Algorithmic Complexity

Spring 2006

Juhani Karhumäki

1 Introduction

The notion of an algorithm is one of the fundamental notions of constructive mathematics. Implicitly the study of algorithms has been a part of mathematics through its history. The oldest and one of the most well-known algorithms, Euclid's algorithm to compute the greatest common divisor, is more than 2000 years old. The analysis of its complexity, due to Lame in 1845, is one of the oldest such considerations. The term **algorithm** comes from the Persian mathematician Abu Abd-allah Mohammad ibn Musa al-Khwarizmi (ca. 780–ca. 850), who developped in 9th century the discrete algorithmic rules (used even today) to compute additions, subtractions, multiplications and divisions of decimal numbers.

The theory of algorithms, as a topic of its own, has born only in the 20th century, in particular in the second half of it. There are three rather separated directions in this research.

The oldest one, originating from 1930's, studies which problems are **in principal** algorithmically decidable, i.e. solvable by a computer. The theory started when Kurt Gödel (1906–1978) proved that there exist **algorithmically undecidable** problems. At the same time he destroyed the dream of David Hilbert (1862-1943), namely that all "reasonably defined" problems would be algorithmically decidable in principle. This part of the theory of algorithms is usually referred to as "Theory of Computability" or "Theory of Recursive Functions".

Another direction arose in connection with a fast development of computers, starting at 1950's. It concentrates into developing fast computer algorithms and methods to analyze those. This can be called "Theory of Designing and Analysing Algorithms". It is the most oriented to (practical) computer science, although it is full of really challenging mathematical problems.

The third direction, which is the topic of this course, lies in-between the above areas. It studies how much certain resource (which is most often time) is needed to solve a given problem. Typically it concentrates into such problems which are in principal algorithmically decidable (even by a trivial argument), but any known solution of those requires so much time that in practise only very small instances can be solved. Also **classifications of complexities of problems** are crucial here.

Fundamental achievements of the theory are the theorem of Cook (1971) proving the existence of so-called NP-complete problems, and so-called $P \stackrel{?}{=} NP$ -problem connected to that. The latter is often referred to as one of the most fundamental open problems in modern mathematics.

In these lectures we concentrate on this third direction of the theory of algorithms.

Recommended literature

Aho-Hopcroft-Ullman , The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

Harel , Algorithmics — The Spirit of Computing, Addison-Wesley, 1987.

Garey-Johnsson , Computers and Intractability: A Guide to the Theory of NP-completeness, Freeman, 1979.

Wilf , Algorithms and Complexity, Prentice-Hall, 1986.

Cormen-Leiserson-Rivest , Introduction to Algorithms, MIT Press, 1989.

Papadimitriou , Computational Complexity, Addison-Wesley, 1994.

Hopcroft-Ullman , Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979.

Finally, a brief plan for these lectures is as follows: After preliminary considerations of algorithms we define our formal model — a Turing Machine — and consider its basic properties. After that we concentrate into the main topic of this course, namely NP-completeness. And finally we compare different complexity classes.

2 Preliminaries

Intuitively by an algorithm we mean a set of instructions, which can be used to solve a certain problem. In order to call such a set an **algorithm** we require:

(i) It consists of a **finite** number of simple operations (elementary operations).

(ii) The instructions are applied **deterministically**.

(iii) The instructions are **universal**, that is they apply to each input instance of the problem.

(iv) The instructions are **terminating**, that is for each legal input the result is obtained in a finite number of applications of operations.

An algorithm can be seen as a **solution** to an **algorithmic problem**. An algorithmic problem, in turn, consists of well-defined inputs, each of which is uniquely connected to its output. Hence, an algorithmic problem is a function $I \rightarrow O$, and an algorithm for this problem is a **method to compute** this function!

Algorithmic problems are usually divided into two caregories:

A) **Desicion problems**, when the output is always "yes" or "no".

B) **Problems to compute an algorithmic function**.

Of course, case A is a special case of B. However, it is normally introduced separately, since in many theoretical considerations it is easier to handle, and still normally problems in case B can be reduced to the "coorresponding" of case A. We concentrate to problems of case A. Moreover, in accordance with formal language theory, inputs are given as words. This means that they must be encoded as finite sequences of elements from a finite set.

As we said an algorithm is a method to compute a function. Hence, from (iv) a natural question arises: How many steps, i.e. applications of operations, are needed? Or more precisely: How many steps are needed for an input (or instance of the problem) of a certain size? Instead of asking the number of steps, that is

computation time, we can ask the need of some other resources, such as the number of memory cells needed. These are the essential questions of this course.

As we indicated a complexity is measured as a function of the size of an input. Our main interest will be on **asymptotic** complexity. Let us consider more precisely **time complexity** (which measures the number of required computation steps); similar considerations apply for **space complexity** (which measures the number of required memory cells). The complexity can be of the following types:

(1) **worst case** complexity, or

(2) **average case** complexity.

The former tells how much time is needed in the worst case in order to obtain outputs on inputs of size n. The latter, in turn, tells the average time on inputs of size n. Of course, the average case complexity is always at most as large as the worst case complexity.

Both of the above complexities can be defined for algorithms, as well as for algorithmic problems. In the case of algorithms the notions are clear. In the case of an algorithmic problem its complexity is the complexity of the "best" algorithm solving this problem. The best, of course, means here the same as the lowest complexity.

By the **analysis of an algorithm** we mean a task of determining its complexity. This is often a managable problem. On the other hand, as we shall see, the problem of determining the complexity of a given algorithmic problem is usually a hopeless task! Of course, any algorithm solving an algorithmic problem provides an **upper** bound for the complexity of this problem. What is difficult is to prove the **lower** bounds, that is that no algorithm solves the problem faster than a given one. To determine the complexity of an algorithmic problem requires to show that the above upper and lower bounds are (asymptotically) equal.

If not otherwise stated by the complexity of an algorithm or an algorithmic problem we mean the **worst case time complexity**.

One of our main goals is to try to classify problems in terms of their complexity. A coarse, but especially theoretically important, classification is obtained by dividing algorithmic problems to

- **tractable**, that is to those which can be solved in polynomial time,

– **intractable**, that is to those for which no polynomial time algorithm is known.

Note that the "definition" of an intractable problem is a bit strange. Indeed, it may vary in time! However, it will be motivated later, and surely it means that such problems are hopeless to be solved in practise. Of course, this might hold for some tractable problems as well. Indeed, if the complexity of the best algorithm for a given problem is n^{1000}, the problem is in practise hopeless.

For algorithms the terms in the above classification are different. We call an algorithm **feasible** if it works in polynomial time and **infeasible** otherwise (i.e. if it does not work in polynomial time).

As we said the complexity is mainly measured as asymptotic complexity. Hence we recall the notions $\mathcal{O}$, o, Ω and Θ: For functions $f, g : \mathbb{N} \to \mathbb{N}$ we write

$$
\begin{aligned}
f = \mathcal{O}(g) \quad & \text{iff} \quad \exists \alpha, n_0 : f(n) \leq \alpha g(n),\ \forall n \geq n_0, \\
f = \Omega(g) \quad & \text{iff} \quad \exists \beta, n_0 : f(n) \geq \beta g(n),\ \forall n \geq n_0, \\
f = \Theta(g) \quad & \text{iff} \quad \exists \alpha, \beta, n_0 : \alpha g(n) \leq f(n) \leq \beta g(n),\ \forall n \geq n_0, \\
f = \mathrm{o}(g) \quad & \text{iff} \quad \lim_{n \to \infty} \frac{f(n)}{g(n)} = 1.
\end{aligned}
$$

Some reasons why the complexity is measured only asymptotically and using the above precisions are as follows:

– they are easier to determine, especially upper bounds which are mostly considered;

– they are important from theoretical points of view;

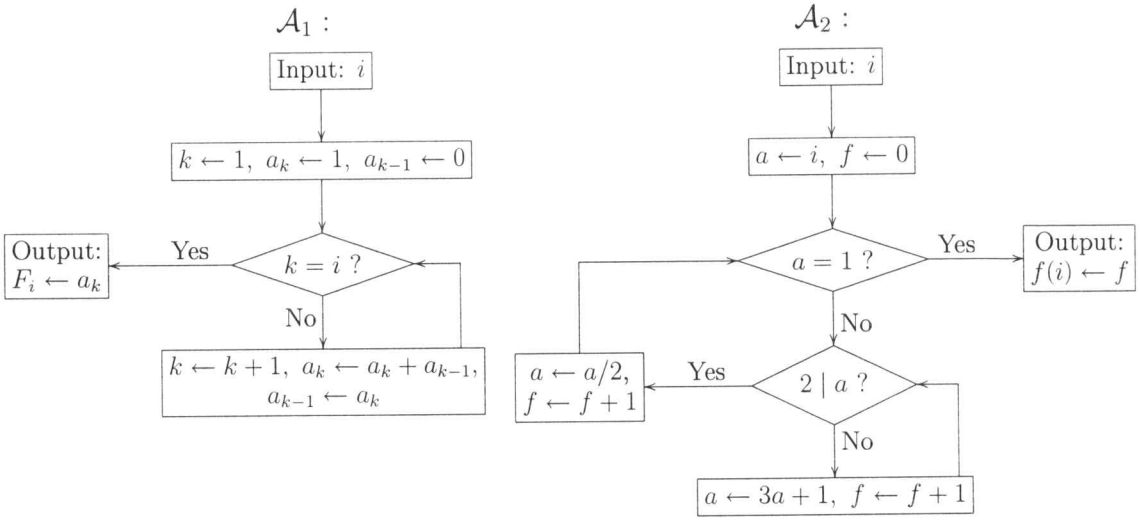
– the order of complexity is much independent of the programming language and/or computer used! For example, each algorithm can be speeded-up by a factor two by merging always two consecutive steps.

Example 2.1. "Is n prime?" This is a well-defined algorithmic problem. This also has a simple solution:

$$\text{For } i = 2, \ldots, \lfloor \sqrt{n} \rfloor \text{ test whether } i \mid n.$$

Each test can be done by an ordinary division algorithm and there are $\lfloor\sqrt{n}\rfloor - 1$ tests. Hence the problem can be solved in polynomial time. "Yes" this is true if the size of the input is n. This, however, is a "wrong" size of the input: The input must be considered as k-aric number (and not unary 1^n) when its length is $\lfloor \log_k n \rfloor + 1$. Hence the number of tests needed is of order $k^{\frac{1}{2}\|n\|}$, where $\|n\|$ denotes the size of the input, that is exponential in $\|n\|$. The considered problem is among the most studied mathematical problems, and still only very recently it was shown to have a polynomial time solution! Agrawal, Kayal, Saxena 2002

Example 2.2. Consider the following two algorithms defined as:



Clearly $\mathcal{A}_1$ computes the ith Fibonacci number: $f_{\mathcal{A}_1}(i) = F_i$, where $F_1 = F_2 = 1$ and $F_{n+2} = F_{n+1} + F_n$. The algorithm $\mathcal{A}_2$, in turn, computes the function

$$f_{\mathcal{A}_2}(i) = \text{the smallest } k \text{ such that } a_k = 1 \ (= \mu k[a_k = 1]),$$

where

$$a_0 = i$$

$$a_{n+1} = \begin{cases} a_n/2 & \text{if } 2 \mid a_n, \\ 3a_n + 1 & \text{otherwise.} \end{cases}$$

Actually we cannot be sure that $\mathcal{A}_2$ terminates for all inputs — and this indeed is not known! (This is so-called **Collatz's Problem**). Hence, $f_{\mathcal{A}_2}$ might be only

a **partial** function. However, it is **algorithmically** defined, and $\mathcal{A}_2$ computes it correctly if it is defined (i.e. $\mathcal{A}_2$ computes correctly "yes-instances" of inputs).

It follows that $\mathcal{A}_2$ is not an algorithm in our earlier sense — does not satisfy (iv). However, it is a **semialgorithm** in the sense that instead of (iv)we require that if the output is defined $\mathcal{A}_2$ computes it correctly.

Consider now the complexities of $\mathcal{A}_1$ and $\mathcal{A}_2$. The number of memory cells is at most 5, and hence the spacecomplexity is $\mathcal{O}(1)$. The time complexity for $\mathcal{A}_2$ is not known, as we already indicated. For $\mathcal{A}_1$ it is easy to determine. However, it depends on what are the elementary operations: if only arithmetic operations are counted then $2i$ steps is enough to compute F_i, that is the complexity is $\mathcal{O}(i)$. Here we assumed that the size of the input is the number i and not its length $\|i\|$. This is natural!

The order of complexity of $\mathcal{A}_1$ would change a bit if all operations needed would be considered elementary operations and additions would be done digit by digit. The complexity would be $\mathcal{O}(i^2)$, as is straightforward to see.

Finally, we note that the following recursive rules

$$F_1 \leftarrow 1,\ F_2 \leftarrow 2,$$
$$F_{n+1} = F_n + F_{n-1}$$

leads to a terrible algorithm. Its complexity is exponential on i. Why?

Example 2.3. Let $\rho \in [0,1]$ be a real number.
Problem: Is ρ algorithmically computable, that is, does there exist an algorithm $\mathcal{A}$ such that:

$$i \longrightarrow \boxed{\mathcal{A}} \longrightarrow \text{the } i\text{th decimal of } \rho$$

Or more precisely: When does such an algorithm exist? The answer is: "almost never". Indeed, the cardinality of the unit interval is nondenumerable, while the number of different algorithms, by (i), is only denumerable!

Of course, the above simple argument does not give any concrete example of a noncomputable real number — only shows the existence of such. At the same time it emphasizes the differences between classical and constructive mathematics.

Let us return still to our intuitive definition of an algorithm. Altough it is mathematically vague, it is enough when we want to show that

- A given problem has an algorithmic solution; or
- A given problem can be solved in a given time (by using certain operations).

Indeed, to show this it is enough to construct an algorithm (of certain type) solving the problem.

On the other hand, if we want to show (as we do want in this course) that

- A given problem has no algorithmic solution; or
- A given problem has no algorithmic solution working in time (or space) f and using operations of certain types,

then the above intuitively defined notion of an algorithm is not precise enough. Instead we have to **formalize** the notion of an algorithm. Indeed to prove that **no algorithm solves a problem** we have to know precisely what are all the algorithms!

Consider now an arbitrary formalization of the notion of an algorithm. The only things required are that the algorithms are precisely defined, and can be put into a list

$$\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \dots \tag{2.1}$$

effectively. Here the word "effectively" means that, for a given i, the ith algorithm can be found. Further here we think about algorithms as solutions to decision problems: $\Sigma^* \to \{0,1\}$.

Claim 2.4. There exists an algorithmic problem such that

- No $\mathcal{A}_i$ solves it; and
- There exists an intuitive algorithm to solve it

Proof. Let us define the required algorithmic problem as follows. With a given $x \in \Sigma^*$ we associate the output $D(x)$ in the following way: We interprite x as an n-aric number, with $n = \|\Sigma\|$, and set

$$D(x) = f_{\mathcal{A}_x}(x) + 1 \pmod 2.$$

Clearly (ii) is satisfied: For a given x, we search for $\mathcal{A}_x$, compute $f_{\mathcal{A}_x}(x)$ and interchange the output.

On the other hand assume that $\mathcal{A}_{i_0}$ computes D. Then

$$f_{\mathcal{A}_{i_0}}(i_0) = D(i_0) = f_{\mathcal{A}_{i_0}}(i_0) + 1 \pmod 2 \tag{2.2}$$

This contradiction proves the claim. □

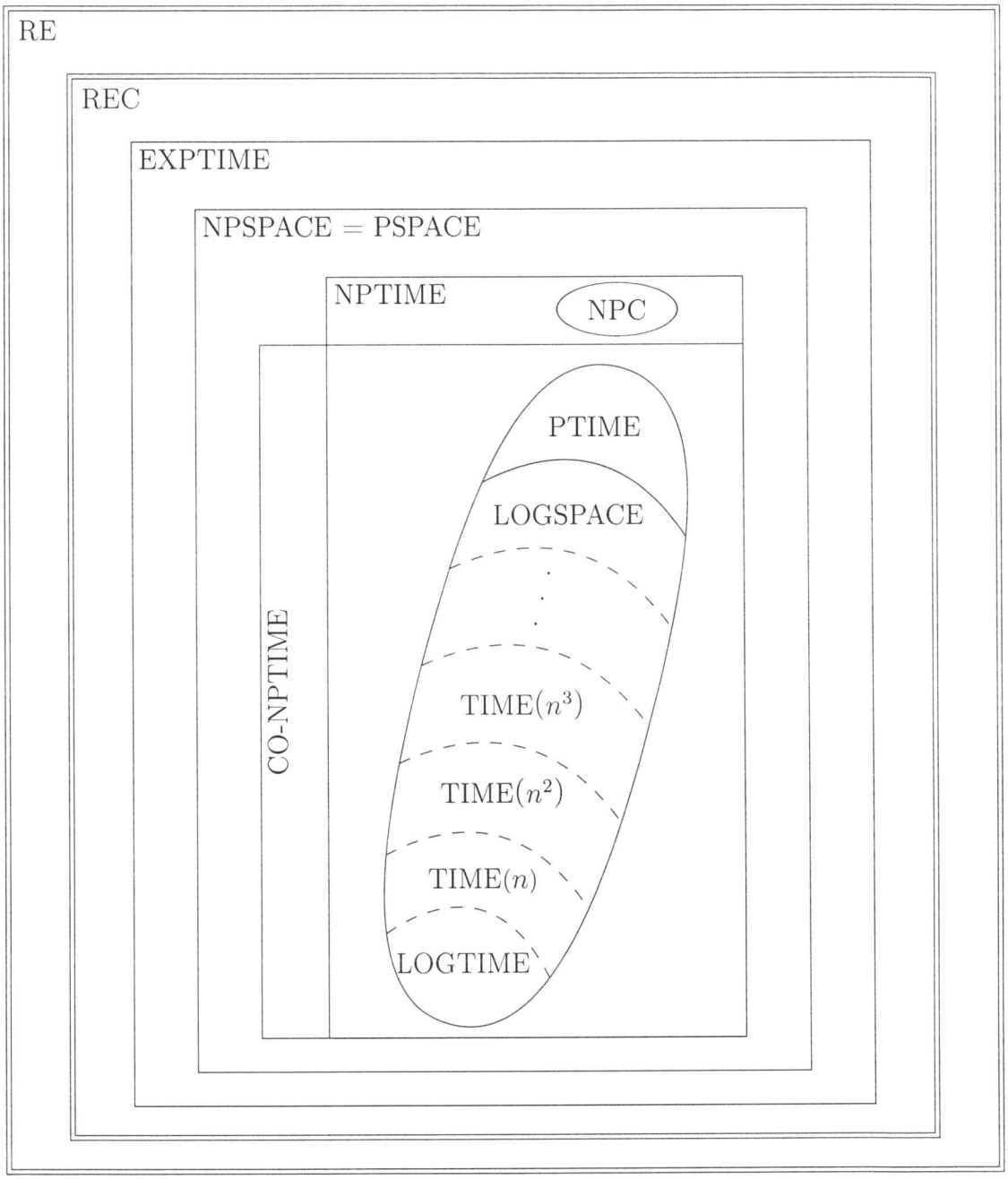
The above is referred to as the **diagonalization problem**. It shows that there does not exist any way of formalizing the notion of an algorithm requiring (2.1). This is at least then when (2.2) is considered as a contradiction. However, (2.2) **is not a contradiction**, if we give up the requirement that $\mathcal{A}$ has to terminate always, that is the requirement (iv), cf. example 2.2.

A conclusion from above is that when formalizing the notion of an algorithm we **have to allow partial functions**, that is algorithms need not terminate for all legal inputs. Moreover (iv) has to be replaced by

(iv') The instructions need not be terminating, but whenever the answer is obtained it is the correct one.

As we saw the requirement (iv) was too restrictive. In a certain sense the same holds for (ii). Clearly it is very natural, since nondeterministic algorithms seems to be of no practical use. However, as we shall see during this course, nondeterministic algorithms are very important from certain **theoretical points of view**. We shall formalize the notions of an algorithm and its nondeterministic variant in the next chapter.

We conclude this chapter by considering the Figure "Basic complexity classes" on the next page, which presents the relations between different complexity classes. The details of that figure are the contents of this course. In this figure



Basic complexity classes

– – – : Design and Analysis of Algorithms

——— : Complexity Theory

═══ : Computability Theory

TIME	refers to time complexity,
SPACE	refers to space complexity,
N	refers to nondeterminism,
P	refers to polynomial,
LOG	refers to logarithmic,
EXP	refers to exponential,
NPC	refers to NP-complete,
CO	refers to complement,
RE	refers to recursively enumerable (i.e. all algorithms),
REC	refers to recursive (i.e. all always terminating algorithms) and
n	refers to complexity n.

We shall return to the Figure several times. However, we make already now the following preliminary remarks:

2.1. For a decision problem P its **complementary** problem co-P is defined as: $P(x) = 0 \Leftrightarrow$ co-$P(x) = 1$. For example the complementary problem of example 2.1 asks "Is n a composite number?"

2.2. Clearly, for each deterministic time class, as well as for REC:

$$P \in \text{PTIME} \Leftrightarrow \text{co-}P \in \text{PTIME}.$$

This is simply because from a solution to P we get a solution for co-P by changing the outputs. For nondeterministic classes this does not work.

2.3. That a deterministic class is included in the corresponding nondeterministic class is trivial, by definitions.

2.4. Similarly each SPACE-class is included in the corresponding TIME-class (since each memory cell must be visited).

2.5. A big problem in Complexity Theory is: It is not known whether any of the inclusions

$$\text{LOGSPACE} \subseteq \text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE}$$

Example 2.5. Let P be the problem "Is n a composite number?". Then co-P is the problem "Is n a prime?". There is no known polynomial time algorithm for co-P, while such exists for P. However, both can be solved in polynomial time nondeterministically. For co-P it is not easy, but we will show it. For P it is easy:

(1) Guess numbers p and q; and

(2) Check that $n = pq$.

Clearly, (1) requires only 1 steps, and (2) can be computed trivially in time $\mathcal{O}(\|n\|^2)$.

As we shall see there is a lot of very natural problems which can be solved in polynomial time nondeterministically, but for which no deterministic polynomial time algorithm is known. Moreover, if any of those can be solved in polynomial time, then a really huge number of important problems would become tractable. Such problems are called **NP-complete**, the class NPC in the Figure. An example of such a problem is the following **Travelling Salesperson Problem**, TSP for short, which asks to find a shortest tour in a weighted graph. Here a tour means a path in graph starting and ending in the same point and visiting in all other points exactly ones, and its length is the sum of the weights of edges.

is proper. In particular, the middle inclusion is the famous P $\stackrel{?}{=}$ NP -problem. As we state in ??, it is known that LOGSPACE $\subsetneq$ PSPACE.

2.6. In some extend the Figure is "relative" in the sense that the classes depend on the model of an algorithm. This, however, is not true in the "higher" classes, as we shall get convinced.

For the model we shall be using, i.e. for complexity classes defined using Turing Machines, we make the following remarks, which will become clear later:

2.7. The inclusion LOGSPACE $\subseteq$ PTIME is quite straightforward: If an input of size n can use at most $\log n$ memory cells and each of those may be filled by at most k different letters, then there are at most $k^{\log n}$ different configurations, i.e. different fillings of the memory cells, and because of the determinism the output must be obtained in at most so many steps. But $k^{\log n}$ is polynomial in n.

Similarly PSPACE $\subseteq$ XTIME for some suitably chosen (what?) X.

2.8. The identity NPSPACE = PSPACE, in turn, is one of the important basic results of complexity theory, as we shall see.

2.9. On the contrary to the case 2.5 one can show that under a pretty mild conditions it holds:

$$\mathrm{SPACE}(f) \subsetneq \mathrm{SPACE}(g)$$

provided that

$$\liminf_{n\to\infty} f(n)/g(n) = 0.$$

Finally, let us consider tentatively **nondeterministic** algorithms. In such an algorithm a computation can continue in different directions in each step. The time complexity is measured as the length of the shortest computation which gives the correct output (i.e. there might be even shorter ones which gives a wrong output). The possibility for different choices in each step can be interpreted such that at the beginning it is guessed which computation is followed, and then we have to verify that the guess was correct in the required number of steps.

Let us consider a concrete example.

3 Turing machines

We formalize the notion of an algorithm as **Turing Machine**, introduced by Alan Turing (1912–1954) 1936. It is a surprisingly simple device, and still, as we shall see, it "can do anything what can be done algorithmically". Before even defining it we want to emphasize two features of Turing Machine. First it is particularly **important for theoretical considerations**, such as for the problem P $\stackrel{?}{=}$ NP. On the other hand it is quite **glumsy as a practical model** of an algorithm. Finally, there are numerous variations of the definition of Turing Machine, we follow that of Papadimitriou.

A **Turing Machine**, TM for short, is a quadruple

$$M = (Q,\Sigma,\delta,s),$$

where

Q is a finite set of **states**,

Σ is a finite set called the **alphabet** of M, with $Q \cap \Sigma = \emptyset$,

$s \in Q$ is the **initial state** and

δ is the **transition** relation of M;

$$\delta \subseteq Q \times \Sigma \times Q \cup \{h, yes, no\} \times \Sigma \times \{R, L, S\}. \tag{3.1}$$

Moreover, Σ contains two special symbols: the **blank** $*$ and the **first symbol** $\triangleright$, and the latter of these satisfies:

$$\text{if } (q, \triangleright, q', a, d) \in \delta, \text{ then } a = \triangleright \text{ and } d = R \tag{3.2}$$

(which, as we shall see, intuitively means that $\triangleright$ is never overwritten, and when read the head moves to the right). Here the special states h, yes and no are called **halting**, **accepting** and **rejecting**. The letters R, L and Stells the direction of the move of the head: **right**, **left** or **stay**. We also assume that

$$\{h, yes, no, R, L, S\} \cap (Q \cup \Sigma) = \emptyset.$$

The transitions of M are usually denoted as

$$(q, a) \to (q', a', d) \text{ with } q \in Q;\ q' \in Q \cup \{h, yes, no\};\ a, a' \in \Sigma;\ d \in \{R, L, S\}. \tag{3.3}$$

What we defined above is a **nondeterministic** Turing Machine, NTM for short, since δ is an arbitrary relation. If it is a function

$$\delta : Q \times \Sigma \to Q \cup \{h, yes, no\} \times \Sigma \times \{R, L, S\},$$

then we have a **deterministic** Turing Machine, DTM for short.

Next we define a **computation** of TM M on an input $w \in (\Sigma \backslash \{*\})^*$. **Intuitively** this is defined as follows: M starts to consider the word $\triangleright w$ such that it is in the initial state s, and the **head** is scanning $\triangleright$. Then according to a transition of M, say (3.3), M

- goes to a new state q';
- the scanned symbol is overwritten to a';
- the head is moved according to d.

This is repeated as long as one of the special states h, yes or no is reached — if reached at all. Consequently a computation may be infinite, i.e. **not terminating**, or finite, i.e. **terminating** due to the fact that M goes to h, yes or no, or next transition is not defined. If it terminates in states yes or no, we define the **output** $M(w)$ to be "yes" or "no", respectively. If it terminates in state h we define the **output** $M(w)$ as follows. If $\triangleright w$ is changed to $\triangleright v$ during the computation then $M(w) = v'$, where v' is the longest prefix of v ending with a letter $\neq *$ or the **empty** word 1, if there is no such prefix.

Formally, the above is defined as follows. A **configuration** of M is a triple $(q; w, u)$ where $q \in Q$ and $w \in \Sigma^+$, $u \in \Sigma^*$. We say that a configuration $(q; w, u)$ **yields** $(q'; w', u')$ **in one step**, in symbols $(q; w, u) \to_M (q'; w', u')$, if M has a transition of the form (3.3), and

(i) $w = w_1 a$, $u = b u_1$, $w' = w_1 a' b$, $u' = u_1$ or
$w = w_1 a$, $u = 1$, $w' = w_1 a *$, $u' = 1$, if $d = R$;

(ii) $w = w_1 b a$, $w' = w_1 b$, $u' = a' u$, if $d = L$;

(iii) $w = w_1 a$, $w' = w_1 a'$, $u' = u$, if $d = S$.

Now the relation "a configuration $(q; w, u)$ **yields** a configuration $(q'; w', u')$" in any number of steps, in symbols $(q; w, u) \to_M^* (q'; w', u')$, is defined as the transitive and reflexive closure of $\to_M$. A **computation** on input w is a sequence $c_0, c_1, \ldots$ of configurations such that

$$c_0 = (s, \rhd, w)$$

and

$$c_i \to_M c_{i+1}$$

for $i = 0, 1, \ldots$. As we noted, it can be either finite or infinite.

Next we consider how TM's can be used as algorithms. Assume first that M **is deterministic**.

1. We say that a DTM M **decides** the language $L \subseteq (\Sigma \setminus \{*\})^*$ (or an **algorithmic problem** L) if

   - $M(w) = \text{yes}$ for $w \in L$, and
   - $M(w) = \text{no}$ for $w \in (\Sigma \setminus \{*\})^* \setminus L$.

   Moreover, we define that L is **recursive** iff there exists a TM M deciding L.

   As an extension of above we define:

2. A DTM M **accepts** the language $L \subseteq (\Sigma \setminus \{*\})^*$, if

   $$\forall w \in (\Sigma \setminus \{*\})^* : M(w) = \text{yes} \Leftrightarrow w \in L.$$

   The languages which are accepted by DTM's are called **recursively enumerable**. In this case we denote $L = L(M)$.

   Finally, we use a DTM as a model to compute an algorithmic function.

3. A DTM M **computes** a **partial** function $f : (\Sigma \setminus \{*\})^* \to \Sigma^*$ if $M(w) = f(w)$ for all $w \in (\Sigma \setminus \{*\})^*$. Further f is called **Turing computable** or **recursive**, if there exists a DTM computing it.

For a nondeterministic TM it is important to define only the case 2 above:

2'. A NTM M **accepts** the language $L \subseteq (\Sigma \setminus \{*\})^*$ if

$$\forall w \in (\Sigma \setminus \{*\})^* : \text{yes} \in M(w) \Leftrightarrow w \in L, \tag{3.4}$$

that is **some** computation on input $w \in L$ gives the output "yes", and no computation on other inputs yields "yes". Again we write $L = L(M)$.

Remarks.

3.1. In accordance with the diagonalization problem TM's allow nonterminating computations.

3.2. When defining Turing Machines as models of computing function we restrict to functions from "words to words". This, however, is no real problem since numbers can be seen as words via their n-aric representations. Similarly, when TM's are used to solve algorithmic problems, the **instances must be encoded** as words over the input alphabet of the machine.

3.3. Note that definitions 2 and 2' are consistent: A DTM accepts a language, iff it accepts it as a NTM.

3.4. In the definitions 2 and 2' we could require also that

$$\text{if } w \notin L, \text{ then } M(w) = \uparrow \text{ (for each computation on } w). \tag{3.5}$$

This is essentially due to a principle that a **terminating computation can be modified** to nonterminating. To see this we assume that a NTM M accepts L and construct a new M' which also accepts L and satisfies both (3.4) and (3.5). M' is like M except that

- for all pairs (q, a) for which M does not contain a transition M' contains $(q, a) \rightarrow (q, a, S)$;
- transitions of M leading to h or no are lead to a new state q_n for which M' contains $(q_n, a) \rightarrow (q_n, a, S)$ for $a \in \Sigma$.

Clearly, the above construction makes each terminating but not accepting computation infinite, and preserves the accepting computations.

3.5. The above ideas immediately yields:

Theorem 3.1. Each recursive language is recursively enumerable.

3.6. As we already said there exist several slightly different, but equivalent definitions of a Turing Machine. In our definition there are — for clarity — several special symbols like $\triangleright$, h, *yes* and *no*. Moreover, our model is **one-way**: the head never goes to the left from $\triangleright$, which **is preserved** during the whole computation. On the other hand the head must be allowed to extend the length of the word: this is done on the second line of (i) on page 15; then the new symbol added to the word is the blank $*$. Note also that once the word is extended it **is never shortened**!

From the point of view of our later theoretical considerations we need two further variants of TM's: a k **word (or tape) Turing Machine**, kTM for short, and an **input preserving Turing Machine**.

Let $k \geq 1$ be an integer. A kTM is like an ordinary TM except that transitions are instead of (3.3) of the form:

$$(q, a_1, \ldots, a_k) \to (q', a'_1, d_1, \ldots, a'_k, d_k), \tag{3.6}$$

where $q \in Q$; $q' \in Q \cup \{h, yes, no\}$; $a_i, a'_i \in \Sigma$ for $i = 1, \ldots, k$; $d_i \in \{R, L, S\}$ for $i = 1, \ldots, k$. Clearly, (3.6) splits into k ordinary transitions:

$$(q, a_i) \to (q', a'_i, d_i) \qquad \text{for } i = 1, \ldots, k,$$

each of which working on their own word. The input is given in the first tape, other tapes containing only the first symbol $\triangleright$. Hence the **initial** configuration is

$$(s; \triangleright, w; \triangleright, 1; \ldots; \triangleright, 1).$$

The transitions respects the rule that $\triangleright$ is never passed to the left. Also other aspects of ordinary TM's are extended to kTM's in a natural way. The **outputs** are read from the 1st tape.

The **input preserving kTM**, with $k \geq 3$, is a kTM satisfying:

- the input word is never changed during the computations, i.e. if with respect to the first tape $(q, a) \to (q', a', d)$ then $a' = a$, and if $(q, *) \to (q', *, d)$ then $d = L$.

– the head on the k'th tape, where the **output** is read, cannot move to the left, i.e. with respect to that tape: if $(q, a) \to (q', a', d)$, then $d \neq L$.

The reasons to define these two models are as follows:

1. A kTM is much more practical to show that something can be done by a Turing Machine, and still it can be simulated by an ordinary Turing Machine, cf. Theorem 3.9.

2. Input preserving TM is needed to define **sublinear** space complexity classes, and **here the output is read from the last tape**, and if there is no output this tape is normally left undefined.

Examples.

3.2. (Right Transfer Machine). Consider a DTM consisting of transitions

$$\begin{aligned} (s, \triangleright) &\to (q, \triangleright, R) & (q_a, *) &\to (h, a, S) \\ (q, a) &\to (q_a, *, R) & (q_a, b) &\to (q_b, a, R), \end{aligned}$$

where $a, b \in \Sigma \setminus \{*\}$. Machine M has for example the following computation:

$$\begin{aligned} (s; \triangleright, aba) &\to (q; \triangleright a, ba) \to (q_a; \triangleright * b, a) \\ &\to (q_b; \triangleright * aa, 1) \to (q_a; \triangleright * ab *, 1) \to (h; \triangleright * aba, 1). \end{aligned}$$

Or more generally, $M(w) = *w$ for each $w \in \big(\Sigma \setminus \{*\}\big)^*$, when we further define $(q, *) \to (h, *, S)$.

Note that we may denote the configurations as $(q, w'\underline{a}v)$ instead of $(q; w'a, v)$. A still more succinct way of doing this is:

$$\begin{array}{c} w'av \\ \uparrow \\ q \end{array}$$

Also in above the transitions are not defined for all pairs (q, a), since some of those are never encountered during the computations.

3.3. (Binary Successor Function). The following DTM computes the function $\mathrm{Bin}(n) \mapsto \mathrm{Bin}(n+1)$, where $\mathrm{Bin}(n)$ is the binary representation of n:

$$\begin{aligned}
&(s,a) \to (s,a,R),\ a \neq * && (q,\triangleright) \to (t,\triangleright,R)\\
&(s,*) \to (q,*,L) && (t,0) \to (t',1,R)\\
&(q,1) \to (q,0,L) && (t',0) \to (t',0,R)\\
&(q,0) \to (h,1,S) && (t',*) \to (h,0,S)
\end{aligned}$$

This works correctly if $n \neq 0$, i.e. the input is different from empty word. To handle this we have to add: $(t,*) \to (h,1,S)$.

3.4. (Palindrome decider). We describe how a TM can decide, whether a given word is a palindrome. The machine

- remembers the first symbol, changes it to $*$ and travels to the end to compare it to the last one,
- it they coincide the machine change the last symbol to $*$ and searches for the first nonblank and repeat above, until the word is of length 0 or 1, when it goes to the state *yes*,
- and if they do not coincide the machine goes to the state *no*.

That these ideas can be realized by a finite number of transitions is easy to conclude.

As another solution to the above problem we describe an input preserving 4TM. The input is in the first tape, the second and third tape contains a number (telling which letters on the right and left are compared) and the fourth tape contains counter (setting the consecutive values for the second and third tapes). The machine operates as follows:

- the contents of 2nd, 3rd and 4th tape are set to be $i = 1$;
- using the second tape as counter the machine search for the i'th letter from the beginning of an input if i (in binary) is the contents of the second tape;

- using the 3rd tape the machine looks for the ith letter from the right in the input;
- the machine remembers these letters and compares those: if they coincide, the number in the 4th tape is increase by 1 (cf. example 3.3) and this number is copied to the second and the third tapes, and a new round is initiated; otherwise the machine goes to state *no*.
- When the input word is shorter than the number in the fourth tape (which can be detected when counting the positions) the machine enters to state *yes*.

It is clear that the machine works correctly. It is also obvious that only a finite number of transitions are needed to realize the above by a 4TM, however, the construction of those would be boring. While this solution is more complex it has a feature that only $\log(|w|)$ space is neede on tapes 2, 3 and 4. Note also that **we left the output tape of input preserving** machine undefined.

3.5. (Copying Machine). The purpose of the machine is to carry out computations

$$(s; \rhd, w) \rightarrow^* (h; \rhd w\#, w)$$

for $w \in (\Sigma \setminus \{*\})^*$. This indeed is not difficult on the transition level. The machine has to be able to **find** a particular symbol, **remember** in the memory one symbol (cf. example 3.2), and **mark** a symbol just read by a transition $(q, a) \rightarrow (q', \overline{a}, R)$. We leave the details as an excercise.

3.6. (Comparison Machine). This machine has to decide the language $L = \{w\#w \mid w \in \Delta^*\}$, where $\# \notin \Delta$. But this is essentially the problem of example 3.4. In fact, by 2TM it can be reduced exactly to that problem, by copying the input up to # to the 2nd tape, and then copying the second w from right to left after #. Finally, the machine of example 3.4 is used on the 2nd tape only.

Next we define **complexity classes** which are crucial for this course. The above definitions are valid for different variants of TM's. Let M be a TM and $f : \mathbb{N} \rightarrow \mathbb{N}$ a function. We say that M **operates in time** f or **in space** f if:

For any $n \in \mathbb{N}$, any computation on any input of length n terminates in at most $f(n)$ steps, or uses at most $f(n)$ cells on any tape.

Similarly for a **input preserving** TM M, and a function $f : \mathbb{N} \to \mathbb{N}$ we say that f **operates in space** f if:

> For any $n \in \mathbb{N}$ any computation on any input of length n uses in all tapes $2, \ldots$ at most $f(n)$ memory cells.

For our considerations the most important cases are those, where machines are used to decide or accept languages. We say that a language $L \subseteq (\Sigma \setminus \{*\})^*$ is **decided or accepted in time** f **or space** f, if there exists a TM deciding or accepting L and operating in time (respectively in space) f. Note that here the machine can be deterministic or not and multi word or not.

When **defining the complexity classes we restrict our considerations to multi word Turing Machines** which is our basic model of an algorithm.

Let $f : \mathbb{N} \to \mathbb{N}$ be a function. We define the following four families of languages:

$$\begin{aligned}
\mathrm{TIME}(f) &= \{L \subseteq \Sigma^* \mid \exists \text{ a } k \text{ word DTM } M \text{ operating in time } f \text{ and deciding } L\};\\
\mathrm{NTIME}(f) &= \{L \subseteq \Sigma^* \mid \exists \text{ a } k \text{ word NTM } M \text{ operating in time } f \text{ and accepting } L\};\\
\mathrm{SPACE}(f) &= \{L \subseteq \Sigma^* \mid \exists \text{ a } k \text{ word input preserving DTM } M \text{ operating in space } f \text{ and deciding } L\};\\
\mathrm{NSPACE}(f) &= \{L \subseteq \Sigma^* \mid \exists \text{ a } k \text{ word input preserving NTM } M \text{ operating in space } f \text{ and accepting } L\}.
\end{aligned}$$

Further we set the important classes

$$\mathrm{P} = \mathrm{PTIME} = \bigcup_{i \geq 1} \mathrm{TIME}(P_i)$$

and

$$\mathrm{NP} = \mathrm{NPTIME} = \bigcup_{i \geq 1} \mathrm{NTIME}(P_i),$$

where P_i is the polynomial $P_i(n) = n^i$. Similarly we define the classes PSPACE and NPSPACE.

Remarks.

3.7. The space complexities are defined using input preserving TM's in order to **obtain sublinear** sapce complexity classes, such as SPACE(log), cf. example 3.4. For time complexities this is not important, since it is natural to require that the whole input is read and this already takes a linear time.

3.8. The definitions of the complexities are natural for deterministic machines: Both accepting and rejecting computations should be taken into account. For nondeterministic machines one could count only accepting computations, since nonaccepting ones have no role in the process of accepting the language. This, however, as we next outline would not make any essential difference.

Let us say that a function $f : \mathbb{N} \to \mathbb{N}$ is **fully time constructible**, if there exists a deterministic kTM which operates on each input of length n in **exactly** $f(n)$ steps. Similarly f is **fully space constructible**, if there exists a deterministic kTM which for each input of length n uses exactly $f(n)$ memory cells on some tape.

We have the following two facts:

Fact 3.7. Let f be a fully time constructible function and M a nondeterministic kTM, which accepts each input in $L(M)$ in at most $f(n)$ steps. Then there exists a multi word NTM accepting L and operating in time $2f(n) + 2n + 2$.

Proof. Let M' be a k'TM operating exactly on time $f(n)$. We use it as a clock to stop the computations of M, if they run too long. More precisely, we construct a $(k + k')$TM M_1 as follows: First using the $(k + 1)$st tape we copy the input to that in $2n + 2$ steps. Then we simulate in odd step M on the first k tapes, and in every even step M' on the last k' tapes, and terminate a computation, if one of those terminates. Moreover we accept, if the computation of M is accepting.

Clearly $L(M_1) = L(M)$ and M_1 operates in time $2f(n) + 2n + 2$. □

Similarly, for space complexities we have, cf. excercises.

Fact 3.8. Let f be fully space constructible function and M an input preserving kTM, which accepts each input in $L(M)$ in space at most $f(n)$. Then there exists an input preserving multi word NTM accepting $L(M)$ and operating in space $f(n) + 1$.

Next we **compare the different variants** of TM's. As we already mentioned the complexity classes depend on the precise model of an algorithm used. For our

most crucial classes P and NP, however it does not matter whether we use TM's with one or several tapes, as we now show.

Theorem 3.9. Let $k \in \mathbb{N}$ and M a (not input preserving) kTM operating in time $T_M(n) = \Omega(n)$ and space $S_M(n)$. There exists a one word TM M' deciding $L(M)$ and operating in time $\mathcal{O}\big(T_M(n)^2\big)$ and space $\mathcal{O}(S_M(n))$. Moreover, if M is deterministic so is M'.

Proof. As stated in the theorem we prove it for TM's used as decision procedures; easy modifications prove it for TM's as computers of algorithmic functions. Our prove will **describe** how M' can be constructed without giving the required transitions.

Consider a configuration of M:

$$c = (q; w_1, v_1; w_2, v_2; \ldots; w_k, v_k)$$

and let $w_i = \rhd u_i a_i$, where $a_i \in \Sigma$ or $a_i = 1$ if $w_i = \rhd$, for $i = 1, \ldots, k$. We define

$$c' = (q; \rhd \# u_1 \overline{a}_1, v_1 \# u_2 \overline{a}_2 v_2 \# \cdots \# u_k \overline{a}_k v_k e)$$

with the convention $\# u_i \overline{a}_i = \overline{\#}$ if $u_i a_i = 1$, and consider it as a configuration of M'. Here $\overline{a}_i$'s are marked copies of a_i's, $\#$ is a marker and e is an endmarker — all new symbols not in Σ. In particular, for the initial configuration c_0 of M we have

$$c_0' = \big(s; \rhd \overline{\#}, w\overline{\#\#} \cdots \overline{\#} e\big).$$

Now a machine M' simulating M is constructed requiring the following three properties:

1) M' computes: $(s; \rhd, w) \to_{M'}^{*} c_0'$.

2) If M computes $c \to_M c_1$, then M' does $c' \to_{M'} c_1'$.

3) If M on input w terminates so does M', and in the same way.

To realize the above we note:

Transitions needed to do 1) are easy to define, since in that step w is moved one cell to the right, cf. example 3.2, marker $\#$ is added between $\rhd$ and w, and to the end of w a **constant** word is added. Moreover, this requires only $\mathcal{O}(|w|)$ steps.

To realize 2) M' has to find out the positions of the heads in each tape and the symbol in those cells. But these are marked so M' can find those by traversing the word, and remember those in its current state, where also q is remembered. Then M' knows what M does on each of its tapes, and M' can do the same again traversing the word: that is overwrite each $\overline{a}_i$, move the markers to the right place, and finally enter to the correct state q' of M. The above may require to enlarge a space in between some markers, but this can be done.

All in all only a finite number of transitions are needed to simulate **all** the steps in 2). Moreover, to simulate one step of M requires

$$\mathcal{O}\Big(\sum_{i=1}^{k} |w_i v_i|\Big) \leq \mathcal{O}\big(kS_M(|w|)\big) = \mathcal{O}\big(S_M(|w|)\big)$$

steps of M'. But since $S_M(|w|) = \mathcal{O}\big(T_M(|w|)\big)$ M' can in $\mathcal{O}\big(T_M(|w|)\big)$ steps simulate one step of 2).

Finally 3) can be realized by M' in one step (and in $\mathcal{O}\big(T_M(|w|)\big)$ if M computes a function).

Since there are $T_M(|w|)$ steps of 2) in any computation of M we have shown how to construct a 1TM M' such that $L(M') = L(M)$ and $T_{M'}(n) = \mathcal{O}\big(T_M(n)^2\big)$. Moreover, $S_{M'}(n) = \mathcal{O}(S_M(n))$ as required. □

Our next simple result confirms the intuition: What can be done in time f, can also be done in space f.

Theorem 3.10. Let $f : \mathbb{N} \to \mathbb{N}$ be such that $f(n) \geq n + 2$ for all n. Then we have $\mathrm{TIME}(f) \subseteq \mathrm{SPACE}(f)$ and $\mathrm{NTIME}(f) \subseteq \mathrm{NSPACE}(f)$.

Proof. Assume that a kNTM M operates in time $f(n) \geq n + 2$. Hence M cannot use more than $f(n)$ cells at any tape. Now, an input preserving machine accepting $L(M)$ and operating in space $f(n)$ is defined as follows. For an input w, it copies w to the second tape, and then uses it as the first tape of M and together with other tapes of M behaves like M on these tapes. Hence the first tape is never changed, so that $L(M)$ is accepted in space $f(n)$ by a input preserving $(k+1)$NTM without output tape.

For deterministic machines the proof is the same. □

Our third simulation result shows that anything what can be accepted by a nondeterministic TM can be accepted by a deterministic one, as well.

Theorem 3.11. Let M be a 1NTM operating in time $T_M(n) = \Omega(n)$. Then there is a deterministic 3TM M' and a constant α such that $L(M') = L(M)$ and M' operates in time $T_{M'}(n) = \mathcal{O}(\alpha^{T_M(n)})$ on inputs in $L(M)$.

Proof. The idea of the proof is clear: Given M we construct M' such that it simulates systematically all computations of M on a given input, and (~~if and~~) when M terminates M' does the same, and moreover in state *yes* if M does so.

We define the **nondeterminism degree** of M as largest number d such that for any pair (q, a) there are at most d transitions of the form $(q, a) \to x$. Then each computation on M of length i is specified by a word from $\{0, \ldots, d-1\}^*$ of length i: The i'th letter of this word tells which transition is used in the i'th step of the considered computation. Consequently for each pair (q, a) the transitions are numbered from 0 up to $d - 1$.

The 3 tapes of M' are working as follows:

1. tape contains the input;
2. tape simulates a finite computation of M;
3. tape generates in a lexicographic order all words over the alphabet $D = \{0, \ldots, d-1\}$.

Next we describe how M' behaves when starting from a configuration

$$(s; \rhd, w; \rhd, 1; \rhd, \beta), \tag{3.7}$$

with $\beta \in D^i$. First M' copies w to the second tape. Then M' carries out on the second tape the computation determined by β. Here the symbols of β tells which transitions of M is applied on the second tape. If this computation terminates in the state *yes*, so is defined M' to do as well. Otherwise, i.e. if it terminates in some other state or does not terminate at all in the considered i steps (which can be detected by the head on tape 3), or β is not a computation of M on input w (which can be detected by the fact that a symbol of β does not correspond to an applicable transition), then M' erases the whole contents of the second tape, and performs on

the 3rd tape a computation $\beta \mapsto S(\beta)$, where $S(\beta)$ is lexicographically next element after β. This transformation on tape 3 is easy to do, cf. 3.3. Finally, M' moves to the configuration

$$(s; \triangleright, w; \triangleright, 1; \triangleright, S(\beta)), \tag{3.8}$$

and a new round of the simulation is started.

That the above can be realized by a deterministic TM should be clear. So it remains to determine the complexity of M' on inputs in $L(M)$.

Let $w \in L(M)$ and $|w| = n$. Then M accepts w in a computation of length at most $T_M(n)$. Consequently, M' accepts w at the latest after performing the computation (3.7) $\rightarrow$ (3.8) for each value of $\beta \in D^i$ for $i = 1, \ldots, T_M(n)$. There are at most

$$\sum_{i=1}^{T_M(n)} d^i = \mathcal{O}(d^{T_M(n)})$$

such computations. Each of those requires at most

$$\mathcal{O}(T_M(n)) + \mathcal{O}(T_M(n)) + \mathcal{O}(T_M(n)) + \mathcal{O}(T_M(n)) = \mathcal{O}(T_M(n))$$

steps. Therefore M' accepts w in at most

$$\mathcal{O}\big(T_M(n) \cdot d^{T_M(n)}\big) = \mathcal{O}\big((d+1)^{T_M(n)}\big)$$

steps. □

The proof of Theorem 3.11 yields directly

Theorem 3.12. A language $L \subseteq (\Sigma \setminus \{*\})^*$ is accepted by a NTM iff it is accepted by a DTM.

Note however, that the word "accept" cannot be replaced by the word "decide". Combining Theorems 3.9, 3.11 ~~and 3.12~~ we obtain

Corollary 3.13. For each nondeterministic 1 word TM M accepting all words of length n in $L(M)$ in $T_M(n)$ steps there exists a deterministic 1 word TM M' which accepts all word of length n in $L(M)$ in time $\mathcal{O}(\alpha^{T_M(n)})$ for some constant α.

Finally we note that the proof of Theorem 3.11 is based on a very trivial idea: Exhaustive search of all possibilities. And **no essentially faster method is known for this simulation**.

Next we consider so-called **speed-up theorems**, that is possibilities of speeding-up a given algorithm. These results clarifies the complexity classes, in particular how they are related to $\mathcal{O}$-symbol.

We start with the following result.

Theorem 3.14 (Linear speed-up). Let $k \geq 2$ and M a deterministic kTM operating in time T_M. There exists another deterministic kTM M' deciding (or accepting) the language $L(M)$ and operating in time $\frac{3}{4}T_M(n) + \mathcal{O}(n)$ (on accepted words of length n).

Proof. The construction of M' is in three steps:

1. After having an input w of M the machine copies it to the 2nd tape, and at the same time compress the four consecutive letters to a word of length 4, i.e. computes $abcd \mapsto [abcd]$. At the end the positions in [ ] are filled by $*$'s. After that M' returns the word back to the first tape. Its length is now only "one fourth" of the original one.

2. The machine M' simulates 4 steps of M by only three steps. In doing so M' operates on symbols of the form $[abcd]$, where one of those might be marked as $\overline{b}$ telling that the head of M is scanning this b.

   Now the crucial observation is that in 4 steps M can move at most 4 steps to right or left — that is in terms of M' only to the next cell! In other words we have one of the following possibilities

$$\underset{\uparrow q}{\alpha\beta\gamma} \rightarrow_M^4 \begin{cases} \underset{\uparrow q'}{\underbrace{\alpha'\beta'\gamma}} & \text{or} \\ \alpha\underset{\uparrow q'}{\underbrace{\beta'\gamma'}} & , \end{cases} \tag{3.9}$$

   where the symbols represent words of length 4. Consider the first alternative, where α, β and q (but not γ) may change. These new values are determined by

transitions of M based on the old values of α, β and q and the position of q in $\alpha\beta$. In other words, the new values can be computed from the old ones (which are remembered by the states of the machine M'). Formally, this means that the transitions of M' can be defined in the following way to simulate (3.9) in 3 steps:

- M' moves to the left remembering q and β (which decides that the machine has to go to the left);
- M' knows now also α so that it can overwrite it to α' with a possible bar if needed, and moves to the right;
- M' can now overwrite β to β' and go to the state q' in the right place (i.e. either in α' or β').

3. Finally if M terminates so does M' in the corresponding way.

Now for any input accepted by M the machine M' needs

$$\mathcal{O}(|w|) + \tfrac{3}{4}T_M(|w|) + \mathcal{O}(1)$$

steps, altogether $\frac{3}{4}T_M(|w|) + \mathcal{O}(|w|)$ steps as required. □

The iteration of the procedure of Theorem 3.14 yields

Corollary 3.15. For each $c > 0$ and a deterministic kTM M, with $k \geq 2$, there exists a deterministic kTM M' accepting (or deciding) the same language as M, say L, such that if M operates on words of L in time $T_M(|w|)$, then M' does it in time $cT_M(|w|) + \mathcal{O}(|w|)$.

The above considerations implies the following result on complexity classes.

Theorem 3.16. Let $f : \mathbb{N} \to \mathbb{N}$ be a function satisfying $\lim_{n\to\infty} \frac{n}{f(n)} = 0$, $f(n) \geq n + 1$ and $q > 1$. Then

$$\mathrm{TIME}(f) = \mathrm{TIME}(qf).$$

Proof. Let $L \in \mathrm{TIME}(qf)$, i.e. L is decided by a deterministic kTM M operating in time $qf(n)$. Choosing $c = \frac{1}{2q}$ in the above corollary we find a deterministic kTM M' deciding L in time $g(n) = \frac{1}{2}f(n) + Kn$ for some K and $n \geq n_0$. Let $n_1 \geq n_0$

such that $g(n) \leq f(n) - 2n$ for $n \geq n_1$. Then M' decides all words w, with $|w| \geq n_1$ in time $f(n)$.

How about the short words? They can be handled by a principle: **Finite number of inputs can always be treated separately**. More precisely, we modify M' such that it first reads the input, remembers it in the state and, if it is shorter than n_1, gives directly the right output. If the input is longer after reading the $n_1 - 1$ first symbols the head returns to the beginning of the input ans starts to simulate M'. For inputs longer than $n_1 - 1$ this new machine operates in time $2n_1 + g(n) \leq f(n)$. This completes the proof. □

Above considerations deserve several

Remarks.

3.9. Theorem 3.16 explains why no $\mathcal{O}$-symbol is used in the definitions of complexities of TM's or complexity classes.

3.10. Clearly, Theorems 3.14 and 3.16 hold for nondeterministic Turing Machines, as well.

3.11. Analogouos results to Theorems 3.14 and 3.16 can be formulated for space complexities. Actually things are even simpler — a linear speed-up is obtained simply considering c consecutive letters as one symbol, and marking the positions of heads by bars.

3.12. The above speed-up results are obtained at the cost that the **description of the algorithm**, i.e. its **Kolmagorov complexity** increases.

3.13. The above linear speed-up is intuitively quite clear — contrary to the following much deeper result.

Proposition 3.17 (Speed-up Theorem). Let $g : \mathbb{N} \to \mathbb{N}$ be a total Turing computable function. There exists a recursive language L such that for any DTM M which decides L there exists another DTM M' such that M' decides L and $g(T_{M'}(n)) \leq T_M(n)$ for almost all n, i.e. for all except finitely many n's.

We do not prove Proposition 3.17 here — the proof is rather complicated. We only note that Proposition means that for some algorithmic problems there **does**

not exist the best solution, or even more strongly any solution can be speeded-up arbitrarily much for almost all values of n. For example, if $g(n) = 2^n$, then $T_{M'} \leq \log_2 (T_M(n))$, for $n \geq n_0$.

Next we consider an important property of TM's, namely the existence of a **universal machine**.

A Turing Machine formalizes a solution of an algorithmic problem, i.e. corresponds a **program of a computer**. A universal TM intends to mimic a computer in the sense that it can simulate any Turing Machine (read program) on its any input. In other words a universal TM U has to satisfy

$$U(M, w) = M(w), \tag{3.10}$$

for any TM M and input w. Is this possible? As stated there is a serious problem. The input of U contains M, i.e. potentially infinitely many states, for example. However U is required to be a fixed TM having a fix alphabet, for example. This guides us to require instead of (3.10) only

$$U(c(M), c(w)) = c(M(w)), \tag{3.11}$$

where c is an **encoding** representing M and w in a fixed finite alphabet (of U).

When doing this encoding let us assume that M is 1 tape deterministic Turing Machine. Such a machine consists of a finite number of quituples

$$t = (q, a, q', a', d) \in Q \times \Sigma \times Q \cup \{h, yes, no\} \times \Sigma \times \{L, R, S\}. \tag{3.12}$$

Here Q and Σ are finite, but can be arbitrarily large. Hence, assume that

$$Q \subseteq \{q_0, q_1, \dots\}$$

and

$$\Sigma \subseteq \{a_0, a_1, \dots\},$$

with the conventions $s = q_0$, $h = q_1$, $yes = q_2$, $no = q_3$ and $\triangleright = a_0$, $* = a_1$. Let $c : (Q \cup \Sigma \cup \{R, L, S\}) \to \{0, 1\}^*$ be a mapping defined as

$$\begin{aligned} c(q_i) &= 01^{2i+4} \qquad i \geq 0, \\ c(a_i) &= 01^{2i+5} \qquad i \geq 0, \\ c(R) &= 01,\ c(L) = 011,\ c(S) = 0111. \end{aligned}$$

Now the transition (3.12) is encoded to the word

$$c(t) = c(q)c(a)c(q')c(a')c(d),$$

and a TM consisting of transitions $t_1, \ldots, t_s$ to the word

$$c(M) = 0c(t_1)0c(t_2)\cdots 0c(t_s)00.$$

Note that here we assume that a TM is a **sequence** of transitions, and not a set of transitions, but this is OK. Consequently, we have encoded a TM into a finite word over a binary alphabet $\{0, 1\}$. Having such an encoded word we can easily find, for example:

- the i'th transition by searching a word in between the i'th and $(i + 1)$'st occurrences of 00.
- the direction in the ith transition by searching the number of 1's just before the $(i + 1)$'st occurrence of 00.

After fixing the above formalism we can rewrite (3.11) as

$$U\big(c(M)c(w)\big) = c(M(w)). \tag{3.11'}$$

Note that here we have required that the output of U is obtained in encoded form — this is necessary if we think about M as a method to compute an algorithmic function, otherwise $c(M(w))$ can also be interpreted as an output of M.

Example 3.18 (Universal Turing Machine). We construct a deterministic 3-tape TM U satisfying (3.11'). Then by Theorem 3.9 it can be transformed to 1-tape machine.

On an input $c(M)c(w)$, the machine U

- copies to second tape the word $c(w)$;
- marks the first symbol of w, or its encoding, in the second tape;
- creates to the third tape word $c(q_0)c(a)$, where a is the first symbol of w, i.e. $a = \mathrm{pref}_1(w)$.

Consequently, **at the beginning**, as well as at any state of the computation,

- the first tape contains the enconding of the machine M (and that of its input w);
- the second tape contains the encoded tape contents of M;
- the third tape contains the information of the next transition of M.

Next a **simulation step** consists of the following stages:

- based on the contents of the third tape, i.e. pair (q, a), U searches for the first tape the triple (q', a', d) determined by (q, a);
- U changes the contents of the third tape from $c(q)c(a)$ to $c(q')c(d')$;
- U makes the changes determined by the transition $(q, a) \to (q', a', d)$ on the second tape.

Moreover, if the new state of M above is **terminating**, M' moves also to the same terminating state after making the changes on the second tape (including the additional ones changing 01^7 to $*$), where the output is.

What we have to get convinced is that the above can be realized by using **only a finite number** of transitions, i.e. a finite number of states since the alphabet of U is already fixed to be $\{0, 1, \triangleright, *\}$.

What is done at the beginning, namely copying a word, creating a **marked** position, and creating a **constant** word, does not cause any problems. So consider a simulation step. Here we need a **comparison machine** to search for (q', a', d), and after finding it from the first tape, we can use **copying machine** to change the contents of the third tape to $c(q')c(d')$, as well as to copy $c(a')$ to the place of the marked position of the second tape. In this latter operations some **new space** or **some compression** of the tape maybe needed, but these can be done by Example 3.2. Finally, the marking has to be moved according to d. Also the operations required when M terminates does not cause problems.

So all in all we concluded, on an intuitive level, that only a finitely many different types of basic machines are needed to construct U. Of course, the detailed construction of U would be rather tedious and boring. However, the size of a universal Turing Machine need not be large: There is such machines with

- 6 states and 6 letters of the alphabet, or
- 7 states and 4 letters of the alphabet

(where, however, the model of TM's is slightly different). Hence such machine contains only about 30 transitions!

4 NP-Completeness

In this chapter we consider several algorithmic problems

- which are trivially algorithmically decidable simply by checking through all finitely many — normally exponentially many — possibilities;
- for which no polynomial time algorithm is known.

Moreover, what is common to these problems is that, if any of those can be solved in polynomial time, then this holds for all of those (and many more) problems. Such problems are called **NP-complete**.

Recall that we already defined the classes PTIME and NPTIME as classes of problems which can be solved in polynomial time by deterministic and nondeterministic TM's, respectively. We emphasize that these classes are independent of whether we consider 1-tape or many-tape machines (or also other modifications of Turing Machines, for example many-head machines). Actually, also other formalizations of algorithms would lead normally to the same classes of problems solved in polynomial time either deterministically or nondeterministically. However, remember that some **formalization is necessary** for the results of this chapter.

We concentrate here on **decision problems** P. As we specified, a DTM **decides** a problem giving an output "yes" or "no" depending on whether the problem instance has or has not the required property. A nondeterministic TM, in turn, solves a problem by **accepting** those instances of the problem which has the required property. We call these instances "yes-instances", in symbols i_y. P_y denotes the set of all yes-instances. Remember also that when instances of problems are inputs for TM's they must be encoded in a suitable way into the alphabet of the machine.

Next we modify the general notion of **reducibility**.

We say that problem P **reduces in polynomial time** to problem P', for short p-**reduces**, or in symbols $P \leq_p P'$, if there exists a DTM M_t such that

(i) M_t operates in polynomial time; and

(ii) M_t transforms an arbitrary $i \in P$ to an instance $M_t(i)$ of P' such that

$$i \in P_y \Leftrightarrow M_t(i) \in P'_y.$$

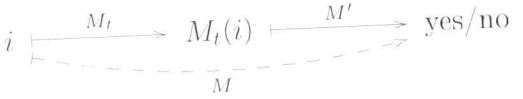
Further we say that P' is **NP-hard**, if each problem in class NPTIME p-reduces to it, and **NP-complete**, if it is NP-hard and in NPTIME.

We defined the above notions for "problems". Since we are operating on TM's we could define those in terms of "languages".

We have a few simple observations.

Fact 4.1. If $P \leq_p P'$ and $P' \in$ PTIME, then $P \in$ PTIME.

Proof. Assume that M_t transforms instances of P to those of P' in time $p(n)$, and that M' decides P' in time $p'(n)$. Then a DTM M deciding P can be constructed as follows:



It follows from (ii) above that:

$$\forall i \in P : i \in P_y \Leftrightarrow M(i) \in P'_y,$$

so that M decides P.

Moreover, M operates, for large enough inputs, in time

$$p(|i|) + p'\big(p(|i|)\big) = p''(|i|)$$

for some polynomial p''. Here we have assumed that $|M_t(i)| \leq p(|i|)$, which is natural since the output of M_t cannot be larger than the number of computation steps (Indeed, we can require that the output is read at the end of the computation, to modify it suitable for M'). Hence, M operates in polynomial time. □

Fact 4.2. If $P \leq_p P'$ and P is NP-hard, so is P'

□

Proof. An easy excersice.

Fact 4.3. If P is NP-complete, then

$$P \in \text{PTIME} \Leftrightarrow \text{PTIME} = \text{NPTIME}\,.$$

Proof. Let P be NP-complete.
"$\Rightarrow$". Assume that $P' \in$ NPTIME. Then by our assumption $P' \leq_p P$ so that $P' \in$ PTIME by Fact 4.1.
"$\Leftarrow$". Since P is NP-complete, it is in NPTIME, and hence also in PTIME. □

Next we illustrate the above notions in examples.

4.4. Define the following problems

TSP (Travelling Salesperson Problem)

Input: Weighted complete graph G with n nodes and a number N.

Output: "Yes", if G contains a cycle with a total weight at most N (A cycle follows the edges of the graph and visits in each of the nodes exactly ones, except the last node which is the same as the first).

HCP (Hamilton Cycle Problem)

Input: Graph G with n nodes;

Output: "Yes" if G contains a (Hamilton) cycle.

We claim that

$$\text{HCP} \leq_p \text{TSP}.$$

To prove this we transform an arbitrary n-node graph G (an instance of HCP) to an instance (G', N) of TSP as follows:

$$N := n$$
$$G': \quad u \overset{1}{\text{——}} v \text{ in } G' \quad \Leftrightarrow \quad u \text{——} v \text{ in } G$$
$$u \overset{2}{\text{——}} v \text{ in } G' \quad \Leftrightarrow \quad u \text{——} v \text{ not in } G.$$

Hence G' is a weighted complete graph.

It follows immediately that

$$G \text{ has a HC} \quad \Leftrightarrow \quad G' \text{ has a cycle of weight} \leq N.$$

So it remains to conclude that the transformation

$$G \mapsto (G', N) \tag{4.1}$$

can be computed in polynomial time. If we want to do it formally by a DTM we have to agree how the instances of HCP and TSP are encoded. A natural way is to list all nodes and edges of graphs, together with their possible weights, as well as number N. Hence, both $|G|$ and $\big|(G', N)\big|$ are of the size $\mathcal{O}(n^2 \log n)$, and the transformation 4.1 is very easy to realize by a DTM operating in (low) polynomial time on $|G|$. □

4.5. We show that

$$\text{MCP(3)} \leq_p \text{SAT}$$

where

MCP(3) (Map 3-Colourability Problem)

Input: A map of n countries

Output: "Yes" if the map can be coloured with three colours such that neighbours have different colours.

SAT (Satisfiability Problem)

Input: A formula of propositional logic with n unknowns;

Output: "Yes" if the formula can be **satisfied**, i.e. is true on some valuation of unknowns.

An instance of MCP(3) is

$$I = \{c_1, \ldots, c_n\} \cup \big\{(c_i, c_j) \mid c_i \text{ and } c_j \text{ are neighbours}\big\}.$$

We associate with I a formula $\alpha(I)$ as follows:

- the unknowns of $\alpha(I)$: $c_{i,j}$, $i = 1, \ldots, n$, $j = 1, 2, 3$ ("c_i has colour j")
- $\alpha(I)$ is a conjunction of the following formulas:

$$(c_{i1} \wedge \neg c_{i2} \wedge \neg c_{i3}) \vee (\neg c_{i1} \wedge c_{i2} \wedge \neg c_{i3}) \vee (\neg c_{i1} \wedge \neg c_{i2} \wedge c_{i3}), \tag{4.2}$$

for $i = 1, \ldots, n$, and

$$\neg\big((c_{i1} \wedge c_{j1}) \vee (c_{i2} \wedge c_{j2}) \vee (c_{i3} \wedge c_{j3})\big), \tag{4.3}$$

for all neighbours c_i and c_j.

It follows that

$$|\alpha(I)| = \mathcal{O}(|I|)$$

and that $\alpha(I)$ can be written in linear time from I by a DTM.

It remains to be shown that

$$I \text{ is } \textbf{3-colourable} \quad \Leftrightarrow \quad \alpha(I) \text{ is } \textbf{satisfiable}.$$

"$\Rightarrow$". Let χ be a 3-colouring of I. Set the valuation μ as follows

$$\mu(c_{i,j}) = \begin{cases} 1 & \text{if } \chi(c_i) = j \\ 0 & \text{otherwise.} \end{cases}$$

It follows directly that with this valuation $\alpha(I)$ assumes value $1 = \text{true}$.

"$\Leftarrow$". Assume that μ satisfies $\alpha(I)$, i.e. $\alpha(I)$ gets the value 1, when the unknowns are fixed according to μ. Now we define the colouring χ as follows

$$\chi(c_i) = j \quad \text{iff} \quad \mu(c_{i,j}) = 1.$$

Now since μ saisfies $\alpha(I)$ it satisfies both (4.2) and (4.3). In particular the former fact implies that each country gets a unique colour, and the latter fact that the neighbours are coloured with different colours. Hence, I is 3-colourable, so that we have proved our claim. □

4.6. Let us finally compare two natural variants of TSP, namely our earlier one and an optimization variant:

TSP: Is there a smaller cycle than N?
TSP(0): Find the smallest cycle.

We claim that:

$$\text{TSP is in PTIME} \quad \Leftrightarrow \quad \text{TSP(0) is in PTIME.}$$

"$\Leftarrow$". Obvious: For a given instance TSP find its optimal solution and test whether it is at most N.

"$\Rightarrow$". Consider an instance G of TSP(0), where G contains n nodes and $W_{\max}$ is the maximal weight. We solve it — using an algorithm to solve TSP — as follows:

1. Solve (G, N_0) where $N_0 = nW_{\max}$. Then we know that the optimum is in the interval $[0, N_0]$.
2. Solve $\left(G, \left\lfloor \frac{1}{2} N_0 \right\rceil\right)$ when we know that the optimum is either in the interval $\left[0, \frac{1}{2} N_0\right]$ or $\left[\frac{1}{2} N_0, N_0\right]$ and based on this
3. Solve either $\left(\frac{1}{4} N_0\right)$ or $\left(\frac{3}{4} N_0\right)$.
4. Repeating the process the optimum is found.

Hence the optimum is found by solving $\mathcal{O}(\log_2 N_0)$ instances of TSP's of size $\mathcal{O}(|G|)$. Meaning that if TSP can be solved in time p, then TSP(0) can be solved in time $\mathcal{O}\big(|G| \cdot p(|G|)\big)$. □

As is clear the above examples really do not need formal definition of an algorithm, i.e. a TM, since we just show that a particular problem is reducable in polynomial time to another one. The situation changes if we want to show that any problem solvable in polynomial time (nondeterministically) is reducable to a fixed other one. Then **we need to formalize the polynomial time nondeterministic algorithms**.

The above in mind we next show the **existence of NP-complete** problems — a fundamental result of complexity theory.

We consider the **Satisfiability Problem** SAT of example 4.5. An instance of this problem is a formula of propositional logic which is built

- from **Boolean variables**,
- by combining those with **logical connectives** such as and $\wedge$, or $\vee$, not $\neg$, implication $\Rightarrow$, equivalence $\Leftrightarrow$, etc.

A formula is **satisfiable** if its value is 1 (= true) for some valuation of the variables. The value of a formula is defined recursively by the truth values of the connectives in the natural way, for example

A	0	1
$\neg A$	1	0

$A \wedge B$:

A \ B	0	1
0	0	0
1	0	1

$A \Rightarrow B = \neg A \vee B$:

A \ B	0	1
0	1	1
1	0	1

In above we didn't fix precisely what are the connectives, in deed in our next proof we use the following n-ary function

$$\bigoplus_{i=1}^{n} x_i \text{ is true} \quad \text{iff} \quad \exists | i : x_i \text{ is true.}$$

Clearly, we have

$$\bigoplus_{i=1}^{n} x_i = \bigvee_{i=1}^{n} \Big(x_i \wedge \big(\bigwedge_{j \neq i} \neg x_j \big) \Big). \tag{4.4}$$

Consequently, $\bigoplus$ can be expressed by using only $\wedge$, $\vee$ and $\neg$. Actually, as known from logic, this is true for any n-ary Boolean function.

Our formalization of SAT asks whether a **given formula combined using $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$ and $\bigoplus$ is satisfiable**.

Theorem 4.7 (Cook, 1971). SAT is NP-complete.

Proof. I We show that SAT is in NPTIME. To do this we have to describe a NTM M which accepts satisfiable instances. This is not difficult: Given a formula α (encoded for a TM) containing N unknowns, the machine M

- guesses the values of these unknowns,
- traverses through the formula α and replaces each subformula containing only one connective to its truth value under above guess, and
- this is repeated as long as the value of α is obtained.

Clearly, M accepts exactly the satisfiable instances. Moreover, since the number of traversals is linear in $|\alpha|$, M operates in square time.

II We show that $L \leq_p SAT$ for any L accepted by a NTM in a polynomial time. Let L be accepted by a NTM

$$M = (Q, \Sigma, \delta, s)$$

operating in time $p(n)$ **on all inputs** of length n. As we saw, we can assume that M is 1-tape machine. We denote

$$Q = \{ s = q_0, q_1, \ldots, q_{k-1} \},$$

$$yes = q_k$$

and

$$\Sigma = \{a_0 = *, a_1, \ldots, a_{m-1}, a_m = \triangleright\}.$$

We associate with each input w a formula $\alpha(w)$ satisfying

$$w \in L(M) \quad \Leftrightarrow \quad \alpha(w) \text{ is satisfiable.} \tag{4.5}$$

Consequently, if the transformation $w \mapsto \alpha(w)$ can be done in polynomial time by a DTM our proof is complete.

Now, let w be an input of M with $|w| = n$. Then $w \in L(M)$ iff

- when starting from a configuration $(s; \triangleright, w)$,
- and performing some at most $p(n)$ computation steps,
- M terminates in the state yes.

Each of these conditions corresponds a part of $\alpha(w)$, namely $\mathrm{Init}(w)$, Trans and End below.

The variables of $\alpha(w)$ will be

$$a_i^{s,t}, \; q_j^{s,t} \qquad \text{for } i = 0, \ldots, m;\; j = 0, \ldots, k;\; s, t = 0, \ldots, p(n).$$

Note that the number of above variables is $\mathcal{O}(p(n))^2$.

We define

$$\alpha(w) = \mathrm{Conf} \wedge \mathrm{Init}(w) \wedge \mathrm{Trans} \wedge \mathrm{End} \tag{4.6}$$

in the following way:

$$\mathrm{Conf} = \bigwedge_{t=0}^{p(n)} \Big(\Big(\bigwedge_{s=0}^{p(n)} \big(\bigoplus_{i=0}^{m} a_i^{s,t} \big) \Big) \wedge \Big(\bigoplus_{s=0;j=0}^{p(n);k} q_j^{s,t} \Big) \Big);$$

$$\mathrm{Init}(w) = q_0^{0,0} \wedge a_m^{0,0} \wedge a_{i_1}^{1,0} \wedge \cdots \wedge a_{i_n}^{n,0} \wedge a_0^{n+1,0} \wedge \cdots \wedge a_0^{p(n),0},$$

where $w = a_{i_1} \cdots a_{i_n}$, with $a_{i_j} \in \Sigma$;

$$\mathrm{Trans} = \Big(\bigwedge_{t=0}^{p(n)} \bigwedge_{s=0}^{p(n)} \bigwedge_{i=0}^{m} \Big(\neg \Big(\bigvee_{j=0}^{k} q_j^{s,t} \Big) \Rightarrow \big(a_i^{s,t} \Leftrightarrow a_i^{s,t+1} \big) \Big) \Big)$$

$$\wedge \Big(\bigwedge_{t=0}^{p(n)} \bigwedge_{s=0}^{p(n)} \bigwedge_{i=0}^{m} \bigwedge_{j=0}^{k} \Big(\big(a_i^{s,t} \wedge q_j^{s,t} \big) \Rightarrow \bigoplus_{r=1}^{l} \big(a_{i_r}^{s,t+1} \wedge q_{j_r}^{s+\tilde{d}_r,t+1} \big) \Big) \Big),$$

where

$$\delta(q_j, a_i) = \big\{(q_{j_r}, a_{i_r}, d_r) \;\big|\; r = 1, \ldots, l\big\}, \qquad \delta(q_k, a_i) = \big\{(q_k, a_i, S)\big\} \tag{4.7}$$

and

$$\hat{d} = \begin{cases} -1 & \text{if } d = L, \\ 0 & \text{if } d = S, \\ 1 & \text{if } d = R; \end{cases}$$

and finally,

$$\text{End} = \bigvee_{s=0}^{p(n)} q_k^{s,p(n)}.$$

We have to show that

(i) $\alpha(w)$ satisfies (4.5), and

(ii) the transformation $w \mapsto \alpha(w)$ can be computed by a DTM.

Let us prove first (ii). It is important to note that the machine M here (and hence also k, m and r) are **constants**. Hence we have

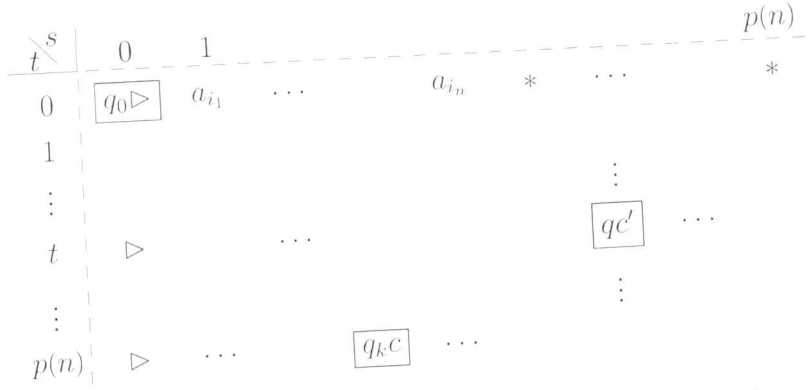
$$|\alpha(w)| = \mathcal{O}\big(p(n)^2\big),$$

when the length of a formula is defined as a total number of characters needed to write it down. However, note that as an (encoded) input of a TM its length is $\mathcal{O}\big(p(n)^2 \log p(n)\big)$, cf. Exercise I-1. Now, to compute $\alpha(w)$ from w it is essentially enough to count up to $p(n)$ several times and when doing that simultaneously writing down $\alpha(w)$. Clearly a DTM can compute $p(n)$ from $|w| = n$ (if necessary we can make $p(n)$ larger and simpler!). Hence (ii) is proved.

We turn to prove the essential part, i.e. (i).

Assume first that $w \in L(M)$. Then w has an accepting computation of length at most $p(n)$. This can be visualized by a $(p(n)+1) \times (p(n)+1)$ matrix where the t'th line contains the configuration of M after the t'th step, i.e. a letter in each entry

and state only in one entry of each line. Moreover, the state in the final line must be $q_k = yes$ — indeed we repeat the line if M terminates earlier.



Note that when defining the unknowns $a_i^{s,t}$ and $q_j^{s,t}$ we let s (– space) and t (– time) to run from $0, \ldots, p(n)$. The above guides us to define the valuation μ evaluating $\alpha(w)$:

$$\mu\big(a_i^{s,t}\big) = \begin{cases} 1 & \text{if after the } t\text{'th step in the } s\text{'th cell there is a letter } a_i \\ & \text{on computation of } w, \\ 0 & \text{otherwise;} \end{cases}$$

$$\mu\big(q_j^{s,t}\big) = \begin{cases} 1 & \text{if after the } t\text{'th step } M \text{ scans the } s\text{'th cell in state } q_j \text{ on} \\ & \text{computation of } w, \\ 0 & \text{otherwise.} \end{cases}$$

It follows directly that each of the components in (4.6) gets the value 1 and so does (4.6). For example, in Conf each $\bigoplus$-term gets the value 1, hence $\mu(\text{Conf}) = 1$, too. Formula Init(w) is equally easy to check. In formula Trans the two implications corresponds those which **are not** and **are involved** in a computation step of M, respectively. And the above valuation makes both of these true. Finally, End gets the value 1 since, due to (4.7) where the δ is extended corresponding to the fact that accepting computations are extended artificially, the last line of the table contains the state yes. Hence, indeed $\mu(\alpha(w)) = 1$.

Second we assume that $\alpha(w)$ is satisfiable, say γ satisfies it. We have to show that w is accepted by M. We note first that since $\gamma(\text{Conf}) = 1$, necessarily

$$\forall t, \forall s : \exists | i : \gamma\big(a_i^{s,t}\big) = 1$$

and

$$\forall t : \exists | j, s : \gamma\left(q_j^{s,t}\right) = 1.$$

In other words, for each $t = o, \ldots, p(n)$, those values of the variables which are equal to 1 **determines the unique configuration** of M, i.e. a candidate for each of the lines in our above illustration. Let c_t, for $t = o, \ldots, p(n)$, be such a configuration. Then, for example, the first letter in c_t is a_i if $\gamma\left(a_i^{0,t}\right) = 1$, and the state in c_t is q_j if $\gamma\left(q_j^{s,t}\right) = 1$, for some (which is actually unique) s, and moreover in c_t M scans the s'th cell of the tape.

Next from the assumption $\gamma(\text{Init}(w)) = 1$ it follows that the configuration c_0 is the initial one of M on input w.

Similarly, from the fact that $\gamma(\text{End}) = 1$ we conclude that the uniquely specified state of $c_{p(n)}$ is the accepting one *yes*.

To conclude the proof we have to show that

$$c_t \rightarrow_M c_{t+1}, \tag{4.8}$$

for $t = o, \ldots, p(n)$. Indeed, as already shown the configurations c_i are well defined, c_0 is the initial one on w, and $c_{p(n)}$ is the final one. Let us fix t. Further let s be a position in c_t, and a_i the symbol in this position. Assume first that s is not being scanned by the head of M. This means that $\gamma\left(q_j^{s,t}\right) = 0$ for all $j = 0, \ldots, k$, and therefore $\gamma\left(\neg\left(\bigvee_{j=0}^{k} q_j^{s,t}\right)\right) = 1$. But since $\gamma(\text{imp}_1) = 1$, where imp_1 is the first implication in Trans, necessarily $\gamma\left(a_i^{s,t} \Leftrightarrow a_i^{s,t+1}\right) = 1$, implying that $\gamma\left(a_i^{s,t}\right) = \gamma\left(a_i^{s,t+1}\right)$. This, in turn, means that the letter in the s'th position of c_{t+1} is the same as the corresponding letter in c_t — as was to be proved.

Second assume that the s'th letter a_i in c_t is being scanned by M in state q_j. Then $\gamma(a_i s, t) = \gamma(q_j s, t) = 1$, and since $\gamma(\text{imp}_2) = 1$, where imp_2 is the second implication of Trans, it follows that

$$\gamma\Big(\bigoplus_{r=1}^{l}\left(a_{i_r}^{s,t+1} \wedge q_{j_r}^{s+\hat{d}_r,t+1}\right)\Big) = 1.$$

But this means that exactly one member of $\bigoplus$ is true, implying that s'th letter of c_{t+1} is obtained by using a transition step of M to c_t, and the state is changed correctly, and finally the head is moved according to this transition.

Hence, indeed (4.8) and the whole theorem is proved. □

Our next result shows that in SAT we can restrict to formulas of special standard form. We say that a formula is in **conjunctive normal form**, CNF for short, if it is of the form

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{n_i} x_{i,j},$$

where each $x_{i,j}$ is either a variable or its negation. Moreover, a formula is in k-**conjunctive** normal form, k-CNF for short, if in above

$$n_i \leq k \qquad \text{for } i = 1, \ldots, n.$$

Now, by i-**CNF Problem** (**CNF-Problem**, respectively) we mean a subproblem of SAT, where only formulas in i-CNF (CNF, respectively) are considered.

Theorem 4.8. 3-CNF is NP-complete.

Proof. Let us prove first that Theorem 4.7 remains if only formulas in CNF are considered. As is well-known each formula can be written equivalently in CNF, but what is its length then! So consider the formula (4.6)

$$\alpha(w) = \mathrm{Conf} \wedge \mathrm{Init}(w) \wedge \mathrm{Trans} \wedge \mathrm{End}.$$

As we noted

$$|\alpha(w)| = \mathcal{O}\big(p(n)^2\big).$$

The formulas $\mathrm{Init}(w)$ and End are already in CNF.

The formula Trans can be treated as follows: The first implication is of constant length and hence can be transformed into CNF by formulas

$$x \Rightarrow y \qquad \equiv \qquad \neg x \vee y$$

and

$$x \Leftrightarrow y \qquad \equiv \qquad (x \Rightarrow y) \wedge (y \Rightarrow x),$$

without changing the length of the formula more than by a constant factor. The same applies for the second implication of Trans after noticing the identity

$$\bigoplus_{i=1}^{p} x_i \equiv \bigvee_{i=1}^{p} x_i \wedge \neg\Big(\bigvee_{\substack{i,j=1 \\ i \neq j}}^{p} (x_i \wedge x_j)\Big) \equiv \bigvee_{i=1}^{p} x_i \wedge \Big(\bigwedge_{\substack{i,j=1 \\ i \neq j}}^{p} (\neg x_i \vee \neg x_j)\Big).$$

The most complicated formula is Conf. We can apply the above formula also here, but the subformula

$$\bigoplus_{s=0;j=0}^{p(n);k} q_j^{s,t}$$

leads to CNF formula of length $\mathcal{O}(p(n)^2)$, which means that the whole Conf can be transformed into a CNF formula of length $\mathcal{O}(p(n)^3)$. So we concluded that the formula $\alpha(w)$ can be transformed into a CNF formula of length $\mathcal{O}(p(n)^3)$. As an input for a TM its length can be assumed to be $\mathcal{O}(p(n)^3 \log p(n))$. Since this is polynomial in n, and clearly the transformation of $\alpha(w)$ into CNF can be realized by a DTM operating in polynomial time, we have proved that the CNF Problem is NP-complete.

To conclude the proof of Theorem 4.7 we show

$$\text{CNF} \leq_p \text{3-CNF}.$$

Since, clearly 3-CNF is in NPTIME, this indeed completes the proof. To prove this consider the formulas

$$x_1 \vee \cdots \vee x_n \tag{4.9}$$

and

$$\begin{aligned}(x_1 \vee x_2 \vee y_2) \wedge (\neg y_2 \vee x_3 \vee y_3) \wedge \cdots \\ \wedge (\neg y_{n-3} \vee x_{n-2} \vee y_{n-2}) \wedge (\neg y_{n-2} \vee x_{n-1} \vee x_n),\end{aligned} \tag{4.10}$$

where $y_2, \ldots, y_{n-2}$ are new variables.

Claim. A valuation $\mu : \{x_1, \ldots, x_n\} \to \{0, 1\}$ satisfies (4.9) if and only if some of its extensions to $\{x_1, \ldots, x_n, y_2, \ldots, y_{n-2}\}$ satisfies (4.10).

First assume that μ satisfies (4.9). Then μ satisfies one of the variables, say x_i, and we can set $\mu(y_j) = 1$ if $j < i$ and $\mu(y_j) = 0$ if $j \geq i$. Conversily, assume that some extension of μ satisfies (4.10). Now, if $\mu(y_2) = 0$ then necessarily $\mu(x_1 \vee x_2) = 1$, and so (4.9) is satisfied. On the other hand, if $\mu(y_{n-2}) = 1$ then $\mu(x_{n-1} \vee x_n) = 1$ and again (4.9) is satisfied. In the remaining case there exists i such that $\mu(y_i) = 1$ and $\mu(y_{i+1}) = 0$. But then necessarily $\mu(x_{i+1}) = 1$, and again (4.9) is satisfied.

By the above construction an arbitrary instance α of CNF can be transformed into an instance $T(\alpha)$ of 3-CNF such that

$$\alpha \text{ is satisfiable} \quad \Leftrightarrow \quad T(\alpha) \text{ is satisfiable.}$$

Further $|T(\alpha)| = \mathcal{O}(|\alpha|)$, so that the transformation can be done in polynomial time. □

Our next result shows an optimality of Theorem 4.8.

Theorem 4.9. 2-CNF is in PTIME.

Proof. Let α be a formula in 2-CNF with n variables. We define three steps:
I We **remove** from α all disjunctions containing both x and $\neg x$ and **replace** each disjunction $x \vee x$ by the formula x. If the formula α' obtained is empty, it is always true, and hence satisfiable, and in general α' is satisfiable iff α is so.

II We **replace** α' by α'', where α'' does not contain in any disjunction only one term. This is done as follows. If α' contains such a term x and it contains also disjunction $\neg x$, then α' is not satisfiable and we are done. Otherwise we remove from α the term x, and all the disjunctions containing x, and erase from all disjunctions the terms $\neg x$. Clearly, the formula thus constructed is satisfiable iff the original one is so.

Finally, we do the same for terms $\neg x$, and repeat the whole procedure, until the required α'' is found.

III We **eliminate** one variable from α'' as follows. If in α'' no variable occurs as negated and nonnegated, then α'' is satisfiable and we are done. Otherwise we pick up an variable x and divide the disjunctions of α'' containing x into two classes

$$x \vee \beta_1, \ldots, x \vee \beta_2$$

and

$$\neg x \vee \gamma_1, \ldots, \neg x \vee \gamma_2,$$

and choose δ such that

$$\alpha'' = \bigwedge_{i=1}^{m} (x \vee \beta_i) \bigwedge_{j=1}^{k} (\neg x \vee \gamma_j) \wedge \delta,$$

where the disjunctions might be in a new order and repetitions are removed.

It follows that

$$\alpha'' \text{ is satisfiable}$$

$$\Updownarrow$$

$$\big((\beta_1 \wedge \cdots \wedge \beta_m) \vee (\gamma_1 \wedge \cdots \gamma_m)\big) \wedge \delta \text{ is satisfiable.}$$

Now, finally as a conclusion of step III we replace our last formula by an equivalent 2-CNF formula

$$\alpha''' = \bigwedge_{i=1;j=1}^{m;k} \big(\beta_i \vee \gamma_j\big) \wedge \delta,$$

where, moreover, repetitions are removed.

In steps I–III

$$\alpha \xrightarrow{\text{I}} \alpha' \xrightarrow{\text{II}} \alpha'' \xrightarrow{\text{III}} \alpha'''$$

we eliminated one variable, without changing the satisfiability, so that in at most $n-1$ rounds a formula, for which the satisfiability is trivial to check, is obtained.

It remains to analyze the complexity:
In steps I and II the formula becomes shorter. Moreover, the former can be done in time $\mathcal{O}(|\alpha|)$, and the latter in time $\mathcal{O}\big(|\alpha|^2\big)$ simply by traversing the formula; the bound $\mathcal{O}\big(|\alpha|^2\big)$ comes since II might require iterative use of the procedure.

The step III is a bit more complicated. Now, unfortunately the new formula α''' can be longer than the old one α''; however, surely

$$|\alpha'''| \leq |\alpha''| + 4n^2. \tag{4.11}$$

This is because $m, k \leq 2n$. Also the transformation III can be done in time $\mathcal{O}(|\alpha'''|)$, and hence also in $\mathcal{O}\big(|\alpha|^2\big)$.

It follows that the whole transformation $\alpha \longrightarrow \alpha'''$ (eliminating one variable) can be done in time $\mathcal{O}\big(|\alpha|^2\big)$. A problem however remains: the size $|\alpha'''|$ is quadratic in

$|\alpha|$, i.e. much larger. But as we concluded in (4.11) during the next full round it can grow only by an additive factor $4(n-1)^2$. Hence, in any step of the algorithm the length of the formula remains smaller than

$$|\alpha| + 4\sum_{i=0}^{n} i^2 = \mathcal{O}(|\alpha|^3).$$

Therefore steps I $\to$ II $\to$ III can always be done in time $\mathcal{O}(|\alpha|^6)$ and the whole algorithm works in time $\mathcal{O}(|\alpha|^7)$.

Of course, the above can be realized by a DTM in polynomial time. However, the degree may be higher. □

After knowing one NP-complete problem, it is rather easy to find other such problems by p-reducing a known NP-complete problem to a new problem which has to be only in the class NPTIME. We shall prove the following p-reductions:

$$\begin{array}{ccccccc} & & & & \text{GCP} & \leq_p & \text{ECP} \leq_p \text{KSP} \\ & & & \swarrow_p & & & \\ w \stackrel{?}{\in} L(M) & \leq_p & \text{CNF} & \leq_p & \text{3-CNF} & & \\ & & & \searrow_p & & & \\ & & & & \text{VCP} & \leq_p & \text{HCP} \leq_p \text{TSP} \end{array}$$

Here 3-CNF, HCP and TSP refer to satisfiability problem for 3-CNF formulas, Hamilton Cycle Problem and Travelling Salesperson Problem. The other problems are defined as follows:

GCP (Graph Colouring Problem)

Input: A natural number K and a graph G with n vertices.

Output: "Yes" if the vertices can be coloured with K colours such that all neighbours have different colours.

ECP (Exact Cover Problem)

Input: A finite set S and a finite family $S_1, \ldots, S_n$ of its subsets.

Output: "Yes" if there exists indices $i_1, \ldots, i_m$ such that

$$S = S_{i_1} \dot{\cup} \cdots \dot{\cup} S_{i_m},$$

where $\dot{\cup}$ denotes a disjoint union.

KSP (Knapsack Problem)

Input: Natural numbers $N, k_1, \ldots, k_n$.

Output: "Yes" if there exist indices $i_1, \ldots, i_m$ such that

$$N = \sum_{j=1}^{m} k_{i_j} \qquad \text{and} \qquad i_t \neq i_s \quad \text{for } t \neq s.$$

VCP (Vertex Cover Problem)

Input: A graph $G = (V, E)$ with $|V| = n$, and a number $K \leq n$.

Output: "Yes" if G has a vertex cover of size $\leq K$, i.e. a subset $V' \subseteq V$ such that $|V'| \leq K$ and

$$(u, v) \in E \qquad \Rightarrow \qquad u \in V' \text{ or } v \in V'.$$

We start with

Lemma 4.10. 3-CNF $\leq_p$ GCP.

Proof. Let

$$\alpha = f_1 \wedge \cdots \wedge f_m,$$

with

$$f_i = x_i \vee y_i \vee z_i$$

be an instance of 3-CNF, where each x_i, y_i and z_i is one of the variables $a_1, \ldots, a_k$ or their negations. We associate α with an instance $I(\alpha)$ of GCP as follows: $I(\alpha) =$

$(G(\alpha); K) = (V(\alpha), E(\alpha); K)$:

$$\begin{aligned} K &= k+1 \\ V(\alpha) &= \{f_1, \ldots, f_m, a_1, \ldots, a_k, \neg a_1, \ldots, \neg a_k, v_1, \ldots, v_k\} \\ E(\alpha) &= \big\{(v_i, v_j) \mid i \neq j\big\} \cup \big\{(v_i, a_j) \mid i \neq j\big\} \\ &\quad \cup \big\{(v_i, \neg a_j) \mid i \neq j\big\} \cup \big\{(a_i, \neg a_i) \mid i = 1, \ldots, k\big\} \\ &\quad \cup \big\{(f_i, a_j) \mid a_j \text{ does not occur in } f_i\big\} \\ &\quad \cup \big\{(f_i, \neg a_j) \mid \neg a_j \text{ does not occur in } f_i\big\}. \end{aligned}$$

Clearly, $G(\alpha)$ can be computed from α in polynomial time by a DTM.

We fix $i \in \{1, \ldots, k\}$ and consider the subset $V(i) = \{v_1, \ldots, v_k, a_i, \neg a_i\}$ of $V(\alpha)$. **It can be coloured** by $K = k+1$ colours: choose a different colour for each of the v_j's, the colour of v_i either to a_i or $\neg a_i$, and a new colour for the other. On the other hand $V(i)$ **requires** $k+1$ **colours**: v_i's must be coloured differently (since $(v_i, v_j) \in E(\alpha)$ if $i \neq j$), a_i and $\neg a_i$ must have different colour, and only at most one of those can be a colour of some v_j namely that of v_i.

The crucial part of the proof is to show:

$$\alpha \text{ is satisfiable} \qquad \text{iff} \qquad G(\alpha) \text{ is } K\text{-colourable.}$$

I Assume that μ $(k+1)$-colours $G(\alpha)$. This means, by above, that v_i's are coloured with different colours, say $\mu(v_i) = c_i$, and one of the vertices a_j and $\neg a_j$, for $j = 1, \ldots, k$, has the colour of v_j and the other has a new colour c_{k+1}. Let y_j be that of the vertices a_j and $\neg a_j$ for which $\mu(y_j) = c_{k+1}$.

Claim 1. $\mu(f_i) \neq c_{k+1}$ for $i = 1, \ldots, m$.

Proof. Fix i. By the construction of $G(\alpha)$

$$\text{the number of neighbours of } f_i \geq 2k - 3.$$

Hence assuming that $k \geq 4$ (which can be done) we have

$$\text{the number of neighbours of } f_i \geq k + 1.$$

Consequently, for some j, both a_j and $\neg a_j$ are neighbours of f_i, proving the claim.

Claim 2. For each i, f_i contains some of the terms $\neg y_1, \ldots, \neg y_k$.

Proof. If this is not the case, then all vertices $\neg y_j$ are neighbours of f_i. Hence $\mu(f_i) \neq \mu(\neg y_j)$ for all j. But by the choice of y_j's $\mu(\neg y_j) = c_j$, i.e. $\mu(f_i) = c_{k+1}$; a contradiction with Claim 1.

From Claim 2. we immediately see that α is satisfiable: Let us give to $\neg y_j$ the value 1 for all j.

II Assume that α is satisfied by the valuation $\mu : \{a_1, \ldots, a_k\} \to \{0, 1\}$. As we noticed at the beginning the verticies v_i, a_i and $\neg a_i$, for $i = 1, \ldots, k$, can be coloured by K colours such that v_i has a colour c_i, one of the vertices a_i and $\neg a_i$ has the colour of v_i and the other the colour c_{k+1}. It remains to be shown that the verticies f_i can be coloured by some of these colours (due to the satisfiability of α).

We define, for $j = 1, \ldots, k$, y_j by the condition

$$y_j = \begin{cases} a_j & \text{if } \mu(a_j) = 1 \\ \neg a_j & \text{otherwise.} \end{cases}$$

Hence $\mu(y_j) = 1$. By our assumption $\mu(\alpha) = 1$, and so $\mu(f_i) = 1$ for all i. Consequently each f_i contains some y_j. Hence we can fix the colour of f_i to be that of y_j, and indeed then the colour of f_i is different to any of its neighbours. This proves that $G(\alpha)$ is K-colourable. □

Lemma 4.11. GCP $\leq_p$ ECP.

Proof. Let

$$i = (G, K) \quad \text{with} \quad G = (V, E) = (\{v_1, \ldots, v_n\}, \{e_1, \ldots, e_m\})$$

be an instance of GCP. We associate with it an instance i' of ECP as follows:

$$S = V \cup \big\{[e, l] \mid e \in E, l \in \{1, \ldots, K\}\big\}$$

and S_i's consists of

$$C_{vl} = \{v\} \cup \big\{[e, l] \mid v \text{ is a vertex in } e\big\}$$

for $v \in V$, $l \in \{1, \ldots, K\}$, and

$$D_{el} = \{[e, l]\}$$

for $e \in E$, $l \in \{1, \ldots, K\}$.

As usual there is no problems to compute $i \mapsto i'$ in polynomial time. So it remains to be proved that

$$G \text{ is } K\text{-colourable} \iff S \text{ has an exact cover: } S = \dot{\bigcup_j} S_{i_j}.$$

I We assume that μ K-colours G, and show that then

$$S = \dot{\bigcup_{v \in V}} C_{v\mu(v)} \,\dot\cup\, \Big(\dot{\bigcup} \{D_{el} \mid [e, l] \notin C_{v\mu(v)}\}\Big). \tag{4.12}$$

Clearly, S is a union of sets on the right hand side. Further the singleton sets D_{el} cannot cause any overlapping. Hence, it suffices to consider the intersections

$$L = C_{v\mu(v)} \cap C_{u\mu(u)}$$

with $u \neq v$. If $[e, l] \in L$, then $(u, v) \in E$ and $\mu(u) = \mu(v)$. This, however, contradicts with the fact that μ is a K-colouring. This proves (4.12).

II Assume that i' possesses an exact cover using sets C_{vl} and D_{el}. Then for each $v \in V$ there exists the unique C_{vl_v} belonging to this cover. We claim that G can be K-coloured by setting

$$\mu(v) = l_v.$$

First of all μ assigns to each vertex a unique colour. It is also a K-colouring since:

$$e = (u, v) \in E,\ \mu(u) = \mu(v) \quad \Longrightarrow \quad [e, l_v] = [e, l_u] \in C_{ul_u} \cap C_{vl_v} = \emptyset. \qquad \square$$

As the final reduction of one branch in Figure of page 50 we prove:

Lemma 4.12. ECP $\leq_p$ KSP.

Proof. Consider an instance

$$P : S = \{a_1, \ldots, a_n\}, \quad S_1, \ldots, S_m$$

of ECP, and associate to each set S_j a binary number

$$\text{bin}(S_j) = \alpha_{jn}\alpha_{j(n-1)}\cdots\alpha_{j1},$$

where

$$\alpha_{ji} = \begin{cases} 0^l1 & \text{if } a_i \in S_j \\ 0^l0 & \text{if } a_i \notin S_j, \end{cases}$$

and $l = [\log_2 n] + 1$. In particular $\text{bin}(S) = (0^l1)^n$. Further let

$$P' : \text{bin}(S);\ \text{bin}(S_1), \ldots, \text{bin}(S_m)$$

be an instance of KSP.

Clearly P' is easily computable from P by a DTM, so that it remains to be shown that

$$S \text{ has an exact cover using } S_i\text{'s}$$

$$\Updownarrow$$

$$\exists i_1, \ldots, i_k : \text{bin}(S) = \sum_{j=1}^{k} \text{bin}(S_{i_j}) \quad \text{with } i_j \neq i_{j'}, \text{ if } j \neq j'.$$

This, however, is obvious from the construction, since the use of 0's at the beginnings of α_{ij}'s eliminates potential problems of carry numbers. □

The following example shows the importance of the fact how the problem instance is encoded.

Example 4.13. We claim that a **Unary Knapsack Problem** is decidable in polynomial time.

UKSP (Unary Knapsack Problem)

Input: Natural numbers N ans $k_1, \ldots, k_m$ given unary.

Output: As in the ordinary Knapsack Problem.

A natural encoding of an instance of UKSP is

$$1^N \# 1^{k_1} \# \cdots \# 1^{k_m},$$

and it is a "yes"-instance iff $N = \sum_{i \in I} k_i$ for some $I \subseteq \{1, \ldots, m\}$. A polynomial time algorithm is based on the following two facts (for more details cf. class):

- The set of "yes"-instances is CF language.
- The question "$w \in L$?" for CF languages can be decided in polynomial time.

Lemma 4.14. 3-CNF $\leq_p$ VCP.

Proof. Let

$$\alpha = \bigwedge_{j=1}^{m} f_j$$

be an instance of 3-CNF, where f_j's consists of variables

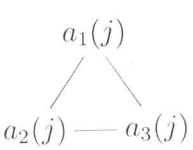
$$x_1, \ldots, x_n$$

and

$$|f_j| = 3$$

for all j. We associate with α an instance $\big(G = (V, e), K\big)$ of VCP as follows:

(i) G contains vertices x_i and $\neg x_i$, and edges x_i —— $\neg x_i$ for all i;

(ii) G contains subgraphs:  for $j = 1, \ldots, m$;

(iii) If $f_j = x_j \vee y_j \vee z_j$, then G contains edges:

$$a_1(j) \text{ —— } x_j \ , \quad a_2(j) \text{ —— } y_j \ \text{ and } \ a_3(j) \text{ —— } z_j \ ;$$

(iv) that is all what is in G;

(v) $K = n + 2m$.

Clearly, the transformation $\alpha \mapsto (G, K)$ is easy to compute.

We have to show:

$$\alpha \text{ is satisfiable} \iff G \text{ has a vertex cover of size at most } K.$$

We start with the following observations:

1. Each vertex cover of G contains necessarily either x_i or $\neg x_i$ for all i.
2. Each vertex cover of G contains at least two of the points $a_1(j)$, $a_2(j)$ and $a_3(j)$ for all j.
3. Only the edges of (iii) depend on α.

It follows from 1 and 2 that **each vertex cover of G contains at least $n + 2m = K$ vertices**. Hence, the existence of a required vertex cover is equivalent to the existence of a vertex cover of size K, which has to be characterized — via edges of (iii) — by the satisfiability of α. Moreover, each vertex cover of size K contains **one** of the vertices x_i and $\neg x_i$ for all i, and **two** of the vertices $a_1(j)$, $a_2(j)$ and $a_3(j)$ for all j.

"$\Leftarrow$": Let $V' \subseteq V$ be a vertex cover of G of size K. We fix the values of the variables x_i by

$$\mu(x_i) = \begin{cases} 1 & \text{if } x_i \in V', \\ 0 & \text{if } x_i \notin V', \text{ i.e., by above, } \neg x \in V'. \end{cases}$$

We claim that this makes $\mu(f_j) = 1$ for all j. Consider a fixed f_j. As we noted only two of the edges of (iii) can be covered by vertices in $\{a_1(j), a_2(j), a_3(j)\}$, so that the third must be covered by either a variable x_i belonging to V' or by a negation of a variable, say $\neg x_i$, belonging to V'. But in both cases $\mu(f_j)$ becomes true:

$$x_i \in V' \Longrightarrow \mu(x_i) = 1 \Longrightarrow \mu(f_j) = \cdots \vee \mu(x_i) \vee \cdots = 1,$$
$$\neg x_i \in V' \Longrightarrow \mu(x_i) = 0 \Longrightarrow \mu(\neg x_i) = 1 \Longrightarrow \mu(f_j) = \cdots \vee \mu(\neg x_i) \vee \cdots = 1.$$

"⇒": Assume that $\mu : \{x_1, \ldots, x_n\} \to \{0, 1\}$ satisfies α. We construct a subset $V' \subseteq V$ as follows:

$$x_i \in V' \quad \text{if} \quad \mu(x_i) = 1,$$
$$\neg x_i \in V' \quad \text{if} \quad \mu(x_i) = 0$$

and V' contains, for each $j = 1, \ldots, m$, two of the vertices $a_1(j)$, $a_2(j)$ and $a_3(j)$ defined as follows:

> Since $\mu(f_j) = 1$, the two first lines of the definition of V' implies that at least 1 of the edges of (iii) becomes covered either by x_i of $\neg x_i$ in V'. The two $a_k(j)$'s selected to V' are chosen such that also the other two edges of (iii) become covered.

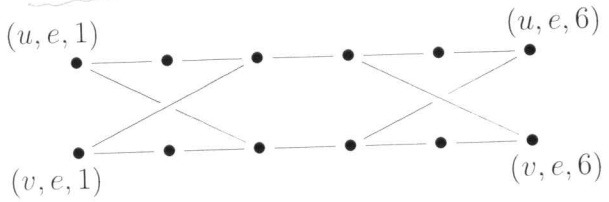
Clearly, $|V'| = n + 2m$. Moreover, by above, it covers all edges in (iii). It also contains one of the vertices x_i and $\neg x_i$, for all i, and hence covers edges in (i), as well as two of the vertices $a_k(j)$, for all j, and hence covers edges in (ii). So V' is a vertex cover of size K. □

Lemma 4.15. VCP $\leq_p$ HCP.

Proof. Let $\big(G = (V, E), K\big)$ be an instance of VCP. We associate with it an instance $G' = (V', E')$ of HCP as follows:

(i) V' contains vertices $a_1, \ldots, a_K$ — edges of these are defined in (iv);

(ii) For each $e = (u, v) \in E$, G' contains a subgraph:



12 vertices

14 edges;

(iii) For each vertex $v \in V$ of G, if the edges connected to v are

$$e_{v(1)}, \ldots, e_{v(\deg v)},$$

then G' contains the edges

$$\big(v, e_{v(i)}, 6\big) \text{ —— } \big(v, e_{v(i+1)}, 1\big)$$

for $i = 1, \ldots, \deg(v) - 1$;

(iv) For each pair $(v, i) \in V \times \{1, \ldots, K\}$ G' contains the edges

$$a_i \text{ —— } (v, e_{v(1)}, 1)$$

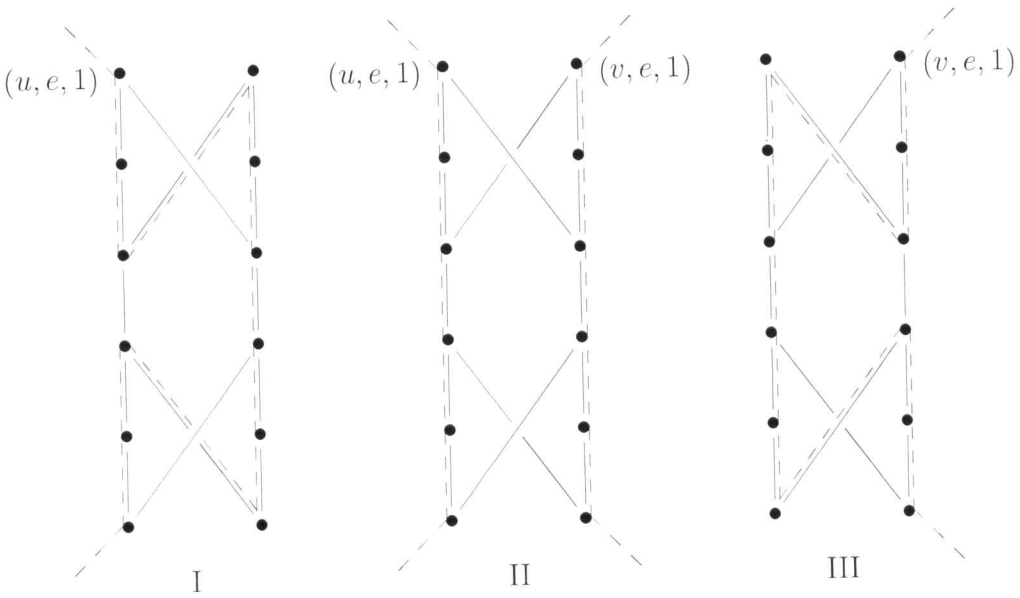
and

$$a_i \text{ —— } (v, e_{v(\deg v)}, 6)\ ;$$

(v) That is the whole construction of G'.

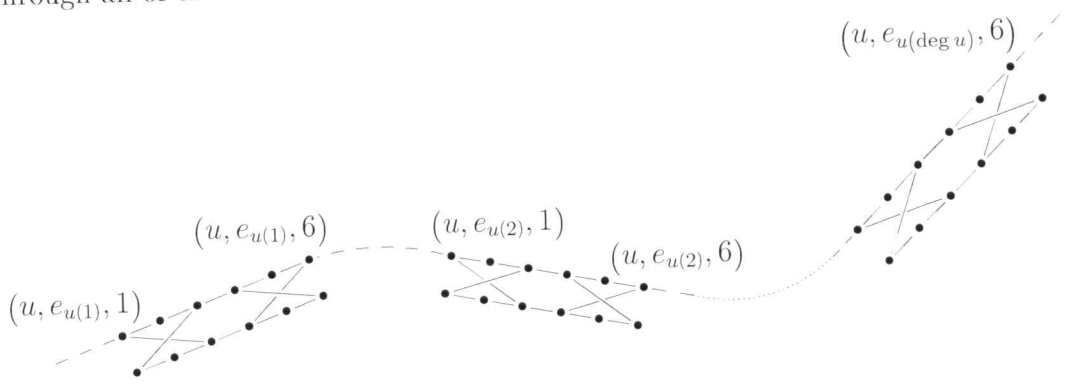
We start with the following observations:

a) The subgraphs of (ii) are connected to other vertices only via vertices $(u, e, 1)$, $(u, e, 6)$, $(v, e, 1)$ or $(v, e, 6)$. Therefore each HC of G' traverses through such subgraphs in one of the following ways (which **means that HC both enters and leaves a subgraph from the same side**, namely from u- or v-side):



b) The above subgraphs are connected to the rest of the graph either by edges of (iii) or edges of (iv). The former of those connect the u-sides (or v-sides) to each other such that a path can leave subgraphs associated with u (or v) only after travelling

through all of them. Therefore any HC is of the form



c) The endpoints of the above subgraphs corresponding to $e_{u(1)}$, i.e. the first edge of u, and $e_{u(\deg u)}$, i.e. the last edge of u, are connected to each a_i for $i = 1, \ldots, K$, by (iv).

Although the graph G' is a bit complicated, it is straightforward to construct it from (G, K) in a polynomial time.

We have to show that

$$G \text{ has a vertex cover of size at most } K \iff G' \text{ has a HC.}$$

"$\Leftarrow$": Assume that G' has a HC

$$C = \langle v_1, \ldots, v_n \rangle \qquad \text{with } n = |V'|.$$

Let us consider a subgraph C_s of C which

- starts from a vertex in $\{a_1, \ldots, a_K\}$,
- ends at a vertex in $\{a_1, \ldots, a_K\}$,
- does not visit these vertices in between.

By b) and c) C_s travels through exactly those subgraphs of (ii) which are connected to a vertex, say v. And by a) when traversing through these subgraphs C_s does it in one of the ways I–III above. Therefore the K vertices $\{a_1, \ldots, a_K\}$ of K' divides C into K subpaths, each of which defining a unique vertex v of V, namely that which corresponds to that $(v, e_{v(1)}, 1)$ or $(v, e_{v(\deg v)}, 6)$ of G' to which C leads from the considered a_i.

We claim that the set V_1 of vertices of G chosen as above is a vertex cover of G of size K. Clearly it is of size K. To prove that it is a vertex cover consider an edge $e = u \text{——} v$ of G. Since C is a HC it travels through the subgraph of (ii) associated to e. In doing so it enters to that subgraph either from u-side or v-side. But then, by the construction of V_1, u or v is in V_1, as was to be proved: e is covered by V_1.

"$\Rightarrow$": Let $V_1 \subseteq V$ be a vertex cover of G of size at most K. We may assume, possibly by adding to V_1 some vertices, that $|V_1| = K$, say

$$V_1 = \{v_1, \ldots, v_K\}.$$

We construct a HC of G', say C, as follows.

1. If $e = u \text{——} v \in E$, then C contains the edges I, II or III of the subgraph associated to e depending on whether

   $$\{u, v\} \cap V_1 = \{u\}, \{u, v\} \text{ or } \{v\}.$$

   Clearly, one of these alternatives holds since V_1 is a vertex cover.

2. C contains the edges

   $$\left(v_i, e_{v_i(j)}, 6\right) \text{——} \left(v_i, e_{v_i(j+1)}, 1\right)$$

   for $i = 1, \ldots, K$ and $j = 1, \ldots, \deg(v_i) - 1$.

3. C contains the edges

   $$a_i \text{——} \left(v_i, e_{v_i(1)}, 1\right)$$

   for $i = 1, \ldots, K$,

   $$a_{i+1} \text{——} \left(v_i, e_{v_i(\deg v_i)}, 6\right)$$

   for $i = 1, \ldots, K-1$, and

   $$a_1 \text{——} \left(v_K, e_{v_K(\deg v_K)}, 6\right).$$

The path C thus constructed is a HC:

- When starting from a_1 C leads to the subgraph associated with the 1st edge $e_{v_1(1)}$ of v_1, and then C travels through all or only half of the vertices of the subgraph, depending on whether the other endpoint of $e_{v_1(1)}$ is in the vertex cover V_1 or not;
- After that C goes through the subgraphs associated with the other edges $e_{v_1(2)}, \ldots, e_{v_1(\deg v_1)}$ of v_1, and moves to a_2;
- Then C repeats the same for v_2;
- Finally, after travelling through the subgraphs associated to v_K C goes back to a_1.

Now, since V_1 is a vertex cover, C visits in all subgraphs of (ii). If it does it twice, then it visits each vertex of a subgraph exactly once (case II). Otherwise it visits these subgraphs according to cases I or III, i.e. altogether again each vertex exactly once. Hence, indeed C is a HC. □

Now, we can combine Lemmas 4.10–4.12, 4.14 and 4.15 and Example 4.4 to

Theorem 4.16. The following problems are NP-complete: GCP, ECP, KSP, VCP, HCP and TSP.

Proof. It is straightforward to conclude that they are indeed in NP. □

Remark 4.1. The number of known NP-complete problems is already several hundreds, if not thousands!

Approximations of NP-complete problems. Here we consider methods of solving some NP-complete problems in polynomial time, not presicely but only approximatively. These methods can be of three different types:

A. "Almost surely". Here we require that $\mathcal{A}$

(i) operates always in polynomial time;

(ii) finds only correct answers;

(iii) finds a solution almost always:

$$\lim_{n\to\infty} \frac{|\{\text{inputs of size } n \text{ for which } \mathcal{A} \text{ works}\}|}{|\{\text{inputs of size } n\}|} = 1.$$

B. **"Usually fast"**. Here we require that $\mathcal{A}$

(i) finds always a correct answer;

(ii) operates in polynomial time **on average**.

C. **"Approximately correct"**. Here we require that $\mathcal{A}$

(i) operates in polynomial time;

(ii) yields always a result, which

(iii) deviates from the correct one only so and so much.

We consider each of these cases in the light of an example.

Case C. The optimization variant of TSP, i.e. TSP(0). As we saw in Example 4.6: TSP $\leq_p$ TSP(0) $\leq_p$ TSP. Further, by Example 4.4, we may restrict (in the sense that the problem remains NP-complete) to the case, where instances are **complete** graphs satisfying the **triangle inequality**:

$$w(u,v) \leq w(u,z) + w(z,v) \tag{4.13}$$

for all edges (u,v), (u,z) and (z,v). Let G be an instance of above type and S_{opt} its optimal solution. We construct an algorithm which finds a solution S such that

$$w(S) \leq 2w(S_{\text{opt}}).$$

Let e be an edge of S_{opt}. Our algorithm operates in the following 4 steps:

I Find a **minimal spanning tree** T of G, i.e. a tree of minimal weight containing all points of G. As is well-known T can be found by the **Greedy-method**, i.e. by performing $\mathcal{O}(|E|^2)$ comparisons of weights of edges.

II Change T into a multigraph T_2 by doubling all the edges.

III Find an **Euler Cycle** W of T_2, i.e. a cycle which traverses through each edge exactly once.

By the Euler's criterium such a cycle exists: A connected (multi-)graph possesses an Euler Cycle iff the **degree** of each of its vertices is even (cf. Graph Theory). Moreover, such a cycle can be found in linear time on $|E|$ by simply traversing through the graph: By starting from any point we continue as long as possible, whence the untouched edges of G forms a smaller graph satisfying Euler's Criterium, so that the induction applies.

IV We modify W to a cycle S as follows:

- We follow W as long as we reach some vertex for the second time;
- If all the vertices are visited before that we include the last edge to W, otherwise we continue (without taking the last edge) to the next vertex of W which has not yet been visited or to the initial vertex if no next vertex exists.

It follows directly that S is a cycle of G. Moreover, it can be found in polynomial time, more precisely by considering edges of G (and T_2) $\mathcal{O}(|E|^2)$ time. Finally, we have

$$\begin{aligned} w(S_{\text{opt}}) &> w(S_{\text{opt}} - e) \\ &\geq w(T) && \text{by the definition of } T \\ &= \frac{1}{2} w(T_2) && \text{by the definition of } T_2 \\ &\geq \frac{1}{2} w(S) && \text{by (4.13).} \end{aligned}$$

So we have proved

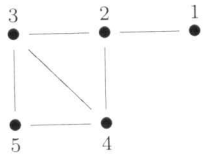
Theorem 4.17. "TSP(0) with an 100% error" can be solved in polynomial time.

There exists much better polynomial time algorithms for TSP(0). For example, 50% error can be achieved.

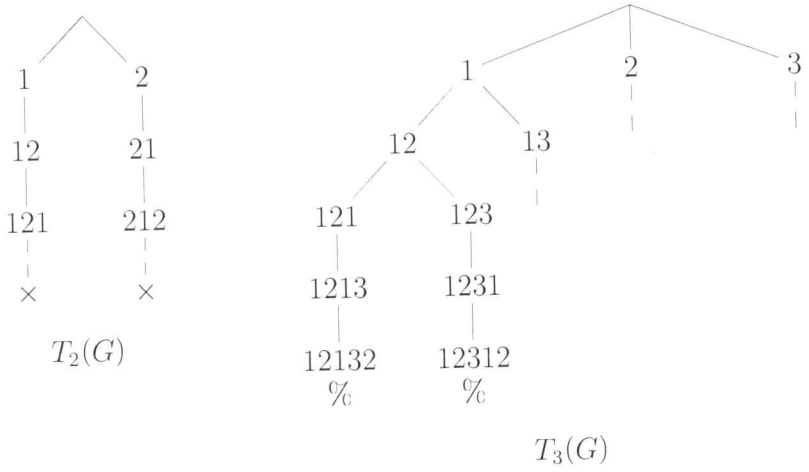
Case B. Graph Colouring Problem with a fixed number K of colours, in symbols GCP(K). It can be shown that this restriction of GCP is NP-complete, even in

the case $K = 3$. Moreover, we shall show that on average it can be solved in **constant** time deterministically! Moreover, the algorithm is trivial based on traversing through the graph.

Example 4.18. Consider the graph G:



Let us try to colour it with 2 and 3 colours as follows: Proceed systematically always using the smallest possible colour, and if a conflict (or legal colouring) is found go backwards and make the first possible colour as little larger as possible. The legal colourings can be represented in nodes of trees as follows:



Consequently, here the l'th level tells the legal colourings of l first vertices. Let us denote this subgraph of G by $H_l(G)$ and the colouring tree by $T_K(G)$. Then we have:

> The number of points of $T_K(G)$ on level l = the number of legal colourings of $H_l(G) = P(K, H_l(G))$.

We need 3 lemmas

Lemma 4.19. If $\sum_{i=1}^{K} s_i = l$, then $\sum_{i=1}^{K} s_i^2 \geq \frac{l^2}{K}$, whenever $s_i \in \mathbb{N}$ for all i.

In other words $A(n,K)$ tells the average size of $T_K(G)$, i.e. the average length of our procedure, which can be used to decide whether G is K-colourable.

$$\begin{aligned}
A(n,K) &= 2^{-\binom{n}{2}} \sum_{G_n} \sum_{l=1}^{n} \left|\{\text{points of } T_K(G) \text{ on level } l\}\right| \\
&= 2^{-\binom{n}{2}} \sum_{G_n} \sum_{l=1}^{n} P\big(K, H_l(G)\big) \\
&= 2^{-\binom{n}{2}} \sum_{l=1}^{n} \sum_{G_n} P\big(K, H_l(G)\big) \\
&= 2^{-\binom{n}{2}} \sum_{l=1}^{n} 2^{\binom{n}{2}-\binom{l}{2}} \sum_{H_l} P\big(K, H_l(G)\big) \\
&\overset{\text{L } 4.21}{\leq} \sum_{l=1}^{n} 2^{-\binom{l}{2}} K^l 2^{l^2\left(1-\frac{1}{K}\right)/2} \\
&\leq \sum_{l=1}^{\infty} K^l 2^{l/2} 2^{-\frac{l^2}{2K}} \\
&= \sum_{l=1}^{\infty} \alpha^l \beta^{-l^2} = \sum_{l=1}^{\infty} \left(\frac{\alpha}{\beta^l}\right)^l = \downarrow
\end{aligned}$$

Here α and β are constants ≥ 1, implying the convergence.

So we have concluded that there exists a constant $h(K)$ such that

$$A(n,K) \leq h(K)$$

for all $n \geq 0$. This can be formulated as

Theorem 4.22. Our above algorithm solves the problem "Is G K-colourable?" in constant time on average.

Remarks.

4.2. A more detailed analysis shows that $h(3) \leq 197$.

4.3. An intuitive explanation for Theorem 4.22 is that "most" of the graphs are not K-colourable and this "can be detected from a small initial part of the graph".

4.4. For another NP-complete set — finding the maximal **independent** subset of G (i.e. no vertices are connected) — an argumentation similar to above leads

Proof.

$$0 \leq \sum_{i=1}^{K} \left(s_i - \frac{l}{K} \right)^2 = \sum_{i=1}^{K} \left(s_i^2 - 2\frac{l s_i}{K} + \frac{l^2}{K^2} \right)$$
$$= \sum_{i=1}^{K} s_i^2 - 2\frac{l^2}{K} + \frac{l^2}{K^2} \leq \sum_{i=1}^{K} s_i^2 - 2\frac{l^2}{K} + \frac{l^2}{K} = \sum_{i=1}^{K} s_i^2 - \frac{l^2}{K}. \qquad \square$$

Lemma 4.20. Let C be an arbitrary K-colouring of l vertices. Then the number of graphs with l vertices for which C is legal is at most

$$2^{l^2\left(1-\frac{1}{K}\right)/2}$$

Proof. Assume that, for $i = 1, \ldots, K$, s_i vertices are coloured with i. Therefore $l = s_1 + \cdots + s_K$. Consider now a graph G with l vertices. If C is legal for G, then G contains at most

$$s_1 s_2 + \cdots + s_1 s_K + s_2 s_3 + \cdots \quad + s_{K-1} s_K$$

edges. Moreover,

$$\sum_{1 \leq i < j \leq K} s_i s_j = \frac{1}{2} \sum_{i \neq j} s_i s_j = \frac{1}{2} \left(\sum_{i,j=1}^{K} s_i s_j - \sum_{i=1}^{K} s_i^2 \right)$$
$$= \frac{1}{2} \left(\sum_{i=1}^{K} s_i \right)^2 - \frac{1}{2} \sum_{i=1}^{K} s_i^2 \leq \frac{1}{2} l^2 - \frac{1}{2} \frac{l^2}{K} = l^2 \left(1 - \frac{1}{K} \right) / 2.$$

Therefore the number of possible G graphs is at most $2^{l^2\left(1-\frac{1}{K}\right)/2}$. $\square$

Lemma 4.21. The number of all legal K-colourings of all graphs with l vertices is at most

$$K^l 2^{l^2\left(1-\frac{1}{K}\right)/2}.$$

Next we set

$$A(n, K) = \sum_{\substack{G \text{ has} \\ n \text{ vertices}}} |T_K(G)| / |\{\text{graphs with } n \text{ nodes}\}|.$$

to an algorithm operating in $\mathcal{O}(n^{\log n})$ steps on average, i.e. to an algorithm operating neither in polynomial nor exponential time (cf. Wilf).

Case A. In this case we just state a result without proving it, cf. Angluin, Valiant STOCS 77.

The problem is to find a Hamilton Cycle in a graph of certain type. The result states:

> "There exists an algorithm $\mathcal{A}$ satisfying: For each $\alpha \in \mathbb{R}$ there exist constants C and M such that for a randomly chosen graph G with n vertices and at least $Cn \log n$ edges the probability that $\mathcal{A}$ finds a HC in $Mn \log n$ steps (i.e. choice of a new vertex) is $1 - \mathcal{O}(n^{-\alpha})$."

Clearly, this $\mathcal{A}$ solves the problem in the sense of A: It runs in polynomial time, finds only correct answers, and does this with a high probability. The algorithm itself is not complicated — but the whole proof would be quite difficult.

Note that the above problem is not our problem HCP. The restriction to graphs with $\Omega(n \log n)$ vertices is to guarantee a high probability that the graph has a HC.

5 Complexity Classes

In this chapter we consider more closely complexity classes of page 10. We recall that these classes were defined by using a **multitape TM** as a model of a computation. We also remind of the following facts:

- we defined the time and space complexities by requiring that the machine operates on **all inputs** within this amount of resources.
- In the case of space complexities we use **input preserving** (or **off-line**) TM's, i.e. the input is never changed during the computations - this allowed **sublinear** complexities.
- Due to linear speed-up (cf. Theorem 3.14 and its variants) the complexity classes are **closed under a constant multiplication**, i.e. for example

$$\mathrm{TIME}(f) = \mathrm{TIME}(qf),$$

  whenever $q > 1$, and $f(n) \geq n + 1$, and $\lim_{n\to\infty} \frac{n}{f(n)} = 0$.
- By convention, we require that the time and space complexities T_M and S_M of a TM M satisfy

$$T_M(n) \geq n + 1 \qquad \text{and} \qquad S_M(n) \geq 1. \tag{5.1}$$

  Intuitively, the first requirement can be interpreted so that M has to read its input before deciding of the acceptance!
- We already defined "well-behaving" functions $f : \mathbb{N} \to \mathbb{N}$ on page 23. Function f is **fully space** (or **time**) **constructible** if there exists a DTM operating exactly in space (or time) $f(n)$ on all inputs of length n. Similarly, it is only **space** (or **time**) **constructible** if there exists a DTM operating in space (or time) $f(n)$ and in **exactly** that on some input of length n. We recall that such functions were used as clocks to control computations!

By the results of the previous chapter it should be clear that it is extremely difficult to show that a particular problem is not in a given complexity class, in

general. Indeed, is TSP in $\bigcup_{i\geq 1} \mathrm{TIME}(n^i)$, or even in $\mathrm{TIME}(n^i)$ for a fixed $i \geq 2$? Nobody knows the answers!

A natural question arise: Can different complexity classes be separated at all? The answer is "yes", even in quite a strong sense, as we shall show here: Even a "mild" (larger than linear) increase of a computation time or space increases the families of languages accepted or decided by TM's.

Theorem 5.1. Let $T(n)$ be a total Turing computable function. There exists a recursive language L such that $L \notin \mathrm{TIME}(T(n))$.

Proof. The proof is based on a **diagonalization argument**. In order to use it we have to **encode** all TM's (of a given type, say deterministic and with k tapes) into a binary alphabet as on page 37:

$$t \mapsto c(t) \in \{0,1\}^+,$$

and

$$M = \{t_1, \ldots, t_s\} \mapsto 0^\alpha 0c(t_1)0 \cdots 0c(t_s)00,$$

where t is now a $(2k+1)$-tuple and $\alpha \geq 0$.

It is worth noting here that $c(M)$ is not unique (even if the order of t_i's would be fixed), since α can be arbitrary. However, for each binary word there exists at most one TM which can be encoded to this word.

Now, for $x \in \{0,1\}^*$, if x is an encoding of a TM we set M_x to be that machine. Hence, the mapping

$$x \mapsto M_x$$

is well-defined partial function. We set

$$L = \left\{ x \in \{0,1\}^* \mid M_x \text{ does not accept } x \text{ in } T(|x|) \text{ steps} \right\}. \tag{5.2}$$

Note that also L is well-defined: If x is an encoding there is no problem, and if it is not an encoding, then clearly the condition in (5.2) is fulfilled, and x is in L.

We have to prove two assertions:

Claim 1. $L \in \mathrm{Rec}$.

By our assumption there exists a DTM M computing $T(n)$. We define an algorithm $\mathcal{A}$ which decides L as follows: On a given w

(i) we simulate M on input $|w|$ in order to obtain $T(|w|)$;

(ii) we test whether M_w is defined (and at the same time get it) and if not we output "yes";

(iii) we simulate M_w on input w $T(|w|)$ steps, and output "yes" if either M_w halts without accepting or does not halt in this simulation, and otherwise we output "no".

It follows that $\mathcal{A}$ is well-defined always terminating algorithm, which moreover decides L. So we only have to get convinced that $\mathcal{A}$ can be realized by a Turing Machine. This however is obvious:

- The machine M gives $T(|w|)$ the number of steps used used in the latter simulation;
- (ii) is easy to check and
- After having M_w and $T(|w|)$ (iii) becomes straightforward.

Claim 2. $L \notin \mathrm{TIME}(T(n))$.

Assume the contrary: L is decided by M' operating in time $T(n)$. Clearly, for some binary word u we have $M_u = M'$. Then

$$\begin{aligned} u \in L \quad &\Rightarrow \quad u \in L(M') = L(M_u) \\ &\Rightarrow \quad M_u \text{ accepts } u \text{ in } T(|u|) \text{ steps} \quad \Rightarrow \quad u \notin L \end{aligned}$$

and

$$\begin{aligned} u \notin L \quad &\Rightarrow \quad u \notin L(M') = L(M_u) \\ &\Rightarrow \quad M_u \text{ does not accept } u \text{ in } T(|u|) \text{ steps} \quad \Rightarrow \quad u \in L. \end{aligned}$$

Hence, we have a contradiction, so our proof of Claim 2 and Theorem 5.1 is complete. □

Corollary 5.2. There exists Turing computable functions $T_i(n)$, for $i \in \mathbb{N}$, such that

$$\mathrm{TIME}(T_i(n)) \subsetneq \mathrm{TIME}(T_{i+1}(n))$$

for all $i \in \mathbb{N}$.

Proof. We first note that whenever $T'(n) \geq T(n)$, for all $n \geq 0$, then $\mathrm{TIME}(T(n)) \subseteq \mathrm{TIME}(T'(n))$. Next if $T(n)$ is Turing computable, then by Theorem 5.1, there exists a recursive L such that $L \notin \mathrm{TIME}(T(n))$. But since L is recursive it can be decided, say, in time $\hat{T}(n)$. Now we set $T'(n) = \max\left\{T(n), \hat{T}(n)\right\}$. whence L is an example of a language in $\mathrm{TIME}(T'(n)) \setminus \mathrm{TIME}(T(n))$, i.e. $\mathrm{TIME}(T(n)) \subsetneq \mathrm{TIME}(T'(n))$.

The proof is completed by induction. □

Remarks.

5.1. The proof of Theorem 5.1 can be straightforwardly modified to space and nondeterministic complexity classes.

5.2. In Theorem 5.1 the complexity of L might be much larger than $T(n)$. This, however, is not necessary, as we show next.

We consider first space complexity results, and start with the following useful lemma.

Lemma 5.3. Let $S(n) \geq \log_2(n)$ for all $n \geq 1$. If L is accepted by a TM operating in space $S(n)$, then it is also accepted by an always terminating TM operating in the same space $S(n)$.

Proof. Let $L = L(M)$. We formulate the proof only for the case when M contains just one work tape (the general case being an obvious modification). Let M contain s states and t symbols.

The crucial point here is the equivalence

$$\begin{array}{c} M \text{ accepts } w \\ \Updownarrow \\ M \text{ accepts } w \text{ in at most } (n+2) \cdot s \cdot S(n) \cdot t^{S(n)} \text{ steps,} \end{array} \tag{5.3}$$

where $n = |w|$. This is because the number here gives an upper bound for different configurations, and each accepted word possesses a repetition-free computation.

Next we note that

$$(n+2)sS(n)t^{S(n)} \leq n^2 s^{S(n)} t^{S(n)} \leq (4st)^{S(n)} \tag{5.4}$$

for $n \geq 1$, $s \geq 2$ and hence for all s and t (by artificially largening s). Note that $S(n) \geq \log_2 n$.

Now we construct M' as follows. It simulates M and at the same time computes the length of the computation on a new tape. Further M' halts accepting, if M accepts, and halts rejecting if M halts and do not accept **or** M' repeats a configuration. The latter property is checked, by (5.3), via the length of the computation.

More precisely: M' uses a new tape as a counter. At the beginning it marks $\lceil \log_2 n \rceil$ cells on that tape and counts at **base 4** starting from zero. After each simulation step of M the machine M' adds one to the counter. Moreover, if the space used by M is larger than that so far marked on the new tape, M' marks a new symbol on that tape. Since M operates in space $S(n)$, the number of marked cells does not exceed $S(n)$.

If M halts then does M' and behaves in the same way. If the counter of M' exceeds the marked space, which of course happens eventually if M did not halt before that, then we claim that M has already repeated a configuration, so that M' can be put into a nonaccepting state. This indeed follows from (5.3) and (5.4), and the corresponding formulas where the space used by M is $i \in [\log_2 n, S(n)]$ instead of $S(n)$. □

Now we are ready to sharpen Theorem 5.1 for space classes.

Theorem 5.4. Let $S_1(n), S_2(n) \geq \log_2 n$, $S_2(n)$ fully space constructible and

$$\liminf_{n\to\infty} \frac{S_1(n)}{S_2(n)} = 0. \tag{5.5}$$

Then there exists a language in $\mathrm{SPACE}(S_2(n)) \setminus \mathrm{SPACE}(S_1(n))$.

Proof. We note first that any k-tape off-line TM operating in space $S(n)$ can be simulated by 1-tape off-line TM operating in the same space: simply consider the k tapes as 1 tape having k tracks, and simulating the moves by travelling back and forth on that.

So it is enough to consider TM's with only one work tape. Consider such machines having the **input** alphabet $\{0, 1\}$ (and symbols $\triangleright$ and $*$). We encode these machines as in the proof of Theorem 5.1. Now, unlike there, it is important that each machine has **arbitrarily long encodings** — hence the use of α! Recall the partial function $w \mapsto M_w$.

Our goal is to construct a TM M of the above type which

- operates in space $S_2(n)$ and
- behaves differently than any such machine operating in space $S_1(n)$ on some input.

Of course, that proves the theorem.

The machine is required to work as follows:

(i) On input w it first marks $S_2(|w|)$ cells on the worktape;

(ii) If M tries to use more space, it halts without accepting;

(iii) M simulates M_w (if such a machine exists) on input w;

(iv) M accepts if the simulation succeeds in the marked cells, i.e. in space $S_2(|w|)$, and M_w halts without accepting; otherwise M rejects.

Again we have to get convinced that such a TM exists. Since $S_2(n)$ is fully space constructible M indeed can perform (i) at the beginning of the computation. Also (ii), after (i), is easy to maintain, as well as (iv). The problematic case is (iii). First M checks that w is an encoding of a required TM, i.e. M_w is defined. Surely this can be done in space $\log_2 n \leq S_2(n)$. If this is not the case M can behave arbitrarily on w. So assume that M_w is defined, when its description is given on the input tape of M. Now if M_w uses t letters on its work tape, the contents of the worktape of M can be expressed as a word of length

$$\lceil \log_2 t \rceil \cdot S_{M_w}(n)$$

on the worktape of M. Hence the simulation of M_w by M succeeds in space $\lceil \log_2 t \rceil \cdot S_{M_w}(n)$: M simply reads the transition from its input tape and changes the work tape correspondingly.

We have to show two assertions, the first one being clear:

Claim 1. $L(M) \in \mathrm{SPACE}(S_2(n))$.

Claim 2. $L(M) \notin \mathrm{SPACE}(S_1(n))$.

Assume the contrary $L(M) = L(\hat{M})$, where $\hat{M}$ operates in space $S_1(n)$ and uses t symbols on its work tape. Further, by lemma 5.3, we may assume that $\hat{M}$ is always terminating. Now, we use a special property of our enconding, namely that we can choose it arbitrarily long, for example we can require, by (5.5), that the encoding $\hat{w}$ of $\hat{M}$ satisfies

$$\lceil \log_2 t \rceil \cdot S_1(|\hat{w}|) < S_2(|\hat{w}|).$$

But then, as we saw, M can simulate $\hat{M}$ on input $\hat{w}$. Therefore by (iv) $L(M) \neq L(M_{\hat{w}}) = L(\hat{M})$, a contradiction. □

Next we modify Theorem 5.4 for time complexity.

Theorem 5.5. Let $T_2(n)$ be fully time constructible and

$$\liminf_{n\to\infty} \frac{(T_1(n))^2}{T_2(n)} = 0. \tag{5.6}$$

Then there exists a language L in $\mathrm{TIME}(T_2(n)) \setminus \mathrm{TIME}(T_1(n))$.

Proof. This is a suitable adaptation of the proof of Theorem 5.4. We construct a DTM (with 4 tapes) which

(i) operates in time $T_2(n)$;

(ii) simulates arbitrary TM M (with any number of tapes) encoded as M_w on input w and

(iii) accepts iff the simulation succeeds in time $T_2(|w|)$ and M_w terminates without accepting (or if M_w is undefined outputs anything).

Note that here we have to encode all DTM's with any number of tapes since the complexity classes are defined in that way (In Theorem 5.4 this was not necessary, since k tapes could be simulated by 1 tape in the same space!). This makes the encoding a bit more complicated, but still would not cause any serious problems.

To ensure (i) for M, it simulates **simultaneously** a DTM M_t operating exactly in time $T_2(n)$ and the considered M_w. This is done in different tapes, so M first copies the input into a new tape, and then runs on that the computation of M_t, while

on the other tape M simulates M_w, using the first tape to remember the encoding of M_w. Therefore M actually operates in time $T_2(n)+\mathcal{O}(n)$, i.e. we shall conclude that a language $L = L(M)$ is not in $\mathrm{TIME}(T_1(n))$. But by the linear speed-up (Theorem 3.14) it is in $\mathrm{TIME}(T_2(n))$, as was to be shown.

Again conditions (i) and (iii) are easy: The constructed M can be required to fulfill those. Note also that it **is not necessary to check** that M_w is defined since, if it is not, the output of M on w can be arbitrary! This is needed here! Condition (ii) requires some comments. Consider a machine M_w **operating in time** $T_1(n)$. It may contain **arbitrary many** tapes and arbitrary many tape symbols, say k and t, respectively. On the other hand, M must be a **fixed** DTM. There are two important observations:

First, by Theorem 3.9, the k-tapes of M_w can be simulated using only 1 tape of M in time $\mathcal{O}\big(T_1(n)^2\big)$, i.e. each step in $\mathcal{O}(T_1(n))$ steps. This, however, required that the transitions of M_w were directly available. Now, they have to be searched from the first tape. This can be done in time $\mathcal{O}(|w|) = \mathcal{O}\big(T_1(|w|)\big)$ by using a fourth tape to remember a current state and symbol. Hence, the above time suffices also here!

Second, since M is fixed, it cannot use directly symbols of M_w: they have to be encoded into an alphabet of M. We can use directly the encoding used in w, which means that each symbol of M_w requires in M t symbols, i.e. a constant times more.

It follows that M can indeed simulate M_w in time $\mathcal{O}\big(T_1(|w|)^2\big)$, i.e. in time $cT_1(|w|)^2$ for some constant c depending on M_w.

Next we proceed as at the end of Theorem 5.4. Let M_1 be an arbitrary DTM operating in time $T_1(n)$, and c_1 the above constant associated to M_1. We choose an encoding w_1 of M_1 such that

$$c_1 T_1(|w_1|)^2 < T_2(|w_1|).$$

By (5.6) and the fact that each TM has arbitrarily long encodings this is possible. Then

– M succeeds to simulate $M_1 = M_{w_1}$ in time $T_2(|w_1|)$ on input w_1 and

– $w_1 \in L(M)$ iff $w_1 \notin L(M_w)$.

It follows that $L(M)$ is different from $L(M')$, where M' is any DTM operating in time $T_1(n)$.

This proofs the theorem. □

Remark 5.3. Theorem 5.5 can be proved in a much stronger form, namely replacing (5.6) by

$$\liminf_{n\to\infty} \frac{T_1(n)\log T_1(n)}{T_2(n)} = 0. \tag{5.7}$$

The proof is essentially as above except that the k tapes of M_w are simulated using two tapes of M. Such a simulation can be performed in time $T(n)\log(T(n))$ — the proof is not very easy, cf. Hopcroft-Ullman.

Example 5.6. Theorems 5.4 and 5.5 allows to conclude, for example,

$$\mathrm{SPACE}(n) \subsetneq \mathrm{SPACE}(n\log n)$$

and

$$\mathrm{TIME}(n) \subsetneq \mathrm{TIME}(n^2\log n) \subseteq \mathrm{TIME}(n^3)$$

but not

$$\mathrm{TIME}(n) \subsetneq \mathrm{TIME}(n\log n).$$

Note also that a sharper version of Theorem 5.5 using (5.7) allows to conclude:

$$\mathrm{TIME}(n) \subsetneq \mathrm{TIME}(n\log^2 n)$$

and

$$\mathrm{TIME}(2^n) \subsetneq \mathrm{TIME}(n^2 2^n).$$

So far we have compared deterministic complexity classes of the same type. Next we point out some simple connections of complexity classes of different types.

Theorem 5.7. The following statements hold:

(i) If $L \in \mathrm{TIME}(f(n))$, then $L \in \mathrm{SPACE}(f(n))$;

(ii) If $L \in \mathrm{SPACE}(f(n))$ and $f(n) \geq \log_2 n$, then there exists a constant c such that $L \in \mathrm{TIME}(c^{f(n)})$ and

(iii) If $L \in \mathrm{NTIME}(f(n))$, then there exists a constant c such that $L \in \mathrm{TIME}(c^{f(n)})$.

Proof. (i) Recall that we assume here that $f(n) \geq n + 1$. Then the result is also formally very clear: It is proved as Theorem 3.10 (where only two first symbols are stored into one cell).

(ii) Assume that L is decided by a DTM M with s states and t symbols, and operating in space $f(n)$. As we have seen we may assume that M is 1 tape machine. Also as in the proof of Lemma 5.3 we see that the number of different configurations on computations in space $f(n)$ is at most

$$s(n+2)f(n)t^{f(n)},$$

which by the assumption $f(n) \geq \log_2 n$ satisfies

$$s(n+2)f(n)t^{f(n)} \leq d^{f(n)}$$

for $n \geq 1$, where d is a constant depending only on M.

Now we construct a TM M' such that

- M' simulates M in such a way that after each step
- M' tests whether a new configuration already occurred earlier, and if no stores it into a special tape and
- if M' recognizes a repetition it rejects; otherwise it accepts if and only if M accepts.

It is obvious that M' decides/accepts the same language as M. So it remains to estimate the complexity of M'.

A configuration of M can be stored in a tape of M' as a word of length at most

$$\log_2 n + 1 + f(n),$$

where the term $\log_2 n$ comes from the binary representation of the position of the head of M on its input tape (and if necessary a few symbols are stored in the same cell). So to generate and store such a configuration

$$\mathcal{O}(\log n + f(n))$$

steps are enough. Whether a new configuration needs to be stored can be checked by comparing it to all old ones, that is, it can be done in time

$$\mathcal{O}\big(d^{f(n)}(\log n + f(n))\big).$$

Finally, since the total number of different configurations is at most $d^{f(n)}$, M' runs in time

$$\mathcal{O}\big(d^{f(n)}(\log n + f(n))d^{f(n)}\big) \leq Ac^{f(n)}$$

for some constants A and c. So (ii) follows from linear speed-up.

(iii) In Theorem 3.11 we showed how to simulate a 1 tape NTM by a 3 tape DTM in time $\mathcal{O}(c^{f(n)})$. Now, we have to simulate a k tape NTM by a deterministic one. But by the construction of Theorem 3.11 this can be done by $2k+1$ tape machine in time $\mathcal{O}(c^{f(n)})$ proving the case (iii). □

Remark 5.4. Note that the case (ii) of Theorem 5.7 extends to nondeterministic space complexity classes (assuming that $f(n) \geq \log_2 n$) as follows:

$$L \in \mathrm{NSPACE}(f(n)) \implies L \in \mathrm{TIME}\big(c^{f(n)}\big) \quad \text{for some } c.$$

Next we prove two fundamental results on complexity classes, namely **Savitch's Theorem** and **Szelepcsényi-Immerman Theorem**.

Theorem 5.8 (Savitch, 1970). If $S(n)$ is fully space constructible and $S(n) \geq \log_2 n$, then

$$\mathrm{NSPACE}(S(n)) \subseteq \mathrm{SPACE}\big(S(n)^2\big).$$

Proof. The proof uses efficiently the fact that, contrary to time, **space can be used again and again**.

Let L be accepted by NTM M operating in space $S(n)$. As we have seen M can be assumed to be one tape machine, and the number of its different configurations is bounded by $c^{S(n)}$, where c is a constant depending only on M. Then we have

$$w \in L(M) \iff w \text{ is accepted in time } c^{S(|w|)}.$$

For two configurations α_1 and α_2 of M we denote

$$\alpha_1 \xrightarrow{(i)} \alpha_2$$

iff α_1 derives α_2 according to M in **at most** 2^i steps. Then we clearly have

$$\alpha_1 \xrightarrow{(i)} \alpha_2 \tag{5.8}$$

$$\Updownarrow$$

$$\exists \alpha' : \alpha_1 \xrightarrow{(i-1)} \alpha' \quad \text{and} \quad \alpha' \xrightarrow{(i-1)} \alpha_2. \tag{5.9}$$

Consequently, to test (5.8), it is enough to test two conditions of (5.9) for all α', and the space needed for that is the space used by α' and the space needed in tests of (5.9). The latter, however, can be done in a common space!

The above leads to the following trivial algorithm:

$\mathcal{ALG}$:

1. Set $n = |w|$ and $m = \lceil \log_2 c \rceil$,
2. Set α_0 to be the initial configuration of M on w,
3. For each accepting configuration α_f of length at most $S(n)$ do:

   $\mathcal{TEST}(\alpha_0, \alpha_f, mS(n))$ and if it succeeds, then accept w,
4. Reject w.

Here the subalgorithm $\mathcal{TEST}$ is defined as follows:

$\mathcal{TEST}(\alpha_1, \alpha_2, i)$:

1. For $i = 0$ check whether $\alpha_1 = \alpha_2$ or $\alpha_1 \to \alpha_2$ and if so return "yes";

2. For $i > 0$ do the following for all configurations α' of length $\leq S(n)$:

   $\mathcal{TEST}(\alpha_1, \alpha', i-1)$ and $\mathcal{TEST}(\alpha', \alpha_2, i-1)$ and if they succeed return "yes";

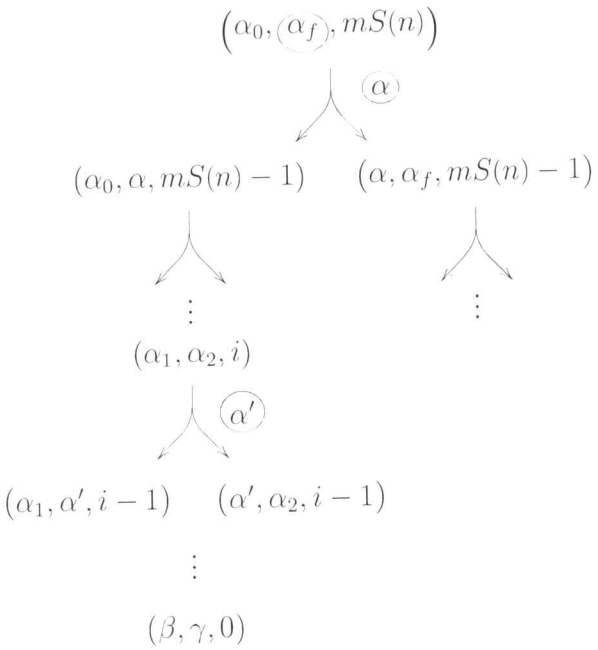
3. Return "no".

It is obvious that $\mathcal{ALG}$ works correctly: It checks through all the possibilities and accepts an input if it finds an accepting computation for it. Note also that the depth of the recursion is correctly chosen, since

$$m \cdot S(n) \geq \log_2 \left(c^{S(n)}\right)$$

and "the halvening of the number of computation steps can be done at most $\log_2$ times."

The above recursive algorithm can be illustrated as a tree as follows:



(5.10)

Here each node denotes a corresponding test, and the cycled elements go through all possibilities!

It remains to be shown how $\mathcal{ALG}$ can be realized by a deterministic TM in the required space $(S(n))^2$.

Note that the above tree contains $2^{mS(n)}$ nodes so that it cannot be remembered in the required space. But fortunately, it is enough to remember only very small

portion of it, namely essentially one path, in order to do all the required checkings! This is the idea making this proof possible.

We construct the required deterministic TM M' as follows. To guarantee that M' can do the test for **all** configurations of length $\leq S(n)$ we order these lexicographically. Then for each configuration α (except the "last" one) there exists the next configuration $s(\alpha)$, and moreover we can assume that M' can compute it from α. Note also that since $S(n)$ was fully space constructible, M' can mark at the beginning the space $S(n)$.

When performing the tests of $\mathcal{ALG}$ M' remembers some quadruples

$$(\alpha_1, \alpha_2, i) \quad \text{and} \quad \alpha', \tag{5.11}$$

as well as whether the left or right branch of the tree is chosen of this particular point. To remember configurations $S(n)+\log_2(n+2)$ memory cells are enough (note that M is input preserving machine!) and to remember i surely $mS(n)$ memory cells is enough. So the whole information (5.11) can be stored in space $\mathcal{O}(S(n))$.

A problem is that in the above tree there is too many nodes to be remembered. This can be overcome by the following

Claim. M' can realize $\mathcal{ALG}$ by remembering at any moment only quadruples of (5.11) from one path of the tree (5.10).

Assuming that the claim is proved, Theorem 5.8 follows immediately: M operates in space $mS(n) \cdot \mathcal{O}(S(n)) = \mathcal{O}\big(S(n)^2\big)$, so that by linear speed-up we can construct another deterministic TM operating in space $S(n)^2$.

Proof of Claim. Let us consider one branching of the tree:

$$\begin{array}{ccc} & (\alpha_1, \alpha_2, i) & \\ & \swarrow \quad \searrow \ \alpha & \\ (\alpha_1, \alpha, i-1) & & (\alpha, \alpha_2, i-1) \end{array} \tag{5.12}$$

The structure of $\mathcal{ALG}$ is such that from the "yes" answers of the daughter nodes $\mathcal{ALG}$ computes "yes" for the node (α_1, α_2, i), and moreover w is accepted if the root of the tree gets the "yes" value for some choice of α's and α_f's.

On the other hand if $\mathcal{ALG}$ gives "no" for some of the daughter nodes, then either α has to be replaced by $s(\alpha)$, or if this is not possible then the node itself gets the value "no".

Now, to conclude the claim we note that M' can do the following computations in terms of (5.12):

(i) $(\alpha_1, \alpha_2, i),\ \alpha \mapsto (\alpha_1, s(\alpha), i-1)$ with the orientation left and preserving (α_1, α_2, i);

(ii) $(\alpha_1, \alpha_2, i),\ (\alpha_1, \alpha, i-1) \mapsto (\alpha, \alpha_2, i-1)$ with the orientation right and preserving (α_1, α_2, i) but deleting $(\alpha_1, \alpha, i-1)$.

These operations corresponds to the cases: **create** a new edge to a path and **replace** the last left edge by the corresponding right one.

All in all M' operates as follows:

- It chooses all possible α_f's in lexicographic order;
- For a given α_f it generates (using (i)) the leftmost path of (5.12) with minimal values of α's;
- Now the last triple $(\beta, \gamma, 0)$ assumes the value "no" or "yes".
- If this is "yes" M' performs (ii) to obtain a value for the corresponding right daughter node, and if also this is "yes", then also the mother node of this assumes the value "yes", and the daughters are removed (and procedure continues in the same way);
- If there is "no" in either of the above stages, then the daughter nodes are removed, the α of the mother node is increased by one and procedure continues by using (i); and if $s(\alpha)$ is not defined, then the mother node gets the value "no" (and the procedure is continued).

Clearly, the above recursive procedure works as intended, i.e. M' has to remember at any stage at most one complete path from the root to some leaf in (5.10). □

As a corollary we obtain a result which shows that the fundamental open problem $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ has no counterpart for space complexity classes:

Corollary 5.9. PSPACE = NPSPACE.

To complete our considerations on hierarchy results we consider still a bit more nondeterministic complexity classes. Actually we present here only results on such space complexity classes.

We start with a so-called **Translation Lemma**.

Lemma 5.10. Let $S_1(n)$, $S_2(n)$ and $f(n)$ be fully space constructible, and moreover $S_2(n) \geq n$ and $f(n) \geq n$ for all n. Then we have

$$\begin{gathered} \mathrm{NSPACE}(S_1(n)) \subseteq \mathrm{NSPACE}(S_2(n)) \\ \Downarrow \\ \mathrm{NSPACE}\big(S_1(f(n))\big) \subseteq \mathrm{NSPACE}\big(S_2(f(n))\big). \end{gathered}$$

Proof. Let M_1 be a TM accepting L_1 and operating in space $S_1(f(n))$. Define

$$L_2 = \big\{ w\$^i \mid M_1 \text{ accepts } w \text{ in space } S_1(|w|+i) \big\},$$

where \$ is a new symbol.

We construct a TM M_2 accepting L_2 as follows: After getting an input $w\$^i$ M_2 marks on its work tape $S_1(|w|+i)$ cells, and then simulates M_1 on its input w, and M_2 accepts if the simulation succeeds in the marked space. Such a M_2 exists since S_1 is fully space constructible, and moreover M_2 operates in space $S_1(n)$.

It follows from our assumptions that there exists a TM M_3 accepting L_2 and operating in space $S_2(n)$.

Using above we construct a TM M_4 accepting the original language L_1 in space $S_2(f(n))$. First, M_4 marks $S_2(f(n))$ cells on its tape. This is possible since S_2 and f, and therefore also their composition $S_2 \circ f$, are fully space constructible, and since $S_2(n) \geq n$ so that $S_2(f(n)) \geq f(n)$ guarantering that the marking of $f(n)$ cells can be done in space $S_2(f(n))$. Second, M_4 on input w operates as M_3 on inputs $w\$^i$ for $i = 0, 1, \ldots$ (Remember that M_4 is nondeterministic!). In order to do that (for a fixed i) M_4 has to know the position of the head of M_3 on the input tape: If it is within w there is no problem — the head of M_4 is in the same place. If, in turn, it is within \$-symbols, then M_4 remembers it as a binary number on a new tape. For $i \leq f(|w|) - |w|$ this requires

$$\log_2(i) \leq \log_2\big(f(|w|) - |w|\big) \leq f(|w|) \leq S_2\big(f(|w|)\big) \tag{5.13}$$

cells, i.e. can be done within space used by M_4.

In details the above simulation of M_3 by M_4 works as follows:

- If M_3 accepts so does M_4;
- If M_3 does not accept, M_4 initiates a new simulation with the next value of i;
- If the new tape of M_4 remembering the position of the head of M_3 tries to use more space than $S_2(f(n))$, M_4 halts without accepting.

Clearly, such an M_4 exists when we assume, by Lemma 5.3, that M_3 is always terminating. By the construction M_4 operates in space $S_2(f(n))$, so that it remains to be proved:

Claim. $w \in L(M_1) \iff w \in L(M_4)$.

First if $w \in L(M_1)$, then it is accepted by M_1 in space

$$S_1\big(f(|w|)\big) = S_1\big(|w| + f(|w|) - |w|\big).$$

Therefore

$$w\$^{f(|w|)-|w|} \in L_2 = L(M_3),$$

so that by (5.13) and the construction of M_4 the word $w \in L(M_4)$.

Second, if $w \in L(M_4)$, then by the construction of M_4,

$$w\$^i \in L(M_3) = L_2$$

for some i. This means that w is accepted by M_1 (in space $S_1(|w| + i)$), meaning that $w \in L(M_1)$. □

Example 5.11. We show that $\mathrm{NSPACE}(n^3) \subsetneq \mathrm{NSPACE}(n^4)$. Indeed, if this is not the case we would have $\mathrm{NSPACE}(n^4) \subseteq \mathrm{NSPACE}(n^3)$ and further:

$$\begin{aligned}
\mathrm{NSPACE}(n^{20}) &\subseteq \mathrm{NSPACE}(n^{15}) && \text{Lemma 5.10, } f(n) = n^5\\
&\subseteq \mathrm{NSPACE}(n^{16})\\
&\subseteq \mathrm{NSPACE}(n^{12}) && \text{Lemma 5.10, } f(n) = n^4\\
&\subseteq \mathrm{NSPACE}(n^{9}) && \text{Lemma 5.10, } f(n) = n^3\\
&\subseteq \mathrm{SPACE}(n^{18}) && \text{Savitch}\\
&\subsetneq \mathrm{SPACE}(n^{20}) && \text{Theorem 5.4}\\
&\subseteq \mathrm{NSPACE}(n^{20}).
\end{aligned}$$

The above example generalizes as follows:

Theorem 5.12. Let $\epsilon > 0$ and $r \geq 1$. Then we have

$$\mathrm{NSPACE}(n^r) \subsetneq \mathrm{NSPACE}(n^{r+\epsilon}).$$

Proof. Take natural numbers s and t such that

$$r \leq \frac{s}{t} < \frac{s+1}{t} \leq r + \epsilon.$$

We show that

$$\mathrm{NSPACE}(n^{s/t}) \subsetneq \mathrm{NSPACE}(n^{(s+1)/t}),$$

from which the claim follows. Assume the contrary:

$$\mathrm{NSPACE}(n^{(s+1)/t}) \subseteq \mathrm{NSPACE}(n^{s/t}).$$

By choosing $f(n) = n^{(s+i)t}$, for $i = 1, \ldots, s$, in Translation Lemma we obtain

$$\mathrm{NSPACE}(n^{(s+1)(s+i)}) \subseteq \mathrm{NSPACE}(n^{s(s+i)}) \tag{5.14}$$

for $i = 1, \ldots, s$. For the considered values of i, $s(s+i) \leq (s+1)(s+i-1)$ so that

$$\mathrm{NSPACE}(n^{s(s+i)}) \subseteq \mathrm{NSPACE}(n^{(s+1)(s+i-1)}). \tag{5.15}$$

Now, applying the above formulas alternatively we derive

$$\begin{aligned}\mathrm{NSPACE}(n^{(s+1)2s}) &\overset{(5.14)}{\subseteq} \mathrm{NSPACE}(n^{s2s}) \overset{(5.15)}{\subseteq} \mathrm{NSPACE}(n^{(s+1)(2s-1)}) \\ &\overset{(5.14)}{\subseteq} \mathrm{NSPACE}(n^{s(2s-1)}) \subseteq \cdots \subseteq \mathrm{NSPACE}(n^{(s+1)s}) \\ &\subseteq \mathrm{NSPACE}(n^{s^2}).\end{aligned}$$

This yields the inclusions:

$$\begin{aligned}\mathrm{NSPACE}(n^{2s^2+2s}) &\subseteq \mathrm{NSPACE}(n^{s^2}) && \text{above} \\ &\subseteq \mathrm{SPACE}(n^{2s^2}) && \text{Savitch} \\ &\subsetneq \mathrm{SPACE}(n^{2s^2+2s}) && \text{Theorem 5.4} \\ &\subseteq \mathrm{NSPACE}(n^{2s^2+2s}).\end{aligned}$$

This, however, is impossible. □

Remark 5.5. The above proof assumes that fractional powers $n^{s/t}$ are fully space constructible, cf. Exercises. Similar results for time classes are more difficult.

Example 5.13. We are in the position to reconsider the fundamental open problems of the complexity theory. We have:

$$\text{LOGSPACE} \subseteq \text{PTIME} \subseteq \text{NPTIME} \subseteq \text{NPSPACE} = \text{PSPACE}$$
$$\subsetneq$$

The inclusions come from Theorems 5.7 and 3.10, the equality from Savitch's Theorem and the proper inclusion from Theorem 5.4. Note also that inside the above hierarchy there exists two infinite hierarchies, namely

$$\begin{array}{ccccc} & & \text{SPACE}(\log n) \subsetneq \text{SPACE}(\log^2 n) \subsetneq \cdots \subsetneq \text{SPACE}(\log^k n) \subsetneq \cdots & & \\ & \nearrow & & \searrow & \\ \text{LOGSPACE} & & & & \text{PSPACE} \\ & \searrow & & \nearrow & \\ & & \text{NSPACE}(\log n) \subseteq \text{NSPACE}(\log^2 n) \subseteq \cdots \subseteq \text{NSPACE}(\log^k n) \subseteq \cdots & & \end{array}$$

The upper chain is proper at any place, by Theorem 5.4, and also the lower one is infinite, due to Savitch's Theorem.

Connected to the above example, and as an evidence how difficult it is to compare time and space complexity classes we give the following surprising example.

Example 5.14. The following inequalities hold:

$$\text{PTIME} \neq \text{SPACE}(n)$$
$$\text{PTIME} \neq \text{SPACE}(\log^k n)$$

for $k \geq 2$, as well as the same results where PTIME is replaced by NPTIME. However, it **is not known which of the three alternatives holds true**!! The proof of these (and many other similar ones due to the proof method) is based on the space hierarchy theorem and the idea used in the proof of Transition Lemma (Lemma 5.10). The details are left as an exercise.

Similar conclusions can be drawn for nondeterministic space classes, for example

$$(\text{N})\text{PTIME} \neq \text{NSPACE}(\log^k n)$$

for $k \geq 2$, and again it is not known which of the possibilities holds.

Next we turn to consider one of the fundamental complexity results, so-called **Szelepcsényi-Immerman Theorem**, which states that nondeterministic space complexity classes, with $S(n) \geq \log_2 n$ and fully space constructible, are closed under complementation. Surprisingly this result was proved only in 1988! Note also that the deterministic space complexity classes are trivially closed under complementation: Change the states "yes" and "no".

In order to prove the theorem we first consider a particular problem. Namely the **reachability problem** in graphs:

Example 5.15. GRP (Graph Reachability Problem)

Input: (Un)directed graph G with n vertices and two vertices u and v.

Output: "Yes" if u and v are connected in G.

As we noted in Exercise IV-5 (and is easy to see) this problem is in PTIME.

Here we are more interested in its space complexities. To simplify notation we assume that vertices are numbers $1, \ldots, n$, and further that $u = 1$ and $v = n$.

Deterministic space: A trivial method of constructing a list of all reachable vertices yields a TM operating in linear space:

$$L_0 = \{1\}$$
$$L_{i+1} = L_i \cup \left\{ j \;\middle|\; k \text{ —— } j \text{ is in } G \text{ with } k \in L_i \right\}.$$

By Savitch's Theorem (Theorem 5.8) and our considerations below, we obtain a TM solving GRP in space $\log^2 n$ (Note that n here is even smaller than the size of the input). This is the best space complexity known for this problem.

Nondeterministic space: We claim that space $\log_2 n$ is enough here. A NTM M with 3 tapes accepting the "yes" instances is described as follows:

- As usual tape 1 contains an encoded instance of the problem;
- Tape 2 remembers a vertex v_1 (which at the beginning is 1);
- Tape 3 chooses an arbitrary vertex v_2 (nondeterministically);
- Now, M tests whether v_1 —— $v_2 \in G$ and if

- "Yes", then if $v_2 = v$ outputs "Yes" and otherwise replaces v_1 by v_2 and continues as on the 3rd line
- "No", terminates without accepting.

Clearly, M operates in space $\log_2 n$, and works correctly.

Actually, we shall need a slight generalization of the graph reachability problem introduced as follows:

Example 5.16. $\mathrm{GRP_C}$ (Graph Reachability Problem **with Counting**)

Input: As in GRP.

Output: The number of vertices reachable from $u = 1$.

Our deterministic TM of Example 5.15 can be easily transformed to TM solving $\mathrm{GRP_C}$ in linear space. The fact that $\mathrm{GRP_C}$ can be computed nondeterministically in space $\log n$ is an essential part of S-I-Theorem.

In order to make the last sentence of Example 5.16 meaningful we have to define how to compute a function by a nondeterministic Turing Machine. We say that a **NTM M computes a function** f if

- All computations on a given input w either yield an output $f(w)$ or "No" (the latter meaning that M goes to the state no);
- There exists a computation on a given input w yielding an output $f(w)$.

Note that this definition is stated for total functions only (since we need only such). Notice also that if a DTM computes a total function, it computes it as a NTM as well.

As the final prerequisities for S-I-Theorem we point out a connection of GRP and computations in TM's:

Let M be a (not necessarily deterministic) TM operating in space $S(n) \geq \log_2 n$. As we concluded, e.g. in the proof of Theorem 5.7, the number of different configurations of M on an input of length n is bounded by $d^{S(n)}$, where d is a constant depending only on M.

Now let $G(M,w)$ be the **(directed) computation graph** of M on w defined as: For two configurations p and q of M of length at most $d^{S(|w|)}$

$$p \longrightarrow q \in G(M,w) \iff p \to_M q.$$

Further we add to $G(M,w)$ a special vertex "yes" which is connected to all accepting configurations. Then denoting by q_0 the initial configuration of M we have:

$$w \in L(M) \iff \text{"yes" is reachable from } q_0 \text{ in } G(M,w).$$

This means that we have translated the **membership** problem of a language ($w \in L(M)$?) into the reachability problem for graphs!

Recalling that

$$\big|G(M,w)\big| \leq d^{S(|w|)}$$

we can interprete observations of Example 5.15 as our earlier proved simulation results:

Since GRP $\in$ PTIME, the problem "$w \in L(M)$?" can be decided in time

$$\big(d^{S(|w|)}\big)^{\alpha} = d'^{S(|w|)}. \qquad \text{(cf. Theorem 5.7 (iii))}$$

Since GRP $\in$ NSPACE($\log n$), $L(M)$ is accepted by a NTM in space

$$\log\big(d^{S(|w|)}\big) = \mathcal{O}\big(S(|w|)\big). \qquad \text{(trivial)}$$

Since GRP $\in$ SPACE$\big(\log^2 n\big)$, $L(M)$ is decided by a DTM in space

$$\log^2\big(d^{S(|w|)}\big) = \mathcal{O}\big(S(|w|)^2\big). \qquad \text{(Savitch)}$$

Now we are ready for a crucial step in S-I-Theorem.

Theorem 5.17. The graph reachability problem with counting, i.e. $\mathrm{GRP_C}$, can be computed in space $\log_2 n$ by a nondeterministic TM.

Proof. Assume that G contains n vertices and w is the given initial vertex. We construct an algorithm consisting of **four** nested loops which computes the number of vertices reachable from w in G. First we describe the algorithm, then conclude that it can be realized by a NTM in space $\log_2 n$, and finally prove its correctness.

The algorithm: Let $S(i)$, for $i = 0, \ldots, n-1$, denote the set of vertices of G which can be reached from w by paths of lengths at most i. Then, clearly, $|S(n-1)|$ is the required value we are computing. We use the **first** loop to compute iteratively the values $|S(0)|, |S(1)|, \ldots, |S(n-1)|$:

$$\begin{aligned} &|S(0)| \leftarrow 1, \\ &\text{for } k = 1, \ldots, n-1 \text{ do: compute } |S(k)| \text{ from } |S(k-1)|. \end{aligned} \tag{5.16}$$

In order to compute (5.16) we use the **second** loop, where the value of the variable l is computed. This is done by testing a condition for all vertices of G in a lexicographic order, and at the end the value of l will give that of $S(k)$:

$$\begin{aligned} &l \leftarrow 0, \\ &\text{for each } u = 1, \ldots, n, \text{ if } \boldsymbol{u \in S(k)} \text{ then } l \leftarrow l + 1. \end{aligned} \tag{5.17}$$

It is obvious that l computes $|S(k)|$ correctly; however we are not defined how the test "$u \in S(k)$" is performed. This is done in the **third** loop. Here again we introduce a new variable v which runs through all vertices of G in a lexicographic order, and we test whether "$v \in S(k-1)$". If the answer to this test (actually done in the fourth loop) is affirmative we increase the value of a new counter m computing elements in $S(k-1)$ by one. Moreover, in this case we do another test namely check whether $u = v$ or $v \longrightarrow u \in G$ (which due to the information in the input can be done), and if the answer to this is affirmative we write $G(v, u)$. After these two "yes" answers we know that $u \in S(k)$, which is formalized by giving a new variable $\mathrm{inc}(l)$ value "true". If after all choices of v, we have not confirmed $u \in S(k)$, we report the converse, i.e. $\mathrm{inc}(l) \leftarrow$ "false".

Now a crucial point of this third loop is as follows: **We return inc(l) only if we get an affirmative answer to the test $m \stackrel{?}{=} |S(k-1)|$ at the end of this loop** — this means that we have succeeded in the test of the coming fourth loop maximal number of times.

The whole loop number three is formally as follows:

$$\begin{aligned} &m \leftarrow 0 \text{ and } \mathrm{incl}(l) \leftarrow \text{"false"}; \\ &\text{for each } v = 1, \ldots, n \text{ repeat:} \\ &\text{if } \boldsymbol{v \in S(k-1)}, \text{ then } m \leftarrow m + 1, \text{ and} \\ &\text{if, moreover, } G(v, u) \text{ then } \mathrm{inc}(l) \leftarrow \text{"true"}, \text{ and finally} \\ &\text{if at the end } m < |S(k-1)|, \text{ then output "no";} \\ &\text{otherwise return } \mathrm{inc}(l). \end{aligned} \tag{5.18}$$

Note that to compute (5.18) we need tests

- $v \in S(k-1)$? (will be done in the next loop),
- $G(v,u)$ (possible by the input information),
- $m < |S(k-1)|$ (possible by (5.16)).

Hence, indeed the computations of (5.18) can be done.

So it remains to test the condition "$v \in S(k-1)$". But this is simply the reachability problem for paths of lengths at most $k-1$, and hence can be done in our **fourth** loop described as:

$$\begin{array}{l} w_0 \leftarrow w, \\ \text{for } p = 1, \ldots, k-1 \text{ do:} \\ \text{guess a vertex } w_p \text{ and check that } G(w_{p-1}, w_p) \text{ and} \\ \text{if not "give up"; and otherwise} \\ \text{if } w_{k-1} = v \text{ report } v \in S(k-1); \\ \text{otherwise "give up".} \end{array} \tag{5.19}$$

Note that this is the second algorithm of Example 5.15.

So we have completed the description of our algorithm.

TM implementation: The above algorithm uses 9 variables, namely

$$k, |S(k-1)|, l, u, m, v, p, w_p \text{ and } w_{p-1}$$

assuming integer values up to n, and one variable namely $\mathrm{inc}(l)$ assuming value "false"/"true". Moreover, for each of these variables only one value is needed to be remembered at any stage. Further the operations of the algorithm are "comparing two numbers", "increasing by one" or "comparing whether $u = v$ of $v \longrightarrow u \in G$", so that a nondeterministic TM with 9 work tapes together with the input and output tapes can compute $S(n-1)$ using only $\log_2 n$ cells on any work tape.

Correctness of the algorithm: We shall prove inductively that it computes $|S(k)|$ correctly on all $k \geq 0$. The case $k = 0$ is clear, so we can consider the general case $k \geq 1$.

We note first that the algorithm gives the correct answer in at least one computation on a given w: This is obtained from a computation computing $|S(k-1)|$

correctly (by induction) and then guessing the paths for all v in $S(k-1)$ correctly. Then, indeed the algorithm gives an output $|S(k)|$.

Second we have to show that when the algorithm gives an output different from "no" the output is correct. Consider such a computation, which necessarily has answered affirmatively to the test $m \stackrel{?}{=} |S(k-1)|$. We have to conclude that in (5.17)

$$l \text{ is incremented} \iff \text{the current } u \in S(k). \tag{5.20}$$

Indeed, then the final value l, which is the output, is correct. As we already noted in the considered computation the final value of m equals to $|S(k-1)|$ (which is correctly computed by induction hypothesis). This means that in loop (5.19) **all v's from** $S(k-1)$ are correctly verified (since (5.19) does not report a path $u \longrightarrow v$ unless there is such) so that the loop (5.17) is left with the value $\mathrm{inc}(l) =$ "true" iff $u \in S(k)$. Therefore (5.20) holds.

This completes the proof of Theorem 5.17. □

Now, we obtain our goal rather easily.

Theorem 5.18 (Szelepscenyi-Immerman, 1988). If $f(n) \geq \log_2 n$ and fully space constructible, then

$$\mathrm{NSPACE}(f(n)) = \text{CO-NSPACE}(f(n)).$$

Proof. Since for any problem P co-(co-P) $= P$ it is enough to prove the implication in one direction. So assume that $L \in \mathrm{NSPACE}(f(n))$, and assume that it is accepted by a NTM M operating in space $f(n)$. We construct a NTM $\bar{M}$ operating in space $\mathcal{O}(f(n))$ and accepting the complement $L^{\complement}$ of L.

We recall the computation graph $G(M, w)$ of M on input w (without the "yes" state). We know that the number of vertices of $G(M, w)$ is at most $d^{f(n)}$ for some constant depending only on M.

Now $\bar{M}$ is constructed as follows:

1) First $\bar{M}$ writes to a new tape the **constant** information of M, i.e. its transitions;

2) $\bar{M}$ marks the space $f(n)$ on a new tape (in order to be able to run through all configurations of M);

3) $\bar{M}$ simulates the TM of Theorem 5.17 on the computation graph $G(M, w)$; and

   - if for some k an accepting configuration is found in $S(k)$, then $\bar{M}$ rejects w,
   - otherwise if $|S(n-1)|$ is computed and no accepting configurations are encountered, then $\bar{M}$ accepts.

Since $|S(n-1)|$ here is bounded by $d^{f(n)}$ the space used by $\bar{M}$ is $\mathcal{O}(f(n))$. Hence, the final machine can be achieved by a linear speed-up.

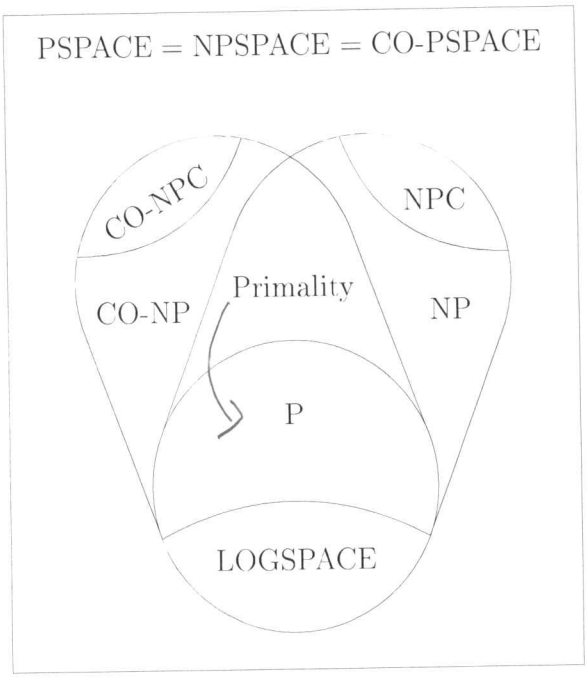
That $\bar{M}$ indeed can simulate the TM of Theorem 5.17 requires a few comments:

- Now, $\bar{M}$ does not have $G(M, w)$, but only transitions of M; this however is enough since in the proof of Theorem 5.17 we used G only to check whether $G(v, u)$ held true;
- The loops using all vertices of G can be realized by using the marked tape.

Hence, our proof of Theorem 5.18 is complete. □

Remark 5.6. Our important Theorems 5.8 and 5.18 clearly point out the difference between time and space complexity classes: no similar variants are known for time classes. If we want to simulate a nondeterministic time class by a deterministic one no better than exponential blow-up is known, cf. Theorem 5.7 (iii). For the complement the situation is analogous.

We conclude the complexity class considerations by reconsidering the basic open problems, and in particular the class CO-NP:



(5.21)

Basic complexity classes

Recall, as we have noted already several times, that the only known proper inclusion is LOGSPACE $\subsetneq$ PSPACE; hence everything else in (5.21) may collapse.

Let us consider a bit more carefully the family CO-NP, i.e. the family of problems the "complements" of which can be accepted in polynomial time nondeterministically.

To clarify what is the **complementary** problem we take a few examples:

P:	$n \in \mathbb{P}$?	SAT	HCP
co-P:	Is n composite?	"always false"	"no hamilton cycles"

Note, however, that if we encode problems into inputs of TM's, then the set of encodings of P_y is not the complement of the set of encodings of co-P_y. This is because the complements contain also words which are not encodings at all. However, these can be "forgotten" (under usual encodings) since they can be separated in a required space or time — usually. Notice also that the problem VALIDITY (which asks to decide whether a given formula is always true) is closely related to co-SAT:

$$\alpha \in \text{SAT}_n \iff \alpha \in \text{co-SAT}_y \iff \neg\alpha \in \text{VALIDITY}_y.$$

As we defined NP-complete problems we can define CO-NP-complete problems: P is **CO-NP-complete** iff

(i) $P \in$ CO-NP, and

(ii) $\forall P' \in$ CO-NP, $P' \leq_p P$.

We have a simple

Fact 5.19. If L is NP-complete, then $L^{\complement} = \Sigma^* \setminus L$ is CO-NP-complete.

Proof. Clearly $L^{\complement}$ is in CO-NP.

Let $L' \in$ CO-NP. Then $\Sigma^* \setminus L' \in$ NP so that it reduces in polynomial time to L, say a TM M_t gives this reduction:

$$\begin{array}{ccc} \Sigma^* \setminus L' & & L \\ \cup & & \cup \\ w & \longmapsto & M_t(w) \end{array} \qquad \text{s.t.} \qquad w \in \Sigma^* \setminus L' \iff M_t(w) \in L.$$

Hence also

$$\begin{array}{ccc} L' & & \Sigma^* \setminus L = L^{\complement} \\ \cup & & \cup \\ w' & \longmapsto & M_t(w') \end{array} \qquad \text{s.t.} \qquad w' \in L' \iff M_t(w') \in L^{\complement}.$$

So, indeed, $L^{\complement}$ is CO-NP-complete. □

Example 5.20. VALIDITY is CO-NP-complete: SAT is NP-complete and so is SAT $\cup$ {nonencodings}. Hence $\{\alpha \mid \alpha$ is always false$\}$ is CO-NP-complete and so is VALIDITY.

As we have mentioned already it is not known whether NP = CO-NP, unlike PSPACE = CO-PSPACE. Of course, if P would equal to NP then also

$$\mathbf{NP} = \mathrm{P} = \text{CO-P} = \mathbf{CO\text{-}NP}.$$

On the other hand, even if P $\neq$ NP it would not be contradictory to have NP = CO-NP. However, this is not expected to be the case.

What we can prove is

Fact 5.21. If a CO-NP complete problem is in NP, then NP = CO-NP

Proof. Let $L \in$ NP be CO-NP-complete.

(i) CO-NP $\subseteq$ NP. Let $L' \in$ CO-NP. Then since L is CO-NP-complete we have a reduction $L' \leq_p L$. Assume that a DTM M_t gives this reduction. Then M_t maps words of L' into L and the words of $L'^{\complement}$ into $L^{\complement}$. So if M is a NTM for L, then the composition $M \circ M_t$ is a NTM accepting L'. Hence $L' \in$ NP, as was to be proved.

(ii) NP $\subseteq$ CO-NP. Now $L^{\complement}$ is in CO-NP and NP-complete. And the proof goes as above: Let $L' \in$ NP. Then $L' \leq_p L^{\complement}$ via a DTM M_t. So if M is a NTM for L, then the machine $M \circ M_t$ accepts $L'^{\complement}$, i.e. $L' \in$ CO-NP. □

Remark 5.7. The fact that a problem P is in NP means that

(i) Yes-instances can be verified in a polynomial time.

On the other hand, the fact that P is in CO-NP means that

(ii) No-instances can be verified in a polynomial time.

Hence if $P \in \text{NP} \cap \text{CO-NP}$, then both yes- and no-instances have a polynomial time verification procedure. This however does not mean — according to the current knowledge — that P is in the class P. The converse, however, is clearly true.

The above should be compared to the identity (cf. Recursive Functions)

$$\text{RE} \cap \text{CO-RE} = \text{Rec}$$

Here we only know that

$$\text{NP} \cap \text{CO-NP} \supseteq \text{P}\,.$$

We conclude our considerations by considering **complete problems with respect to different complexity classes**. Informally, we say that a language L' is **complete** with respect to the family $\mathcal{L}$ if

(i) $L' \in \mathcal{L}$, and

(ii) for each $L \in \mathcal{L}$, L **reduces** to L', in symbols $L \leq L'$.

Formally, the reductions can be of different types: most common are

(iii) **polynomial time** reduction $\leq_{\text{pol}}$, and

(iv) **logarithmic space** reduction $\leq_{\log}$.

According to our earlier considerations the former means that there exists a DTM M translating each instance of L into that of L' operating in polynomial time and satisfying:

$$w \in L \iff M(w) \in L'.$$

In log-space reduction we require that M operates in **logarithmic space** instead of polynomial time.

We can conclude the following basic facts on log-space reductions. Most of the results are very similar to those for $\leq_{\text{pol}}$ shown in Facts 4.1–4.3. In order to clarify our considerations we require that the machine M used in log-space reductions satisfies:

- it is always terminating (Lemma 5.3);
- not only input, but also output tape is word preserving, i.e. overwriting is not allowed, and head can move only to the right (by definition);
- the complexity is measured using only work tapes (by definition).

By Theorem 5.7 (ii) we have:

Fact 5.22. If $L \leq_{\log} L'$, then $L \leq_{\text{pol}} L'$. Consequently if $L \leq_{\log} L'$ and $L' \in$ (N)PTIME, then $L \in$ (N)PTIME.

Due to the basic open problem LOGSPACE $\stackrel{?}{=}$ PTIME, we do not know whether the converse of Fact 5.22 holds true. Despite of that in concrete cases we can often replace $\leq_{\text{pol}}$ by $\leq_{\log}$ as shown in the next

Example 5.23. SAT is NP-complete with respect to log-space reductions. Indeed, the proof of Theorem 4.7 works if we can conclude that the transformation

$$w \to \alpha(w)$$

can be done in log-space (and not only in polynomial time). But this is no problem since to compute $\alpha(w)$ it is enough to do the countings up to $p(n)$, for some polynomials $p(n)$, and this can be done in $\log(n)$ space using binary numbers (cf. proof of Theorem 4.7).

A bit more difficult is

Fact 5.24. If $L \leq_{\log} L'$ and $L' \leq_{\log} L''$, then $L \leq_{\log} L''$.

Proof. Now, by the assumptions we have TM's M_1 and M_2 operating in log-space and satisfying:

$$w \longmapsto M_1(w) = w' \longmapsto M_2(w') = M_2 \circ M_1(w) = w''$$

$$M_2 \circ M_1$$

and

$$w'' \in L'' \iff w \in L.$$

However, there is a problem since $|w'|$ is not $\mathcal{O}(\log |w|)$, but only $\mathcal{O}(|w|^c)$ for some constant (cf. e.g. the proof of Lemma 5.3). To overcome this we note that $|w'|$ in binary is $\mathcal{O}(\log_2 |w|)$!!

The machine M_3 simulating $M_2 \circ M_1$ cannot write down w', but instead computes over and over again the position i of the reading head of M_2, i.e. the i'th output of M_1.

In more details M_3 behaves as follows:

- it simulates the work tapes of M_1 and M_2 using space $\log n$ and $\log(n^c) = \mathcal{O}(\log n)$, respectively;
- remembers in a new tape the position $i \leq \mathcal{O}(\log n)$ of the reading head of M_2;
- in order to simulate a step of M_2, M_3 has to know the symbol M_2 is scanning on its output tape (the other necessary information M_3 already has); this is achieved by simulating a computation of M_1 as long as the i'th output letter is produced — hence M_3 can do that;
- the simulation of M_1 is no problem;
- finally M_3 terminates when M_1 does so without finding any more the i'th output symbol.

Clearly, M_3 works as intended and operates in space $\mathcal{O}(\log n)$, and hence in space $\log_2 n$ after a linear speed-up. □

Remark 5.8. By Example 5.23 and Fact 5.24, we obtain

$$\text{SAT} \in \text{SPACE}(\log n) \implies \text{SPACE}(\log n) = \text{NP}$$

and

$$\text{SAT} \in \text{NSPACE}(\log n) \implies \text{NSPACE}(\log n) = \text{NP}\,.$$

These are illustrations of the usefulness of complete problems: The equality of two complexity classes is reduced to a problem whether a given language is in the "smaller" of these classes!

Next we start to search for complete problems with respect to our basic hierarchy:

$$\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE} = \text{NPSPACE}\,.$$

Note that by Exercise 5/I, cf. also Theorem 5.7 (ii) and Remark 5.4, NLOGSPACE is in a proper place.

LOGSPACE: We have to consider log-space reductions. Moreover, the problem is trivial: Any $L' \neq \emptyset, \Sigma^*$ in LOGSPACE is complete here. Indeed, a log-space reduction of $L \in \text{LOGSPACE}$ to L' is obtained by modifying the machine M deciding L as follows: It simulates M and if that outputs "yes" the new machine writes some $w_y \in L'$, and otherwise it writes some $w_n \notin L'$. Clearly, the formal requirements of reducibility become fulfilled.

NLOGSPACE: Here an example of a complete problem is the GRP from page 88:

1. By example 5.15, GRP $\in$ NLOGSPACE.

2. To prove the log-space reducibility let $L \in \text{NLOGSPACE}$, say L is accepted by a NTM M. We assosiate an input w of M with the computation graph $G(M, w)$ including the "yes"-vertex, cf. page 90:

   $$w \mapsto G(M, w).$$

   Then, as we know,

   $w \in L \iff$ "yes" is reachable from the initial configuration of M in $G(M, w)$.

So it remains to be shown that the above transformation can be done by a DTM M_t in log-space. But even this is very easy: For a given input w M_t marks the space of M and checks in a lexicographic order for each two configurations c_1 and c_2 of M whether $c_1 \to_M c_2$ and if "yes", M_t outputs $c_1 \longrightarrow c_2$. Clearly, such an M_t operates in space $\log n$ and computes $w \mapsto G(M, w)$.

So we have

Theorem 5.25. The graph reachability problem GRP is complete for the class NLOGSPACE under log-space reduction.

Remark 5.9. One can show that 2-CNF from Theorem 4.9 is another complete problem for this family, cf. Papadimitriou.

PTIME: Without proving the P-completeness we present a natural problem from logic being such. The problem is as follows:

CVP (Circuit Value Problem)

Input: Boolean circuit (without unknowns).

Output: The value of the circuit.

For example the following are instances of CVP:

Clearly, the value of the first circuit is T while that of the second is F. Note also that this problem is closely connected to CNF. Indeed, it is exactly CNF for a fixed values of unknowns.

It follows from this observation that CVP $\in$ PTIME, in fact can be solved in square time. That every problem in PTIME can be log-space (and hence also polynomial time) reduced to CVP is proved in Papadimitriou (not particularly difficult). Hence we have

Proposition 5.26. CVP is P-complete under log-space reduction.

Remarks.

5.10. Proposition implies that the question whether (N)LOGSPACE = PTIME are reduced to the questions whether CVP $\in$ (N)LOGSPACE.

5.11. Another example of P-complete problems is the **emptiness problem for context-free grammars**: Given a CF grammar G, is $L(G) \stackrel{?}{=} \emptyset$, cf. Hopcroft-Ullman.

NPTIME: Chapter 4.

PSPACE: In above we restricted CNF in order to obtain CVP. Now we extend it to so-called **validity problem for quantified Boolean formulas**, QBF for short.

Quantified Boolean Formulas are defined as:

(i) variable x is QBF;

(ii) if E_1 and E_2 are QBF's, so are $E_1 \wedge E_2$, $E_1 \vee E_2$ and $\neg E_1$;

(iii) if E is QBF, so is $\forall x E$ and $\exists x E$;

(iv) the above are all QBF's.

In a QBF a variable can be **free** or **bounded**: In (iii) the free occurrences of x in E become bounded in $\forall x E$ and $\exists x E$, while in (ii) the status of an occurrence of a variable remains. Finally, a variable alone is always free.

Further the **domain** of $\forall x$ and $\exists x$ in the formulas $\forall x E$ and $\exists x E$ is the set of all free occurrences of x in these formulas. We call a QBF **bounded** iff it does not contain free occurrences of variables.

With each bounded QBF we can associate its **truth value** by formulas

$$\begin{aligned}
\mu(E \wedge F) &= \mu(E) \wedge \mu(F), \\
\mu(E \vee F) &= \mu(E) \vee \mu(F), \\
\mu(\neg E) &= \neg \mu(E), \\
\mu(\forall x E) &= \mu(E_0) \wedge \mu(E_1), \\
\mu(\exists x E) &= \mu(E_0) \vee \mu(E_1),
\end{aligned}$$

where E_0 and E_1 are obtained from E by replacing all free occurrences of variables x in E by 0 and 1, respectively. Now we define the problem

QBF problem

Input: Bounded quantified Boolean formula.

Output: The value of the formula.

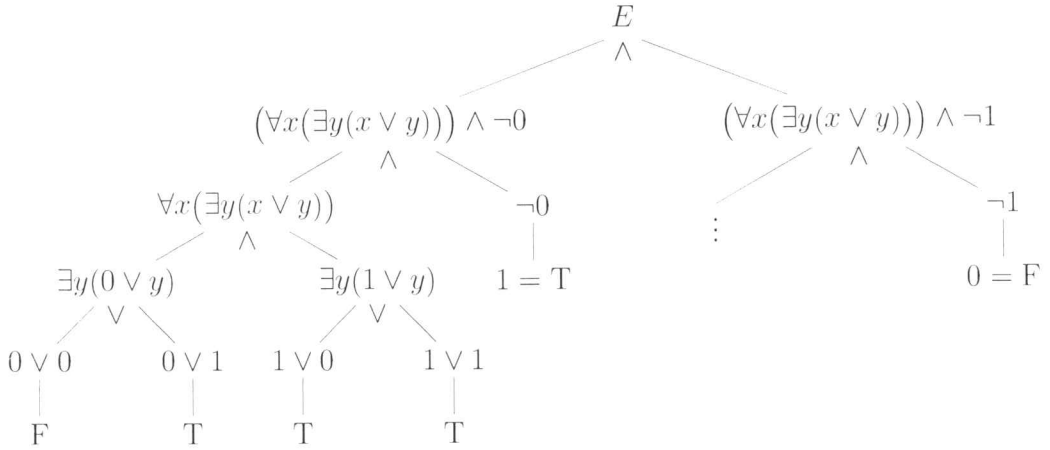
Note that again we have a close connection to CNF problem: A CNF formula α is satisfiable iff $\exists x_1 \cdots \exists x_n \alpha$ is true, where $x_1, \ldots, x_n$ are the variables of α. Hence combining the considerations of the previous case we have:

$$\text{CVP} \leq_{\log} \text{CNF} \leq_{\log} \text{QBF}.$$

Example 5.27. Consider the QBF

$$E = \forall x\big(\forall x\big(\exists y(x \vee y)\big) \wedge \neg x\big),$$

where the arrows indicate the domains. Its truth value can be computed from the following tree:



Thus $\mu(E) = \text{F}$. Note that in this tree there are both $\wedge$- and $\vee$-branches, and moreover each of those can be of two different types.

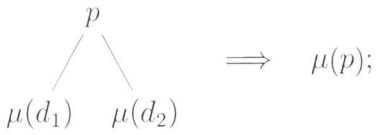
This example directly extends to the following auxiliary result:

Lemma 5.28. QBF is in PSPACE.

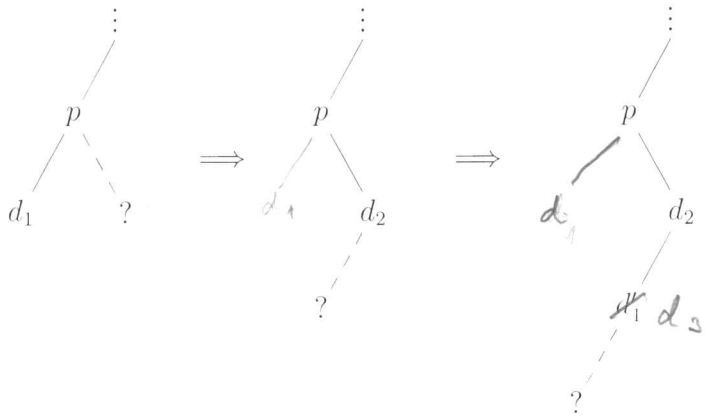
Proof. Let E be an instance of QBF and let μ give its value, or more generally a value of a formula obtained from a bounded QBF by fixing some values of variables.

As we indicated in Example 5.27 $\mu(E)$ can be computed from a tree associated to E. However, the whole tree cannot be stored in polynomial space. But fortunately this is not needed either: at any moment it is enough to remember just one path of a tree (as in the proof of Savitch's Theorem):

- to compute $\mu(p)$ for a node p it is enough to know the values of its descendants:



- a new path to be remembered can be computed from the old one:



Therefore a DTM needs to remember only a path from the root to a vetex in order to be able to compute $\mu(E)$. And the computations are very easy. Hence the requirement of the space is

$$|E| \cdot \text{the depth of the tree} = \mathcal{O}\big(|E|^2\big). \qquad \square$$

Theorem 5.29. QBF is PSPACE-complete under polynomial time reduction.

Proof. By Lemma 5.28, we have to show that QBF is PSPACE-hard. So let M be a DTM operating in space $p(n)$, accepting L and having **only 1 tape** which can

be assumed since M operates in time $\geq n$. We construct a DTM M_t operating in polynomial time and computing

$$\begin{array}{ccc} w & \longmapsto & E(w) \\ \cap & & \cap \\ \Sigma^* & & \text{QBF} \end{array}$$

such that

$$w \in L \iff \mu(E(w)) = 1. \tag{5.22}$$

Our construction uses ideas of proof of Cook's Theorem, so that all details are not presented here. Note also that the direct construction of Theorem 4.7 would give a formula $\exists x_1 \cdots \exists x_n \alpha(w)$ satisfying (5.22) but being of length $\Omega(c^{p(n)})$ for some c (since M operates in time $\Omega(c^{p(n)})$).

Therefore we have to modify the proof of Theorem 4.7. We start by constructing a formula Conf containing variables

$$I : c_{i,x} \qquad i = 0, \ldots, p(n) \quad \text{and} \quad x \in \Sigma \times \{Q \cup \{\#\}\} = \Sigma'.$$

The meaning of Conf is to characterize configurations of M such that it gets the value "true" iff those variables getting value "true" corresponds a configuration of M. We set

$$\text{Conf}(I) = \bigwedge_i \left(\bigoplus_x c_{i,x} \right) \wedge \bigoplus_{\substack{i,x \\ x \notin \Sigma \times \{\#\}}} c_{i,x}.$$

So i runs from 0 to $p(n)$, where $n = |w|$, and x contains a pair representing a symbol of the tape and the state of M including $\#$ as "no state in particular cell".

Next we define QBF's

$$F_j(I_1, I_2),$$

$j = 0, \ldots, p(n) \log c$, such that

(i) I_1 and I_2 are disjoint sets of variables:

$$I_1 = \{c_{i,x} \mid i = 0, \ldots, p(n),\ x \in \Sigma'\},$$
$$I_2 = \{d_{i,x} \mid i = 0, \ldots, p(n),\ x \in \Sigma'\};$$

(ii) $\mu\big(F_j(I_1, I_2)\big) = \text{true} \iff \beta$ and β' represent the configurations I_1 and I_2 such that

$$x_1 \cdots x_{p(n)} = \beta \xrightarrow{\leq 2^j} \beta' = y_1 \cdots y_{p(n)}.$$

Assuming that the formulas $F_j(I_1, I_2)$ are defined we set

$$E(w) = \exists I_0 \exists I_f \Big(F_{p(n)\log c}(I_0, I_f) \wedge \text{Init}(I_0) \wedge \text{Final}(I_f) \Big),$$

where, for example,

$$\exists I_0 = \bigexists_i \bigexists_x c_{i,x},$$

and the formulas $\text{Init}(I_0)$ and $\text{Final}(I_f)$ characterize the initial and accepting configurations of M on w (cf. the proof of Theorem 4.7 for this construction).

It follows that

$$\mu(E(w)) = \text{true}$$

$$\Updownarrow$$

$\exists \beta, \beta' : \beta = (q_0; \rhd, w) \overset{2^{p(n)\log c}}{\longrightarrow} \beta'$, where β' is an accepting configuration.

So it remains to be shown how the formulas $F_j(I_1, I_2)$ are defined. Let us do this recursively:

Initial step. For $j = 0$ we require:

- I_1 and I_2 represent configurations β_1 and β_2 which are enforced by the formulae $\text{Conf}(I_1)$ and $\text{Conf}(I_2)$.
- Either $\beta_1 = \beta_2$ or $\beta_1 \to_M \beta_2$. Again these conditions can be achieved: the first on by using the connective $\Leftrightarrow$ which can enforce the corresponding variables to obtain the same value, and the second one using the formula Trans from the proof of Theorem 4.7.

Induction step. The most natural idea to write

$$F_j(I_1, I_2) = \exists I \Big(\text{Conf}(I) \wedge F_{j-1}(I_1, I) \wedge F_{j-1}(I, I_2) \Big) \tag{5.23}$$

works correctly, but leads to a too long formula $F_{p(n)\log c}\big(I_0, I_f\big)$. Instead we write

$$F_j(I_1, I_2) = \exists I \forall J \forall K \Big(\big((J, K) = (I_1, I) \vee (J, K) = (I, I_2) \big) \Rightarrow F_{j-1}(J, K) \Big) \wedge \text{Conf}(I). \tag{5.24}$$

Here for example $K = I_1$ means the formula which is satisfied iff the corresponding variables get the same value. Of course, the implication, as well as other connectives used, can be eliminated as usual!

Now it is straightforward to see that

(5.23) is true iff (5.24) is true.

Consequently, the exponential growth in the length of $F_j(I_1, I_2)$, hinted by (5.23), can be avoided by using the expressive power of predicate logic. Indeed, in (5.24) the length of the formula grows only by an additive factor of polynomial size in each step.

Hence we have completed the construction of $E(w)$. To estimate its length we first note that the total number of variables in the formula F_j is $\mathcal{O}(jp(n))$, so that each variable can be stored in space $\mathcal{O}(\log n)$. Further recalling $|F_j| \leq |F_{j-1}| + \mathcal{O}(p_1(n))$, for some polynomial $p_1(n)$, and constructions of the proof of Theorem 4.7, we can see that $E(w)$ is of a polynomial size, and can also be generated in polynomial time. □

As in the case of NP-complete problems, also here we can derive rather straightforwardly from a PSPACE-complete problem new similar problems. We state without proofs, cf. Hopcroft-Ullman and Papdimitriou (Problem 20.2.13), two such problems.

I. **Membership problem for NTM's operating in linear space** (or equivalently for **context-sensitive grammars**): "Given a NTM M operating in linear space, and its input word w, decide whether or not $w \in L(M)$."

II. **The equivalence problem for nondeterministic finite automata**: "Given two nondeterministic finite automata $\mathcal{A}_1$ and $\mathcal{A}_2$ decide whether they are equivalent, i.e. $L(\mathcal{A}_1) = L(\mathcal{A}_2)$."