v, we will define its *weight* with the intention that the weight of the root is the number we search for, i.e., (80). We say that a strategy S *agrees* with a node $v = (q_1, a_1, q_2, a_2, \ldots)$ if the answers $a_1, a_2, \ldots$ are those given by S, i.e., $a_i = S(q_1, q_2, \ldots, q_i)$.

We let

$$\mathit{weight}\,(v) \quad = \quad \max_{S_v} |\{r \in \{0,1\}^{r(|x|)} : V(x, r, S_v) = 1\}|$$

where S_v ranges over those strategies, which agree with the node v. Note that $\mathit{weight}\,(v)$ can be 0; it happens in particular if a scenario represented by the node v does not occur in any computation.

Intuitively, we think that Prover wishes to maximize the probability that Verifier will accept the input; then $\mathit{weight}\,(v)$ represents the maximal profit Prover can achieve assuming the initial question–answer scenario has been v.

In order to compute the weights, we will show that they satisfy the equations (81–83) below.

First, for each node v and a sequence of random bits $r \in \{0,1\}^{r(n)}$, let $enough\,(v,r)$ iff there exists a computation that uses the sequence r and whose complete question–answer scenario is v.

We claim the following.
If v is a leaf then

$$weight\,(v) \quad = \quad |\{r : enough\,(v,r)|, \tag{81}$$

if v is a question node then

$$weight\,(v) \quad = \quad \max(weight\,(v0), weight\,(v1)), \tag{82}$$

if v is an answer node then

$$weight\,(v) \quad = \quad |\{r : enough\,(v,r)\}| + \sum_{q \in \{0,1\}^{r(n)}} weight\,(vq) \tag{83}$$

These equations follow from the definition of weights. To prove (83), note that a random sequence r cannot agree with two scenarios vq and vq', for $q \neq q'$; so the sets of r's which contribute to the weights for different successors of v in T are disjoint.

Using the equations (81–83) above, we can show that $weight\,(v)$ can be computed for all nodes in polynomial space, where the actual polynomial depends on the parameters p, r, q. Note that, for each node v, the number $|\{r : enough\,(v,r)\}|$ can be computed in polynomial space by simply looping through all r and simulating V using v as a strategy. Moreover, all weights result from summing up some numbers in the above form. The algorithm visits the tree T in the DFS manner maintaining a stack with the path to the actually visited node, where each node on the path is additionally equipped with the part of its weight computed so far (according to the equations). We leave details to the reader.

8 Approximation algorithms

To be completed.

References

[1] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*, Cambridge University Press, 2009. `http://www.cs.princeton.edu/theory/complexity/`

[2] Oded Goldreich. *Computational Complexity: A Conceptual Perspective*, Cambridge University Press, 2008. `http://www.wisdom.weizmann.ac.il/~oded/cc.html/`

[3] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[4] Wojciech Jaworski and Damian Niwiński. *Information Theory. Synopsis of Lecture*, `http://www.mimuw.edu.pl/~wjaworski/ TI/notatki_20-12.pdf`, 2011.

[5] Ming Li and Paul Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*, Springer, 2008.

[6] Christos Papadimitriou. *Computational complexity*, Addison-Wesley, 1993. Wydanie polskie: *Złożoność obliczeniowa*, WNT Warszawa.

[7] Douglas R. Stinson. *Cryptography: Theory and Practice*, CRC Press, 2006. Wydanie polskie: *Kryptografia. W teorii i w praktyce*, WNT Warszawa.

Damian Niwiński, Warsaw University. Last modified: 21.5.2013.