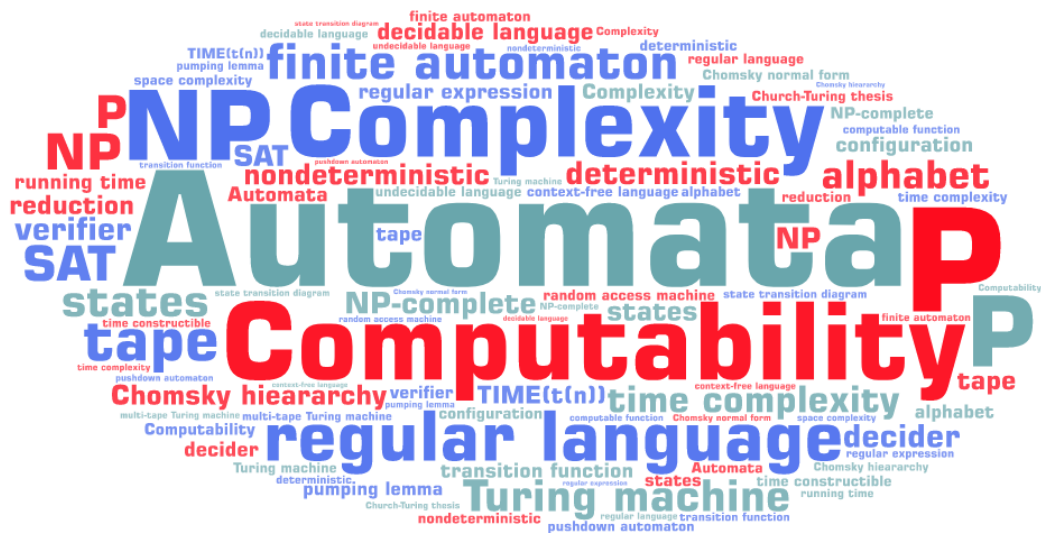


Automata, Computability and Complexity

Lecture notes



Author

Peter Zaspel

February 27, 2023

Contents

0	Primer in Math	3
0.1	Mathematical notions and terminology	3
0.2	Proofs	13
1	Regular languages	20
1.1	Finite automata	20
1.2	Nondeterministic finite automata	31
1.3	Regular Expressions	45
1.4	Nonregular Languages	56
2	Context-Free Languages	60
2.1	Context-Free Grammars	60
2.2	Pushdown Automata	65
2.3	Non-context-free languages	79
3	Turing Machines	82
3.1	Formal definition and examples	82
3.2	Variants of Turing Machines	92
4	Decidability	108
4.1	Decidable languages	109
4.2	Undecidability of the halting problem	115
4.3	Undecidability via mapping reductions	120
4.4	Turing-recognizable languages and beyond	124
5	Time complexity	130
5.1	Measuring complexity	130
5.2	The class P	138
5.3	The class NP	142
5.4	Polynomial time reducibility	150
5.5	NP-Completeness	151
5.6	Hierarchy theorems	167
	Bibliography	171

0 Primer in Math

This chapter briefly summarizes some crucial mathematical notions, terminology and proof techniques that the reader is expected to be familiar with. Hence, we fix basic notation for this lecture and provide a very basic repetition of content.

0.1 Mathematical notions and terminology

0.1.1 Sets and sequences

We start by introducing sets, their elements, and core properties.

Definition (Set) A **set** is a group of objects represented as a unit. Sets may contain any type of object. The objects in a set are called its **elements** or **members**.

Example 0.1 There are different ways to describe sets:

- $\{7, 21, 57\}$,
- $\{1, 2, 3, \dots\}$, (*description by listing*)
- $\{n | n = m^2, m \in \mathbb{N}\}$. (*description by a rule*)

△

Remark Sets have no ordering.

△

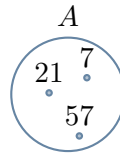
Definition A, B sets.

- \emptyset : **empty set**
- $a \in A$: a is **element** of set A .
- $b \notin A$: b is **not element** of set A .
- $|A|$: **cardinality** of set A , i.e. the number of elements in A

Example 0.2 Let $A = \{7, 21, 57\}$. It holds

- $7 \in A$,
- $10 \notin A$,
- $|A| = 3$.

We can visualize A by a *Venn diagram*:



△

Operations on sets can be introduced as follows.

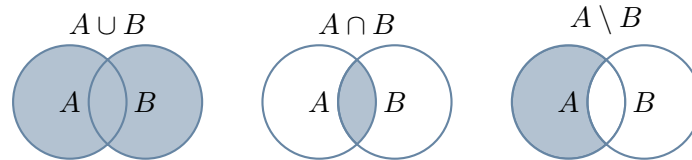
Definition A, B sets.

- $A \cup B$: **union** of sets A and B , i.e. combination of elements of A and B .
- $A \cap B$: **intersection** of sets A and B , i.e. set of elements that are both in A and B .
- $A \setminus B$: **difference** between sets A and B , i.e. set of elements that are contained in A but not in B .

Example 0.3 Let $A = \{7, 21, 57\}$. $B = \{2, 5, 7\}$. It holds

- $A \cup B = \{2, 5, 7, 21, 57\}$,
- $A \cap B = \{7\}$,

with the Venn diagrams



△

Furthermore, we have basic relations between sets.

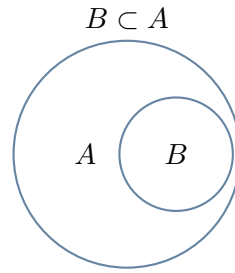
Definition A, B sets.

- $A \subseteq B$: A is **subset** of B , if every member of A is also member of B .
- $A \subsetneq B$ or $A \subset B$: A is **proper subset** of B , if A is subset of B and not equal to B .

Example 0.4 Let $A = \{7, 21, 57\}$, $B = \{21, 57\}$, $C = \{7, 21, 57, 10\}$. It holds

- $B \subseteq A$, $B \subset A$, $B \subsetneq A$,
- $A \subseteq C$, $B \subseteq C$.

We can visualize this via the following Venn diagram:



△

Definition A, B sets.

$A = B$: A and B are **equal**, if $A \subseteq B$ and $B \subseteq A$

Remark It holds $\{7, 57, 21\} = \{7, 57, 57, 57, 57, 21\}$, however we sometimes want to allow for multiple identical elements in one set. This would then be a *multiset*. △

Finally, we introduce the *power set*, which will be of importance, later.

Definition A set.

The **power set** of A is the set of all subsets of A .

Example 0.5 Let $A = \{0, 1\}$. Its power set is $\{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$. △

As we have seen, the elements of sets are not ordered and are unique. If we want to collect objects in an ordered way and / or want to allow for duplicates, we can do that with *sequences* or *k-tuples*.

Definition (Sequence, k-tuple) It holds:

- A **sequence** of objects is a list of objects in some order.
- Finite sequences are called **tuple**.
- A sequence of k elements is a **k-tuple**.
- A sequence of 2 elements is a **pair**.

Example 0.6 It holds:

- $(7, 21, 57)$ is a sequence.
- $(7, 21, 57) \neq (7, 57, 21)$.
- $(7, 7, 5, 2)$ is a 4-tuple.

△

A further operation between two sets that is of importance is the *Cartesian product*:

Definition A, B , sets.

The **Cartesian product** or **cross product** $A \times B$ of A and B is the set of all pairs wherein the first element is a member of A and the second element is a member of B , hence

$$A \times B := \{(a, b) | a \in A, b \in B\}.$$

Example 0.7 Let $A = \{1, 2\}$, $B = \{x, y, z\}$. The Cartesian product of A and B is

$$A \times B = \{(1, x), (1, y), (1, z), (2, x), (2, y), (2, z)\}.$$

△

0.1.2 Functions and graphs

We now would like to introduce some more complex mathematical objects and start with *functions*.

Definition (Function) A **function** or **mapping** sets up an input-output relationship. A function takes an input and produces an output. In every function, the same input always produces the same output.

- $f(a) = b$: f is function whose output value is b for input a ; or: f **maps** a to b .
- The set of possible inputs of f is called **domain**.
- The set of possible outputs of f is called **range**.
- Write $f : D \rightarrow R$, if D / R are the domain / range of f .

Example 0.8 The following is a meaningful function:

$$\text{abs} : \mathbb{Z} \rightarrow \mathbb{Z},$$

with $\text{abs}(a) = \text{abs}(-a) = |a|$.

△

Typical properties of functions are further given by

Definition Let A, B be sets and f a function from A to B , i.e.

$$f : A \rightarrow B.$$

The function f is

- **injective** / **one-to-one** if for all a, a' in A it holds

$$f(a) = f(a') \Rightarrow a = a',$$

- **surjective** / **onto** if

$$\text{range}(f) = B,$$

- **bijective** if it is injective and surjective.

A bijective function is called **bijection** / **correspondance**.

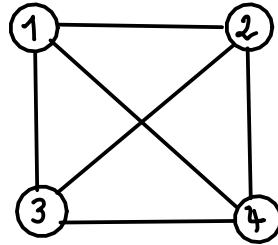
Graphs are often introduced as a data structure in Computer Science. We recall their basic properties.

Definition An **(undirected) graph** $G = (V, E)$ is a set of *nodes* / *vertices* V that are connected by a set of **edges** $E \subseteq \{\{v_1, v_2\} | v_1, v_2 \in V\}$. The number of edges at a node v is the **degree** $\deg(v)$ of the node.

Example 0.9 An example for a graph $G = (V, E)$ is given by the choices

- $V = \{1, 2, 3, 4\}$
- $E = \{\{1, 4\}, \{2, 3\}, \{1, 2\}, \{2, 4\}, \{3, 4\}, \{3, 1\}\}$

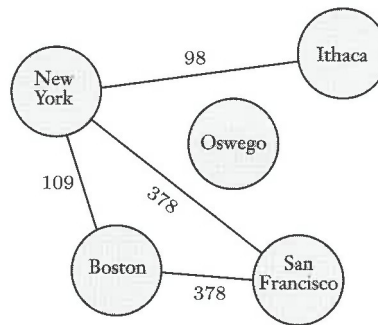
It e.g. holds $\deg(2) = 3$. The graph can be visualized as follows:



△

Remark We usually denote edges of undirected graphs by pairs (a, b) (instead of sets) with $(a, b) = (b, a)$, slightly abusing the common notation. △

Remark We sometimes also use labeled graphs:



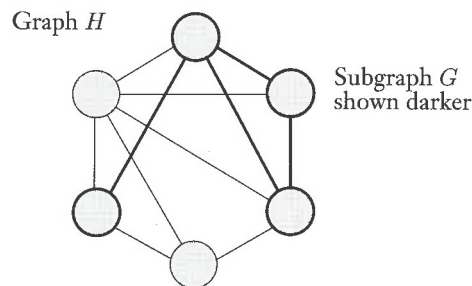
△

Similar to subsets, we can also have *subgraphs*.

Definition G, H graphs.

G is **subgraph** of H if the nodes of G are a subset of the nodes of H .

Example 0.10 The below figure shows a graph G with a subgraph H .



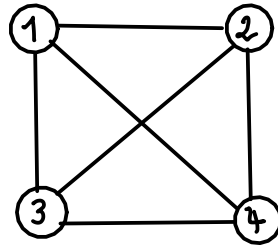
△

In some applications, we further need the concept of a *path*.

Definition $G = (V, E)$ graph.

- A **path** in G is a sequence of nodes connected by edges.
- A **simple path** in G is a path that doesn't repeat any nodes.
- A **cycle** is a path in G that starts and ends in the same node.
- A **simple cycle** is a cycle in G that doesn't repeat any node except for the first and last.

Example 0.11 Let the following graph be given:



It holds

- $(1, 2, 3, 4)$ is a path,
- $(1, 2, 4, 1, 2, 3)$ is another path,
- $(1, 2, 3, 4)$ is a simple path and
- $(1, 2, 3, 1)$ is a cycle.

△

Another important structure is a *tree* that is a specific form of a graph:

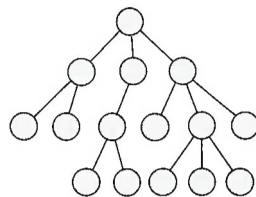
Definition $G = (V, E)$ graph.

- G is **connected**, if every two nodes have a path between them.
- G is a **tree**, if is connected and has no simple cycles.
- The nodes of degree 1 in a tree are the **leaves** of the tree.

Remark Sometimes we call a specifically designated node of a tree *root*.

△

Example 0.12 The following graph is a tree:



△

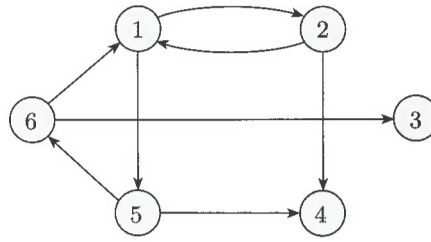
Next, we would like to introduce a *directed graph*. In contrast to an undirected graph, edges of a directed graph have an orientation.

Definition A **directed graph** $G = (V, E)$ is a set of nodes / edges V that are connected by a set of **directed edges** $E \subseteq V \times V$, i.e. the elements of E are *pairs*.

Example 0.13 The directed graph

$$G = (\{1, 2, 3, 4, 5, 6\}, \{(1, 2), (1, 5), (2, 1), (2, 4), (5, 4), (5, 6), (6, 1), (6, 3)\})$$

can be visualized as follows:



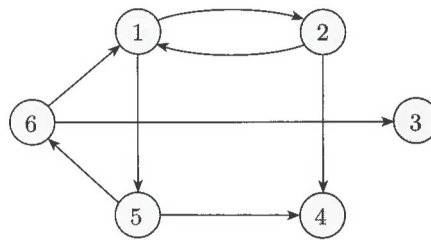
△

Some of the previous definitions for undirected graphs finally need some refinement for directed graphs:

Definition $G = (V, E)$ directed graph.

- The number of arrows pointing away from a particular node in G is the **outdegree**. The number of arrows pointing to a particular node in G is the **indegree**.
- A path in G in which all the arrows point the in the same direction as its steps is called a **directed path**.
- A directed graph is **strongly connected** if a directed path connects every two nodes.

Example 0.14 For the directed graph given by

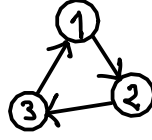


it holds:

- $(6, 1, 5, 4)$ is a directed path and

- the graph is not strongly connected.

An example for a strongly connected graph is the below graph:



△

0.1.3 Strings and languages

In the computing models that we are going to explore, the “data types” on which we compute are *symbols* and *strings*. Let us start by defining what is a symbol.

Definition An **alphabet** is any finite set. The members of an alphabet are the **symbols** of the alphabet.

Example 0.15 We give three examples of alphabets by

- $\Sigma_1 = \{0, 1\}$,
- $\Sigma_2 = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$ and
- $\Gamma = \{0, 1, x, y, z\}$.

△

Now, we can introduce strings.

Definition Σ alphabet. A **string** over Σ is a finite sequence of symbols from Σ .

- If w is a string over Σ , the length of w , i.e. $|w|$, is the number of symbols contained in w .
- The string of length zero is called **empty string**, denoted by ε .

Example 0.16 Let the two alphabets $\Sigma_1 = \{0, 1\}$ and $\Sigma_2 = \{a, b, \dots, z\}$ be given. We have

- 01001 is a string over Σ_1 and
- *abracadabra* is a string over Σ_2 .

△

We further need a few notions for and operations on strings.

Definition Σ alphabet, x string over Σ with $|x| = n$, i.e. $x = x_1x_2 \cdots x_n$, s.th. $x_i \in \Sigma$, y string over Σ with $|y| = m$.

- The **reverse** of x , written $x^{\mathcal{R}}$ is given by $x^{\mathcal{R}} := x_nx_{n-1} \cdots x_2x_1$.
- String z is a **substring** of x if it appears consecutively in x .
- The concatenation of x and y , written xy is the string $x_1 \cdots x_ny_1 \cdots y_m$.

Example 0.17 We fix the alphabet $\Sigma = \{0, 1\}$.

- For a given string $x = 0101$ it holds $x^{\mathcal{R}} = 1010$.
- If we are given strings $x = 001100$ and $y = 11$, then y is a substring of x .
- For given $x = 00$ and $y = 11$ their concatenation is $xy = 0011$.

△

We will spend a major part of our work on characterizing different types of *languages*. Hence, we should first define this notion:

Definition Let Σ be an alphabet. A **language** (over Σ) is a set of strings (over Σ).

Example 0.18

$$\{001, 00, 111, 10\}$$

is a language over the alphabet $\Sigma = \{0, 1\}$.

△

0.1.4 Boolean logic

Without further details, we briefly give a few fundamental notions from Boolean logic.

Definition

- The values TRUE and FALSE are **Boolean values**.
- TRUE and FALSE are represented by 1 and 0.

Definition (Boolean operations) a, b Boolean values.

- The **negation (NOT)** \neg is defined by $\neg 0 := 1$ and $\neg 1 := 0$.
- The **conjunction (AND)** $a \wedge b$ is TRUE if both a and b are TRUE.
- The **disjunction (OR)** $a \vee b$ is TRUE if either of a and b is TRUE.
- The **exclusive or (XOR)** $a \oplus b$ is TRUE if either of a and b , but not both, are TRUE.

- The **equality** operation $a \Leftrightarrow b$ is TRUE if a and b have the same value.
- The **implication** operation $a \Rightarrow b$ is TRUE if a is FALSE or a and b are both TRUE.

Lemma (Distributive law for AND and OR) a, b, c Boolean values.

- $a \wedge (b \vee c)$ equals $(a \wedge b) \vee (a \wedge c)$
- $a \vee (b \wedge c)$ equals $(a \vee b) \wedge (a \vee c)$

0.2 Proofs

The usual reader is expected to have some background on carrying out mathematical proofs. However, it needs to be understood that proofs will be very fundamental to the theory that we will be confronted with. Therefore, we still briefly discuss some recurring statement structures of proofs and give examples for the three most common types of proofs.

Example 0.19 Often, we need to proof a statement of the form

“ P is true, if and only if Q is true.”

The structure of the corresponding proof requires two partial proofs and is given as follows:

Proof:

“ \Rightarrow ” (*forward direction*)

(Here, the proof of the statement “If P is true, then Q is true.” is given.)

“ \Leftarrow ” (*backward direction*)

(Here, the proof of the statement “If Q is true, then P is true.” is given.)

\triangle

Example 0.20 Another common task is to show the equality of two sets, i.e.

“Let A, B be sets. It holds $A = B$ ”

Quite obviously we go via the definition of two sets and thus prove the statement in the following way:

Proof:

“ \subseteq ”

(Here, the proof of the statement “Every member of A is member of B .” is given.)

“ \supseteq ”

(Here, the proof of the statement “Every member of B is member of A .” is given.) \triangle

0.2.1 Proof by construction

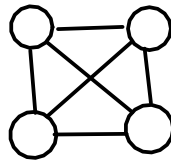
We start our discussion of examples for proof techniques by *proofs by construction*. This type of proof regularly shows up in context of existence proofs. The idea is to explicitly build the object of which we claim the existence. Recalling that a graph is *k-regular*, if every node has degree k , we now state our example theorem, for which we would like to discuss its proof by construction:

Theorem (Example) For each even number n greater than 2 there exists a 3-regular graph with n nodes.

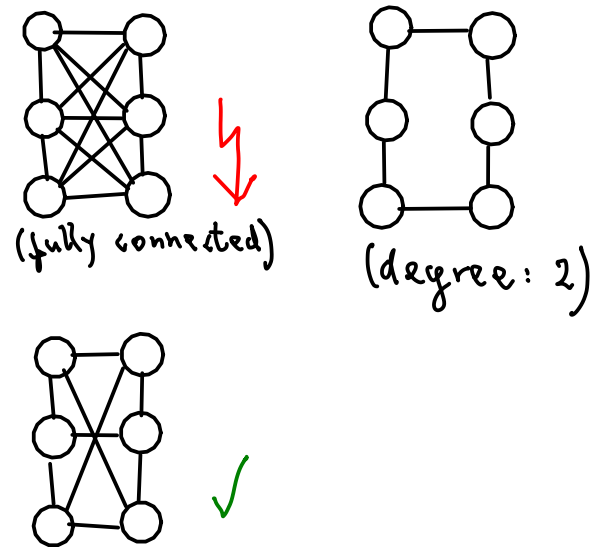
We do not want to immediately give the formal proof, but instead we would like to go through the “mental steps” that someone would go through in order to develop the proof. This is done in the following remark.

Remark (Proof development) The theorem refers to even numbers n greater than 2. Hence, quite obviously, we are interested in the cases $n = 4, 6, 8, \dots$. For all these n s, we now want to explicitly build 3-regular graphs. Well, maybe not for all of them, since it is claimed that there are infinitely many such graphs. Nevertheless, we would like to start with small n and see, while constructing such graphs, whether there is an obvious construction rule that we can generalize.

For $n = 4$, a graph that fulfills the properties of the theorem can be quite easily given.

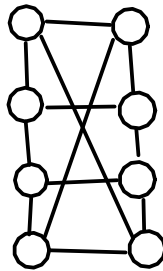


It is simply a fully connected graph with four nodes. We then start to extend the idea to $n = 6$:

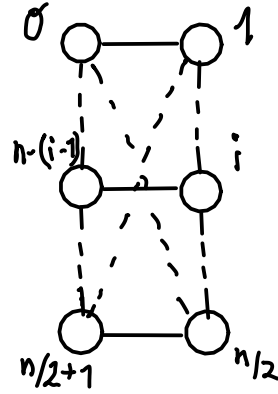


The top left graph would again be the fully connected graph. However, by looking into the details of this graph, we see that the requirement of a node degree of 3 is not fulfilled. Hence, this simple generalization is not possible. The top right graph is a next attempt. Here, we connect all nodes in a circle-like way. However, still, this graph has only a fixed node degree of 2 for all nodes. The bottom left graph is a refinement of this idea. We add edges, such that all nodes are horizontally and vertically connected and fix the degree of the “corner nodes” by diagonal edges. This is a 3-regular graph.

For $n = 8$ we try to replicate the last idea that we had for $n = 6$. That is, we add another pair of nodes, connect all nodes horizontally and vertically and fix the node degree for the corner nodes:



The above resulting graph is quite obviously a 3 regular graph. It seems like we have found a general rule on how to extend the previous constructions to general n . Let us give it a try:



In the schematic description of the graph (above), we enumerate nodes from 0 to $n - 1$. Node pairs are stacked on top of each other and we connect all nodes horizontally and vertically. Moreover, we add the two diagonal edges. We can formalized that graphical description by giving the graph

$$G = (V, E), \quad V = \{0, \dots, n - 1\},$$

with edges

$$\begin{aligned} E = & \{ \{i, i + 1\} | \forall i \text{ such that } 0 \leq i \leq n - 2 \} \cup \{ \{n - 1, 0\} \} \\ & \cup \{ \{i, n - (i - 1)\} | \forall i \text{ such that } 1 < i < n/2 \} \\ & \cup \{ \{0, n/2\}, \{1, n/2 + 1\} \}. \end{aligned}$$

It should be noted here that we are not done yet, with the proof. Inded, we only developed the proof idea. \triangle

Next, we actually carry out the proof.

Proof

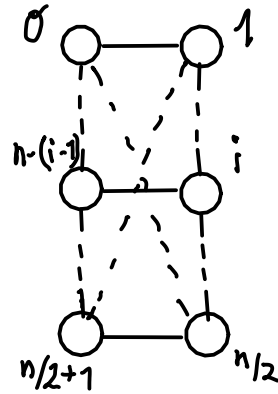
Let n be an even number greater than 2. We proof the statement by construction of a general graph $G = (V, E)$ with n nodes that fulfills the properties. The graph is given as

$$G = (V, E), \quad V = \{0, \dots, n - 1\},$$

with edges

$$\begin{aligned} E = & \{ \{i, i + 1\} | \forall i \text{ such that } 0 \leq i \leq n - 2 \} \cup \{ \{n - 1, 0\} \} \\ & \cup \{ \{i, n - (i - 1)\} | \forall i \text{ such that } 1 < i < n/2 \} \\ & \cup \{ \{0, n/2\}, \{1, n/2 + 1\} \}. \end{aligned}$$

We imagine to stack the nodes pair-wise on top of each other and number them clock-wise starting from the top left node. Thereby we obtain the following general graph:



The first and second set in E are all outer edges. The third set contains the inner horizontal edges and the last set has the two diagonal edges.

We briefly analyze the constructed graph and observe that all inner pairs (i.e. those that are not in the top or bottom row) will always have edges to the top, bottom and left/right. Hence, they have a degree of 3. Moreover, the nodes in the top and bottom row have edges to the left/right and top/bottom and one additional diagonal edge attached. Therefore, they also have a degree of 3. As a result, G is an 3-regular graph of size n . \square

Remark As the construction in this proof has been rather easy and also had a very simple to interpret visual representation, we could briefly argue that the construction is correct. In more complex cases, an additional proof of the claim that the construction fulfills the properties, e.g. by a proof by induction over the size of the graph, might become necessary. \triangle

Remark An interested reader might further ask herself/himself whether the statement still holds, if we make small changes. What about starting from $n = 2$? What about odd number of nodes? Sometimes such thoughts help to better get a feeling of the given object of interest. \triangle

0.2.2 Proof by contradiction

Next, we would like to discuss the concept of a proof by contradiction. This type of proofs constructs the argument by first assuming that the opposite of the (to be proven) statement is correct. Starting from this assumption, several steps of conclusions are made until a point at which we end up in a self-contradictory statement. If this contradiction is found, it becomes clear that the assumption was wrong, hence the original statement was true.

We pick a well-known very simple statement to give an example for a proof by contra-

diction.

Theorem (Example) $\sqrt{2}$ is irrational.

Proof:

We carry out a proof by contradiction. Therefore we assume that $\sqrt{2}$ is rational, thus we immediately conclude

$$\sqrt{2} = \frac{p}{q} \quad p, q \in \mathbb{Z}.$$

After reducing the fraction, we obtain

$$\sqrt{2} = \frac{m}{n} \quad m, n \in \mathbb{Z}$$

such that at least one of m, n is odd. Next we multiply the equation by n and obtain

$$n\sqrt{2} = m. \quad (0.1)$$

A further squaring gives

$$2n^2 = m^2,$$

hence m^2 is even. Since squares of odd numbers are odd, also m has to be even. As a consequence, we can write it as

$$m = 2k \quad k \in \mathbb{Z}.$$

Now, we substitute $2k$ for m in equation (0.1) and obtain

$$2n^2 = (2k)^2 = 4k^2$$

and after division by 2

$$n^2 = 2k.$$

Consequently n^2 is even, so is n . This means that n, m are both even, in contradiction to the observation that at least one of them has to be odd. \square

0.2.3 Proof by induction

We conclude this section by discussing proofs by induction. This proof type is used to prove statements of the form

$$\text{For all } i \in \mathbb{N}, i \geq i_0, \text{ it holds } \mathcal{P}(i).$$

where $\mathcal{P}(i)$ is some statement or property that depends on i .

We start by giving the general structure of such proofs:

Remark Proof structure

Basis (“ $i = i_0$ ”)

(A proof is given that shows that it holds $\mathcal{P}(i_0)$)

Induction step (“ $i \rightarrow i + 1$ ”)

(It is assumed that the statement $\mathcal{P}(i)$ holds.¹⁾ Using this assumption it is shown that $\mathcal{P}(i + 1)$ is true.) □

△

Finally, we give a very simple sum formula, for which we present an exemplary induction proof.

Theorem (Example) For each $n \in \mathbb{N}$ with $n \geq 1$, it holds

$$\sum_{i=1}^n i = n(n+1)/2$$

Proof

We prove this statement by induction over n .

$n = 1$:

$$1 = \sum_{i=1}^1 i = 1(1+1)/2 = 1$$

$n \rightarrow n + 1$:

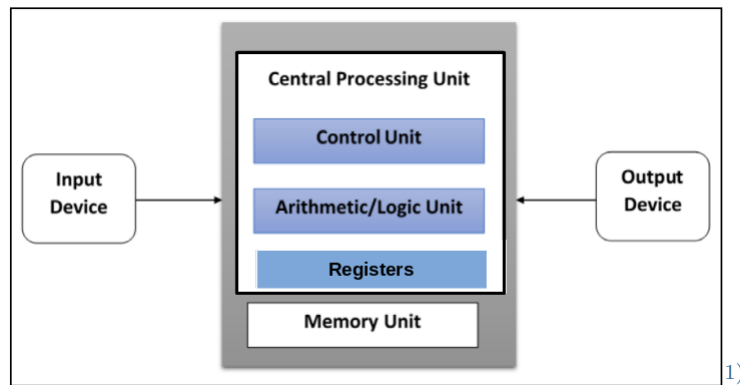
$$\begin{aligned} \sum_{i=1}^{n+1} i &= \left(\sum_{i=1}^n i \right) + (n+1) \\ &\stackrel{\text{(I.H.)}}{=} n(n+1)/2 + (n+1) \\ &= \frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)((n+1)+1)}{2} \end{aligned}$$

□

¹⁾This is called the *induction hypothesis*.

1 Regular languages

In theory of computation we are interested to model, characterize and analyze “computing” as we know it from practical applications. Our nowadays computers are based on the *von Neumann computer architecture*.



In this architecture, we have many complex components ranging from *input / output devices* over *memory* to *control* and *processing units*. Ideally, we would want to start modeling “computing” using this architecture. However, it has been recognized that such a complex model is not ideal as an entry point to theoretical considerations.²⁾

Therefore, we will start with a fairly simple compute model in which our “machine” gets the input as a string of symbols, has no memory, and generates an output that is either *accept* or *reject*. While this sounds not very interesting, we will see that indeed this model is already relatively powerful. We will call this just described “machine” *finite automaton*. A finite automaton thus can accept or reject strings. All strings that are accepted by a fixed finite automaton will form a *language*³⁾ and all languages that are *recognized* by *some* finite automaton will be called *regular languages*. A regular language is one type of *formal language*. We will see other formal languages later.

1.1 Finite automata

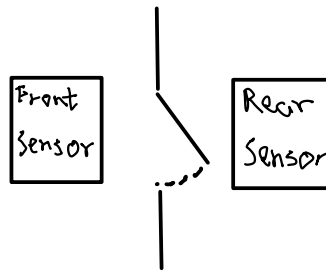
So far, we have learned that finite automata (FA) will have strings as inputs and somehow “magically” create an output that is *accept* or *reject*. But how do they work internally? We would like to introduce the computing methodology of an FA by comparing it to a very simple controller in a technical device. For the sake of this introduction, we try to model a controller for an automatic door.

¹⁾This figure has been taken from [1].

²⁾Later, we will see that the model of *Random Access Machines* comes close to the von Neumann architecture.

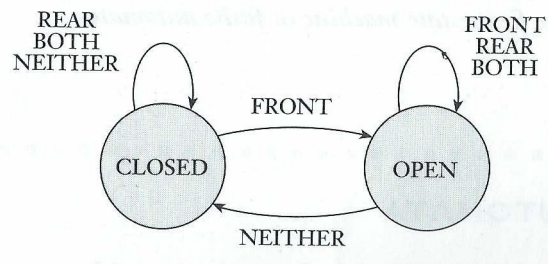
³⁾A reader that is not familiar with *languages* should reconsider the previous chapter.

Example 1.1 (Automatic door) We model a controller for an automatic door within the following situation:



In the above setup, the automatic door is supposed to open itself, as soon as a person is in the region of the front sensor. At the same time, the controller has to make sure that the sliding door is not going to hit someone in the region of the rear sensor. Certainly, the door should also close itself, if no one is around or can be hit.

The controller will operate based on a sequence of sensor signals that indicate whether the front, the rear, both or no sensor is signalling that a person is present. Internally, the controller model will have two states, OPEN and CLOSED, and the different signals will make the controller either stay in a given state or switch to the other state. We can visualize the necessary behavior of the controller by the following *state transition diagram*.



Actually, the diagram is a labeled, directed graph in which the nodes correspond to states and the edges with their labels indicate the signals that will lead to a transition among the states. Quite obviously, if the door is in the CLOSED state, it is supposed to stay closed, as long as no one arrives or the rear sensor (or both sensors) indicates that the door should not be opened. However, if the rear sensor is gives no signal and the front sensor indicates a person approaching, the CLOSED state is changed to the OPEN state. Similar suitable conditions hold for the OPEN state.

Another way of giving the exact same information is by the below table.

		input signal			
state		NEITHER	FRONT	REAR	BOTH
		CLOSED	OPEN	CLOSED	CLOSED
	OPEN	CLOSED	OPEN	OPEN	OPEN

Nevertheless, the state transition diagram is a visually more helpful way to express the situation. \triangle

As we will see, finite automata will internally work very similar to the just given door controller model.

Let us start with our first formal definition:

Definition 1.1 (Finite automaton) A **finite automaton** (FA) M is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where

1. Q is a finite set called the **states**,
2. Σ is a finite set called the **alphabet**,
3. $\delta : Q \times \Sigma \rightarrow Q$ is the **transition function**,
4. $q_0 \in Q$ is the **start state**, and
5. $F \subseteq Q$ is the **set of accept states / final states**.

In theory of computer science and other fields, it is quite common to characterize more complex theoretical objects via a tuple of several defining components. The reader should have seen this before for graphs, which are characterized as a tuple $G = (V, E)$, containing a set of nodes V and a set of edges E . FAs are given via three finite sets, a *state* and a transition function.

Remark (Interpretation of the components of an FA) We would like to reconsider the example of the door controller to better understand the different components of an FA. A finite automaton has *states*. These pretty much exactly correspond to the states that we have seen in the door controller. As we will see later, these states somewhat encode knowledge, and thereby replace memory that we know from more elaborated compute models.

In case of the door controller example, we had a sequence of different signals that was an input to the controller. In case of FAs, we can use the *alphabet* to explicitly define the objects from which we form the input. For the door controller, the alphabet would have been $\{\text{FRONT}, \text{REAR}, \text{BOTH}, \text{NEITHER}\}$. FAs take as input finite sequences of elements of an alphabet, i.e. *strings*.

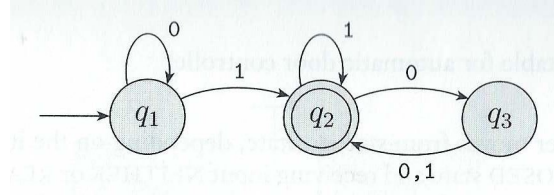
A finite automaton has a transition function that maps a given state and a given (input) symbol to a new state. This corresponds to the transitions that we have seen in the controller example. Equivalently, we will also be able to describe a transition function via a *state transition diagram* (STD) or a *state transition table*.

In addition to the controller example, we further see a *start state* in the FA definition. This state indicates the initial state in which the FA is, before the first string is read. Moreover, the set of *accept states* defines the output of the FA: Whenever the automaton is in one of the accept states when the last letter of the input string has been read, it

will *accept* the string. Otherwise it will reject the string. \triangle

After all these theoretical considerations, we now come to a first example of an FA:

Example 1.2 We initially, visually describe the FA by an STD:



This STD has two additional features over the STD from the door controller example. First, we have the arrow pointing from the left to node q_1 . This arrow indicates the start state. Moreover, node q_2 has a double-circle as outer boundary. This indicates that this node is in the set of accept states.

Let us pick the example input string 1101 on which we will run the FA: Initially the FA is in state q_1 . When reading the first 1, it switches to state q_2 . Reading another 1, it stays in state q_2 . Then, because of reading a 0, it switches to state q_3 . The last input symbol moves it finally back to state q_2 . Since q_2 is an accept state, the FA accepts the input string 1101. Otherwise, it would have rejected it.

While the STD is a meaningful visualization of an FA, we should still formalize the description of the FA via its original definition. Hence, we here discuss the FA

$$M_1 = (Q, \Sigma, \delta, q_1, F)$$

with the states $Q = \{q_1, q_2, q_3\}$, the alphabet $\Sigma = \{0, 1\}$, the transition function δ , described by the *state transition table*

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

the start state q_1 and the set of accept states $F = \{q_2\}$. \triangle

The previous definition for FAs has only characterized the “shape” of it but not the “computing” mechanism. Until now, we have motivated this mechanism, but we did not formally define it. This is done now by defining the formal conditions under which an FA accepts or rejects a string.

Definition 1.2 (Strings accepted by M) Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and $w = w_1w_2 \cdots w_n$ be a string over alphabet Σ .

M **accepts** w if there exists a sequence of states r_0, r_1, \dots, r_n , such that all following three conditions hold:

1. $r_0 = q_0$ (M starts in start state.)
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \dots, n-1$
(State change follows transition function.)
3. $r_n \in F$ (M ends up in accept state)

If M does not accept w , it **rejects** it.

Example 1.3 Let us reconsider the previous example, where we had an input string $w = 1101$. We informally stated that this string is accepted by M_1 . In the course of reading string w , the FA traverses the nodes

$$q_1, q_2, q_2, q_3, q_2.$$

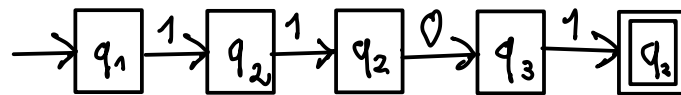
These nodes form the sequence of nodes r_0, r_1, \dots, r_n that is discussed in Definition 1.2. Indeed, the first node q_1 is the start state (corresponding to condition 1). Then, the consecutive nodes are reached by reading string w and following the rules of the transition function (corresponding to condition 2). Finally, the last node in that sequence is q_2 , which is in the set of accept states (corresponding to condition 3). Since all three conditions are fulfilled, the string w is accepted. \triangle

Remark (Computations of FAs) In some sense, Definition 1.2 introduces the concept of a *computation* that is carried out. Specifically a sequence of states for which the first two conditions of that definition hold, could be called a *computation of M on w* . The third condition then decides, whether it is an accepting or rejecting computation.

While a *computation* is not a classical term in FA theory, we would like to stick to this notion, as it will help us to describe some technical aspects of FAs in an easier way. In our previous example, we thus had the computation

$$c = q_1, q_2, q_2, q_3, q_2.$$

of M_1 on 1101. Moreover, we would like to introduce an appropriate visualization for it. The just given example can hence be visualized by



In the visualization, the states are not given as circles but as boxes to indicate that this is not an STD but a sequence of states that follows the first two conditions of Definition 1.2. We finish this remark by formalizing the definition of a computation:

Definition 1.3 (Computation of an FA) Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and $w = w_1w_2 \cdots w_n$ be a string over alphabet Σ .

A **computation of M on w** is a sequence of states $c = r_0, r_1, \dots, r_n$, such that the following two conditions hold:

1. $r_0 = q_0$ (M starts in start state.)
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \dots, n-1$
(State change follows transition function.)

We call c an **accepting computation**, if $r_n \in F$, otherwise we call it a **rejecting computation**.

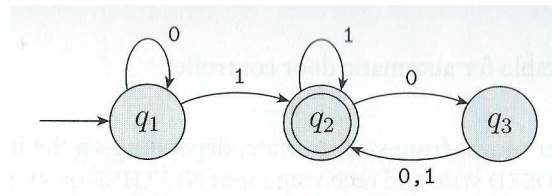
△

As mentioned before, a given FA implicitly defines a language:

Definition 1.4 (Language of machine M) Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton. The **language of machine M** $L(M)$ is the set of all strings that are accepted by M . We say: M **recognizes** $L(M)$.

Remark (Terminology) We note that strings are *accepted* by FAs, while languages are *recognized*. It is advisable to avoid to mix these terms up. △

Example 1.4 We continue our example for FA M_1 and analyze the language $L(M)$ that is recognized by this machine. As a reminder, we considered the FA with the following STD:



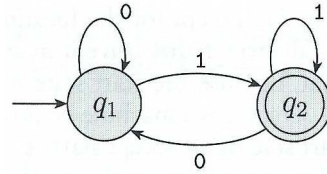
In this analysis, we have to find all input strings that will bring the FA to the final state q_2 : If we only consider the first two nodes (from left to right), strings that can have an arbitrary number of 0s and at least one 1 will for sure be accepted by the FA. Further interpreting the additional accepted strings due to the third node requires a bit of practicing. Basically, arbitrary sequences of 0s and 1s can be appended, as long as after the last 1, an even number of 0 (including zero 0s) is given. Therefore the language of machine M_1 is

$$L(M_1) = \{w \mid w \text{ contains at least one 1 and an even number of 0s follow the last 1}\}.$$

△

We carry out our a second (even simpler) example of describing and analyzing an FA:

Example 1.5 We are given the following STD:



Formally, it corresponds to the FA

$$M_2 = (\{q_1, q_2\}, \{0, 1\}, \delta, q_1, \{q_2\}),$$

with the transition function given by the state transition table

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

For an input string $w = 1101$ it will traverse the states q_1, q_2, q_2, q_1, q_2 , finally reaching the accept state q_2 . Therefore, the string is accepted.

Alternatively, we use input string 110 . In this case the states q_1, q_2, q_2 and q_1 are traversed, leading to a rejection of the string.

By briefly analysing the STD, it becomes apparent that the language of all strings (over $\{0, 1\}$) that end with a 1 is recognized by M_2 , i.e.

$$L(M_2) = \{w | w \text{ ends in a } 1\}.$$

△

Remark If an FA M accepts no string, it still recognizes the empty language \emptyset . △

As motivated before, we classify all languages recognized by a finite automaton as *regular languages*.

Definition 1.5 (Regular language) A language is called a **regular language** if some finite automaton recognizes it.

Obviously, all languages discussed, so far, are regular languages.

Until now, we started from an FA and identified its language. Now, we start from a given (regular) language and want to build an FA that recognizes it.

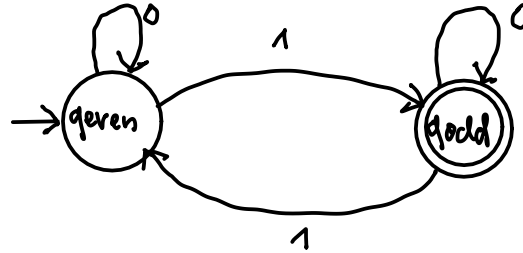
Example 1.6 Let the regular language

$$A = \{w \text{ string over } \{0, 1\} | w \text{ contains an odd number of } 1\text{s}\}$$

be given.⁴⁾ We are looking for an FA that recognizes this language.

The FA that we will build needs to recall whether the number of 1s read so far is even or odd. On a computer that follows the von Neumann architecture, we would use the memory and store and update this information in a variable. This is not possible in FAs. Instead, we use the *states* to “store” information. Hence we use states q_{even} and q_{odd} to indicate whether an even or odd number of 1s has been read. On reading a 1 the FA has to switch to the opposite state. If a 0 is read, it stays in the current state.

The resulting FA can be described using the below STD:



△

As initially stated, theory of computation aims at modelling, characterizing and analyzing “computing”. So far, we have modeled our first simple “machine”, i.e. an FA, for computing. That machine model implicitly defines a class of languages, i.e. regular languages. Our first formal statement will characterize properties of these regular languages with regard to some operations.

Let us first introduce these operations:

Definition 1.6 (Regular operations) Let A and B be languages. We define the regular operations **union**, **concatenation** and **star** as follows:

- **Union:** $A \cup B := \{x \mid x \in A \text{ or } x \in B\}$
- **Concatenation:** $A \circ B := \{xy \mid x \in A \text{ and } y \in B\}$.
- **Star:** $A^* := \{x_1x_2 \dots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$.

Example 1.7 Let $\Sigma = \{a, b, \dots, z\}$ be an alphabet. We are given two languages $A = \{good, bad\}$ and $B = \{day, night\}$. Using the just introduced regular operations, we can e.g. build the following new languages:

- $A \cup B = \{good, bad, day, night\}$,
- $A \circ B = \{goodday, goodnight, badday, badnight\}$,
- $A^* = \{\varepsilon, good, bad, goodgood, goodbad, badgood, badbad, goodgoodgood, goodgoodbad, \dots\}$.

△

⁴⁾How can we know that this language is regular? Well, with some practicing, one will develop an eye for this.

Indeed, as we will show, regular languages are *closed under the above regular operations*. What does that mean? It means that whenever we pick regular languages and apply the above operations to these languages the resulting languages will again be regular languages.

While from the first reading, this statement does not seem to be very special, it gives us a good sense of how we might potentially construct regular languages and what is the expressive power of formal languages.

We will individually state and prove this closure property for all three regular operations. Let us start with the union:

Theorem 1.1 The class of regular languages is closed under the union operation. In other words, if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.

In theory of computing, we will prove all such statements. Since this is our first proof in this field, we will try to very slowly approach the statement that needs to be proven:

Remark (Proof idea) So what are we given and what do we have to show? We are given two regular languages A_1 and A_2 and need to show that their union $A_1 \cup A_2$ is also a regular language. However, with this information, we do not have a starting point for the proof, yet. For now, we only know that regular languages are languages recognized by finite automata. Let us use that information.

Since A_1 and A_2 are regular languages, there exist two FAs M_1, M_2 with

$$M_1 = (S, \Sigma, \delta_1, s_0, F_1), \quad M_2 = (T, \Sigma, \delta_2, t_0, F_2)$$

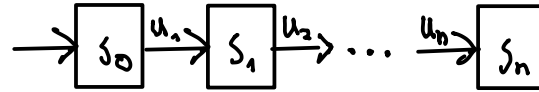
that recognize these languages, i.e. $L(M_1) = A_1$, $L(M_2) = A_2$. We have been lazy here and assumed that A_1 and A_2 are languages over the same alphabet.⁵⁾ Now, we need to show that, given M_1 and M_2 , there exists a third FA $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes the language $A_1 \cup A_2$.

Often, the easiest way to show existence of an object is by explicitly building the object and showing that it obeys the required properties. Hence, given FAs M_1 and M_2 we need to build FA M , such that the latter recognizes the union of the languages that are recognized by the first two FAs.

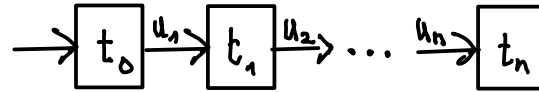
This seems to be a clear objective. However, how to approach such a proof? Here, we have to dig a bit deeper into the definitions that we have seen so far. According to Definition 1.6, the union operation between two languages A and B builds a new language by combining the individual members of that language in such a way that a

⁵⁾Indeed, we could have easily fixed this issue by merging two potentially non-equal alphabets for A_1 and A_2 into a bigger new alphabet. That new alphabet would then be used in this proof. Due to this easily possible fix, we could have also stated that “Without loss of generality, we assume that A_1 and A_2 have the same alphabet”.

string is contained in the new language if it has been contained in A or⁶⁾ B . Translated to our setting, this implies that our new automaton M has to accept a string u if this string would be accepted by M_1 or M_2 . According to Definitions 1.2 and 1.3, this means that there exists for $u \in A_1$ an accepting computation c_1



by M_1 or for $u \in A_2$ and accepting computation c_2



by M_2 . As a result, we need to build a new FA M that has an accepting computation for u , if M_1 or M_2 has an accepting computation for it.

We are now at the core of what we need to do in the proof. So now, we need the actual *proof idea*. Finding such an idea has a lot to do with practice, looking at similar proofs in the field and also with some intuition. So the reader should not be surprised if she/he would not immediately come up with a good idea or would need some time to develop it.

The proof idea is as follows: We build a new FA M that jointly simulates FAs M_1 and M_2 . This means that all of its computations on an input string jointly follow the computation paths of M_1 and M_2 and accept if one of the FAs would accept. Hence, a computation of M on string u would have to look like this:



The “formalization trick” that we will use is the introduction of new states in M that are pairs of all states of M_1 and M_2 , thus we will have states (s, t) and the set of states will be $Q = S \times T$. A transition between two such a pair states (s, t) , (s', t') for given input symbol u_i is carried out, whenever M_1 would go from s to s' and M_2 would go

⁶⁾By *or*, we indeed mean the “Boolean or” and not the “exclusive or” that is often referred to in colloquial English. In colloquial English, we would have to say that it is contained in A or B or in both of them.

from t to t' for u_i . Subsequently, M accepts the “joint” computation, if, for the last state reached in the computation (s_n, t_n) , it holds that s_n is an accept state in M_1 or t_n is an accept state in M_2 . In this case, M indeed accepts all strings from $A_1 \cup A_2$. \triangle

Remark In the just given proof idea, we have quite intensively, and in a rather non-formal way, discussed how to arrive at the core of the statement that we need to prove. This was done to give first time readers a good entry point into proving such statements. Future proof discussions will be shorter and will more immediately touch the core idea. \triangle

With all the intuition we gained so far, we will now do the formal proof.

Proof

Let A_1 and A_2 be regular languages. Without loss of generality, we assume that they are given over the same alphabet Σ . Since they are regular, we know that there exist FAs M_1, M_2 such that

$$A_1 = L(M_1), \quad M_1 = (S, \Sigma, \delta_1, s_0, F_1),$$

$$A_2 = L(M_2), \quad M_2 = (T, \Sigma, \delta_2, t_0, F_2).$$

To show that $A_1 \cup A_2$ is regular, we construct, given M_1 and M_2 , a new FA M such that

$$L(M) = A_1 \cup A_2, \quad M = (Q, \Sigma, \delta, q_0, F).$$

The individual components of M are

1. the states $Q = S \times T$,
2. the transition function δ with

$$\delta((s, t), a) = (\delta_1(s, a), \delta_2(t, a)) \quad \forall s \in S, t \in T, a \in \Sigma,$$

3. the start state $q_0 = (s_0, t_0)$, and
4. accept states

$$F = \{(s, t) | s \in F_1 \text{ or } t \in F_2\}.$$

The new FA simultaneously carries out the computation by M_1 and M_2 on a given input string. If M_1 or M_2 accepts an input string, M accepts it. Thereby, it recognizes the union of the recognized languages of both machines. \square

The next statement on closure with respect to regular operations would be

Theorem 1.2 The class of regular languages is closed under the concatenation operation. In other words, if A_1 and A_2 are regular languages, then $A_1 \circ A_2$ is regular.

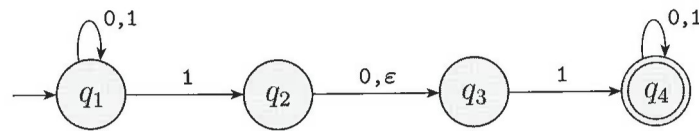
A proof for this statement is technically feasible using (deterministic) finite automata. However, it would be very involved. Interestingly, there is another machine model, *nondeterministic finite automata*, which will allow us to find a much simpler proof for the above theorem and for the theorem on the closure with respect to the star operation. Therefore, we will first use the next section to introduce this other machine model. Afterwards, we will come back to these closure theorems.

1.2 Nondeterministic finite automata

In the previous section, we got to know our first machine or computing model, the *finite automaton*. Finite automata carry out computations in a *deterministic* way. We know deterministic computation from the von Neumann architecture and from all classical (sort, search, etc.) algorithms. In context of an FA that is currently in state q , a *deterministic* computation means that an input symbol a leads to an exactly pre-defined transition to the one and only possible successor state q' (thus $q' = \delta(q, a)$). This also implies that the visualization of an FA computation looks like a chain, since no “branching” is possible. As a side-effect, FAs also require that, for each state, transitions have to be given for *all* symbols of the alphabet, thus an FA cannot “do nothing” for some input.

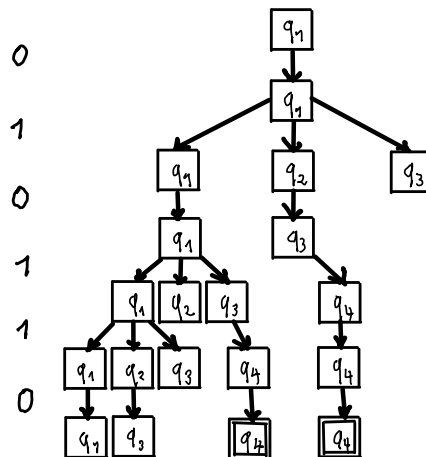
This section is concerned with *nondeterministic finite automata* (NFA). In contrast to deterministic FAs, nondeterministic FAs allow several successor states (or even none) for a given fixed input symbol. Let us immediately give an example to clarify the idea.

Example 1.8 The following STD describes a nondeterministic FA N_1 :



We notice a few differences over an STD for an FA: First of all, from a give state, there can be several outgoing edges with the same symbol (symbol 1 from state q_1). Moreover, from a given state, there can also be no outgoing edge for some symbol (symbol 1 from state q_2). Finally, we see the additional label ε (between states q_2 and q_3), which stands for the *empty symbol*.

The easiest interpretation of these changes can be given by considering computations of the corresponding NFA on the input string $w = 010110$. For a deterministic FA, the visualization of the computation would have been a chain. For a nondeterministic FA, it is a *tree*:



Let us carry out the computation for string $w = 010110$: NFA N_1 is initially in state q_1 . On input 0, we only have – as for FAs – one transition to the successor state q_1 . However, when reading 1 afterwards, we have two options for successor states, namely q_1 and q_2 . These two options imply that we split the computation into two branches, as seen in the computation tree. Actually, in the tree, we see three branches. This goes back to the ε label that is attached to the transition between q_2 and q_3 .⁷⁾ A transition with an ε label is carried out without reading a single symbol. Therefore, in addition to the second branch that goes to q_2 , we also add a computation branch that skips forward to q_3 . We however keep the branch towards q_2 , since we could read other characters there and therefore we could still do other operations from there.

As we “detached” three different computation branches from each other, we now have to follow all these three branches independently. Therefore, on input of 0, we are in state q_1 with one of the branches and have a transition to the same state. In the second branch, we are in state q_2 and read 0. This brings us to state q_3 . While in the third branch, we are in state q_3 and encounter another new situation. Here, we have *no* transition for the input 0. In the computing model of NFAs this leads to a drop of the computation branch. We no longer follow it.

We carry out a further read of a symbol. This time, it is again 1. Starting at the end of the computation branch that is in state q_1 , we, as seen before, split the computation in three branches towards states q_1 , q_2 and q_3 . Being in state q_3 with the other computation branch, we proceed to q_4 .

The whole process is continued until all input symbols are read. At this point (see computation tree), we still have four active computation branches being in states q_1 , q_3 , q_4 and q_4 . We notice that only q_4 is an accept state. So how to interpret that?

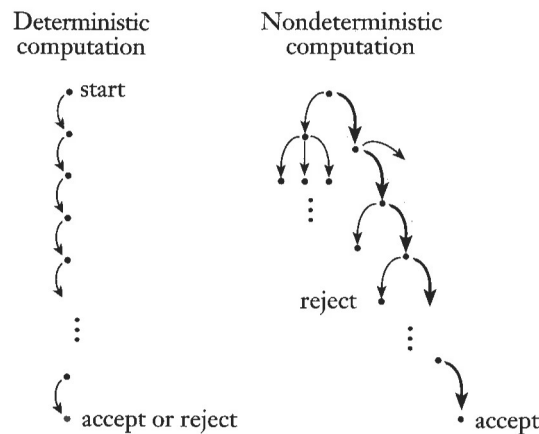
Indeed, a nondeterministic FA *accepts* a given input string, if there exists a computation branch that ends in an accept state, otherwise it rejects it. As we have two computation

⁷⁾Indeed, ε is not part of the underlying alphabet, but a “special character” to express an additional functionality of NFAs. In the context, in which it is used here, it stands for an *empty symbol*. In other cases, it stands for the *empty string*.

branches that end in an accept state – quite obviously – there exists a computation branch that ends in an accept state. Therefore, we accept the input string 010110. Studying further the computation tree, we also see that the shorter strings 01011 and 0101 would also be accepted, while 010, 01 and 0 would not be accepted. \triangle

We summarize the new “features” of NFAs, in comparison to (deterministic) FAs:

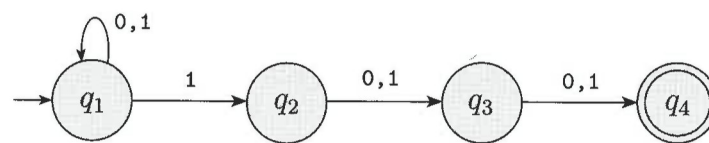
Remark (Comparison of NFAs to FAs) An NFA adds more flexibility to the concept of computation. Whenever an NFA has more than one transition for a given input symbol, we split the computation for each of these transitions. If, for a given state, there is no transition given for an input symbol, we drop the computation (branch). An ε -transition, where ε is not part of the alphabet, is done without reading a symbol and also leads to a split in the computation. Finally, all strings are accepted for which there exists a computation branch that ends in an accept state. These new concepts lead to a different style of computation. While deterministic FAs have computations that look like a chain, nondeterministic FAs have computations that look like a tree:



\triangle

We continue with a second example before we come to the actual definition of an NFA.

Example 1.9 We describe the NFA N_2 over the alphabet $\Sigma = \{0, 1\}$ by this STD:



Let us try to interpret, what kind of strings are accepted by this STD. Obviously only those computation branches lead to acceptance that end in state q_4 . Therefore, it makes sense to go from this state backwards. The transition between q_3 and q_4 implies, that all

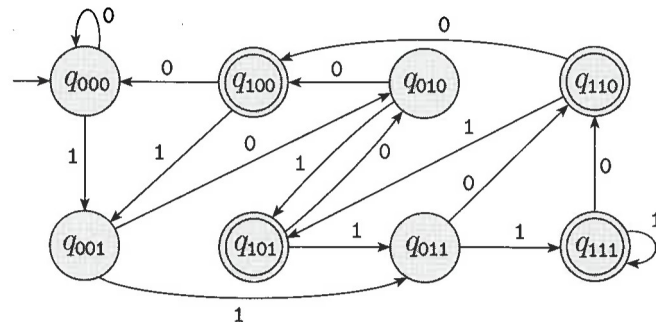
strings that are accepted have to have a zero or one as last character. The next transition (backwards, i.e. q_2 to q_3) implies that another arbitrary character is read. Due to the transition from the start state q_1 to q_2 that only allows the symbol 1, we identify our first set of accepted strings. These are all strings that start with a 1 and then have two arbitrary symbols over the alphabet.

However, we still ignored the one (nondeterministic) transition, that allows for having a loop in state q_1 for 0 and 1. This small loop is very powerful. It implies that we can read an arbitrary (potentially also empty) sequence of symbols before we enter the required pattern of a 1 and two arbitrary values. As a result, we identified the recognized language:

$$L(N_2) = \{w \text{ string over } \Sigma \mid w \text{ has a 1 at the third position from the end}\}.$$

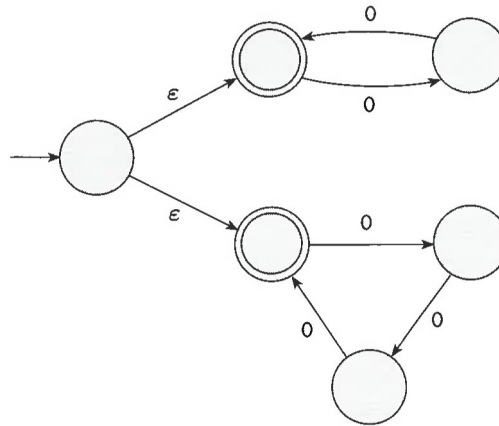
△

Remark (Expressivity of NFAs) NFAs, via their nondeterministic construction, seemingly allow to have an “easier” way to describe languages. If we were to construct a (deterministic) finite automaton that recognizes the language from the previous example, it would have to be more complex than the NFA that we have seen in that example. Indeed, the smallest (D)FA that recognizes the same language as N_2 is given by the following STD.



However, are NFAs more powerful than FAs, i.e. can they describe more (complex) languages? We will leave the answer to this question open as a small “cliff hanger” and come back to it, later. △

Example 1.10 (ε -transitions) Since ε -transitions are a rather surprising concept, which might not be immediately accessible to all readers, we give another very classical example for its use.



The above STD describes NFA N_3 . This NFA recognizes the union of two languages. How do we come to that observation? It is the double- ε -transition in the start state that implies this. Since ε is the empty symbol, the NFA can immediately match that with the input by not having read a single actual symbol. Therefore, N_3 immediately starts two independent computation branches that continue from the destination states of these transitions. Hence, we at the same time compute on the same input in the two top states and the three bottom states. We accept strings, if there exist accepting computation branches in one of the two simultaneous computations.

This construction should remind the reader of the proof for Theorem 1.1 on the closure of regular languages under the union operation. In that proof, we constructed exactly these two simultaneous calculation branches. However, the construction was much more involved, since we did not have the flexible concept of an ε -transition.

To conclude this example, we give the language recognized by N_3 as

$$L(N_3) = \{0^k | k = 2i, i \in \mathbb{N}_0\} \cup \{0^k | k = 3i, i \in \mathbb{N}_0\}.$$

△

After we have gathered some intuition, we now come to the formal definition of NFAs.

Definition 1.7 (Nondeterministic finite automaton) A **nondeterministic finite automaton** (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

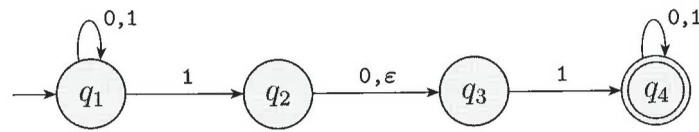
1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta : Q \times \Sigma_\varepsilon \rightarrow \mathcal{P}(Q)$ is the transition function, with $\Sigma_\varepsilon := \Sigma \cup \{\varepsilon\}$,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

Let us highlight the changes compared to FAs.

Remark (Comparison of NFA and FA definition) The definition of NFAs looks very similar to the one of (deterministic) FAs. We identically have states Q , an alphabet Σ , a start state q_0 and a set of accept states F . The difference of the formal description lies in the transition function. While it still takes as first argument a state, the second argument adds to the alphabet Σ the symbol ε , to allow for ε -transitions. More importantly, this input tuple is now mapped to a *set of states* and no longer to a *single state*.⁸⁾ This allows to have multiple successor states, or none (if the transition function maps the input tuple to an empty set). \triangle

We immediately apply this formalism to the NFA N_1 in the first example of this section.

Example 1.11 Recall that the NFA N_1 of Example 1.8 was given by the STD



Given the definition of an NFA as 5-tuple, N_1 is defined by

$$N_1 = (Q, \Sigma, \delta, q_1, F), \quad Q = \{q_1, q_2, q_3, q_4\}, \quad \Sigma = \{0, 1\}, \quad F = \{q_4\},$$

with transition function

δ	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	\emptyset
q_2	$\{q_3\}$	\emptyset	$\{q_3\}$
q_3	\emptyset	$\{q_4\}$	\emptyset
q_4	$\{q_4\}$	$\{q_4\}$	\emptyset

As discussed, the transition function maps the current state and the input symbol to a subset of the set of states.

With a little bit of practicing, we further figure out that N_1 recognizes the language

$$L(N_1) = \{w \mid w \text{ contains either } 101 \text{ or } 11 \text{ as substring}\}.$$

\triangle

While we already informally stated, when an NFA accepts a string, we still need the formal definition.

Definition 1.8 (Strings accepted by NFA N)

Let $N = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite automaton and w be a string over alphabet Σ .

⁸⁾ Recall here, that $P(Q)$ is the *power set* of Q , i.e. the set of all subsets of Q .

N **accepts** w if we can write w as $w = y_1 y_2 \dots y_m$, $y_i \in \Sigma_\varepsilon$ and if there exists a sequence of states r_0, r_1, \dots, r_m (in Q), such that all following three conditions hold:

1. $r_0 = q_0$ (N starts in start state.)
2. $r_{i+1} \in \delta(r_i, y_{i+1})$, for $i = 0, \dots, m-1$
(State change follows transition function.)
3. $r_m \in F$ (N ends up in accept state)

If N does not accept w , it **rejects** it.

It might be surprising that the definition for acceptance of a string by an NFA looks almost identical to the definition of the acceptance of a string by an FA. We should have a look into the details:

Remark (String acceptance definition of NFAs vs. FAs) Both definitions are almost identical, while describing very different concepts. One seemingly small but important change of the definition for NFAs in contrast to FAs is the changed view on the input string w . In FAs, the input string is fixed over the alphabet Σ . In NFAs, we start from the fixed input string w , however “derive” from it all potential input strings that can have an arbitrary (finite) number of ε s being added at arbitrary positions.⁹⁾ Hence, if we have the input string 101, we also consider simultaneously input strings

$$\varepsilon 101, 1\varepsilon 01, 10\varepsilon 1, 101\varepsilon, \varepsilon\varepsilon 101, \varepsilon 1\varepsilon 01, \dots$$

We need this to cover ε -transitions. The second change is in the second condition. Here, the successor has to be an *element of the set* of successor states. This allows to model multiple or no transitions. These are all changes.

Why can we have such a similar definition, if our computing model is that different? The reason lies in the way, how the definition is given. The definition requires the *existence* of a sequence of states, that fulfills these properties. For FAs, the sequence of states that the FA traverses for a fixed input string is always deterministic, i.e. there is one and only one sequence of these states (that fulfills conditions 1 and 2). For NFAs, we potentially have many such sequences in the tree of computation branches. But still, we only ask for the existence of *one* such sequence that fulfills all three properties. Consequently, asking for the *existence* of such a sequence (with small changes in the conditions) is a very flexible approach and needs to be carefully interpreted. \triangle

Similar to the case of FAs, we would like to introduce the non-traditional terminology of a *computation (branch)* for NFAs:

⁹⁾ Recall that ε here stands for the empty symbol, so indeed all these strings are identical.

Definition 1.9 (Computation branch of an NFA) Let $N = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite automaton and $w = w_1w_2 \cdots w_n$ be a string over alphabet Σ .

A **computation branch of N on w** allows for a modification of the input string w to a string $w = y_1y_2 \cdots y_m$, $y_i \in \Sigma_\epsilon$ and is a sequence of states $c = r_0, r_1, \dots, r_m$, such that the following two conditions hold:

1. $r_0 = q_0$ *(N starts in start state.)*
2. $r_{i+1} \in \delta(r_i, w_{i+1})$, for $i = 0, \dots, m-1$
(State change follows transition function.)

We call c an **accepting computation branch**, if $r_n \in F$, otherwise we call it a **rejecting computation branch**.

We have already seen that all computation branches for a fixed, given input string form a computation tree of N on w .

While we “recycled” the notation $L(N)$ of a language of an NFA from FAs before, we still need to define it properly:

Definition 1.10 (Language of NFA N) Let $N = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite automaton.

The **language of N** $L(N)$ is the set of all strings that are accepted by N . We say: N **recognizes** $L(N)$.

It is now time to come back to our “cliff hanger”. Previously, we asked ourselves whether the set of languages that are recognized by some NFA might be larger¹⁰⁾ than the set of languages recognized by some FA. Intuitively, we might answer “Yes.”, since NFAs seem to be a way more flexible approach to describe languages. However, surprisingly, this intuition is wrong:

Theorem 1.3 Every nondeterministic finite automaton has an equivalent deterministic finite automaton, i.e. there exists for every NFA N a DFA M such that $L(M) = L(N)$.

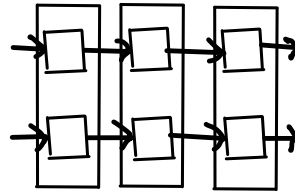
In other words, all languages that can be recognized by an NFA, can also be recognized by some FA. Hence NFAs are not “more powerful” than FAs, but indeed “equally powerful”. How can we prove this? A first answer is given by

¹⁰⁾ We do not discuss whether NFAs could recognize fewer languages than FAs, since, quite obviously, we can immediately write down an FA as an NFA. Therefore, all languages recognized by FAs are for sure recognized by NFAs.

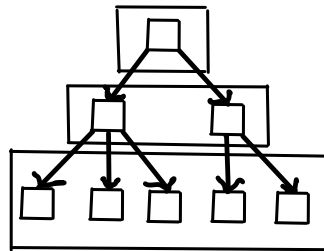
Remark (Proof idea) The proof requires to build for an arbitrary NFA N an FA M such that M recognizes the same language as N . Structurally, we had a similar task in the proof of Theorem 1.1. In that case, we had to build from two FAs a third FA with some properties. Here, we have to build an FA from an NFA. In some sense, the core proof idea even sounds similar:

The idea of the proof for Theorem 1.3 is to simulate the computation of N on a new FA M by simultaneously going through all computation branches of N .

How to do that? Recall from the proof of Theorem 1.1 that we simultaneously computed on both input FAs M_1 , M_2 by introducing states $Q_1 \times Q_2$ with appropriate transitions to achieve a computation of the form



From a visual perspective, we will now do the same, but will combine all states of one *level* of the computation tree into one state. Hence, a simultaneous calculation on all computation branches will have the form



With a little bit of intuition, we observe that the states of our to be constructed machine will not only be tuples of states of the old machine but *all subsets of the states of N* . Therefore the set of states of the new NFA will be $\mathcal{P}(Q)$, where Q is the set of states of N . We “connect” these “set states” via transitions that follow the transitions of the nondeterministic FA N . \triangle

Remark (Cost of the construction) Imagine we would measure the *cost* of the description of a language by the amount of states that we need to describe it by an FA. In that case, the just given proof idea should alert us. Indeed, we start from an NFA with a given amount of states and build an FA with exponentially larger number of states, as $|\mathcal{P}(Q)|$ is exponentially larger than $|Q|$.¹¹⁾ Most likely, the much lower *cost* of a representation of some languages by an NFA over an FA has been the basis for our intuition that NFAs are “more powerful” than FAs.

¹¹⁾ Certainly, there might in many cases be “cheaper” ways to give an equivalent FA for an NFA.

Luckily, we are only interested in the *class of recognized languages*, and not in any cost model. Still, we should keep these thoughts in mind, since we will later, in context of so-called *Turing machines*, be interested in *costs* in terms of *running time*. There, exactly this differentiation between costs based on a description via a deterministic or via a nondeterministic machine will give rise to the famous complexity classes P and NP . \triangle

After this side remark, we come to the actual proof:

Proof

We carry out a proof by construction: Let $N = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic FA with $A = L(N)$. We aim at constructing the deterministic FA $M = (Q', \Sigma, \delta', q'_0, F')$ such that $L(M) = A$.

First, we do a slightly simplified construction that ignores ε -transitions in N . Assuming this, we set $Q' = \mathcal{P}(Q)$ (following the proof idea). The definition of the transition function δ' follows the idea of making a successor state in M represent the union of all successor states of the given set of states R in N :

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a) \quad \text{for all } R \in Q', a \in \Sigma.$$

Effectively, we hence pick the state R from M , which represents several states in N . For each of these states in N , we look up the successor states. The union of all these successor states is the successor state in M . This corresponds to simultaneously traversing all computation branches of N .

Furthermore, we set the initial state as $q'_0 = \{q_0\}$ and obviously define the set of accept states as

$$F' = \{R \in Q' \mid R \text{ contains an accept state of } N\}.$$

Since with a computation of M we simultaneously traverse all calculation branches of N and accept if one of the branches, in the final step, reaches an accept of N , the FA M accepts a string, whenever it is accepted by N .

This would conclude the proof. However, for didactic reasons, we first skipped over the issue of ε -transitions. We now present the necessary extension of the previous construction to incorporate ε -transitions. To this end, we introduce the set

$$E(R) = \{q \in Q \mid \text{there exists a sequence } s_0, s_1, \dots, s_m \in Q, m \geq 0, \\ \text{s.th. } s_0 \in R, s_m = q, s_{i+1} \in \delta(s_i, \varepsilon) \text{ for } 0 \leq i \leq m-1\}.$$

It contains for a state R in M , which is a subset of the states of N , the set of all the states in N that can be reached from the set of states $R \subseteq Q$ by an arbitrary number of ε -transitions. Hence, in some sense, $E(R)$ is the state R grown by the neighborhood of all states that are somehow “reachable” by ε -transitions.

With this new definition, we construct a modified transition function

$$\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a)) \quad \text{for all } R \in Q', a \in \Sigma$$

and the modified start state $q'_0 = E(\{q_0\})$. This is sufficient to extend M to the case of ε -transitions. Hence, we have shown that

$$L(M) = A = L(N).$$

□

Theorem 1.3 immediately leads to

Corollary 1.1 A language is regular if and only if some nondeterministic finite automaton recognizes it.

Proof

“ \Leftarrow ”:

Let A be a language recognized by an NFA, thus there exists NFA N such that $L(N) = A$. According to Theorem 1.3, there exists an FA M with $L(M) = L(N) = A$, hence A is regular.

“ \Rightarrow ”:

Let A be a regular language. Consequently, there exists an FA M with $L(M) = A$. Since every FA can be written as an NFA, there also exists an NFA N such that $L(N) = L(M) = A$.

□

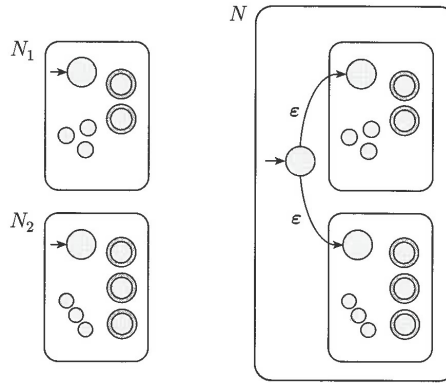
With Corollary 1.1, we learned something essential: Whenever we want to prove a statement on regular languages, we can use the representation via FAs *or* NFAs to carry out the proof. This brings us back to the end of the last section, where we started to discuss and prove statements on the closure of regular languages under regular operations.

Specifically, we already stated *and* proved

Theorem The class of regular languages is closed under the union operation. In other words, if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.

However, indeed, the use of NFAs makes the proof idea and construction much easier. Therefore, for the sake of training, we carry out the proof again, via NFAs.

Remark (Proof idea) In Example 1.10, we already noticed that the union of two languages can be easily described by NFAs via a starting state that is connected via two ε -transitions to the starting states of those NFAs that we want to combine. The visual idea of the proof is thus as follows:



We start from two NFAs N_1 and N_2 and build an NFA that recognizes the language $L(N_1) \cup L(N_2)$ by attaching their start states via ε -transitions to a new joint start state. \triangle

Proof

Let A_1 and A_2 be two arbitrary but fixed regular languages. According to Corollary 1.1, it is sufficient to build an NFA N that recognizes $A_1 \cup A_2$ in order to conclude the proof. Again following Corollary 1.1, we know that there exist NFAs

$$N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1), \quad L(N_1) = A_1,$$

$$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2), \quad L(N_2) = A_2.$$

We construct a new NFA $N = (Q, \Sigma, \delta, q_0, F)$ such that $L(N) = A_1 \cup A_2$. It has the states $Q = \{q_0\} \cup Q_1 \cup Q_2$, observing that q_0 is the new start state. Moreover, we define the transition function δ by

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & \text{if } q \in Q_1, \\ \delta_2(q, a) & \text{if } q \in Q_2, \\ \{q_1, q_2\} & \text{if } q = q_0 \text{ and } a = \varepsilon, \\ \emptyset & \text{if } q = q_0 \text{ and } a \neq \varepsilon, \end{cases} \quad \text{for all } q \in Q, a \in \Sigma_\varepsilon.$$

In other words, N contains both machines N_1 and N_2 . Moreover, we connect the new start state via ε -transitions to the old start states of N_1 and N_2 . Finally the union of the accept states of N_1 and N_2 are the new accept states of N , thus $F = F_1 \cup F_2$.

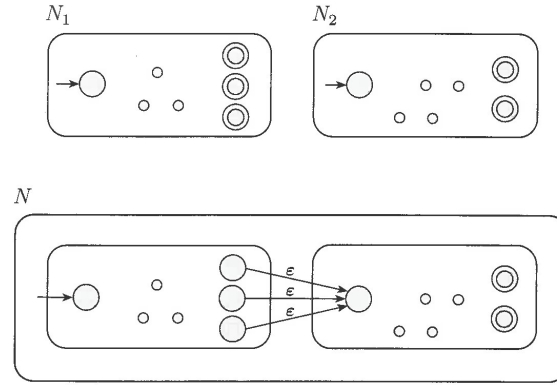
Quite obviously, N recognizes $L(N_1) \cup L(N_2)$. \square

We recall that we further already stated

Theorem The class of regular languages is closed under the concatenation operation. In other words, if A_1 and A_2 are regular languages, then $A_1 \circ A_2$ is regular.

However, we already indicated that it would be hard to prove this theorem using FAs. Due to Corollary 1.1, we can now use NFAs to carry out the proof:

Remark (Proof idea) The proof idea can be best expressed in a visual way:



Here N_1 and N_2 are the NFAs that recognize A_1 and A_2 , respectively. The main idea is to simply connect all accept states of the first NFA via an ε -transition to the start state of the second NFA. Rather obviously, this gives an NFA that recognizes the concatenation of A_1 and A_2 . \triangle

Proof

We carry out a proof by construction. Let A_1, A_2 be two arbitrary but fixed regular languages with their associated NFAs

$$N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1), \quad L(N_1) = A_1,$$

$$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2), \quad L(N_2) = A_2.$$

To prove the statement, we need to construct an NFA $N = (Q, \Sigma, \delta, q_0, F)$ such that $L(N) = A_1 \circ A_2$. We construct N as follows:

1. $Q = Q_1 \cup Q_2$,
2. for all $q \in Q, a \in \Sigma_\varepsilon$ it holds

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & \text{if } q \in Q_1 \setminus F_1, & (\text{main part of } N_1) \\ \delta_1(q, a) & \text{if } q \in F_1 \text{ and } a \neq \varepsilon, \\ \delta_1(q, a) \cup \{q_2\} & \text{if } q \in F_1 \text{ and } a = \varepsilon, & (\text{gluing together } N_1, N_2) \\ \delta_2(q, a) & \text{if } q \in Q_2, & (\text{main part of } N_2) \end{cases}$$

3. $q_0 = q_1$, and
4. $F = F_2$.

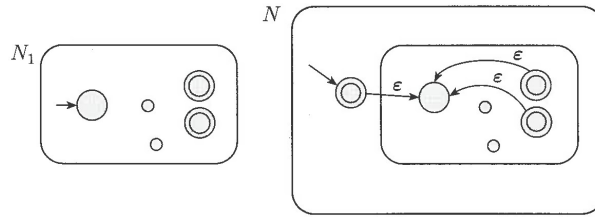
Accepting computations by N first follow transitions in N_1 to (old) accept states in N_1 . Then an ε -transition lets the computation continue in N_2 from the start state of N_2 . Finally, accepting computations have to reach the accept states of N_2 . This corresponds to accepting strings from $A_1 \circ A_2$. \square

The third and last statement that we need for the closure of regular languages under regular operations is as follows:

Theorem 1.4 The class of regular languages is closed under the star operation.

Remark (Proof idea) For an arbitrary but fixed regular language A , we need to show that A^* is still a regular language. We consider an NFA N_1 with $L(A) = A$. The proof idea is to convert this NFA into a new NFA N that recognizes A^* .

We again start with the visual proof idea:



In principle, we simply connect the accept states to the start state. This allows to generate arbitrary concatenations of strings in A . However, we also have to make sure that the empty string ε is part of the set of accepted strings. Therefore, we explicitly have to add a new start state that takes into consideration the empty string.¹²⁾ That new start state is connected via an ε -transition to the original start state. \triangle

Proof

Let A be an arbitrary but fixed regular language and $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ an NFA that recognizes A , i.e. $L(N_1) = A$. We construct the NFA $N = (Q, \Sigma, \delta, q_0, F)$ such that it recognizes A^* .

The new NFA N is given by

1. $Q = \{q_0\} \cup Q_1$,
2. for all $q \in Q, a \in \Sigma_\varepsilon$ it holds

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & \text{if } q \in Q_1 \setminus F_1, & (\text{main part of } N_1) \\ \delta_1(q, a) & \text{if } q \in F_1 \text{ and } a \neq \varepsilon, \\ \delta_1(q, a) \cup \{q_1\} & \text{if } q \in F_1 \text{ and } a = \varepsilon, & (\text{loop to old start state}) \\ \{q_1\} & \text{if } q = q_0 \text{ and } a = \varepsilon, & (\text{adding } q_0) \\ \emptyset & \text{if } q = q_0 \text{ and } a \neq \varepsilon, \text{ and} \end{cases}$$

3. $F = \{q_0\} \cup F_1$.

¹²⁾ Why is it not possible to simply add the old start state to the set of accept states in N ? The interested reader should sit down and develop a small example showing that this idea could lead to acceptance of strings in $L(N)$ that are not part of $L(N_1)^*$.

The newly constructed NFA N now also accepts the empty string and arbitrary concatenations of strings accepted by N_1 . This is the language A^* . \square

We finally have shown that regular languages are closed under regular operations. While this might be a nice structural statement, it will also have very practical implications, as we will see in the next section.

1.3 Regular Expressions

Previously, we had a first look at computing models. We would like to use this section to build up a connection of these computing models and their recognized languages to objects that are known by many computer scientists from their practical work: *regular expressions*. Regular expressions are used by Unix tools like `awk` or `grep`, in a programming language like *Perl* and in compilers. In all these cases, they are used to describe strings by patterns, i.e. languages.

In this section, we will show that the expressive power of regular expressions is equivalent to the expressive power of (N)FAs, hence the regular expressions describe regular languages.

We start with an example that shall intuitively (re-)introduce the notation of regular expressions.

Example 1.12 Let the regular expression

$$(0 \cup 1)0^*$$

be given. We characterize the individual components and derive the described language. The characterized language of $(0 \cup 1)$ is $\{0, 1\}$. Moreover, the characterized language of 0^* is $\{0\}^*$. Then, the notation $(0 \cup 1)0^*$ is the short form of $(0 \cup 1) \circ 0^*$, i.e. the concatenation of the two previous languages $\{0, 1\} \circ \{0\}^*$. As a result the characterized language of the regular expression is the language of all strings over $\{0, 1\}$ that start with 0 or 1 and are followed by an arbitrary number of zeros (including none). \triangle

Having gained that first intuition, we come up with the formal definition.

Definition 1.11 Let Σ be an alphabet. A **regular expression** (RE) R over the alphabet Σ describes a language $L(R)$ over Σ . It is inductively defined such that R is RE if R is

1. a for some $a \in \Sigma$, with $L(a) = \{a\}$
2. ε , with $L(\varepsilon) = \{\varepsilon\}$
3. \emptyset , with $L(\emptyset) = \emptyset$
4. $(R_1 \cup R_2)$ (R_1, R_2 REs), with $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$

5. $(R_1 \circ R_2)$ (R_1, R_2 REs), with $L(R_1 \circ R_2) = L(R_1) \circ L(R_2)$
6. (R_1^*) , (R_1 RE), with $L(R_1^*) = L(R_1)^*$

Parentheses in an RE can be omitted. Then, evaluation is done in the precedence order $*$, \circ , \cup . If clear from the context, the concatenation operator \circ does not have to be written down. Moreover, “ $R_1 = R_2$ ” means $L(R_1) = L(R_2)$.

The empty string ε and the empty language \emptyset may create some confusion. Nevertheless, we ignore this first and continue with an example:

Example 1.13 Let the alphabet $\Sigma = \{0, 1\}$ be given. The language described by the regular expression 0^*10^* is given by

$$L(0^*10^*) = \{w \in \Sigma^* \mid w \text{ has exactly a single } 1\}.$$

Moreover, the language described by the regular expression $(\Sigma\Sigma)^*$ is

$$L((\Sigma\Sigma)^*) = \{w \in \Sigma^* \mid w \text{ is string of even (and } 0) \text{ length}\}.$$

△

Remark (Warning about common mistakes) A common mistake of beginners in the field is to e.g. write

$$0^*10^* = \{w \in \Sigma^* \mid w \text{ has exactly a single } 1\}$$

to indicate the language described by 0^*10^* . This statement is *wrong*, as the object on the left-hand side of the equation is a regular expression, which is *not* a language. Only $L(0^*10^*)$ on the left-hand side is correct.

Similarly, later, we may want to describe *languages* in a quick way. Therefore, we use regular expressions. However, it is still crucial to write $L(R)$, where R is some regular expression, if we want to describe a *language*. Hence, writing R for a *language*, if R is a regular expression *describing* the language of interest, would be wrong. △

Now we come back to the empty string and the empty language and try to clarify some confusion upfront.

Remark Intuitive, it should be clear that the union of a language and the empty language gives the (first) language. Therefore, we have for a regular expression R

$$R \cup \emptyset = R.$$

Similarly, if we concatenate to some language the language containing only the empty string, we again get the original language, hence

$$R \circ \varepsilon = R.$$

However, building the union of a language with the language with only the empty string, which can be expressed by $R \cup \varepsilon$ does not necessarily give us R , i.e. the original language. A counter example is $R = \emptyset$ in that case, we have $L(R \cup \varepsilon) = \{\emptyset, \varepsilon\}$. Similarly, $R \circ \emptyset$ is not necessarily equal to R since, e.g. for $R = \emptyset$ we have $L(R \circ \emptyset) = \emptyset$. Therefore, some care has to be taken with empty strings and empty languages. \triangle

The main statement of this section is

Theorem 1.5 A language is regular if and only if some regular expression describes it.

Hence, we formalize our initially made informal statement that the expressive power of (N)FAs and regular expressions is the same.

The remainder of this section is concerned with the proof of the above theorem. As we are confronted with an equivalence statement (“if and only if”), we will split up the statement into Lemmata, giving the two necessary directions. One of these directions, will be easier to prove, the other will require quite some terminology and another automaton model. We start with the easier direction via

Lemma 1.1 If a language is described by a regular expression, then it is regular.

Proof

Let Σ be an alphabet and R be an arbitrary, but fixed RE over Σ . Our aim is to show that for the given RE R , we can build an NFA N that recognizes the language described by R , i.e. $L(N) = L(R)$. Due to Corollary 1.1, we then know that the language described by R is a regular language.

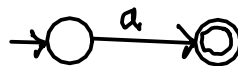
In Definition 1.11, a regular expression R has been inductively introduced from six elementary cases. We make use of this inductive construction. Indeed, if we are able to build for all six elementary cases corresponding NFAs, we have shown that for arbitrary R , the construction of an NFA recognizing $L(R)$ is possible. Hence, we go through all cases:

1. $R = a$ ($a \in \Sigma$)

Here, we have to build an NFA recognizing $L(R) = \{a\}$. This is easily done with the NFA $N_1 = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$ with

$$\delta(q, b) = \begin{cases} \{q_2\} & \text{if } q = q_1, b = a, \\ \emptyset & \text{else.} \end{cases}$$

Its STD is given by



and we see easily that $L(N_1) = L(R)$. ✓

2. $R = \epsilon$

We need to build an NFA for the language $L(R) = \{\epsilon\}$. The necessary NFA N_2 is given by the STD



and the formal description $N_2 = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$ with $\delta(q, b) = \emptyset$ for all $q \in Q$, $b \in \Sigma$. Obviously, we have $L(N_2) = L(R) = \{\epsilon\}$. ✓

3. $R = \emptyset$

Our aim is to build an NFA, recognizing the empty language. It is given via $N_3 = (\{q_1\}, \Sigma, \delta, q_1, \emptyset)$, with $\delta(q, b) = \emptyset$ for all $q \in Q$ and $b \in \Sigma$. Its STD is



We thus have $L(N_3) = L(R) = \emptyset$. ✓

4. $R = R_1 \cup R_2$

Here, want to build an NFA for the language $L(R)$. According to the definition of REs, we know that this language is given by $L(R_1) \cup L(R_2)$. Thereby, assuming inductively, that we have already built NFAs recognizing $L(R_1)$ and $L(R_2)$, we only need to build an NFA recognizing their union. However, this is already guaranteed by Theorem 1.1 (closure wrt. union operation). ✓

5. $R = R_1 \circ R_2$

This case follows the same idea as the previous case. Thus Theorem 1.2 (closure wrt. concatenation operation) guarantees the existence of an NFA N_5 such that $L(N_5) = L(R) = L(R_1) \circ L(R_2)$. ✓

6. $R = R_1^*$.

Here, we can use Theorem 1.4 (closure wrt. star operation) similar to the arguments before. ✓

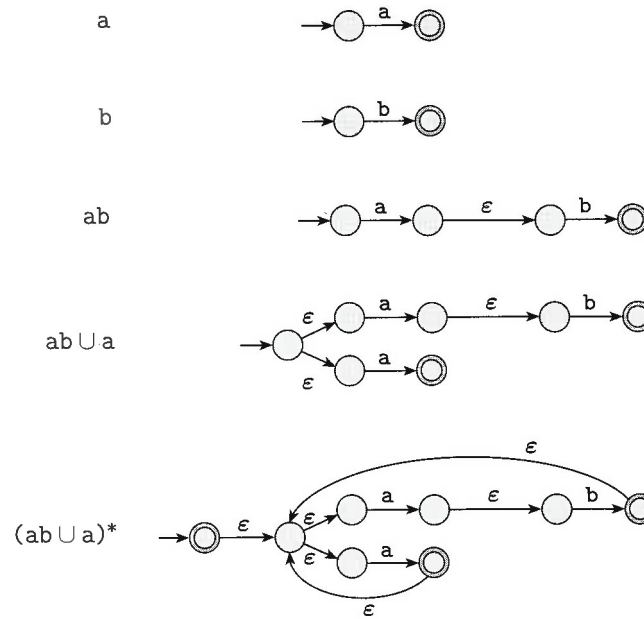
□

We give an example of the construction idea of the previous proof for a concrete regular expression.

Example 1.14 (Proof construction example) Let the alphabet $\Sigma = \{a, b\}$ be given. Let further the regular expression R with

$$R = (ab \cup a)^*$$

over Σ be given. We would like to find the NFA N that recognizes the language $L(R)$. To do that, we follow exactly the construction of the proof of Lemma 1.1. The inductive steps of this construction are as follows:



Please note that there might be “smaller” NFAs recognizing the language that is described by R . However, the above construction can always be made “automatically” out of a regular expression. \triangle

Now, we come to the way more involved direction of Theorem 1.5:

Lemma 1.2 If a language is regular, then it is described by a regular expression.

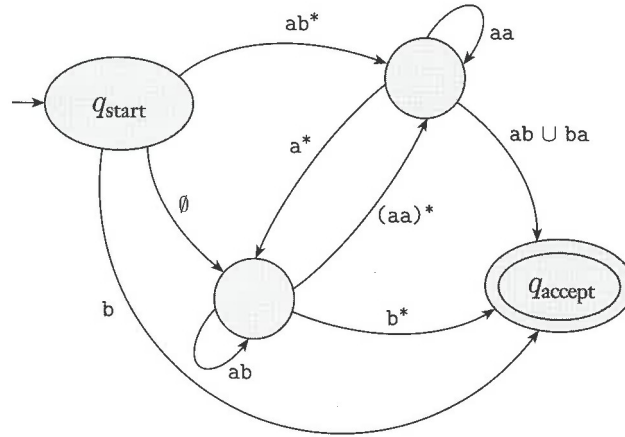
In order to prove this statement, we need another type of automaton, a *generalized nondeterministic finite automaton*.

Definition 1.12 A **generalized nondeterministic finite automaton** (GNFA), $(Q, \Sigma, \delta, q_{start}, q_{accept})$, is a 5-tuple, with

1. Q is the finite set of states,
2. Σ is the finite input alphabet,
3. $\delta : Q \times Q \rightarrow \mathcal{R}$ (\mathcal{R} : set of all REs over Σ), where $\delta(q, q_{start})$ and $\delta(q_{accept}, q)$ are undefined for all $q \in Q$.
4. $q_{start} \in Q \setminus \{q_{accept}\}$ is the start state, and
5. $q_{accept} \in Q \setminus \{q_{start}\}$ is the accept state.

We would like to explain this model via an example:

Example 1.15 For an alphabet $\Sigma = \{a, b\}$, we give the example of the following (STD for a) GNFA:



Let us discuss its characteristics: A GNFA is an NFA, such that we associate to each transition a *regular expression* instead of a symbol of the alphabet. The start state and the accept state are distinct. Moreover, we require that each state, which is not the start or the accept state, has transitions to all other states, including itself. For the start state, there exist outgoing transitions to all other states, but no incoming connections. At the same time, the accept state has only incoming connections that come from all states. These conditions are implemented by the slightly cryptic Definition 1.12.

How does the above GNFA operate on an input string? The GNFA can sequentially read several input symbols at a time as long as they match a regular expression along one of the transitions. Hence, the above GNFA would for $a, ab, abb, abbb$, etc. move from the start state to the top right state. If the further two letters are ab or ba it would continue to the accept state.

Thereby, not only a single string is accepted per computation branch but all strings that are in the language described by the regular expression that we obtain by concatenating the regular expressions along a q_{start} -to- q_{accept} path. \triangle

The above explanations and observations lead to the following definition of accepted strings:

Definition 1.13 A GNFA $G = (Q, \Sigma, \delta, q_{start}, q_{accept})$ **accepts** a string $w \in \Sigma^*$, if there exists a decomposition $w = w_1 w_2 \cdots w_k$ ($w_i \in \Sigma^*$) and a sequence of states q_0, q_1, \dots, q_k such that

1. $q_0 = q_{start}$ is the start state
2. $q_k = q_{accept}$ is the accept state, and
3. for each $i = 1, \dots, k$ we have $w_i \in L(R_i)$, where $R_i = \delta(q_{i-1}, q_i)$.

Otherwise w is **rejected**. $L(G)$ is the **language recognized by G** .

Remark While the extension to regular expressions instead of just symbols feels rather natural, some of the other conditions, like the requirements on the existence of transitions, look a bit special. These specific requirements allow us to have a slightly simpler path through the required proofs for Lemma 1.2. \triangle

We will now discuss the necessary steps to prove Lemma 1.2. Hence, we want to show that each regular language can be described by a regular expression:

Remark (Roadmap for the proof of Lemma 1.2) To prove Lemma 1.2, we start from a regular language A . By definition, we know that there exists a deterministic FA M , such that $L(M) = A$. We then do two steps. First, we convert that FA M to a GNFA G that recognizes the same language, i.e. $L(G) = L(M)$. Second, we convert the GNFA G to a regular expression R such that $L(R) = L(G)$. As soon as we have completed the second step, we are done with the proof. \triangle

The upcoming two lemmata (with proofs) will cover the just described steps.

Lemma 1.3 For each FA M , there is a GNFA G , such that $L(G) = L(M)$.

Proof

Let M be an arbitrary, but fixed FA with

$$M = (Q, \Sigma, \delta, q_0, F).$$

We construct the GNFA $G = (Q', \Sigma, q_{start}, q_{accept})$ such that $L(G) = L(M)$. Our construction starts from M and adds additional (replacing) start and accept states q_{start} , q_{accept} . State q_{start} is connected via ϵ -transition to the old start state q_0 . State q_{accept} gets incoming ϵ -transitions from all previous accept states. Moreover, in the new GNFA G we replace the old transition function in the FA M by a new one that replaces multiple symbols in Σ that lead to the same transition $q_1 \rightarrow q_2$ by the regular expression that describes the language containing exactly those symbols. Other transitions that are required by the GNFA model are “filled up” with \emptyset regular expressions.¹³⁾

Technically, this construction leads to the following definition of G :

1. $Q' = Q \cup \{q_{start}, q_{accept}\}$
2. $\delta'(q_1, q_2) = \begin{cases} \epsilon & \text{if } q_1 = q_{start}, q_2 = q_0 \\ \epsilon & \text{if } q_1 \in F, q_2 = q_{accept} \\ \mathcal{U}(\{a \in \Sigma \mid \delta(q_1, a) = q_2\}) & \text{if } q_1 \in Q, q_2 \in Q \\ \emptyset & \text{for all other to be defined } (q_1, q_2) \end{cases}$

Note here that we introduced for a language A the operation $\mathcal{U}(A)$ that builds the regular expression describing the union over all elements contained in A .¹⁴⁾ We further define

¹³⁾ Transitions with the \emptyset RE will never be traversed, since $L(R \circ \emptyset) = \emptyset$.

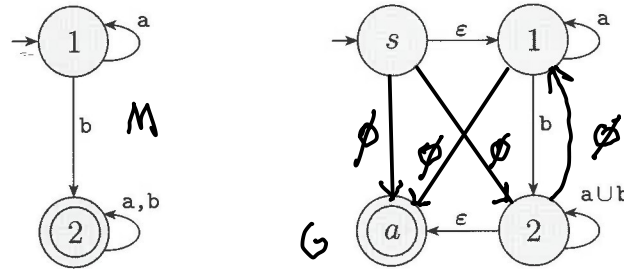
¹⁴⁾ For example: $\mathcal{U}(\{a, b, c\}) = a \cup b \cup c$.

\mathcal{U} such that $\mathcal{U}(\emptyset) = \emptyset$.

From the description of the overall idea, it should be evident that the constructed automaton is a GNFA recognizing $L(M)$. \square

Since the technicalities of the construction discussed in the proof might not be immediately accessible to all readers, we give an example for a concrete DFA.

Example 1.16 The following two STDs give on the left-hand side a FA M for which the STD on the right-hand side describes the GNFA as it would be constructed via the previous proof.



\triangle

Our second mental step is to go from the GNFA to a regular expression:

Lemma 1.4 For each GNFA G , there is a regular expression R , such $L(R) = L(G)$.

We start with the proof idea and give the actual proof afterwards.

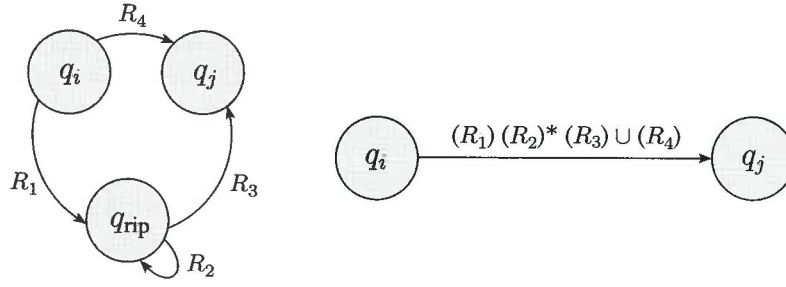
Remark (Proof idea) The core proof idea is to iteratively remove states of the GNFA, while updating the regular expressions in the neighborhood such that still the same language is recognized.

In a more detailed sense, we start from a GNFA $G = (Q, \Sigma, \delta, q_{start}, q_{accept})$. By definition, we have $|Q| \geq 2$, since the start and accept state are distinct. Then, as long as $|Q| > 2$, we do:

Build $G' = (Q', \Sigma, \delta', q'_{start}, q'_{accept})$ such that $L(Q') = L(Q)$ and $|Q'| = |Q| - 1$ and afterwards, replace G by G' .

As soon as we only have two states left in G , the regular expression that we are looking for is given on the only transition that exists between the start and the accept state.

So how does this “state removal step” look like? In principle, we select one state $q_{rip} \in Q \setminus \{q_{start}, q_{accept}\}$ that we would like to remove, remove it, and update the other surrounding transitions as shown below:



On the left-hand side, we see a view on the situation before q_{rip} is removed. On removal of q_{rip} , we have to update the transition between state q_i and state q_j . This transition then (in addition to R_4) has to “cover” the contribution of the path $q_i, q_{rip}(\dots, q_{rip}), q_j$ through the STD. This additional contribution is here the language described by the regular expression

$$(R_1)(R_2)^*(R_3).$$

△

Proof

Let the GNFA $G = (Q, \Sigma, \delta, q_{start}, q_{accept})$ be arbitrary but fixed. We develop the following tail-recursive algorithm that takes as input a GNFA G and returns a regular expression R :

```

1: function CONVERT( $G = (Q, \Sigma, \delta, q_{start}, q_{accept})$ )
2:    $k \leftarrow |Q|$ 
3:   if  $k = 2$  then
4:     return  $\delta(q_{start}, q_{accept})$ 
5:   else
6:     if  $k > 2$  then:
7:       select  $q_{rip} \in Q \setminus \{q_{start}, q_{accept}\}$ 
8:        $Q' \leftarrow Q \setminus \{q_{rip}\}$ 
9:       for all  $q_i \in Q' \setminus \{q_{accept}\}, q_j \in Q' \setminus \{q_{start}\}$  do
10:         $R_1 \leftarrow \delta(q_i, q_{rip})$ 
11:         $R_2 \leftarrow \delta(q_{rip}, q_{rip})$ 
12:         $R_3 \leftarrow \delta(q_{rip}, q_j)$ 
13:         $R_4 \leftarrow \delta(q_i, q_j)$ 
14:         $\delta'(q_i, q_j) \leftarrow (R_1)(R_2)^*(R_3) \cup (R_4)$ 
15:       end for
16:        $G' \leftarrow (Q', \Sigma, \delta', q_{start}, q_{accept})$ 
17:       return CONVERT( $G'$ )
18:     end if
19:   end if
20: end function

```

We claim that for the regular expression R returned by $\text{CONVERT}(G)$, it holds that $L(R) = L(G)$.

We prove this statement via induction on the number of states k of the GNFA.

$k=2$

In this case, G has exactly one transition with the regular expression R' from the start to the accept state. Step 4 of the algorithm immediately returns R' and we have $L(R) = L(G) = L(R')$, since R' is the only regular expression in G . ✓

$k-1 \rightarrow k$

We are given a GNFA G with k states and assume that we have gone through steps 7-15 to build G' . For now, we prove $L(G) = L(G')$. To do this, we show the mutual inclusions of the two sets:

1. $L(G) \subseteq L(G')$

Let w be an arbitrary but fixed string over Σ that is accepted by G . We need to show that it is also accepted by G' .

Since w is accepted by G , there exists a decomposition $w = w_1 \cdots w_m$, $w_i \in \Sigma^*$ and a sequence of states q_0, \dots, q_m such that

$$q_0 = q_{\text{start}}, \quad q_m = q_{\text{accept}}, \quad w_i \in L(\delta(q_{i-1}, q_i)) \text{ for all } i = 1, \dots, m.$$

In the construction of G' , the state q_{rip} is removed from G . Either, this state is on the computation path q_0, \dots, q_m for w , or not. We distinguish both cases:

- a) Case q_{rip} not contained in sequence q_1, \dots, q_{k-1}

If q_{rip} is not on the computation branch of G on w , we expect that the computation branch of G' on w is not altered (in terms of the traversed states). Therefore, we only have to check, whether for consecutive states q_{i-1}, q_i , along which the computation branch of G on w reads w_i , we still have $w_i \in \delta'(q_{i-1}, q_i)$ in the new computation branch of G' , such that this branch is still accepting. In other words we require $L(\delta'(q_{i-1}, q_i)) \supseteq L(\delta(q_{i-1}, q_i))$. Following the definition of CONVERT , we indeed have

$$\begin{aligned} L(\delta'(q_{i-1}, q_i)) &\stackrel{\text{CONVERT}}{=} L((R_1)(R_2)^*(R) \cup (\delta(q_{i-1}, q_i))) \\ &= L((R_1)(R_2)^*(R)) \cup L(\delta(q_{i-1}, q_i)) \\ &\supseteq L(\delta(q_{i-1}, q_i)). \end{aligned}$$

In this case, w is still accepted by G' . ✓

- b) Case q_{rip} contained in sequence q_1, \dots, q_{m-1}

In this case, we know that the accepting computation branch of G on w will go through a sub-sequence $q_{\text{rip}}, \dots, q_{\text{rip}}$ (of size at least one) at least once.¹⁵⁾ We pick an arbitrary but fixed sub-sequence of this type and obtain the nodes q_i

¹⁵⁾ It can be several q_{rip} since q_{rip} can have a loop.

and q_j in the computation branch via which we enter and exit that sequence, i.e.

$$\dots, q_i, q_{rip}, \dots, q_{rip}, q_j, \dots$$

By the construction of G' in CONVERT, q_{rip} is removed, leading to a shorter computation branch of G' on w with

$$\dots, q_i, q_j, \dots$$

at some point. We now have to check, whether this change has affected the acceptance of the computation branch (when moving from G to G').

A modified transition $\delta'(q_i, q_j)$ is introduced by CONVERT with

$$\delta'(q_i, q_j) = (\delta(q_i, q_{rip}))(\delta(q_{rip}, q_{rip}))^*(\delta(q_{rip}, q_j)) \cup (R_4).$$

Here, the language $L((\delta(q_i, q_{rip}))(\delta(q_{rip}, q_{rip}))^*(\delta(q_{rip}, q_j)))$ contains the same sub-strings as described/accepted by the computation

$$\dots, q_i, q_{rip}, \dots, q_{rip}, q_j, \dots$$

of G on w . Consequently, the computation \dots, q_i, q_j, \dots of G' on w via $\delta'(q_i, q_j)$ covers the same sub-strings. Hence, the removal of q_{rip} does not affect the acceptance along the move from q_i to q_j .

As we have shown this for an arbitrary but fixed pair q_i, q_j of states surrounding q_{rip} sequences along computations in G , this holds for all such pairs. Moreover, following the previous case, all other consecutive pairs q_{i-1}, q_i along the computation branch are anyway not affected. Therefore, w is also accepted by G' . ✓

Overall, we conclude that $L(G) \subseteq L(G')$. ✓

2. $L(G) \supseteq L(G')$

In this proof direction, we consider an arbitrary but fixed string w' over Σ that is accepted by G' and have to show that it is also accepted by G .

We pick two arbitrary but fixed consecutive states q'_{i-1}, q'_i in the accepting computation branch for w' . Hence, Q' reads some sub-string w'_i going from q'_{i-1} to q'_i . By the definition of CONVERT, we have

$$\delta'(q'_{i-1}, q'_i) = (\delta(q'_{i-1}, q_{rip}))(\delta(q_{rip}, q_{rip}))^*(\delta(q_{rip}, q'_i)) \cup (\delta(q'_{i-1}, q'_i))$$

and there existed in G a calculation sub-path immediately from q'_{i-1} to q'_i and from q'_{i-1} via $q_{rip}(s)$ to q'_i . Since we have $w_i \in L(\delta'(q'_{i-1}, q'_i))$ and due to above definition of $\delta'(q'_{i-1}, q'_i)$, we hence know that w_i is read by going immediately from q'_{i-1} to q'_i in Q or it is read by going the detour via q_{rip} . In both cases, this does not affect the acceptance of w in G . Therefore, it holds $L(G) \supseteq L(G')$. ✓

We conclude that $L(G) = L(G')$. ✓

Finally, we come back to our induction proof, in which, according to the induction hypothesis, $\text{CONVERT}(G')$ provides a regular expression R such that $L(R) = L(G')$, since $|G'| = k - 1$. However, we have just shown $L(G) = L(G')$, thus we conclude

$$L(R) = L(G') = L(G),$$

and $\text{CONVERT}(G)$ returns the regular expression that describes $L(G)$. \square

Remark After having gone through this proof, we might ask ourselves, why we had these rather complicated conditions on the transitions of a GNFA. To pick some example, we had to require to have distinct start and accept states to make sure that the CONVERT algorithm finally ends up in one minimalistic case, where we have only two nodes with one transition and the final regular expression. As another example, we required to have all (non-start/accept) states having transitions to all other states, including themselves. This guaranteed that we indeed always have this heavily used loop structure between arbitrary states q_i , q_j and q_{rip} . \triangle

As we have proved Lemma 1.3 and 1.4, we have –according to our roadmap for the proof of Lemma 1.2 – proved Lemma 1.2. Moreover, this concludes the proof of Theorem 1.5. \square

We summarize that a language is regular if and only if some regular expression describes it. This means that the descriptive power of regular expressions and finite automata is equivalent.

1.4 Nonregular Languages

So far, all of our interest was focused on *regular* languages. We learned that they are the class of languages associated to FAs, NFAs and regular expressions. Some reader might already have asked her- or himself what comes beyond regular languages, i.e. “What are non-regular languages?”.

In this section, we will give a first answer to this question by discussing examples of non-regular languages. However, more importantly, we will develop the necessary theoretical tools to show that a given language is not regular.

Let us start with an example:

Example 1.17 (Nonregular language) The language

$$B = \{0^n 1^n \mid n \geq 0\}$$

is not a regular language. Intuitively, we can try to argue that an FA would have to be able to “recall” an arbitrary number of 0s, to be able to compare their count with

the count of ones. However, since FAs “store” information in their states, we would not be able to store that arbitrary number of 0s in a finite number of states, which is one limitation by FAs. \triangle

Having this counting argument sounds like a good approach to detect non-regular languages. Unfortunately, this intuition might also fool us, sometimes. Indeed, the language

$$D = \{w \mid w \text{ has an equal number of occurrences of } 01 \text{ and } 10 \text{ as substring}\}$$

is regular.

To cut the story short, we need a theoretical tool that allows us to decide whether a language is not regular. This tool is given by the *Pumping Lemma*.

Theorem 1.6 (Pumping Lemma) Let Σ be an alphabet and A be a language over Σ . If A is a regular language, then there is a number p , the **pumping length**, where, if s is any string in A of length at least p , then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

¹⁶⁾ Trying to formulate this statement more freely, it says that for all regular languages, it holds that strings beyond a certain size contain parts that can be repeated / “pumped up” arbitrarily often while keeping the string in the same language.

How can the Pumping Lemma help? We see this in the following example.

Example 1.18 We repeat our previous example and consider

$$B = \{0^n 1^n \mid n \geq 0\}.$$

We prove, using the Pumping Lemma, that B is not a regular language.

Proof (by contradiction)

We start by assuming that B is a regular language. Since B is regular, the Pumping Lemma holds. Therefore, there exists an integer p , the pumping length, for which the conditions of the Pumping Lemma are fulfilled.

Next, we pick a string that is for sure in B (and that is at least of size p), such that we break the conditions of the Pumping Lemma. In this case, we select the string

$$s = 0^p 1^p.$$

The length of this string is $2p > p$, thus, according to Theorem there exists a splitting $s = xyz$ such that for all $i \geq 0$ it holds

$$xy^iz \in B.$$

¹⁶⁾ While not mentioned explicitly, we allow that either x or z may be the empty string ε .

Now, we match the string s with the xyz splitting pattern and show that in all cases of dividing s into x , y , and z , this condition is not fulfilled:

1. Case (y consists only of 0s)

In this case, for $i = 2$, we would expect $xyyz$ to be part of the language. However, $xyyz$, would have more 0s than 1s. Therefore, it cannot be part of B . \nexists

2. Case (y consist only of 1s)

This case is symmetric to the previous case, i.e. $xyyz$ would have more 1s than 0s. \nexists

3. Case (y consists of both 0s and 1s)

However, again, $xyyz$ is not be in B , since independent of the number of occurrences 0s and 1s in y , yy would always have a pattern of the form $\cdots 0 \cdots 1 \cdots 0 \cdots 1 \cdots$, i.e. we would “mix” 0s and 1s. \nexists

Since we have shown that the statement of the Pumping Lemma no longer holds if we assume that B is regular, it has to be nonregular. \square

\triangle

We continue with a second example:

Example 1.19 The language $F = \{ww \mid w \in \{0, 1\}^*\}$ is nonregular.

Proof

We assume that F is regular. In this case, the Pumping Lemma guarantees the existence of a pumping length p . Then we choose

$$s = 0^p 10^p 1,$$

which is a string in B . According to the Pumping Lemma, there exists a splitting $s = xyz$. The Pumping Lemma further implies $|xy| \leq p$. This however means that both x and y only contain 0s. If we start pumping, i.e. for $i = 2$ we get

$$xyyz = 0^{k_1} 0^{k_2} 0^{k_2} z$$

($k_1 = |x|$, $k_2 = 0$), thus $xyyz$ is not in F . \nexists

\square

\triangle

While the above examples seem to be easy to prove, the actual challenge in proofs for nonregularity via the Pumping Lemma lies in finding an appropriate sample string s .

We now come to the proof of the Pumping Lemma itself. Indeed, the limitation by FAs to a finite number of states will play a crucial role.

Proof

Let A be an arbitrary but fixed regular language. Then there exists an FA

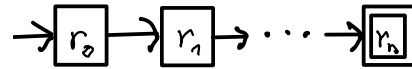
$$M = (Q, \Sigma, \delta, q_1, F)$$

such that $L(M) = A$. We need to show the existence of an integer p such that the three conditions of the Pumping Lemma are fulfilled. We choose

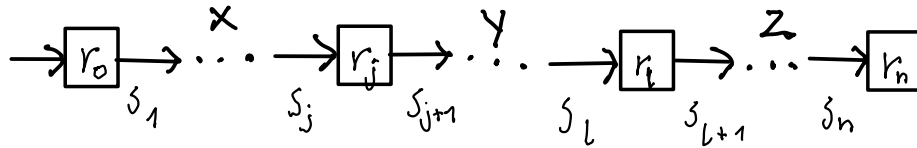
$$p = |Q|$$

and prove all three statements.

Let now $s = s_1 \cdots s_n \in A$ be such that $n \geq p$.¹⁷⁾ Since s is accepted by M , there exists an accepting computation of M on s of the form $c = r_0, r_2, \dots, r_n$, which we can visualize by

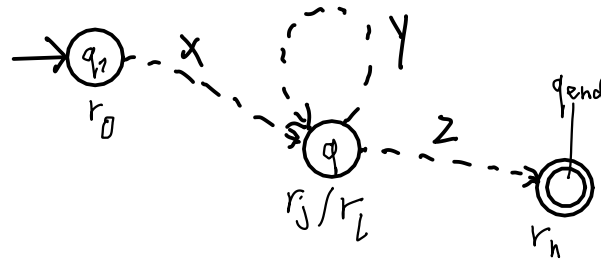


Since r_0, \dots, r_n are $n + 1 > n \geq p$ states, at least one state appears more than once in the computation. Let (j, l) be a pair of state indices such that r_j is the first such state in c that is repeated and $r_l = r_j$ is the second occurrence of this state.



By the selection of the pair (j, l) , it has to hold that $l \leq p = |Q|$. (Otherwise, there would be at least $p + 1$ different states.). We assign

- $x = s_1 \cdots s_j$,
- $y = s_{j+1} \cdots s_l$, and
- $z = s_{l+1} \cdots s_n$.



By reading x we get from r_0 to r_j ($= q$ in the above figure), and by reading z we get from $r_l (= q)$ to r_n , which is an accept state. Since reading y resulted in a loop from/to the state $r_j = r_l (= q)$, strings with an arbitrary repetition of y , i.e. xy^iz , $i \geq 0$, will still be accepted by the FA. Thereby, the first condition is fulfilled. Since $j \neq l$ also the second condition, $|y| > 0$ is fulfilled. Moreover, as we have stated before $l \leq p$ leading to $|xy| \leq p$, i.e. the third condition, is fulfilled. \square

With this, we close this chapter. We have seen various forms of expressing the important language class *regular languages*. In this last section, we have identified a tool to prove that a given language is not regular.

¹⁷⁾ We do not need to consider the case of $|s| < p$ since the Pumping Lemma makes not statement for this case.

2 Context-Free Languages

With the intuition from the last chapter, we are now aware that there exist more languages than just regular languages. So what language class comes beyond the regular languages? Our immediate answer to this question is the upcoming chapter in which we will introduce the class of *context-free languages*.

We will start this chapter by introducing *context-free grammars* (CFG). Like regular expressions, they are a means to describe languages. However, as we will also see, they are “more powerful” than regular expressions. In real-world applications, CFGs are used to, e.g., specify the syntax of programming languages. Hence, parsers for programming languages are built from CFGs.

The language class counterpart for context-free grammars will be *context-free languages* (CFL), similar to regular languages that are the language class counterpart for regular expressions. We will prove that regular languages are contained in the set of CFLs. Moreover, we will introduce another computation model, the *pushdown automaton*, that recognizes context-free languages.

2.1 Context-Free Grammars

As discussed before, CFGs are a means to describe languages. We start with their definition.

Definition 2.1 A **context-free grammar** (CFG) is a 4-tuple (V, Σ, R, S) , where

1. V is a finite set called the **variables**,
2. Σ is a finite set, disjoint from V , called the **terminals**,
3. R is a finite set of (substitution) **rules** / **productions** of the form

$$A \rightarrow w,$$

with $A \in V$ and $w \in (V \cup \Sigma)^*$ (i.e. $w = \varepsilon$ is possible), and

4. $S \in V$ is the **start variable**.

As we will see, CFGs “generate” strings. We come up with an example of a CFG.

Example 2.1 A simple example for a context-free grammar is given by

$$G_1 = (\{A, B\}, \{0, 1\}, R, A).$$

Thereby, we encode the statement that A, B are variables, $0, 1$ are terminals and A is

the start variable. Moreover, we have the rules / productions R given by

$$A \rightarrow 0A1$$

$$A \rightarrow B$$

$$B \rightarrow \varepsilon$$

△

The above definition just gives the syntax of CFGs. We have to associate to that their semantics.

Remark (String generation) As we stated before, CFGs “generate” strings. This is done via *derivation*, which we explain by the following pseudo-algorithm:

1. Write down the start variable.
2. Find a variable that is written down and a rule starting with that variable. Replace the written down variable with the right-hand side of the rule.
3. Repeat step 2 until no variables remain.

△

Let us apply this to the previous example.

Example 2.2 Starting from the previous example one potential derivation of a string could be given by

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 00B11 \Rightarrow 0011.$$

Note however, that this derivation is not the only one that we could come up with. Another example is

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000111.$$

Certainly, many other derivations can be found. Intuitively, it will be clear that the set of all these strings that we can derive with this CFG forms a language, the *language of* G_1 . It should be easy to see that the language of G_1 is $\{0^n 1^n | n \geq 0\}$.¹⁾ △

Informally, we should have now gained the main interpretation/semantics of CFGs. Still, we need to come up with a proper definition.

Definition 2.2 Let $G = (V, \Sigma, R, S)$ be a CFG, $u, v, w \in (V \cup \Sigma)^*$ and $A \in V$.

- If $A \rightarrow w$ is a rule of the grammar, we say that uAv **yields** uwv , written

$$uAv \Rightarrow uwv.$$

¹⁾The attentive reader should be alerted by this language and could start to think about consequences of the relationship of languages of CFGs and regular languages.

- We write

$$u \xRightarrow{*} v,$$

if $u = v$ or if a sequence $u_1, u_2, \dots, u_k \in (V \cup \Sigma)^*$, $k \geq 0$ exists, such that

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_k \Rightarrow v.$$

- The **language of the grammar** G , $L(G)$, is

$$L(G) = \{w \in \Sigma^* \mid S \xRightarrow{*} w\}.$$

- A language that is generated by a CFG is called **context-free language** (CFL).

Effectively, $L(G)$ is the set of all strings that can be derived from the start state by the rules of the CFG. Moreover, we have just introduced a new class of languages, the *context-free languages*. We will soon study the properties of this new class of languages. However, beforehand, we come up with some more practical considerations.

Remark (Notation) We can abbreviate several lines of productions for the same variable on the left-hand side by joining the right-hand sides with a “|”. In the previous example, we would thus have

$$A \rightarrow 0A1 \mid B$$

for the “ A productions”.

△

We give a second example of a CFG.

Example 2.3 Let the grammar $G_2 = (\{S\}, \{(\,,\,)\}, R, S)$ with the rule(s)

$$S \rightarrow (S) \mid SS \mid \varepsilon$$

be given. The language of G_2 is the set of all strings of properly nested parentheses. △

Remark Note that a CFG can be *ambiguous*, i.e. there might be more than one way to generate a given string. △

We also would like to have a first idea of how to design a CFG that describes a specific language.

Example 2.4 Let the language

$$L = \{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}$$

be given. We look for a CFG G_3 with $L(G_3) = L$. To this end, we take the following steps:

1. Construct the grammar for $\{0^n 1^n | n \geq 0\}$:

$$S_1 \rightarrow 0S_11|\varepsilon$$

2. Construct the grammar for $\{1^n 0^n | n \geq 0\}$:

$$S_2 \rightarrow 1S_20|\varepsilon$$

3. Combine them:

$$S \rightarrow S_1|S_2$$

We finally end up with the new CFG G of the form

$$G(\{S, S_1, S_2\}, \{0, 1\}, R, S)$$

with the following rules

$$\begin{aligned} S &\rightarrow S_1|S_2 \\ S_1 &\rightarrow 0S_11|\varepsilon \\ S_2 &\rightarrow 1S_20|\varepsilon \end{aligned}$$

△

Let us now make a first structural statement on our new language class.

Lemma 2.1 For every FA M there exists a CFG G generating the same language, i.e. $L(G) = L(M)$. Hence, every regular language can be generated by a CFG.

This immediately tells us that regular languages are a subset of the set of all context-free languages. Using all other knowledge that we have collected so far, we could say that CFGs are at least as powerful in describing languages as regular expressions.

The proof is rather simple.

Proof

Let an arbitrary but fixed FA $M = (Q, \Sigma, \delta, q_0, F)$ be given. We need to construct a CFG G that generates the same language, i.e. $L(G) = L(M)$.

The CFG G that we are looking for is initially given by

$$G = (Q, \Sigma, R, q_0).$$

We observe that the variables are the states of the FA, while the terminals are the symbols of the alphabet of the FA. Moreover, we add for each pair of state $q_i \in Q$ and symbol $a \in \Sigma$ that leads to a transition to state q_j via $\delta(q_i, a) = q_j$ in the FA M a rule

$$q_i \rightarrow aq_j$$

For all accepting states $q_i \in F$ we further add rules

$$q_i \rightarrow \varepsilon$$

It is obvious from the construction that each derivation of a string s by G has a one-to-one correspondence to an accepting calculation of M on s . \square

Next, we want to work a little bit more on the syntax of context-free grammars. Often, we are interested to build algorithms for CFGs. Such algorithms are easier to develop, if we provide CFGs in a “normalized” form using the *Chomsky normal form*.

Definition 2.3 A context-free grammar $G = (V, \Sigma, R, S)$ is in **Chomsky normal form** if every rule is of the form

$$A \rightarrow BC \tag{2.1}$$

$$A \rightarrow a \tag{2.2}$$

where $a \in \Sigma$, $A \in V$, $B, C \in V \setminus S$. In addition, we permit the rule

$$S \rightarrow \varepsilon.$$

It is obvious that a CFG in Chomsky normal form is a context-free grammar. However, are we also sure that all context-free languages can be described / generated by CFGs in Chomsky normal form? If “Yes.”, then CFGs in Chomsky normal form are equivalent to general CFGs. The answer is given by

Theorem 2.1 Any context-free language is generated by a context-free grammar in Chomsky normal form.

Hence, the answer is “Yes.”. To conclude this section, we would like to have a look at the proof idea for this theorem. It will not be a proof, since we will be giving the construction but would still have to show the correctness of the construction.

Remark (Proof idea) We need to show that for a given arbitrary but fixed CFG $G = (V, \Sigma, R, S)$ there exists a CFG G' in Chomsky normal form such that $L(G') = L(G)$. In the following, we will give a strategy for a conversion of G to G' . The correctness of this strategy will not be covered.

1. Step (start state)

We add a new start state S' and a rule $S' \rightarrow S$ to make sure that the start state is not present on the right-hand side of the productions.

2. Step (elimination of ε -rules)

For all non-start variables, we do not allow to have rules with an ε on the right-hand side. We use this small algorithm to remove them:

- a) Choose $A \in V \setminus \{S'\}$.
- b) Remove all rules $A \rightarrow \varepsilon$.
- c) For each occurrence of A on the right-hand side of a production, add new production(s) with that occurrence deleted, e.g. if we have $R \rightarrow uAvAw$, we further add $R \rightarrow uvAw$, $R \rightarrow uAvw$ and $R \rightarrow uvw$. Moreover, for the special case of $R \rightarrow A$, add $R \rightarrow \varepsilon$, unless $R \rightarrow \varepsilon$ has been previously removed.
- d) If there is an ε -rule left (excluding $S' \rightarrow \varepsilon$), go to a).

3. Step (unit rules)

We present a small algorithm to remove “unit rules” i.e. rules of the form $A \rightarrow B$ with $A, B \in V$:

- a) Choose a unit rule $A \rightarrow B$, $A, B \in V$.
- b) Remove the unit rule.
- c) For all rules of the form $B \rightarrow u$ with $u \in (V \cup \Sigma)^*$ add $A \rightarrow u$, unless this was a previously removed unit rule.
- d) If there is a unit rule left, go to a).

4. Step (finalization)

The following two steps conclude the conversion to a CFG in Chomsky normal form:

- a) Go over all rules $A \rightarrow u_1 \cdots u_k$, $k \geq 2$, $u_i \in V \cup \Sigma$ and replace them such that $u_i \in \Sigma$ are replaced by new variables U_i and rules $U_i \rightarrow u_i$ are added. As a result, we will only have variables on the right-hand sides (in case of terms with more than one terminal/variable).
- b) Go over all rules $A \rightarrow U_1 \cdots U_k$, $k \geq 3$, $U_i \in V$ and replace them by rules

$$A \rightarrow U_1 A_1, \quad A_1 \rightarrow U_2 A_2, \quad \dots \quad A_{k-2} \rightarrow U_{k-1} U_k,$$

where the A_i are new variables.

At the end of these steps, the resulting CFG is in Chomsky normal form and generates the same language as G . The correctness proof for this construction is skipped. \triangle

2.2 Pushdown Automata

Finite automata and nondeterministic finite automata are the associated computing models of regular languages. In this section, we are going to introduce *pushdown automata* (PDAs), for which we will show that they recognize those languages that are generated by context-free grammars, and vice versa.

A pushdown automaton is an NFA with the addition of an (infinite) *stack*.²⁾ We immediately proceed to its definition.

Definition 2.4 A **pushdown automaton** (PDA) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function
(with $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$),
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Let us briefly study the new “features” of this computing model.

Remark (Interpretation of PDAs) As stated before, a PDA is an NFA with a stack. To give full flexibility, we add an additional *stack alphabet* Γ to the definition. It contains all symbols that can be pushed to the stack.

As usual, the additional functionality is encoded in the transition function. For PDAs, we extend the transition function by an additional input from the stack alphabet and map the resulting triple to sets of tuples of states and symbols from the stack alphabet. How to interpret these additional features? The third input to the transition function corresponds to a symbol (or no symbol, i.e. ε) that is read/taken from the stack. The additional stack symbol in the output corresponds to a symbol that is pushed onto the stack. Hence, we can now combine state transitions with operations on an (infinite) stack. \triangle

Before we come to the formal definition of accepted input strings, we should give an example that clarifies the ideas associated to the PDA model.

Example 2.5 We want to give a pushdown automaton that recognizes the language

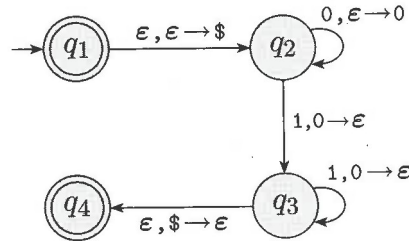
$$L = \{0^n 1^n \mid n \geq 0\},$$

thus our standard example for a nonregular language. The main idea of a stack-based “implementation” of an automaton works as follows: The automaton pushes all zeros that are read on the stack. With the first occurrence of a one, it starts taking one zero from the stack for each one that is read. The automaton only accepts the string, if the reading of consecutive ones leads to an empty stack. If more ones or less ones show up or

²⁾The reader is expected to know the concept of a *stack*, since this is one of the standard data structures in Computer Science.

if another zero shows up, the automaton rejects the string. Moreover, the empty string is accepted.

The following state transition diagram describes a pushdown automaton that exactly implements the idea outlined above.



Before we analyse the functionality of the automaton / STD, we should briefly cover the modified labels associated to the edges / transitions. There, the first symbol before the comma is the symbol read from the input string. After the comma, the first stack symbol corresponds to the symbol that we take from the stack. Behind the arrow, the second symbol is the symbol that is pushed onto the stack. Therefore, if we have a transition with a label of the form

$$a, b \rightarrow c,$$

which points to a node q , this means

Read a from the input, take b from the stack, push c onto the stack and proceed to state q .

If we have $a = \varepsilon$, a transition corresponds to a pure interaction with the stack, without any reading. In case of $b = \varepsilon$, we read a symbol from the input and only push a symbol onto the stack. However, we do not take a symbol from the stack. Similarly, for $c = \varepsilon$, we read a symbol from the input and take a symbol from the stack. However, we do not push a symbol onto the stack.

Let us now come back to the interpretation of the STD. Coming from the starting state, we push the $\$$ symbol onto the stack. We do this to have a marker for the empty stack. The loop at state q_2 pushes the zero symbol to the stack as long as a zero is read from the input. With the first read of a one, a zero is taken from the stack and the automaton proceeds to q_3 . In case of further readings of a one, further zeros are taken from the stack and the automaton stays in q_3 . Only if one calculation branch reaches the end of the input string and then takes the “empty stack marker” $\$$ from the stack, the computation branch will end in the accepting state q_4 .³⁾

We finish this example by giving the formal implementation of the PDA. It is a six-tuple

$$P_1 = (Q, \Sigma, \Gamma, \delta, q_1, F),$$

³⁾Why is the transition from q_3 to q_4 only considered at the end of the input string? Indeed, it is not. However, all calculation branches that would still have input symbols left when reaching q_4 will be terminated and therefore do not contribute to the recognized language.

with the set of states $Q = \{q_1, \dots, q_4\}$, the input alphabet $\Sigma = \{0, 1\}$, the stack alphabet $\Gamma = \{0, \$\}$ and the set of accept states $F = \{q_1, q_4\}$. Moreover we need an extended state transition table for δ , which is given by

Input:	0			1			ϵ		
Stack:	0	\$	ϵ	0	\$	ϵ	0	\$	ϵ
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$			$\{(q_3, \epsilon)\}$			
q_3						$\{(q_3, \epsilon)\}$			$\{(q_4, \epsilon)\}$
q_4									

where the empty cells correspond to the empty set \emptyset . \triangle

Now, we are prepared to give the definition of string acceptance by

Definition 2.5 (Strings accepted by PDA P) Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA. P **accepts** an input string w , if w can be written as $w = y_1 y_2 \dots y_m$ ($y_i \in \Sigma_\epsilon$) and sequences of states $r_0, r_1, \dots, r_m \in Q$ and strings $s_0, s_1, \dots, s_m \in \Gamma^*$ (representing the sequence of stack contents) exist such that

1. $r_0 = q_0, s_0 = \epsilon$, (P starts in start state with empty stack.)
2. for $i = 0, \dots, m-1$ ($a, b \in \Gamma_\epsilon, t \in \Gamma^*$):
 $(r_{i+1}, b) \in \delta(r_i, y_{i+1}, a)$, while $s_i = at$ and $s_{i+1} = bt$, and
($State / stack$ change follows transition function.)
3. $r_m \in F$. (P ends up in accept state.)

If P does not accept w , it **rejects** it.

This definition is very close to the NFA definition. The main difference is certainly in the additional existence of a stack. Note that the s_0, s_1, \dots, s_m are indeed full snapshots of the current content of the stack. In that sense we also have to understand the condition “while $s_i = at$ and $s_{i+1} = bt$ ”. It describes the requirement that a has to be on the top of the old stack snapshot a , while b becomes the new head of the stack s_{i+1} .

As before for NFAs, we would also like to introduce the non-standard notation of a computation branch of a PDA by

Definition 2.6 (Computation branch of a PDA) Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA and w be an input string over alphabet Σ .

A **computation branch of P on w** allows for a modification of the input string w as a string $w = y_1 y_2 \dots y_m$, $y_i \in \Sigma_\epsilon$ and is a sequence of tuples of states and stack contents $c = (r_0, s_0), (r_1, s_1), \dots, (r_m, s_m)$, such that the following two conditions hold:

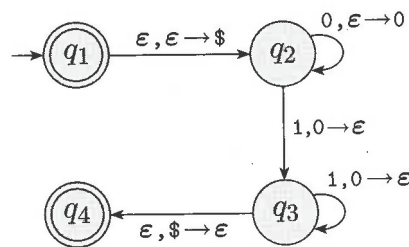
1. $r_0 = q_0, s_0 = \epsilon$
2. for $i = 0, \dots, m-1$ ($a, b \in \Gamma_\epsilon, t \in \Gamma^*$):
 $(r_{i+1}, b) \in \delta(r_i, y_{i+1}, a)$, while $s_i = at$ and $s_{i+1} = bt$, and

We call c an **accepting computation branch**, if $r_m \in F$, otherwise we call it a **rejecting computation branch**.

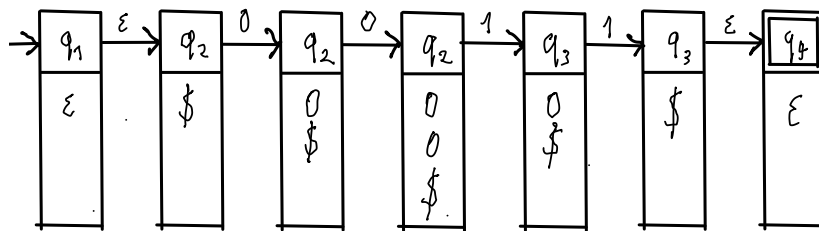
In the computation branch definition, we switch from two sequences for states and stack contents to one sequence of tuples of state and stack content. We do this to stress that the information that characterizes the status of the PDA is no longer just the state, but the conjunction of state and stack content.

We give an example for such a computation branch.

Example 2.6 We continue Example 2.5 and recall that the STD for the PDA recognizing the language $L = \{0^n 1^n | n \geq 0\}$ is given by



The accepting computation branch for P_1 on $w = 0011$ can be visualized by



So obviously, we treat w as the string $w = \varepsilon 0011 \varepsilon$ and the automaton goes through the shown combinations of states and stack contents. Note that the stack content is written vertically to clarify where is the top of the stack (which is the first letter of each stack content s_i in the definition). \triangle

Finalizing our usual pattern of introducing a new computation model, we give the definition for the language of a PDA.

Definition 2.7 (Language of a PDA) Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA. The **language of P** , $L(P)$, that is **recognized** by P is defined by

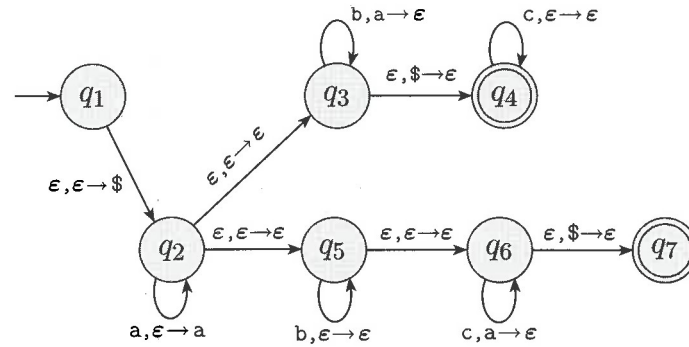
$$L(P) = \{w \in \Sigma^* \mid w \text{ accepted by } P\}.$$

To strengthen our understanding of PDAs, we give a second example.

Example 2.7 Let the language

$$L = \{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } (i = j \text{ or } i = k)\}$$

be given. The STD for a PDA recognizing L is given by



Note that the two options $i = j$ or $i = k$ are achieved by the two non-deterministic $\epsilon, \epsilon \rightarrow \epsilon$ transitions that are leaving q_2 . \triangle

The remainder of this section will deal with the fundamental theorem

Theorem 2.2 A language is context-free if and only if some pushdown automaton recognizes it.

This statement establishes the equivalence of PDAs and CFGs. Since we have already shown that all regular languages can be generated by a CFG, it also implies that all regular languages can be recognized by a PDA. However, this is also intuitively clear from the definition of a PDA.

We will decompose the above theorem into two lemmata and start with the easier direction.

Lemma 2.2 If a language is context-free, then some pushdown automaton recognizes it.

The proof of this lemma relies on the introduction of an *extended pushdown automaton*. We will introduce this model informally in

Remark (Extended pushdown automaton) An *extended pushdown automaton* (ePDA) is a PDA that allows to push strings $u \in \Gamma^*$ instead of a single symbols on the stack. To

achieve that, we modify the transition function in Definition 2.4 to

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow \mathcal{P}(Q \times \Gamma_\varepsilon^*).$$

Moreover, we adapt the acceptance notion in Definition 2.4 such that the second condition becomes

2. for $i = 0, \dots, m-1$ ($a \in \Gamma_\varepsilon$, $b \in \Gamma_\varepsilon^*$, $t \in \Gamma^*$):

$$(r_{i+1}, b) \in \delta(r_i, y_{i+1}, a) \text{ while } s_{i+1} = bt \text{ and } s_{i+1} = bt$$

△

We claim that we can always find for an ePDA an equivalent PDA by

Lemma 2.3 Let P' be an arbitrary ePDA with recognized language $L(P')$. There exists a PDA P accepting the same language, i.e. $L(P) = L(P')$.

Let us give a proof sketch.

Remark (Proof sketch) Let

$$P' = (Q', \Sigma, \Gamma, \delta', q_0, F)$$

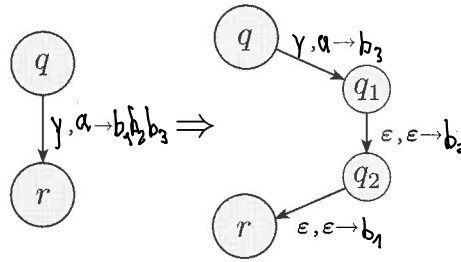
be an arbitrary, but fixed ePDA. We construct a PDA

$$P = (Q, \Sigma, \Gamma, \delta, q_0, F)$$

recognizing the same language as follows:

1. Initially set $Q = Q'$, $\delta(q, y, a) = \emptyset$ for all q, y, a .
2. For each transition $(r, b) \in \delta'(q, y, a)$, $|b| = l$ do:
 - a) $Q = Q \cup \{q_1, \dots, q_{l-1}\}$, i.e. we add $l-1$ new states.
 - b) $\delta(q, y, a) = \delta(q, y, a) \cup \{(q_1, b_l)\}$
 $\delta(q_1, \varepsilon, \varepsilon) = \{(q_2, b_{l-1})\}$
 $\delta(q_2, \varepsilon, \varepsilon) = \{(q_3, b_{l-2})\}$
 \vdots
 $\delta(q_{l-1}, \varepsilon, \varepsilon) = \{(r, b_1)\}$
 $\delta(q, y, a) = \emptyset$ for all $q \in \{q_1, \dots, q_{l-1}\}$ and not $(a = \varepsilon \wedge s = \varepsilon)$,
 i.e. we add a sequence of states such that the individual letters of b are pushed (in reverse order) onto the stack.

The rule 2. b) applied for a transition of the form $(r, b_1 b_2 b_3) \in \delta'(q, y, a)$ would lead to



We claim that this construction will lead to P recognizing the same language as P' . \triangle

Now, we come back to Lemma 2.2. We proof that if a language is context-free, then some pushdown automaton recognizes it. The idea will be to build an ePDA that nondeterministically evaluates all possible string derivations following the productions of a given input CFG.

Proof

Let L be a context-free language. Therefore, there exist a CFG $G = (V, \Sigma, R, S)$ such that $L(G) = L$. We show that there exists an ePDA

$$P = (Q, \Sigma, \Gamma, \delta, q_0, F)$$

such that $L(P) = L(G)$. According to Lemma 2.3, we can then find a PDA recognizing the language and we have completed the proof.

Let us thus now turn our attention to the ePDA. Its stack alphabet becomes

$$\Gamma = V \cup \Sigma \cup \{\$, \}$$

thus we can push strings over all variables and symbols used in the CFG (and an additional symbol $\$$) onto the stack. Moreover, we have states

$$Q = \{q_{start}, q_{loop}, q_{accept}\},$$

have the start state q_{start} and obviously have the set of accept states $F = \{q_{accept}\}$.

The ePDA carries out the following algorithm:

1. It places $\$$ and the start variable of G onto the stack, leading to transitions

$$\delta(q_{start}, \varepsilon, \varepsilon) = \{q_{loop}, S\$ \}.$$

2. It repeats the following steps forever:

- a) If $A \in V$ is on top of the stack, replace it nondeterministically by the right-hand sides of all productions for A , leading to transitions

$$\delta(q_{loop}, \varepsilon, A) = \{(q_{loop}, w) | \text{where } "A \rightarrow w" \text{ is a rule in } R\}.$$

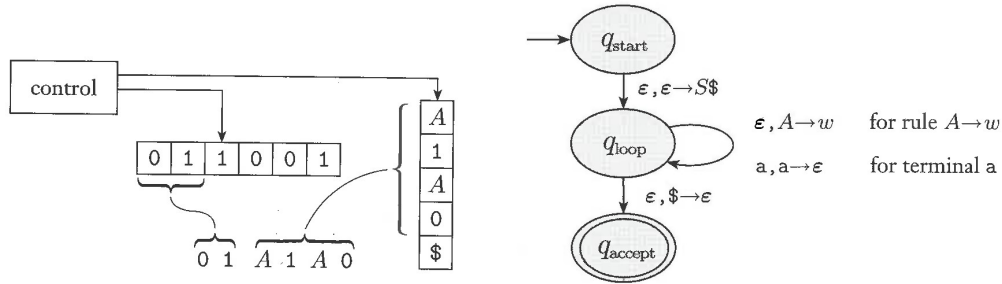
- b) If $a \in \Sigma$ is on top of the stack, read the next input symbol and compare it to a . If they match, proceed, else reject the computation branch. This is implemented by the transition

$$\delta(q_{loop}, a, a) = \{(q_{loop}, \varepsilon)\}.$$

- c) If \$ is on top of the stack, enter the accept state with no further transitions, indicating the end of the string. This transition is given by

$$\delta(q_{loop}, \varepsilon, \$) = \{(q_{accept}, \varepsilon)\}.$$

The above construction can also be visualized by



By construction, only those inputs are accepted that can be generated using G , i.e. the language of G . \square

We now come to the second direction of the statement of Theorem 2.2.

Lemma 2.4 If a pushdown automaton recognizes some language, then it is context-free.

Remark (Warning) The different definitions, lemmata and proofs that we need to prove this lemma have the potential to initially cause some headache. It sometimes helps to read these contents once, put them away and repeat this on the next day, maybe over several days. \triangle

In the proof for Lemma 2.4, we need another flavor of a PDA, which we call *modified PDA* (mPDA):

Remark (modified PDA) A *modified PDA* is a PDA

1. that has a single accept state,
2. that empties the stack before accepting, and
3. that has transitions such that each transition either pushes or takes a symbol to / from the stack (i.e. doing this at the same time is not possible and doing nothing with the stack is impossible).

We don't give a formal definition here. \triangle

Lemma 2.5 Let P be an arbitrary but fixed PDA with language $L(P)$. There exists an mPDA P' such that $L(P') = L(P)$.

We sketch a proof:

Remark (Proof sketch) By the three following steps, we can gradually convert a PDA to an mPDA recognizing the same language:

1. We convert a PDA to a PDA with a single accept state by adding a new accept state being reached from the old accept states by $\varepsilon, \varepsilon \rightarrow \varepsilon$ -transitions. The old accept states are removed from the set of accept states.
2. We convert a PDA with a single accept state to a PDA that additionally empties the stack before accepting as follows: First we need to have a stack symbol, e.g. \$, that allows us to detect an empty stack as discussed in Example 2.5. Then we insert before the single accept state further states that implement the logic of removing all elements from the stack.
3. Finally we convert the previous PDA to an mPDA by replacing all push-and-take (at the same time) transitions by a transition to a new state that only takes the top stack element and a transition from that new state to the old next state that pushes a an element onto the stack. In case of transitions with neither a push nor a take operation, we add a new state, do a transition to that new state – while pushing some symbol – and do a take operation while moving to the old next state for the exact same symbol.

It would remain to show the equality of the accepted languages. \triangle

We now introduce a rather technical lemma.

Lemma 2.6 Let $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$ be an mPDA and let $G(P) = (V, \Sigma, R, S)$ be a CFG built from P with variables $V = \{A_{pq} | p, q \in Q\}$, the same alphabet Σ and the starting variable $S = A_{q_0 q_{accept}}$. Moreover the rules R are given by

1. for each $p, q, r, s \in Q, t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε) add the rule

$$A_{pq} \rightarrow aA_{rs}b,$$

2. for each $p, q, r \in Q$ add the rule

$$A_{pq} \rightarrow A_{pr}A_{rq}, \text{ and}$$

3. for each $p \in Q$ add the rule

$$A_{pp} \rightarrow \varepsilon.$$

An arbitrary A_{pq} in $G(P)$ generates a string $x \in \Sigma^*$, i.e. $A_{pq} \xRightarrow{*} x$, if and only if x can bring P from state p with empty stack to state q with empty stack.

We should interpret the lemma as follows: Let us assume, we have an mPDA P . That mPDA will have computation branches on some input strings. Along a partial

computation branch, i.e. one going from some node p to some node q , we read partial input strings. Then the variables A_{pq} in $G(P)$ generates all partial input strings that could be read along that partial computation branch. However, we have the further condition that this reading process of the mPDA had to end in an empty stack in state q , if the stack was empty in p .

The proof for Lemma 2.6 is not given immediately. Instead we use it to first prove our “big” Lemma 2.4.

Proof of Lemma 2.4

Let P' be a PDA with language $L(P')$. Following Lemma 2.5, there exists an mPDA P such that $L(P) = L(P')$.

Let further $G(P)$ be the CFG constructed via Lemma 2.6. We make the observation that all mPDAs start with an empty stack and, when reaching the accepting state of mPDAs, the stack will be empty again. As a consequence, the statement of Lemma 2.6 is applicable to the starting variable $A_{q_0q_{accept}}$ of the CFG $G(P)$ and it holds that the starting variable of $G(P)$ generates a string $x \in \Sigma^*$, if and only if x can bring P from its starting state to its accept state. Therefore a string w is in $L(G(P))$ if and only if it is in $L(P)$. \square

The above proof just combines all the work that we have done before and applies Lemma 2.6, which obviously describes the CFG that is equivalent to the original (m)PDA. The hard part is the proof of Lemma 2.6.

Proof of Lemma 2.6

We prove the “if and only if” statement by dealing individually with both directions.

“ \Rightarrow ”:

The proof is done by induction on the number of steps k in the derivation of x from A_{pq} .

- $k=1$

In the base case, only rule $A_{pp} \rightarrow \varepsilon$ is applicable. Obviously ε will bring a computation of P from state p with empty stack to p with empty stack. \checkmark

- $k \rightarrow k+1$

We consider the derivation $A_{pq} \xRightarrow{*} x$ with $k+1$ steps and carry out the induction step by splitting off the first step.

In the first step of the derivation, the rule is either

$$A_{pq} \rightarrow aA_{rs}b \quad \text{or} \quad A_{pq} \rightarrow A_{pr}A_{rq}.$$

We walk through both cases.

1. Case $A_{pq} \rightarrow aA_{rs}b$

We assume that the stack is empty when being in state p during a computation. Let the string derived from A_{rs} be called y . Then it follows $A_{pq} \xRightarrow{*} x = ayb$. Since the derivation of y will only need k steps, the induction hypothesis provides that y brings P from an empty stack in r to an empty stack in s .

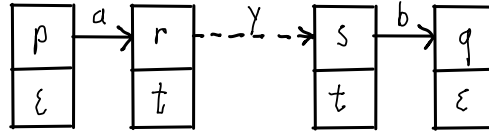
Since $A_{pq} \rightarrow aA_{rs}b$ was added to $G(P)$, it holds for P :

$$(r, t) \in \delta(p, a, \varepsilon) \quad (2.3)$$

and

$$(q, \varepsilon) \in \delta(s, b, t) \quad (2.4)$$

The situation can be visualized as follows.



Studying the computation branch of P going from p to q , we get the following setting

- a) In state p , the stack is empty and we read a . By (2.3) we obtain:
- b) In state r , the stack contains t and we read y . By the induction hypothesis⁴⁾ we get:
- c) In state s , the stack contains t and we read b . By (2.4) we come to:
- d) In state q , the stack is empty.

Thereby, we have shown that x brings P from p with empty stack to q with empty stack. ✓ In particular, in q , the stack is empty again and x is partial string in P .

2. Case $A_{pq} \rightarrow A_{pr}A_{rq}$

We assume that the stack of P is empty when being in state p during a computation. Let us assume that we derive from A_{pr} the string y and from A_{rq} the string z .

According to the induction hypothesis, y will bring P from state p with empty stack to state r with empty stack. Moreover, z will bring P from state r with empty stack to state q with empty stack. Therefore the combined string $x = yz$ will bring P from state p with empty stack to state r with empty stack and further to q with empty stack. ✓

Thereby, we have shown the first direction. ✓

“ \Leftarrow ”

We carry out a proof by induction over the number of steps k of the computation (branch) in P .

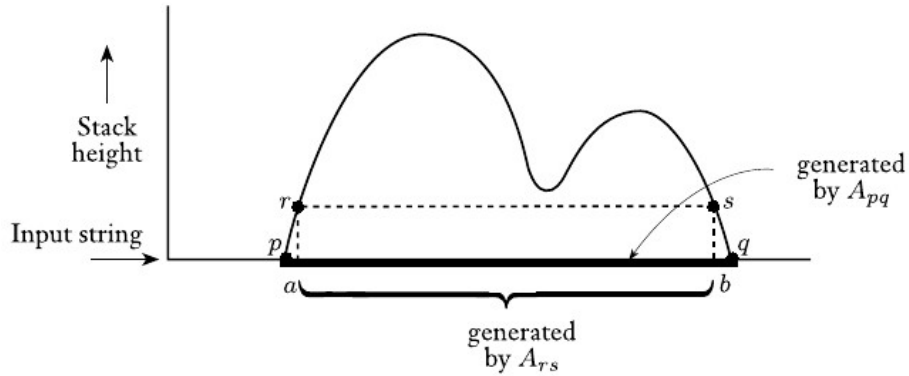
• $k = 0$

In this case, the computation in P can only remain in a state p with empty stack. This corresponds to reading $x = \varepsilon$. In $G(P)$ we find the rule $A_{pp} \rightarrow \varepsilon$. Its only derived string is $x = \varepsilon$. ✓

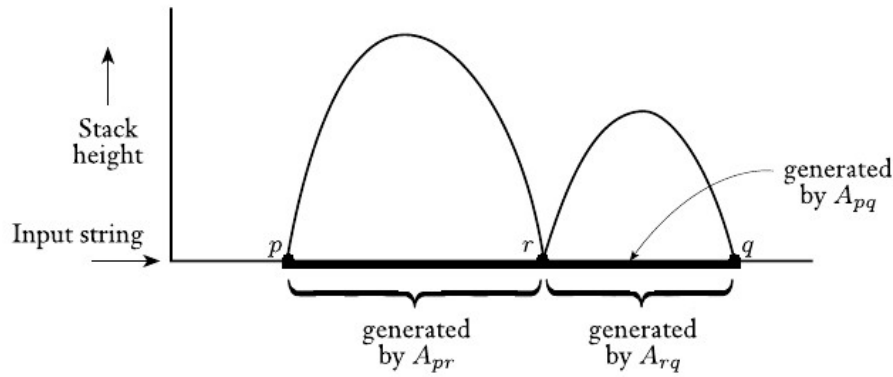
⁴⁾ Actually, we are a bit lazy here, since going from state r to state s only takes $k - 1$ steps. Therefore, we would have to do a strong induction, where we assume that the statement holds for all $k' < k + 1$. And since we go here from $k + 1$ to $k - 1$ we would even have to prove the base case $k = 2$.

- $k \rightarrow k + 1$

Now we consider a partial computation branch with $k + 1$ steps bringing P from p to q with empty stack before and after the computation while reading x . We either have the case that during this computation, the stack never becomes empty. This case is visualized by



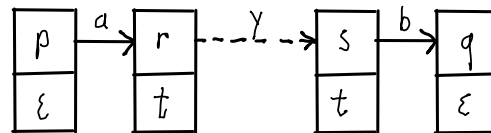
The second case is that the stack becomes empty (at least) at some state r during the computation between p and q . This case is visualized by



Let us consider both cases:

1. Case “stack never becomes empty”

Since P is an mPDA, it can either push symbols or take symbols from the stack. Therefore, the symbol $t \in \Gamma$ that is pushed in the first step when leaving p , reading a and going to r , is the symbol that is last taken from the stack in the very last step reading b , leaving s going to q (see above figure on that case). We thus obtain the following (partial) computation branch.



However, as an immediate consequence, the transitions

$$(r, t) \in \delta(p, a, \epsilon),$$

$$(q, \varepsilon) \in \delta(s, b, t)$$

have to be present in P . Since these transitions are present in P , we obtain following the definition of $G(P)$ in $G(P)$ the rule

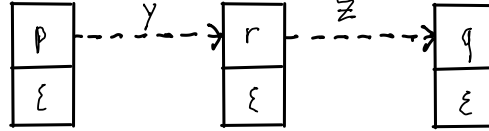
$$A_{pq} \rightarrow qA_{rs}b.$$

We know that x brings P from p to q and has as first letter a and as last letter b . We call the remainin string (between a and b) y , i.e. define y via $x = ayb$. Let us study the properties of y . Either push or take operations are possible in the computaton of the mPDA P . Moreover the identical symbol t is on the stack in r and s . Therefore, the calculation associated to y never “touches” t on the stack. As a consequence, if we were starting the computation of P on r with empty stack, y would still bring us to s in P with empty stack.

Since y corresponds to a computation of $k + 1 - 2 = k - 1$ steps and since the requirement of y bringing us from empty stack to empty stack is fulfilled, we are allowed to apply the induction hypothesis, which gives us that A_{rs} generates the string y . Combining that with the rule $A_{pq} \rightarrow aA_{rs}b$, we have just shown that A_{pq} generates $ayb = x$. ✓

2. Case “stack becomes empty”

Let r be the first state in which the stack becomes empty. The resulting partial computation branch can be visualized by



Obviously, the number of steps of the partial computation branches from p to r , reading some string y and from r to q reading some string z is in both cases smaller or equal to k . Moreover, the computations for y and z go both from an empty stack to an empty stack. Consequently, we can apply the induction hypothesis and obtain that A_{pr} generates a string y and A_{rq} generates a string z . With the matching rule $A_{pq} \rightarrow A_{pr}A_{rq}$ in $G(P)$, we obtain that A_{pq} generates $x = yz$. ✓

This concludes the proof of the second direction and therefore the proof of the overall lemma. \square

Since we have just shown Lemma 2.6, we have concluded the discussion of Lemma 2.4. Moreover, we have shown Lemma 2.2, before. These were the two directions of Theorem 2.2, which we have thereby proved. Consequently, we have established the equivalence between context-free languages and pushdown automata.

2.3 Non-context-free languages

In this concluding section of this chapter, we are going to have a fairly quick look at languages beyond the context-free languages. To be more specific, we will introduce the Pumping Lemma for context-free languages that will allow us to prove that a language is not context-free. Moreover, we recapitulate the languages and automata models seen so far and will put them in the perspective of the so-called *Chomsky hierarchy*.

Let us jump into the topic by formulating

Theorem 2.3 (Pumping Lemma for context-free languages) If A is a context-free language, then there is a number p , the **pumping length**, where if s is any string in A of length at least p , then s may be divided into five pieces $s = uvxyz$, satisfying the following conditions

1. For each $i \geq 0$, $uv^i xy^i z \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

In the theorem, either v or y may be the empty string. We will not discuss the proof, here. However, we immediately go forward to an example of using the theorem to show that a given language is not context-free.

Example 2.8 We use the Pumping Lemma for context-free languages to show that the language

$$B = \{a^n b^n c^n | n \geq 0\}$$

is not context-free.

Proof

We start by assuming that B is a context-free language. Therefore, the Pumping Lemma for CFLs guarantees that there exists a pumping length p , such that the known conditions hold. We now select

$$s = a^p b^p c^p.$$

In the following, we will show that all possible divisions of s into $uvxyz$ will violate one of the conditions of the Pumping Lemma.

For now, we note that condition 2) implies that either v or y may be empty. Our analysis can be made by distinguishing two cases.

1. Case (both v and y contain only one type of symbol)
In this case, v does not contain both as and bs or both bs and cs . The same statement can be made for y . As a consequence uv^2xy^2z will never contain an equal number of as , bs and cs at the same time, violating condition 1). ✗
2. Case (at least one of v and y contains more than one type of symbol)

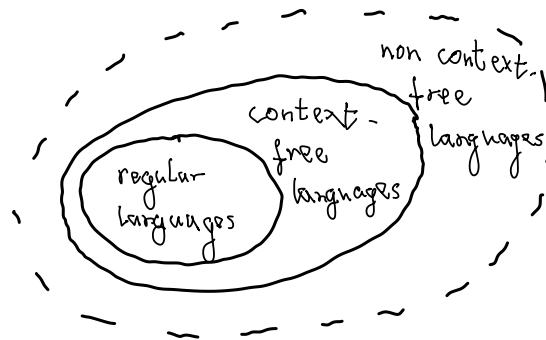
In this case uv^2xy^2z will not have the symbols in the correct order, again contradiction to condition 1). \nexists

Therefore, B must be not context-free. \square

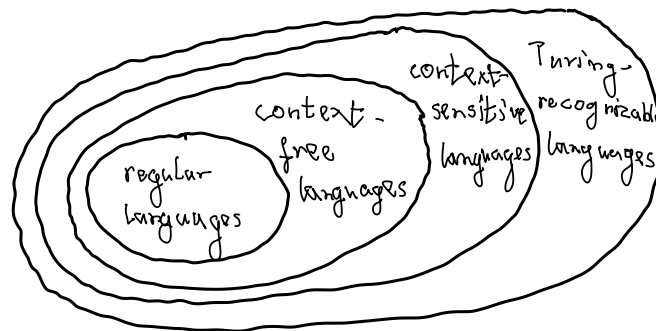
\triangle

We would like to summarize our findings of the first chapters by

Remark (Chomsky hierarchy) So far, we have seen that regular languages exist and are described by FAs and NFAs. Moreover, regular languages are a proper subset of context-free languages, where CFLs are described by pushdown automata. This section implies that there have to exist *non*-context-free languages. Our immediate conclusion of these observations should be that the “world of formal languages” looks at least as follows:



In “reality”, the “world of formal languages” has the following general structure:



Consequently, beyond context-free languages, the next more complex class of languages are *context-sensitive languages*. These are recognized by *linear-bound automata*. If we then still try to look for a more complex language class, we end up with *Turing-recognizable languages*. These languages are associated to *Turing machines*.

Classical literature in the field has a notion for this hierarchy of languages. Indeed, we call the hierarchy

0. Turing-recognizable languages,
1. Context-sensitive languages,
2. Context-free languages,

3. Regular languages,
the **Chomsky Hierarchy** of languages. \triangle

In the following Chapter, we will immediately proceed to Turing-recongnizable languages and the associated Turing machines.

3 Turing Machines

In the previous chapters, we have seen (nondeterministic) finite automata and pushdown automata as the two major computing models. If we were to characterize their “computing abilities”, we could argue that finite automata are a model for devices with a very low amount of “memory” that is hard-wired in the states. In contrast, PDAs are a model for devices with infinite memory. However, that memory is only present in a stack, hence random access to data is not available.

This chapter is concerned with the introduction of a new computing model, which is called *Turing machine* (TM). Continuing the above discussion, Turing machines are a model for devices with infinite and random-access memory. As we will see, TMs indeed are the closest match to the von Neumann architecture and therefore are widely used in theory of computation as *the* model for real computers. Consequently, also all remaining chapters of this work will use Turing machines as computing model.

In the following, we will first formally introduce TMs with examples. Afterwards, we will discuss *multitape Turing machines* and *non-deterministic Turing machines* for which we will show equivalence to TMs. Moreover, we partially discuss *random access machines*, which are also equivalent computing models. The equivalence to the latter ones will allow us to claim that TMs mostly match the von Neumann architecture.

3.1 Formal definition and examples

As we are already quite used to reading formal definitions of computing models, we immediately proceed to

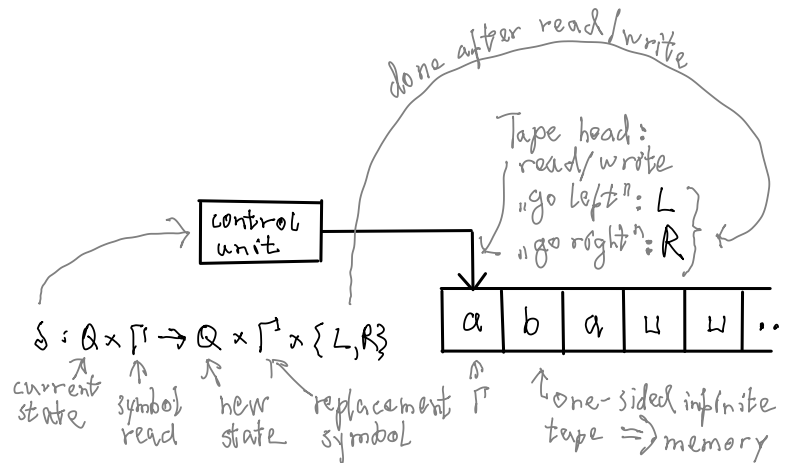
Definition 3.1 (Turing machine) A **Turing machine** is a 7-tuple, $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the **blank symbol** \sqcup
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

We discuss the functioning of TMs by

Remark (Interpretation of Turing Machines) In principle, Turing machines are finite automata with the additional feature to be able to read from and write to a tape. That

tape is infinite in one direction, i.e. it has a beginning on the left-hand side, but no end. The following figure characterizes the components of a TM:



A Turing machine has a control unit and a tape. The control unit has access to the tape via the *tape head* that can be moved to the left and to the right on the tape. Only symbols below the tape head can be written / read by the TM.

The “computation” of a TM works as follows. In the initial state, there is a string (terminated by an infinite amount of blank symbols) given on the tape.¹⁾ Then, the transition function indicates how the control unit operates. The input to the transition function is the current state and the symbol that is currently under the tape head, hence the TM “reads” that symbol. The output of the transition function is the next state, a symbol that is forced to be written to the current location of the tape head and the direction of the tape head movement after the write operation, i.e. one element to the left or one element to the right. The computation of a TM may not terminate, thus it can run forever. It only terminates if it enters the q_{accept} or the q_{reject} state. However, this termination is immediate, i.e. these two states cannot be left anymore. The initial string on the tape is accepted, if the machine finally enters the accept state.

As a side-remark, we still have to discuss how the tape head operates in case it “hits” the left end of the input tape. In that case, it stays on the leftmost tape entry, even if further moves to the left are indicated. \triangle

Later, we will also have a look at the tape content when the TM accepts the input. Then, we will call that content the *output* of the TM. However, for now, we simply consider the input and its acceptance.

We come to a first example.

Example 3.1 We would like to discuss a Turing machine that accepts an input string,

¹⁾Hence TMs do not read strings like FAs. Instead they just operate on the tape.

if it is contained in the language

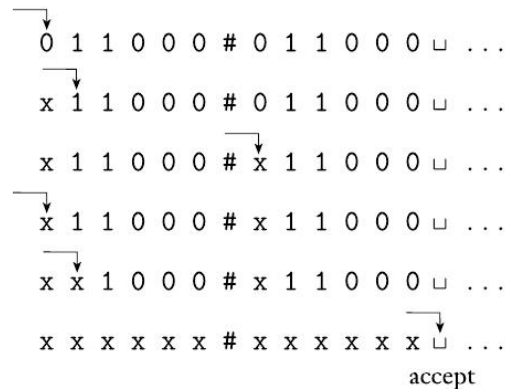
$$B = \{w\#w \mid w \in \{0,1\}^*\}.$$

As we will often see, it is much easier to describe the functionality of a Turing machine by something like an algorithm rather than by an STD.²⁾ Therefore, for now, we just give the “behavior” of the TM that we are looking for as a pseudo-algorithm:

M_1 = “On input string w :

1. Zig-zag across the tape to corresponding positions on either side of the $\#$ symbol to check whether these positions contain the same symbol. If they do not, or if no $\#$ is found, *reject*. Cross off symbols as they are checked to keep track of which symbols correspond.
2. When all symbols to the left of the $\#$ have been crossed off, check for any remaining symbols to the right of the $\#$. If any symbols remain, *reject*; otherwise, *accept*.”

The following snapshots of the tape content and the tape head give an idea of the operation of the algorithm:



Initially the string 011000#011000 is given on the tape. The TM reads the first symbol, memorizes that it read a 0 symbol and then crosses off the symbol (by overwriting it with an x). Then, the tape head proceeds to the second part of the input (after the $\#$ symbol) and compares that symbol with the 0. If it is also a 0 it crosses it off and moves back to the beginning (after the first crossed off symbol). Then this next symbol, a 1, is read, crossed off, and the tape head again proceeds to the second part (right of the $\#$ and all crossed off symbols). Again, that symbol is compared to the 1 of the first part of the string. If it's the same symbol, it is again crossed off and the tape head goes back to the beginning. The whole procedure is repeated as long as the symbols are identical and the end of both strings is not yet reached. Since both ends of the input strings are simultaneously reached, the string is accepted.

²⁾In fact, many of the TMs that we will study later are hard or almost impossible to specify by an STD.

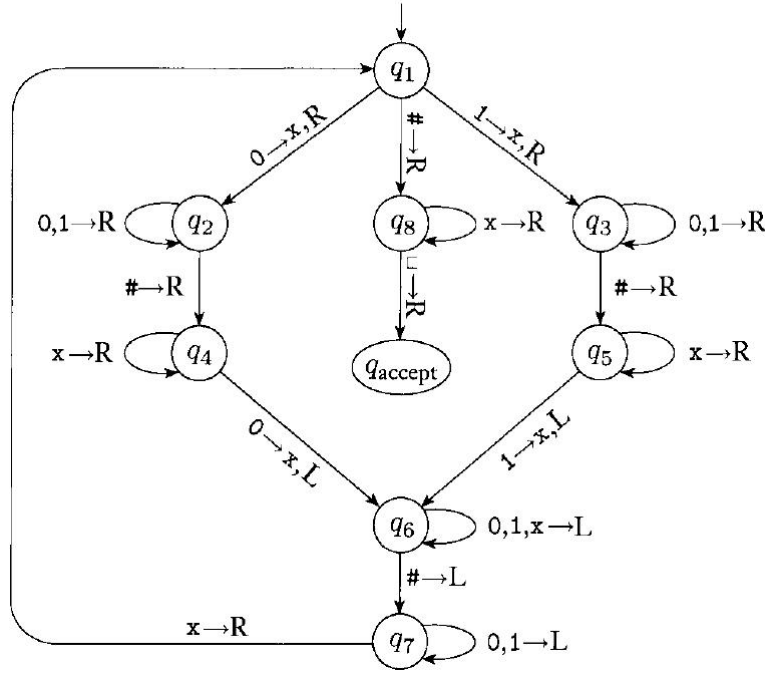
Next, we formally introduce the TM that implements the above behavior. It is given by the 7-tuple

$$M_1 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$$

with the states

$$Q = \{q_1, \dots, q_8, q_{\text{accept}}, q_{\text{reject}}\}$$

the input alphabet $\Sigma = \{0, 1, \#\}$ and the tape alphabet $\Gamma = \{0, 1, \#, x, \sqcup\}$. Then, the transition function is implicitly given by the following STD:



Note that the STD is not complete. We are lacking the q_{reject} state into which we enter whenever the TM gets an input that is not specified in the shown STD.

How to interpret this STD? Assume that we have an edge label of the form

$$a \rightarrow b, D,$$

going from a state q to q' . Then, the TM only takes this transition, if it is in state q and reads under the tape head the symbol a . As a consequence of that, it then writes the symbol b and moves the tape head to direction $D \in \{L, R\}$, while going to state q' . A label of the form $a \rightarrow D$ abbreviates the label $a \rightarrow a, D$.

If we inspect the STD a little bit, we observe that the branch of the left-hand side describes the reading of a zero in the first string, the movement beyond the delimiter symbol $\#$ and further crosses x and the reading and overwriting of a corresponding 0 in the second string. The right-hand side branch is symmetric to it and does the same for the 1 symbol. In states q_7 and q_8 , the TM moves back to the last x of the first string and then places, while going to q_1 , the tape head on the first available non- x symbol in the

first string. The central branch is entered, as soon as the first part of the string is fully “crossed out”, i.e. the delimiter symbol is the next symbol to start from. In this case, the TM still has to check, whether the second string is also already fully “crossed out”. If this is the case, the TM accepts. As said before, since we have to give for all possible inputs a transition to some state, all missing transitions go to the rejecting state. \triangle

While we might not have developed a full intuition of the TM model, yet, we should still slowly proceed to the formalization of the semantics of the model. Further examples will come.

At this point, we should recall the notion of a *computation (branch)* that we have introduced for FAs and PDAs. For FAs, it is simply a sequence of states, thus the current “status” of an FA is fully characterized by its current state. For PDAs, the computation branch is a sequence of tuples of a state and the full content of the stack. Therefore, the current “status” of a PDA is fully characterized by the state and the full stack content. In case of Turing machines, we call this current “status” the *configuration* of the TM. The “status” or configuration is fully characterized by the current tape content to the left of the head, the current state and the current tape content below and to the right of the head (i.e. encoding, state tape head position and tape):

Definition 3.2 (Configuration of a TM) Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a Turing machine, $u \in \Gamma^*$, $v \in \Gamma^\omega$ and $q \in Q$. The setting

$$u q v$$

is a **configuration** of the Turing machine, where

- u is the current tape content to left of the head,
- q is the current state, and
- v is the current tape content below (v_1) and to the right of the head (being terminated by blanks \sqcup).

Since the string to the right of the tape head is infinite, we needed a small piece of new notation, characterized by

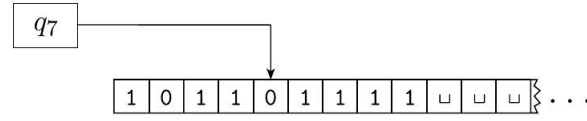
Definition 3.3 Let Σ be an alphabet. Then Σ^ω is the set of all *infinite strings* over Σ .

While we might come up with a more formal approach to define *infinite strings*, we just keep it like this and hope that the idea is intuitively clear.

Let us give an example for a configuration.

Example 3.2 (Configuration of a TM) Let us assume that the TM is at some stage of the currently ongoing computation in state q_7 and the tape and tape head is given as

below:



Then the corresponding configuration is

$$1011q_701111.$$

Note that we did not write down the infinitely many blank symbols at the end. We will always make it like this. \triangle

Next, we would like to define what are valid “transitions” between two configurations:

Definition 3.4 Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ be a Turing machine, $a, b, c \in \Gamma$, $u \in \Gamma^*$, $v \in \Gamma^\omega$ and $q_i, q_j \in Q$. A configuration C_1 **yields** a configuration C_2 if M can legally go from C_1 to C_2 in a single step. Hence, we say

- (*leftward move*): $uaq_i bv$ **yields** $uq_j acv$
if it holds $\delta(q_i, b) = (q_j, c, L)$,
- (*rightward move*): $uaq_i bv$ **yields** $uacq_j v$
if it holds $\delta(q_i, b) = (q_j, c, R)$,
- (*L-move, left tape end*): $q_i bv$ **yields** $q_j cv$
if it holds $\delta(q_i, b) = (q_j, c, L)$,
- (*R-move, left tape end*): $q_i bv$ **yields** $cq_j v$
if it holds $\delta(q_i, b) = (q_j, c, R)$.

Indeed, this definition is a bit unwieldy. However, it exactly formalizes our informally given notion of reading from and writing to the tape and of moving the tape head to the left and to the right. To pick an example, we briefly study the “rightward move”. Hence we start with configuration

$$uaq_i bv$$

and read b , write c and move to the right. This modifies the above configuration such that the b is changed to c and c is afterwards on the left-hand side of the state. Hence we obtain configuration

$$uacq_j v.$$

The third and fourth case in Definition 3.4 reflect the special handling of the left end of the tape, as discussed before.

Now, let us identify a few special configurations by

Definition 3.5 (Characterization of configurations) Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a Turing machine and $C = u q v$ a configuration of M . We call C

- **start configuration on input $w \in \Sigma^*$** , if $u = \varepsilon$, i.e. head is at leftmost end of tape, $q = q_0$ and $v = w \circ \{\sqcup\}^\omega$,
- **accepting configuration**, if $q = q_{\text{accept}}$,
- **rejecting configuration**, if $q = q_{\text{reject}}$,
- **halting configuration**, if it is accepting or rejecting configuration.

Specifically the *start configuration on input w* formalizes the idea to have inputs to the TM being placed on the tape. Note here also that while there is exactly one accepting or rejecting state, there can be many accepting / rejecting configurations.

After all this preparatory work, we can finally come up with

Definition 3.6 (Accepted input & recognized language)

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a Turing machine and $w \in \Sigma^*$ its input. M **accepts** w , if a sequence of configurations C_1, C_2, \dots, C_k exists, such that

1. C_1 is start configuration on input w
2. C_i yields C_{i+1} , $i = 1, \dots, k-1$, and
3. C_k is accepting configuration.

The **language of M** is

$$L(M) := \{w \in \Sigma^* \mid M \text{ accepts } w\}.$$

We say that $L(M)$ is **recognized** by M .

It now becomes clear that the notion of configurations allows us to give a definition for acceptance of a string that looks very much like the one that we have seen for (N)FAs and PDAs.

Similarly, we can now introduce a *computation of M on w* by

Definition 3.7 (Computation of a TM)

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a Turing machine and $w \in \Sigma^*$ its input. A **computation of M on w** is a sequence of configurations C_1, C_2, \dots , such that

1. C_1 is start configuration on input w
2. C_i yields C_{i+1} , $i = 1, \dots, k-1$, and

Finally, we introduce the big class of *Turing-recognizable* languages by

Definition 3.8 A language is called **Turing-recognizable** or **recursively enumerable** if some Turing machine recognizes it.

There is one speciality about TMs that we have not seen for (N)FAs or PDAs. Indeed, TMs can run forever, i.e. it can happen that they don't terminate. So a general TM can have the following three possible behaviors for a given input string:

- It enters the accept state, thus the string is *accepted*, or
- it enters the reject state, thus the string is *rejected*, or
- it loops forever, thus the string is (again!) *rejected*.

However, how to deal with looping TMs? How do we know whether the computation takes long time or whether the computation indeed loops forever? From a practical perspective, there is no easy answer to that. Therefore, in many cases, we would like to restrict ourselves to TMs that will not loop for some input. Here, we need

Definition 3.9 A TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ **halts** on a given input $w \in \Sigma^*$, if a sequence of configurations C_1, C_2, \dots, C_k exists, such that

1. C_1 is start configuration on input w
2. C_i yields C_{i+1} , $i = 1, \dots, k-1$, and
3. C_k is halting configuration.

Then, we give a specific notion for TMs that will always halt and that recognize a specific language by

Definition 3.10 (Decider) We call a TM M a **decider**, if it halts on all inputs $w \in \Sigma^*$. A decider M that recognizes $L(M)$ is said to **decide** $L(M)$.

As always, if we have found a specific computing model, it also implies the introduction of a class of languages. For *deciders*, i.e. TMs that always halt, these are *decidable languages*:

Definition 3.11 (Decidable languages) We call a language **Turing-decidable**, **decidable** or **recursive** if there exists a TM (i.e. a decider), that decides it.

We trivially conclude

Lemma 3.1 Every decidable language is Turing-recognizable.

After this definition-heavy part, we should come to a second example to strengthen our understanding of these notions.

Example 3.3 We would like to find a TM M_2 that decides the language

$$A = \{0^{2^n} \mid n \geq 0\} .$$

Hence, we would like to find a TM that halts on all inputs and recognizes A . The language is simply the set of all strings composed of zeros with length 2^n or – alternatively expressed – the unary representation of the numbers 2^n .

An algorithm describing the necessary behavior of M_2 is given as

- $M_2 =$ “On input string w :
1. Sweep left to right across the tape, crossing off every other 0.
 2. If in stage 1 the tape contained a single 0, *accept*.
 3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, *reject*.
 4. Return the head to the left-hand end of the tape.
 5. Go to stage 1.”

To understand the algorithm better, we should have a look at some sample input. Let the input

00000000

be given. After a first sweep, it becomes

$x0x0x0x0$.

A further sweep gives

$xxx0xxx0$

and then

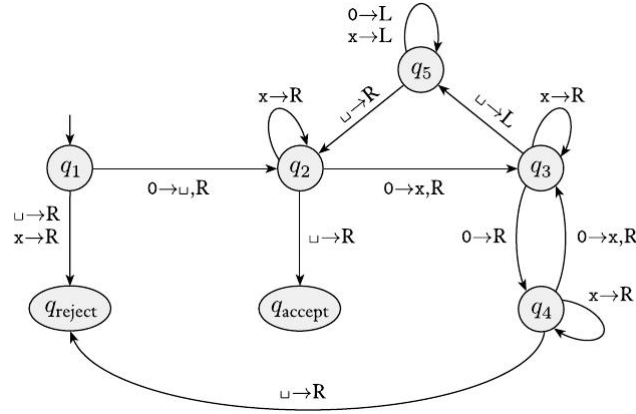
$xxxxxxx0$

leading to an acceptance of the input. What is the idea of the algorithm? It essentially uses the *integer factorization* of the length of the input string. For the above string with a length of 8, we have

$$8 = 2 \cdot 2 \cdot 2 .$$

In each sweep, it divides the number of zeros by two. If finally an uneven number remains, the initial size for the input string was not a power of two and it rejects. Otherwise, it accepts.

Now, we would like to transform the algorithm to an STD. It is given below:



The reader should take a moment to go through the STD and to map it to the original algorithm. As a short remark, it should be noted that the resulting TM does not *cross out* the first zero, but replaces it by a blank symbol “ \sqcup ” to indicate that this is the first symbol on the tape, making a detection of the left end of the tape easier.

To complete the definition of the TM, we give it formally by

$$M_2 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$$

with states $Q = \{q_1, \dots, q_4, q_{\text{accept}}, q_{\text{reject}}\}$ and alphabets $\Sigma = \{0\}$ and $\Gamma = \{0, x, \sqcup\}$. The transition function is supposed to be given by the above STD.

If we take the string $w = 0000$ as an input, the TM M_2 goes through the following configurations:

$q_1 0000$	$\sqcup q_5 x 0 x \sqcup$	$\sqcup x q_5 x x \sqcup$
$\sqcup q_2 000$	$q_5 \sqcup x 0 x \sqcup$	$\sqcup q_5 x x x \sqcup$
$\sqcup x q_3 00$	$\sqcup q_2 x 0 x \sqcup$	$q_5 \sqcup x x x \sqcup$
$\sqcup x 0 q_4 0$	$\sqcup x q_2 0 x \sqcup$	$\sqcup q_2 x x x \sqcup$
$\sqcup x 0 x q_3 \sqcup$	$\sqcup x x q_3 x \sqcup$	$\sqcup x q_2 x x \sqcup$
$\sqcup x 0 q_5 x \sqcup$	$\sqcup x x x q_3 \sqcup$	$\sqcup x x q_2 x \sqcup$
$\sqcup x q_5 0 x \sqcup$	$\sqcup x x q_5 x \sqcup$	$\sqcup x x x q_2 \sqcup$
		$\sqcup x x x \sqcup q_{\text{accept}}$

△

Remark In the previous example, we had to overwrite the first symbol on the tape by a special symbol to mark the left end of the tape. There is also a way to avoid this, by having a more complex logic. The interested reader should take a moment to think about the necessary construction. △

This concludes our introduction to Turing machines. Nevertheless, we will still see many examples of Turing machines in the sections and chapters to come.

3.2 Variants of Turing Machines

In this section, we are going to introduce three further important computing models, namely multitape Turing machines, nondeterministic Turing Machines and Random Access Machines. For each of these models, we will discuss its equivalence to (standard) TMs.

3.2.1 Multitape Turing Machines

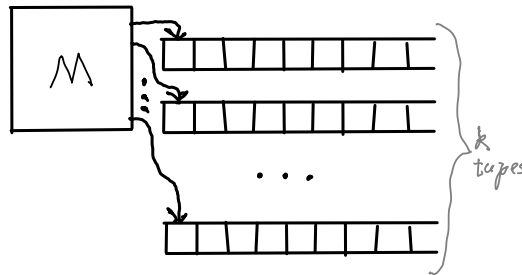
A *multitape Turing machine* is a Turing machine with several tapes and several tape heads. It is given by

Definition 3.12 (Multitape Turing machine) A **multitape Turing machine** with k tapes is a 7-tuple, $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the **blank symbol** \sqcup
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state,
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Let us discuss the definition in

Remark (Multitape Turing machine interpretation) Multitape TMs are, as said before, TMs with several tapes and several tape heads. A visual representation of the situation is given below.



In principle, a k -tape TM behaves like a TM. However, it can move the k tape heads on k tapes at the same time, can read from k tapes and write to k tapes. This results in the modified transition function, which has k -tuples of symbols read and written and a

k -tuple of moves to the left or to the right. The input to a multitape TM is placed on the first tape. All other behavior is identical to a TM. \triangle

Since the reader at this point of reading this work will have gained a fairly good general intuition of the definition of string acceptance and the language of a given automaton model, we just skip all these notions and immediately proceed to an example.

Example 3.4 We aim at finding a 2-tape TM that recognizes the language

$$L = \{ \#w\#w^R\# \mid w \in \{0,1\}^* \} ,$$

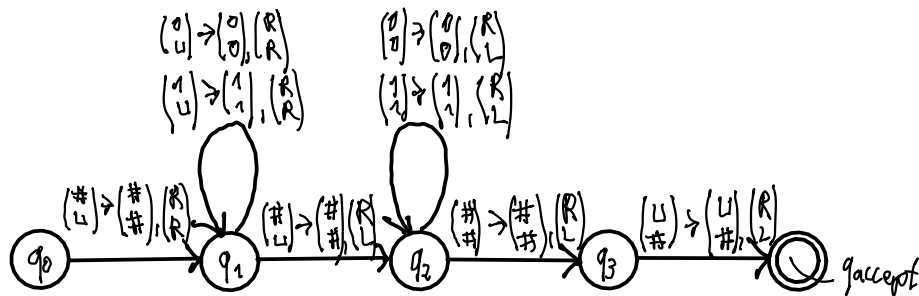
hence all strings that have two consecutive substrings (delimited by #s) over the alphabet $\{0,1\}$ such that the second substring is the first substring in reversed order.

The pseudo-algorithm for a two-tape TM M_3 recognizing this language is given by

$M_3 =$ “On input string s :

1. Copy the first part of the string s until the second # to the second tape.
If no two #s exist, *reject*.
2. Move the tape head of the first tape further to the right, while moving the tape head of the second tape back to the left while comparing the content of the first to the content of the second tape. If all symbols are identical until the first tape head reaches a blank symbol, *accept*. In any other case, *reject*.”

The following STD implements this multitape TM.



Note that this is again a slightly simplified STD, since we do not show the reject state. Indeed, we would have to add for all symbol pairs that are read from the two tapes and for which we do not have a transition in the STD a separate transition to the rejecting state.

If we analyze the STD, we observe that the loop in state q_1 is the main copy operation while the loop in state q_2 is the main comparison operation. The remaining transitions detect the ends of the substrings and of the overall input string.

We give the sequence of configurations that leads to the acceptance of the input string $\#10\#01\#$:

q_0	#	1	0	#	0	1	#	
q_0	#	\square	\square	\square	\square	\square	\square	
#	q_1	1	0	#	0	1	#	
#	q_1	\square	\square	\square	\square	\square	\square	
#	1	q_1	0	#	0	1	#	
#	1	q_1	\square	\square	\square	\square	\square	
#	1	0	q_1	#	0	1	#	
#	1	0	q_1	\square	\square	\square	\square	
#	1		0	#	q_2	0	1	#
#	1	q_2	0	#		\square	\square	\square
#		1	0	#	0	q_2	1	#
#	q_2	1	0	#	\square		\square	\square
	#	1	0	#	0	1	q_3	#
q_3	#	1	0	#	\square	\square		\square
	#	1	0	#	0	1	#	q_{accept}
q_{accept}	#	1	0	#	\square	\square	\square	

Note that we used here a notation in which we just “stack” two configurations on top of each other (with the same state) to cover the two-tape case. \triangle

We come to the main result connected to multitape TMs:

Theorem 3.1 A language is Turing-recognizable if and only if some multitape Turing machine recognizes it.

Hence this theorem establishes the equivalence of single-tape and multi-tape TMs. While it is obvious that there exists for every singletape TM a multitape TM that recognizes the same language, the other direction of this statement is indeed nontrivial and can be summarized by

Theorem 3.2 For every multitape Turing machine M , there exists a singletape Turing machine S that recognizes the same language.

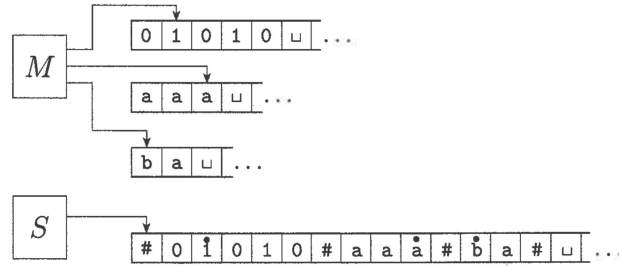
It should be noted that with growing complexity of the computing models that we are working with, the proofs will more and more use high-level arguments. More fine-grained correctness checks for some statements will be mostly left to the reader. Given this ob-

ervation, the following proof might have been called a “proof sketch” in earlier parts of this work but is now considered more as a proof:

Proof (sketch) for Theorem 3.2

Let M be an arbitrary but fixed k -tape TM that recognizes the language $L = L(M)$. We construct a standard TM S , such that $L(S) = L = L(M)$.

The main idea of the proof is to simulate the execution of the multitape TM M by S . The following figure sketches the main idea.



First of all, we need to put the content of the k different tapes on the one single tape of the TM S . This is achieved by sequentially putting the non-blank symbols of each tape as substrings with delimiters $\#$ on the single tape.

In addition, we need to find a way to “store” the tape head location for all tapes on the tape of S . This is achieved by extending the tape alphabet of the singletape TM such that for each symbol from the tape alphabet of M , we add a second symbol that has an additional marker, e.g. a dot on the top. We then indicate the location of the tape head by replacing the original symbol by the symbol with the marker, as shown above.

The pseudo-algorithm that then simulates the computation of M on S goes as follows:

Input: $w = w_1 \cdots w_n$:

1. Initialize tape of S :

$$\#w_1w_2\cdots w_n \underbrace{\#\dot{\sqcup}\#\dot{\sqcup}\#\cdots\#\dot{\sqcup}\#}_{k-1 \text{ times}}$$

2. For a single move of M do

- a) Scan over $k + 1$ $\#$ s to obtain symbols under k heads. The symbols are “stored” in the move to a specific state (e.g. a state q_{011} for a three tape TM with heads on symbols 0, 1, 1).
- b) Pass again over tape to apply changes following the transitions of M , which are encoded by new transitions in S .

The above algorithm does not yet cover the challenging issue that all k tapes of M are in principle infinite tapes. This, however, can be resolved as follows: Whenever a “dot”,

i.e. one of the tape heads, is moved in a move to the right on a delimiter symbol # (i.e. it would end up on a blank symbol on the original multi-tape tape), the resulting # is replaced by a \sqcup and a # and all remaining symbols are shifted by one entry.

This concludes the proof. It should be clear that the resulting TM S exactly mimics the computations of the k -tape TM M . \square

At this point, we should make a short

Remark (Practicability of multi- to singletape TM conversion) We should stress here that the above construction to convert a multitape TM to a singletape TM is mostly of theoretical use. Hence it guides us to the required existence result. However, even for very small multitape TMs with only a very small alphabet, the description of the resulting singletape TM will become tremendously large, i.e. infeasible. \triangle

This concludes our discussion of multitape TMs for now. Later, we will often use multitape TMs due to their flexibility of having several storage locations in between we can copy content.

3.2.2 Nondeterministic Turing Machines

Nondeterministic Turing machines are the rather obvious extension of the nondeterminism concept seen in NFAs for TMs. They are given by

Definition 3.13 (Nondeterministic Turing machine) A **nondeterministic Turing machine** (NTM) is a 7-tuple, $N = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the **blank symbol** \sqcup
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state.
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

The major difference is in the transition function, which includes transitions to several states or no state. ϵ -transitions, as we have seen them with NFAs are usually not part of the definition of an NTM. This allows us to use the same definition of a configuration as for deterministic TMs.

Definition 3.14 (Configuration of an NTM) Let $N = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ be a nondeterministic Turing machine, $u \in \Gamma^*$, $v \in \Gamma^\omega$ and $q \in Q$. The setting

$$u q v$$

is a **configuration** of the Turing machine, where

- u is the current tape content to left of the head,
- q is the current state, and
- v is the current tape content below (v_1) and to the right of the head (being terminated by blanks \sqcup).

The notions of a starting, accepting, etc. configuration are also identical to the deterministic case. Then the slightly different definition of the transition function leads to a small deviation in the *yields* relation.

Definition 3.15 Let $N = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ be a nondeterministic Turing machine, $a, b, c \in \Gamma$, $u \in \Gamma^*$, $v \in \Gamma^\omega$ and $q_i, q_j \in Q$. A configuration C_1 **yields** a configuration C_2 if N can legally go from C_1 to C_2 in a single step. Hence, we say

- (leftward move): $u a q_i b v$ **yields** $u q_j a c v$
if it holds $(q_j, c, L) \in \delta(q_i, b)$,
- (rightward move): $u a q_i b v$ **yields** $u a c q_j v$
if it holds $(q_j, c, R) \in \delta(q_i, b)$,
- (L-move, left tape end): $q_i b v$ **yields** $q_j c v$
if it holds $(q_j, c, L) \in \delta(q_i, b)$,
- (R-move, left tape end): $q_i b v$ **yields** $c q_j v$
if it holds $(q_j, c, R) \in \delta(q_i, b)$.

For NTMs, a configuration can yield several or no other configuration. This is different to TMs, where a configuration can yield only exactly one other configuration. Now, we can introduce the acceptance of a string and the resulting language by

Definition 3.16 (Accepted input & recognized language)

Let $N = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ be a nondeterministic Turing machine and $w \in \Sigma^*$ be its input. N **accepts** w , if a sequence of configurations C_1, C_2, \dots, C_k exists, such that

1. C_1 is start configuration on input w
2. C_i yields C_{i+1} , $i = 1, \dots, k-1$, and
3. C_k is accepting configuration.

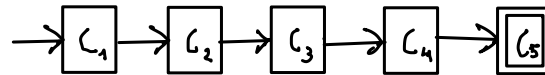
The **language of N** is

$$L(N) := \{w \in \Sigma^* \mid N \text{ accepts } w\}.$$

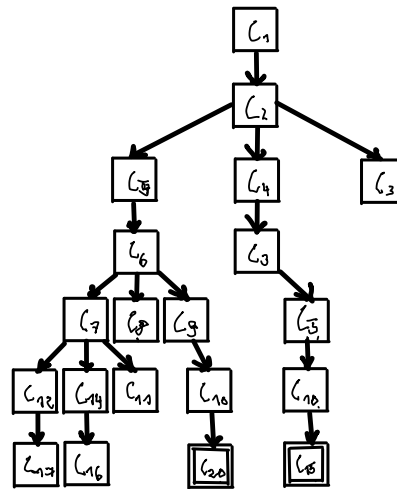
We say that $L(N)$ is **recognized** by N .

Due to the way in which we formulate the definition (“if a sequence [...] *exists*”), we here use indeed the exact same definition as for the deterministic case. However, it should be clear that the different transition function and therefore the different nature of the *yield* relation will still lead to a nondeterministic computation behavior as discussed in

Remark (Deterministic vs. nondeterministic TM) Standard, i.e. deterministic, TMs have a computation behavior as seen for FAs, i.e. we have a fully deterministic unique computation (branch) like



Nondeterministic TMs lead to many different computation branch that can be visualized in a tree like



The acceptance definition above indicates that we accept a string as soon as *one* computation branch in that tree is accepted, i.e. if there *exists* one accepting computation branch. \triangle

To clarify the notion of a *computation branch* further, we introduce as seen for previous machine models

Definition 3.17 (Computation branch of an NTM)

Let $N = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a non-deterministic Turing machine and $w \in \Sigma^*$ be its input. A **computation branch of N on w** is a sequence of configurations C_1, C_2, \dots , such that

1. C_1 is start configuration on input w
2. C_i yields C_{i+1} , $i = 1, \dots, k-1$, and

After this theory-heavy introduction, we come to an example that follows [?]

Example 3.5 We build an NTM N that recognizes the language

$$L = \{1^n \mid n > 2, n \text{ is not prime}\}.$$

Hence the language contains all numbers, in unary representation that are larger than two and not prime. It is well known that for an $n > 2$ with this condition it holds that there exist two integers $p, q \geq 2$ such that $n = p \cdot q$.

We use this observation to build an NTM that first non-deterministically “guesses” values p and q and then compares their product against n . If they match, the computation branch is accepted.

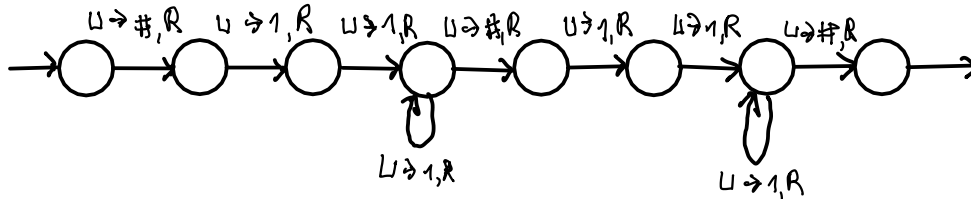
Note that a resulting NTM will indeed build a countable infinite number of computation branches, as it basically “tries out” all possible combinations of p and q .

Let us try to clarify the algorithmic idea of the necessary NTM further by simultaneously giving an algorithm and providing an example.

Input w given:	$w = 111111$
1) initialize tape with w	\downarrow 111111
2) go to end of tape	111111 \downarrow
3) add delimiter #	111111# \downarrow
4) nondeterministically write $p > 2$ ones after delimiter	111111#11 \downarrow
5) add delimiter	111111#11# \downarrow
6) nondeterministically write $q > 2$ ones after delimiter	111111#11#111 \downarrow
7) go two delimiters to the left	111111#11#111
8) multiply p and q	111111#11#111#111111 \downarrow
9) go to beginning	\downarrow 111111#11#111#111111
10) compare two strings	xxxxxx#11#111#xxxxxx \downarrow
11) <i>accept</i> , if strings are identical	<i>accept</i>

We will not give the full implementation in terms of an STD. However, we have a short look at the most interesting parts of the required STD.

First, we deal with the implementation of the nondeterministical “guessing” of the values p and q . The following partial STD realizes the steps 3) to 6) in the above algorithm.



The two non-deterministical loops can create an arbitrary number of 1s, as required here. We have seen the implementation of step 10) already in Example 3.1 (despite the necessity to ignore the two binary values in the middle). It mostly remains the multiplication of two numbers, of which the implementation is left to the interested reader. \triangle

We come to the main result on NTMs in this section.

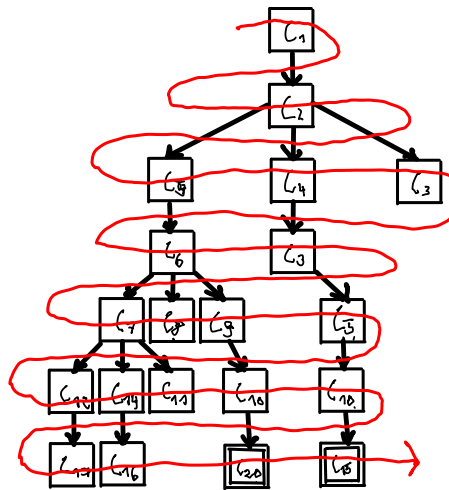
Theorem 3.3 A language is Turing-recognizable if and only if some nondeterministic Turing machine recognizes it.

As before, the direction from a TM to an NTM is immediately clear. Therefore, our main task is to prove

Theorem 3.4 For every nondeterministic Turing machine N , there exists a deterministic (singletape) Turing machine S that recognizes the same language.

Proof

Before we go into the details, we sketch the high-level idea: First, we consider the tree of calculation branches of a given NTM, as discussed before. Then, we build a multitape TM that traverses that tree from the starting configuration (i.e. the root) by breadth-first-search to find a accepting configuration.



Now, let us enter the actual proof:

The diagram shows a Turing machine M with three tapes:

- input tape:** A sequence of cells containing 0, 0, 1, 0, followed by a blank cell (□) and an ellipsis (...).
- simulation tape:** A sequence of cells containing x, x, #, 0, 1, x, followed by a blank cell (□) and an ellipsis (...).
- address tape:** A sequence of cells containing 1, 2, 3, 3, 2, 3, 1, 2, 1, 1, 3, followed by a blank cell (□) and an ellipsis (...).

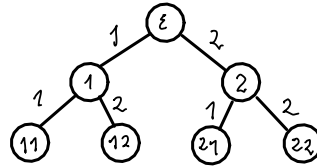
Arrows indicate that the machine M is connected to all three tapes.

These addresses are found as follows: Let d be the maximum outgoing degree over all nodes in the STD that describes the transition function of N , i.e.

$$d = \max_{q \in Q, a \in \Gamma} |\{(q', b, D) | (q', b, D) \in \delta(q, a)\}|.$$

³⁾What this exactly means, will become clear soon.

Then the content of the address tape is a string $\alpha \in \{1, \dots, d\}^*$. This address α represents a node in the computation tree as follows:



Hence each node will be named following the sequence of edges along which it is reached. Note that it can also happen that such an address represents a non-existing node.

The algorithm of the deterministic multitape TM M is as follows:

On input w :

1. Place w on input tape.
2. Loop over all possible strings $\alpha \in \{1, \dots, d\}^*$ on address tape in “string ordering”, i.e. enumerate following the length (e.g. for $d = 2$ we get the order $\emptyset, 1, 2, 11, 12, 21, 22, 111, 112, \dots$).
 - a) Copy input tape to simulation tape.
 - b) Execute on the simulation tape the deterministic sequence of configurations starting from the computing tree root to the configuration (node) indicated by the address α on the address tape.
 - c) *Accept*, if the reached configuration described by α is, according to the logic of the NTM N an accepting configuration.
 - d) *Reject*, if all configurations on the same tree level are rejecting configurations or a rejection happens along the branch to reach them.

The above deterministic multitape Turing machine M will accept, if and only if there is an accepting configuration in the original NTM N . After conversion to an equivalent TM, we get the result. \square

Note that, in principle, we would not need step 2. d) in the algorithm from the above proof, since only the acceptance is of relevance. Adding that rule further allows to obtain a decider, if the original NTM is a *decider*, which brings us to

Definition 3.18 (Nondeterministic decider) We call a nondeterministic TM N a **decider**, if all valid computation branches halt on all inputs $w \in \Sigma^*$. A decider N that recognizes $L(N)$ is said to **decide** $L(N)$.

With step 2. d) we thus also get

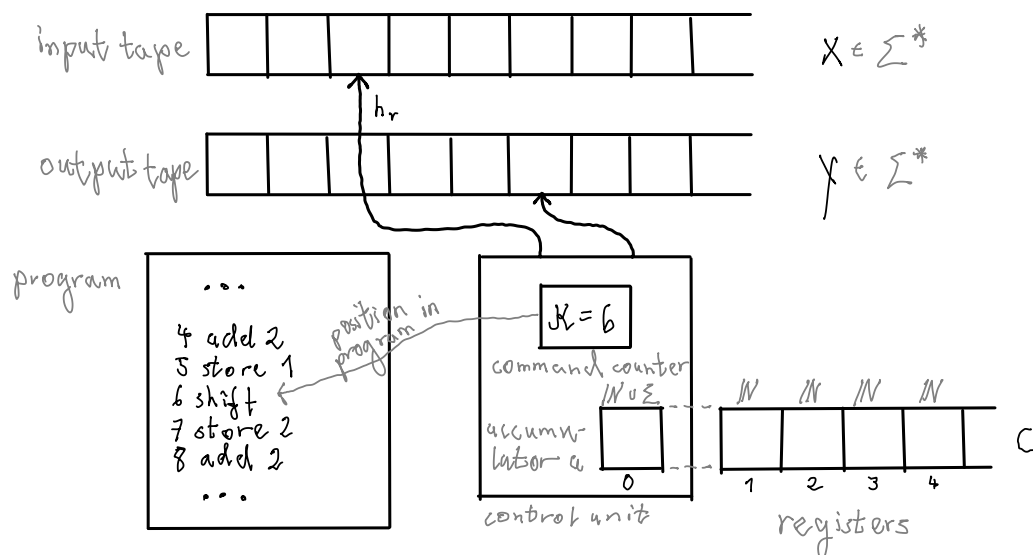
Theorem 3.5 A language is decidable if and only if some nondeterministic Turing machine decides it.

This concludes, for now, the discussion of NTMs, for which we have just established the equivalence to deterministic TMs.

3.2.3 Random Access Machines

The third equivalent machine model that we are going to discuss here are *random access machines* (RAMs). The discussion follows []. RAMs come closest to the classical von Neumann architecture and mimic the behavior of a very simple modern computer. They are introduced by

Remark (Structure of a RAM) The basic structure of a random access machine is given below.



The RAM consists of a control unit that can execute commands, which it gets from a *program*. The program is a finite sequence of commands. The control unit keeps track of the position in the program by a command counter κ . Moreover, the control unit has access to an input tape x over an alphabet Σ and an output tape over the same alphabet. The input tape is a read-only tape, hence only a single symbol can be read at a time. Once it is read, the tape head at location h_r moves to the right and cannot come back. The output tape is write-only, i.e. we can only append symbols to the right-hand side end of the tape. Input and output tape replace I/O devices such as keyboard, screen and/or printer.

In addition to the different tapes, the control unit also has random access⁴⁾ to registers $c(0), c(1), c(2), \dots$, which are modeled as a function $c : \mathbb{N}_{>0} \rightarrow \mathbb{N} \cup \Sigma$. The registers can take values from \mathbb{N} and can be accessed in a read-write manner. Initially, all registers are set to 0. Register $c(0)$ plays a special role. It is called *accumulator* and is also abbreviated by a . It can take – both – values from the tape alphabet Σ and the natural numbers and is the main memory location on which arithmetic operations are carried out. \triangle

Obviously, we did not formally define a RAM. Let us comment on that using the following

Remark (Formalization) In the previous sections and chapters, we have become quite used to give formal definitions for all components of a computing model. We will skip this for RAMs, as a true formalization of a RAM is quite involved and we will not use RAMs in later sections and chapters. \triangle

Similarly, we will also stick to a rather informal approach to define the concept of *program execution* of a RAM.

Remark (Program execution) If a program is executed by a RAM, the RAM goes through a sequence of configurations of the form

$$C_i = (\kappa, a, c, x, h_r, y).$$

Hence each configuration is defined by the command counter κ , the content of the accumulator a , the content of the registers c , the content of the input tape x , the position h_r of the tape head on the input tape and the content of the output tape y .

To get from a configuration C_i to C_{i+1} , the RAM

1. reads the command at position κ in the program,
2. (maybe) reads one symbol from the input tape, doing $h_r = h_r + 1$ afterwards,
3. (maybe) reads the content of register(s) and/or accumulator,
4. (maybe) changes the content of either register or accumulator,
5. (maybe) outputs one symbol, and
6. changes the command counter or stops.

Which part of the given “(maybe)” steps is actually executed, depends on the read command. Below, we give a possible list of commands including the resulting actions and their interpretation.

Command	Action	Interpretation
load i	$a = c(i); \kappa = \kappa + 1$	load value from register into accum.
store i	$c(i) = a; \kappa = \kappa + 1$	store value from register into accum.
add i	$a = a + c(i); \kappa = \kappa + 1$	add value from register to accum.

⁴⁾This is what causes the name of this model.

sub i	$a = \max(a - c(i), 0); \kappa = \kappa + 1$	subtract value from reg. from accum.
read	$a = x_{h_r}; h_r = h_r + 1; \kappa = \kappa + 1$	read symbol from input tape
print a	$y = ya; \kappa = \kappa + 1 \ (a \in \Sigma)$	write symbol to output tape
shift	$a = \begin{cases} a/2 & \text{if } a \text{ even} \\ (a-1)/2 & \text{else} \end{cases}; \kappa = \kappa + 1 \ (a \in \mathbb{N})$	perform bit-shift on accum.
goto i	$\kappa = i \ (i \in \mathbb{N}_{>0})$	jump to program line i
goto a	$\kappa = a \ (a \in \mathbb{N}_{>0})$	jump to program line a
if a=i goto j	if $a = i$ then $\kappa = j$, else $\kappa = \kappa + 1 \ (i \in \mathbb{N} \cup \Sigma)$	conditional program line jump
ind load i	$a = c(c(i)); \kappa = \kappa + 1 \ (a \in \mathbb{N})$	indirect load
ind store i	$c(c(i)) = a; \kappa = \kappa + 1 \ (a \in \mathbb{N})$	indirect store
ind add i	$a = a + c(c(i)); \kappa = \kappa + 1 \ (a \in \mathbb{N})$	indirect addition
ind sub i	$a = \max(a - c(c(i)), 0); \kappa = \kappa + 1 \ (a \in \mathbb{N})$	indirect subtraction
load const i	$a = i; \kappa = \kappa + 1 \ (i \in \mathbb{N})$	load constant into accum.
add const i	$a = a + i; \kappa = \kappa + 1 \ (i \in \mathbb{N})$	add constant to accum.
sub const i	$a = \max(a - i, 0); \kappa = \kappa + 1 \ (i \in \mathbb{N})$	subtract constant from accum.
end	stop	terminate program execution

△

Readers that know assembly / machine language will observe that the above commands are very similar. In fact, the just introduced RAM model is indeed a very simple computer with a puristic (machine) instruction set.

Let us come to an example of a RAM.

Example 3.6 We build a RAM to add up two numbers $p \in \mathbb{N}$ and $q \in \mathbb{N}$, which are given as strings $u \in \{0, 1\}^n$ and $v \in \{0, 1\}^n$ in binary representation. Hence we choose the tape alphabet $\Sigma = \{0, 1, \#\}$ and assume that the input tape initially is given by

$$x = u\#v\#$$

where we have $p = \sum_{i=1}^n u_i 2^{n-i}$. The algorithmic idea of the RAM implementation follows the steps

1. Convert u to integer p and store p in a register.
2. Convert v to integer q and store q in a register.
3. Add p and q .
4. Write $p + q$ to output tape as binary number.

We first analyze how to carry out the conversion from binary number to an integer. We use the example of the binary string 1011, which corresponds to the integer $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 2^0 = 11$ with a final delimiter, thus we deal with an input $w = 1011\#$. In a non-RAM pseudo code, we could implement the conversion as

```

p = 0
i = 1
while  $w_i \neq \#$  do
     $p = 2p + w_i$ 
     $i = i + 1$ 

```

For the above input string, we would thus get

i	p
	0
1	1
2	$1 \cdot 2 + 0 = 2$
3	$1 \cdot 2 \cdot 2 + 0 \cdot 2 + 1 = 5$
4	$1 \cdot 2 \cdot 2 \cdot 2 + 0 \cdot 2 \cdot 2 + 1 \cdot 2 + 1 = 11$

Next we give an implementation in terms of a RAM program for the reading and addition of the values p and q .

line	RAM command	low-level interpretation	high-level interpretation
1	read	$a = w_i$	
2	if a=# goto 9		while $w_i \neq \#$ do
3	store 3	$c(3) = a$	
4	load 1	$a = c(1)$	
5	add 1	$a = a + c(1)$	$c(1) = 2c(1) + w_i$
6	add 3	$a = a + c(3)$	
7	store 1	$c(1) = a$	
8	goto 1		
9	read	$a = w_i$	
10	if a=# goto 17		while $w_i \neq \#$ do
11	store 3	$c(3) = a$	
12	load 2	$a = c(2)$	
13	add 2	$a = a + c(2)$	$c(2) = 2c(1) + w_i$
14	add 3	$a = a + c(3)$	
15	store 2	$c(2) = a$	
16	goto 9		
17	load 1	$a = c(1)$	
18	add 2	$a = a + c(2)$	$a = c(1) + c(2)$

What remains is to convert back the value in the accumulator a to binary representation. The implementation of this task is left to be done by the interested reader. \triangle

In principle, we would like to show that TMs and RAMs are equivalent. However, the equivalence of these machines needs to be carefully formulated. Let us consider the direction from a TM to a RAM:

Theorem 3.6 Let M be a single-tape TM. There exists a random access machine that terminates and gives as output 1, whenever M accepts a given input string w and that terminates and gives as output 0 or loops forever, whenever M rejects a given input string w .

Here, we translated the notion of acceptance in the “world of RAMs” by writing a 1 for acceptance on the output tape.

Intuitively, it should be clear that the above statement is correct, since a RAM seems to be “more powerful” than a TM. A proof would require to give a RAM that implements a TM. It is possible to do this, however very tedious. Therefore, we skip the proof.

The other direction of the equivalence is given by

Theorem 3.7 Let R be a random access machine with an arbitrary but fixed given program. For an arbitrary given input $x \in \Sigma^*$ it is assumed to compute an output $y(x) \in \Sigma^*$, if it terminates. There exists a multi-tape Turing machine that accepts the input $x \in \Sigma^*$ after writing $y(x) \in \Sigma^*$ on one of its tapes.

In this statement, we had to come up with a meaningful translation from concepts in a RAM to concepts in a TM. We match these concepts by considering a TM with at least two tapes and define the equivalence via the ability to map an input on one tape to an output on another tape in the same way as the RAM.

We also skip this proof, since it is an even more tedious task. In fact, we would have to “implement” the logic of each of the commands of the RAM on a TM. This is all possible, but way too involved to be discussed, here.

As a summary of the two previous theorems, we state

Theorem 3.8 Turing machines and random access machines have the same expressive power.

This statement has to be understood in the sense of the two previous interpretations of translations between RAMs and TMs. Overall, we have just established that TMs are as powerful as a simple model for a modern computer.

4 Decidability

In the previous chapters, our main focus was on the description of (formal) languages that are recognized by some computing model. We also learned that these formal languages are put in a hierarchy by the Chomsky Hierarchy. With the last chapter, we identified the computing model associated to the highest level of the Chomsky Hierarchy, a Turing machine. Moreover, we even figured out that TMs are equivalent to some very simple computer that follows the von Neumann architecture.

Having reached this most relevant level of computing model, we now make a slight shift in our objective. As we know that we have a model for a simple computer, we now try to map and understand notions that we know from the “computer world” in context of Turing machines.

One of these notions is the *algorithm*. Certainly, the reader is expected to know the properties of an algorithm. Still, let us give a rough idea: An algorithm is a (written down) sequence of deterministic operations, which compute for a given input an output. Often, we further require that the algorithm terminates after a finite number of operations.

The question is, how can we associate algorithms to Turing machines? A partial answer to this is the *Church-Turing thesis*.

Remark (Church-Turing thesis) On the one side, there exists an intuitive idea of an algorithm, as discussed before. On the other side, there exists a formalization of computation via (non-)deterministic TM, random access machines and other models. In 1936, Alonso Church and Alan Turing postulated the hypothesis / assumption that both concepts are equal. This statement is called *Church-Turing thesis*. \triangle

Indeed, we have no proper means to prove such a statement formally, as an algorithm in general is not a fully formalized object. Therefore, it is common sense to accept this hypothesis in Computer Science.

Accepting that we have an equivalence between general algorithms and e.g. Turing machines, we might next ask us, what are *problems* that can be solved by a computer? Intuitively, we want to call a problem *solvable*, if there exists an algorithm that terminates after a finite number of operations and that carries out that task / solves the problem. Following the Church-Turing thesis, we thus essentially ask for a decider, i.e. a Turing machine that always terminates, that solves the problem. And as the language class associated to deciders are the decidable languages, we call all those problems *solvable* for which there exists a decidable language that describes it. On the opposite site, we associate *unsolvable* problems with undecidable languages.

The main objective of this chapter is to identify several solvable and unsolvable problems and to prove that they are (un-)solvable. In terms of languages, this thus means that we study decidable and undecidable languages.

The remainder of this chapter is structured as follows. First we give examples of decidable languages, which we will mostly find in context of FAs and CFGs. Afterwards, we introduce and prove the most well-known undecidable problem in theory of computation, the *halting problem*. Since we want to be able to “re-use” statements on undecidability of languages to prove that other languages are undecidable, we further introduce *mapping reductions* that allow for such a knowledge transfer. We conclude the chapter by having a look at undecidable languages.

4.1 Decidable languages

In this section, we will establish that regular and context-free languages are decidable. Before that, we will however need to make a remark on a slight shift in our way to deal with proofs in context of Turing machines.

Remark (Algorithms, notation and proof details) The upcoming languages will often describe rather complex objects, like graphs, finite automata, etc. If we want to work with such objects in a Turing machine, we somehow have to translate these objects to a string representation. For natural numbers, we have seen in the previous chapter that a natural number can be represented by e.g. an unary or binary representation. However, carrying out even simple arithmetic operations on these numbers, might lead to tremendously complex Turing machines.

Therefore, we now strongly simplify the notation and the implementation of Turing machines in proofs by agreeing on the fact that we will only use (pseudo-)algorithms to describe future Turing machines. Moreover, for complex objects O_1, O_2, \dots we will use the notation $\langle O_1 \rangle, \langle O_2 \rangle, \dots$ that indicates that they are in a meaningful way represented by a string. As a result, proofs will be on a much higher abstraction level. Thus, we assume that the reader would, in principle, be able to carry out the skipped parts by her-/himself. \triangle

Let us come to our first decidable language.

Theorem 4.1 The language

$$A_{FA} = \{ \langle B, w \rangle \mid B \text{ is a deterministic FA that accepts input string } w \}$$

is decidable.

Coming back to the previous remark, this language thus contains all strings that describe a combination of a deterministic FA and input string, such that the input string is accepted by the FA. Translating this back to the language of “problems”, we could say that the problem of testing whether a given FA accepts an input string is decidable, thus solvable.

Proof

We build a 3-tape TM M that accepts the input string $\langle B, w \rangle$ if w is accepted by the FA B . Otherwise it rejects it. Certainly, M also rejects all invalid inputs.¹⁾

The first tape (T1) of the 3-tape TM initially contains the input $\langle B, w \rangle$. After an initialization, only w will remain on the this tape. The second tape (T2) contains after the initialization

$$\langle \text{current state} \rangle \# \langle \text{accepting state} \rangle \langle \text{accepting state} \rangle \dots ,$$

hence the current state and the list of all accepting states. The third tape (T3) contains, after the initialization, the transition function of B , represented as a list of transition rules. A single transition rule $\delta(q_i, a) = q_j$ is represented as an entry of the form

$$\# \# \langle i \rangle \# a \# \langle j \rangle .$$

An algorithm for the TM M is give by

On input $\langle B, w \rangle$:

1. Initialize the tapes.
2. If the symbol under the head of T1 is \sqcup , compare current state on T2 to list of accepting states on T2. If it matches, *accept*, otherwise *reject*.
3. Read symbol under the head of T1 and move the head to the right.
4. Find for current state on T2 and current input symbol the transition in T3.
5. Store new state to T2.
6. Go to 1.

From the algorithm, it should be clear that the TM is able to correctly check the acceptance of the input string. Due to step 2. and the finite computation of an FA for a given input, TM M will also always halt. Hence it is a (multitape) decider²⁾ for the above language. \square

The same result can also be obtained for NFAs.

Theorem 4.2 The language

$$A_{NFA} = \{ \langle B, w \rangle \mid B \text{ is an NFA that accepts input string } w \}$$

is decidable.

¹⁾Here and in the following, we always assume that the first operation of each TM is to check, whether the input follows all required specifications and to reject, if this is not the case.

²⁾As seen before, we can always find for a multitape decider a (singletape) decider that recognizes the same language. Hence we in the following no longer stress, whether it is a singletape or a multitape decider.

Here, we can either do the same construction as for FAs or we use following proof.

Proof Build an appropriate³⁾ multi-tape TM N with the following algorithm:

On input $\langle B, w \rangle$:

1. Read $\langle B, w \rangle$.
2. Convert $\langle B, w \rangle$ to the description of an equivalent FA C giving $\langle C, w \rangle$.
3. Run TM M from the proof of Theorem 4.1 with input $\langle C, w \rangle$.
4. Use *accept* / *reject* from M .

Hence, we convert the input and reuse the already constructed machine. The resulting machine obviously solves the correct problem and terminates due to the terminating conversion and the decider property of M . \square

With a similar conversion strategy, we can also prove

Theorem 4.3 The language

$$A_{REX} = \{ \langle R, w \rangle \mid R \text{ is a regular expression that generates string } w \}$$

is decidable.

Certainly, we are not limited to string acceptance / membership decidability statements. Another interesting task is to check whether a given FA only recognizes the empty language. This problem is also “solvable”, as we see in

Theorem 4.4 The language

$$E_{FA} = \{ \langle A \rangle \mid A \text{ is a FA and } L(A) = \emptyset \}$$

is decidable.

Proof

We build an appropriate multitape-TM with the algorithm

³⁾Again, we assume here that the reader develops a feeling for how a multi-tape TM should be constructed.

On input $\langle A, w \rangle$:

1. Store A on tapes.
2. Mark start state.
3. Mark any state that has a transition coming into it from any already marked state.
4. If at least one new state was marked, go to 3.
5. If non of the accept states are marked, *accept*, else *reject*.

To mentally visualize the idea of this algorithm, we can also imagine to consider the STD of the FA A , i.e. a directed graph. Then the subsequent “marking” of states corresponds to solving the reachability problem for the accept state(s). If the accept state is reachable, the recognized language is not the empty language, and we reject.

Due to step 4. and the finite number of states in the FA, the TM will terminate and, by construction, gives the correct answer. \square

Another decidable problem is the check for equality of the recognized languages for two FAs.

Theorem 4.5 The language

$$EQ_{FA} = \{\langle A, B \rangle \mid A \text{ and } B \text{ are FAs and } L(A) = L(B)\}$$

is decidable.

We leave the proof as a homework for the interested reader.

Next, we have a look at context-free grammars. We claim that the check for a string to be generated by a given CFG is also a “solvable problem”.

Theorem 4.6 The language

$$A_{CFG} = \{\langle G, w \rangle \mid G \text{ is a CFG that generates string } w\}$$

is decidable.

To carry out the proof, we need the small

Lemma 4.1 Let G be a CFG in Chomsky normal form and $w \in L(G)$ with $|w| =: n \geq 1$. Any derivation of w will have $2n - 1$ steps.

Indeed, we do not need this exact number of steps. However, we will need that the number of steps for the derivation is finite, which is certainly guaranteed by this lemma. We skip the proof of the supportive lemma and come to the proof for Theorem 4.6.

Proof of Theorem 4.6

We build a multi-tape TM with the following algorithm:

On input $\langle G, w \rangle$:

1. Store G, w appropriately on tapes.
2. Convert G into an equivalent CFG in Chomsky form.
3. If $|w| \geq 1$, enumerate all possible derivations that can be generated in $2n - 1$ steps. Otherwise, if $|w| = 0$, enumerate all derivations with one step. In both cases, compare all enumerated strings to w and *accept*, if there is a match, otherwise *reject*.

Step 2. can be carried out in finite time as we have seen it in the various proof parts of Lemma 2.4. Moreover, Lemma 4.1 guarantees that w , if it is in the language generated by G , is enumerated in step 3. This assures both, the correctness of the constructed TM (i.e. it recognizes A_{CFG}) and termination of the TM. \square

The TM constructed in the above proof motivates us to make a

Remark (Efficiency) It is very important to notice that the statement that a problem/language is decidable is by no means a statement on the efficiency of a potentially existing TM that decides the problem/language. Taking the example of the MTM built in the proof of Theorem 4.6, one might come up with a more efficient approach that does not require to enumerate *all* derivations of the given length. However, for decidability, this is not relevant. The search for and classification of *efficient* TMs for given problems is considered in Chapters ?? and ??. \triangle

Similar to Theorem 4.4, we also observe that we can decide the problem of checking whether a given CFG generated the empty language.

Theorem 4.7 The language

$$E_{CFG} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$$

is decidable.

The proof is very similar to the one for Theorem 4.4.

Proof

We build a multi-tape TM with the following algorithm:

On input $\langle G \rangle$:

1. Store G appropriately on tapes.
2. Mark all terminal symbols.
3. Mark all variables A for which there is the rule $A \rightarrow U_1 U_2 \cdots U_k$ and at the same time all U_1, U_2, \dots have been marked before.
4. If at least one new variable has been marked, go to step 3.
5. If the start variable has not been marked, *accept*, otherwise *reject*.

Correctness and the halting property are immediately clear, due to the proof construction from Theorem 4.4. \square

Out of routine, we might be tempted to next state that

$$EQ_{CFG} = \{\langle G, H \rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$$

is decidable. However, this is *wrong*. As we will discuss later, it is undecidable.

Recall from previous chapters that we established subset relationships between two language classes A and B , i.e. $A \subseteq B$. To achieve that, we showed that for an arbitrary but fixed language in class A (for which we assumed to know the automaton from class A that recognizes it) we can find an automaton associated to language class B that recognizes the language. Keeping that view in mind, we can state

Theorem 4.8 The set of context-free languages is contained in the set of decidable languages.

Proof

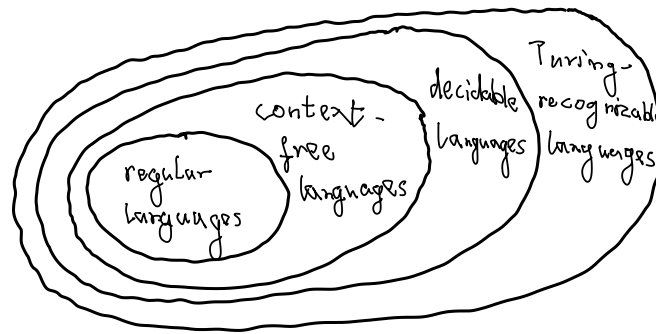
Let L be a context-free grammar. Therefore, there exists a PDA P , such that $L(P) = L$. Moreover, according to Lemma 2.4, there also exists a CFG G such that $L(G) = L(P) = L$. We build a (multitape) decider D , such that $L(D) = L(G) = L$ with the following algorithm:

On input w :

1. Replace w by $\langle G, w \rangle$ on the first tape.
2. Launch the decider built for A_{CFG} with the input from the first tape.
3. *Accept*, if the decider accepts, otherwise *reject*.

Note that in the algorithm, we “hard code” G as the task was only to give some decider that recognizes L . We do not ask for a Turing machine that converts a CFG or a PDA. The correctness of the constructed TM is clear. Certainly, it also terminates, since we use a previously build decider. \square

This result establishes another (new) view on the “world of languages”:



4)

4.2 Undecidability of the halting problem

In this section, we start into studying undecidable problems. Specifically, we would like to establish the following fundamental result in computability theory.

Theorem (Undecidability of the halting problem) The language

$$HALT_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

is undecidable.

In other words, the problem of figuring out, whether a given Turing machine halts on a given input is not “solvable” (in terms of having a decider for it).

To be able to prove this statement, we will first briefly recall the notions of *countable* and *uncountable* sets. The proof for a statement on a famous uncountable set will follow a so-called *diagonalization argument*, which is an important proof technique for undecidability. Specifically, we will use this methodology to first show the undecidability of the language

$$A_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w\}.^{5)}$$

⁴⁾Actually, we did not prove the full picture, yet, as we did not yet show that decidable languages are a proper subset of Turing-recognizable languages. We will come back to this at the end of this chapter.

⁵⁾The attentive reader should notice here that, while A_{FA} and A_{CFG} were decidable, A_{TM} is “suddenly” *not* decidable.

The knowledge on the undecidability of this language is finally used to show the undecidability of the halting problem.

We start by reminding ourselves of the mathematical formalism to compare the “size” of two sets.

Definition 4.1 Let A, B be sets. A and B have the **same size / same cardinality**, if there exists a bijective function f with $f : A \rightarrow B$.

Readers that might feel uncomfortable with the notion of an *injective*, *surjective* or *bijective* function should briefly refer to Section 0.1.2. Let us give a very quick

Example 4.1 The set of natural numbers $\mathbb{N} = \{1, 2, \dots\}$ and the set of even numbers $\{2, 4, \dots\}$ have the same cardinality. We see that by giving the bijection

$$f(n) = 2n.$$

△

Next, we introduce two important “size classes” for sets.

Definition 4.2 A set A is **countable**, if it is either finite or has the same size as \mathbb{N} . Otherwise it is **uncountable**.

A typical further example (besides of the even numbers above) for a countable set is given in

Lemma 4.2 The set \mathbb{Q} of rational numbers is countable.

The proof for this lemma can be found in introductory academic books to mathematics. The well-known example for an uncountable set is the set of real numbers.

Theorem 4.9 The set \mathbb{R} of real numbers is uncountable.

We will give a proof for this statement, since the proof uses the so-called *diagonalization argument* that we will later also use to prove that A_{TM} and thereby $HALT_{TM}$ is undecidable.

Proof

The proof is done by contradiction. We assume that there exists a bijection between

the natural and real numbers, $f : \mathbb{N} \rightarrow \mathbb{R}$. Consequently, each $n \in \mathbb{N}$ has an associated real value $y^{(n)} = f(n)$, $y^{(n)} \in \mathbb{R}$. Let us write down the decimal representation of this number with

$$y^{(n)} = y_0^{(n)} + \sum_{i=1}^{\infty} y_i^{(n)} \cdot 10^{-i},$$

where $y_0 \in \mathbb{Z}$ and $y_i^{(n)} \in \{0, \dots, 9\}$. Then, obviously, we can write down function f with its whole range \mathbb{R} as a table

n	$f(n)$				
1	$y_0^{(1)}$	\cdot	$y_1^{(1)}$	$y_2^{(1)}$	$y_3^{(1)}$...
2	$y_0^{(2)}$	\cdot	$y_1^{(2)}$	$y_2^{(2)}$	$y_3^{(2)}$...
3	$y_0^{(3)}$	\cdot	$y_1^{(3)}$	$y_2^{(3)}$	$y_3^{(3)}$...
\vdots	\vdots	\cdot	\vdots	\vdots	\ddots

Now, we construct a values $x \in \mathbb{R}$, for which it is guaranteed, that it is not in the range of f . To this end, we consult again the above table and select (in yellow) the following digits:

n	$f(n)$				
1	$y_0^{(1)}$	\cdot	$y_1^{(1)}$	$y_2^{(1)}$	$y_3^{(1)}$...
2	$y_0^{(2)}$	\cdot	$y_1^{(2)}$	$y_2^{(2)}$	$y_3^{(2)}$...
3	$y_0^{(3)}$	\cdot	$y_1^{(3)}$	$y_2^{(3)}$	$y_3^{(3)}$...
\vdots	\vdots	\cdot	\vdots	\vdots	\ddots

We now build a value x , such that it will be in each position different to those highlighted digits in the diagonal. Our constructed value is thus given by

$$x = \sum_{i=1}^{\infty} x_i \cdot 10^{-i}, \quad \text{with } x_i \neq y_i^{(i)} \text{ and } x_i \neq \{0, 9\} \quad \text{for all } i = 1, 2, \dots$$

If we compare that to the above, highlighted table, we observe that x is built such that it differs by at least one decimal digit from any other number in the range of f . Moreover, we excluded 0 and 9 to avoid ambiguous cases like $0.1\bar{9} = 0.2$. As a consequence of this construction, we have found $x \in \mathbb{R}$ such that it is not in the range of f . \nexists \square

Due to the geometric view seen above, the proof methodology is called *diagonalization argument*.

Next, we can prove the undecidability of A_{TM} , which we state in

Theorem 4.10 The language

$$A_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

is undecidable.

Proof

The proof goes by contradiction using a diagonalization argument. We assume that there exists a decider H with the functionality

$$H(\langle M, w \rangle) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } w \\ \text{reject} & \text{if } M \text{ does not accept } w \end{cases}.$$

Then we can also build a TM D that takes as input a TM as string, i.e. $\langle M \rangle$, calls H with $\langle M, \langle M \rangle \rangle$ and outputs the opposite of $H(\langle M, \langle M \rangle \rangle)$. Hence D is given as

$$D(\langle M \rangle) = \begin{cases} \text{accept} & \text{if } M \text{ does not accept } \langle M \rangle \\ \text{reject} & \text{if } M \text{ accepts } \langle M \rangle \end{cases}.$$

As we have countably many Turing machines, we can enumerate all of them as M_1, M_2, \dots . This allows us to write down a table that contains in cell (i, j) the result of applying Turing machine M_i to the input $\langle M_j \rangle$:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	\dots
M_1	<i>accept</i>		<i>accept</i>		
M_2	<i>accept</i>	<i>accept</i>	<i>accept</i>	<i>accept</i>	
M_3					\dots
M_4	<i>accept</i>	<i>accept</i>			
\vdots			\vdots		

Note that we did not write down *rejects* in this table, since these rejections might also be given by loops. Nevertheless, since we know that H is a *decider*, the following table that gives the answers of H , is indeed complete:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	\dots
M_1	<i>accept</i>	<i>reject</i>	<i>accept</i>	<i>reject</i>	
M_2	<i>accept</i>	<i>accept</i>	<i>accept</i>	<i>accept</i>	\dots
M_3	<i>reject</i>	<i>reject</i>	<i>reject</i>	<i>reject</i>	
M_4	<i>accept</i>	<i>accept</i>	<i>reject</i>	<i>reject</i>	
\vdots			\vdots		

Since D is a Turing machine, one of the rows of the table will be related to D (as we enumerate *all* TMs):

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	\dots	$\langle D \rangle$	\dots
M_1	<u>accept</u>	reject	accept	reject		accept	
M_2	accept	<u>accept</u>	accept	accept	\dots	accept	\dots
M_3	reject	reject	<u>reject</u>	reject		reject	
M_4	accept	accept	reject	<u>reject</u>		accept	
\vdots			\vdots		\ddots		
D	reject	reject	accept	accept		<u>?</u>	
\vdots			\vdots				\ddots

Let us analyze, what are the entries of the row associated to D . Recall for this that each diagonal element in this table contains the result of $H(\langle M_i, \langle M_i \rangle \rangle)$. If we carefully analyze the definition of D , we however see that D gives exactly the opposite answer to these diagonal elements. This is all well defined until we come to the row of D . There however, suddenly, D is supposed to reject $\langle D \rangle$ if and only D accepts $\langle D \rangle$. This self-contradictory definition leads to the contradiction. \nexists \square

We finally come back to our main result of this section.

Theorem 4.11 (Undecidability of the halting problem) The language

$$HALT_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

is undecidable.

Proof

The proof is carried out by contradiction. Let us assume that there exists a decider R , such that $L(R) = HALT_{TM}$. Given this decider, we construct a decider S with $L(S) = A_{TM}$, which works as follows:

On input $\langle M, w \rangle$:

1. Run decider R with input $\langle M, w \rangle$.
2. If R rejects, *reject*.
3. If R accepts, simulate M with input w until halting.
4. Use output of M .

Steps 2. and 3. “filter out” all inputs that could lead to an infinite loop. Therefore S always halts. Moreover, it recognizes the language A_{TM} since it accepts if M accepts. We conclude that S is a decider for A_{TM} . This, however, is in contradiction to the undecidability of A_{TM} . \nexists \square

4.3 Undecidability via mapping reductions

In the last section, we invested a bit of work to prove the undecidability of A_{TM} and thereby the undecidability of the halting problem. The proof for the undecidability of the halting problem implicitly transferred knowledge on A_{TM} to knowledge on $HALT_{TM}$.

In this section, we develop a methodology to make such a transfer explicit. Hence, we transfer knowledge on properties of a given language to other languages. We call this methodology a (*mapping*) *reduction* of one problem to another. We will see that, intuitively, if a problem A *reduces* to a problem B , we can use the solution to B to solve A . In some sense, solving A is therefore not harder than solving B . As soon as we have made that statement more precise, we will be able to prove that if A is undecidable and reducible to B , then B is undecidable. This will enable us to show undecidability for many problems without having to explicitly go via a complex argument like the diagonalization, again.

To introduce reductions, we have to do something that most likely many readers already had in the back of their minds: We need to extend our understanding of TMs beyond the level of an automaton that accepts or rejects to a machine that computes for an input an output.

Definition 4.3 A function $f : \Sigma^* \rightarrow \Sigma^*$ is a **computable function**, if some decider M , on every input w , halts with $f(w)$ on its tape.

Hence, whenever we construct *computable functions*, we mean that we construct a TM that executes the computation of that function. Certainly it also has to halt on all inputs. Therefore it is a decider. From a practical perspective, we will often consider multitape TMs for such applications and might get some input on the first tape and write the output to the second tape.

Example 4.2 The addition of two integer numbers in binary representation is a computable function. \triangle

We need a computable function in the central new object of this section:

Definition 4.4 Let A, B be languages over Σ . A is **mapping reducible** to B , short $A \leq_m B$, if there is a computable function $f : \Sigma^* \rightarrow \Sigma^*$, such that for all $w \in \Sigma^*$, it holds

$$w \in A \quad \text{iff} \quad f(w) \in B.$$

f is called **reduction** from A to B .

As soon as we have found a reduction from A to B , we therefore can “re-cast” the question of membership of w in A to the membership of $f(w)$ in B . Also, since it is an

“if and only if”, i.e. an equivalence, statement, we can also conclude from $w \notin A$ that $f(w) \notin B$.

So how does that help us? A first answer is given by

Theorem 4.12 Let $A \leq_m B$. If B is decidable, then A is decidable.

Proof

Let M be a decider for B and f be a reduction from A to B with decider F . A decider N for A is given the following algorithm:

On input $\langle w \rangle$:

1. Compute $f(w)$ by executing F with input w .
2. Run M on $f(w)$.
3. If M accepts, *accept*. Otherwise *reject*.

Since F and M are decidable, N is decidable. It remains to check, whether $L(N) = A$: If $w \in A$ it holds that $f(w)$ is in B and N , via M , returns *accept*. Also, if $w \notin A$, it holds that $f(w) \notin B$ and N , via M , returns *reject*. Therefore, N recognizes A . \square

While the above statement is certainly useful, we are mostly interested in a simple conclusion from it.

Corollary 4.1 Let $A \leq_m B$. If A is undecidable, then B is undecidable.

Proof

The statement that we want to prove is just the contraposition of the statement of Theorem 4.12. \square

A first example for the use of reductions to prove undecidability can be given, if we re-prove

Theorem The language

$$HALT_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

is undecidable.

Proof via reduction

We carry out the proof following the lines of Corollary 4.1. If we can build a reduction

from A_{TM} to $HALT_{TM}$, we are done. To this end, we need to construct a computable function f of the form

$$\begin{aligned} f : \Sigma^* &\rightarrow \Sigma^* \\ \langle M, w \rangle &\mapsto \langle M', w' \rangle \end{aligned}$$

such that

$$\langle M, w \rangle \in A_{TM} \Leftrightarrow \langle M', w' \rangle \in HALT_{TM}.$$

A TM F implementing the reduction f is given by the following algorithm:

On input $\langle M, w \rangle$:

1. If $\langle M, w \rangle$ does not follow the specification, output some $\langle M', w' \rangle$ such that M' will not halt on w' .
2. Construct a new machine M' with the algorithm:

On input x :

- a) Execute M on x .
- b) If M accepts, *accept*, if M rejects, enter infinite loop.

3. Output $\langle M', w \rangle$.

Obviously, F is a decider. Therefore, we only have to show that

$$\langle M, w \rangle \in A_{TM} \Leftrightarrow \langle M', w' \rangle \in HALT_{TM}$$

to establish that A_{TM} is mapping reducible to $HALT_{TM}$.

1. " \Rightarrow ":

This case is simple, as in case of $\langle M, w \rangle \in A_{TM}$, we know that M accepts w . In that case, also M' accepts w . Therefore, it halts on w . ✓

2. " \Leftarrow ":

We show the contraposition of the "backward direction", which is

$$\langle M, w \rangle \notin A_{TM} \Rightarrow \langle M', w' \rangle \notin HALT_{TM}.$$

Here, we have to go through the three possible reasons for $\langle M, w \rangle \notin A_{TM}$. First, $\langle M, w \rangle \notin A_{TM}$ could not follow the input specifications. This case is covered by step 1. of the algorithm of F , which produces an appropriate output that is not in $HALT_{TM}$.

Second, M could reject w . In that case, the produced M' will not halt on w , according to step b) in the definition of M' . Therefore, it holds $\langle M', w' \rangle \notin HALT_{TM}$.

Third, M could loop for w . In that case, the produced M' will already loop in step a) of its algorithm and it holds $\langle M', w' \rangle \notin HALT_{TM}$. ✓

We conclude that A_{TM} is mapping reducible to $HALT_{TM}$ and thus get via Corollary 4.1 the result. \square

We discuss another language, for which we can use a reduction from A_{TM} to prove undecidability.

Theorem 4.13 The language

$$REGULAR_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$$

is undecidable.

Proof

We build the mapping reduction $A_{TM} \leq_m REGULAR_{TM}$. The rest is done by Corollary 4.1.

The mapping reduction TM F is given by the following algorithm:

On input $\langle M, w \rangle$:

1. If $\langle M, w \rangle$ does not follow the specification, output some $\langle M_1 \rangle$ such that $L(M_1)$ is not a regular language.
2. Construct a new machine M_1 with the algorithm:

On input x :

- a) If x is of form $0^n 1^n$, *accept*.
- b) Otherwise run M on w and accept if M accepts w .^a

^aWe indeed run M on w and not on x , which is the input to M_1 .

3. Output $\langle M_1 \rangle$.

From reading the description of the decider F , it might not really be obvious, why it gives the correct reduction. We see the correctness of the reduction in the formal discussion of the required equivalence

$$\langle M, w \rangle \in A_{TM} \Leftrightarrow \langle M_1 \rangle \in REGULAR_{TM}.$$

1. “ \Rightarrow ”:

If $\langle M, w \rangle \in A_{TM}$, then we know that M accepts w . In the algorithm for M_1 , step b), M will therefore always accept w , thus M_1 will accept for *all* input strings x . Consequently the language of M_1 is $L(M_1) = \Sigma^*$. This, however, is a regular language. Therefore, it holds $\langle M_1 \rangle \in REGULAR_{TM}$.

2. “ \Leftarrow ”:

We prove the contraposition, i.e. $\langle M, w \rangle \notin A_{TM} \Rightarrow \langle M_1 \rangle \notin REGULAR_{TM}$. To this end, we go through all relevant cases.

First, $\langle M, w \rangle$ could not follow the specification. This case is covered by step 1. of the algorithm, which produces an M_1 with $\langle M_1 \rangle \notin REGULAR_{TM}$.

Second, M might reject w . In that case, step a) of the construction of M_1 makes sure that the language of M_1 contains at least $\{0^n 1^n | n \geq 0\}$. It does not contain any further symbols, as step b) will give for all other inputs a rejection. Thereby, $L(M_1) = \{0^n 1^n | n \geq 0\}$. which is not a regular language. Hence we obtain $\langle M_1 \rangle \notin REGULAR_{TM}$.

Third, M might loop. Here, the same reasoning as for the rejection case holds.

□

4.4 Turing-recognizable languages and beyond

In this section, we dare to have a look beyond the somewhat simple statement that a language is “not decidable”/undecidable. In essence, this statement only indicates that we cannot find a decider for the language. However, it does not give us yet an idea of the language classes that exist beyond decidable languages.

One of these language classes are certainly *Turing-recognizable languages*. This section will indicate that there exist indeed languages that are Turing-recognizable, but not decidable. Interestingly, we will be able to show that there exist languages that are even *not Turing-recognizable*. While our initial statement on that will be a non-constructive one, we will also give an example of a language that is not Turing-recognizable. Reductions will finally help us to extend this knowledge to other languages.

Let us start with a further analysis of the language A_{TM} . We state

Theorem 4.14 The language

$$A_{TM} = \{\langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w\}$$

is Turing-recognizable.

As we already know that A_{TM} is undecidable, this theorem establishes that indeed there exists a language that is Turing-recognizable, but not decidable. Thereby we obtain

$$\{L \text{ language} | L \text{ decidable}\} \subsetneq \{L \text{ language} | L \text{ Turing-recognizable}\},$$

thus that the set of decidable languages is a proper subset of the set of Turing-recognizable languages. Consequently, our earlier picture was indeed correct.

We immediately come to the very short proof of this theorem.

Proof

We build a TM U that recognizes the language A_{TM} using the following algorithm:

On input $\langle M, w \rangle$:

1. Simulate M on input w .
2. If M accepts, *accept*, if M rejects, *reject*.

Obviously it is possible to implement the operations of a Turing machine on a Turing machine. This concludes the proof. \square

So why is the constructed TM U not a decider? Indeed, if M runs forever on w , U will also not terminate. In other literature, the above constructed Turing machine, i.e. a TM that runs a TM on itself, is often called *Universal Turing Machine*.

In a next step, we would like to make another structural statement on formal languages. This time, we establish that there have to be languages outside of the class of Turing-recognizable languages.

Theorem 4.15 There exist languages that are not Turing-recognizable.

The result is obtained by comparing the cardinality of the set of all Turing machines and the cardinality of the set of all languages. Our first supportive lemma for this is

Lemma 4.3 Fixing a non-empty alphabet Σ , the set of all Turing Machines over Σ is countable.

The following proof sketch is partially based on [1].

Remark (Proof sketch) We first want to show that we can represent a given TM M (over alphabet Σ) by a finite binary string. To this end, we first map all objects in the description of M to integers of bound range or finite strings of such integers.

- We have finitely many symbols in Σ . These can be mapped to a finitely many different integers.
- The input string is finite and thus can be mapped a finite string of (bound) integers.
- We have finitely many states, which can be mapped to finitely many different integers.
- We have finitely many symbols in Γ , which can be mapped to finitely many different integers.
- L and R for “left” and “right” can be mapped to two different integers.

- One transition $\delta(q_i, a_j) = (q_k, a_l, L/R)$ can be mapped to a string of five integers.
- We have finitely many transitions, therefore the resulting string of strings of five integers is also finite.
- The choice of start, accept and reject state is further described by a total of three integers.

As we have a bound range for the integers, we can find for each of them a fixed-size binary representation. Then all above information can be stored in a potentially very long, but finite string of such finite-size binary strings. Therefore, each TM M can be described by a finite string of binary numbers.

Since we can simply enumerate all finite strings of binary numbers, the set of all Turing machines is countable. \triangle

Next, we discuss the cardinality of the set of all infinite binary sequences.

Lemma 4.4 The set $\mathcal{B} = \{0, 1\}^\omega$ of all infinite binary sequences is uncountable.

Proof

We assume that \mathcal{B} would be countable. Then, there would exist a bijection $f : \mathbb{N} \rightarrow \mathcal{B}$. With the diagonalization argument from the proof of Theorem 4.9, we can however then build a $b \in \mathcal{B}$, such that b is not in the range of f . \nexists \square

In order to complete the proof of Theorem 4.15, we show that the set of all languages has the same cardinality as the set \mathcal{B} , above, and immediately conclude from that

Lemma 4.5 Fixing a non-empty alphabet Σ , the set of all languages over Σ is uncountable.

Proof

We construct a bijection between $\mathcal{B} = \{0, 1\}^\omega$ and the set \mathcal{L} of all languages over Σ . Since Σ^* is countable, we can enumerate all strings in Σ^* , i.e.

$$\Sigma^* = \{s_1, s_2, s_3, \dots\}.$$

Let $f : \mathcal{L} \rightarrow \mathcal{B}$ be a function that associates a language $L \in \mathcal{L}$ to its “characteristic sequence”, i.e.

$$c_1 c_2 c_3 c_4 \dots \quad \text{with } c_i \in \begin{cases} 1 & s_i \in L \\ 0 & s_i \notin L \end{cases}.$$

Hence for each of the infinitely many, but countable strings $s_i \in \Sigma^*$ we state by an indicator 0/1, whether it is contained in L . To clarify this further, we give an example.

$$\begin{array}{rcl}
\Sigma^* & = & \{ \quad \varepsilon, \quad 0, \quad 1, \quad 00, \quad 01, \quad 10, \quad 11, \quad 000, \quad 001, \quad \dots \quad \} \\
L & = & \{ \quad \quad 0, \quad \quad 00, \quad 01, \quad \quad \quad 000, \quad 001, \quad \dots \quad \} \\
f(L) & = & \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad \dots
\end{array}$$

For the above function f , we observe that it is surjective since \mathcal{L} contains all potential combinations of strings from Σ^* , i.e. $f(\Sigma^*) = \mathcal{B}$. Moreover, it is injective by construction. As a consequence f is bijective, hence there exists a bijection between \mathcal{L} and \mathcal{B} . Since \mathcal{B} is uncountable following Lemma 4.4, we see that \mathcal{L} is also uncountable. \square

We conclude that Theorem 4.15 holds, since there are more languages than Turing machines, which establishes the relationship

$$\{L \mid L \text{ Turing-recognizable language}\} \subsetneq \{L \mid L \text{ language}\},$$

Now that we know that there are languages that are not Turing-recognizable, we would like to find an example for such a language. With a bit of work, we will establish that the complement of the language A_{TM} is not Turing-recognizable.

To come to this point, we need

Definition 4.5 We call a language L **co-Turing-recognizable** if \bar{L} is Turing-recognizable.

As an intermediate step, we further have to discuss

Theorem 4.16 A language is decidable if and only if it is Turing-recognizable and co-Turing-recognizable.

Proof

We prove both directions.

1. “ \Rightarrow ”:

Let L be a decidable language. By definition, it is also Turing-recognizable. The complement of L , thus $\bar{L} = \Sigma^* \setminus L$ is also Turing-recognizable (and decidable) since we can in the decider for L simply switch between acceptance and rejection of the output. ✓

2. “ \Leftarrow ”:

Let L, \bar{L} be Turing-recognizable with M_1 and M_2 being TMs that recognize these languages. We build a TM M that executes step-wise in parallel M_1 and M_2 on an input and accepts, if M_1 accepts and rejects, if M_2 accepts.

M is a decider since, for arbitrary input $w \in \Sigma^*$, it holds either $w \in L$ or $w \in \bar{L}$. Therefore M_1 or M_2 will accept and thus halt. Consequently M halts. Furthermore the language recognized by M is L , since M accepts if M_1 accepts. ✓ \square

The above theorem allows us to establish the anticipated result

Corollary 4.2 The language $\overline{A_{TM}}$ is not Turing-recognizable.

Proof

The contraposition of Theorem 4.16, gives that a language L is undecidable if and only if L is not Turing-recognizable or \overline{L} is not Turing-recognizable. Since A_{TM} is undecidable and L is Turing-recognizable following Theorem 4.14, we conclude that $\overline{A_{TM}}$ is not Turing-recognizable. \square

As for undecidable languages, we can also use mapping reducibility to transfer the knowledge on $\overline{A_{TM}}$ being not Turing-recognizable to other languages. To this end, we discuss

Theorem 4.17 Let $A \leq_m B$. If B is Turing-recognizable, then A is Turing recognizable.

The proof for this theorem follows exactly the structure of the proof for Theorem 4.12. Therefore, we skip it here. The immediate contraposition to this statement is

Corollary 4.3 Let $A \leq_m B$. If A is not Turing-recognizable, then B is not Turing-recognizable.

Hence, we have established

Lemma 4.6 Let B be a language over Σ . If it holds $\overline{A_{TM}} \leq_m B$, then B is not Turing-recognizable.

Let us further observe in the definition of mapping reducibility between a language A and a language B that we can consider the contraposition of the statement $w \in A \Leftrightarrow w \in f(B)$, which is $w \notin A \Leftrightarrow w \notin f(B)$. Thereby we obtain that $A \leq_m B$ is equivalent to $\overline{A} \leq_m \overline{B}$. We immediately conclude from that

Lemma 4.7 Let B be a language over Σ . If it holds $A_{TM} \leq_m \overline{B}$, then B is not Turing-recognizable.

This will allow us to extend our knowledge on a language to be not Turing-recognizable to the language EQ_{TM} (as anticipated earlier).

Theorem 4.18 The language

$$EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$$

is not Turing-recognizable.

Proof

As discussed before, we need to show $A_{TM} \leq_m \overline{EQ_{TM}}$. We build the TM F that performs the reduction as follows:

On input $\langle M, w \rangle$:

1. If $\langle M, w \rangle$ does not follow the specification, output ...
(*Do something useful.*)
2. Construct a new machine M_1 with the algorithm:

On input x :

a) *Reject.*

3. Construct a new machine M_2 with the algorithm:

On input x :

a) Run M on w . If it accepts, *accept.*

4. Output $\langle M_1, M_2 \rangle$.

Obviously, F is a decider. We thus only need to show $\langle M, w \rangle \in A_{TM} \Leftrightarrow \langle M_1, M_2 \rangle \in \overline{EQ_{TM}}$. As soon we have shown that, we are done.

1. " \Rightarrow ":

In this case, M accepts w . Therefore, M_1 will still always reject and M_2 always accepts. We immediately conclude that the languages $L(M_1)$ and $L(M_2)$ are not the same, thus $\langle M_1, M_2 \rangle \in \overline{EQ_{TM}}$. ✓

2. " \Leftarrow ":

We show the equivalent $\langle M, w \rangle \notin A_{TM} \Rightarrow \langle M_1, M_2 \rangle \notin \overline{EQ_{TM}}$. In this case, M rejects w . This can be due to an inappropriate input, M entering a reject state and M looping. We skip the inappropriate input and consider immediately M entering on input w the reject state. In that case, M_1 will reject all input. Similarly M_2 rejects everything. Hence we have $L(M_1) = L(M_2) = \emptyset$ and we conclude that $\langle M_1, M_2 \rangle \notin \overline{EQ_{TM}}$.

Similarly, if M loops on input w , M_1 rejects every input while M_2 loops on every input. Therefore, $L(M_1) = L(M_2) = \emptyset$ and we conclude that $\langle M_1, M_2 \rangle \notin \overline{EQ_{TM}}$. ✓

□

5 Time complexity

In this final chapter, we again slightly change our view point. While in the previous chapter, we tried to characterize languages or problems with respect to their feasibility to be “solvable” or “computable”, which we expressed by (un-)decidability, we now only focus on decidable problems and characterize these with respect to the running time that is needed to decide these problems. We will introduce *time complexity classes* that distinguish between problems that have different asymptotic running time requirements. As an outcome, we will introduce the important complexity classes P and NP for problems that have a polynomial running time on deterministic and nondeterministic deciders. Withing the problems in NP , we will further identify problems that are specifically challenging. Those problems will be called *NP-complete*. As part of this discussion, we will introduce one of the big open mathematical / computer science problems, which is the question

$$P = NP?$$

In the first section, we discuss how to measure running time and thereby the time complexity of a given Turing machine. As part of this, we briefly review the usual notation for asymptotic running time statements. The two subsequent sections introduce the complexity classes P and NP with examples and proof techniques to show that a given problem is in P or NP . This is followed by a fifth chapter, in which we extend the notion of mapping reductions to the context of running times. Then, in the next section, *NP-complete* problems are discussed. A major part of this section will be invested into the proof of the famous *Cook-Levin theorem*, which states that the satisfiability problem for boolean formulas is *NP-complete*. The chapter is concluded by a series of theorems that expose further structure between time complexity classes.

5.1 Measuring complexity

We start this section by the defining the *time complexity* of a *deterministic* decider, which is given by

Definition 5.1 Let M be a deterministic decider, i.e. a deterministic TM that halts on all inputs. We define the **running time** or **time complexity of M** as the function $f : \mathbb{N} \rightarrow \mathbb{N}$, such that $f(n)$ is the maximum number of *steps* or *transitions* that M uses on any input of length n .

In that case, we say that “ M runs in time $f(n)$ ” or “ M is an $f(n)$ time Turing machine”.

It is expected that the reader is has already some intuition about running time concepts in context of algorithms and data structures. Given that, we make the

Remark (Worst case time complexity) The running time given above corresponds to the *worst case* running time, as we consider the maximum running time over all inputs. We will stick to this worst case analysis of problems / Turing machines. At the same time, we notice that there are also other types of running time analysis, namely *average case* analysis, giving the average running time over all inputs, and the *best case* analysis, giving the shortest possible running time over all inputs. \triangle

As usual for standard algorithms, we use asymptotic bounds for $n \rightarrow \infty$, i.e. “large n ” to describe time complexities. Let us invest into a short reminder for these concepts.

Definition 5.2 Let f, g be functions of form $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$. We say that

$$f(n) = O(g(n))$$

if there exists a real number $c > 0$ and an integer $n_0 > 0$ such that for every integer $n \geq n_0$, we have

$$f(n) \leq c g(n).$$

In this case we say that $g(n)$ is an **(asymptotic) upper bound** for $f(n)$.

Remark ($f(n) = O(g(n))$) As the reader most likely knows, the statement $f(n) = O(g(n))$ is a typical notational convenience in computer science for $f(n) \in O(g(n))$, as $O(g(n))$ is indeed the set of all functions that fulfill the given property. We usually stick to this “mis-nomer” since it often simplifies “calculating” with complexities. Still, we sometimes switch to the function set view, if it seems to be more appropriate. \triangle

We give a few examples for asymptotic upper bounds in

Example 5.1 We compute an asymptotic upper bound for some terms in n :

1. It holds

$$f_1(n) = 5n^3 + 2n^2 + 22n + 6 = O(n^3).$$

To show this, we would have to choose $c = 6$ and $n_0 = 10$ in the above definition and would get that it holds

$$5n^3 + 2n^2 + 22n + 6 \leq 6n^3 \quad \text{for all } n \geq 10.$$

It certainly also holds that $f_1(n) \in O(n^4)$, $f_1(n) \in O(n^5)$, etc.

2. It holds

$$f_2(n) = 3n \log_2 n + 5n \log_2 \log_2 n + 2 = O(n \log n),$$

where we can skip the basis of the logarithm in the upper bound, since the basis only corresponds to a constant factor.

3. If we have a function $f_3(n)$ with

$$f_3(n) = O(n^2) + O(n),$$

then it follows

$$f_3(n) = O(n^2).$$

In that case, we usually write

$$f_3(n) = O(n^2) + O(n) = O(n^2).$$

4. It holds for some constant $c \in \mathbb{R}$

$$f_4(n) = 2^{cn} = 2^{O(n)}.$$

5. Let $t : \mathbb{N} \rightarrow \mathbb{R}_{\geq 1}$ be a monotonically increasing function. It holds

$$O(2^{t(n)}) \subseteq 2^{O(t(n))}.$$

Let $f_5(n)$ be a function such that $f_5(n) \in O(2^{t(n)})$. Hence there exists $c \in \mathbb{R}$ and $n_0 \in \mathbb{N}$ such that for all $n > n_0$ we have $f_5(n) \leq c2^{t(n)}$. We further have

$$f_5(n) \leq c2^{t(n)} = (2^{\log_2 c})2^{t(n)} = 2^{t(n) + \log_2 c}.$$

As we know that $t(n) + \log_2 c = O(t(n))$, we also have constants c', n'_0 (both depending on c), such that for all $n' \geq n'_0 \geq n_0$ it holds $t(n') + \log_2 c \leq c't(n')$ and thus

$$f_5(n') \leq c2^{t(n')} \leq 2^{t(n') + \log_2 c} \leq 2^{c't(n')},$$

which is what we need to show for $f_5(n) \in 2^{O(t(n))}$.

△

There are some typical cases of asymptotic upper bounds, for which we have a name, as given in

Definition 5.3 Let $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ be given.

- For $f(n) = O(n^p)$ ($p \in \mathbb{N}_0$), f has a **polynomial bound**.
- For $f(n) = 2^{O(n^\delta)}$ ($\delta \in \mathbb{R}_{\geq \delta}$), f has an **exponential bound**.

Another well-known asymptotic bound is given by

Definition 5.4 Let f, g be functions of form $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$. We say that

$$\mathbf{f(n)} = \mathbf{o(g(n))}$$

if for any real number $c > 0$ there exists an integer $n_0 > 0$ such that for every integer $n \geq n_0$, we have

$$f(n) < c g(n),$$

which is equivalent to

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

We also give some example for this asymptotic notation.

Example 5.2 It holds

1. $\sqrt{n} = o(n)$,
2. $n = o(n \log \log n)$,
3. $n \log \log n = o(n \log n)$,
4. $n \log n = o(n^2)$,
5. $n^2 = o(n^3)$, and
6. $f(n) \neq o(f(n))$ for all $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$.

△

After this reminder, we can introduce the *time complexity class* for a deterministic single-tape decider by

Definition 5.5 Let $t : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ be a function. The **time complexity class**, $\mathbf{TIME}(t(n))$, is the set of all languages that are decidable by an $O(t(n))$ time single-tape deterministic Turing machine.

This definition maps the knowledge on the running time of a single-tape decider to a class of languages. Hence, we are, e.g., given a language A that is decided by a single-tape TM in $O(n^2)$ steps, i.e. transitions, the above definition implies that $A \in \mathbf{TIME}(t(n))$.

We want to give an example of how to find the time complexity class for a given language.

Example 5.3 We look for the time complexity class, in which the language

$$A = \{0^k 1^k \mid k \geq 0\}$$

is contained. To this end, we give a decider that recognizes this language by

$M_1 =$ “On input string w :

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.”

Now, we perform a running time analysis of the different stages (1.–4.) of the algorithm, where we have $n = |w|$, i.e. n is the size of the input string w .

We go over each stage of the algorithm:

1. We move to the right over n symbols and move back to the beginning, hence do $O(n)$ steps.

2. According to stage 1. the tape content is of form $0^{k_1}1^{k_2}$ with $k_1 + k_2 = n$. The loop is stopped after $\min(k_1, k_2)$ steps. Thereby, we execute stage 3. up to $n/2$ times.
3. Each scan corresponds to a move to the end of the tape and to the front, with $O(n)$ steps.
4. We move once from left to right with $O(n)$ steps.

Putting together the running times for the individual stages, we obtain an upper bound for the running time of

$$O(n) + (n/2)O(n) + O(n) = O(n) + O(n^2) = O(n^2).$$

With this, we obtain the result that

$$A \in \text{TIME}(n^2).^{1)}$$

△

While we introduced the “running time” of a Turing machine over all deterministic deciders, we explicitly pointed to a *single-tape* decider in the definition of the time complexity class $\text{TIME}(t(n))$. The reasoning behind this is discussed in

Remark (Time complexity depends on machine model) In Chapter 3, we have shown that, in terms of the recognized languages, all deciding TMs are equally powerful. If we now consider the complexity of algorithms, we will observe and prove that different decider models will lead to different running time (complexities). Indeed, this will show that statements on decidability are more theoretical results while statements on complexity have a clear impact on real-world applications. △

We give an example for the just gained insight.

Example 5.4 We start with the same language as in the previous example,

$$A = \{0^k 1^k \mid k \geq 0\}.$$

However, this time, we give a *two-tape* deterministic decider for this language by

$M_3 =$ “On input string w :

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Scan across the 0s on tape 1 until the first 1. At the same time, copy the 0s onto tape 2.
3. Scan across the 1s on tape 1 until the end of the input. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0s are crossed off before all the 1s are read, *reject*.
4. If all the 0s have now been crossed off, *accept*. If any 0s remain, *reject*.”

¹⁾One can even show, see [2], that $A \in \text{TIME}(n \log n)$ by considering a more efficient decider.

We quickly collect the asymptotic number of steps / transitions for each stage:

1. We go to the end of the input string and back with $O(n)$ steps.
2. In worst case, the string contains n 0s, thus we need still $O(n)$ steps for the scan operation.
3. With the argument from Example 5.3, the scan stops after a maximum of $n/2$ steps and moves back, thus we need $O(n)$ steps.
4. We move a maximum of $O(n)$ steps to the right.

In each stage, we need $O(n)$ steps, thus a total of $O(n)$ steps. Thereby the running time, thus the maximum number of steps / transitions becomes for this two-tape decider $O(n)$.

△

This example gives a first indicator why we need to “normalize” the time complexity class $\text{TIME}(t(n))$ to a fixed deterministic decider model, namely *single-tape* deciders. Further insight is given by

Theorem 5.1 Let $t(n)$ be a function such that $t(n) \geq n$. For every $t(n)$ time (deterministic) multi-tape decider there exists an equivalent $O(t^2(n))$ time (deterministic) single-tape decider.

We need to give a warning about the interpretation of the above statement in

Remark (Warning) As we will see soon, Theorem 5.1 only makes an existence statement, i.e. for each multi-tape deterministic decider with $t(n)$ time complexity *there exists* a deterministic single-tape decider. It does *not* imply that this is the fastest decider that exists.

Recalling the two previous examples, the theorem seems to match with the observation that we have a running time of $O(n^2)$ for the single-tape decider and a running time of $O(n)$ for the multi-tape decider. However, we also have the footnote at the end of Example 5.3 that states that there exists a $O(n \log n)$ time single-tape decider for the discussed language. This shows that Theorem 5.1 only provides an *upper bound* for the complexity of an equivalent single-tape decider. Still there can be faster single-tape deciders. △

We come to the proof of Theorem 5.1.

Proof

To better follow the proof, the reader should go back to Theorem 3.2, i.e. the statement that there exists for all deterministic multi-tape TMs an equivalent deterministic single-tape TM, and study again the TM construction in the associated proof. We briefly cite the algorithm of the single-tape TM S that simulates the original k -tape TM M in that proof:

Input: $w = w_1 \cdots w_n$:

1. Initialize tape of S :

$$\#w_1w_2\cdots w_n\underbrace{\#\sqcup\#\sqcup\#\cdots\#\sqcup\#}_{k-1 \text{ times}}$$

2. For a single move of M do

- a) Scan over $k + 1$ $\#$ s to obtain symbols under k heads. The symbols are “stored” in the move to a specific state (e.g. a state q_{011} for a three tape TM with heads on symbols 0, 1, 1).
- b) Pass again over tape to apply changes following the transitions of M , which are encoded by new transitions in S .

It only remains to study the number of transitions used in the individual stages of the algorithm, noticing that we certainly use a multi-tape *decider* M and a resulting single-tape *decider* in this proof.

1. The initialization only requires $O(n) + O(k)$ steps, as the input of size n is written and $k - 1$ empty tapes are created. However, we consider k as a constant. This is why we only need $O(n)$ steps.
2. a) One scan needs a number of steps that is in the order of the maximum size of the tape of S during the full execution of S . We observe that, asymptotically, the size of the tape of S is always the sum over the sizes of the tapes in the original multi-tape TM M . However, for the original multi-tape TM M , we know that it has a running time of $t(n)$. Therefore, each of these tapes can only contain up to $t(n)$ symbols. In worst case, we thus obtain

$$O(\text{“tape length of } S\text{”}) = O(kt(n)) = O(t(n)).$$

Hence one scan requires $O(t(n))$ steps.

- b) “applying changes, following transitions of M ” corresponds to another scan with $O(t(n))$ steps. However, we also recall from the earlier proof the issue of the potentially required extension of the “virtual tapes” on the tape of machine S . In worst case, we have to extend each of the tapes by one new symbol, requiring $O(k^2t(n))$ operations. As k is a constant, we however still have $O(t(n))$ operations.

As we know that stage 1. is only executed once, while stages 2. a) and 2. b) are executed $t(n)$ times, we obtain as overall running time of S

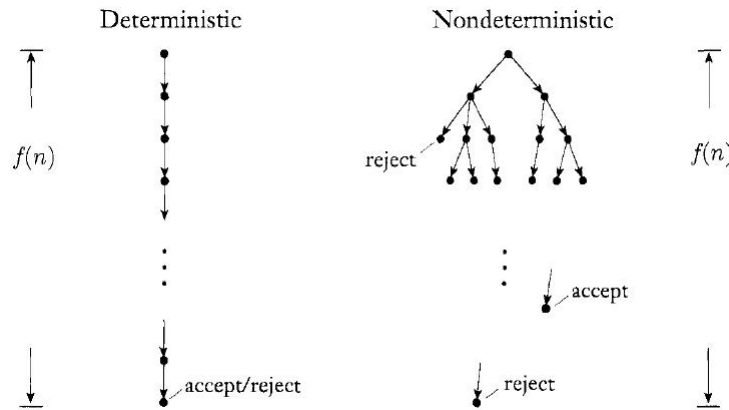
$$O(n) + t(n) \cdot O(t(n)) = O(n) + O(t^2(n)) = O(t^2(n)).$$

□

So far, all statements on running time were made for *deterministic* deciders. Now, we turn to *nondeterministic* deciders.

Definition 5.6 Let N be a nondeterministic decider. The **running time** of N is the function $f : \mathbb{N} \rightarrow \mathbb{N}$, giving the maximum number of steps $f(n)$ that N uses on any branch of its computation on any input of length n .

Hence this definition extends the running time from deterministic to nondeterministic deciders by considering the running time over the longest possible computation path, as we can see in the below comparison between the deterministic and nondeterministic running time.



When comparing single-tape and multi-tape deterministic deciders, we observed that a naive conversion between them resulted in a quadratic increase in the running time (going from multi-tape to single-tape). Similarly, we now consider the move from a nondeterministic single-tape decider to a deterministic single-tape decider.

Theorem 5.2 Let $t(n)$ be a function, such that $t(n) \geq n$. For every $t(n)$ time nondeterministic single-tape decider there exists an equivalent $2^{O(t(n))}$ time deterministic single-tape decider.

Hence, if we are given a nondeterministic single-tape decider and convert it naively to a deterministic single-tape decider we get an *exponential* (!) increase in the running time, which is truly a complexity explosion.

Again, we have to note here that it is still possible that there exists an equivalent deterministic single-tape decider with a much lower running time.

We come to the proof of Theorem 5.2.

Proof

In the proof, we use the construction that we introduced in the proof of Theorem 3.4,

in which we found for a nondeterministic single-tape TM N an equivalent deterministic single-tape TM S . As a starting point, we briefly summarize the ideas of that earlier proof:

- Consider the tree of calculation paths of N , where each node is a configuration.
- Build a deterministic multi-tape TM M that traverses the tree from the root by breath first search to find an accepting configuration. To this end, a computation from the root node (i.e. the starting configuration) to the currently visited node (i.e. configuration) in the search is carried out.
- Convert the deterministic multi-tape TM to a deterministic single-tape TM.

Certainly, we use here a nondeterministic *decider* N and generate deterministic *deciders* M and S . However, the remaining construction is the same.

We discuss the running time of this approach. For the tree of computation branches of N , we observe that the maximum depth of that tree is $t(n)$ as this is the running time of N . Moreover, we assume to have $b \in \mathbb{N}$ as the maximum number of leaves in each node of that tree. It is clear that b is a constant. The reader is further expected to know basic properties of the tree data structure. Following this knowledge, and given b , we observe that the maximum possible number of leaves in the tree of computation branches is $b^{t(n)}$. Even more, the maximum number of nodes in the full tree is $O(b^{t(n)})$. As a consequence of the above considerations, we establish that the multi-tape decider M carries out a breath first search over $O(b^{t(n)})$ nodes. Moreover, in each node, we perform a computation going from the tree root to the node itself with $O(t(n))$ steps. Consequently we obtain an upper bound for the number of steps of M as

$$\begin{aligned} O(t(n)b^{t(n)}) &= O\left(b^{\log_b t(n)} b^{t(n)}\right) = O\left(b^{\log_b t(n) + t(n)}\right) \\ &= O\left(2^{\log_2 b(\log_b t(n) + t(n))}\right) = 2^{O(\log_2 b(\log_b t(n) + t(n)))} \\ &= 2^{O(t(n))}. \end{aligned}$$

After a conversion of the multi-tape decider M to the single-tape decider S , we obtain, according to Theorem 5.1 the upper bound for the number of steps as

$$\left(2^{O(t(n))}\right)^2 = 2^{2O(t(n))} = 2^{O(t(n))}.$$

□

This concludes our introduction into the measurement of running time for deciders.

5.2 The class P

Let us first observe that we can roughly state that problems with a polynomial running time, i.e. a running time with a polynomial bound, can still be solved by modern computers, while problems with an exponential running time usually cannot be solved for larger input size.

Our second observation goes back to the previous section, where we learned that we might have an exponential running time difference between nondeterministic and deterministic Turing machine models. At the same time, switching between two deterministic machine models can be done with a polynomial running time difference, as we have seen it between multi-tape deciders and single-tape deciders. Both statements can be generalized to non-TM machine models, i.e. we would always have an exponential difference between nondeterministic and deterministic models and a polynomial difference within deterministic models.

Taking both above observations into account, it is a meaningful decision to combine all problems of *deterministic* polynomial running time into one class:

Definition 5.7 The complexity class **P** is the set of all languages that are decidable in polynomial time on a deterministic single-tape Turing machine, i.e.

$$\mathbf{P} = \bigcup_{k \in \mathbb{N}_0} \text{TIME}(n^k) .$$

We can thus understand the problems / languages in **P** as those problems, which are still solvable on nowadays computers. Moreover, a given problem in **P** still has a polynomial running time, even if we express it using another deterministic machine model.

In the following, we will discuss and prove for some problems that they are members of **P**. Before we start such proofs, we need to make a technical remark.

Remark (Representation of complex input objects) In Chapter 4, we introduced the notation $\langle O \rangle$ to indicate that we consider some representation, i.e. an encoding, of a complex object O . Back in that chapter, the length of that representation had no impact on theoretical result, as the required number of steps was not relevant.

In contrast to that earlier situation we now have to make sure that the representation that we use can still be converted in polynomial running time to a typical encoding as we would use it on nowadays computers. Otherwise, we might, just by the choice of the encoding, formulate a statement on the complexity of an algorithm that does not reflect the reality of modern computers.

A typical example, where we have to be careful, is the representation of integers. Here, a unary encoding of integers would need exponential running time to be converted to the typical binary encoding of modern computers. Therefore, we should choose deciders that work over binary representations.

An example of a proper representation is the choice of the use of an adjacency matrix to represent a graph. Here, if we consider running time in the number of nodes, the access time to the adjacency matrix is still polynomial in the number of nodes. \triangle

We summarize the typical strategy that we will use for proofs of a language A to be in

P in

Remark (Strategy for proofs of $A \in \mathbf{P}$) In a proof for membership $A \in \mathbf{P}$, we always start by giving stages of an algorithm that describes a deterministic decider for a language A . Different to earlier sections, we will even no longer mention details of a Turing machine (i.e. tape head movement), but will only give a classical algorithmic description, while keeping the behavior of a TM in mind. Still, we have to briefly argue, why the given algorithm decides A .

The proof for the polynomial time bound for the number of steps can usually be done in a less precise way than a typical complexity analysis: We only have to show that we neither execute more than polynomially many stages nor does any of the stages require more than polynomially many steps. It is fine, not to use terminology for TMs to describe the decider, as we know or assume that we can convert any deterministic machine model in polynomial running time to a deterministic decider TM. \triangle

We come to our first example of a problem in **P**.

Theorem 5.3 We define the problem / language *PATH* by

$$PATH = \{\langle G, s, t \rangle \mid G \text{ is a directed graph that has a path from } s \text{ to } t\}.$$

It holds $PATH \in \mathbf{P}$.

Proof

We give a the breath first search algorithm M that decides *PATH* and show that the algorithm has polynomial running time.

Input $\langle G, s, t \rangle$:

1. Mark node s .
2. Repeat until no further nodes are marked:
 - a) Scan all edges E of $G = (V, E)$. If edge $(a, b) \in E$ is found such that a is marked while b is not marked, mark node b .
3. If t is marked, *accept*, else *reject*.

The algorithm carries out a breath first search between s and t and therefore finds a path between s and t , if it exists. Obviously, it also always terminates, hence algorithm M decides *PATH*.

We have a look at the stages. Clearly, 1. and 3. are executed once. Stage 2. a) is repeated up to m times, where m is the length of the longest path (that does not visit nodes twice). As a result, $1 + 1 + m$ stages are executed. Using knowledge on algorithms and data structures, we know that $O(m) = O(|V|)$, thus $O(|V|)$ stages are executed, being clearly polynomial in the size of the input $|\langle G, s, t \rangle|$.

Investigating the number of steps executed in each stage, we identify that stage 1. and 3. obviously require (less than) polynomially many steps. Stage 3. a) requires to go over the representation of the list of edges E and over the list of nodes with the marks. This is also possible in polynomial running time.

Summarizing the number of stages and the steps carried out in the stages, we execute polynomially many stages with a polynomial step count and therefore get an upper bound of a polynomial number of steps. \square

For our next example of a problem in \mathbf{P} , we recall

Definition 5.8 Let x, y be natural numbers. x and y are **relatively prime**, if it holds for their greatest common divisor (gcd)

$$gcd(x, y) = 1.$$

Then, we get

Theorem 5.4 We define the problem / language $RELPRIME$ by

$$RELPRIME = \{\langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime}\}.$$

It holds $RELPRIME \in \mathbf{P}$.

Hence, the problem of figuring out whether two given numbers are relatively prime is in \mathbf{P} . We come to the proof.

Proof

The proof is carried out by using the *Euclidean algorithm* that computes the greatest common denominator. The Euclidean algorithm E is given by

Input $\langle x, y \rangle$: $(x, y \in \mathbb{N} \text{ in binary representation})$

1. Repeat until $y = 0$:
 - a) Set $x = x \bmod y$
 - b) Exchange x and y .
2. Output x .

Then algorithm R that decides $RELPRIME$ is given by

Input $\langle x, y \rangle$: $(x, y \in \mathbb{N} \text{ in binary representation})$

1. Run E on $\langle x, y \rangle$.
2. If output of E is 1, *accept*, else *reject*.

The correctness of the Euclidean algorithm should be known. Then, algorithm R immediately implements the definition for relative prime numbers and both E and R certainly terminate. Therefore, R is a decider for $RELPRIME$.

It remains to discuss the asymptotic running time of R . Obviously, we only need to analyze the running time of E to get the running time of R . We show that stages 1. a) and 1. b) are repeated less than $2 \max(\log_2 x, \log_2 y)$ times. To this end, we observe that latest starting from the second execution of 1. a), each application of this stage at least halves x . Hence, the loop 1. consecutively reduces the magnitude of the larger value by a factor of 2, leading to a required logarithmic number of repetitions in the loop 1. The resulting upper bound of $2 \max(\log_2 x, \log_2 y)$ further accounts for the initial step (in which no reduction by a factor of 2 is achieved).

As the binary logarithm of x or y is exactly the length of their binary representation, we thus obtain $O(n)$ repetitions in the loop 1. In addition, stage 1. a) and 1. b) can certainly be implemented in polynomial time over the input size. Overall, we thus obtain a polynomial running time over the input size. \square

Without proof, we finally observe

Theorem 5.5 It holds $A_{CFG} \in \mathbf{P}$, hence every context-free language is a member of \mathbf{P} .

Note that this language is a good example for the fact that we cannot use all algorithms that we have used for showing decidability, before. Indeed, the algorithm from the proof of Theorem 4.6 would result in an exponential running time. Instead, an algorithm based on dynamic programming is required, see e.g. [2].

5.3 The class NP

In the first section of this chapter, we learned in Theorem 5.2 that there might be a big, i.e. exponential, gap between the running time of a deterministic and a nondeterministic decider, if they are implemented via a single-tape deterministic decider. Note again, that the theorem only provided an upper bound for the running time. In fact, it is unknown, whether there “must” be this gap.

As problems decided by nondeterministic TM may thus be qualitatively different to deterministically decidable problems, we are going to introduce classes of languages that are decided by non-deterministic deciders. More specifically, we will introduce the important class **NP** of all languages that are decided by a non-deterministic decider in polynomial running time. As we will see, there are two characterizations for this class.

We will introduce both and will show that they are equivalent. Then we give some examples for problems that are in **NP**.

Let us briefly recall that the *running time* of a nondeterministic decider is given via the maximum number of steps on the longest branch of computation. With that, we can introduce

Definition 5.9 Let $t : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ be a function. The **nondeterministic time complexity class**, $\mathbf{NTIME}(t(n))$, is the set of all languages that are decidable by an $O(t(n))$ time nondeterministic single-tape Turing machine.

Obviously, this is the immediate analogon of the $\mathbf{TIME}(t(n))$ complexity class, however this time for nondeterministic deciders. Our first definition of the class **NP** simply mimics the complexity class **P** for the nondeterministic case. It is given by

Definition 5.10 (The class NP) The complexity class **NP** is the set of all languages that are decidable in polynomial time by a nondeterministic single-tape Turing machine, i.e.

$$\mathbf{NP} = \bigcup_{k \in \mathbb{N}_0} \mathbf{NTIME}(n^k).$$

We obviously have

Corollary 5.1

$$\mathbf{P} \subset \mathbf{NP}$$

As stated before, there also exists a second definition for **NP**. We have seen that the above definition of **NP** is the obvious extension of the definition of the class **P** to nondeterministic deciders. However, this definition does not expose too much structure about problems in **NP**. Indeed, all problems in **NP** share a common algorithmic idea, which we will now develop. To this end, let us start with

Definition 5.11 (Verifier) A **verifier for a language** A is a deterministic decider V_A that takes as input $\langle w, c \rangle$ (w, c strings) such that it holds:

$$w \in A \text{ iff there exists } c \text{ such that } \langle w, c \rangle \text{ is accepted by } V_A.$$

c is called **certificate**.

The verifier V_A is supposed to decide, whether $w \in \mathcal{A}$. However – and this makes it a verifier, not a decider for A – it can only make this statement if an appropriate additional information, i.e. the certificate c , is given. Let us fill this abstract idea with life by

Example 5.5 We consider the language

$$NOTPRIME = \{\langle n \rangle \mid n \in \mathbb{N} \text{ is not a prime number}\}.$$

We build a verifier $V_{NOTPRIME}$ for the above language. As first input, it gets the string representation $\langle n \rangle$ of a natural number. We further provide as certificate a tuple $\langle p, q \rangle$, where $p, q \in \mathbb{N}_{>1}$. The verifier computes $p \cdot q$ and checks, whether

$$n = p \cdot q.$$

Obviously, it holds $\langle n \rangle \in NOTPRIME$ if and only if there exist $p, q > 1$ such that $n = p \cdot q$. Therefore, the verifier accepts, if $n = p \cdot q$. Otherwise, it rejects. \triangle

Looking at the example, the idea of the verifier does not seem to be very efficient. Indeed, if we were to use the verifier to actually check for a value n whether it is not prime in the above example, we would have to run the verifier for *all* tuples $p, q \in \mathbb{N}$ with $p, q < n$ to get our result. Also, a verifier is deterministic, not nondeterministic. So where is the connection to languages that are recognized by nondeterministic deciders?

The connection goes via the structure of the problems that can be solved by nondeterministic deciders. Indeed, all such problems can be solved by using a verifier, if we in addition use the ability of nondeterminism to replace the inefficient “trying out” by nondeterministically “guessing” the necessary certificate. Thus nondeterministic deciders simply call the verifier and will for sure find the necessary certificate, if it exists.

Before we go into the details of this idea, we briefly limit the running time of the verifier by

Definition 5.12 Let A be a language and V_A be its verifier that takes the input $\langle w, c \rangle$. We call V_A **polynomial time verifier** if it has a polynomial running time in the size of w . Moreover A is called **polynomially verifiable**, if there exists a polynomial time verifier for it.

This gives us all we need to introduce our second definition for the class **NP**.

Definition 5.13 (Alternative definition for NP) **NP** is the class of languages for which there exist polynomial time verifiers.

Let us summarize the idea of this second definition in

Remark The second definition of **NP** is a very constructive one. It implicitly states that all languages in **NP** can be efficiently verified as long as we have a good certificate. That implies that a nondeterministic approach to build a decider via V_A . In this approach, we executed the steps

1. nondeterministically guess c
2. run V_A on $\langle w, c \rangle$
3. accept w if V_A accepts $\langle w, c \rangle$

△

We claim

Theorem 5.6 Definitions 5.10 and 5.13 are equivalent, thus it holds: Let A be a language. There exists a polynomial time verifier for A if and only if there exists a nondeterministic polynomial time decider that recognizes A .

Proof

We need to show the equivalence in the statement.

“ \Rightarrow ”:

Let A be a language and V_A its polynomial time verifier. Let $m(n)$ be the length of the longest certificate c that can be processed by V_A for an input w of size n . We construct a nondeterministic decider for A by

Input w : ($|w| = n$)

1. Nondeterministically select string c , such that $|c| \leq m(n)$.
2. Run V_A on $\langle w, c \rangle$.
3. If V_A accepts, *accept*, otherwise *reject*.

We have to show that this algorithm describes a nondeterminist decider for A and has polynomial running time.

Stage 3. leads to non-deterministic acceptance, if there exists a certificate c such that $\langle w, c \rangle$ is accepted by V_A . Following the definition of a verifier, this is, however, equivalent to the statement that w is contained in A . Moreover, the nondeterministic selection of c is surely a process that terminates, while running V_A terminates since V_A is a decider. Therefore, the proposed algorithm halts and is a decider for A .

When it comes to the running time, we immediately get from the requirements on V_A that stage 2. is executed in polynomial running time. Therefore, the only critical part is the running time of stage 1. There, we produce a certificate that has up to size $m(n)$, which is the length of the longest certificate that can be processed by V_A for an input w of size n . However, as we know that V_A runs in polynomial running time in the size of w , the largest possible certificate also only has a polynomial length. Therefore, we

have $m(n) \in O(n^k)$ for some positive integer k . Consequently, also stage 1. runs in polynomial time and we get the required result. ✓

“ \Leftarrow ”:

Let A be recognized by a nondeterministic polynomial time decider N . We construct the verifier V_A . To this end, we recall the proof of Theorem 3.4, where we build a deterministic TM for a given nondeterministic TM. In that proof, we consider the tree of calculation branches of a nondeterministic TM. In that tree, we have nodes that represent the configurations that the TM takes in the computation paths for a fixed given input w . Furthermore, we assign each of these configurations an encoding by a sequence of integers. This sequence provides a unique identifier for each configuration but also describes the exact steps that a given computation path has to follow to reach a given configuration.

With this brief reminder, we now introduce the verifier V_A by

Input $\langle w, c \rangle$: (w, c strings)

1. Considering c as the encoding of a computation path in N , simulate N on w following path c .
2. If calculation branch c accepts, accept, otherwise reject.

We need to show that V_A is a polynomial time verifier. First we show that it follows the properties of a verifier for A . We know that a string w is contained in A if and only if there exists an accepting computation branch in N . This is however, by construction of V_A , equivalent to the existence of a certificate c that makes V_A accept a given input w . Hence, V_A fulfills the verifier condition. It is also deterministic, since stage 1. just “steps through” the given decider N in a deterministic fashion. Further V_A is a decider, since it halts on all inputs as N halts on all inputs.

Finally, we have to show the polynomial time bound. This is however also fulfilled, since the longest possible computation branch in N is by definition of (nondeterministic) polynomial time complexity polynomial in the size of the input. ✓

□

We now have a look at problems that are in **NP**. We start with the *Hamiltonian path problem*.

Definition 5.14 Let $G = (V, E)$ be a directed graph. A **Hamiltonian path** in G is a directed path that goes through each node exactly once.

We have

Theorem 5.7 For the language

$$HAMPATH = \left\{ \langle G, s, t \rangle \left| \begin{array}{l} G \text{ is a directed graph with a Hamiltonian path} \\ \text{from } s \text{ to } t \end{array} \right. \right\}$$

it holds $HAMPATH \in NP$.

Proof

We carry out the proof via Definition 5.10, i.e. we directly build a nondeterministic polynomial time decider for *HAMPATH*:

Input $\langle G, s, t \rangle$: ($G = (V, E)$ directed graph, $s, t \in V$)

1. Calculate $m = |V|$.
2. Nondeterministically select a sequence of m nodes $p_1, \dots, p_m \in V$.
3. Check for repetitions in p_1, \dots, p_m . If a repetition exists, *reject*.
4. Check whether $s = p_1$ and $t = p_m$, if either fails, *reject*.
5. Check whether $(p_1, p_2), (p_2, p_3), \dots, (p_{m-1}, p_m)$ are edges in G . If one of them fails to be an edge, *reject*.
6. *Accept*.

Let us briefly analyze what the TM does. First it builds an arbitrary (maybe even non-existing) path of size $m = |V|$ in G . The remaining steps simply enforce the different conditions this path has to fulfill in order to exist and to be a Hamiltonian path. Hence, quite obviously, if the path exists, it is found by this TM and therefore it recognizes *HAMPATH*. Moreover, the TM is a decider, since it is clear that all steps can be implemented in a terminating fashion.

Abbreviating the proof a bit, we finally briefly remark that all stages only require up to polynomially many nondeterministic steps, thus the (nondeterministic) polynomial running time bound holds. \square

In the proof of our second example, we will use the verifier construction. Let us first introduce some more graph notions.

Definition 5.15 Let $G = (V_G, E_G)$ be an undirected graph. A **clique** is a subgraph $H \subset G$, $H = (V_H, E_H)$, such that there exist edges between all pairs of nodes V_H . A **k-clique** is a clique with $|V_H| = k$.

We have another problem in **NP** by

Theorem 5.8 For the language

$$CLIQUE = \{\langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique.}\}$$

it holds $CLIQUE \in NP$.

Proof sketch

We give a proof sketch following Definition 5.13. Hence we build a polynomial time verifier V_{CLIQUE} where a clique is used as certificate.

Input $\langle\langle G, k \rangle, C\rangle$: (G directed graph, k clique size, C clique certificate)

1. Test whether C is a subgraph with k nodes in G . If not, *reject*.
2. Test whether C contains edges for all pairs of nodes. If not, *reject*.
3. *Accept*.

It would remain to check, whether V_{CLIQUE} is indeed a polynomial time verifier, which is however quite clear. \square

Just to clarify that we not only have problems in **NP** that are graph problems, we further introduce

Definition 5.16 A *multiset* is a set that additionally allows to contain the same elements more than once.

and state

Theorem 5.9 For the language

$$SUBSETSUM = \left\{ \langle S, t \rangle \left| \begin{array}{l} S = \{x_1, \dots, x_k\} \text{ is a multiset and for some mul-} \\ \text{tiset } \{y_1, \dots, y_l\} \subset S \text{ we have } \sum_{i=1}^l y_i = t \end{array} \right. \right\}$$

it holds $SUBSETSUM \in NP$.

We leave the proof as an exercise for the reader.

Now that we know the classes **P** and **NP**, we can finally introduce the “**P** vs. **NP** problem” by

Remark (*P* vs. *NP* problem) From all that we have learned so far, we know that **P** is the class of languages for which the membership can be *decided* “quickly” i.e. in (deterministic) polynomial running time, while **NP** is the class of languages for which the membership can be *verified* “quickly”.

One of the big open questions of our time is

$$P = NP \quad \text{or} \quad P \neq NP \quad ?$$

Indeed, this is one of the ten *Millennium Problems*, for which which the Clayton institute will grant one million dollar to the person that gives the answer to this problem.

Certainly, following Corollary 5.1, we can already be more precise about the $\mathbf{P} \neq \mathbf{NP}$ part. Indeed, as $\mathbf{P} \subset \mathbf{NP}$, we know that $\mathbf{P} \neq \mathbf{NP}$ corresponds to the statement that \mathbf{P} is a proper subset of \mathbf{NP} , $\mathbf{P} \subsetneq \mathbf{NP}$.

A common hypothesis is that \mathbf{P} is indeed a proper subset of \mathbf{NP} , thus $\mathbf{P} \neq \mathbf{NP}$. However, so far, also no one was able to convincingly²⁾ show that a language in \mathbf{NP} exists that is not in \mathbf{P} . \triangle

In the later sections of this chapter, we will shed some more light on this open question.

5.4 Polynomial time reducibility

In this brief intermediate section, we would like to extend the notion of reducibility from Section 4.3 to the context of time complexities.

Again, we start by considering the implementation of a function by a decider. However, this time, we limit the running time in

Definition 5.17 A function $f : \Sigma^* \rightarrow \Sigma^*$ is a **polynomial time computable function**, if some (deterministic) polynomial time decider M , on every input w , halts with just $f(w)$ on its tape.

This definition now induces a new reduction notion in

Definition 5.18 Let A, B be languages over Σ . A is **polynomial time (mapping) reducible** to B , short $A \leq_P B$, if there is a polynomial time computable function $f : \Sigma^* \rightarrow \Sigma^*$, such that for all $w \in \Sigma^*$, it holds

$$w \in A \quad \text{iff} \quad f(w) \in B.$$

f is called **polynomial time reduction** from A to B .

Hence, the polynomial time reduction transfers elements from a language A to a language B in polynomial time. A first use for this new concept is given by

Theorem 5.10 If $A \leq_P B$ and $B \in P$, then $A \in P$

Proof

Let M be a deterministic polynomial time decider for B and F be the deterministic

²⁾There have been many attempts to show $\mathbf{P} \neq \mathbf{NP}$. However, finally, the scientific community always found too many non-fixable issues in these proofs.

polynomial time decider implementing the polynomial time reduction from A to B . We construct the decider N for A :

Input w :

1. Run F on w giving output $f(w)$.
2. Run M on $f(w)$. If M accepts, *accept*, otherwise *reject*.

N recognizes as TM A since $w \in A$ if and only if $f(w) \in B$ and N accepts if and only if M accepts $f(w)$. Also, N becomes a decider, since F and M are deciders. In stage 1. of the algorithm, F has polynomial running time. As a result, the string $f(w)$ is also at most of polynomial size, since F has polynomial running time. Therefore, in stage 2., M with polynomial running time runs on an input of polynomial size, still giving polynomial running time. Overall, hence, N has polynomial running time. \square

The just shown result is only a very simplistic example of results that we obtain by using polynomial time reductions. In the next section, this new tool will be brought into a more significant use.

5.5 NP-Completeness

In this section, we are going to study a special class of problems in **NP**, namely **NP-complete** problems. Roughly speaking, these problems are “as hard as the whole class **NP**”. More accurately, we will be able to show that if one of these problems is in **P** then it holds $\mathbf{P} = \mathbf{NP}$.³⁾

The main result of this section, besides of introducing the notion of **NP-completeness**, will be the *Cook-Levin theorem*, which states that the satisfiability problem *SAT* of Boolean formulas is **NP-complete**.

Let us start with the main

Definition 5.19 (NP-complete) A language B is called **NP-complete** if it holds

1. $B \in \mathbf{NP}$, and
2. $A \leq_P B$ for all $A \in \mathbf{NP}$.

Hence, in words, it says that a problem is **NP-complete**, if it is in **NP** and we can reduce all other problems in **NP** to it in polynomial time.

³⁾Note that if we have $A \in \mathbf{NP}$ and we prove $A \in \mathbf{P}$ this does *not* immediately imply $\mathbf{P} = \mathbf{NP}$. However, if we have that A is **NP-complete** and we prove that $A \in \mathbf{P}$ then it holds for sure $\mathbf{P} = \mathbf{NP}$.

Theorem 5.11 If B is **NP**-complete and $B \in \mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$.

Proof

This immediately follows from Theorem 5.10. Indeed, since B is **NP**-complete and a member of \mathbf{P} , all problems $A \in \mathbf{NP}$ can be reduced in polynomial time to B and thereby, by Theorem 5.10, all problems in **NP** are in \mathbf{P} . \square

Next, we indicate that we can use polynomial time reductions to “transport” **NP**-completeness from one language in **NP** to another one in **NP** by

Theorem 5.12 If B is **NP**-complete and $B \leq_P C$ for a $C \in \mathbf{NP}$, then C is **NP**-complete.

Proof Obviously, for the **NP**-completeness of C we need to show that $C \in \mathbf{NP}$ and $A \leq_P C$ for all languages $A \in \mathbf{NP}$. The first part, $C \in \mathbf{NP}$, is already part of the conditions of the statement. Therefore, we only need to show the reduction property.

From the requirements of the statement, it holds that $B \leq_P C$, thus there exists a polynomial time reduction from B to C . Since B is **NP**-complete, there exist polynomial time reductions from all problems $A \in \mathbf{NP}$ to B . Moreover, the composition of two polynomial time reductions is a polynomial time reduction. Therefore, it exists a polynomial time reduction from all $A \in \mathbf{NP}$ to C . \square

Now, we are slowly approaching the Cook-Levin theorem. To this end, the reader is reminded of a few definitions from Boolean logic. We start by introducing Boolean formulas in

Definition 5.20 (Syntax of Boolean formulas) An **atomic (Boolean) formula** has the form A_i , where $i = 1, 2, \dots$ **(Boolean) Formulas** are defined by the following inductive process:

1. All atomic formulas are formulas.
2. For every formula F , $\neg F$ is a formula.
3. For all formulas F and G , also $(F \vee G)$ and $(F \wedge G)$ are formulas.

A formula of the form $\neg F$ is called **negation** of F . A formula of the form $(F \vee G)$ is called **disjunction** of F and G , and $(F \wedge G)$ is the **conjunction** of F and G . Any formula F which occurs in another formula G is called a **subformula** of G .

For convenience, we also allow atomic formulas to be called like some letter of the alphabet, e.g. A, B, C, \dots , i.e. not just A_1, A_2, \dots

Example 5.6 (Boolean formulas) Let A, B, C, D be atomic formulas and X_1, \dots, X_N

atomic formulas. Then the following objects are boolean formulas:

- $F = A \vee (B \wedge C)$,
- $F = (\neg A \wedge B) \vee (C \wedge D)$,
- $F = \neg(A \vee B \vee C)$,
- $\bigwedge_{i=1}^N X_i := X_1 \wedge X_2 \wedge \dots \wedge X_N$, and
- $\bigvee_{i=1}^N X_i := X_1 \vee X_2 \vee \dots \vee X_N$.

△

The “meaning” of Boolean formulas is introduced in

Definition 5.21 (Semantics of boolean formulas) The elements of the set $\{0, 1\}$ are called **truth values**. An **assignment** is a function $\mathcal{A} : \mathbf{D} \rightarrow \{0, 1\}$ that assigns an atomic formula a truth value, where \mathbf{D} is any subset of the atomic formulas. Given an assignment \mathcal{A} , we extend it to a function $\mathcal{A}' : \mathbf{E} \rightarrow \{0, 1\}$, where $\mathbf{E} \supseteq \mathbf{D}$ is the set of formulas that can be built up using only the atomic formulas from \mathbf{D} . \mathcal{A}' obeys the following rules.

1. For every atomic formula $A_i \in \mathbf{D}$, $\mathcal{A}'(A_i) = \mathcal{A}(A_i)$.
2. $\mathcal{A}'((F \wedge G)) = \begin{cases} 1 & \text{if } \mathcal{A}'(F) = 1 \text{ and } \mathcal{A}'(G) = 1 \\ 0 & \text{otherwise} \end{cases}$
3. $\mathcal{A}'((F \vee G)) = \begin{cases} 1 & \text{if } \mathcal{A}'(F) = 1 \text{ or } \mathcal{A}'(G) = 1 \\ 0 & \text{otherwise} \end{cases}$
4. $\mathcal{A}'(\neg F) = \begin{cases} 1 & \text{if } \mathcal{A}'(F) = 0 \\ 0 & \text{otherwise} \end{cases}$

For convenience, we will, in the following, write \mathcal{A} instead of \mathcal{A}' , if we refer to \mathcal{A}' .

Example 5.7 We give three examples of formulas, assignments to the atomic formulas and the resulting assignments for the full formulas:

1. $F = A \vee (B \wedge C)$, $\mathcal{A}(A) = 1$, $\mathcal{A}(B) = 0$, $\mathcal{A}(C) = 0 \Rightarrow \mathcal{A}(F) = 1$.
2. $F = \neg(A \vee B \vee C)$, $\mathcal{A}(A) = 0$, $\mathcal{A}(B) = 0$, $\mathcal{A}(C) = 1 \Rightarrow \mathcal{A}(F) = 0$.
3. $F = (\neg A \wedge B) \vee (C \wedge D)$, $\mathcal{A}(A) = 1$, $\mathcal{A}(B) = 0$, $\mathcal{A}(C) = 1$, $\mathcal{A}(D) = 0 \Rightarrow \mathcal{A}(F) = 0$.

△

Finally, we need to understand what is a *satisfiable* Boolean formula.

Definition 5.22 (Satisfiability of Boolean formulas) Let F be a formula and let \mathcal{A} be an assignment. If \mathcal{A} is defined for every atomic formula A_i occurring in F , then \mathcal{A} is called **suitable** for F .

If \mathcal{A} is suitable for F , and if $\mathcal{A}(F) = 1$, then we write $\mathcal{A} \models F$. In this case we say F **holds** under the assignment \mathcal{A} , or \mathcal{A} is a **model** for F .

A formula F is **satisfiable** if F has at least one model, otherwise F is called **unsatisfiable** or **not satisfiable**.

We also give examples for satisfiability in

Example 5.8 (Satisfiability) It holds that

$$F = (A \vee B) \wedge C$$

is satisfiable, as e.g. \mathcal{A} with $\mathcal{A}(A) = 1, \mathcal{A}(B) = 0, \mathcal{A}(C) = 1$ is a model for it. On the other hand

$$F = (\neg A \wedge A) \wedge B$$

is known to be not satisfiable. \triangle

We come to the fundamental statement by Cook and Levin in

Theorem 5.13 (Cook-Levin theorem) The language SAT with

$$SAT = \{\langle F \rangle \mid F \text{ is satisfiable Boolean formula}\}$$

is **NP**-complete.

Let us summarize, how we are going to approach the proof for it in

Remark (Roadmap for proof of Cook-Levin theorem) We will go through the following steps in order to show that SAT is **NP**-complete.

1. We prove that $SAT \in \mathbf{NP}$.
2. We find a reduction from an arbitrary $A \in \mathbf{NP}$ to SAT . Hence, for a given string $w \in A$, we find a Boolean formula F such that

$$w \in A \Leftrightarrow \langle F \rangle \in SAT.$$

Here, we will build a formula F that encodes the behavior of a nondeterministic decider N with $L(N) = A$ on w .

3. We prove that the found reduction has polynomial running time. \triangle

Certainly, the Cook-Levin theorem establishes

Corollary 5.2 $SAT \in P$ iff $P = NP$

Hence, if someone were to find a proof for $SAT \in \mathbf{P}$, we would have $\mathbf{P} = \mathbf{NP}$ and vice versa.

It follows the (tedious) proof of the Cook-Levin theorem. We try to use brief formulations to keep its length at an acceptable level.

Proof (Cook-Levin)

1. Showing $SAT \in \mathbf{NP}$:

We give a polynomial time verifier for SAT :

Input $\langle\langle F \rangle, \langle \mathcal{A} \rangle\rangle$: (F formula, \mathcal{A} assignment)

- a) evaluate assignment $\mathcal{A}(F)$
- b) *accept*, if $\mathcal{A}(F) = 1$, else *reject*.

By construction, it is clear that this TM is a verifier for SAT . Moreover, Stage 1. can be easily carried out in polynomial running time on a deterministic TM. Hence, it is a polynomial time verifier. \checkmark

2. Constructing reduction $A \leq SAT$ for all $A \in \mathbf{NP}$:

We are looking for a (mapping) reduction from an arbitrary $A \in \mathbf{NP}$ to SAT . Hence, our task is to find a computable mapping $f : \Sigma^* \rightarrow \Sigma^*$ such that

$$w \in A \quad \text{iff} \quad f(w) \in SAT$$

Let us assume that we are given an arbitrary, but fixed $A \in NP$. Instead of considering the language A itself, we do the construction for the corresponding nondeterministic decider N with $L(N) = A$. Since $A \in \mathbf{NP}$, we know that any w is in A if and only if N accepts w in polynomial time.

Let us fix an upper bound for the running time by $p(n)$ (with n the length of the input), where $p(n)$ is a polynomial in n . With this bound, we know that $w \in A$ if and only if there exists a sequence of configurations $C_0, \dots, C_{p(n)}$ ⁴⁾ such that

- a) C_0 is starting configuration for input w ,
- b) C_i yields C_{i+1} for $i = 1, \dots, p(n) - 1$, and
- c) $C_{p(n)}$ is halting configuration.

start

move

accept

Given these observations, it follows that we can find a reduction from A to SAT by finding an equivalent conversion of a sequence of configurations with the properties

start

move

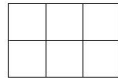
accept

to a Boolean formula F .

⁴⁾Indeed, the last configuration would have some index $k \leq p(n)$ as $p(n)$ is the *upper* bound. However, we assume that we can extend the number of steps such that the last configuration is actually $C_{p(n)}$.

We want to develop a mental picture of such a sequence of configurations. To this end, we introduce the concept of a *tableau for N on w* , which is visualized below.

#	q_0	w_1	w_2	...	w_n	\sqcup	...	\sqcup	#
#									#
#									#
#									#



The tableau is a table with $p(n) + 1$ rows. Each row is one configuration. Given the delimiter symbols on the left and on the right, the tableau has a total of $p(n) + 3$ columns, as $p(n)$ is the maximum number of non-blank symbols on the tape.

In fact, we will be converting this tableau representation to a Boolean formula. To this end, we first observe that the tableau of an accepting calculation fulfills several properties:

- cell : Properties of a given table cell.
- start : Properties of the starting configuration for input w .
- move : Properties of “yielding”.
- accept : Properties of the accepting configuration.

We convert these properties to Boolean formulas

$$F_{cell}, F_{start}, F_{move}, F_{accept}.$$

An accepting computation will then be equivalent to the formula

$$F = F_{cell} \wedge F_{start} \wedge F_{move} \wedge F_{accept}.$$

To construct these formulas, we need a couple of definitions:

- Q : States of N ,
- Γ : Tape alphabet of N .
- $C = Q \cup \Gamma \cup \{\#\}$: Alphabet of symbols in the tableau.
- $cell[i, j] \in C$: Symbol in the tableau at row i and column j .

We start by giving the atomic formulas that we will have in F :

$$X_{i,j,s} \hat{=} \text{atomic formula indicating whether } cell[i, j] = s.$$

Hence, $\mathcal{A}(X_{i,j,s}) = 1$ iff $cell[i, j] = s$. Now, we give the definition of F_{cell} by

$$F_{cell} = \bigwedge_{\substack{0 \leq i \leq p(n) \\ 1 \leq j \leq p(n)+3}} \left[\left(\bigvee_{s \in C} X_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s, t \in C \\ s \neq t}} (\overline{X_{i,j,s}} \vee \overline{X_{i,j,t}}) \right) \right].$$

Let us go over the formula, while noticing that we need to understand what it means that this formula is satisfiable. The outer conjunction goes over all cells, hence the formula is only satisfiable, if the inner conditions are satisfiable for all cells. Then $\bigvee_{s \in C} X_{i,j,s}$ is satisfiable, if one of the $X_{i,j,s}$ is true. Also, $\bigwedge_{\substack{s, t \in C \\ s \neq t}} (\overline{X_{i,j,s}} \vee \overline{X_{i,j,t}})$ is only satisfiable, if in each pair of variables, at least one of them is true. In other words, no more than one variable is turned on. Thereby,

$$\left(\bigvee_{s \in C} X_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s, t \in C \\ s \neq t}} (\overline{X_{i,j,s}} \vee \overline{X_{i,j,t}}) \right)$$

is only satisfiable, if one and only one of the variables $X_{i,j,s}$ for fixed i, j but changing $s \in C$ is assigned the truth value 1. What does that mean?

Indeed, this is the “Boolean formula way” to express that we enforce that $cell[i, j] \in C$. Hence, this is what we need to fulfill property cell. ✓

Next, we define the Boolean formula F_{start} by

$$F_{start} = X_{0,1,\#} \wedge X_{0,2,q_0} \wedge X_{0,3,w_1} \wedge X_{0,4,w_2} \wedge \dots \wedge X_{0,n+2,w_n} \\ \wedge X_{0,n+3,\sqcup} \wedge \dots \wedge X_{0,p(n)+2,\sqcup} \wedge X_{0,p(n)+3,\#}$$

We easily see that this formula becomes true, iff the first row of the tableau is

#	q_0	w_1	...	w_n	\sqcup	...	\sqcup	#
---	-------	-------	-----	-------	----------	-----	----------	---

However, this is exactly the starting configuration for given w . Therefore, property start is fulfilled. Subsequently, we define F_{accept} by

$$F_{accept} = \bigvee_{\substack{0 \leq i \leq p(n) \\ 1 \leq j \leq p(n)+3}} X_{i,j,q_{accept}}.$$

This formula is satisfiable, if and only if somewhere in the tableau, there is an accepting state (reached). Hence, property accept is fulfilled.

The somewhat hardest part is the derivation of the formula F_{move} .⁵⁾ We will try to make its definition as precise as possible, however, will skip over some (communicated) details to keep the proof readable.

As a starting point we define $\gamma_{ij} := cell[i, j]$ and consider a 2×3 window of cells in the tableau:

⁵⁾ Here, we follow [1] and not [2].

$\gamma_{i,j-1}$	$\gamma_{i,j}$	$\gamma_{i,j+1}$
$\gamma_{i+1,j-1}$	$\gamma_{i+1,j}$	$\gamma_{i+1,j+1}$

It is easy to see that $\gamma_{i+1,j}$ can be determined from the top row and, in case one of the top row symbols is a state, from the transition function δ from the nondeterministic decider N . We will use this information in a moment. Let us before that introduce the general formula

$$F_{move} = \bigwedge_{i=0}^{p(n)-1} Y_i$$

where we assume that $\mathcal{A}(Y_i) = 1$ if and only if C_i yields C_{i+1} . Hence, F_{move} is only satisfied, if move is fulfilled. Then, we give further details on the Y_i by defining

$$Y_i = \bigwedge_{1 \leq j \leq p(n)+3} A_{ij} \vee B_{ij}.$$

We introduce the formula B_{ij} such that it is satisfied if the conditions

- a) The symbol for the state in the configuration C_i is sufficiently far away from $\gamma_{i,j}$ such that it cannot influence $\gamma_{i+1,j}$ and
- b) $\gamma_{i+1,j} = \gamma_{i,j}$

are both fulfilled. Note that the first condition certainly implies the second one. However, we still have to encode both of them. How to phrase the above condition as a Boolean formula? We give an “approximative” answer by

$$B_{ij} \approx \left(\bigwedge_{k \in \{j-1, j, j+1\}} \left(\bigvee_{s \in \Gamma} X_{i,k,s} \right) \right) \wedge \left(\bigvee_{s \in \Gamma} (X_{i,j,s} \wedge X_{i+1,j,s}) \right).$$

The first big formula (with respect to the outermost conjunction) is only satisfied, if all symbols in the top row are from the tape alphabet Γ , encoding the first part of the condition. The second big formula is only satisfied, if it holds $\gamma_{i+1,j} = \gamma_{i,j}$, which is the second part of the condition. However, why do we say that this is only approximatively the right formula? Indeed, we skip the “boundary treatment”, i.e. those $\gamma_{i+1,j}$ that are close to the left-hand side or the right-hand side of the tableau.

Recalling that we defined $Y_i = \bigwedge_{1 \leq j \leq p(n)+3} A_{ij} \vee B_{ij}$, we observe that until now we have only encoded by the B_{ij} s the situation that we do not need the information from the transition function of N to compute $\gamma_{i+1,j}$. The second case is encoded by A_{ij} which is satisfied if the conditions

- a) $\gamma_{i,j}$ is a symbol that represents the state of the configuration C_i and
- b) one of the possible transitions being in state $\gamma_{i,j} = q$, reading symbol $\gamma_{i,j+1}$ leads to the appropriate change of symbols from the top row to the bottom row

are both fulfilled.⁶⁾ As the decider can move its tape head to the left and to the right, we need to treat both cases. In the first case, for a transition $(q', \beta, L) \in \delta(q, \alpha)$ the corresponding 2×3 window of the tableau has the form

ϕ	q	α
q'	ϕ	β

Similarly, for a transition $(q', \beta, R) \in \delta(q, \alpha)$ the corresponding 2×3 window of the tableau has the form

ϕ	q	α
ϕ	β	q'

Both cases can finally be translated to the Boolean formula

$$A_{ij} \approx \left(\bigvee_{\phi \in \Gamma} \bigvee_{(q', \beta, L) \in \delta(q, \alpha)} (X_{i,j-1,\phi} \wedge X_{i,j,q} \wedge X_{i,j+1,\alpha} \wedge X_{i+1,j-1,q'} \wedge X_{i+1,j,\phi} \wedge X_{i+1,j+1,\beta}) \right) \\ \wedge \left(\bigvee_{\phi \in \Gamma} \bigvee_{(q', \beta, R) \in \delta(q, \alpha)} (X_{i,j-1,\phi} \wedge X_{i,j,q} \wedge X_{i,j+1,\alpha} \wedge X_{i+1,j-1,\phi'} \wedge X_{i+1,j,\beta} \wedge X_{i+1,j+1,q'}) \right),$$

which is the formula, which is only satisfiable if and only if the two conditions given above are fulfilled. Again, we skip here the boundary cases.

With the introduction of A_{ij} , we have completed the formulas Y_i , which concludes the construction of F_{move} . Since F_{move} has been the final part of our overall formula F , describing the accepting computation branch for w , we are done with building the required reduction. ✓

3. Showing that the introduced reduction has polynomial running time:

We analyze the size of the constructed Boolean formula F . To this end, we first observe that the tableau contains $O(p(n)^2)$ cells. Hence we have a total of $O(|C| \cdot p(n)^2)$ variables. However, since the size of C is a constant, that is independent of the input size of the reduction, we see that this is still $O(p(n)^2)$.

We briefly go through the size of all partial terms of F :

- F_{cell} : Its size is a constant multiple of the number of cells.
 $\Rightarrow O(p(n)^2)$

⁶⁾ At this point, we assume that the decider N , when extending it to $p(n)$ steps, was modified such that subsequent configurations of accepting configurations remain identical.

- F_{start} : Its size is a constant multiple of the number of cells of the first row.
 $\Rightarrow O(p(n))$
- F_{accept} : Its size is a constant multiple of the number of cells. $\Rightarrow O(p(n)^2)$
- F_{cell} : Its size is a large constant multiple of the number of cells. $\Rightarrow O(p(n)^2)$

Note that we used for F_{cell} that the number of possible transitions in N is a constant in the input size, since each transition is encoded in the input.

Combining all results, we observe that the size of F is $O(p(n)^2)$. Then, storing F costs $O(p(n)^2 \log(n))$, since the binary representation of the indices of the variables needs $O(\log(n))$ space. Certainly, $p(n)^2 \log(n)$ can be bound from above by another polynomial $q(n)$, thus $O(p(n)^2 \log(n)) \in O(q(n))$. Thereby, we understand that the storage costs of F are also bound by some polynomial.

In principle, one would now have to go through all the details of the necessary conversion process to the formula F in order to make sure that the given polynomial storage size of F at no point in the reduction process is exceeded by some exponential cost to produce some part of the formula. However, given the discussed reduction steps, none of the steps has such issues. Therefore, we can safely conclude that the reduction can be done in polynomial time. ✓

□

With the above result and Theorem 5.12, we have now a tool to show **NP**-completeness for other languages in **NP**: As soon as we can show for a language A in **NP** that we can reduce SAT in polynomial time to A , we know that A is **NP**-complete. However, if we were to actually carry out such proofs, the language SAT turns out to be not very “handy”. Essentially, we would have to build the reduction for arbitrary Boolean formulas F . As these can be very complex, the “technical implementation” of such proofs can become very involved.

A solution to this problem comes by moving over to Boolean formulas in some *normal form*.

Definition 5.23 (Conjunctive Normal Form) A **literal** is an atomic formula or the negation of an atomic formula. (In the former case, the literal is called **positive**, and **negative** in the latter.)

A formula F is in **conjunctive normal form (CNF)**, if it is a conjunction of disjunctions of literals, i.e. (with $L_{i,j} \in \{A_1, A_2, \dots\} \cup \{\neg A_1, \neg A_2, \dots\}$)

$$F = \left(\bigwedge_{i=1}^n \left(\bigvee_{j=1}^{m_i} L_{i,j} \right) \right)$$

A formula F' is in **3-conjunctive normal form (3CNF)**, if it is a conjunction of

exactly 3 disjunctions of literals, i.e.

$$F' = \left(\bigwedge_{i=1}^n \left(\bigvee_{j=1}^3 L_{i,j} \right) \right)$$

Example 5.9 (Boolean formulas in (3)CNF) We observe that the Boolean formula

$$(X_1 \vee \neg X_2 \vee X_3) \wedge (X_2 \vee X_4) \wedge (X_5 \vee X_1 \vee X_4 \vee \neg X_2)$$

is in conjunctive normal form. Moreover, the formula

$$(X_1 \vee X_4 \vee \neg X_2) \wedge (X_1 \vee X_2 \vee X_3) \wedge (\neg X_4 \vee X_2 \vee X_1) \wedge (X_1 \vee X_5 \vee X_7)$$

is in 3-conjunctive normal form. \triangle

If we are able to show that also the satisfiability problem for formulas in 3CNF is **NP**-complete, then we have a much better tool for doing **NP**-completeness proofs, as reduction algorithms would only have to convert these standardized formulas. Indeed, this is the case, according to

Corollary 5.3 The language 3SAT with

$$3SAT = \{\langle F \rangle \mid F \text{ is in 3CNF and is satisfiable}\}$$

is NP-complete.

We give its (a bit lengthy) proof.

Proof

Since SAT is in **NP**, 3SAT is obviously also in **NP**. Hence, we “only” need to show that all $A \in \mathbf{NP}$ can be reduced to 3SAT. We will do this in a two-step approach by modifying the proof of the Cook-Levin theorem. In the first step, we will be converting, in polynomial time, the formula F that is built in Cook-Levin to an equivalent formula F' in CNF. We recall here, that two formulas F and F' are called equivalent if it holds $\mathcal{A}(F) = \mathcal{A}(F')$ for all possible assignments \mathcal{A} . In the second step, we will be constructing in polynomial time for an arbitrary formula F' in CNF a formula F'' , such that F'' is satisfiable if and only if F' is satisfiable. Both conversion steps imply that we have a polynomial time reduction from any problem $A \in \mathbf{NP}$ to formulas in 3CNF, which concludes the proof.

1. Conversion of F in Cook-Levin theorem to equivalent CNF F'

We start by observing that

$$F_{cell} = \bigwedge_{\substack{0 \leq i \leq p(n) \\ 1 \leq j \leq p(n)+3}} \left[\left(\bigvee_{s \in C} X_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{X_{i,j,s}} \vee \overline{X_{i,j,t}}) \right) \right]$$

is obviously already in CNF. Moreover,

$$F_{start} = X_{0,1,\#} \wedge X_{0,2,q_0} \wedge X_{0,3,w_1} \wedge X_{0,4,w_2} \wedge \dots \wedge X_{0,n+2,w_n} \\ \wedge X_{0,n+3,\sqcup} \wedge \dots \wedge X_{0,p(n)+2,\sqcup} \wedge X_{0,p(n)+3,\#}$$

is also in CNF with clauses of size one. Next,

$$F_{accept} = \bigvee_{\substack{0 \leq i \leq p(n) \\ 1 \leq j \leq p(n)+3}} X_{i,j,q_{accept}}$$

is just one big clause, thus also in CNF. The only non-CNF formula that is present in F is the formula F_{move} , for which we recall that it is given by

$$F_{move} = \bigwedge_{i=0}^{p(n)-1} \left(\bigwedge_{1 \leq j \leq p(n)+3} A_{ij} \vee B_{ij} \right) \text{ with} \\ A_{ij} \approx \left(\bigvee_{\phi \in \Gamma} \bigvee_{(q',\beta,L) \in \delta(q,\alpha)} (X_{i,j-1,\phi} \wedge X_{i,j,q} \wedge X_{i,j+1,\alpha} \wedge X_{i+1,j-1,q'} \wedge X_{i+1,j,\phi} \wedge X_{i+1,j+1,\beta}) \right) \\ \wedge \left(\bigvee_{\phi \in \Gamma} \bigvee_{(q',\beta,R) \in \delta(q,\alpha)} (X_{i,j-1,\phi} \wedge X_{i,j,q} \wedge X_{i,j+1,\alpha} \wedge X_{i+1,j-1,\phi'} \wedge X_{i+1,j,\beta} \wedge X_{i+1,j+1,q'}) \right), \\ B_{ij} \approx \left(\bigwedge_{k \in \{j-1,j,j+1\}} \left(\bigvee_{s \in \Gamma} X_{i,k,s} \right) \right) \wedge \left(\bigvee_{s \in \Gamma} (X_{i,j,s} \wedge X_{i+1,j,s}) \right).$$

Here, we apply the usual distributive laws from Boolean algebra to convert F_{move} to an equivalent CNF formula F'_{move} . We observe that all sub-formulas in F_{move} are of size linear in the input size n . Therefore, in worst case, the conversion to CNF leads to a polynomial increase in the formula size, thus can be carried out in polynomial time.

Finally, we construct

$$F' = F_{cell} \wedge F_{start} \wedge F_{accept} \wedge F'_{move},$$

which is obviously then also in CNF. This last step is also possible in polynomial time. ✓

2. Constructing F'' in 3CNF from arbitrary F' in CNF such that F'' satisfiable iff F' satisfiable:
 Since F' is in CNF, it has the form

$$F' = \bigwedge_{i=1}^S \left(\bigvee_{j=1}^{k_i} L_{ij} \right),$$

where the L_{ij} are literals and we introduce the notation

$$C_i = \bigvee_{j=1}^{k_i} L_{ij}$$

for the clauses. We need to have a formula in CNF, such that $k_i = 3$ for all clauses. Hence, we need to invest some work on formulas for which we have $k_i \neq 3$.

- a) Case: $k_i < 3$:

In this case it holds

$$C_i = L_{i1} \quad \text{or} \quad C_i = l_{i1} \vee l_{i2}.$$

For $k_i = 1$, we build the clause

$$C'_i = L_{i1} \vee L_{i1} \vee L_{i1}$$

and for $k_i = 2$, we build the clause

$$C'_i = L_{i1} \vee L_{i2} \vee L_{i2}.$$

Obviously, the C'_i are satisfiable iff the C_i are satisfiable and the construction is doable in polynomial time.

- b) Case: $k_i > 3$:

For simplicity, we set $k := k_i$. We build

$$C'_i = C_i^{(1)} \wedge C_i^{(2)}$$

such that C'_i is satisfiable iff C_i is satisfiable and $|C_i^{(1)}| = k_1 + 1$, $C_i^{(2)} = (k - k_1) + 1$. Hence the size of C'_i is $|C'_i| = k + 2$. Recalling that we have

$$C_i = \bigvee_{j=1}^k L_{ij},$$

we introduce

$$C_i^{(1)} = \left(\bigvee_{j=1}^{k_1} L_{ij} \right) \vee X \quad \text{and} \quad C_i^{(2)} = \left(\bigvee_{j=k_1+1}^k L_{ij} \right) \vee \neg X,$$

where X is a new atomic formula. Now, we need to show that C_i is satisfiable iff C'_i is satisfiable.

i. “ \Rightarrow ”:

Since C_i is satisfiable, there exists an assignment \mathcal{A} such that $\mathcal{A}(C_i) = 1$. From that, it follows that it holds $\mathcal{A}(L_{ij}) = 1$ for at least one j . We now pick a j' such that $\mathcal{A}(L_{ij'}) = 1$.

Without loss of generality, we assume that $j' \leq k_1$. (Otherwise, we flip the construction.) Then, we have $\mathcal{A}(\bigvee_{i=1}^{k_1} L_{ij'}) = 1$. In principle, we already know by this that $C_i^{(1)}$ is satisfiable. However, we do not have a suitable assignment, as \mathcal{A} does not assign X a value. Hence, we need to extend \mathcal{A} to $\tilde{\mathcal{A}}$ such that it is also suitable for X .

In the construction of $\tilde{\mathcal{A}}$, we choose $\tilde{\mathcal{A}}$ such that $\tilde{\mathcal{A}}(X) = 0$. In this case, we still have $\tilde{\mathcal{A}}(C_i^{(1)}) = 1$, but also get $\tilde{\mathcal{A}}(\neg X) = 1$. The latter observation leads to the fact that indeed also $\tilde{\mathcal{A}}(C_i^{(2)}) = 1$ holds. Thereby, C'_i is satisfiable. ✓

ii. “ \Leftarrow ”:

Since C'_i is satisfiable, there exists an assignment \mathcal{A} such that $\mathcal{A}(C'_i) = 1$. This is why we have that $\mathcal{A}(C_i^{(1)}) = 1$ and $\mathcal{A}(C_i^{(2)}) = 1$. If we know, how the assignment \mathcal{A} operates on X , we can make a statement of on the satisfiability of C_i . Since there are only two possible results for $\mathcal{A}(X)$, we just go over both of them.

If $\mathcal{A}(X) = 1$ then we have

$$1 = \mathcal{A}(C_i^{(2)}) = \mathcal{A}\left(\neg X \vee \bigvee_{j=k_1+1}^k L_{ij}\right) = \mathcal{A}(\bigvee_{j=k_1+1}^k L_{ij}),$$

thus the first part of C_i is fulfilled and we have $\mathcal{A}(C_i) = 1$ and C_i is satisfiable. If $\mathcal{A}(X) = 0$ then we have

$$1 = \mathcal{A}(C_i^{(1)}) = \mathcal{A}\left(X \vee \bigvee_{j=1}^{k_1} L_{ij}\right) = \mathcal{A}(\bigvee_{j=1}^{k_1} L_{ij}),$$

thus the second part of C_i is fulfilled and we have $\mathcal{A}(C_i) = 1$ and C_i is in all cases satisfiable. ✓

We conclude that C_i is satisfiable iff C'_i is satisfiable. Recalling that we have

$$C'_i = C_i^{(1)} \wedge C_i^{(2)}$$

constructed for the situation that $|C_i| = k > 3$, we still need to argue, why this decomposition helps us to build clauses of size exactly three.

The argument for this is an inductive one. Given a formula C_i we choose in the above splitting $k_1 = 2$, then the first clause $C_i^{(1)}$ will always be of size three, while the second clause will be of size $k - k_1 + 1 = k - 2 + 1 = k - 1$. Hence, we repeatedly decrease the size of the original clause C_i by one, while

generating new clauses of size three. We do this, until the remaining clause $C_i^{(2)}$ is also of size three. Then we are done. This process can be carried out in polynomial time and produces a sequence of clauses that can be joined in a 3CNF being satisfiable iff C_i was satisfiable.

Overall, we have now shown that we can convert each clause in F' either to one clause of size three or to a CNF with clauses of size three, where the output of the conversion is satisfiable iff the original clause is satisfiable. Combining all converted clauses in a conjunction to F'' gives thus a 3CNF that is satisfiable iff F' is satisfiable. ✓

As we have converted $A \in \mathbf{NP}$ to a CNF and then to a 3CNF with the required properties, we have found

$$A \leq_p 3SAT \quad \text{for all } A \in \mathbf{NP}.$$

□

If we look carefully at the above proof, we have just also shown

Corollary 5.4 For all boolean formulas F in CNF there exists a boolean formula F' in 3CNF such that F is satisfiable if and only if F' is satisfiable. F' can be built from F in polynomial time.

With Corollary 5.3, we indeed now have the main tool for showing **NP**-completeness of other languages in **NP**. We close this section by giving an example of such a statement and proof.

Theorem 5.14 The problem *CLIQUE* is NP-complete.

Proof:

We carry out the proof using Theorem 5.12. From Theorem 5.8, we already know that *CLIQUE* $\in \mathbf{NP}$. Therefore, it remains to show that

$$3SAT \leq_p CLIQUE$$

in order to prove **NP**-completeness of *CLIQUE*.

As a reminder, *CLIQUE* is given by

$$CLIQUE = \{ \langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique} \}.$$

Let an arbitrary but fixed formula F in 3CNF be given. We have

$$F = \bigwedge_{i=1}^k (L_{i1} \vee L_{i2} \vee L_{i3}).$$

From F with k clauses, we construct the the string $\langle G, k \rangle$, where we still need to specify the graph $G = (V, E)$. The nodes V in our construction are given by

$$V = \{L_{ij} | i = 1, \dots, k, j = 1, \dots, 3\}.$$

Hence, we identify each literal in the formula F with an node in G . Then, the edges are given by

$$E = \{(L_{ij}, L_{i'j'}) | j, j' = 1, \dots, 3, i \neq i', L_{ij} \neq \neg L_{i'j'}\}.$$

The condition $i \neq i'$ implies that no edge is present between nodes that are associated to literals that are in the same clause. Moreover, $L_{ij} \neq \neg L_{i'j'}$ indicates that no edge is present between contradictory literals like X_2 and $\neg X_2$.

Constructing this graph from the given formula is obviously doable in polynomial time. We finally need to show that F is satisfiable if and only if G has a k -clique. If this is true, we have found the necessary polynomial time reduction from $3SAT$ to $CLIQUE$ and we are done. We prove the equivalence:

1. " \Rightarrow ":

Since F is satisfiable, there exists an assignment \mathcal{A} such that $\mathcal{A}(F) = 1$. Hence all clauses of F are satisfiable and therefore $\mathcal{A}(L_{i1} \vee L_{i2} \vee L_{i3}) = 1$ for all $i = 1, \dots, k$. Within each clause, we therefore have at least one of L_{i1} , L_{i2} and L_{i3} bein true under the assignment \mathcal{A} . Without loss of generality, we choose for each clause C_i an index $j(i) \in \{1, 2, 3\}$ with $\mathcal{A}(L_{ij(i)}) = 1$.

Given this notation, the subgraph $H = (V_H, E_H)$ with $V_H = \{L_{ij(i)}\}_{i=1}^k$ and $E_H = E \cap (V_H \times V_H)$ forms a k -clique in G . This is true, since $|V_H| = k$ and all notes in H are fully connected, since $i \neq i'$, $L_{ij} \neq \neg L_{i'j'}$ for all $(L_{ij(i)}, L_{ij(i')}) \in E_H$. ✓

2. " \Leftarrow ":

Since G has a k -clique, then there exists a subgraph $H = (V_H, E_H)$ such that $V_H \subset V$, $|V_H| = k$, $E_H \subset E \cap V_H \times V_H$ and $E_H = V_H \times V_H$. We assume that we can enumerate all nodes in V_H such that $V = \{v_1, \dots, v_k\}$.

Analyzing H , we observe that it holds $v_i = L_{ij(i)}$ for some mapping $j(i) \in \{1, 2, 3\}$ and $i = 1, \dots, k$, since $E_H = V_H \times V_H$ implies that the v_i s have to correspond to literals in different clauses.

Now, we construct a suitable assignment \mathcal{A} for F with

$$\mathcal{A}(v_i) = \mathcal{A}(L_{ij(i)}) = 1 \quad \text{for } i = 1, \dots, k.$$

This is possible, since $E_H = V_H \times V_H$ also implies that all $v_i = L_{ij(i)}$ are non-contradictory. Since $\mathcal{A}(v_i) = \mathcal{A}(L_{ij(i)}) = 1$ for $i = 1, \dots, k$, it also holds $\mathcal{A}(L_{i1} \vee L_{i2} \vee L_{i3}) = 1$ for $i = 1, \dots, k$, thus $\mathcal{A}(F) = 1$. ✓

□

This concludes our discussion of **NP**-completeness.

5.6 Hierarchy theorems

Until now, we mainly considered the time complexity of languages / problems on the level of **P**, **NP** and **NP-complete** languages. Indeed, it is unknown, whether these problems are actually different in their “difficulty”. In this section, we will be considering statements that allow us to differentiate between time complexity classes for deterministic deciders, implying that giving more computing time to a deterministic decider actually leads to a larger class of problems that can be solved.

If we start entering a very precise discussion of running times, we need to first make the following

Remark (Time complexity function) In the Definitions 5.1 of the running time of a deterministic decider, we considered f to be a general function $f : \mathbb{N} \rightarrow \mathbb{N}$. Certainly, our intuition is that we have for f functions like n^2 , n^3 , $n \log n$, etc. with integer rounding. However, as we did not really specify $f : \mathbb{N} \rightarrow \mathbb{N}$ further, f does not have to be “nice”. Hence, it could even happen that the evaluation of that function by a decider takes longer than the value $f(n)$ that the function gives as running time limitation of that decider. \triangle

To rule out the “non-nice” functions to describe running time, we give

Definition 5.24 Let f be a function of form $f : \mathbb{N} \rightarrow \mathbb{N}$, where $f(n)$ is at least $O(n \log n)$. We call f **time constructible** if a computable function exists that maps the string 1^n (i.e. the unary representation of n) to the binary representation of $f(n)$ in time $O(f(n))$.

Note that the unary input is used here to imply that $f(n)$ is indeed a function in the input *size* of the TM. Moreover, while the unary input is mandatory, the output can be unary or binary. Further, it is easily possible to show that our usual functions like $n \log n$, $n\sqrt{n}$, n^2 , 2^n , etc. are time constructible.

We give the main result of this section by

Theorem 5.15 (Hierarchy theorem) Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be an arbitrary time-constructible function. There exists a language A that is decidable in $O(f(n))$ time by a deterministic decider, but not deterministically decidable in time $o(f(n)/\log f(n))$.

Already the statement of the theorem is somewhat a challenge to understand. A less formal version of it would be that by increasing the time complexity by *more* than a logarithm in the input size, we increase the complexity class. Hence, we in some sense can gain the intuition that by looking at the class $TIME(t(n))$ for larger and larger functions $t(n)$, the number of members actually increases. However, only an increase of more than $\log t(n)$ leads to having more languages in that class.

The proof is highly non-trivial.

Proof:

The proof is done by construction. We build a deterministic decider D with $L(D) = A$, such that

1. D decides A in $O(f(n))$ time, however,
2. A cannot be decided in $o(f(n)/\log f(n))$ time.

We first give the decider D :

Input: w

1. $n = |w|$.
 2. Compute $f(n)$ and store $\lceil f(n)/\log f(n) \rceil$ in binary counter. The counter is decremented before every simulation step of stage 4. If the counter hits 0, *reject*.
 3. If w does not have the form $\langle M \rangle 10^*$, where M is some TM, *reject*.
 4. Simulate M on w .
 5. If M accepts, *reject*. If M rejects, *accept*.
1. D has $O(f(n))$ running time:

Stage 1. requires a linear number of steps in the input size. Since f is time constructible, computing and storing $\lceil f(n)/\log f(n) \rceil$ is possible in $O(f(n)/\log f(n))$ giving the running time of stage 2. Also, checking the format of the input w , carried out in stage 3., should be doable in time $O(f(n))$.

The more challenging part is the running time of stage 4. To be able to discuss its running time, we need to give an implementation of what it means to “simulate” M by D . We just give a sketch, here:

We build a *single*-tape decider with 3 “tracks”. On the first track, we store the information of the tape of M . On the second track, we store the current state of M and M ’s transition function. On the third track, we store the step counter from stage 2. of D . The different tracks are joined on a single tape by either “interleaving” the content of the three different tracks or by using a tape alphabet with triple symbols.

We claim that it is possible to keep the content of the second track and the third track “close” to the tape head position on the first track, since moving the content of these tracks is “cheap”. This comes from the fact that

- a) the length of the second track is constant⁷⁾ in the size of M and thus has no influence on the asymptotic running time, and
- b) the length and calculation time of the counter on the third track is $O(\log(f(n)/\log f(n))) \in O(\log f(n))$.

⁷⁾In fact, it is not exactly constant, since the tape alphabet of M can be different to the one of D . However, it is possible to balance this with the calculation time of 10^* .

The b) part is thereby adding a time overhead of $O(\log f(n))$.

Following the above statements, we give a running time analysis for stage 4. First of all, due to the counter, the running time of the simulation is limited to $f(n)/\log f(n)$. Since keeping the content of the second track close to the tape head on the first track is done (almost) in constant time, we keep for that movement the overall running time of $O(f(n)/\log f(n))$. Only the movement of the content of the third track with the position on the tape head on the first track adds an overhead of $O(f(n))$, thus the overall simulation time becomes $O(f(n))$.

Since, certainly, stage 5. can also be executed in $O(f(n))$, we obtain that A is decidable by D in $O(f(n))$ time. ✓

2. $L(D) = A$ is not decidable in $o(f(n)/\log f(n))$: We prove the undecidability in the given time budget using a diagonalization-like argument in a proof by contradiction. To this end, we assume that there exists a decider D' that can decide A in $g(n) \in o(f(n)/\log(f(n)))$ time. We will show that D and thereby A is constructed in a way that it will always act in a different way than any decider that fulfills the time budget, therefore D' cannot decide A .

We use the assumed to be existing decider D' and a string $s \in L(10^*)$ jointly as input $\langle D' \rangle s$ to D . Hence, in D we set $M = D'$. In D , we thus then simulate D' on the input $\langle D' \rangle s$ in stage 4. According to our assumption, this simulation is carried out in $g(n)$ steps, where we know from $g(n) \in o(f(n)/\log f(n))$ that it holds

$$g(n) < \frac{f(n)}{\log f(n)} \quad \text{for all } n \geq n_0,$$

with some $n_0 \in \mathbb{N}$. We use this information to understand the behavior of D .

- a) Case $|\langle D' \rangle| \geq n_0$:

In this case, the asymptotic bound holds, thus $g(n) < f(n)/\log f(n)$. Therefore, the simulation terminates and D rejects if D' accepts. Consequently, D will always recognize a different language than D' , thus $L(D') \neq L(D)$. ✗

- b) Case $|\langle D' \rangle| < n_0$:

At least for $s = 10^{n_0}$ the input $\langle D' \rangle 10^{n_0}$ is of size equal or larger than n_0 . In this case, again, D rejects, if D' accepts. Consequently, at least for the inputs with suffix 10^{n_0} , D will behave differently to D' , hence $L(D') \neq L(D)$. ✗

□

As an immediate consequence of the hierarchy theorem, we obtain

Corollary 5.5 Let $f_1, f_2 : \mathbb{N} \rightarrow \mathbb{N}$ be time-constructible and $f_1(n) = o(f_2(n)/\log f_2(n))$, then

$$TIME(f_1(n)) \subsetneq TIME(f_2(n)).$$

Since it holds for any two real numbers $1 \leq \varepsilon_1 < \varepsilon_2$ that $n^{\varepsilon_1} = o(n^{\varepsilon_2} / \log n^{\varepsilon_2})$, we also get

Corollary 5.6 For $\varepsilon_1, \varepsilon_2 \in \mathbb{R}$ with $1 \leq \varepsilon_1 < \varepsilon_2$ and n^{ε_1} is at least in $O(n \log n)$, it holds

$$TIME(n^{\varepsilon_1}) \subsetneq TIME(n^{\varepsilon_2}).$$

This shows clearly that by increasing the exponent p in a running time limit of the form n^p , we get deterministic complexity classes with more and more languages.

We conclude this section by making a statement about the relationship between **P** and a complexity class that we did not use before, namely

Definition 5.25 The complexity class **EXPTIME** is the set of all languages that are decidable “in exponential time” on a deterministic single-tape Turing machine, i.e.

$$\mathbf{EXPTIME} = \bigcup_{k \in \mathbb{N}_0} TIME(2^{n^k}).$$

An immediate consequence from Corollary 5.6 is now the major result

Corollary 5.7

$$\mathbf{P} \subsetneq \mathbf{EXPTIME}.$$

Overall, we learned in this section, that we have a clear and straight-forward hierarchy of time complexity classes for deterministic deciders. However, the more pressing question on the relationship between the deterministic polynomial time complexity classes and the nondeterministic polynomial time complexity classes remains open.

Bibliography

- [1] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Pearson Addison-Wesley, Boston, MA, 3 edition, 2007.
- [2] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013.