An Introduction to the Theory of Computing

Dennis Komm

Lecture Notes
Foundations of Computing II
Autumn Term 2020, University of Zurich

Version: November 18, 2021
Contact: dennis.komm@inf.ethz.ch

Computers allow us to automate many tasks of everyday life. There is, however, a strict limit as to what kind of work can be automated. The aim of this lecture is to study these limits. In 1936, Alan Turing showed that there are well-defined problems that cannot be solved by computers (more specifically, by algorithms); this does not address a lack of computational power in the sense of stating that today's computers are too slow, but as computers get faster, we may hope to solve these problems eventually. The inability to solve these problems will prevail, no matter how the performance of computers will improve. It is noteworthy that the modern computers were not present in Turing's time. Following his footsteps, we ask "Which problems can be solved automatically and which cannot?" In order to do this, we introduce a formalization of the term "algorithm." We start with a restricted class, the finite automata, which we then extend by making the underlying model more and more powerful, eventually arriving at the Turing machine model.

We then prove the existence of problems that cannot be solved by Turing machines. The proof uses arguments similar to proving that there are more real numbers than natural numbers and to Gödel's incompleteness theorem. One of the most famous such problems is the halting problem, e.g., to decide for a given Turing machine whether it finishes its work on a given input within finite time or runs forever. Rice's theorem even states that most questions about the semantics of Turing machines cannot be decided in general. As a consequence of the Church-Turing thesis, there are no algorithms for semantic problems about computer programs as well.

After that, we will solely focus on decidable problems, i.e., on problems for which we can design Turing machines that solve them. We then ask with how much effort this can be done; in particular, we are interested in the time that needs to be spent. We will classify problems into tractable and intractable problems, thereby discussing the $\mathcal{P}$ vs. $\mathcal{NP}$ problem. Proving Cook's theorem, we show the existence of an $\mathcal{NP}$-complete problem; and using reductions, we prove that there is a large number of such problems, for which neither nontrivial lower bounds nor efficient algorithms are known.

Contents

1 Regular Languages

In order to study problems and algorithms designed to solve them, we first need to define the terms "problem" and "algorithm" formally; in this chapter, we start by looking at particular subclasses of both. While the kind of problems we define is a lot more general than it seems to be on first sight, the algorithms considered in the following are indeed very limited. However, they make up an important building block for the more general models we will consider later.

1.1 Formal Languages

First, we need to define the notion of an **alphabet**, which is simply any finite nonempty set. Alphabets are usually denoted by the Greek letter Σ; let us give a few examples.

- $\Sigma_{\text{bin}} = \{0, 1\}$ is the binary alphabet,
- $\Sigma_{\text{dec}} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is the alphabet of decimal digits,
- $\Sigma_{\text{eng}} = \{a, b, \ldots, z\}$ is the lower-case English alphabet,
- $\Sigma_{\text{greek}} = \{\alpha, \beta, \ldots, \omega\}$ is the lower-case Greek alphabet, and
- $\Sigma_{\text{sym}} = \{\square, \triangle, \bigcirc\}$ is some alphabet of three symbols.

The elements of an alphabet are called **letters**, **characters**, or simply **symbols**. Next, we need the notion of a **word**, which we also call a **string**; words are defined with respect to a given alphabet. A word is any sequence that can be obtained by concatenating any finite number of letters of this alphabet. We say that, e.g., 011 is a word "over" the alphabet Σ_{bin}. A special word is the empty word, which we denote by ε, and which consists of no letter at all. The length of a word w, denoted by $|w|$, is the number of positions in w, e.g., $|011| = 3$. The empty word has length zero, thus $|\varepsilon| = 0$, and it is the only word of this length.

A **language** "over" an alphabet Σ is any (finite or infinite) set that contains words over Σ only. Since languages are sets, we have the usual operations $L_1 \cup L_2$, $L_1 \cap L_2$, and $L_1 \setminus L_2$ defined on any two languages L_1 and L_2. Furthermore, we have the relations $L_1 \subsetneq L_2$, $L_1 \subseteq L_2$, $L_1 = L_2$, and $L_1 \neq L_2$.

Furthermore, for a given alphabet Σ, we define the following languages.

- For any $k \in \mathbb{N}$, Σ^k is the language that contains all words over Σ that have length exactly k; here and anywhere subsequently, we define $\mathbb{N}$ to contain 0.
- $\Sigma^1 = \Sigma$ is the language that contains all words that each have exactly one letter; strictly formally speaking, Σ^1 and Σ are two different things since the former contains words and the latter contains letters, but this can be ignored.
- $\Sigma^0 = \{\varepsilon\}$ is the language that only contains the empty word (observe that $\{\varepsilon\} \neq \emptyset$).
- $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$ is the **Kleene closure** (or simply the **Kleene star**) of Σ and contains all words over Σ, i.e., all words that are obtained by concatenating letters from Σ in any way.

- $\Sigma^+ = \Sigma^* \setminus \Sigma^0 = \Sigma^1 \cup \Sigma^2 \cup \dots$ contains all nonempty words over Σ.

We observe that any language L over an alphabet Σ is a subset of Σ^*, i.e., $L \subseteq \Sigma^*$; by $L^{\complement} = \Sigma^* \setminus L$ we denote the **complement of L**. It is important to keep in mind that alphabets are always finite, while languages may have an infinite size; in particular, Σ^* and Σ^+ are infinite for every Σ (recall that alphabets are never empty).

By concatenating languages, we can obtain new languages as follows.

- For two languages L_1 and L_2 with $L_1, L_2 \subseteq \Sigma^*$, we define the **concatenation of L_1 and L_2** as $L_1 \circ L_2 = L_1 L_2 = \{vw \mid v \in L_1 \text{ and } w \in L_2\}$.
- The **kth power of a language L** is defined inductively by $L^0 = \{\varepsilon\}$, $L^1 = L$, and, for any $k \in \mathbb{N}$, $L^{k+1} = L^k L$. In other words, L^{k+1} contains all words that are obtained by concatenating any $k+1$ words from L. It is important to observe that $\varepsilon w = w = w\varepsilon$ for every word w. Also note that, for every $k \geq 1$,

  $$\varepsilon \in L^k \iff \varepsilon \in L \,.$$

- The **Kleene closure of a language L** is defined by $L^* = L^0 \cup L^1 \cup L^2 \cup \dots$.
- Similarly to alphabets, we define $L^+ = L^1 \cup L^2 \cup \dots$.

Now we have all we need to define the special class of problems we are interested in, and which are called **decision problems**. In essence, a decision problem is to determine whether a given word is contained in a given language.

Definition 1.1 (Decision Problem). *A decision problem is given by a language L over some alphabet Σ. A valid input is any word $w \in \Sigma^*$, and the task is to "decide" whether $w \in L$ or $w \notin L$.*

Although Definition 1.1 may seem rather artificial and restricted, it is actually quite general, as we can model many intuitive problems as decision problems. Consider, e.g., the task to check whether a given number is prime. We can simply define

$$L_{\text{PRIME}} = \{w \in \Sigma_{\text{dec}}^* \mid w \text{ is the decimal representation of a prime number}\} \,,$$

and checking whether a given number x is prime is then nothing else but interpreting x as a word over Σ_{dec} and deciding whether

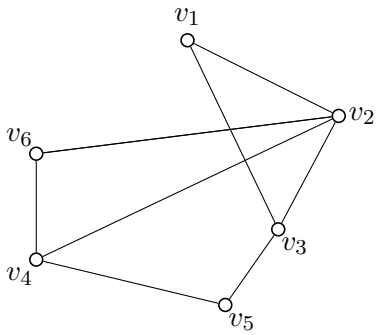
$$x \in L_{\text{PRIME}} \quad \text{or} \quad x \notin L_{\text{PRIME}} \,.$$

Alternatively, we can also define

$$L_{\text{PRIME}} = \{w \in \Sigma_{\text{bin}}^* \mid w \text{ is the binary representation of a prime number}\} \,,$$

which simply means that we use a different encoding of the inputs.

As another example, consider the famous **Hamiltonian cycle problem**, HC for short. Here, the task is to find a cycle in a given graph that visits every vertex exactly once.



(a)

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

(b)

Figure 1.

We know that we can encode a graph using its **adjacency matrix**. Consider, e.g., the graph in Figure 1a; its adjacency matrix (Figure 1b) can be encoded over the alphabet $\Sigma_{\text{graph}} = \{0, 1, \#\}$ by the word

$$x = 011000\#101101\#110010\#010011\#001100\#010100 \ ,$$

and checking whether it contains a Hamiltonian cycle can be formalized by deciding whether x is contained in the language

$$L_{\text{HC}} = \{w \in \Sigma_{\text{graph}}^* \mid w \text{ encodes a graph with a Hamiltonian cycle}\} \ .$$

Now we want to study how such problems can be solved by means of computers. While the above examples show that the notion of decision problems allows us to model rather complex tasks, we start with simpler problems, i.e., languages that have a rather simple structure. For such languages, the corresponding decision problems can be solved by comparatively simple devices. Later on, however, we will allow more and more complicated computations that will eventually lead to the notion of Turing machines, which are able to solve the above-mentioned decision problems L_{PRIME} and L_{HC} and many others.

1.2 Deterministic Finite Automata

Our first idea to solve simple decision problems is to use a straightforward pattern recognition mechanism. We read a given word letter by letter; depending on which letter we read, we enter a **state** that saves information about the prefix of the word read so far. After the whole word is processed that way, we end in some state, which can either be **accepting** or **nonaccepting**; in the former case, the word is contained in the language, otherwise it is not. This approach is formalized by so-called **deterministic finite automata**, DFAs for short. Sometimes, we will speak of a DFA "reading a word," and subsequently accepting or rejecting the word.

Definition 1.2 (Deterministic Finite Automaton, DFA). *A DFA A is a quintuple $A = (Q, \Sigma, \delta, q_0, F)$, where*

- *Q is a finite set of* ***states****,*
- *Σ is an alphabet, called the* ***input alphabet****,*
- *$\delta\colon Q \times \Sigma \to Q$ is the* ***transition function****,*
- *$q_0 \in Q$ is the* ***start state****, and*
- *F is the set of* ***accepting states****.*

Definition 1.2 does not exactly seem intuitive. In order to get a better feeling for a given DFA, we usually choose another representation, which is given by a diagram. To this end, we use the following conventions.

- The states are represented by circles,
- a transition $\delta(q, l) = q'$ with $q, q' \in Q$ and $l \in \Sigma$ is represented by an arrow from q to q' labeled by l,
- the start state is marked by an incoming arrow, and
- all accepting states are marked by a double circle.

An example is shown in Figure 2. It is important to keep in mind that a DFA has exactly one start state, but an arbitrary number of accepting states. Moreover, a DFA is always complete in the sense that every state has an outgoing transition for every letter from the input alphabet.

DFAs can be used to accept simple languages. Consider, e.g., the language

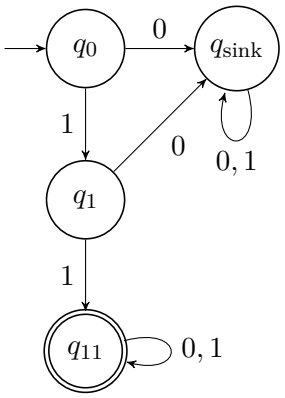
$$L_{11} = \{w \in \{0,1\}^* \mid w \text{ begins with } 11\}\ ,$$

which contains all binary words that begin with 11. A DFA A_{11} that accepts this language reads a given word x letter by letter. If the first letter is 0, it enters a state q_{sink}, which is nonaccepting, since in this case x cannot be in L_{11}; this state cannot be left, no matter which letter appears afterwards. Conversely, if x begins with the letter 1, another state q_1 is entered; intuitively, this state means "the first letter of x is 1," whereas q_{sink} means "x is not in L_{11}." If the second letter is 0, again x cannot be in L_{11}, which is why A_{11} enters the state q_{sink} also in this case. If, however, the second letter is 1, we have $x \in L_{11}$, and thus A_{11} now enters an accepting state q_{11}, which again cannot be left. The diagram representation of A_{11} is shown in Figure 2b.

Another way to represent a DFA is by means of a **transition table**, which is nothing else than a shorthand to define δ. The transition table of A_{11} is depicted in Figure 2c. For obvious reasons, we will use the diagram representation when designing DFAs for given languages. However, we will later need the formal representations in order to argue about the limitations of DFAs.

- $Q = \{q_0, q_1, q_{11}, q_{\text{sink}}\}$,
- $\Sigma = \{0, 1\}$,
- q_0,
- $\delta(q_0, 0) = q_{\text{sink}}, \delta(q_0, 1) = q_1$,
  $\delta(q_1, 0) = q_{\text{sink}}, \delta(q_1, 1) = q_{11}$,
  $\delta(q_{11}, 0) = \delta(q_{11}, 1) = q_{11}$,
  $\delta(q_{\text{sink}}, 0) = \delta(q_{\text{sink}}, 1) = q_{\text{sink}}$, and
- $F = \{q_{11}\}$.

(a)



(b)

State	0	1
$\rightarrow q_0$	q_{sink}	q_1
q_1	q_{sink}	q_{11}
$* \; q_{11}$	q_{11}	q_{11}
q_{sink}	q_{sink}	q_{sink}

(c)

Figure 2.

For a given DFA A, we can define the **language Lang(A) of A** as the language of all words that are accepted by A. Formally, we extend the transition function δ to a function $\hat{\delta}\colon Q \times \Sigma^* \to Q$, i.e., $\hat{\delta}$ is defined on words instead of single letters. This is done inductively in a straightforward fashion. First, we define $\hat{\delta}(q, \varepsilon) = q$. Second, for a word w, which ends with the letter a, i.e., $w = va$ for some (possibly empty) word v, we have

$$\hat{\delta}(q, w) = \delta(\hat{\delta}(q, v), a) \; .$$

Intuitively, $\hat{\delta}(q, w)$ corresponds to the state in which the DFA ends if it starts in state q and reads the word w. Since a DFA A accepts exactly those words that, starting in the start state q_0, end in an accepting state, we can define the language of A by

$$\text{Lang}(A) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\} \; .$$

As we will see shortly, there are languages for which we cannot design any DFA. However, the languages for which we can form an important class, namely the so-called class of **regular languages**.

Definition 1.3 (Regular Language). *A language L is called **regular** if there is a DFA A with* $\text{Lang}(A) = L$. *The class of the regular languages is*

$$\mathcal{L}_{\text{reg}} = \{L \mid L \text{ is regular}\} \; .$$

In the following sections, we will learn about alternative ways to define regular languages, some of which are different automata models.

1.3 Nondeterministic Finite Automata

In this section, we introduce a seemingly more powerful model of pattern recognition, namely the so-called **nondeterministic finite automata**, NFAs for short. The main difference

between NFAs and DFAs is that the action of an NFA is not necessarily "determined" by its transition function and the given word. If a DFA is in some state q and reads the letter l, then it enters a fixed state q'. For an NFA, there may be a choice, i.e., there may be multiple states that can be entered, or even no state at all. Besides that, NFAs are defined analogously to DFAs.

Definition 1.4 (Nondeterministic Finite Automaton, NFA). *An NFA N is a quintuple $N = (Q, \Sigma, \delta, q_0, F)$, where*

- *Q is a finite set of **states**,*
- *Σ is the **input alphabet**,*
- *$\delta\colon Q \times \Sigma \to \mathrm{Pow}(Q)$ is the **transition function**,*
- *$q_0 \in Q$ is the **start state**, and*
- *F is the set of **accepting states**.*

We see that, as opposed to DFAs, δ is not a function that maps pairs of states and letters to single states, but to sets of states. Such a set may contain any number of states; in particular, it can be empty, contain a single state, or all of them. The extended transition function $\hat{\delta}$ is defined in the obvious way, i.e., it takes a state and a word and returns a set of states. The language of an NFA N is defined as

$$\mathrm{Lang}(N) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\} .$$

In order to grasp this expression, let us put it into words. $\mathrm{Lang}(N)$ contains all words for which there exists a **run** (**traversal**, **tour**, or **computation**) through N that ends in an accepting state. This means that, again starting in the start state q_0, we can follow transitions that are labeled with the corresponding letters; if there is more than one possibility, we can pick any and continue (if possible). If we are able to end up in an accepting state, the word is contained in $\mathrm{Lang}(N)$. Another point of view is that, in any state which offers multiple possibilities to continue, we can make a **nondeterministic guess**. If there is a sequence of guesses that allows us to enter an accepting state for a given word, the word is accepted. It is irrelevant what happens if we guessed differently; all that counts is whether there is one way to guess correctly. For words not in $\mathrm{Lang}(N)$, there is no such run, i.e., no matter which transitions are followed whenever there are multiple possibilities, N never ends in an accepting state.

Another property of an NFA is that it is not necessarily complete (in contrast to DFAs). If we enter a state and a letter is read for which there is no outgoing transition, we say that the NFA is "stuck." In this case, the word is not accepted in this run. However, as noted above, if there is an accepting run, the word is still accepted by the NFA.

An NFA for the language L_{11} is shown in Figure 3a, and we note that it contains one state fewer than the DFA A_{11} shown in Figure 2b. Indeed, the ability to make nondeterministic guesses often allows us to design automata that are somewhat simpler. As a second example,

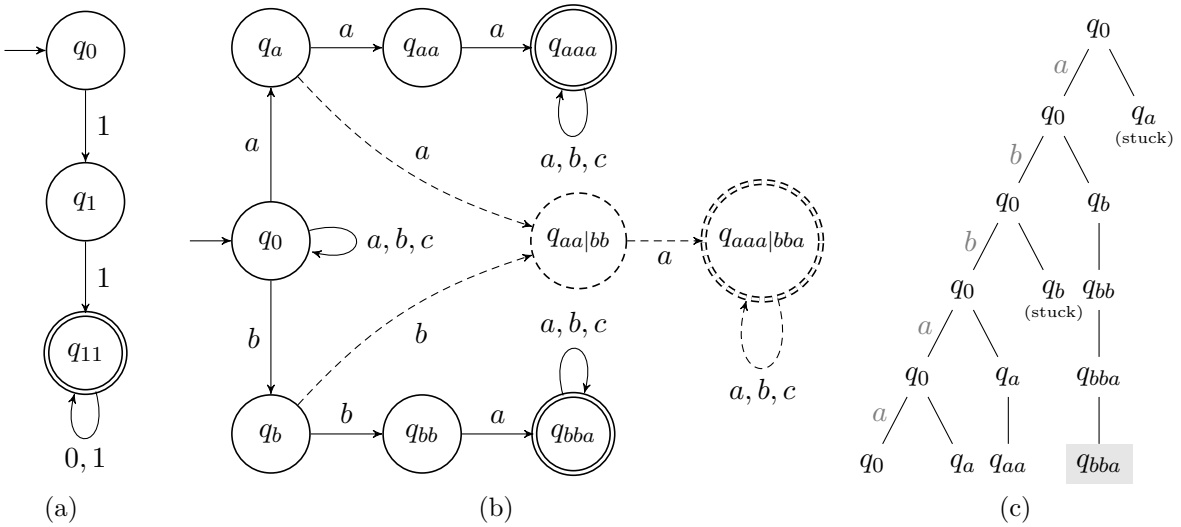


Figure 3.

consider the language

$$L_{aaa,bba} = \{w \in \{a,b,c\}^* \mid w \text{ contains the string } aaa \text{ or the string } bba\} .$$

An NFA for this language can initially read a (potentially empty) prefix of the given word until it guesses nondeterministically which of the two strings (if any) is contained in it; to this end, it loops in the start state. According to its guess, it then either enters q_a, q_{aa}, q_{aaa} or q_b, q_{bb}, q_{bba}, respectively; q_{aaa} and q_{bba} are accepting states, which allow the NFA to read any suffix. Conversely, if the given word does not contain the string aaa or bba, the NFA will never accept it, no matter how the guess at the beginning is made. The resulting NFA is shown in Figure 3b; there it is also indicated how some of the states can be merged in order to get a smaller NFA with only five states.

In order to get a better feeling for how an NFA N works on a given word w, we can visualize its computations using a **tree**. The root of the tree corresponds to the start state q_0. Then we consider the first letter of w, and add a child for each state that can be entered by N if this letter is read. We continue this process with all children in an iterative fashion. If there is no outgoing transition that can be followed for the given letter, N is stuck and the vertex of the tree becomes a leaf. Every computation that does not get stuck results in a leaf that corresponds to a state in which N is after reading the whole word and making the corresponding guesses. If there is such a leaf with an accepting state, N accepts in the corresponding computation, and hence w is in $\mathrm{Lang}(N)$. Every path from the root to a leaf corresponds to a computation of N on w, i.e., every branch represents a nondeterministic guess. An example for the NFA shown in Figure 3b and the word $abbaa$ is depicted in Figure 3c; we see that an accepting computation is given by the sequence $q_0, q_0, q_b, q_{bb}, q_{bba}, q_{bba}$ of states.

A natural question is whether NFAs are more powerful than DFAs, i.e., whether there is a language for which we can design an NFA, but no DFA. As a matter of fact, there is not. Both models have the same **expressive power**. This means that, for every NFA N,

there is a DFA A that accepts the same language N accepts, and vice versa; we also say that N and A **equivalent**.

Theorem 1.5. *Every DFA can be converted into an equivalent NFA.*

Proof. This implication is easy to see. Let A be a DFA with $\text{Lang}(A) = L$. Then A can be regarded as an NFA that makes no nondeterministic guesses at all. Formally, the only thing we need to change is to convert any transition $\delta(q, a) = q'$ to $\delta(q, a) = \{q'\}$. Clearly, the resulting NFA N accepts the language L. □

Theorem 1.6. *Every NFA can be converted into an equivalent DFA.*

Proof. This implication is not that straightforward. Let $N = (Q_N, \Sigma_N, \delta_N, q_{0,N}, F_N)$ be an NFA with $\text{Lang}(N) = L$. What we need to do is to design a DFA $A = (Q_A, \Sigma_A, \delta_A, q_{0,A}, F_A)$ with $\Sigma_A = \Sigma_N$ and $\text{Lang}(A) = L$. In what follows, we describe a method called the **powerset construction** that achieves this goal. The idea is to have A simulate all possible nondeterministic guesses of N in parallel; i.e., if N has two outgoing transitions from a state q labeled with l that lead to a state q' and a state q'', respectively, A has a state that contains the information "after reading l, N could be in either q' or q''." We denote such a state by $\langle\{q', q''\}\rangle$. Formally, this process can be described as follows.

- The start state of A is $q_{0,A} = \langle\{q_{0,N}\}\rangle$.
- For every letter $l \in \Sigma_A$, we create a new state of A that "contains" all states $p_{0,N}, p_{1,N}, \ldots, p_{m,N}$ of N with $p_{i,N} \in \delta_N(q_{0,N}, l)$; this state is denoted by

  $$q = \langle\{p_{0,N}, p_{1,N}, \ldots, p_{m,N}\}\rangle \ ,$$

  and we set $\delta_A(q_{0,A}, l) = q$.
- We iterate this process with the newly obtained states, e.g., with q. This way, the states of A correspond to sets of states of N (possibly also the empty set or Q_N); the fact that $Q_A \subseteq \text{Pow}(Q_N)$ is responsible for the construction's name.
- The process terminates if no new state is obtained and the transitions of all known states are defined.
- The accepting states F_A of A are exactly those states that "contain" at least one state from F_N.

Now consider some word $w \in \text{Lang}(N)$. This means that there is a run through N such that N ends in an accepting state, i.e., w induces a sequence

$$(p_{0,N}, p_{1,N}, \ldots, p_{|w|,N})$$

with $p_{0,N} = q_{0,N}$ and $p_{|w|,N} \in F_N$. By construction, w leads to a run

$$(p_{0,A}, p_{1,A}, \ldots, p_{|w|,A})$$

such that $p_{i,N}$ is one of the states that are contained in $p_{i,A}$. It follows that $p_{0,A}$ is the start state of A, and that the accepting state $p_{|w|,N}$ is contained in the state $p_{|w|,A}$, which implies that $p_{|w|,A}$ is also an accepting state. Therefore, A accepts w. By the same reasoning, if N does not accept w, then neither does A. □

The easiest way to apply the powerset construction is by explicitly constructing the transition table of the DFA from the NFA. First, let us have a look at the simplified NFA with five states depicted in Figure 3b. By definition, the start state of the DFA is $\langle\{q_0\}\rangle$. Now we look at the outgoing transitions. Reading an a, the NFA can either stay in q_0 or change to the state q_a, which leads to a new state $\langle\{q_0, q_a\}\rangle$. Likewise, we obtain a state $\langle\{q_0, q_b\}\rangle$ if the letter b is read. The NFA stays in q_0 if a c is read; analogously, the DFA stays in $\langle\{q_0\}\rangle$. This leads to the first row

State	a	b	c
$\to \langle\{q_0\}\rangle$	$\langle\{q_0, q_a\}\rangle$	$\langle\{q_0, q_b\}\rangle$	$\langle\{q_0\}\rangle$

of the transition table of the DFA we are about to design. We observe that there are two new states for which we need to specify the outgoing transitions. The first one represents the state "the NFA is in one of the two states q_0 or q_a." So what happens in this case if an a is read? If the NFA is in q_0, it either stays in q_0 or it again changes to the state q_a; if it is in q_a, it changes to $q_{aa|bb}$. Conversely, if a b is read, the NFA stays in q_0 or changes to q_b if it was in q_0; it gets stuck if it was in q_a, since there is no outgoing transition labelled b. Last, if a c is read, the NFA stays in q_0 or gets stuck. With this, we obtain

State	a	b	c
$\to \langle\{q_0\}\rangle$	$\langle\{q_0, q_a\}\rangle$	$\langle\{q_0, q_b\}\rangle$	$\langle\{q_0\}\rangle$
$\langle\{q_0, q_a\}\rangle$	$\langle\{q_0, q_a, q_{aa\|bb}\}\rangle$	$\langle\{q_0, q_b\}\rangle$	$\langle\{q_0\}\rangle$

as the first two rows of the transition table. We continue in this fashion until we do not find any new state and the transitions of all states found are defined. The next state is $\langle\{q_0, q_b\}\rangle$, the one after that $\langle\{q_0, q_a, q_{aa|bb}\}\rangle$, and so on. We finally get

	State	a	b	c
$\to$	$\langle\{q_0\}\rangle$	$\langle\{q_0, q_a\}\rangle$	$\langle\{q_0, q_b\}\rangle$	$\langle\{q_0\}\rangle$
	$\langle\{q_0, q_a\}\rangle$	$\langle\{q_0, q_a, q_{aa\|bb}\}\rangle$	$\langle\{q_0, q_b\}\rangle$	$\langle\{q_0\}\rangle$
	$\langle\{q_0, q_b\}\rangle$	$\langle\{q_0, q_a\}\rangle$	$\langle\{q_0, q_b, q_{aa\|bb}\}\rangle$	$\langle\{q_0\}\rangle$
	$\langle\{q_0, q_a, q_{aa\|ab}\}\rangle$	$\langle\{q_0, q_a, q_{aa\|bb}, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_b\}\rangle$	$\langle\{q_0\}\rangle$
	$\langle\{q_0, q_b, q_{aa\|ab}\}\rangle$	$\langle\{q_0, q_a, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_b, q_{aa\|bb}\}\rangle$	$\langle\{q_0\}\rangle$
$*$	$\langle\{q_0, q_a, q_{aa\|bb}, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_a, q_{aa\|bb}, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_b, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_{aaa\|bba}\}\rangle$
$*$	$\langle\{q_0, q_a, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_a, q_{aa\|bb}, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_b, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_{aaa\|bba}\}\rangle$
$*$	$\langle\{q_0, q_b, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_a, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_b, q_{aa\|bb}, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_{aaa\|bba}\}\rangle$
$*$	$\langle\{q_0, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_a, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_b, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_{aaa\|bba}\}\rangle$
$*$	$\langle\{q_0, q_b, q_{aa\|bb}, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_a, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_b, q_{aa\|bb}, q_{aaa\|bba}\}\rangle$	$\langle\{q_0, q_{aaa\|bba}\}\rangle$

as the complete transition table. Note that this DFA has ten states, i.e., twice as many as the original NFA.

As a second example, consider the NFA in Figure 3a. If a one is read, the NFA enters the state q_1; conversely, there is no outgoing edge labeled zero, which means that the set of states entered is empty. This leads to the first line

State	0	1
$\to \langle\{q_0\}\rangle$	$\langle\emptyset\rangle$	$\langle\{q_1\}\rangle$

of the transition table. There are two new states, one of which is the empty set, which corresponds to the NFA being stuck; once this state is entered, it is never left, independent

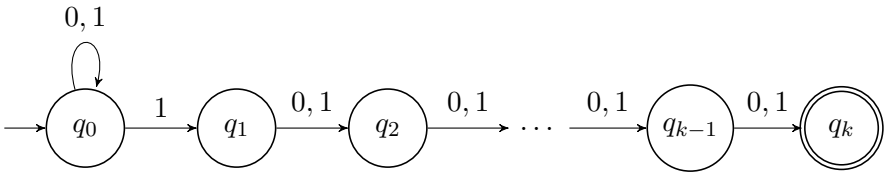


Figure 4.

of the input symbols read. After applying the powerset construction, we obtain

State	0	1
$\to \langle\{q_0\}\rangle$	$\langle\emptyset\rangle$	$\langle\{q_1\}\rangle$
$\langle\emptyset\rangle$	$\langle\emptyset\rangle$	$\langle\emptyset\rangle$
$\langle\{q_1\}\rangle$	$\langle\emptyset\rangle$	$\langle\{q_{11}\}\rangle$
$* \langle\{q_{11}\}\rangle$	$\langle\{q_{11}\}\rangle$	$\langle\{q_{11}\}\rangle$

as the complete transition table of the DFA accepting L_{11}.

We observe that the DFA designed in the proof of Theorem 1.6 is generally a lot larger than the given NFA. Since, in principle, the so-designed DFA A can have a single state for every possible combination of states of the NFA N, it seems reasonable to assume cases where

$$|Q_A| = 2^{|Q_N|} .$$

We might of course wonder whether this is only due to the powerset construction; maybe there is another way of conversion that does not increase the number of states that drastically. Unfortunately, this "exponential blowup" is indeed necessary. In what follows, we give a language for which any DFA needs exponentially more states than an NFA. For every $k \in \mathbb{N}$, consider the language

$$L_{\text{right},k} = \{w \in \{0,1\}^* \mid \text{ the } k\text{th letter from the right is } 1\} .$$

Let us first design an NFA $N_{\text{right},k}$ with $k+1$ states for $L_{\text{right},k}$. The idea is that $N_{\text{right},k}$ guesses when the kth position from the right is read. Before that, arbitrarily many letters can be read; after the guess, exactly $k-1$ more letters are read. Words not in the language cannot be accepted this way; $N_{\text{right},k}$ is shown in Figure 4.

Now we formally prove that every DFA has to have a number of states that is exponentially larger than that of $N_{\text{right},k}$.

Lemma 1.7. *Every DFA for $L_{\text{right},k}$ has at least 2^k states.*

Proof. For a contradiction, suppose that this is not the case, i.e., there is a DFA $A_{\text{right},k}$ with $\text{Lang}(A_{\text{right},k}) = L_{\text{right},k}$ and with at most $2^k - 1$ states. Now consider all binary words of length k. All these words each induce a tour through $A_{\text{right},k}$, ending in some state. Since there 2^k such words, but only at most $2^k - 1$ states, by the **pigeonhole principle**, two binary words have to end up in the same state q; let $b = b_1 b_2 \dots b_k$ and $b' = b'_1 b'_2 \dots b'_k$ with $b_i, b'_i \in \{0,1\}$ be two such words, and let j with $1 \le j \le k$ be the first position at which

they are different. Without loss of generality, let the jth bit of b be one and that of b' zero. Now consider the two words

$$b_1 b_2 \dots b_{j-1} \underbrace{1 b_{j+1} \dots b_k 0 \dots 0}_{k \text{ letters}} \quad \text{and} \quad b'_1 b'_2 \dots b'_{j-1} \underbrace{0 b'_{j+1} \dots b'_k 0 \dots 0}_{k \text{ letters}}$$

that are obtained by concatenating $k - (k - j + 1) = j - 1$ zeros to b and b', respectively. By the definition of $L_{\text{right},k}$, the first one has to be accepted by $A_{\text{right},k}$, while the second one must not be accepted. However, by assumption, after reading the first k letters of both words, the DFA is in q. After that, reading $j - 1$ zeros leads to ending up in the same state q'. As a consequence, either both words get accepted or they both do not get accepted, which is a contradiction. □

Note that our arguments do not rely on the powerset construction, but we proved that no DFA for $L_{\text{right},k}$ can have fewer than 2^k states, not matter how it is obtained.

1.4 Nondeterministic Finite Automata with ε-Transitions

Let us introduce yet another automata model. The so-called **nondeterministic finite automata with ε-transitions**, ε-NFAs for short, are NFAs with the added property that transitions are allowed to be labeled with the empty word ε. When reading a word w, such an **ε-transition** can always be followed without reading any letter from w. Formally, the transition function δ is extended to be defined on $\{\varepsilon\} \cup \Sigma$ instead of Σ. This can come in very handy when proving some given language to be regular. Consider the language

$$L_{ab,ba} = \{w \in \{a, b, c\}^* \mid w \text{ starts with } ab \text{ or ends with } ba\} .$$

An ε-NFA for this language guesses which of the two constraints (if any) is satisfied and follows a corresponding ε-transition; see Figure 5a.

We now define ε-close(q) to be the **ε-closure of a given state q**, which simply denotes the set of all states that are reachable from q by following only ε-transitions. Formally, we define ε-close(q) inductively by

- $q \in \varepsilon$-close(q) and
- if there is an ε-transition from some $q' \in \varepsilon$-close(q) to a state q'', then $q'' \in \varepsilon$-close(q).

Instead of defining ε-closures on single states, they can also be defined in a straightforward fashion on sets of states. We do not need this, however, with one exception; we define

$$\varepsilon\text{-close}(\emptyset) = \emptyset .$$

Consider the ε-NFA shown in Figure 5b and its start state q_0. We initially set ε-close(q_0) = $\{q_0\}$. Second, there is an ε-transition from q_0 to q_2; therefore, we obtain ε-close(q_0) = $\{q_0, q_2\}$. After that, we add q_1 to ε-close(q_0) due to the ε-transition from q_2 to q_1. Since there are no additional ε-transitions to any state that is not already contained in the ε-closure of q_0, we finally have ε-close(q_0) = $\{q_0, q_1, q_2\}$.

In what follows, we prove that the expressive power of ε-NFAs is not larger than that of DFAs. We again start with the simpler direction, which can be seen analogously to Theorem 1.5.

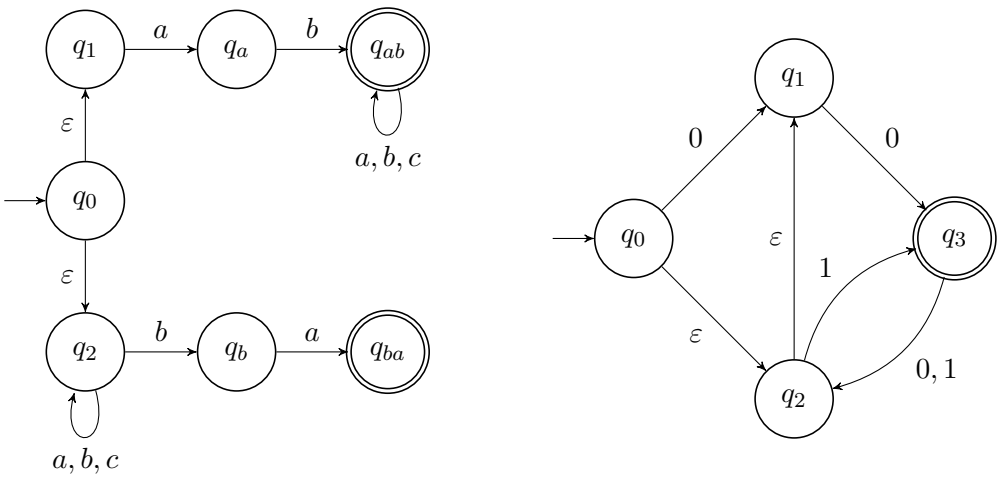


Figure 5.

Theorem 1.8. *Every DFA can be converted into an equivalent ε-NFA.*

Proof. This implication is again easy to see since every DFA can be regarded as an ε-NFA with neither nondeterministic guesses, nor ε-transitions. □

Now we show how to convert a given ε-NFA to an equivalent DFA. The idea of the proof follows a similar approach as the proof of Theorem 1.6 while paying special attention to ε-transitions.

Theorem 1.9. *Every ε-NFA can be converted into an equivalent DFA.*

Proof. We use a variant of the powerset construction that incorporates the possibility to follow ε-transitions after every letter from the input word that is read; we call the resulting method the **powerset construction with ε-closures**. Let $N = (Q_N, \Sigma_N, \delta_N, q_{0,N}, F_N)$ denote the given ε-NFA with $\mathrm{Lang}(N) = L$, from which we design a DFA $A = (Q_A, \Sigma_A, \delta_A, q_{0,A}, F_A)$ with $\Sigma_A = \Sigma_N$ and $\mathrm{Lang}(A) = L$. The construction works as follows.

- The start state of A is $q_{0,A} = \langle \varepsilon\text{-close}(q_{0,N}) \rangle$.
- For every letter $l \in \Sigma_A$, we create a new state of A that contains all states from $\varepsilon\text{-close}(p_{1,N}), \varepsilon\text{-close}(p_{2,N}), \ldots, \varepsilon\text{-close}(p_{m,N})$ of N with $p_{i,N} \in \delta_N(p, l)$ for any $p \in \varepsilon\text{-close}(q_{0,N})$; this state is denoted by

  $$q = \langle \varepsilon\text{-close}(p_{1,N}) \cup \varepsilon\text{-close}(p_{2,N}) \cup \cdots \cup \varepsilon\text{-close}(p_{m,N}) \rangle \ ,$$

  and we set $\delta_A(q_{0,A}, l) = q$.
- As in the original construction, we iterate this process with the newly obtained states.

- The process terminates if no new state is obtained.

Now consider a word $w \in \mathrm{Lang}(N)$. As in the proof of Theorem 1.6, this induces a run through N, which may use ε-transitions between any two letters of w. This can be translated into a corresponding run through A that ends in an accepting state. Whenever the run through N follows an ε-transition, this transition is taken into consideration by adding the corresponding states due to the ε-closures. Thus, $w \in \mathrm{Lang}(A)$. Conversely, if N does not accept w, then neither does A. □

As for NFAs without ε-transitions, the easiest way to construct a DFA from the given ε-NFA is to define the DFA's transition table right away. Let us consider the ε-NFA depicted in Figure 5b. As noted above, the start state q_0 of the ε-NFA has the ε-closure $\varepsilon\text{-close}(q_0) = \{q_0, q_1, q_2\}$, which corresponds to the start state $\langle\{q_0, q_1, q_2\}\rangle$ of the DFA we are about to design. Now we consider all three states q_0, q_1, and q_2, and see where we get reading the letter 0. From q_0, we go to q_1, and we have $\varepsilon\text{-close}(q_1) = \{q_1\}$; from q_1, we go to q_3, and $\varepsilon\text{-close}(q_3) = \{q_3\}$. From q_2, there are no outgoing transitions labeled 0; note that, in this case we do not follow the ε-transition.[1] The new state of the DFA is thus $\langle\{q_1, q_3\}\rangle$.

Next, we look at reading the letter 1 while being in either q_0, q_1, or q_2. From both q_0 and q_1, there are no outgoing transitions labeled 1; from q_2, we go to q_3, and we have $\varepsilon\text{-close}(q_3) = \{q_3\}$ This leads to the first row

	State	0	1
→	$\langle\{q_0, q_1, q_2\}\rangle$	$\langle\{q_1, q_3\}\rangle$	$\langle\{q_3\}\rangle$

of the DFA's transition table. Now we continue this procedure with the two new states $\langle\{q_1, q_3\}\rangle$ and $\langle\{q_3\}\rangle$ of our DFA, which gives

	State	0	1
→	$\langle\{q_0, q_1, q_2\}\rangle$	$\langle\{q_1, q_3\}\rangle$	$\langle\{q_3\}\rangle$
*	$\langle\{q_1, q_3\}\rangle$	$\langle\{q_1, q_2, q_3\}\rangle$	$\langle\{q_1, q_2\}\rangle$
*	$\langle\{q_3\}\rangle$	$\langle\{q_1, q_2\}\rangle$	$\langle\{q_1, q_2\}\rangle$

as intermediate transition table with two new states, namely $\langle\{q_1, q_2, q_3\}\rangle$ and $\langle\{q_1, q_2\}\rangle$. We follow the above approach, and finally get the table

	State	0	1
→	$\langle\{q_0, q_1, q_2\}\rangle$	$\langle\{q_1, q_3\}\rangle$	$\langle\{q_3\}\rangle$
*	$\langle\{q_1, q_3\}\rangle$	$\langle\{q_1, q_2, q_3\}\rangle$	$\langle\{q_1, q_2\}\rangle$
*	$\langle\{q_3\}\rangle$	$\langle\{q_1, q_2\}\rangle$	$\langle\{q_1, q_2\}\rangle$
*	$\langle\{q_1, q_2, q_3\}\rangle$	$\langle\{q_1, q_2, q_3\}\rangle$	$\langle\{q_1, q_2, q_3\}\rangle$
	$\langle\{q_1, q_2\}\rangle$	$\langle\{q_3\}\rangle$	$\langle\{q_3\}\rangle$

as a result.

[1] Formally, since there is no outgoing transition labeled 1, the resulting state is $\emptyset$ and $\varepsilon\text{-close}(\emptyset) = \emptyset$.

1.5 Regular Expressions

Another pattern recognition mechanism are **regular expressions**. Regular expressions can be defined inductively as follows; for a given regular expression R, $\mathrm{Lang}(R) \subseteq \Sigma^*$, for some alphabet Σ, denotes the language described by R.

- The empty word ε is a regular expression that describes the language $\mathrm{Lang}(\varepsilon) = \{\varepsilon\}$, which contains the empty word only.
- The empty set $\emptyset$ is a regular expression describing the empty language $\mathrm{Lang}(\emptyset) = \emptyset$.
- Every letter $l \in \Sigma$ is a regular expression that describes the language $\mathrm{Lang}(l) = \{l\}$.
- If R is a regular expression, then R^* is a regular expression that describes the language $\mathrm{Lang}(R^*) = \mathrm{Lang}(R)^*$; the regular expression R^+ is defined in the obvious way.
- If R_1 and R_2 are regular expressions, then $R_1 + R_2$ is a regular expression that describes the language $\mathrm{Lang}(R_1 + R_2) = \mathrm{Lang}(R_1) \cup \mathrm{Lang}(R_2)$.
- If R_1 and R_2 are regular expressions, then $R_1 \circ R_2 = R_1 R_2$ is a regular expression that describes the language $\mathrm{Lang}(R_1 R_2) = \mathrm{Lang}(R_1)\mathrm{Lang}(R_2)$.

In order to evaluate a given regular expression, we need to fix the **precedence**. The following rules are analogous to those of arithmetic; exponentiation (Kleene star) before multiplication (concatenation) before addition (union).

- The Kleene star is of highest precedence, e.g., $R_1 + R_2^* \neq (R_1 + R_2)^*$.
- Next comes concatenation, e.g., $R_1 R_2 + R_3 \neq R_1(R_2 + R_3)$.
- Finally comes union.
- Otherwise, parentheses have to be used.

Moreover, there are a couple of rules we need to know about. First, let us state basic ways how to treat the special expressions ε and $\emptyset$. For every regular expression R, we have

- $\varepsilon R = R = R\varepsilon$,
- $\emptyset R = \emptyset = R\emptyset$, and
- $\emptyset + R = R = R + \emptyset$.

Second, we have laws of **commutativity** for the union operation, and **distributivity** and **associativity** for binary operations. More formally, for any three regular expressions R_1, R_2, and R_3, we have

- $R_1 + R_2 = R_2 + R_1$,
- $R_1(R_2 R_3) = (R_1 R_2)R_3$,

- $R_1 + (R_2 + R_3) = (R_1 + R_2) + R_3$, and
- $R_1(R_2 + R_3) = R_1R_2 + R_1R_3$.

Consider the language L_{11} of binary words that start with 11 as described above. A regular expression can be obtained by a bottom-up approach. First, the regular expression 1 gives the language $\text{Lang}(1) = \{1\}$ that only contains the single word 1. Second, the regular expression 11 describes the language $\text{Lang}(1)\text{Lang}(1)$, which describes the regular expression that corresponds to $\text{Lang}(11) = \{11\}$. Furthermore, $0 + 1$ corresponds to $\text{Lang}(0 + 1) = \{0\} \cup \{1\}$, and $(0 + 1)^*$ represents $\text{Lang}(0 + 1)^* = (\{0\} \cup \{1\})^*$. Finally, we obtain

$$11(0 + 1)^*$$

as a regular expression for L_{11}. Let us give a few more examples.

- $(0 + 1)^*101(0 + 1)^*$ is a regular expression for the language of binary words that contain the pattern 101.
- $(ab)^* + (ba)^* + b(ab)^* + a(ba)^*$ is a regular expression for the language that contains all words that consist of alternating as and bs (including ε, a, and b).
- $(\varepsilon + b)(ab)^*(\varepsilon + a)$ is also a regular expression for the preceding language.

We now show that regular expressions and DFAs have the same expressive power, i.e., for every regular expression there is an equivalent DFA, and vice versa.

Theorem 1.10. *Every regular expression can be converted into an equivalent DFA.*

Proof. The idea behind this proof is to design an ε-NFA from a given regular expression in a bottom-up fashion. We will make sure that all intermediate ε-NFAs have exactly one start state and one accepting state. We start with smallest subexpressions for which we build ε-NFAs that each consist of two states; one of them is the start state, the other one is an accepting state.

- We start with the subexpression $\emptyset$, for which we build an ε-NFA as shown in Figure 6a.
- For the subexpression ε, we build an ε-NFA as depicted in Figure 6b.
- For every subexpression that corresponds to a single letter $l \in \Sigma$, we apply the construction in Figure 6c.

Now we iteratively build larger and larger automata for larger and larger subexpressions, yielding an ε-NFA for the complete expression.

- Let R_1 and R_2 be two subexpressions for which we already designed two ε-NFAs N_1 and N_2. Suppose the given regular expression contains the subexpression R_1R_2. Then we build an ε-NFA for R_1R_2 by connecting N_1 and N_2 by an ε-transition. Since both automata have exactly one start state and one accepting state, this is always possible. The accepting property of N_1 is removed such that the resulting ε-NFA again has one start state and one accepting state. The construction is shown in Figure 6d.

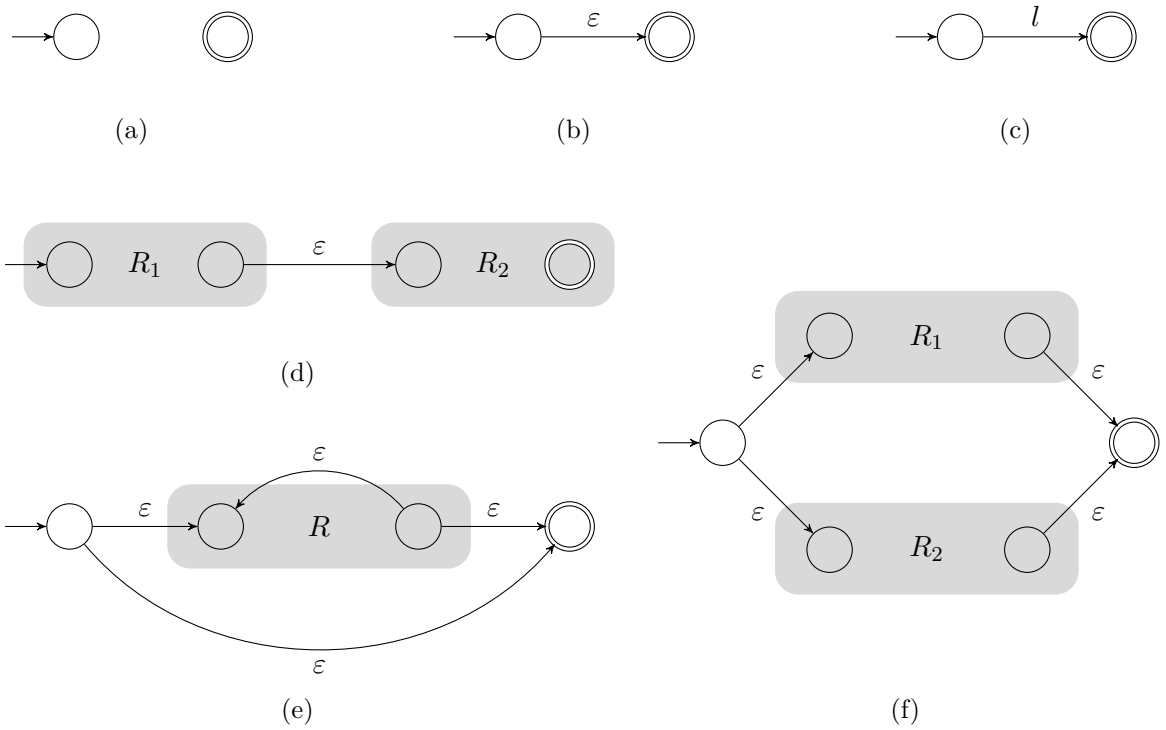


Figure 6.

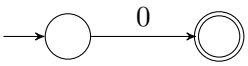
- If the subexpression R^* appears where we already designed an ε-NFA for the subexpresion R, we design an ε-NFA by the construction shown in Figure 6e. Note that the new ε-NFA has both a start and accepting state.
- For two subexpressions R_1 and R_2 with ε-NFAs N_1 and N_2, we can build an ε-NFA N for $R_1 + R_2$ by adding a new start state that has an ε-transition to each of the start states of N_1 and N_2. Likewise, the two accepting states are made nonaccepting and connected by two ε-transitions to a new accepting state. The resulting ε-NFA is shown in Figure 6f.

The claim then follows since every ε-NFA can be converted into an equivalent DFA, as stated by Theorem 1.9. □

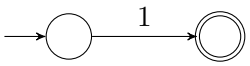
Let us give an example in order to apply the construction used in the proof of Theorem 1.10. To this end, consider the regular expression

$$01^* + 1 \,.$$

We first build ε-NFAs for the smallest subexpressions 0 and 1 as shown in Figure 6c, yielding

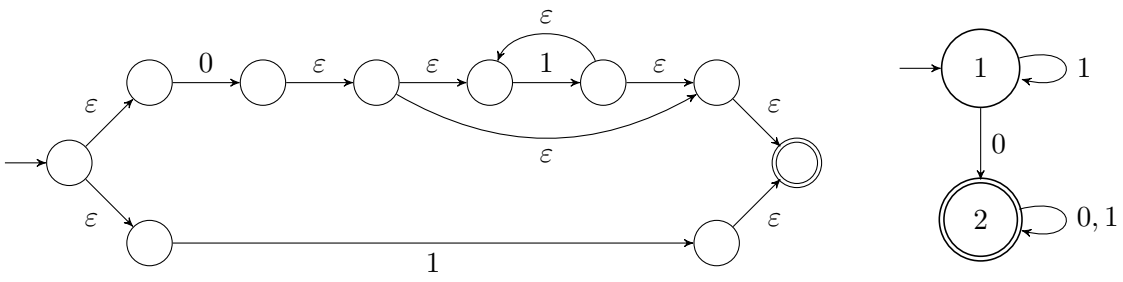


and



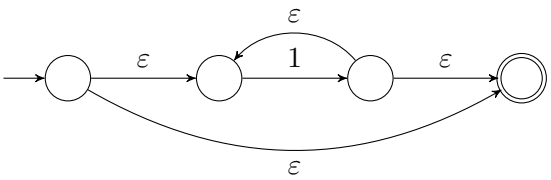
.

Following the precedence constraints of regular expressions, we consider the next largest subexpression 1^* and make use of the second ε-NFA we just build. This leads to

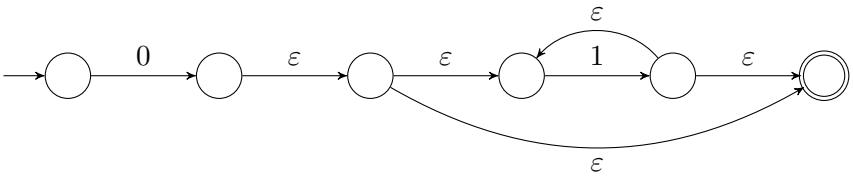


(a) (b)

Figure 7.



as a next step, following the construction from Figure 6e. After that, we build the ε-NFA



for 01^* by using the construction from Figure 6d. Finally, we take the preceding ε-NFA and the one for 1 and obtain an ε-NFA for the complete regular expression; the result is shown in Figure 7a.

We note that the final ε-NFA has many unnecessary ε-transitions. However, with these transitions present, we are on the safe side and can easily argue that the iterative construction works, which is why we stick to it.

Now we have a look at the reverse operation.

Theorem 1.11. *Every DFA can be converted into an equivalent regular expression.*

Proof. We show how to design a regular expression from a given DFA A with m states using dynamic programming. For ease of presentation, we assume that the states of A are simply labeled $1, 2, \ldots, m$ with $q_0 = 1$ being the start state.

- $R_{ij}^{(0)}$ is a regular expression that corresponds to going from state i to j without using another state. If there is a transition from i to j labeled with k letters $a_1, a_2, \ldots, a_k$, we set $R_{ij}^{(0)} = a_1 + a_2 + \cdots + a_k$; if there is no transition from i to j, $R_{ij}^{(0)} = \emptyset$.

  A special case is that $i = j$. If there is a loop on i labeled by k letters $a_1, a_2, \ldots, a_k$, we set $R_{ii}^{(0)} = \varepsilon + a_1 + a_2 + \cdots + a_k$; if there is no loop, we set $R_{ii}^{(0)} = \varepsilon$.

- Now we iteratively allow more and more intermediate states. First, we compute $R_{ij}^{(1)}$, i.e., we look at all possibilities to go from i to j while allowing to use the intermediate state 1. There are two possibilities.
  - Either the intermediate state 1 is not used, i.e., we can go from i to j directly, which corresponds to the regular expression $R_{ij}^{(0)}$;
  - or the intermeditate state 1 is used. In this case, we can split the tour from i to j into three parts. First, a transition from i to 1 is taken. Second, we can follow any number of loops in 1 (if there is a loop). Last, a transition from 1 to j is taken.

  To sum up, we obtain

  $$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)}\left(R_{11}^{(0)}\right)^* R_{1j}^{(0)} ,$$

  where we have already computed all expressions involved.

  We can generalize this approach for $R_{ij}^{(k)}$ with $k \geq 1$, i.e., for the case that we allow intermediate states $1, 2, \ldots, k$. Suppose we have already computed $R_{ij}^{(k-1)}$ for all i and j, and now allow to use the intermediate state k. Then we obtain

  $$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}\left(R_{kk}^{(k-1)}\right)^* R_{kj}^{(k-1)}$$

  by the same reasoning as above, i.e., the two terms represent the following two possibilities.
  - The intermediate state k is not used and thus only the intermediate states $1, 2, \ldots, k-1$ appear on the tour;
  - or k is used as intermediate state on the way from i to j. This again means that we can split the tour into three parts. We can represent this tour by

    where we mark all the occurrences of the state k. The first part now is a tour that starts at i and ends with the first occurrence of k. We know that on this tour only the states $1, 2, \ldots, k-1$ appear; this corresponds to the expression $R_{ik}^{(k-1)}$. Likewise, the tour from the last appearance of k to j corresponds to $R_{kj}^{(k-1)}$ and constitutes the third part. For the second part, consider any two consecutive occurrences of the state k. On any such tour, again only the states $1, 2, \ldots, k-1$ appear, which thus corresponds to $R_{kk}^{(k-1)}$.

Finally, let $i_1, \ldots, i_r$ denote the accepting states of A. We note that the language $\mathrm{Lang}(A)$ is described by the regular expression

$$R_{1i_1}^{(m)} + R_{1i_2}^{(m)} + \cdots + R_{1i_r}^{(m)} ,$$

which is thus equivalent to A. □

Again, let us consolidate our intuition by applying the procedure described in the proof of Theorem 1.11 to a concrete DFA, namely the simple one with two states shown in Figure 7b. To sum up, we get

$$R_{11}^{(0)} = \varepsilon + 1, \quad R_{12}^{(0)} = 0, \quad R_{21}^{(0)} = \emptyset, \quad \text{and } R_{22}^{(0)} = \varepsilon + 0 + 1 \,.$$

Now we allow the intermediate state 1, yielding

- $R_{11}^{(1)} = R_{11}^{(0)} + R_{11}^{(0)}\left(R_{11}^{(0)}\right)^* R_{11}^{(0)} = (\varepsilon + 1) + (\varepsilon + 1)(\varepsilon + 1)^*(\varepsilon + 1) = 1^*$,
- $R_{12}^{(1)} = R_{12}^{(0)} + R_{11}^{(0)}\left(R_{11}^{(0)}\right)^* R_{12}^{(0)} = 0 + (\varepsilon + 1)(\varepsilon + 1)^*0 = 1^*0$,
- $R_{21}^{(1)} = R_{21}^{(0)} + R_{21}^{(0)}\left(R_{11}^{(0)}\right)^* R_{11}^{(0)} = \emptyset + \emptyset(\varepsilon + 1)^*(\varepsilon + 1) = \emptyset$, and
- $R_{22}^{(1)} = R_{22}^{(0)} + R_{21}^{(0)}\left(R_{11}^{(0)}\right)^* R_{12}^{(0)} = (\varepsilon + 0 + 1) + \emptyset(\varepsilon + 1)^*0 = \varepsilon + 0 + 1$.

Also allowing the states 1 and 2 as intermediate states, we get

- $R_{11}^{(2)} = R_{11}^{(1)} + R_{12}^{(1)}\left(R_{22}^{(1)}\right)^* R_{21}^{(1)} = 1^* + 1^*0(\varepsilon + 0 + 1)^*\emptyset = 1^*$,
- $R_{12}^{(2)} = R_{12}^{(1)} + R_{12}^{(1)}\left(R_{22}^{(1)}\right)^* R_{22}^{(1)} = 1^*0 + 1^*0(\varepsilon + 0 + 1)^*(\varepsilon + 0 + 1) = 1^*0(0 + 1)^*$,
- $R_{21}^{(2)} = R_{21}^{(1)} + R_{22}^{(1)}\left(R_{22}^{(1)}\right)^* R_{21}^{(1)} = \emptyset + (\varepsilon + 0 + 1)(\varepsilon + 0 + 1)^*\emptyset = \emptyset$, and
- $R_{22}^{(2)} = R_{22}^{(1)} + R_{22}^{(1)}\left(R_{22}^{(1)}\right)^* R_{22}^{(1)} = (\varepsilon + 0 + 1) + (\varepsilon + 0 + 1)(\varepsilon + 0 + 1)^*(\varepsilon + 0 + 1) = (0 + 1)^*$.

Since 2 is the only accepting state, it follows that $R_{12}^{(2)}$ is the regular expression that corresponds to the given DFA; of course, it would not have been necessary to compute the other expressions (and the corresponding intermediate steps) in this case.

Let us now summarize one of the key points we learned about so far.

Theorem 1.12. *The following statements are equivalent.*

1. *L is a regular language.*
2. *There is a DFA A with* $\mathrm{Lang}(A) = L$.
3. *There is an NFA N with* $\mathrm{Lang}(N) = L$.
4. *There is an ε-NFA N with* $\mathrm{Lang}(N) = L$.
5. *There is a regular expression R with* $\mathrm{Lang}(R) = L$.

1.6 Closure Properties of Regular Languages

A very convenient fact about regular languages is that they are closed under many operations such as union, intersection, complement, etc. In this section, we will discuss a few of them. In order to prove the following claims, we use different approaches that rely on the fact that, if a language is regular, this implies the existence of a corresponding DFA, NFA, ε-NFA, or regular expression.

We start by **complementing** a given language L over an alphabet Σ, i.e., by considering the language $L^{\complement} = \Sigma^* \setminus L = \{w \in \Sigma^* \mid w \notin L\}$.

Lemma 1.13. *If a language L is regular, then $L^{\complement}$ is a regular language, too.*

Proof. Since L is regular, there is a DFA $A = (Q, \Sigma, \delta, q_0, F)$ with $\mathrm{Lang}(A) = L$. Now we can easily design a DFA $\overline{A}$ with $\mathrm{Lang}(\overline{A}) = L^{\complement}$ that is equal to A with the only difference that the accepting states are complemented. Hence, we get $\overline{A} = (Q, \Sigma, \delta, q_0, Q \setminus F)$. Now let $w \in L$; thus, w ends in an accepting state of A. Since this state is made nonaccepting in $\overline{A}$, we immediately get $w \notin \mathrm{Lang}(\overline{A})$. Conversely, if w is not accepted by A, it has to be accepted by $\overline{A}$. It follows that $\overline{A}$ is a DFA for $L^{\complement}$, i.e., $L^{\complement} = \mathrm{Lang}(A)^{\complement} = \mathrm{Lang}(\overline{A})$. □

Next, for any word w, let w^{R} denote the **reversal of w**, i.e., the word written backwards. For any language L, the **reversal of L**, denoted by L^{R}, is the language of all reversals of words in L. For example, we have $L^{\mathsf{R}} = \{a, ba, cb, bba\}$ for $L = \{a, ab, bc, abb\}$.

Lemma 1.14. *If a language L is regular, then L^{R} is a regular language, too.*

Proof. Again, since L is regular, there is a DFA $A = (Q, \Sigma, \delta, q_0, F)$ with $\mathrm{Lang}(A) = L$. Now we design an ε-NFA $N = (Q \cup \{q_0'\}, \Sigma, \delta', q_0', F')$ with $\mathrm{Lang}(N) = L^{\mathsf{R}}$. The transition function δ' is obtained by reversing all transitions given by the original transition function δ. Then we add a new start state q_0' that is connected to all states from F by ε-transitions. Finally, we set $F' = \{q_0\}$, i.e., the only accepting state of N is the starting state of A. Now let $w \in L$, and thus w ends in an accepting state of A. More specifically, consider the tour

$$(p_0, p_1, \ldots, p_{|w|})$$

with $p_0 = q_0$ and $p_{|w|} \in F$. This implies that there is a tour

$$(q_0', p_{|w|}, p_{|w|-1}, \ldots, p_0)$$

through N, and by the definition of N the transition from q_0' to $p_{|w|}$ is an ε-transition and $p_0 \in F$. This tour thus corresponds to $\varepsilon w^{\mathsf{R}} = w^{\mathsf{R}}$. Hence, w^{R} is accepted by N. Conversely, if w is not accepted by A, there cannot be an accepting tour through N induced by w^{R}. Hence, $L^{\mathsf{R}} = \mathrm{Lang}(A)^{\mathsf{R}} = \mathrm{Lang}(N)$. □

As an example to apply the construction used in the proof of Lemma 1.14, consider the language

$$L_{01,11} = \{w \in \{0,1\}^* \mid w \text{ starts with 01 or is a string of 1s with positive even length}\},$$

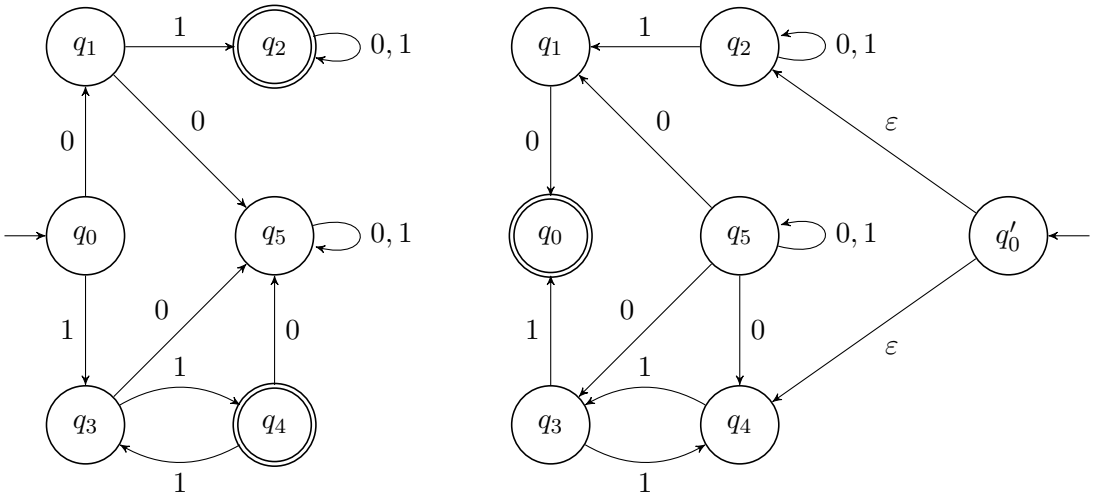


Figure 8.

for which Figure 8a shows a DFA, and Figure 8b gives the ε-NFA for $L^{\mathsf{R}}_{01,11}$ (with a redundant state q_5).

The proof of Lemmata 1.13 and 1.14 used the existence of a DFA for a given regular language. However, due to Theorem 1.12, we also know that we can design regular expressions for regular languages. The following lemmata can be proven in a straightforward fashion due to this fact.

Lemma 1.15. *If a language L is regular, then L^* is a regular language, too.*

Proof. Since L is regular, there is a regular expression R with $\mathrm{Lang}(R) = L$. Due to the definition of regular expressions, the regular expression $(R)^*$ corresponds to the language L^*, which is therefore also regular. □

So far, we dealt with unary operations, i.e., operations on a single regular language. Now let us consider binary operations, i.e., how to obtain regular languages by combining two given regular languages in a particular way.

Lemma 1.16. *If two languages L_1 and L_2 are regular, then L_1L_2 is a regular language.*

Proof. Since L_1 and L_2 are regular, there are two regular expressions R_1 and R_2 with $\mathrm{Lang}(R_1) = L_1$ and $\mathrm{Lang}(R_2) = L_2$. Due to the definition of regular expressions, we have that the regular expression R_1R_2 corresponds to the language L_1L_2. □

Lemma 1.17. *If two languages L_1 and L_2 are regular, then $L_1 \cup L_2$ is a regular language.*

Proof. This time, the regular expression $R_1 + R_2$ corresponds to the language $L_1 \cup L_2$. □

While the proof for the union of two regular expressions was almost trivial, it seems to be more tricky if we think about the intersection. Specifically, we did not define any operation on regular expressions R_1 and R_2 with $\mathrm{Lang}(R_1) = L_1$ and $\mathrm{Lang}(R_2) = L_2$ that corresponds to $L_1 \cap L_2$. However, in this case, we can again use the existence of two DFAs A_1 and A_2 with $\mathrm{Lang}(A_1) = L_1$ and $\mathrm{Lang}(A_2) = L_2$. From these, we design a third DFA, which we call the **product automaton** of A_1 and A_2.

Lemma 1.18. *If two languages L_1 and L_2 are regular, then $L_1 \cap L_2$ is a regular language.*

Proof. Since L_1 and L_2 are regular, there are two DFAs $A_1 = (Q_1, \Sigma_1, \delta_1, q_0, F_1)$ and $A_2 = (Q_2, \Sigma_2, \delta_2, p_0, F_2)$ with $\mathrm{Lang}(A_1) = L_1$ and $\mathrm{Lang}(A_2) = L_2$. Without loss of generality, we assume that $\Sigma_1 = \Sigma_2$, and $Q_1 = \{q_0, q_1, \ldots, q_{m-1}\}$ and $Q_2 = \{p_0, p_1, \ldots, p_{n-1}\}$. Now we design a DFA A with $\mathrm{Lang}(A) = \mathrm{Lang}(A_1) \cap \mathrm{Lang}(A_2)$ by a construction that is somewhat related to the powerset construction. The idea is that $A = (Q, \Sigma_1, \delta, r_0, F)$ simulates A_1 and A_2 in parallel on the given word w. If and only if the computation on w ends in both an accepting state of A_1 and A_2, then A accepts w. The idea is that $Q = Q_1 \times Q_2$, i.e., each state r_{ij} with $0 \leq i \leq m-1$ and $0 \leq j \leq n-1$ corresponds to the two states q_i and p_j. For every letter $l \in \Sigma_1$, we add a transition from r_{ij} to $r_{i'j'}$ labeled l if and only if there is such a transition from q_i to $q_{i'}$ in A_1 and from p_j to $p_{j'}$ in A_2, i.e.,

$$\delta(r_{ij}, l) = r_{i'j'} \iff \delta_1(q_i, l) = q_{i'} \text{ and } \delta_1(p_j, l) = p_{j'} \,.$$

We define the start state of A to be $r_0 = r_{0,0}$, i.e., the state that corresponds to q_0 and p_0. The set of accepting states F of A contains all states r_{ij} such that q_i is accepting for A_1 and p_j is accepting for A_2, i.e.,

$$r_{ij} \in F \iff q_i \in F_1 \text{ and } p_j \in F_2 \,.$$

Now suppose $w \in L_1 \cap L_2$; then both A_1 and A_2 accept w, which implies that w ends in some state $q_i \in F_1$ for A_1 and in some state $p_j \in F_2$ for A_2. In this case, w ends in $r_{ij} \in F$ by construction. Conversely, if A_1 or A_2 does not accept w, then A neither accepts w. □

Let us consider an example in order to apply the construction used in the proof of Lemma 1.18. To this end, consider the two languages L_1 and L_2 that are given by the two regular expressions

$$R_1 = (b^*ab)^*aa(a+b)^* \quad \text{and} \quad R_2 = (a^*ba)^*bb \,.$$

Two DFAs A_1 with $\mathrm{Lang}(A_1) = L_1$ and A_2 with $\mathrm{Lang}(A_2) = L_2$ and with three and four states, respectively, are shown in Figure 9, together with the resulting product automaton A. Note that A contains two unreachable states, namely r_{11} and r_{12}, which can be deleted. Moreover, the states r_{02}, r_{03}, r_{12}, and r_{23} can be merged into a single nonaccepting state with a loop.

We observe that, by applying the same construction as in the proof of Lemma 1.18, we can give an alternative proof of Lemma 1.17. The only difference is that a state r_{ij} is contained in F if $q_i \in F_1$ or $p_j \in F_2$, but not necessarily both conditions have to be true.

Lemma 1.19. *If two languages L_1 and L_2 are regular, then $L_1 \setminus L_2$ is a regular language.*

Proof. The claim immediately follows from observing that $L_1 \setminus L_2 = L_1 \cap \overline{L}_2$ together with Lemmata 1.13 and 1.18. □

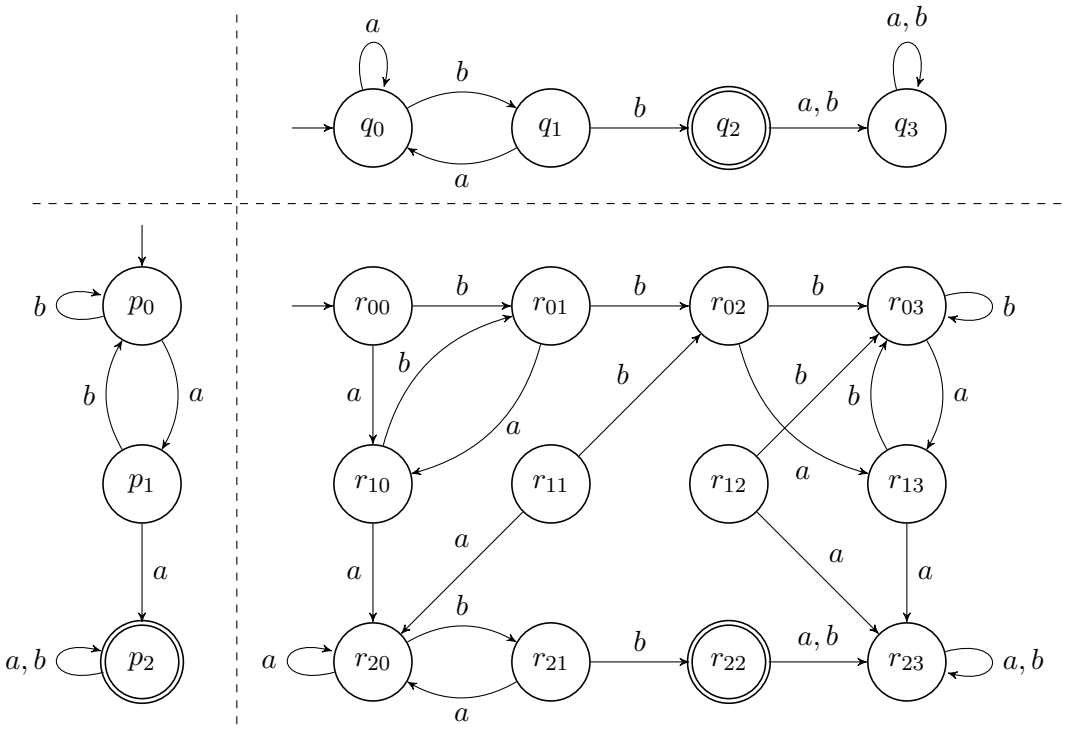


Figure 9.

1.7 The Pumping Lemma for Regular Languages

We already indicated that DFAs (NFAs, regular expressions, respectively) are rather limited when it comes to decision problems. In other words, there are many decision problems (i.e., languages) for which we cannot design DFAs; these languages are therefore not regular. Intuitively, the reason is that DFAs only have a finite memory.

How do we formally prove that a language is not regular? We first identify a property that is true for every regular language. After that, we can show a given language to be nonregular by proving that this property is not true. Let us start with an example. Consider the DFA A shown in Figure 10a and suppose it reads the word $w = cacbbc$, which is accepted. Reading w induces a tour

$$(q_0, q_2, q_3, q_4, q_2, q_1, q_5)$$

through A, and we observe that the state q_2 appears twice in this tour; see Figure 10b. We can decompose w into three parts; the first part x corresponds to all letters that are read until q_2 appears for the first time, i.e., $x = c$; the second part y continues after x until q_2 is encountered for the second time, i.e., $y = acb$; the last part z makes up the remainder of w, i.e., $z = bc$. Note that, starting in q_0, the word x results in A being in state q_2; after that, y induces a loop, and A is again in q_2 after y is read; finally, starting at q_2, z leads to the accepting state q_5.

Now consider the word $xyyz$; again, here the x part leads to A being in q_2; then a loop is made by reading the part y, and A is again in state q_2; next, another loop is made by reading y once more; since A is once again in q_2, reading z now results in A being in q_5,

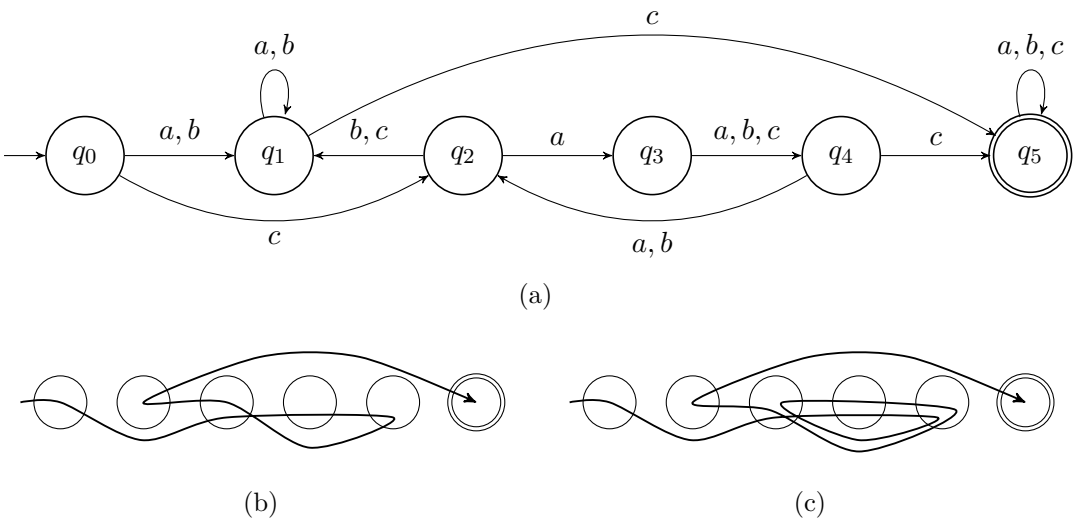


Figure 10.

and thus this word is accepted, too. The tour induced by reading $xyyz$ is

$$(q_0, q_2, q_3, q_4, q_2, q_3, q_4, q_2, q_1, q_5) \ ,$$

which is shown in Figure 10c, and we see that a similar reasoning also works for $xyyyz$, $xyyyyz$, and so on. This idea is formalized by the following lemma.

Lemma 1.20 (Pumping Lemma for Regular Languages). *Let L be a regular language. Then there is a constant n_0 such that, for every word $w \in L$ with $|w| \geq n_0$, there is a decomposition $w = xyz$ such that*

1. $|xy| \leq n_0$,
2. $|y| \geq 1$, *and*
3. $xy^\ell z \in L$ *for every* $\ell \in \mathbb{N}$.

Proof. Since L is regular, there is a DFA $A = (Q, \Sigma, q_0, \delta, F)$ with $\mathrm{Lang}(A) = L$ by definition. The set Q of A's states is finite; let n_0 denote its size, i.e., $n_0 = |Q|$. Now consider any word $w \in L$ with $|w| \geq n_0$. Since A is a DFA for L, when reading w we end up in an accepting state. More specifically, this induces a tour

$$\overline{p} = (p_0, p_1, \ldots, p_{|w|})$$

through A with $p_0 = q_0$ and $p_{|w|} \in F$. We observe that, since $\overline{p}$ visits $|w| + 1 \geq |n_0| + 1$ states, at least two states p_i and p_j with $i \neq j$ have to be the same, i.e., $p_i = p_j = q$ for some state $q \in Q$; assume that q is the first such state. It follows that we can write $\overline{p}$ as

$$\overline{p} = (\underbrace{p_0, p_1, \ldots, p_{i-1}, q}_{x}, \underbrace{p_{i+1}, \ldots, p_{j-1}, q}_{y}, \underbrace{p_{j+1}, \ldots, p_{|w|}}_{z}) \ ,$$

where x, y, and z are defined to be the parts of w that correspond to the indicated parts of $\overline{p}$.

- We first observe that Property 1 follows from the fact that the prefix of xy of w corresponds to the first repetition of the state q, which has to happen within the first n_0 letters of w.
- Moreover, the middle part y of w cannot be empty, which is formalized by Property 2.
- Lastly, we observe that, starting in the state q, reading the word z will always result in ending up in the accepting state $p_{|w|}$. Therefore, the tour

$$(\underbrace{p_0, p_1, \dots, p_{i-1}, q}_{x}, \underbrace{p_{i+1}, \dots, p_{j-1}, q}_{y}, \underbrace{p_{i+1}, \dots, p_{j-1}, q}_{y}, \underbrace{p_{j+1}, \dots, p_{|w|}}_{z}) \ ,$$

  which corresponds to the word $xyyz$, and the word

$$(\underbrace{p_0, p_1, \dots, p_{i-1}, q}_{x}, \underbrace{p_{j+1}, \dots, p_{|w|}}_{z}) \ ,$$

  which corresponds to xz, are also contained in L. The same is true for any $xy^\ell z$ with $\ell \in \mathbb{N}$, which shows Property 3.

In words, the part y of the word can be "pumped" (also zero times). The idea is that a DFA does not recall the way it took to end up in a certain state. □

Now let us give an example of how to prove the nonregularity of a given language. To this end, consider the language

$$L_{ab} = \{w \in \{a, b\}^* \mid w = a^k b^k \text{ with } k \in \mathbb{N}\} \ ,$$

which contains all words that consist of a sequence of as followed by the same number of bs, i.e.,

$$L_{ab} = \{\varepsilon, ab, aabb, aaabbb, \dots\} \ .$$

We now use the pumping lemma to prove that this language is not regular.

Theorem 1.21. *L_{ab} is not regular.*

Proof. For a contradiction, assume that L_{ab} were regular. Let n_0 be the constant from the pumping lemma for regular languages (i.e., Lemma 1.20). Consider the word $w = a^{n_0} b^{n_0} \in L_{ab}$. Since $|w| \geq n_0$, there has to be a decomposition of $w = xyz$ that fulfills the three conditions stated by the lemma. First, we know that $|xy| \leq n_0$ due to Property 1, and therefore y consists only of as. Thus, xy^2z, which must be in L_{ab} if L_{ab} is regular, consists of more than n_0 as (since $|y| \geq 1$ due to Property 2), followed by exactly n_0 bs. However, this word is not in L_{ab}, which contradicts Property 3 and therefore the assumption that L_{ab} is regular. □

This concludes our studies of regular languages as a first approach to model easy decision problems. In the next section, we will learn about more powerful models that accept more general languages than DFAs.

1.8 Historical and Bibliographical Notes

There is a rich literature on automata theory and regular languages. This chapter is based on the two textbooks by Hopcroft et al. [11] and Hromkovič [12]. DFAs, as we defined them here, were introduced after Turing machines in the mid-1950s [13, 20, 21]. Both NFAs and the powerset construction (sometimes also referred to as the subset construction) were proposed by Rabin and Scott in 1959 [25].

Regular expressions are nowadays standard tools to parse, e.g., user input; they date back to 1956 and in particular to Kleene [16]. They are found in text editors, such as `vim`, programming languages, such as `perl`, or text processing tools, such as `grep`. The algorithm applied in the proof of Theorem 1.11 is due to Kleene's original publication [16] and therefore called Kleene's algorithm. The algorithm applied in the proof of Theorem 1.10 is due to McNaughton and Yamada [19].

Many of the closure properties of regular languages we described were also already observed by Kleene [16].

Note that the pumping lemma only states an implication. Indeed, there are nonregular languages for which the properties of the pumping lemma hold, e.g., the language $\{w \in \{a, b\}^* \mid |w|_a \neq |w|_b\}$. The pumping lemma, as formulated in Lemma 1.20, is actually unnecessarily restrictive; we can reformulate it to speak about any word $w \in \Sigma^*$ instead of any word $w \in L$ with $|w| \geq n_0$, and then draw the conclusion that if $w \in L$, all pumped versions are in L as well, and if $w \notin L$, then neither are any of the pumped words [12]. The proof uses the exact same arguments as the original one, with the additional assumption that, if w does not end in an accepting state, then neither do any of the pumped words. With this, we can prove the above language to be nonregular. However, even with this generalization, the pumping lemma remains the statement of an implication, i.e., there are nonregular languages that satisfy the properties of this generalization as well. A statement of an equivalence is given by the Myhill-Nerode theorem, which was proven in 1958 [22].

2 Context-Free Languages

A particular language that we proved to be nonregular is L_{ab}, which consists of all words $a^k b^k$ with $k \in \mathbb{N}$. We have seen that the finiteness of DFAs is not sufficient to accept exactly the words in that language. In this chapter, we will learn about a class of languages to which also L_{ab} belongs; these languages are called **context-free** languages. It is possible to describe them with a kind of generalized ε-NFA, which we will do shortly. Before that, however, we will learn about another method to describe context-free languages, which at first glance does not seem to be related to any automata model.

2.1 Context-Free Grammars

A grammar is a system of **rewriting rules**. These rules describe how symbols can be substituted by other symbols in a specific way. Such substitutions can be repeated until only certain symbols remain. An example is given by the following rules.

- Initially, S is written down.
- Any S can be replaced by aSb; this can be iterated any number of times.
- Finally, any S can be deleted; a valid word does not contain S.

Now let us look at what kind of words can be generated in this way. We start by writing down an S, which we can then replace by aSb. Since there is again an S in the word, we again replace it by aSb, which leads to $aaSbb$. Finally, we delete S and obtain the word $aabb$. We see that by replacing S one more time by aSb before deleting S, we get the word $aaabbb$. Thinking about it, we see that these rules allow us to exactly obtain all words from L_{ab}. Let us formalize this concept.

Definition 2.1 (Context-Free Grammar, CFG). *A CFG G is a quadruple $G = (\Sigma_\mathrm{N}, \Sigma_\mathrm{T}, S, P)$, where*

- *Σ_N is an alphabet, called the **nonterminals** or **variables**,*
- *Σ_T is an alphabet, called the **terminals**,*
- *$S \in \Sigma_\mathrm{N}$ is the **start symbol**, and*
- *P is the set of **productions**. A production $p \in P$ is a pair $p = (A, \alpha)$ with $A \in \Sigma_\mathrm{N}$ and $\alpha \in (\Sigma_\mathrm{N} \cup \Sigma_\mathrm{T})^*$, which, for simplicity, is usually written as*

  $$A \to \alpha \ ,$$

  *and which states that the nonterminal A can be replaced by the word α. In this context, A is called the **head of p** and α is the **body of p**.*

Following Definition 2.1, we can formally define the CFG G_{ab} for the language L_{ab}, which we described above, as $G_{ab} = (\{S\}, \{a, b\}, S, P)$ with $P = \{S \to aSb, S \to \varepsilon\}$.

The language $\text{Lang}(G)$ of a CFG $G = (\Sigma_\text{N}, \Sigma_\text{T}, S, P)$ is the set of all terminal words that can be derived in G. To formalize this notion, we first need to define a **derivation step**, which is simply any application of a production rule. Suppose there is a production

$$A \to \gamma \in P \quad \text{with } A \in \Sigma_\text{N} \text{ and } \gamma \in (\Sigma_\text{N} \cup \Sigma_\text{T})^* \,.$$

Then a derivation step using this rule is written as

$$\alpha A \beta \Rightarrow \alpha \gamma \beta \quad \text{with } \alpha, \beta \in (\Sigma_\text{N} \cup \Sigma_\text{T})^* \,.$$

We can, e.g., derive the word a^3b^3 in G_{ab}, starting with S, by a sequence of derivation steps that each correspond to applying a rule from G_{ab}, which can be formalized by

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb \,.$$

Any sequence of derivation steps that starts with S and ends with a terminal word is called a **derivation**. Similar to the transition function δ of a DFA, which we extended to $\hat{\delta}$, we can extend the derivation relation $\Rightarrow$ to formalize any number of derivation steps instead of a single one; this relation is denoted by $\overset{*}{\Rightarrow}$ and hence $\alpha \overset{*}{\Rightarrow} \beta$ should be read as "β can be derived from α in some finite number (possibly zero) of derivation steps." Since $\text{Lang}(G)$ contains exactly those terminal words that can be derived in G, we get

$$\text{Lang}(G) = \{w \in \Sigma_\text{T}^* \mid S \overset{*}{\Rightarrow} w\} \,.$$

In the previous example, we have $\text{Lang}(G_{ab}) = L_{ab}$. As another example, consider the language $L_{ab,ba}$, which contains all words over $\{a, b, c\}$ that start with ab or end with ba (see Figure 5a). A CFG for this language is given by $G_{ab,ba} = (\{S, X, Y\}, \{a, b, c\}, S, P)$ with

$$\begin{aligned} P = \{&S \to abX \mid Y, \\ &X \to aX \mid bX \mid cX \mid \varepsilon, \\ &Y \to aY \mid bY \mid cY \mid ba\} \,, \end{aligned}$$

where we use $S \to \alpha \mid \beta$ as a shorthand for $S \to \alpha, S \to \beta$ for some $\alpha, \beta \in (\Sigma_\text{N} \cup \Sigma_\text{T})^*$. Using the above rules, indeed only words of the desired form can be generated. The start symbol S either generates the prefix ab, which can then be extended by any arbitrary word over $\{a, b, c\}$; or an arbitrary prefix, which can then only be extended to a word that ends with ba.

As another example, consider simple arithmetic expressions, which are defined iteratively as follows.

- a is an arithmetic expression (a is some terminal) and
- if x and y are arithmetic expressions, then $(x+y)$, $(x-y)$, $x \cdot y$, and x/y are arithmetic expressions.

An example for an arithmetic expression is $(a + a) \cdot (a \cdot (a + a \cdot a \cdot a) + a)$. A CFG for $G_\text{arith} = (\{S, A, B, C\}, \{(,), a, +, -, \cdot, /\}, S, P_\text{arith})$ contains the rules

$$P_\text{arith} = \{S \to A,$$

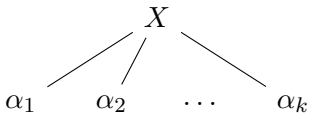
$$A \to (ABA) \mid ACA \mid a,$$
$$B \to + \mid -,$$
$$C \to \cdot \mid / \} \, .$$

Note that the nonterminals B and C could be left out by extending the rules with A as head. Consider the arithmetic expression $(a \cdot (a - a) + a)$. A derivation of this word in G_{arith} is given by

$$\begin{aligned} S &\Rightarrow A \Rightarrow (ABA) \Rightarrow (A + A) \Rightarrow (ACA + A) \Rightarrow (A \cdot A + A) \\ &\Rightarrow (A \cdot (ABA) + A) \Rightarrow (A \cdot (A - A) + A) \\ &\Rightarrow (a \cdot (A - A) + A) \Rightarrow (a \cdot (a - A) + A) \\ &\Rightarrow (a \cdot (a - a) + A) \Rightarrow (a \cdot (a - a) + a) \, , \end{aligned}$$

which is rather hard to follow. In order to visualize derivations of CFGs, we can make use of so-called **parse trees**, which give an easier way to describe derivations, and which have

- the start symbol S as root,
- terminals as leaves, and
- nonterminals as inner vertices such that the application of a rule $X \to \alpha_1 \alpha_2 \dots \alpha_k$ with $X \in \Sigma_{\text{N}}$ and $\alpha_1, \alpha_2, \dots, \alpha_k \in \Sigma_{\text{N}} \cup \Sigma_{\text{T}}$ is visualized by

  as part of the parse tree.

An example of a parse tree for the word $(a \cdot (a - a) + a)$ derived in the CFG G_{arith} is shown in Figure 11a. Observe that parse trees are not necessarily unique, i.e., a fixed word can induce different parse trees; see, e.g., the two trees for the word $a \cdot a \cdot a$ generated by G_{arith} shown in Figures 11b and 11c.

Similarly to the regular languages, which are characterized by the fact that there are DFAs that accept them, we can also characterize languages which are generated by CFGs.

Definition 2.2 (Context-Free Language). *A language L is called **context-free** if there is a CFG G with $\text{Lang}(G) = L$. The class of the context-free languages is*

$$\mathcal{L}_{\text{cf}} = \{L \mid L \text{ is context-free}\} \, .$$

2.2 Normalizing Context-Free Grammars

For the subsequent investigations, we need to transform given CFGs into a particular form, i.e., we only want rules of some certain kind. Since the head of rules of CFGs is always a

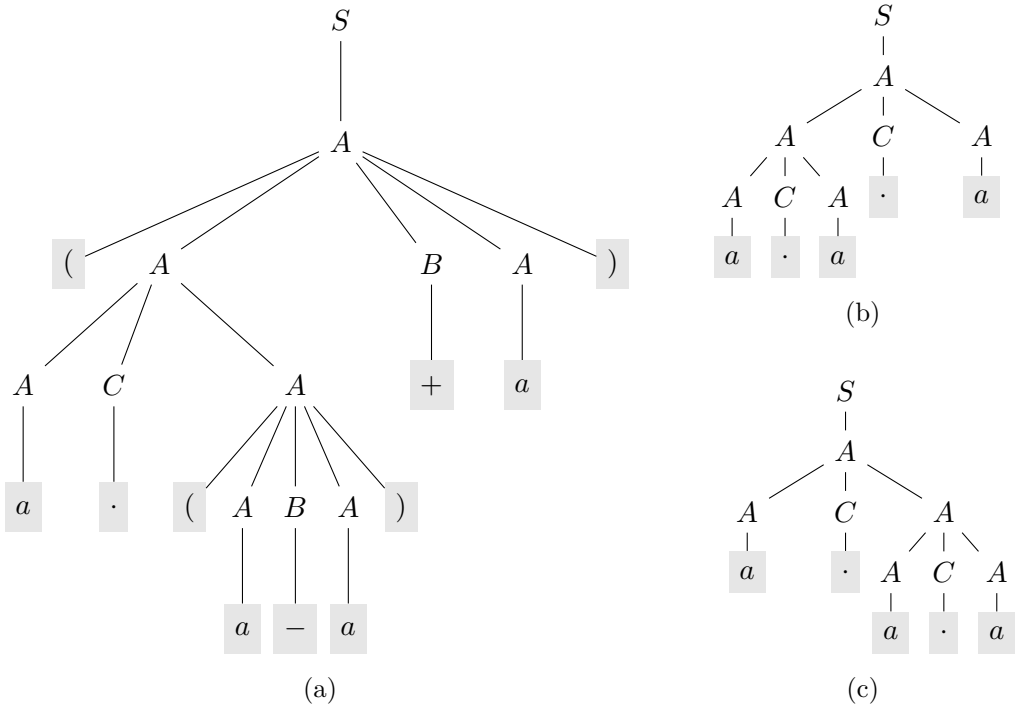


Figure 11.

single nonterminal, this basically refers to the bodies of the rules. In order to avoid making the following constructions unnecessarily complicated, we make the assumption that ε is never in the language of any of the given CFGs; this implies that there is no rule $S \to \varepsilon$.

We call a CFG **normalized** if its rules only have a specific form and there are no terminals or nonterminals that are never used in any derivation; more specifically,

- there are no **ε-productions**, i.e., rules of the form $A \to \varepsilon$ for any $A \in \Sigma_\mathrm{N}$,
- there are no **unit productions**, i.e., rules of the form $A \to B$ for any $A, B \in \Sigma_\mathrm{N}$, and
- there are no **useless symbols**, i.e., nonterminals or terminals which never appear in any derivation starting with S.

We will now describe three methods that, applied in the correct order, transform any given CFG into an equivalent one for which these three properties hold. In what follows, consider any CFG $G = (\Sigma_\mathrm{N}, \Sigma_\mathrm{T}, S, P)$ with $\varepsilon \notin \mathrm{Lang}(G)$.

2.2.1 Eliminating ε-Productions

We change the rules P such that all rules $A \to \varepsilon$ with $X \in \Sigma_\mathrm{N}$ are removed. This is done in two steps, which result in a new set P' of rules.

Step 1. Find all **nullable** nonterminals, i.e., variables $X \in \Sigma_\mathrm{N}$ such that $X \overset{*}{\Rightarrow} \varepsilon$. This is done in an iterative fashion.

Initialize. Set $\text{Null}_1 = \{X \in \Sigma_\text{N} \mid X \to \varepsilon\}$, i.e., the set of all nonterminals that appear as head of ε-productions.

Iterate. $\text{Null}_{i+1} = \text{Null}_i \cup \{X \in \Sigma_\text{N} \mid X \to \alpha \text{ and } \alpha \in \text{Null}_i^*\}$, and repeat this until $\text{Null}_{i+1} = \text{Null}_i$, i.e., until no new nonterminals are added.

As a result, we obtain $\text{Null}_i = \{X \in \Sigma_\text{N} \mid X \overset{*}{\Rightarrow} \varepsilon\}$. The idea behind this is that, in every iteration step, we add those variables which can be replaced, in one derivation step, by nonterminals from which we already know that they are nullable.

Step 2. We now construct the set P' of new rules. Consider some rule $X \to X_1X_2 \dots X_k \in P$ with $k \in \mathbb{N}^+$, and suppose that $m \leq k$ of the X_is are nullable. P' contains 2^m versions of this rule, where the nullable X_is, in all possible combinations, are present or absent. This simulates that the absent X_is are substituted by ε.

There is an important exception, which prevents us from creating another ε-production. If $k = m$ (i.e., all X_is are nullable), then we do not add the rule where all X_is are absent to the new set P' of rules. The head of such a production is nullable already. This works since $\varepsilon \notin \text{Lang}(G)$.

As a result, P' does not contain any rule of the form $X \to \varepsilon$ with $X \in \Sigma_\text{N}$.

As an example, consider the CFG $G = (\{S, A, B, C\}, \{a, b, c\}, S, P)$ with $P = \{S \to aAB, S \to C, A \to aB, A \to \varepsilon, B \to bA, B \to \varepsilon, C \to AB, C \to c\}$. We apply the first step in order to compute all nullable symbols. Due to the rules $A \to \varepsilon$ and $B \to \varepsilon$, we set $\text{Null}_1 = \{A, B\}$. Now we compute Null_2, i.e., we add all rules whose bodies are any word of A and B. The only rule that only contains As or Bs in its body is $C \to AB$, and thus we set $\text{Null}_2 = \text{Null}_1 \cup \{C\} = \{A, B, C\}$. After that, we compute Null_3. Since there are no rules (except those already considered) whose bodies are words consisting of A, B, and C only, we have $\text{Null}_3 = \text{Null}_2$, and the procedure terminates.

As a second step, we consider all rules with nonterminals from Null_2 as body, and construct all rules by removing any combination of these nonterminals. With this,

$$
\begin{aligned}
&S \to aAB && \text{becomes } S \to a, S \to aB, S \to aA, \text{ and } S \to aAB,\\
&A \to aB && \text{becomes } A \to a, \text{ and } A \to aB,\\
&B \to bA && \text{becomes } B \to b, \text{ and } B \to bA, \text{ and}\\
&C \to AB && \text{becomes } C \to A, C \to B, \text{ and } C \to AB.
\end{aligned}
$$

Finally, the new rules are

$$
\begin{aligned}
P' &= (P \setminus \{A \to \varepsilon, B \to \varepsilon\})\\
&\quad \cup \{S \to a, S \to aB, S \to aA, A \to a, B \to b, C \to A, C \to B\}\\
&= \{S \to aAB, S \to aA, S \to aB, S \to aAB,\\
&\quad A \to a, A \to aB, B \to b, B \to bA, C \to A, C \to B, C \to AB, C \to c\}\ ,
\end{aligned}
$$

and we set $G' = (\Sigma_\text{N}, \Sigma_\text{T}, S, P')$ as a CFG equivalent to G.

2.2.2 Eliminating Unit Productions

Next, we want to eliminate all unit productions, i.e., all rules of the form $X \to Y$ with $X, Y \in \Sigma_N$. A first idea is to expand the productions, e.g.,

$$A \to B, B \to a, \text{ and } B \to b \quad \text{is replaced by} A \to a \text{ and } A \to b\ .$$

However, this leads to problems if there are cycles such as the three rules $A \to B$, $B \to C$, and $C \to A$. If we encounter something like this, the expansion will not terminate. Instead, we find all **unit pairs** (X, Y) with $X, Y \in \Sigma_N$ such that $X \overset{*}{\Rightarrow} Y$ using unit productions only. Now, if there is a sequence of derivation steps

$$X \Rightarrow Y_1 \Rightarrow Y_2 \Rightarrow \ldots \Rightarrow Y_n \Rightarrow \alpha\ ,$$

where $Y_n \to \alpha$ is no unit production, then this can be replaced by $X \to \alpha$.

In order to replace unit productions systematically, there are again two steps involved.

Step 1. First, we find all unit pairs (X, Y) of G as above.

Initialize. (X, X) is a unit pair for every $X \in \Sigma_N$.

Iterate. If (X, Y) is a unit pair with $X, Y \in \Sigma_N$ and there is a rule $Y \to Z$ with $Z \in \Sigma_N$, then (X, Z) is a unit pair. Since there is a finite number of nonterminals, this procedure terminates in finite time. Specifically, if there is a cycle such that $X \overset{*}{\Rightarrow} X$ using unit productions, this will lead to a unit pair (X, X), which was already considered.

Step 2. We now again construct the set P' of new rules. To this end, for every unit pair (X, Y), we add to P' (new) productions $X \to \alpha$, where $Y \to \alpha$ is a non-unit production in P; this includes the case $X = Y$.

Afterwards, we delete all unit productions.

As an example, consider the CFG $G = (\{A, B, C, D\}, \{a, b, d, e\}, A, P)$ with $P = \{A \to aB, B \to C, B \to D, B \to b, C \to D, D \to de\}$. Following the first step, we first compute all unit pairs. We start with (A, A). Since there is no unit production with A as head, this is the only unit pair with A at the first position. Next, we add (B, B). Due to the unit productions $B \to C$ and $B \to D$, we add the two pairs (B, C) and (B, D). After that, (C, C) is added, and also (C, D) due to $C \to D$. Finally, we add (D, D) as the only pair with D at the first position.

Now we move to the second step. For every unit pair (X, Y), we add every rule $X \to \alpha$ for a non-unit production $Y \to \alpha$. The first unit pair is (A, A) and thus gives the rule $A \to aB$. Next, we add $B \to b$ due to the unit pair (B, B) and the rule $B \to b \in P$. Since there is no non-unit production with head C, the unit pair (B, C) does not lead to any rule. Due to (B, D) and the rule $D \to de \in P$, we add $B \to de$. C is no head of any non-unit production, and thus (C, C) does not give any new rule. The two unit pairs (C, D) and (D, D) and the rule $D \to de \in P$ finally yield two new rules $C \to de$ and $D \to de$.

We can summarize these two steps by a table as follows.

Unit pair	(A, A)	(B, B)	(B, C)	(B, D)	(C, C)	(C, D)	(D, D)
Production	$A \to aB$	$B \to b$		$B \to de$		$C \to de$	$D \to de$

We note that, due to the unit pairs (X, X) for all $X \in \Sigma_N$, all non-unit productions are preserved; we finally get a new set

$$P' = \{A \to aB, B \to b, B \to de, C \to de, D \to de\}$$

of rules of G', which is again equivalent to G.

2.2.3 Eliminating Useless Symbols

As already mentioned, useless symbols are nonterminals and terminals that never appear in any derivation. There are two different kinds of useless symbols. Let $X \in \Sigma_N \cup \Sigma_T$.

- X is **generating** if $X \overset{*}{\Rightarrow} w$ for $w \in \Sigma_T^*$; every terminal is generating since it generates itself in zero derivation steps.
- X is **reachable** if there is some derivation $S \overset{*}{\Rightarrow} \alpha X \beta$ with $\alpha, \beta \in (\Sigma_N \cup \Sigma_T)^*$.
- X is called **useless** if it is not reachable or not generating.

Obviously, if all useless symbols (and the corresponding rules) are removed, this does not change the language of the CFG. We again give a method that works in two steps.

Step 1. We start by computing the generating symbols of the given CFG, i.e., we remove the nongenerating symbols.

Initialize. Set $\text{Gen}_1 = \{x \mid x \in \Sigma_T\}$, the set of all terminals.

Iterate. $\text{Gen}_{i+1} = \text{Gen}_i \cup \{X \in \Sigma_N \mid \text{ there is } X \to \alpha \in P \text{ with } \alpha \in \text{Gen}_i^*\}$, and repeat this until $\text{Gen}_{i+1} = \text{Gen}_i$, i.e., until no new nonterminals are added.

The new CFG is $G' = (\text{Gen}_i \cap \Sigma_N, \Sigma_T, S, P')$ with

$$P' = \{X \to \alpha \in P \mid X \in \text{Gen}_i \cap \Sigma_N \text{ and } \alpha \in \text{Gen}_i^*\} .$$

Step 2. Next, we compute the reachable symbols of a CFG G' with a set P' of rules, i.e., we remove the nonreachable symbols.

Initialize. Set $\text{Reach}_{N,1} = \{S\}$ and $\text{Reach}_{T,1} = \emptyset$.

Iterate. For every rule $X \to \alpha \in P'$ with $X \in \text{Reach}_{N,i}$ and $\alpha \in (\Sigma_N \cup \Sigma_T)^*$, $\text{Reach}_{N,i+1} = \text{Reach}_{N,i} \cup \{Y \in \Sigma_N \mid Y \text{ appears in } \alpha\}$ and $\text{Reach}_{T,i+1} = \text{Reach}_{T,i} \cup \{a \in \Sigma_T \mid a \text{ appears in } \alpha\}$, and repeat this until $\text{Reach}_{N,i+1} = \text{Reach}_{N,i}$ and $\text{Reach}_{T,i+1} = \text{Reach}_{T,i}$.

The new CFG is $G'' = (\text{Reach}_{N,i}, \text{Reach}_{T,i}, S, P'')$ with

$$P'' = \{X \to \alpha \in P' \mid X \in \text{Reach}_{N,i}\} .$$

Again, let us consolidate our intuition by an example. To this end, consider the CFG $G = (\{S, A, B, C\}, \{a, b, c\}, S, P)$ with $P = \{S \to A, S \to AB, A \to Aa, A \to a, B \to bB, B \to Bb, C \to c\}$. First, we compute the generating symbols following the first step, and set $\mathrm{Gen}_1 = \{a, b, c\}$. Due to $A \to a$ and $C \to c$, we obtain $\mathrm{Gen}_2 = \{a, b, c, A, C\}$, and then, due to $S \to A$, we obtain $\mathrm{Gen}_3 = \{a, b, c, A, C, S\}$; then the procedure terminates. The new CFG is $G' = (\{S, A, C\}, \{a, b, c\}, S, P')$ with

$$P' = \{S \to A, A \to Aa, A \to a, C \to c\} .$$

Next, we compute the reachable symbols of G'. Let $\mathrm{Reach}_{\mathrm{N},1} = \{S\}$ and $\mathrm{Reach}_{\mathrm{T},1} = \emptyset$. Due to $S \to A$, we have $\mathrm{Reach}_{\mathrm{N},2} = \{S, A\}$, but still $\mathrm{Reach}_{\mathrm{T},2} = \emptyset$, and due to $A \to a$, we have $\mathrm{Reach}_{\mathrm{T},3} = \{a\}$ while $\mathrm{Reach}_{\mathrm{N},3} = \mathrm{Reach}_{\mathrm{N},2}$; then the procedure terminates. The final CFG is $G'' = (\{S, A\}, \{a\}, S, P'')$ with

$$P'' = \{S \to A, A \to Aa, A \to a\} .$$

The two steps must be applied in the given order, i.e., removing nonreachable symbols first and then nongenerating ones might lead to a CFG that still contains useless symbols. Consider the CFG $G = (\{S, A, B\}, \{a\}, S, P)$ with $P = \{S \to AB, S \to a, A \to a\}$. Since all symbols are reachable, removing nonreachable symbols does not change the CFG, and hence we get $G' = G$. Removing the nongenerating symbols means to remove B and the rule $S \to AB$, which gives the CFG $G'' = (\{S, A\}, \{a\}, S, P'')$ with $P'' = \{S \to a, A \to a\}$, which contains a nonreachable symbol A.

2.2.4 The Correct Order of Applying the Transformations

It is crucial to apply the transformations in the given order given above, i.e., first remove the ε-productions, then remove unit productions, and finally remove any useless symbols. Diverging from this order may result in a CFG that is not normalized. This can, e.g., happen if unit productions are removed before ε-productions. Let us revisit the example above where we removed ε-productions. During this procedure, we added the rule $C \to A$ due to the original rule $C \to AA$ and the fact that A is nullable; we observe that the new CFG contains a new unit production.

2.2.5 The Chomsky Normal Form

The final modification we apply to a given CFG G restricts it to rules of a certain form, which makes sure that within a single derivation step a nonterminal is replaced by either two nonterminals or a single terminal.

Definition 2.3 (Chomsky Normal Form). *Let G be a CFG with $\varepsilon \notin \mathrm{Lang}(G)$. G is in Chomsky normal form, ChNF for short, if all rules are either of the form*

- *$X \to YZ$, for $X, Y, Z \in \Sigma_{\mathrm{N}}$, or*
- *$X \to l$, for $X \in \Sigma_{\mathrm{N}}$ and $l \in \Sigma_{\mathrm{T}}$.*

Suppose G is normalized according to Sections 2.2.1 to 2.2.4. Consider any rule p from G. Either p has the form $X \to l$ for $X \in \Sigma_N$ and $l \in \Sigma_T$, which is already fine, or $X \to \alpha$ with $|\alpha| \geq 2$ (since there are no unit productions, α cannot be a single nonterminal).

What remains to be done is

- arrange that bodies of length 2 or more consist of nonterminals only and
- break bodies of length 3 or more into a set of rules each of which has a body of exactly 2 nonterminals.

We again proceed in two steps.

Step 1. For every $l \in \Sigma_T$ that appears in a body of length 2 or more, create a new $X_l \in \Sigma_N$ and add the rule $X_l \to l$.

In each of the bodies of length 2 or more, replace l by X_l.

Step 2. Due to the first step, every rule of length 2 or more now consists of nonterminals only. All rules with a body of length 2 are again already fine.

So consider any rule $X \to Y_1 Y_2 \dots Y_k$ with $k \geq 3$, where $X, Y_1, Y_2, \dots, Y_k \in \Sigma_N$. We now introduce new nonterminals $Z_1, Z_2, \dots, Z_{k-2}$ and replace the above rule by

$$X \to Y_1 Z_1,\ Z_1 \to Y_2 Z_2,\ Z_2 \to Y_3 Z_3, \dots,\ Z_{k-3} \to Y_{k-2} Z_{k-2},\ Z_{k-2} \to Y_{k-1} Y_k\ .$$

As an example, consider the CFG $G = (\{S, A, B, C, D\}, \{a, b, c, d\}, S, P)$ with $P = \{S \to Aa, A \to BCD, B \to cd, C \to bc, D \to d\}$. We transform G by applying the first step, leading to a new CFG $G' = (\{S, A, B, C, D, X_a, X_b, X_c, X_d\}, \{a, b, c, d\}, S, P')$ with

$$\begin{aligned} P' = \{&S \to AX_a, \quad X_a \to a, \\ &A \to BCD, \\ &B \to X_c X_d, \quad X_c \to c, X_d \to d, \\ &C \to X_b X_c, \quad X_b \to b, \\ &D \to d\}\ . \end{aligned}$$

Applying the second step, we obtain a CFG G'' that is equal to G' except that it contains an additional nonterminal Z and that the two rules $A \to BZ$ and $Z \to CD$ replace the rule $A \to BCD$.

2.3 The CYK Algorithm

Suppose we want to figure out whether a word w is in a given regular language L. In other words, we would like to solve the instance w of the decision problem L. In the previous chapter, we have seen a number of ways to achieve this; one possibility is to design a DFA A for L and then check whether A accepts w, i.e., whether $w \in \mathrm{Lang}(A)$. So far so good, but what happens if we consider context-free languages? We already know that such a language L is uniquely described by a given CFG G with $\mathrm{Lang}(G) = L$. But how to determine

whether $w \in \mathrm{Lang}(G)$? Indeed, for us it seems to be a lot harder to grasp what a CFG "is doing" than what a DFA "is doing." In order to determine whether $w \in \mathrm{Lang}(G)$ or not, we will use the **CYK algorithm**, which again uses a dynamic programming approach.

Without loss of generality, let G be a CFG in ChNF with $\varepsilon \notin \mathrm{Lang}(G)$, and let $w = a_1 a_2 \dots a_n$ be a word for which we want to determine whether $w \in \mathrm{Lang}(G)$ or not. The idea of the CYK algorithm is to compute, for every subword $a_i a_{i+1} \dots a_j$, the set $N_{i,j}$ of all nonterminals $X \in \Sigma_{\mathrm{N}}$ with

$$X \overset{*}{\Rightarrow} a_i a_{i+1} \dots a_j \ .$$

As a consequence, $w \in \mathrm{Lang}(G)$ if and only if $S \in N_{1,n}$. We start with shortest subwords (i.e., single letters), and consider larger and larger subwords, such that we can use intermediate solutions for smaller subwords to compute solutions for larger ones.

Consider a word of length 5, say $w = a_1 a_2 a_3 a_4 a_5 \in \mathrm{Lang}(G)$. Since G is in ChNF, a valid derivation of w has a form like

$$\begin{aligned} S &\Rightarrow X_1 X_2 \Rightarrow X_{11} X_{12} X_2 \Rightarrow X_{111} X_{112} X_{12} X_2 \Rightarrow X_{111} X_{112} X_{12} X_{21} X_{22} \\ &\overset{*}{\Rightarrow} a_1 a_2 a_3 a_4 a_5 \ . \end{aligned}$$

On a high level, the idea of the CYK algorithm is to do this backwards. We take the word $a_1 a_2 a_3 a_4 a_5$ and, as a first step, try all possibilities of the last couple of derivations that replaced single nonterminals by terminals. One of the possibilities is the nonterminal string $X_{111} X_{112} X_{12} X_{21} X_{22}$ right before the terminating rules $X_{111} \to a_1$, $X_{112} \to a_2$, $X_{12} \to a_3$, $X_{21} \to a_4$, and $X_{22} \to a_5$ are applied to derive the word w. Then, we consider all possibilities how this string $X_{111} X_{112} X_{12} X_{21} X_{22}$ may have been derived; one such possibility is that $X_{111} X_{112} X_{12} X_2$ was changed to the former string by applying the rule $X_2 \to X_{21} X_{22}$ to the last nonterminal, which is what actually happened in our sample derivation. We then continue in this fashion until a single nonterminal is left. If this is start terminal S, we know that $a_1 a_2 a_3 a_4 a_5$ can be derived in G. However, it is crucial that we consider all possibilities.

As a more concrete example, consider the CFG in ChNF $G = (\{S, A, B, C\}, \{a, b\}, S, P)$ with

$$\begin{aligned} P = \{ & S \to AB \mid BC, \\ & A \to BA \mid a, \\ & B \to CC \mid b, \\ & C \to AB \mid a \} \ , \end{aligned}$$

and suppose we want to decide whether $baaba \in \mathrm{Lang}(G)$. As described above, we first consider all nonterminals from which the single letters of this word can be derived; e.g., both A and C can be replaced by the letter a, while b can only be derived from B; thus $N_{1,1} = N_{4,4} = \{B\}$ and $N_{2,2} = N_{3,3} = N_{5,5} = \{A, C\}$. We arrange the possibilities in a table as follows.

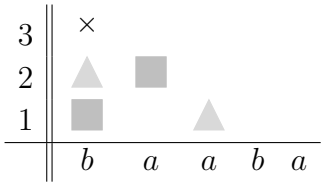
1	$\{B\}$	$\{A,C\}$	$\{A,C\}$	$\{B\}$	$\{A,C\}$
	b	a	a	b	a

One of the possibilities is that *baaba* was derived from $BAABC$. If this were the case, the derivation step before that may have been $BACC$, and the rule $C \to AB$ was applied to the first C. Another possibility is that the rule $A \to BA$ was applied to the first A of the nonterminal word $AABC$. We take all possibilities into account by extending the table as follows.

2	$\{S, A\}$	$\{B\}$	$\{S, C\}$	$\{S, A\}$	
1	$\{B\}$	$\{A, C\}$	$\{A, C\}$	$\{B\}$	$\{A, C\}$
	b	a	a	b	a

The row marked 2 contains all nonterminals that derive all subwords of length two that are composed of nonterminals in the row marked 1 in the same column and the column to the right in the given order. Consider, e.g., the first two columns in row 1. The sets are $\{B\}$ and $\{A, C\}$; this means that the prefix *ba* of the word was derived from either BA or BC. In the first column of row 2, we put the set with all nonterminals X that are head of a rule $X \to BA$ or $X \to BC$. Thus, due to $S \to BC$ and $A \to BA$, we add S and A. The second and third column of row 1 give the pairs AA, AC, CA, CC, and hence we add B due to the rule $B \to CC$. We continue in this fashion until row 2 is completed; note that it contains one column fewer than row 1. We see that row 2 contains nonterminals that allow the derivation of all subwords of length 2 of *baaba*. For instance, in the first column, the two nonterminals S and A both derive the prefix *ba* (i.e., the subword of length 2 starting at the first column) via $S \Rightarrow BC \overset{*}{\Rightarrow} ba$ or $A \Rightarrow BA \overset{*}{\Rightarrow} ba$; likewise, the two nonterminals S and C in the third column derive the subword *ab* of length 2 starting at the third column via $S \Rightarrow AB \overset{*}{\Rightarrow} ab$ or $C \Rightarrow AB \overset{*}{\Rightarrow} ab$.

Next, we fill row 3 with all nonterminals that derive subwords of *baaba* of length 3. Here, we have to be more careful than for subwords of length 2. Consider the first column of row 3. Thus, we look at the following positions



and combine each letter represented by the two ▲ or by the two ■. In our example, the ■ are both $\{B\}$; thus we search for all rules with body BB and find none. Next, we consider the ▲; the first one (in the first column) is $\{S, A\}$ and the second one (the third column) is $\{A, C\}$, which gives four bodies SA, SC, AA, and AC. Also here, we do not find any corresponding rules. Therefore, we write the empty set in the first column of row 3. Continuing in this fashion, we extend the table to

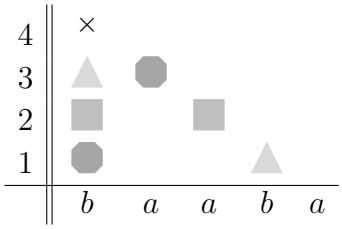
3	$\emptyset$	$\{B\}$	$\{B\}$		
2	$\{S, A\}$	$\{B\}$	$\{S, C\}$	$\{S, A\}$	
1	$\{B\}$	$\{A, C\}$	$\{A, C\}$	$\{B\}$	$\{A, C\}$
	b	a	a	b	a

and now need to continue with row 4, i.e., searching for all nonterminals that allow to derive subwords of length 4. Since the complete word has length 5, there are two of those

words, namely the prefix *baab* and the suffix *aaba*. Let us consider the first column that represents the subword *baab*. From any such nonterminal, the next derivation step leads to two nonterminals X_1X_2 such that

- X_1 yields the word b of length 1 and X_2 yields the word aab of length 3, or
- X_1 yields the word ba of length 2 and X_2 yields the word ab of length 2, or
- X_1 yields the word baa of length 3 and X_2 yields the word b of length 1.

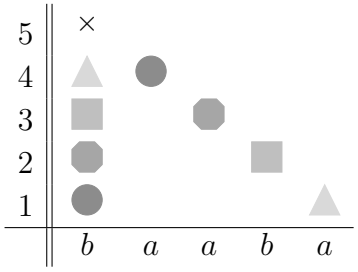
All those nonterminals have already been computed in the previous steps, and thus we fill out this cell by considering the cells



of the table. Applying the same scheme to the second column of row 4 leads to

4	$\emptyset$	$\{S,A,C\}$			
3	$\emptyset$	$\{B\}$	$\{B\}$		
2	$\{S,A\}$	$\{B\}$	$\{S,C\}$	$\{S,A\}$	
1	$\{B\}$	$\{A,C\}$	$\{A,C\}$	$\{B\}$	$\{A,C\}$
	b	a	a	b	a

as intermediate table. Finally, we fill out row 5 with those nonterminals that derive the only subword of length 5, i.e., the word itself. To this end, we again have to take into account all possible subwords that can be derived from the nonterminals X_1X_2 as above. Thus we consider the cells



and obtain

5	$\{S,A,C\}$				
4	$\emptyset$	$\{S,A,C\}$			
3	$\emptyset$	$\{B\}$	$\{B\}$		
2	$\{S,A\}$	$\{B\}$	$\{S,C\}$	$\{S,A\}$	
1	$\{B\}$	$\{A,C\}$	$\{A,C\}$	$\{B\}$	$\{A,C\}$
	b	a	a	b	a

as the complete table. Since the starting symbol S appears in row 5, we know that there is a derivation of *baaba* in G. In the last table, we marked such a derivation

$$S \Rightarrow AB \Rightarrow BAB \Rightarrow BACC \Rightarrow BAABC \overset{*}{\Rightarrow} baaba\ .$$

Note that the gray boxes and connections between them correspond to the parse tree induced by this derivation.

The above illustration can be easily generalized to find out whether an arbitrary word w is in the language of a given CFG in ChNF as follows.

- Start with finding all nonterminals that derive all subwords of length 1 (i.e., the letters) of w.
- Find all nonterminals Y that derive all subwords of length ℓ by considering all possible derivation steps $Y \to X_1X_2$, i.e., all possible combinations of deriving subwords w_1 and w_2 from X_1 and X_2; these are all subwords such that w_1w_2 is the considered subword of length ℓ. Since $1 \leq |w_1|, |w_2| \leq \ell - 1$, all nonterminals of interest have already been computed in previous steps.
- Repeat this until all nonterminals are computed that derive the single subword of length $|w|$, i.e., w itself. If S is among those nonterminals, w is in the language of the CFG; otherwise, it is not.

With the CYK algorithm, we now finally have a means to solve decision problems that are given by context-free languages.

2.4 The Pumping Lemma for Context-Free Languages

While the class of context-free languages is larger than that of regular languages, they still have significant limitations, i.e., we can define rather simple languages that are not generated by any CFG. In order to prove that, for a given language L, there is no CFG G with $\mathrm{Lang}(G) = L$, we proceed similar to proving that a language is not regular. In other words, we deduce a property that is true for all context-free languages similar to the pumping lemma for regular languages. In fact, we will also perform a sort of pumping. However, the ability to pump up words from the given language will now not be the consequence of the existence of a DFA, but of a CFG.

Let us start with an example. Consider the CFG $G = (\{S, A, B, C\}, \{a, b, c\}, S, P)$ in ChNF with $P = \{S \to AB, B \to CC, C \to BA, A \to a, B \to b, C \to c\}$ together with the derivation

$$S \Rightarrow AB \Rightarrow ACC \Rightarrow ABAC \Rightarrow aBAC \Rightarrow abAC \Rightarrow abaC \Rightarrow abac$$

of the four-letter word $abac \in \mathrm{Lang}(G)$. The last four steps generate the terminal symbols, while the first three steps increase the sequence of nonterminals; the corresponding parse tree T_{abac} is shown in Figure 12a. There is a path (S, B, C, B, b) in T_{abac} of length 4 from the root S to the leaf b. On this path, the nonterminal B appears twice; the first time, it is the root of the subtree T_{bac} which derives the subword *bac*, and the second time it is the

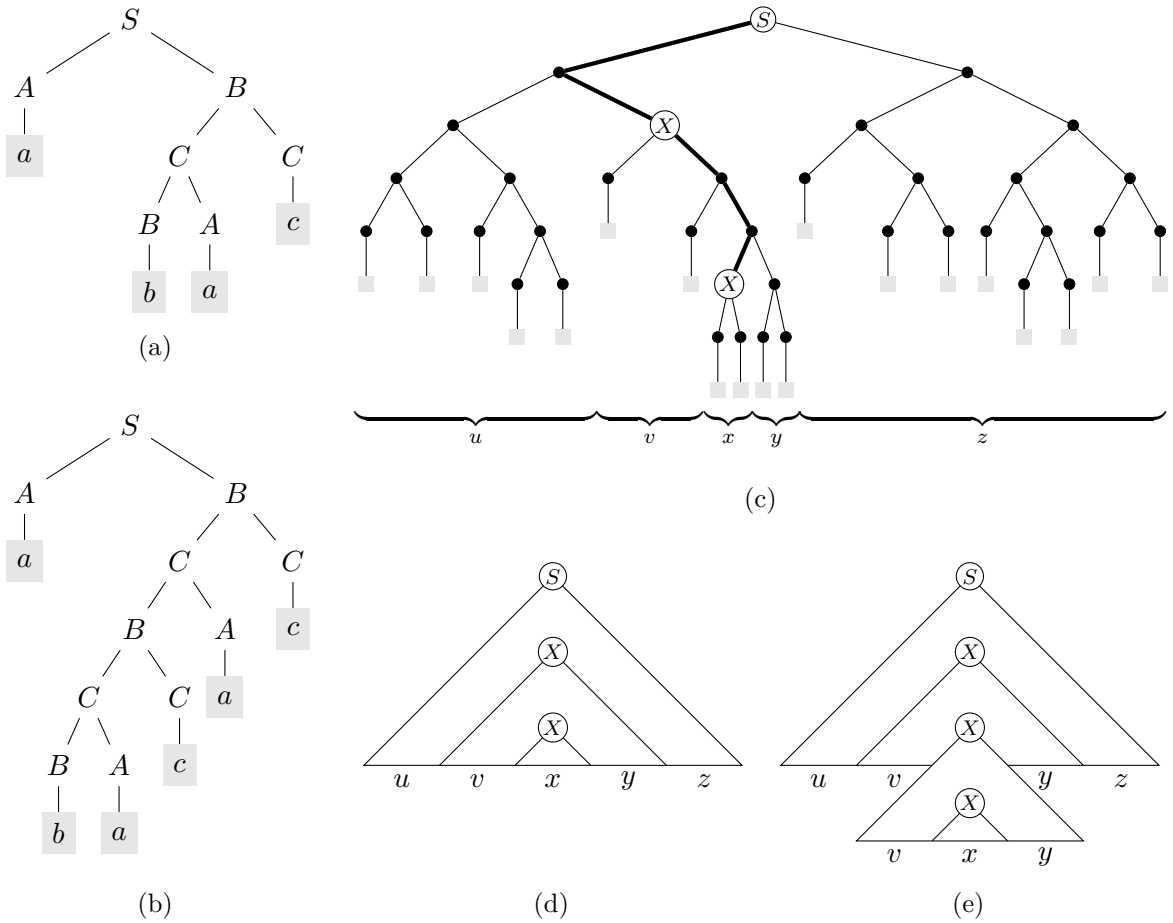


Figure 12.

root of T_b which derives the single letter b. Since both these trees have the same root, we can simply exchange the subtree T_b by the subtree T_{bac}, which leads to a new parse tree T_{abacac}, which is shown in Figure 12b. In doing this, we have shown that the corresponding word *abacac* is also in the language of G.

Basically, we again applied a kind of "pumping" in order to derive another word in the language. It is therefore not surprising that generalizing this idea allows us to prove the following lemma. Here, we pick a word from the given language that ensures that there is a repetition of a nonterminal in the corresponding parse tree on a path from the root to some leaf.

Lemma 2.4 (Pumping Lemma for Context-Free Languages). *Let L be a context-free language. Then there is a constant n_0 such that, for every word $w \in L$ with $|w| \geq n_0$, there is a decomposition $w = uvxyz$ such that*

1. $|vxy| \leq n_0$,
2. $|vy| \geq 1$, *and*
3. $uv^\ell xy^\ell z \in L$ *for every* $\ell \in \mathbb{N}$.

Proof. Let $G = (\Sigma_\mathrm{N}, \Sigma_\mathrm{T}, S, P)$ be a CFG in ChNF with $|\Sigma_\mathrm{N}| = m$; let $n_0 = 2^m$. Let $w \in L$ have at least n_0 letters and consider a derivation of w in G together with the resulting parse tree T. Obviously, T has at least n_0 leaves, each one labeled with a single terminal. Every leaf has one parent, while every inner vertex has two children, except those that are parents of leaves.

For a contradiction, assume that T has height less than $m+1$, i.e., there is no path of length $m+1$ from the root to any leaf. This means that T has less than 2^m leaves (since T is binary except for the parents of leaves). This immediately contradicts $|w| \geq 2^m$, because the leaves of T correspond to letters of w. Therefore, there is at least one path

$$\overline{p} = (X_1, X_2, \ldots, X_k, l)$$

of length $k \geq m+1$ (i.e., with at least $k+1 \geq m+2$ vertices) from the root $X_1 = S$ to some leaf l with $X_i \in \Sigma_\mathrm{N}$ and $l \in \Sigma_\mathrm{T}$. Since there are only m nonterminals in Σ_N in total, at least one nonterminal has to repeat at least once during $\overline{p}$. Let X be one such nonterminal and let i and j with $1 \leq i, j \leq k$ and $i < j$ be such that $X = X_i = X_j$; without loss of generality, let i and j be the largest integers with the property, i.e., X_i and X_j are the last two occurrences of X in $\overline{p}$ if there is more than one repetition.

An example of such a parse tree is shown in Figure 12c; Figure 12d gives a more high-level picture. Now we can decompose w according to this tree. Let x denote that part of w that is generated from X_j, i.e., which corresponds to the subtree of T rooted at the second appearance of X. Let v and y be the two substrings left and right of x that correspond to the subtree rooted at the first appearance X_i of X. Furthermore, let u be the maximum-length substring left of v, and let z be the maximum-length substring right of y. Now let us consider the three properties stated by the lemma.

- We note that vxy corresponds to the part of T in which X is repeated at least once; the root of this subtree is X_i and there is an inner vertex X_j. For some X, such a subtree has to exist, and we know from the reasoning above that this subtree has a height of at most $m+1$; thus, Property 1 follows.
- Since G is in ChNF (in particular, there is no unit production $X \to X$), it cannot be the case that both v and y are empty, which is stated by Property 2.
- Property 3 states that v and y can be "pumped" simultaneously for any number of times, which leads to words that are also contained in L. In particular, a parse tree for $uv^\ell xy^\ell z$ is obtained by ℓ times iteratively replacing the subtree rooted at X_j by the subtree rooted at X_i; this is possible since X_i and X_j are the same nonterminal and thus all corresponding rules can be used. A example for $\ell = 2$ is shown in Figure 12e. The case $\ell = 0$ then means that the subtree rooted at X_i is replaced by the subtree rooted at X_j.

The claim of the lemma follows. □

Now let us apply this new tool in order to prove that a given language is not context-free, i.e., is not generated by a CFG. To this end, consider

$$L_{abc} = \{a^k b^k c^k \mid k \in \mathbb{N}^+\} .$$

Theorem 2.5. *L_{abc} is not context-free.*

Proof. For a contradiction, assume that L_{abc} were context-free. Let n_0 be the constant from the pumping lemma for context-free languages (i.e., Lemma 2.4). Consider the word $w = a^{n_0}b^{n_0}c^{n_0} \in L_{abc}$. Since $|w| \geq n_0$, there has to be a decomposition of $w = uvxyz$ that fulfills the three conditions stated by the lemma. First, we know that $|vxy| \leq n_0$. Now we make a case distinction according to the concrete decomposition.

1. Suppose vxy only contains as. Due to Property 2, vy contains at least one a. Consequently, the word uv^2xy^2z cannot be in L_{abc} since it contains more as than both bs and cs. We can argue in an analogous way if vxy consists of bs only or of cs only.

2. Suppose vxy contains both as and bs. Then, again due to Property 2, uv^2xy^2z contains more as or more bs (or both) than cs. A similar argument can be applied in the case that vxy consists of bs and cs only.

3. Note that the case that vxy contains as, bs, and cs cannot occur due to Property 1 and $|w| = a^{n_0}b^{n_0}c^{n_0}$.

It follows that uv^2xy^2z is not in L_{abc}, which contradicts Property 3 and thus the assumption that L_{abc} is context-free. □

We note that applying the pumping lemma for context-free languages is more complicated than applying the lemma for regular languages. This is due to the fact that a more involved case distinction has to be made as there are more ways as to what the decomposition actually looks like.

2.5 The Chomsky Hierarchy

We have introduced the concept of grammars in a way that restricts the set of allowed rules. More specifically, so far we have only spoken about CFGs, which have the property that the head of every rule consists of a single nonterminal. In this section, we want to have a quick look at what happens when we relax this property or, conversely, make it even stricter.

Let us start with the latter, and suppose we only allow rules that contain a single nonterminal as head. Moreover, for the body of any rule, we now demand that it is either of the form wX or w with $X \in \Sigma_{\mathrm{N}}$ and $w \in \Sigma_{\mathrm{T}}^*$; we also allow the rule $X \to \varepsilon$. Now we claim that such grammars can mimic the work of DFAs. The idea behind this is that every derivation in such a grammar always has the form vX or v with $X \in \Sigma_{\mathrm{N}}$ and $v \in \Sigma_{\mathrm{T}}^*$. As an example, recall that DFAs can count modulo some number. Consider the language

$$L_{3\mathrm{m}4} = \{w \in \{0,1,2\}^* \mid (|w|_0 + 2|w|_1) \bmod 4 = 3\}$$

and let us design a grammar $G_{3\mathrm{m}4}$ for this language while obeying the above restrictions on the rules. To this end, we use four nonterminals Q_1, Q_2, Q_3, and Q_4. As explained above, every derivation in $G_{3\mathrm{m}4}$ contains at most one nonterminal, and this nonterminal is always

at the end of the intermediate word (i.e., the word "derived" so far), preceded by a word $x \in \{0,1,2\}^*$. The rules of our grammar are designed such that the invariant

if Q_i is the only nonterminal in the intermediate word, then $(|x|_0 + 2|x|_1) \bmod 4 = i$

is always true. It follows that Q_0 is the start symbol, since the initial derivation Q_0 is preceded by the empty word, and clearly $(|\varepsilon|_0 + 2|\varepsilon|_1) \bmod 4 = 0$. How do the productions with Q_0 as head look like? We add three of them, one for every terminal, such that the above invariant holds. This means that, if a 0 is generated, the number $(|x|_0 + 2|x|_1) \bmod 4$ increases by 1 and so does the index i of the nonterminal at the end of the intermediate word; if a 1 is generated, i has to increase by 2 (since 1s count double); and if a 2 is generated, i does not change at all. This yields the three rules

$$Q_0 \to 0Q_1, \quad Q_0 \to 1Q_2, \quad \text{and } Q_0 \to 2Q_0 \,.$$

We continue in this fashion with all remaining states; e.g., we set

$$Q_3 \to 0Q_0, \quad Q_3 \to 1Q_1, \quad \text{and } Q_3 \to 2Q_3$$

as the rules with Q_3 as head. So far, so good, but it remains to add a terminating rule, i.e., the possibility to get rid of the nonterminal at the end if $x \in L_{3\text{m}4}$. We observe that this is exactly the case if, following the invariant, this nonterminal is Q_3. Hence we add the rule $Q_3 \to \varepsilon$, and with this obtain the grammar $G_{3\text{m}4} = (\{Q_0, Q_1, Q_2, Q_3\}, \{0,1,2\}, Q_0, P)$ with

$$\begin{aligned} P = \{&Q_0 \to 0Q_1 \mid 1Q_2 \mid 2Q_0, \\ &Q_1 \to 0Q_2 \mid 1Q_3 \mid 2Q_1, \\ &Q_2 \to 0Q_3 \mid 1Q_0 \mid 2Q_2, \\ &Q_3 \to 0Q_0 \mid 1Q_1 \mid 2Q_3 \mid \varepsilon\} \,. \end{aligned}$$

We observe that every word over $\{0,1,2\}$ has a unique derivation in this grammar. Consider, e.g., $010021 \in L_{3\text{m}4}$ with the derivation

$$Q_0 \Rightarrow 0Q_1 \Rightarrow 01Q_3 \Rightarrow 010Q_0 \Rightarrow 0100Q_1 \Rightarrow 01002Q_1 \Rightarrow 010021Q_3 \Rightarrow 010021 \,,$$

while there is no terminating derivation for the word $01012 \notin L_{3\text{m}4}$ since

$$Q_0 \Rightarrow 0Q_1 \Rightarrow 01Q_3 \Rightarrow 010Q_0 \Rightarrow 0101Q_2 \Rightarrow 01012Q_2$$

still contains the nonterminal Q_2, which cannot be removed.

Now consider the DFA $A_{3\text{m}4}$ for $L_{3\text{m}4}$ as shown in Figure 13a. Every state q_i of $A_{3\text{m}4}$ corresponds to the nonterminal Q_i of $G_{3\text{m}4}$. In particular, for every rule $Q_i \to aQ_j$ in P, there is a transition $\delta(q_i, a) = q_j$ in $A_{3\text{m}4}$. Furthermore, q_0 is the start state and q_3 is the single accepting state. As a consequence, for any derivation $Q_0 \overset{*}{\Rightarrow} xQ_i$ in $G_{3\text{m}4}$, $A_{3\text{m}4}$ ends in state q_i when reading x. With this, it is probably not very surprising that we call grammars as such above **regular grammars** since they are indeed equivalent to DFAs. To prove this, the above construction can be generalized to prove the following theorem.

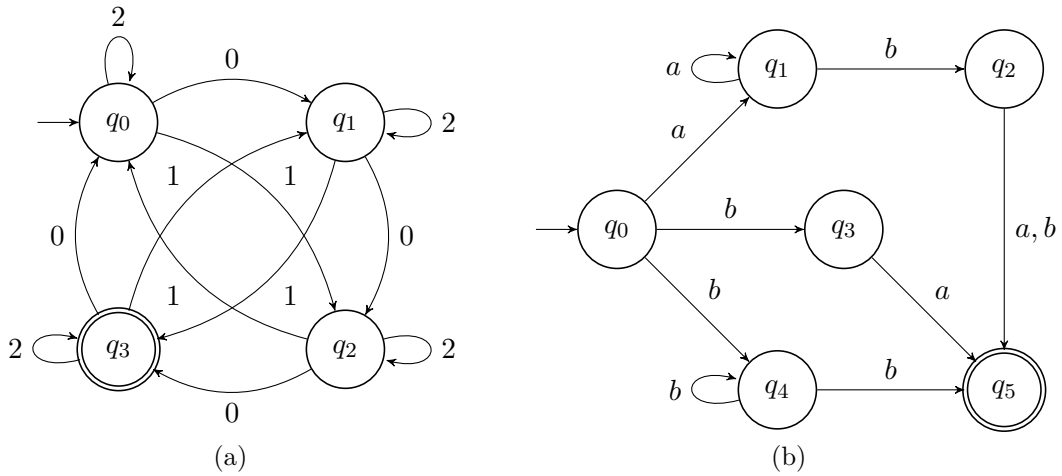


Figure 13.

Theorem 2.6. *Every DFA can be converted into an equivalent regular grammar.*

Proof. Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA with $\text{Lang}(A) = L$; without loss of generality, assume that $Q = \{q_0, q_1, \ldots, q_{m-1}\}$. We design a regular grammar $G = (\Sigma_\text{N}, \Sigma_\text{T}, Q_0, P)$ with $\Sigma_\text{T} = \Sigma$ as follows. For every state $q_i \in Q$, we add a nonterminal Q_i, and for every transition $\delta(q_i, a) = q_j$ with $a \in \Sigma$, we add a rule $Q_i \to aQ_j$. Finally, we add a rule $Q_i \to \varepsilon$ for every $q_i \in F$. Consider any word $w \in \text{Lang}(A)$ with $|w| = n$. Then w induces a run

$$(p_0, p_1, \ldots, p_n)$$

through A with $p_0 = q_0$ and $p_n \in F$. For every two consecutive states p_i and p_{i+1} with $0 \leq i \leq n-1$ of this run, A follows a transition $\delta(p_i, a) = p_{i+1}$, where a is the $(i+1)$th letter of w. Now let j and j' be such that $q_j = p_i$ and $q_{j'} = p_{i+1}$. Then the rule $Q_j \to aQ_{j'}$ is by definition contained in P, and since A is a DFA (which implies that there is exactly one transition for every state and letter), this is the only such rule. It follows that there is one unique way to generate the word $wQ_{j''}$ in G; since $q_{j''} = p_n \in F$ is accepting, the rule $Q_{j''} \to \varepsilon$ is also contained in P, which gives that w is generated by G. Conversely, if $w \notin \text{Lang}(A)$, then there is a unique derivation $wQ_{j'''}$ in G, where the nonterminal at the end cannot be removed; thus G does not generate w. □

In order to show that regular grammars cannot generate more words than DFAs, we prove the following theorem. To this end, we first have to normalize the given regular grammar G as described in Section 2.2. As before, suppose that $\varepsilon \notin \text{Lang}(G)$. Let us quickly discuss why the normalizing process yields a grammar that is also regular.

- Removing ε-productions as in Section 2.2.1 from a regular grammar leads to adding rules of the form $X \to w$ with $X \in \Sigma_\text{N}$ and $w \in \Sigma_\text{T}^*$; thus, the resulting grammar is still regular.
- Removing unit productions as in Section 2.2.2 only adds rules with bodies that are regular due to G being regular.

- Removing unnecessary symbols from G as in Section 2.2.3 only removes rules and therefore also ensures that the resulting grammar is regular.

As a result, we get a regular grammar that has rules of the form $X \to w$ and $X \to wY$ with $X, Y \in \Sigma_{\mathrm{N}}$ and $w \in \Sigma_{\mathrm{T}}^{+}$. Following an approach similar to the one used in Section 2.2.5, we take one final normalization step. To this end, consider any rule as above with $w = w_1 w_2 \dots w_k$ with $w_i \in \Sigma_{\mathrm{T}}$ for some $k \geq 2$. We introduce new nonterminals $Z_1, Z_2, \dots, Z_{k-2}$ and replace the rule above by

$$X \to w_1 Z_1,\ Z_1 \to w_2 Z_2,\ Z_2 \to w_3 Z_3, \dots,\ Z_{k-1} \to w_k \quad (Z_{k-1} \to w_k Y,\ \text{respectively}) ;$$

e.g., a rule $A \to aabaB$ is transformed into a sequence

$$A \to aZ_1, \quad Z_1 \to aZ_2, \quad Z_2 \to bZ_3, \quad \text{and } Z_3 \to aB .$$

With a regular grammar normalized in this way, we can easily follow the steps taken in the proof of Theorem 2.6 backwards.

Theorem 2.7. *Every regular grammar can be converted into an equivalent DFA.*

Proof. Without loss of generality, let $G = (\Sigma_{\mathrm{N}}, \Sigma_{\mathrm{T}}, S, P)$ be a normalized regular grammar with $\mathrm{Lang}(G) = L$. All rules are of the form $X \to aY$ or $X \to a$ with $X, Y \in \Sigma_{\mathrm{N}}$ and $a \in \Sigma_{\mathrm{T}}$. From this, we design a DFA with the same language by essentially following the approach from the proof of Theorem 2.6 backwards. We have to be a little more careful though since the rules of G may in fact be ambiguous. Thus, we design an NFA N with $\mathrm{Lang}(N) = L$, which can then be converted to an equivalent DFA by Theorem 1.6.

Without loss of generality, let $\Sigma_{\mathrm{N}} = \{Q_0, Q_1, \dots, Q_{m-1}\}$ with $S = Q_0$ being the start symbol. We define N to have $m + 1$ states $q_0, q_1, \dots, q_m$ with q_0 being the start state, and q_m being the single accepting state.

- For every rule $Q_i \to aQ_j$ with $0 \leq i, j \leq m - 1$, we add the state q_j to the set $\delta(q_i, a)$; i.e., $q_j \in \delta(q_i, a)$.
- For every rule $Q_i \to a$, we add q_m to $\delta(q_i, a)$.

Let $w \in \mathrm{Lang}(G)$ and let $w = w_1 w_2 \dots w_k$. Hence, there is a derivation in G of the form

$$P_1 \Rightarrow w_1 P_2 \Rightarrow w_1 w_2 P_3 \Rightarrow w_1 w_2 \dots w_{k-1} P_k \Rightarrow w_1 w_2 \dots w_k$$

with $P_i \in \Sigma_{\mathrm{N}}$ and $P_1 = Q_0 = S$. By construction, there is thus a run $(p_1, p_2, \dots, p_k)$ with $p_1 = q_0$ (and $p_i = q_j$ if and only if $P_i = Q_j$ with $1 \leq i \leq k$ and $1 \leq j \leq m - 1$) through N reading the word $w_1 w_2 \dots w_{k-1}$. Since P_k can be replaced by the terminal w_k, there is a transition labeled w_k from p_k to the accepting state q_m. Thus, w is accepted by N. It is easy to see that, if there is no derivation of w in G, i.e., $w \notin \mathrm{Lang}(G)$, there is also no accepting computation for N on w. It follows that $\mathrm{Lang}(N) = L$. □

The DFA resulting from an application of the construction for the regular grammar $G = (\{Q_0, Q_1, Q_2, Q_3\}, \{a, b\}, Q_0, P)$ with

$$\begin{aligned} P = \{&Q_0 \to aQ_1 \mid bQ_3 \mid bQ_4, \\ &Q_1 \to aQ_1 \mid bQ_2, \\ &Q_2 \to a \mid b, \\ &Q_3 \to a, \\ &Q_4 \to bQ_4 \mid b\} \end{aligned}$$

is shown in Figure 13b.

Since there is a regular grammar for any DFA and vice versa, we can add the point

6. There is a regular grammar G with $\mathrm{Lang}(G) = L$.

to the five equivalent points of Theorem 1.12.

After having investigated what happens if we restrict CFGs, let us now do the opposite and allow more general rules instead of restricting them further. Once again, the heads of all rules of a CFG are single nonterminals. In a way, this means that the rules are independent of the context of the given nonterminal X. If there is a rule $X \to \alpha$, this rule can be applied whenever we find X in any intermediate derivation, independent of its "context," i.e., where X is located. For a **context-sensitive grammar**, CSG for short, this changes. Here, we allow rules of the form $\alpha \to \beta$ with $\alpha, \beta \in (\Sigma_\mathrm{N} \cup \Sigma_\mathrm{T})^*$; the only restriction is that $|\alpha| \leq |\beta|$. In this case, how a nonterminal X can be replaced depends on the surrounding nonterminals and terminals (there may be productions of the sort $aX \to ab$ and $cX \to cd$ with $a, b, c, d \in \Sigma_\mathrm{T}$, etc.). This adds quite some (substantial) power to grammars. In the preceding section (more specifically, Theorem 2.5), we have learned that there is no CFG generating $L_{abc} = \{a^k b^k c^k \mid k \in \mathbb{N}^+\}$. Now consider the CSG $G_{abc} = (\{S, A, B, C\}, \{a, b, c\}, S, P)$ with

$$\begin{aligned} P = \{&S \to aBSc \mid aBc, \\ &Ba \to aB, \\ &Bc \to bc, \\ &Bb \to bb\} \,. \end{aligned}$$

The idea of G_{abc} is to first use the rules of the first line to generate any (nonempty) word of the form $(aB)^k c^k$, for some $k \in \mathbb{N}^+$; then, the rules of the second line can be used to reorder the symbols to $a^k B^k c^k$; last, the rules of the fourth line can be used to replace the nonterminals B by terminals b, but only if they are at the correct position. For the word $a^3 b^3 c^3 \in L_{abc}$, we get the derivation

$$\begin{aligned} S &\Rightarrow aBSc \Rightarrow aBaBScc \Rightarrow aBaBaBccc \\ &\Rightarrow aBaaBBccc \Rightarrow aaBaBBccc \Rightarrow aaaBBBccc \\ &\Rightarrow aaaBBbccc \Rightarrow aaaBbbccc \Rightarrow aaabbbccc \,. \end{aligned}$$

In order to see that G_{abc} indeed generates L_{abc}, we note that all words from this language can be generated in the above way. Now we argue that every word with a derivation in

G_{abc} has the above form. In particular, every word over Σ_T with a derivation in G_{abc} has the same number of as, bs, and cs, because as, Bs, and cs can only be generated together. Moreover, bs can only be generated when neighboring the leftmost c or another b, and there cannot be any a to the right of a b.

We have now seen three different types of grammars; regular grammars, which can be extended to context-free grammars, and the yet more general context-sensitive grammars. Grammars can be arranged systematically as follows.

Definition 2.8 (Chomsky Hierarchy). *Let $G = (\Sigma_\mathrm{N}, \Sigma_\mathrm{T}, S, P)$ be a grammar. Then G is called a*

- ***type-0 grammar**;*
- ***type-1 grammar** or context-sensitive grammar if P only contains rules of the form*

  $$\alpha \to \beta \qquad \text{with } \alpha, \beta \in (\Sigma_\mathrm{N} \cup \Sigma_\mathrm{T})^* \text{ and } |\alpha| \leq |\beta|;$$

- ***type-2 grammar** or context-free grammar if P only contains rules of the form*

  $$X \to \alpha \qquad \text{with } X \in \Sigma_\mathrm{N},\ \alpha \in (\Sigma_\mathrm{N} \cup \Sigma_\mathrm{T})^*; \text{ and}$$

- ***type-3 grammar** or regular grammar if P only contains rules of the form*

  $$X \to wY, \quad X \to w, \quad \text{and } X \to \varepsilon \qquad \text{with } X, Y \in \Sigma_\mathrm{N} \text{ and } w \in \Sigma_\mathrm{T}^*.$$

We already know that L_{ab} is generated by no type-3 grammar, but by a type-2 grammar, while L_{abc} is generated by no type-2 grammar, but by a type-1 grammar. In general, type-i grammars are more powerful than type-$(i+1)$ grammars in the sense that the former generate more languages than the latter; the only exception is that the empty word is never generated by CSGs, but by some CFGs (although we excluded this case).

2.6 Nondeterministic Pushdown Automata

In this section, we equip ε-NFAs with a particular kind of additional memory, namely a last-in, first-out memory, i.e., a **stack**. With every letter read from the input word, such a nondeterministic stack automaton or **nondeterministic pushdown automaton** (NPdA for short) makes its decisions depending on the state it is currently in, the letter it currently reads, and the top symbol of the stack. Let us continue with a formal definition.

Definition 2.9 (Nondeterministic Pushdown Automaton, NPdA). *An NPdA P is a septuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where*

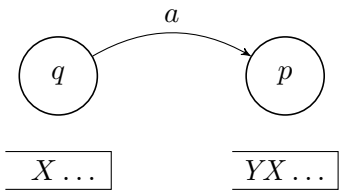
- *Q is a finite set of **states**,*

- Σ *is the* ***input alphabet****,*
- Γ *is the* ***stack alphabet****,*
- $\delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathrm{Pow}(Q \times \Gamma^*)$ *is the* ***transition function****,*
- $q_0 \in Q$ *is the* ***start state****,*
- Z_0 *is the* ***start symbol****, and*
- F *is the set of* ***accepting states****.*

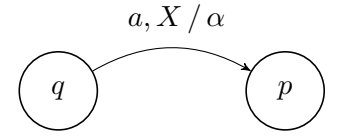
The language $\mathrm{Lang}(P)$ of an NPdA P is defined in the obvious way. More specifically, as for NFAs, NPdAs accept a word if there exists a computation on this word that ends in an accepting state. To understand how this works in detail, let us have a closer look at the transitions of NPdAs.

- The transition function δ takes a triple (q, a, X) where $q \in Q$, $a \in \Sigma$ or $a = \varepsilon$, and X is a stack symbol;
- δ returns a finite set (since NPdAs are nondeterministic) of pairs (p, γ) where $p \in Q$ is the new state and γ is the string of stack symbols that replaces X at the top of the stack. We distinguish the following cases depending on how γ looks like.
  - If $\gamma = \varepsilon$, then the stack is popped, i.e., X is deleted;
  - if $\gamma = X$, then the stack is unchanged; and
  - if $\gamma = YZ$, then X is replaced by Z and Y is pushed onto the stack.

As an example,



visualizes a transition $\delta(q, a, X) = \{(p, YX)\}$ where the top symbol X is replaced by YX, which basically means that Y is pushed onto the stack. For the graphical representation of an NPdA, we use a different approach to indicate the change of the stack content; by labeling a transition by $a, X/\alpha$, we mean that the letter a is read from the input word, and X is the top symbol, which is replaced by α if the transition is followed. Thus, if there is a transition $(p, \alpha) \in \delta(q, a, X)$, we have



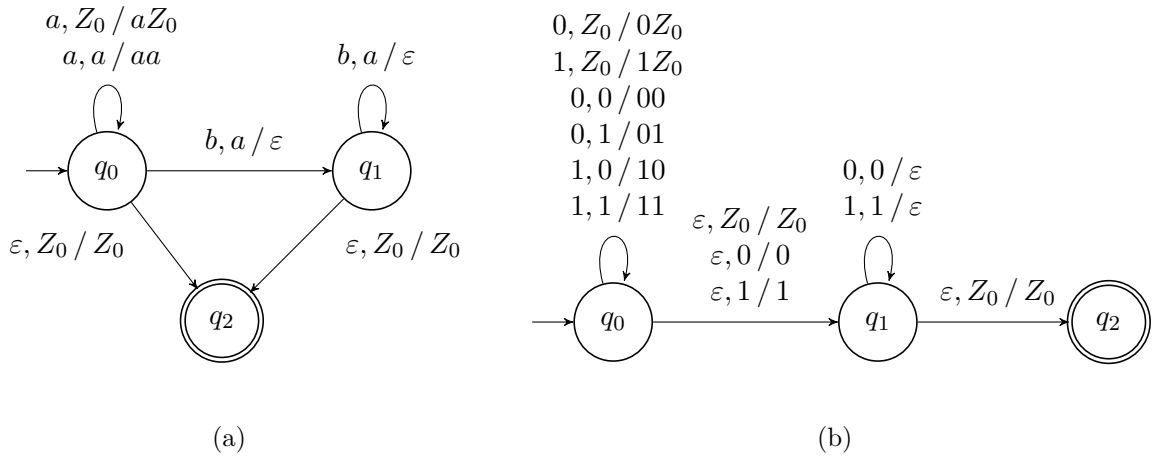


Figure 14.

as a part of the corresponding diagram. All other conventions are similar to those of NFAs, e.g., the start state is indicated by an incoming arrow, etc.

NPdAs are more powerful than DFAs, which we demonstrate by designing an NPdA P_{ab} for the nonregular language $L_{ab} = \{a^k b^k \mid k \in \mathbb{N}\}$. The idea of the construction is as follows.

- While reading as, P_{ab} stays in the start state and pushes as onto its stack.
- While reading bs, as are popped from the stack for every symbol read; on the first b encountered, the state is changed.
- If the stack is empty, an accepting state is entered. This state does not have any outgoing transitions; hence, the word is not accepted in this computation if any further letter has to be read.

P_{ab} is shown in Figure 14a. Now let us investigate how P_{ab} works on the word a^3b^3. At the beginning, P_{ab} finds its stack initialized as $\boxed{Z_0}$ and reads the first letter a. Therefore, the transition $a, Z_0/aZ_0$ is followed, which leads to a stack content $\boxed{aZ_0}$ while P_{ab} stays in q_0. Then the second and third a are read and the transition $a, a/aa$ is followed both times, yielding a stack content $\boxed{aaaZ_0}$. If the first b is read, the transition $b, a/\varepsilon$ leads to $\boxed{aaZ_0}$ and the new state q_1. The second and third b result in popping the two as and the stack content becomes $\boxed{Z_0}$. Finally, since there is no more letter to be read and Z_0 is the top symbol on the stack, P_{ab} enters the accepting state q_2. Note that P_{ab} also accepts the empty word.

Conversely, consider any word w not in L_{ab}. If w starts with b, P_{ab} is stuck since there is no transition for the case that b is read and Z_0 is the top symbol of the stack. Thus, suppose w has a nonempty prefix a^k with $k \in \mathbb{N}^+$. If this is followed by fewer than k bs, w is not accepted, because there is no transition from q_1 for the case that no more letter is read and a is the top symbol of the stack. If more than k bs follow the k as, P_{ab} is again

stuck since there is no transition for the case that Z_0 is the top symbol and a b is read. Finally, after the first b is read, reading another a also results in P_{ab} getting stuck as there is no transition from q_1 or q_2 that can be followed.

We note that P_{ab} is even deterministic; hence, we simply call P_{ab} a DPdA. For the language

$$L_{\text{pal}} = \{ww^{\mathsf{R}} \mid w \in \{a, b\}^*\}$$

of palindromes over $\{a, b\}$ of even length, we need the nondeterminism. Here, an NPdA P_{pal} reads the input word and pushes all letters onto the stack. When P_{pal} reaches the middle of the word, it makes a nondeterministic guess, and starts popping all symbols from the stack while reading the rest of the word; this is shown in Figure 14b. If the word $01100110 \in L_{\text{pal}}$ is read, this leads to a stack content $\boxed{0110Z_0}$ after reading the first four letters. Then P_{pal} makes a nondeterministic guess that half the word is read and changes to q_1 with neither reading a letter nor modifying the stack. After that, it loops in q_1 and pops a letter from the stack if and only if the same letter is read. If and only if this is successful, i.e., if the complete word is read this way and the stack is emptied at the same time, then a transition to the accepting state q_3 is made.

NPdAs accept exactly the context-free languages, and are thus equivalent to CFGs. One way to prove this formally requires to modify NPdAs in a particular way, which we will omit in this introduction. We thus state the following two theorems without proof.

Theorem 2.10. *Every NPdA can be converted into an equivalent CFG.* □

Theorem 2.11. *Every CFG can be converted into an equivalent NPdA.* □

As a consequence, we obtain the following theorem, which characterizes context-free languages.

Theorem 2.12. *The following statements are equivalent.*

1. *L is a context-free language.*
2. *There is an NPdA P with* $\text{Lang}(P) = L$.
3. *There is a CFG G with* $\text{Lang}(G) = L$.

With this, we conclude our studies of context-free languages, and now turn to yet more general ones, which finally allow us to model decision problems as introduced in Section 1.1.

2.7 Historical and Bibliographical Notes

Using (context-free) grammars to generate (natural) languages was first proposed by the linguist Noam Chomsky [2] in 1956, after whom the terms Chomsky normal form (he proposed this normal form in 1959 [3]) and Chomsky hierarchy are named. The presentation of the normalization methods introduced in Sections 2.2.1 to 2.2.4 are taken from the textbook by Hopcroft et al. [11].

The CYK algorithm has its name from Cocke, Younger [28], and Kasami, who proposed it independently. The pumping lemma for context-free languages is due to Bar-Hillel et al. [1]. A generalization is known as Ogden's lemma [23]. Remarkably, Swiss German is one of the textbook examples for a natural language that is not context-free [7] as it allows to derive words of the from ww. As an example, consider the sentence "Me cha säge, dass

$$\underbrace{\text{d'Assistänte}}_{1}\ \underbrace{\text{de Studänte}}_{2}\ \text{d'Ufgabe}\ \underbrace{\text{hälfe}}_{1}\ \underbrace{\text{lööse}}_{2}\text{."}$$

Note that this is not possible in standard German, where one would do a construction of the sort ww^{R}, namely "Man kann sagen, dass

$$\underbrace{\text{die Assistierenden}}_{1}\ \underbrace{\text{den Studierenden}}_{2}\ \text{die Aufgaben}\ \underbrace{\text{lösen}}_{2}\ \underbrace{\text{helfen}}_{1}\text{."}$$

We can even take it two steps further with "Me cha säge, dass

$$\underbrace{\text{d'Rektorin}}_{1}\ \underbrace{\text{de Profässer}}_{2}\ \underbrace{\text{d'Assistänte}}_{3}\ \underbrace{\text{de Studänte}}_{4}$$

$$\text{d'Ufgabe}\ \underbrace{\text{wott}}_{1}\ \underbrace{\text{la}}_{2}\ \underbrace{\text{hälfe}}_{3}\ \underbrace{\text{lööse}}_{4}\text{."}$$

The analogous construction in standard German gives "Man kann sagen, dass

$$\underbrace{\text{die Rektorin}}_{1}\ \underbrace{\text{den Professor}}_{2}\ \underbrace{\text{die Assistierenden}}_{3}\ \underbrace{\text{den Studierenden}}_{4}$$

$$\text{die Aufgaben}\ \underbrace{\text{lösen}}_{4}\ \underbrace{\text{helfen}}_{3}\ \underbrace{\text{lassen}}_{2}\ \underbrace{\text{will}}_{1}\text{."}$$

In Theorems 1.5 and 1.6, we have seen that NFAs and DFAs have the same expressive power. This is not the case for DPdAs and NPdAs; there are languages for which the nondeterminism is somewhat "needed." An example is the language L_{pal}, for which we have designed an NPdA (see Figure 14a) while there is provably no DPdA [11]. It is noteworthy that, if we consider the language $\{wcw^{\mathsf{R}} \mid w \in \{a,b\}^*\}$ instead, we can easily design a DPdA [11]. It follows that the class of languages accepted by DPdAs is a strict superset of the regular languages and a strict subset of the context-free languages.

3 Turing Machines

In Section 2.6, we augmented finite controls (in other words, DFAs) by a stack memory. This allowed us to accept languages such as L_{ab}, which cannot be accepted by finite controls without any additional memory. Still, the data structure of a stack is rather limited in that it does not allow for random access. In this chapter, we exchange the stack by a more general memory, namely a tape. This tape also contains the input at the beginning. So far, the input word was read once from left to right, and the computation ended if the complete word was "consumed." In particular, there was no way to read any part of the input multiple times. This is different for the following model of computation, which may look at arbitrary positions of the input for any number of times and even modify them. We call this machine a **Turing machine**, or TM for short.

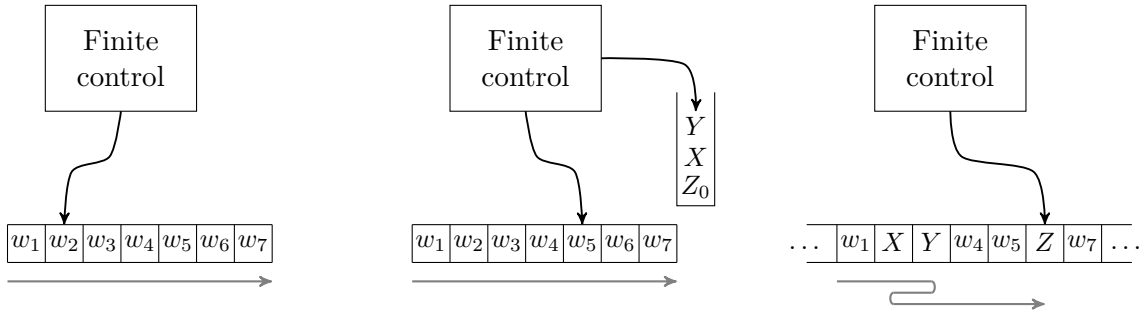
Definition 3.1 (Turing Machine, TM). *A TM M is a septuple $M = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$, where*

- *Q is a finite set of **states**,*
- *Σ is the **input alphabet**,*
- *Γ is the **tape alphabet** and $\Sigma \subseteq \Gamma$,*
- *$\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ is the **transition function**,*
- *$q_0 \in Q$ is the **start state**,*
- *$\sqcup$ is the **blank symbol** with $\sqcup \in \Gamma$ but $\sqcup \notin \Sigma$, and*
- *F is the set of **accepting states**.*

Turing machines are (if not stated otherwise) deterministic. The tape of a TM consists of **tape cells**, which initially contain the input word, one letter per cell. The tape length is infinite to the left and to the right; the cells right of the last letter of the input each contain a blank. At any given point of its computation, a TM "scans" a cell of the tape. It does so using its **tape head** which it may move on the tape in a sequential manner. Such a tape head was not considered explicitly for DFAs, NFAs, or NPdAs since, as mentioned above, they only read the input word once from left to right, anyway. We can, however, in this context, think of a read-only tape that is read once; the different high-level sketches are shown in Figure 15. What all three models do have in common is the **finite control**, i.e., the finite set of states. Let us take a look at the transition function of a given TM.

- The transition function δ takes a pair (q, X) where $q \in Q$ and $X \in \Gamma$;
- δ returns a triple (p, Y, D) where $p \in Q$ is the new state, $Y \in \Gamma$ is a tape symbol that replaces X on the scanned tape cell, and D is a direction, which is either L or R.

Although TMs are deterministic, they do not necessarily have to be "complete" in the sense of DFAs, which means that there may be "missing" transitions. In the case that a



(a) DFA/NFA (b) NPdA (c) TM

Figure 15.

TM cannot follow a transition, it gets stuck and the input word is not accepted; we say that the TM "halts" and that the input is "rejected." Another property that is not consistent with previous machine models is that accepting states cannot be left. If a TM at any point is in an accepting state, the input word is accepted immediately; also in this case the TM "halts." It follows that accepting states do not have any outgoing transitions. At any given point in time, a TM is in a certain **configuration**, which is given by

- the current state,
- the current position of the tape head, and
- the current string that is written on the tape.

Such a configuration of a given TM is denoted by the shorthand

$$X_1 X_2 \dots X_{i-1} q X_i X_{i+1} \dots X_n \ ,$$

which we interpret as follows.

- The word $X_1 X_2 \dots X_n$ is currently written on the tape, where X_1 and X_n are the leftmost and rightmost, respectively, "non-blank" letters (an exception is that the TM scans one of the leading or trailing blanks),
- the current state is q, and
- the tape head currently scans the ith symbol from the left (i.e., the letter written right of the state).

The change from one configuration to another is called a **move** of the TM. Recall that such a move involves possibly changing the state, replacing the symbol in the scanned cell, and moving the tape head either to the left or to the right. Such a move is defined by the $\vdash$-relation (analogously to the derivation $\Rightarrow$-relation for CFGs). We distinguish two cases depending on whether the head moves to the left or to the right.

- If a transition $\delta(q, X_i) = (p, Y, \mathrm{L})$ is followed, this leads to a move

$$X_1 X_2 \ldots X_{i-1} q X_i \ldots X_n \vdash X_1 X_2 \ldots X_{i-2} p X_{i-1} Y \ldots X_n \ .$$

  We have to take special attention to the cases that the head is either at the beginning or the end of the word $X_1 X_2 \ldots X_n$.

  - If $i = 1$, i.e., the tape head scans X_1, then we obtain

$$q X_1 X_2 \ldots X_n \vdash p \sqcup Y X_2 \ldots X_n \ .$$

  - If $i = n$ and $Y = \sqcup$, i.e., the tape head scans X_n and replaces it by a blank, then $\sqcup$ "joins" the blanks right of X_{n-1} and is thus not written down as part of the configuration, i.e.,

$$X_1 X_2 \ldots q X_n \vdash X_1 X_2 \ldots p X_{n-1} \ .$$

- If the transition followed is $\delta(q, X_i) = (p, Y, \mathrm{R})$, we get a move

$$X_1 X_2 \ldots X_{i-1} q X_i \ldots X_n \vdash X_1 X_2 \ldots X_{i-1} Y p X_{i+1} \ldots X_n \ .$$

  There are again two special cases

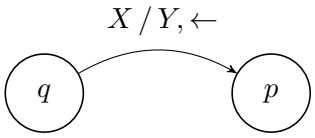
  - If $i = n$, we get

$$X_1 X_2 \ldots q X_n \vdash X_1 X_2 \ldots X_{n-1} Y p \sqcup \ .$$

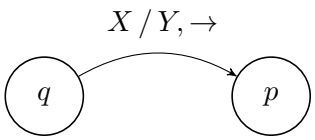
  - If $i = 1$ and $Y = \sqcup$, the result is

$$q X_1 X_2 \ldots X_n \vdash p X_2 \ldots X_n \ .$$

We use a graphical representation similar to those of the other models of computing. For a transition $\delta(q, X) = (p, Y, D)$, we have



if $D = \mathrm{L}$ and



if $D = \mathrm{R}$ as part of the given TM's diagram.

Consider a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$. M's **start configuration** (also called the **initial configuration**) $q_0 w$ corresponds to the situation where only the input word w is written on the tape, M scans the cell with first letter of w and is in the start state q_0. Recall that M accepts a word as soon as M enters an accepting state; hence, the input

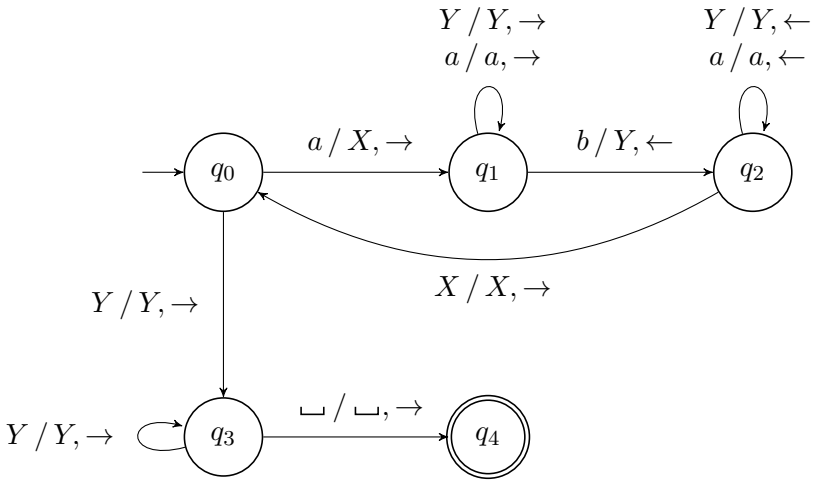


Figure 16.

is accepted if an accepting state appears in the current configuration. Defining $\vdash^*$ in the obvious way, we can formally define the language of M by

$$\mathrm{Lang}(M) = \{w \in \Sigma^* \mid q_0 w \vdash^* \alpha p \beta \text{ with } \alpha, \beta \in \Gamma^* \text{ and } p \in F\} .$$

Let us design TMs for two languages. First, we consider the nonregular language L_{ab} and design a TM M_{ab} with $L_{ab} = \mathrm{Lang}(M_{ab})$. M_{ab} is shown in Figure 16 and works using the following idea. The first a is replaced by the symbol X, which marks it as "checked." After that, M_{ab} moves its tape head to the right over all as until it scans a b, which is then marked as checked by replacing it by Y. After that, the tape head is moved to the left over all as and Ys until the rightmost X is encountered. This is repeated until all as are replaced; this is detected by scanning a Y next to an X. Finally, it is checked whether also all bs have been replaced by checking whether the last Y is followed by a $\sqcup$. If the input word diverges from the form $a^k b^k$ in any way, M_{ab} gets stuck and thus rejects the word.

The computation of M_{ab} on the word $aabb \in L_{ab}$ can be expressed by a sequence of moves

$$\begin{aligned} q_0aabb &\vdash Xq_1abb \vdash Xaq_1bb \vdash Xq_2aYb \vdash q_2XaYb \vdash Xq_0aYb \vdash XXq_1Yb \\ &\vdash XXYq_1b \vdash XXq_2YY \vdash Xq_2XYY \vdash XXq_0YY \vdash XXYq_3Y \\ &\vdash XXYYq_3\sqcup \vdash XXYY\sqcup q_4\sqcup , \end{aligned}$$

where the last configuration is accepting since q_4 is an accepting state; thus $aabb$ is accepted. Conversely, the word $aaba \notin L_{ab}$ yields

$$\begin{aligned} q_0aaba &\vdash Xq_1aba \vdash Xaq_1ba \vdash Xq_2aYa \vdash q_2XaYa \vdash Xq_0aYa \vdash XXq_1Ya \\ &\vdash XXYq_1a \vdash XXYaq_1\sqcup \end{aligned}$$

and, since M_{ab} is stuck in q_1 (there is no outgoing transition for the case that a blank is read), $aaba$ is rejected.

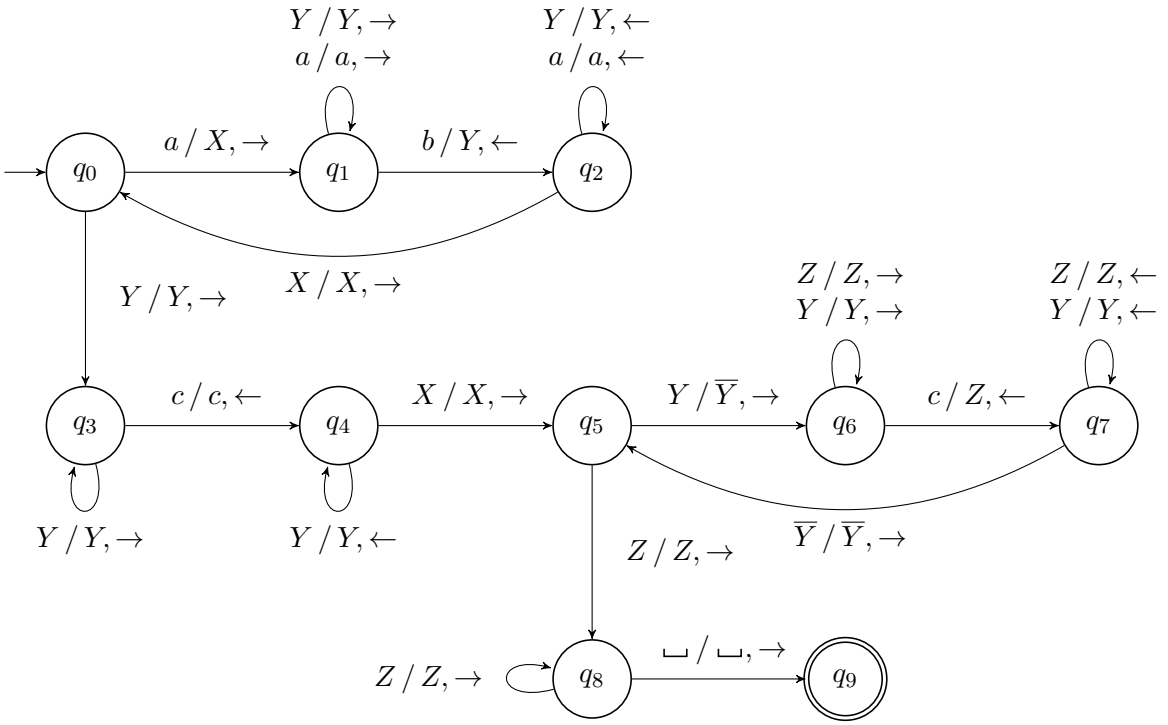


Figure 17.

So far, so good, but we know that L_{ab} can also be accepted by an NPdA. To see that TMs are more powerful than that, let us also design a TM M_{abc} for the non-context-free language L_{abc}. The idea is to iterate the strategy of M_{ab} two times; see Figure 17. First, as are again replaced by Xs and bs by Ys as above. If the input word is $a^k b^k c^k$, we now have a work $X^k Y^k c^k$ on the tape. As a second step, M_{abc} replaces Ys by $\overline{Y}$s and cs by Zs in the same fashion.

The languages for which we can design TMs are called **recursively enumerable**. Designing a TM M for a given language L means that all words in L are accepted by M. However, this does not determine what happens with words that are not in L (in contrast to DFAs or NPdAs). If a word w is not in L, then M must not accept w; but it may either reject w or it may not halt. The latter case is obviously something we would like to avoid, since there is generally no way to test whether a given TM will run forever or halt eventually. There is an important subset of the recursively enumerable languages for which we have TMs that are guaranteed to halt, i.e., every input is either accepted or rejected; such languages are called **recursive languages**.

Definition 3.2 (Recursively Enumerable Language, Recursive Language). *A language L is called **recursively enumerable** if there is a TM M with* $\mathrm{Lang}(M) = L$. *The class of the recursively enumerable languages is*

$$\mathcal{L}_{\mathrm{RE}} = \{L \mid L \text{ is recursively enumerable}\} .$$

*L is called **recursive** if there is a TM M with $\mathrm{Lang}(M) = L$ and M always halts. The class of the recursive languages is*

$$\mathcal{L}_{\mathrm{R}} = \{L \mid L \text{ is recursive}\} .$$

In the remainder of this chapter and throughout the subsequent one, these two classes of languages will be our main object of study. Specifically, we are interested in classifying decision problems into those that are recursive, or recursively enumerable but not recursive, or not even recursively enumerable. We use the following terms.

- If a language L is recursive, we also call L **decidable**, and say that there is a TM which **decides** L.
- If a language L is recursively enumerable, we also call L **semi-decidable**, and say that there is a TM which **recognizes** L.

3.1 Restrictions and Extensions of Turing Machines

There are many different models of TMs that are all equally powerful in terms of their expressive power. We name a few of them, without formally proving that they can accept the exact same languages as TMs according to Definition 3.1. Note that, while the languages of the TMs are not changed, their size (in particular the number of states), may increase significantly. Moreover, the number of moves the TM makes may increase; this point will be addressed in Chapter 5.

- **TMs with a single accepting state.** A TM can have any number of accepting states in general. However, we can easily modify a TM M into a TM M' that only has one accepting state. Since accepting states are never left, but M immediately accepts its input when one of them is entered, we can simply change all transitions to the accepting states of M to point to a single accepting state of M'.
- **TMs with a sink.** A TM can get stuck in a nonaccepting state in which case the input word is rejected. We can convert a TM M into a TM M' that only possesses one such state q_{reject} by adding the "missing transitions" and having them point to q_{reject}. If in M there is no outgoing transition from some state q and a tape letter X, M would get stuck in q when reading X. In this situation, M' changes to q_{reject} where it then gets stuck since there are no outgoing transitions from q_{reject} at all. We call such a state q_{reject} the sink or simply the "rejecting state" of M'.
- **TMs that accept by halting.** Following the above two points, suppose we are given a TM M that contains a single accepting state and a single rejecting state q_{reject}. We can then convert M to a TM M' where q_{reject} contains a transition to itself (i.e., a loop) for every letter of the tape alphabet (including the blank); without loss of generality, the tape head is moved to the right every time the transition is followed. This way, M' runs in an infinite loop (i.e., it does not halt) whenever the input word is not in the language $\mathrm{Lang}(M) = \mathrm{Lang}(M')$.

- **TMs with stationary moves.** As we have defined it, a TM has to move its head either to the left or to the right with every computational move. We can easily allow a third type of move S for "stationary." The transition function then returns a triple with the last entry being L, R, and S. If it is S, the tape head stays on the scanned cell. Such a TM M can be converted into a TM M' without stationary moves by simulating an S-move by an L-move directly followed by an R-move, such that the latter does not change the tape content.

- **Multitape TMs, MTMs.** When designing TMs for given languages, it is often convenient to assume that they are allowed to have more than one tape. The number of tapes may be arbitrary, but fixed, i.e., it may not depend on the input length. With every move, such an MTM M changes its state and the head positions of every tape. We call the first tape of M the input tape and the other tapes M's working tapes. An MTM with one input tape and k working tapes is simply referred to as a ***k*-MTM**.

  A k-MTM M with a tape alphabet Γ can be simulated by a usual TM M' with one tape. Suppose that Γ contains m letters (including the blank $\sqcup$) $X_1, X_2, \ldots, X_m$. At any given point in time t, let $Y_{t,i,j} \in \Gamma$ denote the letter currently written on the jth position of the ith tape of M with $t \in \mathbb{N}$, $j \in \mathbb{Z}$, and $0 \leq i \leq k$ (assigning index 0 to the input tape); thus we can think of the jth position of all tapes as a $(k+1)$-tuple $(Y_{t,0,j}, Y_{t,1,j}, \ldots, Y_{t,k,j})$, which is interpreted as "the input tape contains the letter $Y_{t,0,j}$ at the jth position at time step t, the first working tape contains the letter $Y_{t,1,j}$ at the jth position at time step t," and so on. The tape alphabet Γ' of M' now consists of all such tuples; i.e., any possible contents of any of the cells of M at a fixed position. It thus contains the letters

$$\begin{bmatrix} X_1 \\ X_1 \\ \vdots \\ X_1 \end{bmatrix}, \begin{bmatrix} X_1 \\ X_1 \\ \vdots \\ X_2 \end{bmatrix}, \begin{bmatrix} X_1 \\ X_1 \\ \vdots \\ X_3 \end{bmatrix}, \ldots, \begin{bmatrix} X_2 \\ X_1 \\ \vdots \\ X_1 \end{bmatrix}, \begin{bmatrix} X_2 \\ X_1 \\ \vdots \\ X_2 \end{bmatrix}, \begin{bmatrix} X_2 \\ X_1 \\ \vdots \\ X_3 \end{bmatrix}, \ldots, \begin{bmatrix} X_m \\ X_m \\ \vdots \\ X_{m-1} \end{bmatrix}, \begin{bmatrix} X_m \\ X_m \\ \vdots \\ X_{m-2} \end{bmatrix}, \begin{bmatrix} X_m \\ X_m \\ \vdots \\ X_m \end{bmatrix}.$$

  At time step t, where, without loss of generality, M's tapes do not contain any non-blank left of the first cell or right of the nth cell, the single tape of M' therefore contains the string

$$\begin{bmatrix} Y_{t,0,1} \\ Y_{t,1,1} \\ \vdots \\ Y_{t,k,1} \end{bmatrix}, \begin{bmatrix} Y_{t,0,2} \\ Y_{t,1,2} \\ \vdots \\ Y_{t,k,2} \end{bmatrix}, \begin{bmatrix} Y_{t,0,3} \\ Y_{t,1,3} \\ \vdots \\ Y_{t,k,3} \end{bmatrix}, \ldots, \begin{bmatrix} Y_{t,0,n} \\ Y_{t,1,n} \\ \vdots \\ Y_{t,k,n} \end{bmatrix}$$

  between the leftmost and rightmost non-blank. In order to keep track of the $k+1$ head positions of M, we have to further extend the tape alphabet Γ' of M'. So far, every component X_i of a letter from Γ' is a letter from Γ; now we also allow a second component, which we denote by $\boxed{X_i}$ for every $X_i \in \Gamma$. An example of this construction for a 2-MTM that works over a binary alphabet (thus with a tape alphabet $\{0, 1, \sqcup\}$)

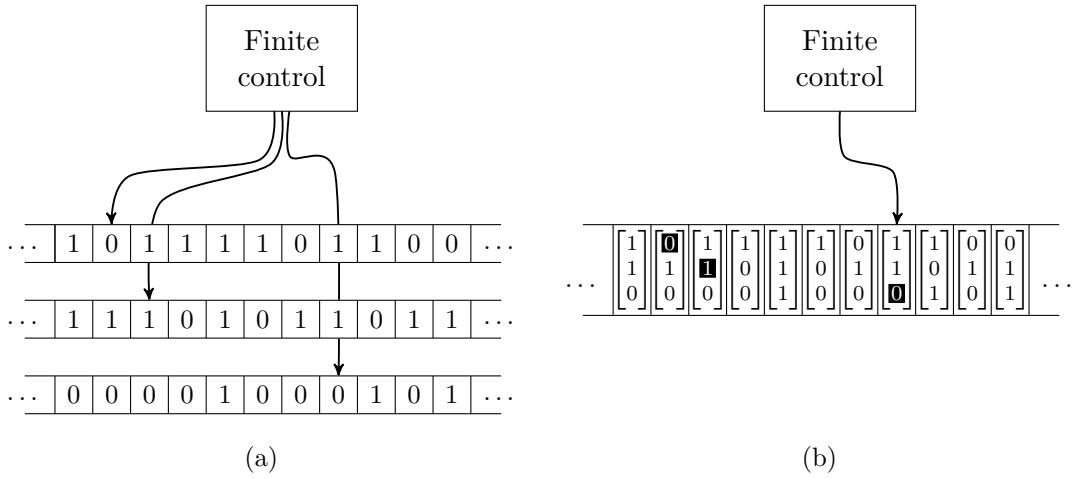


Figure 18.

is shown in Figure 18. We note that the tape alphabet of M' is exponentially larger than that of M, more specifically,

$$|\Gamma'| = (2m)^{k+1} ,$$

and also M' has to contain a lot more states than M.

Moreover, a single move of M needs to be simulated by a possibly very large number of moves made by M'. M' needs to search its whole tape for the $k+1$ positions of the corresponding tape heads of M and change them (and the cells to the left or right) accordingly.

- **Multitape TMs with read-only input tape, Input-RO-MTMs.** An Input-RO-MTM M is an MTM with the restriction that M cannot change the content of its input tape, i.e., it has read-only access on that tape.[2]

- **Semi-infinite TMs.** We can modify a TM M such that it uses its tape as if it is of infinite length only on the right side. To this end, we design a 1-MTM M' from M that simulates on its input tape the part of the input tape of M right of the initial position, and on its working tape the part left of the tape of M; the latter is done in reverse order, and the initial position is marked by a special symbol. M' can then be converted to a TM M'' with a single tape.

  However, it is not allowed to further restrict the "length" of the input tape by making it finite; see Section 3.4.

Since all the above definitions are equivalent, we choose among them in order to design TMs for given languages or to show that no TMs for a given language exist. In the former case, we will usually choose a general model such as an MTM; in the latter case, we will study more restricted models.

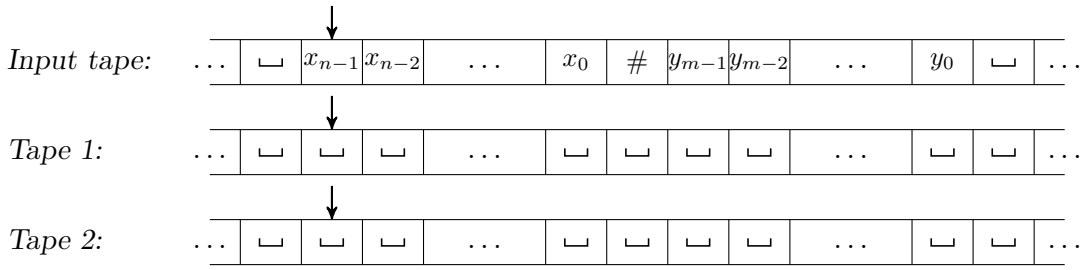
[2]This kind of TM is of special interest when analyzing the space complexity of TMs, which is not coverered by these lecture notes.

3.2 Turing Machines that Compute Functions

In order to show how TMs compute, we leave decision problems for a minute and focus on TMs that compute functions. The input is encoded on the input tape at the beginning, and after the computation is done, the value of the function applied to the input values is written on the first working tape.

We design an 2-MTM M (recall that this implies one input tape and two working tapes) that adds two binary numbers x and y, which are both at least 1. The working tapes of M are simply called "tape 1" and "tape 2." We assume that x has n bits $x_{n-1}, x_{n-2}, \ldots, x_0$ and y has m bits $y_{m-1}, y_{m-2}, \ldots, y_0$, and that they are written with the least significant bit on the right; there are no leading 0s, so both strings start with 1. Furthermore, both numbers are separated by a #. We assume that M is allowed not to move any of the heads in any step, which is indicated by the direction $\downarrow$.

We know that this implies that there also exists a TM M' that is in accordance with the original definition (M' has one tape and can only move the head left or right) that can do the exact same calculations as M. Recall that, since M does not accept a language, we do not need to define an accepting state, but only want to ensure that the correct value is written on tape 1 when M halts. The initial configuration of M is as follows.



M is shown in Figure 19; to make the transition graph as simple as possible, we use the symbol $*$ as a wildcard. A transition of M is labelled

$$X_0, X_1, X_2 \,/\, Y_0, Y_1, Y_2; D_0, D_1, D_2$$

if, on the input tape X_0 is read and replaced by Y_0, on tape 1, X_1 is read and replaced by Y_1, and on tape 2, X_2 is read and replaced by Y_2. The head on the input tape is moved in direction D_0, the head on tape 1 in direction D_1, and the one on tape 2 in direction D_2. Let us briefly explain how it works. In q_0, M runs over all 0s and 1s on the input tape until it encounters the first #. The heads on tape 1 and 2 are not moved. If M finds #, it copies all bits from the input tape to tape 2 in q_1. Eventually, it arrives at the end of the input tape (reading a blank $\sqcup$), and goes to state q_2 while moving the head on tape 2 one position to the left. This means that this head is now positioned on y_0. In q_2, M moves the head on the input tape back until it is positioned on the cell that contains y_{m-1}. As soon as it finds #, it makes one more move to the left, entering q_3. Now, the head on the input tape is positioned on the cell that contains x_0.

At this point in time, M begins with adding the two numbers x and y. It does so while moving from right to left on all tapes simultaneously. Let us first ignore all transitions that have a $\sqcup$ anywhere but on tape 1. If, for instance, M finds a 0 on the input tape and a 1

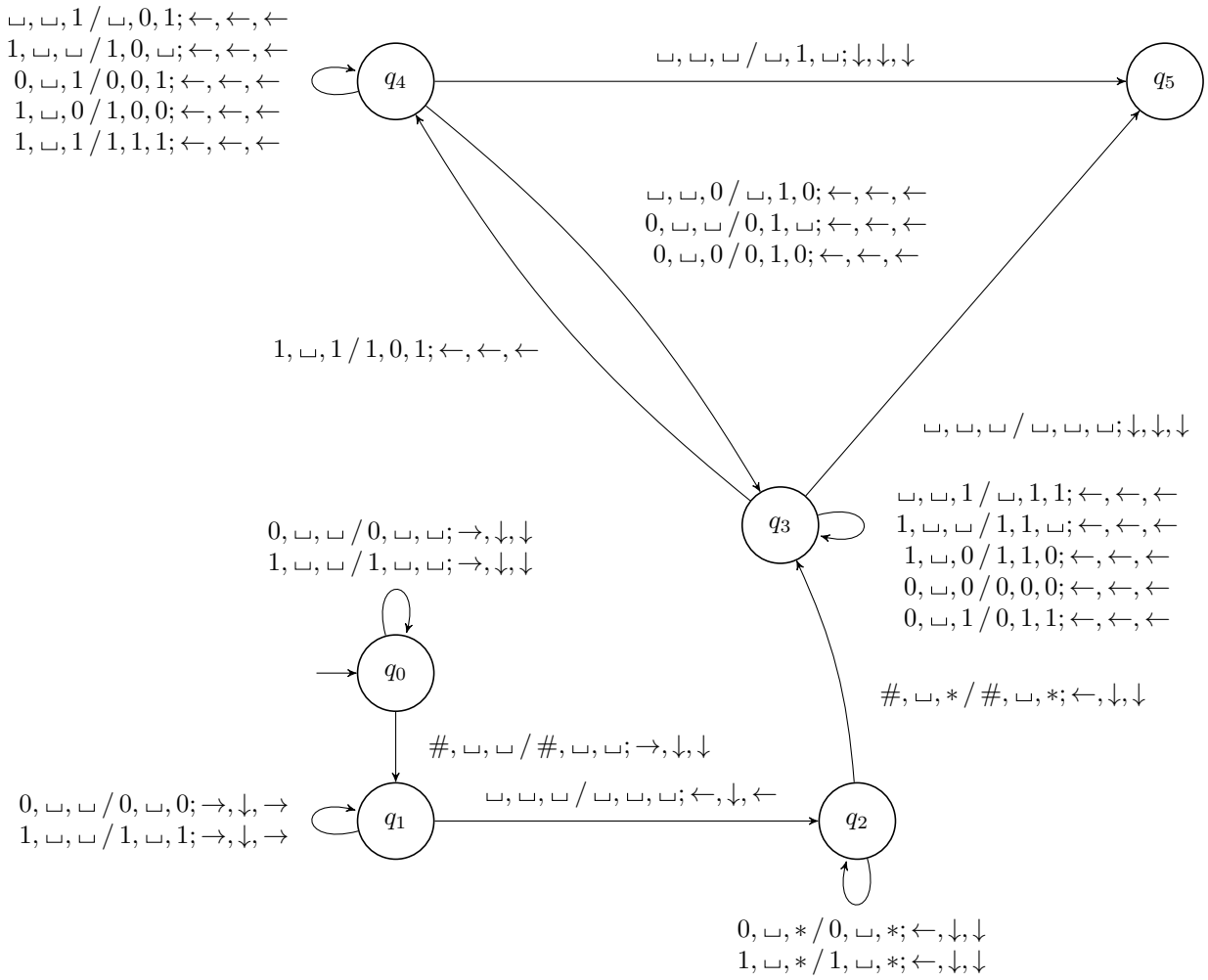


Figure 19.

on tape 2, then it writes a 1 on tape 1. This is indicated by the transition

$$0, \sqcup, 1 \,/\, 0, 1, 1; \leftarrow, \leftarrow, \leftarrow$$

in the figure. If it reads both a 1 on the input tape and on tape 2, it writes a 0 to tape 1, but changes to state q_4. In this state, M knows that it has to take into consideration a carry of 1. The only way to go back to state q_3 is to read both a 0 on the input tape and on tape 2. Indeed, if there is, for instance, a 1 on the input tape and a 0 on tape 2, but we have a carry, the right thing to do is write 0 on tape 1 and maintain the carry, which is formalized by the transition

$$1, \sqcup, 0 \,/\, 1, 0, 0; \leftarrow, \leftarrow, \leftarrow$$

and staying in q_4. From states q_3 or q_4, M goes to q_5 when the work is done, that is, when it finds blanks on both the input tape and tape 2. In the case that it is in q_4, it must write one more 1 to tape 1.

This approach works fine if $n = m$, that is, both binary strings have the same length. However, this is not the case in general. Therefore, M interprets the shorter number as if it contains leading zeros. Since the tape is infinite to the left side, this can easily be done by interpreting blanks as zeros, which is the reason for transitions such as

$$\sqcup, \sqcup, 1 \,/\, \sqcup, 0, 1; \leftarrow, \leftarrow, \leftarrow$$

when being in q_4.

It is obvious that describing TMs on such a low level is tedious work. We are therefore happy to describe them on a more intuitive level. Once we understand the model, it is, for instance, clear that a TM can find the first or last position of the input word, the first occurrence of some given symbol, and so on.

In the above case, M can be described as follows.

- M first scans the input tape to find the symbol #.
- Then, M copies the binary string between # and the first $\sqcup$ to tape 2.
- After that, M moves the heads of its input tape and tape 2 to the left such that the pointer of the input tape is located on x_0, and the head of tape 2 on y_0.
- M then adds the binary values cell per cell from right to left, while writing the result of every single binary addition onto tape 1. Depending on whether it must consider a carry in the current addition step or not, it is in one of two states. Since x and y do not necessarily have the same length in their binary representation, M treats blanks on the left of the words as 0s as long as bits are found on one of the two tapes. In the first step where a blank is encountered on both the input tape and tape 2, M either writes a 1 on tape 1 if it is in the state that indicates that there is a carry and halts, or it halts without any further action if there is no carry.

From now on, we will stick to such a high-level description of TMs in order to investigate which languages are recursively enumerable or even recursive, and which are not.

3.3 The Church-Turing Thesis and the Universal Turing Machine

As we have just seen, we can use TMs in order to compute the sum of two binary numbers. It follows that there is a TM for the language $\{x\#y\#z \mid x, y, z \in \Sigma_{\text{bin}^*} \text{ and } \text{Val}(x) + \text{Val}(y) = \text{Val}(z)\}$, where $\text{Val}(x)$ denotes the value of the binary string x. Clearly, this language is neither regular (for a pumping lemma constant n_0, consider the word $1^{n_0}\#10^{n_0}\#1^{n_0+1}$) nor context-free (which implies the former and can be shown using the same word).

Similarly to the TM that adds binary numbers, we can design a TM that subtracts binary numbers. With a little more effort, we can use our TM for adding binary numbers in order to design a TM for multiplying binary numbers by repeated addition. With this, we can design a TM, e.g., for exponentiation and a TM for division. Being able to divide two numbers allows us to compute the residue of division. With this, we can finally build a TM for L_{PRIME}, i.e., for primality testing.

With similar reasoning, we can deduce TMs for many other problems, and it is widely believed that we can design TMs for exactly those problems that we consider computable on an intuitive level.

Theorem 3.3 (Church-Turing Thesis). *Turing machines can compute exactly what we understand as computable in an intuitive sense.* □

Note that this statement is a thesis, i.e., it is on an axiomatic level, which means it cannot be proven, but has to be trusted, for which up to today we have firm reasons. Indeed, we cannot prove the Church-Turing thesis since the term "computable in an intuitive sense" (also referred to as **effectively computable**) is not well defined. As stated above, the thesis is widely believed and we will assume it to be true in what follows. The even more interesting point under this assumption is the contraposition, which basically states that anything that we cannot compute by means of TMs is not computable in whatever way.

So, following the Church-Turing thesis (Theorem 3.3), we can identify each TM with a computer program, and TMs that always halt with algorithms. But what is the equivalent of a computer then? A computer, in an abstract way, is a device that can run any kind of program given that it is expressed in a language understood by this computer.

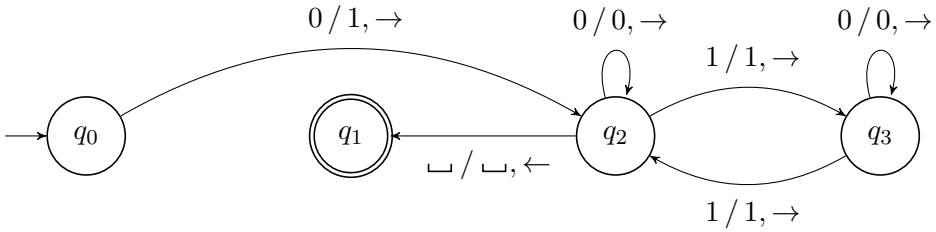
In what follows, we will design a special kind of TM that is able to simulate any given TM on any given input; it is thus called the **universal Turing machine**. Consider TMs with a binary input alphabet and a single accepting state; without loss of generality, this state is q_1. Moreover, let $X_1 = 0$, $X_2 = 1$, and $X_3 = \sqcup$ be all letters from the tape alphabet, and let $D_1 = \mathrm{L}$ and $D_2 = \mathrm{R}$ be the two directions in which the tape head can move. Every single TM can be uniquely described by its transition function. Each single transition

$$\delta(q_i, X_j) = (q_k, X_l, D_m)$$

of M can be written down as

$$0^{i+1}10^j10^{k+1}10^l10^m \ ,$$

and multiple transitions encoded this way can be encoded sequentially and separated by 11. As an example, consider the TM



with four states and the transitions

$$\begin{aligned} &\delta(q_0, 0) = (q_2, 1, \mathrm{R}), \quad \delta(q_2, 0) = (q_2, 0, \mathrm{R}), \quad \delta(q_2, 1) = (q_3, 1, \mathrm{R}), \\ &\delta(q_3, 0) = (q_3, 0, \mathrm{R}), \quad \delta(q_3, 1) = (q_2, 1, \mathrm{R}), \quad \text{and } \delta(q_2, \sqcup) = (q_1, \sqcup, \mathrm{L}) \ , \end{aligned}$$

which is encoded by the binary string

0101000100100 11 00010100010100 11 00010010000100100 11

0000101000010100 11 00001001000100100 11 0001000100100010 .

We denote this encoding by Code(M). Consider the input word $w = 0101$, which leads to a computation, i.e., a sequence

$$q_0 0101 \vdash 1q_2 101 \vdash 11q_3 01 \vdash 110q_3 1 \vdash 1101q_2 \sqcup \vdash 110q_1 1$$

of moves, resulting in M accepting w. Now suppose that we are given M encoded in binary as above. In Code(M), the substring 111 never appears. We thus use 111 as a delimiter between Code(M) and the word w. The string Code(M)111w is unambiguously interpretable and allows us to simulate the work of M on w as just as before, with the only difference that the transitions are encoded in a different way. Since, in Code(M), 0 is encoded as 0 and 1 is encoded as 00, we rewrite w as 010010100. Initially, M is in state q_0, encoded by 0 and reads to first letter of w, i.e., 0. We then search Code(M) for a transition with the prefix 010 (which encodes $\delta(q_0, 0)$); there has to be at most one such transition.

This mechanical work can be done by a TM as well. We call this TM the **universal Turing machine U** since it can simulate any given TM, encoded as Code(M) on any given word w; we will stick to assuming that w is a word over Σ_{bin}. U is a Input-RO-MTM with three working tapes, which we call tape 1, tape 2, and tape 3.

- First, U checks whether the input x is of the form Code(M)111w, i.e., whether there is a substring 111 in the input. The prefix before this substring is then checked with respect to whether it is a valid encoding of a TM. If x does not have the correct form, U changes to a nonaccepting state without any outgoing transition, i.e., U rejects x.

- The part of x behind 111, i.e., the binary word w is copied to tape 1, encoded in the same fashion as in Code(M), i.e., a 0 is encoded by 0 and 1 is encoded by 00; letters are separated by 1s. The head of U on tape 1 is then moved to the first letter of the encoded word w.

- On tape 2, U writes the encoding of q_0 of M according to Code(M), i.e., 0.

- The contents of tape 1 and tape 2 of U are now used to store the current configuration of M working on w; tape 1 stores the current tape content of M, and tape 2 stores the current state of M. The input tape is used to look up the transitions of M.

- Now U searches for the transition corresponding to M being in state q_0 (as indicated by tape 2) and scanning the first letter of w (as indicated by tape 1). To this end, it searches the input tape for the encoding of a transition with the corresponding prefix. If no such transition is found U rejects its input x. If it finds the encoding $010^j10^{k+1}10^l10^m$ of such a transition (and there is at most one), U imitates M's move by changing the encoding of the current state on tape 2 to 0^{k+1}, by changing the content on tape 1 by replacing 0^j by 0^l, and by moving the head on tape 1 as indicated by 0^m.

  If, e.g., a single 0 is replaced by two 0s on tape 1, some parts of the word written on that tape need to be shifted to the right or to the left. For this purpose, U uses tape 3.

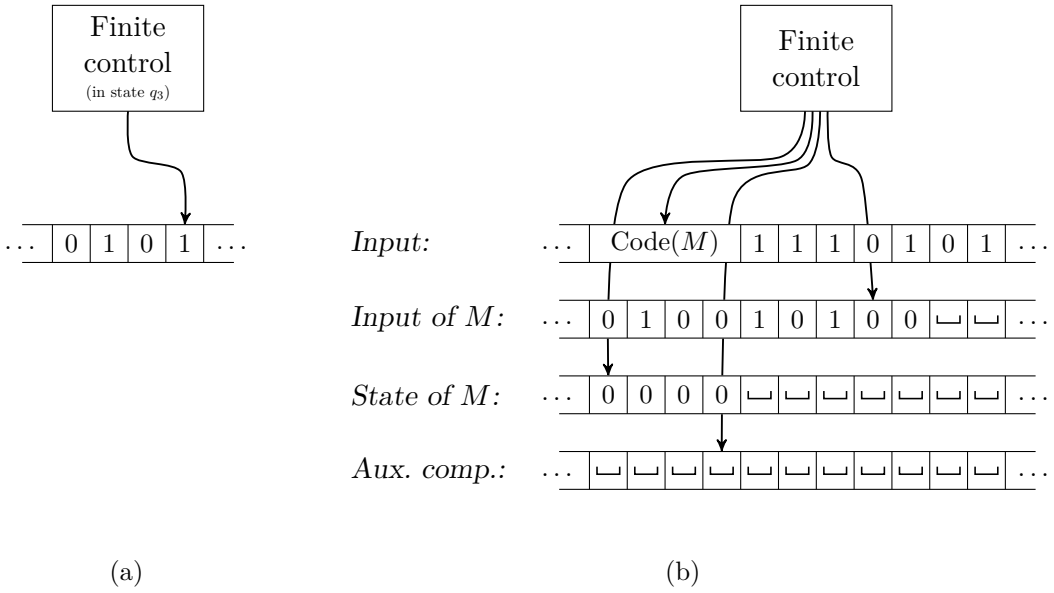


Figure 20.

- This procedure is then repeated until U either writes the encoding 00 of the accepting state of M onto tape 2 or does not find a possible transition. In the former case, U accepts x; in the latter case, U rejects x.

The TM U is sketched in Figure 20, where the input word to M is 0101. We see that we have the following implications if the input x of U has the correct form, i.e., $\text{Code}(M)111w$.

- If M accepts w, i.e., $w \in \text{Lang}(M)$, then U accepts x;
- if M rejects w, then U rejects x; and
- if M does not halt on w, then neither does U.

In particular, M halts on w if and only if U halts on x. Hence, U simulates M on w, and we may consider U as the formalization of a computer that runs programs or algorithms. Summing up, U accepts exactly those words x that can be interpreted as $\text{Code}(M)111w$ for a TM M where M accepts w (i.e., $w \in \text{Lang}(M)$). Sometimes, to keep the notation simple, we will write (M, w) instead of $\text{Code}(M)111w$. We call the language

$$L_{\text{U}} = \{(M, w) \mid w \in \text{Lang}(M)\}$$

the **universal language** and, since $\text{Lang}(U) = L_{\text{U}}$, i.e., the TM U recognizes L_{U}, we have proven the following theorem.

Theorem 3.4. *L_{U} is recursively enumerable.* □

A naturally arising question is of course whether L_{U} is even recursive. Our reasoning about U does not suffice as a proof since U might not halt if $x \notin L_{\text{U}}$ (which happens if and only if M does not halt on w). In the next chapter, we will show that L_{U} is not recursive, but before that we need powerful tools (which we will introduce in Section 4.3) in order to formally prove such a claim.

3.4 Relations to Other Models of Computing

We have already discussed that it is sufficient to assume the tape of the TM to be infinite in one direction. However, we are not allowed to restrict the size of the tape further. Suppose we are given a TM with finite memory, i.e., with a tape length that is fixed from the start. Since the input word is initially written on the tape and has an arbitrary length, we need to use the model of an Input-RO-MTM with one working tape of finite length. It can be shown that such TMs, which we call CTMs, are even no more powerful than the simplest computational model we have looked at, i.e., DFAs, which we state without a proof.

Theorem 3.5. *Every CTM can be converted into an equivalent DFA.* □

TMs are more powerful than both DFAs and NPdAs, since the latter cannot accept $L_{abc} = \{a^k b^k c^k \mid k \in \mathbb{N}^+\}$, as proven in Theorem 2.5, while there is a TM for this language; see Figure 17. A 2-NPdA is an extension of a usual NPdA with the only difference that it has two stacks instead of one. In every computing step, it can change both stacks in the obvious way, that is, pop the top symbol and push any sequence of symbols onto the stack.

Theorem 3.6. *Every TM can be converted into an equivalent 2-NPdA.*

Proof. In what follows, we show how to simulate a TM using a 2-NPdA. The idea behind this is to simulate the tape of the TM using the two stacks in such a way that the top of one stack corresponds to the position of the writing head and a step from left to right is simulated by moving a symbol from one stack to the other.

Let M be a TM. We simulate the work of M on a 2-NPdA P as follows. One important difference between NPdAs (and thus 2-NPdAs) and TMs is that NPdA only read the input once. However, other than DFA, their work is not terminated if the whole input is read. We call the two stacks of P "stack 1" and "stack 2." The two start symbols are called $Z_{0,1}$ and $Z_{0,2}$, respectively. In our intuition, stack 1 is located left of stack 2. Moreover, think of stack 1 as being rotated by 90 degrees to the right, and stack 2 by 90 degrees to the left.

Reading its whole input, P first pushes it onto stack 1. We realize that the input word $X = X_1X_2 \dots X_n$ now is on this stack with X_n being on the top, right above X_{n-1}, and so on. If the input has length 5, the situation is as follows.

Stacks of P: $\boxed{Z_{0,1}\ X_1\ X_2\ X_3\ X_4\ X_5}$ $\boxed{\ Z_{0,2}}$

Next, the symbols are popped from stack 1 to stack 2, which leads to the following situation.

Stacks of P: $\boxed{Z_{0,1}}$ $\boxed{X_1\ X_2\ X_3\ X_4\ X_5\ Z_{0,2}}$

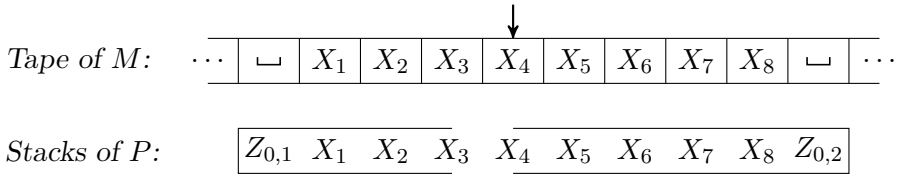
All the transitions of P necessary for this moving (as well as all following steps) are ε-transitions, that is, P does not need to read its own input from now on.

Now we are ready to start the simulation. For this, we interpret the top symbol of stack 2 as the cell of M's tape on which the tape head is located. The remaining content of stack 2 describes the content of the tape right of the tape head, whereas stack 1 describes the tape to the left of the tape head. Changing the symbol at the position of the writing head now can be easily simulated. A move to the right on the tape corresponds to moving the

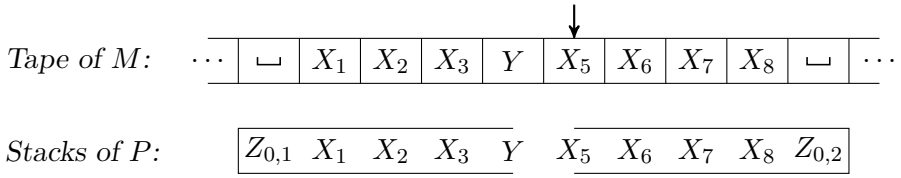
top symbol from stack 2 to stack 1, whereas a move to the left corresponds to moving the top symbol from stack 1 to stack 2. If the symbol is changed by M, P just pushes the new symbol and pops the old one.

There is one special case we need to be careful about. If, for instance, on stack 1, only the start symbol $Z_{0,1}$ is encountered by P, that is, the tape head of M is on the left end of input, and M makes a move to the left, stack 1 remains unchanged and a blank symbol is pushed onto stack 2. The case that the tape head is on the right end of the input is handled analogously.

This way, P can imitate any transition of M while moving between its states that imitate the states of M with the only difference that an operation on the tape is changed to operations on the two stacks as described above. If, for instance, M has its tape head on the symbol X_4 and the input has length 8, we have the following situation.



If M replaces X_4 by Y and moves the tape head to the right, this corresponds to the following situation.



As a result, we conclude that two stacks are as powerful as a tape, and therefore 2-NPdAs are as powerful as TMs. □

3.5 Historical and Bibliographical Notes

In his seminal paper "On computable numbers with an application to the Entscheidungsproblem" [27], Alan Turing introduced the notion of **computing machines**, which today we call Turing machines in his honor. His contribution was not only the formal definition of the term "algorithm," but also pointing out the limitations of automated work; this is what our next chapter is devoted to. More or less simultaneously with Turing, other researchers proposed models for what is computable in an intuitive sense, such as Church's **λ-calculus** [4], and other models by Kleene [15] and Post [24]. It has turned out that all these models are equivalent, which hints towards the correctness of the Church-Turing thesis. Although all these works were predated by Gödel's **incompleteness theorem** [10], Turing's approach can be considered more constructive – and the dawn of computer science.

4 Computability

Now that we have a mathematical formalization of algorithms, we strive to explore the limitations of computers, i.e., we study problems that cannot be solved algorithmically. In this chapter, we will learn about languages that are **undecidable**, and this is meant in the broadest sense. We have, e.g., already seen that there languages, e.g., L_{ab}, which are not decidable by DFAs, but that are decidable by more powerful models of computation, e.g., NPdAs. Here, we want to investigate what the most general model of computation, i.e., TMs, cannot compute. Due to the Church-Turing thesis (Theorem 3.3), it follows that these problems cannot be solved by means of computers, i.e., algorithmically.

4.1 Infinite Sets

Before we analyze what TMs can and cannot compute, we make an excursion into **set theory**. In particular, we are interested in comparing the sizes of different sets. Intuitively, the size of a set A is at most as large as the size of a set B, if we can map every element of A to one unique element of B. If that way every element from B is assigned an element from A, we conclude that A and B have the same size; in the former case, such a mapping is called an injective function, in the latter case we call it a bijective function (or also a "pairing" of the elements of A and B).

Definition 4.1 (Comparison of Sets). *If there is an injective function $f\colon A \to B$ between two sets A and B, then $|A| \leq |B|$. If f is a bijective function, then $|A| = |B|$.*

Now we apply this definition to a number of infinite sets. The first two sets we want to compare are the even natural numbers $\mathbb{N}_{\text{even}} = \{0, 2, 4, \dots\}$ and the odd natural numbers $\mathbb{N}_{\text{odd}} = \{1, 3, 5, \dots\}$. It does not seem very surprising that both sets have the same size, since every even number is followed by an odd one and vice versa.

Theorem 4.2. $|\mathbb{N}_{\text{odd}}| = |\mathbb{N}_{\text{even}}|$.

Proof. In order to prove the claim, we make use of Definition 4.1. To this end, we need to define a bijection between $\mathbb{N}_{\text{odd}}$ and $\mathbb{N}_{\text{even}}$, i.e., we have to assign every element from $\mathbb{N}_{\text{odd}}$ to exactly one element from $\mathbb{N}_{\text{even}}$. Consider the function $f_1\colon \mathbb{N}_{\text{odd}} \to \mathbb{N}_{\text{even}}$ with $f_1(x) = x - 1$, which maps every odd number to exactly one even number. □

Next, we want to compare the natural numbers $\mathbb{N}$ and the positive natural numbers $\mathbb{N}^+$. Both sets are infinite, but our intuition is that they do not have the same size; in particular we have $\{0\} = \mathbb{N} \setminus \mathbb{N}^+$, i.e., the number zero is contained in one set, but not in the other. Rather surprisingly, both sets do have the same size.

Theorem 4.3. $|\mathbb{N}^+| = |\mathbb{N}|$.

Proof. We again apply Definition 4.1. To this end, we need to define a bijection between $\mathbb{N}^+$ and $\mathbb{N}$, i.e., we have to assign every element from $\mathbb{N}^+$ to exactly one element from $\mathbb{N}$

and hit every element from $\mathbb{N}$ with this assignment. Consider the function $f_2 \colon \mathbb{N}^+ \to \mathbb{N}$ with $f_2(x) = x - 1$, i.e., 1 is mapped to 0, 2 is mapped to 1, and so on. It is easy to see that f_2 is indeed a bijection. Every element from $\mathbb{N}^+$ is mapped to a unique element from $\mathbb{N}$ (namely the next smaller one), and every element from $\mathbb{N}$ is mapped to a unique element from $\mathbb{N}^+$ (namely the next larger one). □

Next, consider the integers $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$. This time, there are roughly twice as many elements in $\mathbb{Z}$ than in $\mathbb{N}$ (namely all negative numbers). Still both sets have the same size.

Theorem 4.4. $|\mathbb{Z}| = |\mathbb{N}|$.

Proof. We once more apply Definition 4.1. This time, consider the function $f_3 \colon \mathbb{Z} \to \mathbb{N}$ with $f_3(0) = 0$, $f_3(x) = |2x|$ for $x < 0$, and $f_3(x) = 2x - 1$ for $x > 0$; in words, negative numbers are mapped to even numbers and positive numbers are mapped to odd numbers. Also f_3 is a bijection since every element from $\mathbb{Z}$ is mapped to a unique number from $\mathbb{N}$ and vice versa. □

Sets that have the same size as $\mathbb{N}$ are called **countable**, because of the existence of a bijection between them and the natural numbers; indeed, according to this bijection we can speak of the zeroth, the first, the second, or the hundredth element of the set.

Definition 4.5 (Countable and Uncountable Sets). *An infinite set is called* ***countable*** *if it has the same size as* $\mathbb{N}$*. If a set is larger than* $\mathbb{N}$*, it is called* ***uncountable****.*

As a consequence of Theorems 4.3 and 4.4, both $\mathbb{N}^+$ and $\mathbb{Z}$ are countable. Now let us consider an infinite set that intuitively seems to be yet larger, namely the rational numbers $\mathbb{Q}^+ = \{x/y \mid x \in \mathbb{N}, y \in \mathbb{N}^+\}$. In particular, we see that between any two rational numbers, there are infinitely many other rational numbers; e.g., between 0 and 1, we have 1/2, 1/3, 1/4, and so on. Surprisingly, this does not imply that $\mathbb{Q}^+$ is larger than $\mathbb{N}$.

Theorem 4.6. $|\mathbb{Q}^+| = |\mathbb{N}|$*, i.e.,* $\mathbb{Q}^+$ *is countable.*

Proof. Once more, we apply Definition 4.1. This time, we give a injective function from $\mathbb{Q}^+$ to $\mathbb{N}$. As an intermediate step, we show how to enumerate all pairs $P = \{(x, y) \mid x \in \mathbb{N}, y \in \mathbb{N}^+\}$. To this end, consider the table shown in Figure 21a. We enumerate all the pairs as indicated in Figure 21b; this leads to a enumeration

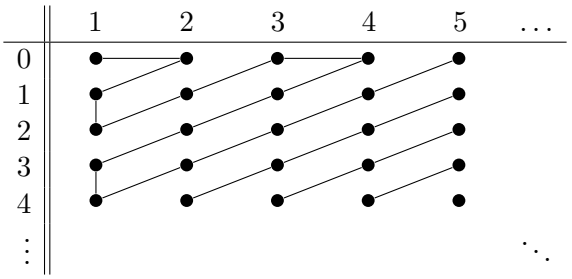
$$(0,1), (0,2), (1,1), (2,1), (1,2), (0,3), (0,4), (1,3), (2,2), (3,1), (4,1), \dots ,$$

and consequently

$$f_4((x,y)) = \begin{cases} g(x,y) + x & \text{if } x + y \text{ is even,} \\ g(x,y) + y - 1 & \text{if } x + y \text{ is odd} \end{cases}$$

	1	2	3	4	5	...
0	(0, 1)	(0, 2)	(0, 3)	(0, 4)	(0, 5)	
1	(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)	
2	(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)	
3	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)	
4	(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)	
⋮						⋱

(a)



(b)

Figure 21.

with

$$g(x, y) = \sum_{i=1}^{x+y-1} i \ .$$

As a second step, we can easily give an injective function $f_5 \colon \mathbb{Q}^+ \to P$ by $f_5((x/y)) = (x, y)$. We note that f_5 is not bijective as, e.g., no rational number is mapped to the pair $(2, 4)$. Therefore, $\mathbb{Q}^+$ is not larger than $\mathbb{N}$ and thus countable. □

It is not very difficult to extend the proof of Theorem 4.6 to the general rational numbers $\mathbb{Q}$. To summarize, in this section we have shown the unintuitive results "$|\mathbb{N}| + 1 = |\mathbb{N}|$," "$2 \cdot |\mathbb{N}| = |\mathbb{N}|$," and even "$|\mathbb{N}| \cdot |\mathbb{N}| = |\mathbb{N}|$."

4.1.1 Hilbert's Hotel

The results of the previous section can be demonstrated by the following example. Consider a hotel, called **Hilbert's hotel**, which contains infinitely many rooms numbered room 1, room 2, room $3, \ldots$. Each room is occupied by exactly one guest, thus there are infinitely many guests and no room is free (as shown in Figure 22a); for ease of presentation, let guest i be the guest who resides in room i, for every $i \in \mathbb{N}^+$.

Now suppose a new guest arrives, and asks for a room. The porter asks each of the guests to move to another room; in particular, guest i is reassigned to room $i + 1$. This way, there is no guest assigned to room 1, and the new guest can move into this room; this is sketched in Figure 22b. If we think about it, this is exactly the same situation as in Theorem 4.3, which exemplifies that "infinity plus 1 is still infinity."

We can even go one step further. This time, instead of one new guest, countably infinitely many new guests arrive. Still, they can easily be assigned an empty room each. The porter simply asks guest i to move from room i to room $2i$. This way, all rooms with an odd index are now available and can be assigned to the new guests; this is sketched in Figure 22c. Here, we have just seen an illustration of the fact that "two times infinity is also still infinity."

Finally, assume there are countably many buses bus 1, bus 2, bus $3, \ldots$ arriving that carry countably many guests each, where guest (i, j) denotes the jth guest of the ith bus

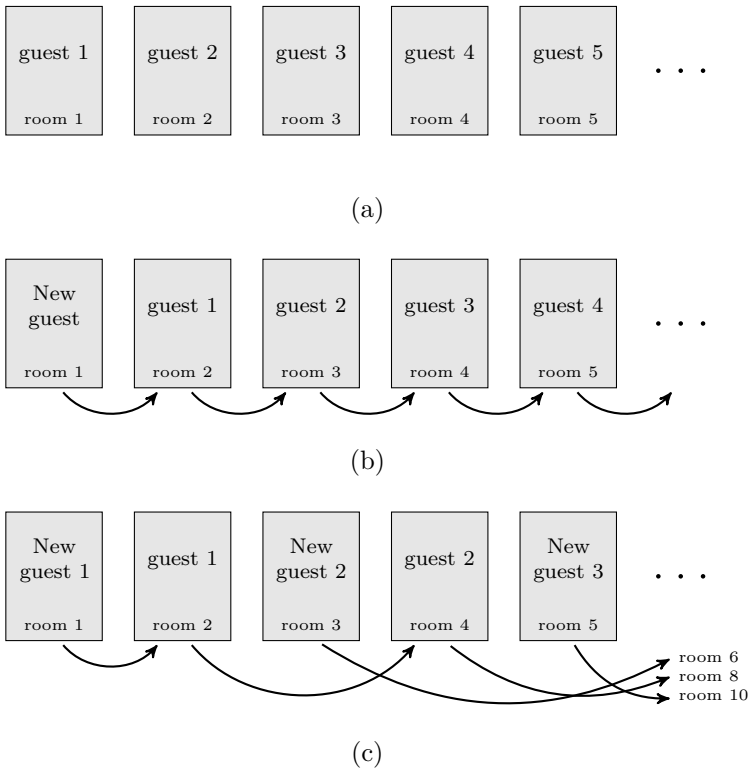


Figure 22.

that arrives. The porter remains unimpressed, as he knows that he can easily assign the guests to rooms according to f_4 from the proof of Theorem 4.6. However, he follows a different approach, and relocates the guests that already reside in the hotel to rooms with even index as before. Next, a unique prime number p_i larger than 2 is assigned to bus i. Finally, guest (i, j), i.e., the jth guest from the ith bus, gets the room with index p_i^j. Since there are infinitely many prime numbers and every natural number has a unique prime factorization, each guest gets a single room. We can interpret this as an illustration of "infinity times infinity is still infinity."

4.1.2 An Uncountable Set

Now we have a surprising insight, namely that there are many infinite sets that have the same size as $\mathbb{N}$ although they contain elements that are not contained in $\mathbb{N}$. Naturally, the question arises whether there are any uncountable sets, and if so, what they look like. The first answer is positive, and we consider the real numbers in this section as an example. Similarly to the rational numbers, there are infinitely many real numbers between any two given real numbers. Indeed, every rational number is also a real number, but there are also real numbers such as $\sqrt{2}$, π, or e, which are not rational. We already know, however, that this fact alone does not show that the real numbers are larger than the rational ones.

Theorem 4.7. $\mathbb{R}$ *is uncountable.*

Proof. We even prove a far stronger statement, namely that already the interval $[0,1]_{\mathbb{R}}$ between 0 and 1 is uncountable. For a contradiction, assume $[0,1]_{\mathbb{R}}$ were countable, i.e., we can enumerate those numbers. Thus, we have the first real number a_1, the second real number a_2, etc. Since $0 \in \mathbb{N}$, we enumerate these numbers by $0, 1, 2, \ldots$. Since every a_i is a real number between 0 and 1, we can write it as

$$0.a_{i1}a_{i2}a_{i3}\ldots\ ,$$

where a_{ij} is a_i's jth decimal place. This way, we get a table

Number	Real number								
0	0.	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	$\ldots$
1	0.	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	
2	0.	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{38}	
3	0.	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	
$\vdots$				$\vdots$						$\ddots$

representing the hypothetical enumeration of $[0,1]_{\mathbb{R}}$. Now we show that this table cannot be complete in that there is at least one real number between 0 and 1 missing. To this end, we create a real number $b = 0.b_1b_2b_3\ldots$ with $b_i \neq a_{ii}$, i.e., b_i is different from a_i at the ith decimal place (and possibly other positions, of course). We have to be careful, however, because for some numbers there are two representations, e.g., $0.001 = 0.000\overline{9}$. To make sure that b is not in the above enumeration, also not with another representation, we ensure that b contains neither 0s nor 9s. Formally, we define

$$b_i = \begin{cases} 1 & \text{if } a_{ii} = 8 \text{ or } a_{ii} = 9, \\ a_{ii} + 1 & \text{else} \end{cases}$$

for every $i \in \mathbb{N}$. For a contradiction, suppose b appears in the hypothetical enumeration of $[0,1]_{\mathbb{R}}$. This means that there is some $j \in \mathbb{N}$ such that $b = a_j$. Now consider the j position of j, i.e., a_{jj}. By construction, $b_j \neq a_{jj}$, and thus b cannot be a_j. As a consequence, b is not contained in the above enumeration, which is a contradiction to its existence. It follows that $[0,1]_{\mathbb{R}}$ cannot be enumerated and is thus not countable. □

The technique applied in the proof of Theorem 4.7 is called **Cantor's diagonal argument**. As an example, consider the hypothetical table

Number	Real number								
0	0.	2	5	6	5	1	4	0	5	$\ldots$
1	0.	6	8	0	0	7	1	4	3	
2	0.	6	1	7	3	9	0	1	9	
3	0.	8	8	7	4	0	8	4	8	
$\vdots$				$\vdots$						$\ddots$

leading to a number $b = 0.3185\ldots$ not contained in this table.

Note that we can give a similar proof when comparing $\mathbb{N}$ with its power set $\text{Pow}(\mathbb{N})$. For every set $S \in \text{Pow}(\mathbb{N})$, consider its **characteristic vector**, which has a 1 at the ith position if and only if $i \in S$; otherwise, it contains a 0 at this position; intuitively, the size of $\text{Pow}(\mathbb{N})$ is $2^{|\mathbb{N}|}$. Suppose we are able to enumerate all sets in $\text{Pow}(\mathbb{N})$ by their characteristic vectors. Then we can again apply Cantor's diagonal argument and show that there is a set whose characteristic vector is missing, and hence "$2^{|\mathbb{N}|} > |\mathbb{N}|$." More generally speaking, we can show that the power set of any set is always larger than the set itself.

4.2 The Diagonalization Language

Such a diagonalization argument can also be applied to decision problems in order to prove that there is such a problem for which no TM exists. Such a language is thus not recursively enumerable since there is no TM that accepts exactly the words of this language. The idea behind the following proof is that there is a countable set of TMs, but an uncountable number of decision problems, i.e., languages.

In what follows, consider the binary alphabet $\Sigma_{\text{bin}} = \{0, 1\}$. Without loss of generality, let us restrict our attention to that alphabet as input alphabet of the TMs we study. We can impose a total ordering $\prec$ on all words from Σ_{bin}^* such that, for all such words w and w',

- if $|w| < |w'|$, then $w \prec w'$, and
- if $|w| = |w'|$, then $w \prec w'$ if $w = u0x$ and $w' = u1y$, i.e., w has a 0 at the first position from the left at which w and w' differ, while w' has a 1 at this position.

We therefore have

$$\varepsilon \prec 0 \prec 1 \prec 00 \prec 01 \prec 10 \prec 11 \prec 000 \prec 001 \prec 010 \prec 011 \prec 100 \prec \dots ,$$

which we call the **canonical order of Σ_{bin}^***, and which allows us to speak of the ith binary word with respect to this order.

Next, we would like to have an enumeration of all TMs as well. As we have seen in Chapter 3, TMs can be encoded in binary, but of course not every binary word is the correct encoding of a TM. However, using the canonical order of Σ_{bin}^*, we can also impose an order on all TMs. We enumerate all binary words as above, and check for every word whether it is a valid encoding of a TM. If the first word is found that actually encodes a TM, we define this TM to be the first TM M_1. Then we continue until we find the next word that encodes a TM; the corresponding TM is called M_2. This way, we get an enumeration $M_1, M_2, M_3, \dots$ of all TMs.

An analogous argument for computer programs is easy to follow. Suppose we define a canonical order on all words over the ASCII (UNICODE, respectively) alphabet and design a program that enumerates all those words; after every generation of a word, this word is supplied to, say, a C compiler. In most of the cases, the compiler will, of course, not be able to process the word and thus exit with an error message. However, eventually the string

```
int main(){}
```

will be generated and compiled, which then equals the first program in C according to the above order; the second string that is successfully compiled is

```
int main(␣){}
```

and so on, which gives us an enumeration $C_1, C_2, C_3, \ldots$ of all valid C programs.

Now consider the enumeration $w_1, w_2, w_3, \ldots$ of all binary words and the enumeration $M_1, M_2, M_3, \ldots$ of all TMs (encoded as binary words) with a binary input alphabet. Every TM M_i accepts a specific set of binary words, i.e., a language $\text{Lang}(M_i) \subseteq \Sigma_{\text{bin}}^*$. We design a table whose rows correspond to TMs and whose columns correspond to binary words. Specifically, the entry in the ith row and the jth column is 1 if M_i accepts w_j, i.e., $w_j \in \text{Lang}(M_i)$; this entry is 0 if $w_j \notin \text{Lang}(M_i)$. Now we define a language

$$L_{\text{diag}} = \{w_i \mid w_i \notin \text{Lang}(M_i)\} ,$$

i.e., L_{diag} contains the ith binary word w_i if and only if this word is not accepted by the ith TM M_i. Consider the hypothetical table T

	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	$\ldots$
M_1	1	0	1	1	1	1	1	0	0	
M_2	0	0	0	0	0	0	1	0	1	$\ldots$
M_3	1	0	0	1	0	1	0	0	0	
M_4	0	0	1	1	0	0	1	0	0	
$\vdots$				$\vdots$						$\ddots$

constructed as described above; in this case w_1 would not be contained in L_{diag}, but w_2 would and so would w_3; w_4 would not be included, etc.

Note that it is impossible to "write down" T since it is of infinite size in both dimensions. Here, T is just a means to visualize the language L_{diag} on an intuitive level.

Theorem 4.8. *L_{diag} is not recursively enumerable.*

Proof. For a contradiction, suppose L_{diag} were recursively enumerable. This means that there is a TM M for L_{diag}, i.e., $\text{Lang}(M) = L_{\text{diag}}$. Since M is a TM, it must appear in the enumeration of all TMs. In other words, there has to be an $i \in \mathbb{N}^+$ such that i is the index of M according to the enumeration of all TMs; i.e., $M = M_i$. It therefore has to follow that $L_{\text{diag}} = \text{Lang}(M_i)$. Now consider the ith binary string w_i; this word is either in L_{diag} or it is not.

- If $w_i \in L_{\text{diag}}$, then $T_{ii} = 0$ and M_i does not accept w_i, i.e., $w_i \notin \text{Lang}(M_i)$.
- If $w_i \notin L_{\text{diag}}$, then $T_{ii} = 1$ and M_i accepts w_i, i.e., $w_i \in \text{Lang}(M_i)$.

As a consequence, we obtain the equivalence

$$w_i \in L_{\text{diag}} \iff w_i \notin \text{Lang}(M_i) ,$$

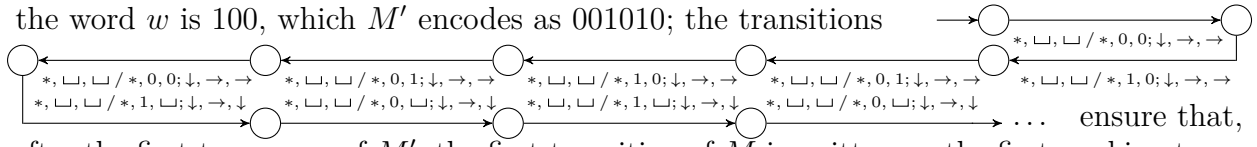
which is a direct contradiction to $L_{\text{diag}} = \text{Lang}(M_i)$. Therefore, M_i cannot exist and L_{diag} cannot be recursively enumerable as a consequence. □

We see that the ith row of T corresponds to the characteristic vector of the language $\text{Lang}(M_i)$. With L_{diag}, we have designed a language with a characteristic vector that is different from all those in T; more specifically, the characteristic vector of L_{diag} is different from that of $\text{Lang}(M_i)$ at the ith position (and possibly other positions). Note that this is exactly the same application of Cantor's diagonal argument as in the proof of Theorem 4.7.

4.3 Reductions and Basic Properties

We have now encountered a decision problem, which is not recursively enumerable, namely L_{diag}; since recursive languages are a subset of recursively enumerable languages, we also know that L_{diag} is not recursive. In what follows, we would like to build up a theory of nonrecursive decision problems. In particular, we are interested in problems that are of more practical use; in fact, L_{diag} does seem like a rather artificial problem.

To this end, we need to design TMs that simulate a given TM on a given word (or even multiple TMs on multiple words) similar to the proof of Theorem 3.4. In contrast to the universal TM U, however, we design a TM M' which simulates a TM M on a word w without being given M and w as input, but they are "hardwired" into M'. As an example, suppose that the first transition of M is $\delta(q_0, 0) = (q_1, 0, \text{L})$, which is encoded as 0101001010, and the word w is 100, which M' encodes as 001010; the transitions



... ensure that, after the first ten moves of M', the first transition of M is written on the first working tape, and the word w is written on the second working tape. After all transitions of M and the complete word w are written down, M' simulates M on w the same way U would.

We start with two general observations that speak about recursively enumerable and recursive languages. In the following, a "$*$" in the superscript of a TM M indicates that M always halts.

Theorem 4.9. *If a language L is recursive, then $\overline{L}$ is recursive, too.*

Proof. Suppose L is recursive and let M^* be a TM that decides L, i.e., for every input w, M^* accepts w if $w \in L$, and M^* rejects w if $w \notin L$; in particular M^* always halts. Without loss of generality (see Section 3.1), we assume that M^* has exactly one accepting state and one rejecting state. Thus, every word given to M^* as input ends in exactly one of these two states. We design a TM $\overline{M}^*$ which is a copy of M^* with the only difference that the accepting and the rejecting state are switched. Any word w that is accepted by M^* is thus rejected by $\overline{M}^*$ and vice versa. As a result, $\overline{M}^*$ accepts $\overline{L}$ and is guaranteed to halt, and hence $\overline{L}$ is recursive. □

Note that a direct consequence of Theorem 4.9 is that, if a language L is not recursive, then neither is $\overline{L}$.

Theorem 4.10. *If both a language L and $\overline{L}$ are recursively enumerable, then L is recursive.*

Proof. Let M be a TM for L and let $\overline{M}$ a TM for $\overline{L}$. Every word w is either in L or in $\overline{L}$; thus w is accepted either by M or by $\overline{M}$. We design a TM M^* which simulates M and $\overline{M}$

on its input w at the same time. Eventually, either M or $\overline{M}$ has to accept w. If M accepts w, M^* accepts; if $\overline{M}$ accepts, M^* rejects. As a result, M^* accepts L and is guaranteed to halt, and hence L is recursive. □

An implication of Theorems 4.9 and 4.10 is that, for any language L, one of the following points is true.

- Both L and $\overline{L}$ are recursive;
- neither L nor $\overline{L}$ are recursively enumerable; or
- L ($\overline{L}$, respectively) is recursively enumerable but not recursive and $\overline{L}$ (L, respectively) is not recursively enumerable.

Next, we introduce a general technique that allows us to prove that a given language is not recursive or even not recursively enumerable. To this end, suppose we already know that a language L_1 is not recursive (recursively enumerable, respectively). A **reduction** establishes a connection between L_1 and a second language L_2 implying that

> if L_2 were recursive (recursively enumerable, respectively),
> then L_1 would be recursive (recursively enumerable, respectively), too.

The two facts that L_1 is already known to be nonrecursive (not recursively enumerable), but it would be recursive (recursively enumerable) if L_2 would be recursive (recursively enumerable), immediately implies that L_2 cannot be recursive (recursively enumerable). A reduction is carried out by a special kind of algorithm (i.e., a TM that always halts) which converts inputs for L_1 to inputs for L_2.

Definition 4.11 (Reduction). *Let L_1 and L_2 be two languages. If there is a TM R which converts inputs x for L_1 to inputs $y = R(x)$ for L_2 such that*

$$x \in L_1 \iff y \in L_2 \,,$$

*we say that **L_1 reduces to L_2** or that there is a **reduction from L_1 to L_2**, denoted by $L_1 \leq_\mathsf{r} L_2$.*

We now formally prove that such a reduction is indeed a valuable tool in order to prove a language to be nonrecursive (not recursively enumerable).

Theorem 4.12. *Let L_1 and L_2 be two languages, and suppose $L_1 \leq_\mathsf{r} L_2$.*

- *If L_1 is not recursive, then neither is L_2;*
- *if L_1 is not recursively enumerable, then neither is L_2.*

Proof. We start by proving the first claim. As usual, we use the terms "recursive" and "decidable" interchangeably. Since there is a reduction from L_1 to L_2, there is a TM R that converts inputs for L_1 to inputs for L_2 as in Definition 4.11. Consider any input x for L_1. Our goal is to decide whether $x \in L_1$ or $x \notin L_1$; if we can achieve this, L_1 is recursive.

For contradiction, suppose L_2 were recursive and L_1 were not; hence, there is a hypothetical TM M_2^* that decides L_2. We show how to design a TM M_1^* that decides L_1 using M_2^*. To this end M_1^* first converts x (which it receives as input) to an input y for L_2 using the TM R. As a result, $y \in L_2$ if and only if $x \in L_1$ due to Definition 4.11. The word y is then given to M_2^* as input, i.e., M_1^* simulates M_2^* on y. Since M_2^* decides L_2, M_2^* gives the correct answer as to whether $y \in L_2$ or $y \notin L_2$ after finite time by accepting or rejecting y. This answer is then also correct for the question concerning whether $x \in L_1$ or $x \notin L_1$. Thus, M_1^* accepts if M_2^* accepts and rejects if M_2^* rejects. Consequently, M_1^* decides L_1, and L_1 is therefore recursive by definition. This, however, is a contradiction to our assumptions, and thus also L_2 cannot be recursive.

As for the second claim, we can do a similar reasoning with a hypothetical TM M_2 for L_2. The only difference to the above argumentation is that M_2 does not necessarily halt if $y \notin L_2$. However, M_2 always accepts y if $y \in L_2$. Likewise, M_1 accepts any x with $x \in L_1$, and hence L_1 is recursively enumerable, which again contradicts our assumptions. Therefore, L_2 cannot be recursively enumerable. □

In the subsequent sections, we will use reductions in order to show that a number of languages are not recursive.

4.4 The Universal Language

We have already studied the language L_U, which contains all words $\mathrm{Code}(M)111w$ (which we simply write as "(M, w)") with $w \in \mathrm{Lang}(M)$ in Section 3.3 and proved that it is recursively enumerable (see Theorem 3.4) since the universal TM U accepts L_U. However, since U simply simulates M on w, it runs forever if M runs forever on w; i.e., it might not halt if $(M, w) \notin L_\mathrm{U}$. Of course, it may be the case that there is a more sophisticated way of testing whether a given word is in L_U that always gives an answer within finite time; in this case, L_U were recursive. However, in what follows, we will show that there is no TM for L_U that is guaranteed to halt.

Theorem 4.13. *L_U is not recursive.*

Proof. We know that, if L_U were recursive, then also its complement $\overline{L}_\mathrm{U}$ would be recursive (see Theorem 4.9). Thus, if we succeed in showing that $\overline{L}_\mathrm{U}$ is not recursive, then L_U cannot be recursive. We have to be careful here, because $\overline{L}_\mathrm{U}$ also contains the binary strings that are not proper encodings of TMs and binary strings separated by "111." Formally, the language is thus defined as

$$\begin{aligned}\overline{L}_\mathrm{U} = {} & \{x \mid x \text{ does not encode a TM and a word separated by } 111\} \\ & \cup \{\mathrm{Code}(M)111w \mid M \text{ is a TM and } M \text{ does not accept } w\}\end{aligned}$$

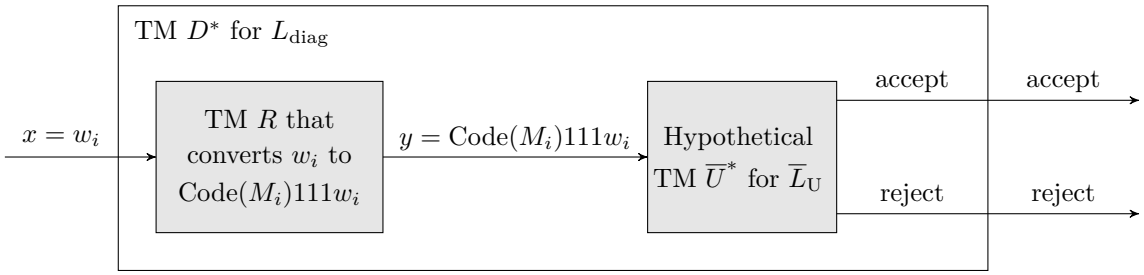


Figure 23.

We reduce the problem of deciding whether a given word is in L_{diag} to deciding whether some word is in $\overline{L}_{\text{U}}$, i.e., $L_{\text{diag}} \leq_{\text{r}} \overline{L}_{\text{U}}$. If then we would have a TM for deciding $\overline{L}_{\text{U}}$ (i.e., if this language were recursive), we could use it to decide L_{diag} (i.e., this language were also recursive). As before, we mark all TMs with "$*$" if they are guaranteed to halt. So suppose we have a TM $\overline{U}^*$ that decides $\overline{L}_{\text{U}}$. Let $x \in \{0,1\}^*$ be any word for which we want to know whether it is in L_{diag}. What does that mean? The word x is at some position, say i, in the enumeration of all possible binary strings; thus, we just write w_i instead of x. Deciding whether $w_i \in L_{\text{diag}}$ just means to decide whether this word is in $\text{Lang}(M_i)$, i.e., accepted by the ith TM. If w_i is not in $\text{Lang}(M_i)$, then it is in L_{diag}; if it is in $\text{Lang}(M_i)$, then it is not in L_{diag}. Our reduction is illustrated in Figure 23. The idea is that we can use the hypothetical TM $\overline{U}^*$ to do the work for us. To this end, we design a TM D^* that decides L_{diag} by first giving $x = w_i$ to a TM R that computes the index i and the ith TM M_i. After that, R outputs the word $y = \text{Code}(M_i)111w_i$, which is a valid input for $\overline{U}^*$. It expects the part before the three 1s to be the encoding of a TM and the string behind them to be some binary word. Then, by the definition of $\overline{L}_{\text{U}}$, $\overline{U}^*$ will accept if and only if the TM M_i that is encoded does not accept w_i. This is exactly the case if $x = w_i$ is in L_{diag}; therefore

$$x \in L_{\text{diag}} \iff y \in \overline{L}_{\text{U}} \ ,$$

and we can simply give the same answer to decide whether x is in L_{diag}. As a result, D^* decides L_{diag}, which is a contradiction to Theorem 4.8. Thus, neither $\overline{L}_{\text{U}}$ nor L_{U} are recursive. □

As a consequence of Theorems 3.4 and 4.13, $L_{\text{U}} \in \mathcal{L}_{\text{RE}} \setminus \mathcal{L}_{\text{R}}$. The above proof even shows that $\overline{L}_{\text{U}}$ is not recursively enumerable. With this, we can separate all language classes we have learned about so far by giving a language that is in one of them, but not in any subclass; see Figure 24.

4.5 The Halting Problem

As mentioned above, we use the terms "language" and "problem" interchangeably, and we may thus refer to L_{U} as the universal problem, which is to figure out whether a given TM accepts a given word. A related problem is the **halting problem** $\boldsymbol{L}_{\text{H}}$; here, we are also given the encoding of a TM M and a binary string w as input, but instead of deciding whether M accepts w, we want to know whether M halts on w, i.e., we define

$$L_{\text{H}} = \{(M, w) \mid M \text{ halts on } w\} \ .$$

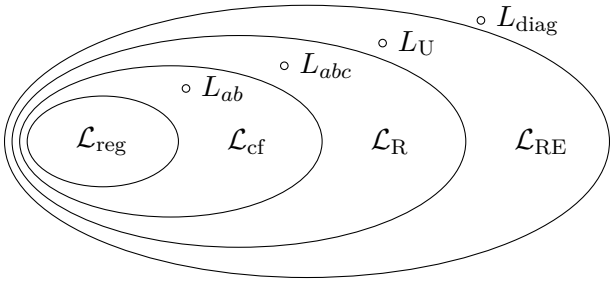


Figure 24.

We first show that there is a TM H that accepts L_{H}.

Theorem 4.14. *L_{H} is recursively enumerable.*

Proof. The proof can be done analogously to the proof of Theorem 3.4. A TM H recognizing L_{H} works the same way as the universal TM U on a given input $x = (M, w)$. The only difference is that H also accepts x if M rejects w. This way, if M halts (no matter whether w is accepted or rejected) on w, H accepts x; if M runs forever on w, H also runs forever, and thus does not accept x. Hence, $\text{Lang}(H) = L_{\text{H}}$, and L_{H} is recursively enumerable as a consequence. □

We will now prove that the halting problem is not recursive. In essence, this means that we cannot decide whether a given TM halts on a given word in general. Since we consider TMs that halt to be algorithms, we can draw the conclusion that it is not possible to decide whether a given computer program is actually an algorithm. To see that this is indeed a problem, consider the computer program GoCon, which is given in pseudocode in Algorithm 4.1 and uses a function "isprime( )," which returns true if and only if its argument is a prime number. Is GoCon an algorithm, i.e., does this program, which takes no input, eventually halt? By inspection, we observe that this only happens if an even number n is found which cannot be written as $n = p + q$, where p and q are prime numbers. In other words, GoCon halts if and only if we find an even number that cannot be written as the sum of two prime numbers.

However, whether such an even number exists, is one of the most famous open questions in mathematics, which is is known as Goldbach's conjecture. Thus, answering whether GoCon halts means proving or disproving Goldbach's conjecture. Of course, this is not a formal proof that halting cannot be decided, yet it shows that the question of whether a given program halts or not cannot be answered by simply "taking a close look" at it. Understanding in detail how GoCon works and under which condition the while-loop is left, does not help at all.

Algorithm 4.1. The program GoCon.

```
n = 4;                                                    /* Initialization. */
p = 1;

while (true)
   p = p + 1;
   if (isprime(p))
      q = n − p;
      if (isprime(q))                                     /* Both p and q are prime and n = p + q. */
         print(n . " = " . p . " + " . q);
         n = n + 2;
         p = 1;
   if (p > n/2)                                           /* No primes p and q with n = p + q were found. */
      return;
end
```

We will present two different approaches to prove the subsequent theorem which formally states that "halting" cannot be decided in general, i.e., we cannot design an algorithm that decides whether a given TM halts on a given input (while we may of course be successful for a specific class of special TMs).

Theorem 4.15. *L_{H} is not recursive.*

Proof. We give a reduction from L_{U}, i.e., we show $L_{\mathrm{U}} \leq_{\mathsf{r}} L_{\mathrm{H}}$. Since we know L_{U} is not recursive due to Theorem 4.13, neither is L_{H}. We design a TM U^* that decides L_{U} given a TM H^* that decides L_{H}. Note that U^* is different from the universal TM U for L_{U}, which is not guaranteed to halt. U^* passes its input x to a TM R, which first checks whether it has the correct form, i.e., whether $x = \mathrm{Code}(M)111w$ for a TM M and a binary string w.

- If not, the word $y = x$ is output by R and given to H^*. For sure, H^* will not accept x since it does not have the correct form, and U^* should not accept x. Thus, the answer (output) of H^* is correct as answer (output) for U^*.
- If, however, x has the correct form, R inspects the encoding $\mathrm{Code}(M)$ of M, and creates $\mathrm{Code}(M')$, which is the encoding of a TM M' that works similarly to M but with one important difference. M' has one additional state q in which it always runs into an infinite loop. If a transition is not present in M in some non-accepting state (that is, a state in which M might get stuck and thus halt without accepting), R adds this "missing" transition such that it ends in q. Therefore, when M halts without accepting, M' does not halt; this idea was already described in Section 3.1. Then, $y = \mathrm{Code}(M')111w$ is given to H^* as input.

We thus get the following implications.

- If H^* accepts $\mathrm{Code}(M')111w$, we know that M' halts on w and therefore accepts w. But in this case, also M accepts w; this is a direct consequence of the way that M' is constructed from M.

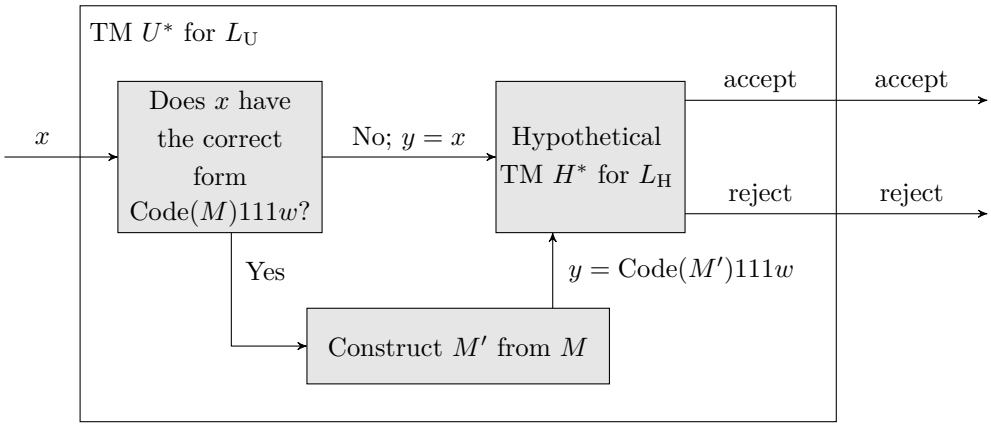


Figure 25.

- If H^* does not accept Code(M')111w, we know that M' runs into an infinite loop on w. In this case, M either halts on w without accepting or it also does not halt. In either case, M does not accept w. The idea of the construction is shown in Figure 25, where the two steps of R are shown separately.

This yields the equivalence

$$M \text{ accepts } w \iff M' \text{ halts on } w \,.$$

Therefore, U^* can take the answer of H^* for the input Code(M')111w as a correct answer for its own input x. This means that U^* is a TM that decides L_{U}, but this leads to a contradiction to Theorem 4.13. Thus, there is no TM H^* that decides L_{H}, and hence L_{H} is not recursive. □

Note that a similar approach can be used to reduce L_{H} to L_{U}. Now we give an alternative proof that does not use a reduction; however, the ideas employed in this proof incorporate the same ones (in particular, Cantor's diagonal argument) as in the proof of Theorem 4.8.

Alternative Proof. For a contradiction, suppose L_{H} were recursive, and let H^* be a TM that decides L_{H}. Consider a TM M that does the following. On a binary input word w, M first computes the index $i \in \mathbb{N}^+$ of w according to the canonical order, i.e., $w_i = w$; furthermore, M computes the encoding of the ith TM M_i. Now M uses H^* to decide whether M_i halts on w_i. If so, M runs into an infinite loop; if M_i does not halt on w_i, M halts, e.g., by moving to an accepting state.

Since M is a TM, there also has to be an index $j \in \mathbb{N}^+$ such that $M_j = M$. Now we ask whether M_j halts on the word w_j. By the design of M, we get the equivalence

$$M_j \text{ halts on } w_j \iff M \text{ does not halt on } w_j \,,$$

which is a direct contradiction to $M_j = M$; hence H^* cannot exist, and consequently L_{H} is not recursive. □

4.6 Rice's Theorem

While one may argue that L_{diag} is a rather artificial problem, the undecidability of the halting problem seems to have quite some impact on practice. In this section, we prove an even more general result, namely that almost all problems that deal with the semantics of TMs are undecidable. To this end, we need the formal definition of a property of a recursively enumerable language.

Definition 4.16 (Property of Recursively Enumerable Languages). *A **property of the recursively enumerable languages** $\mathcal{E} \subseteq \mathcal{L}_{\text{RE}}$ is any collection of recursively enumerable languages. $\mathcal{E}$ is called **nontrivial** if both $\mathcal{E} \neq \emptyset$ and $\mathcal{E} \neq \mathcal{L}_{\text{RE}}$, i.e., $\mathcal{E}$ neither contains no recursively enumerable language nor all of them.*

As a consequence of Definition 4.16, we can define intuitive properties by a collection of the corresponding recursively enumerable languages; e.g., the property of "being a regular language" is formalized by $\mathcal{E}_{\text{reg}} = \mathcal{L}_{\text{reg}}$; the property of "being empty" is formalized by $\mathcal{E}_{\text{empty}} = \{\emptyset\}$. For every such property $\mathcal{E}$, let $\text{TM}(\mathcal{E})$ denote the set of all encodings of TMs that accept languages in $\mathcal{E}$; e.g., $\text{TM}(\mathcal{E}_{\text{reg}})$ contains encodings of exactly those TMs that accept a regular language. The following theorem states that it is undecidable whether a given TM has a language with a given nontrivial property.

Theorem 4.17 (Rice's Theorem). *For every nontrivial property of the recursively enumerable languages $\mathcal{E}$, $\text{TM}(\mathcal{E})$ is not recursive.*

Proof. Consider a language $L \in \mathcal{E}$ with a TM E with $\text{Code}(E) \in \text{TM}(\mathcal{E})$, i.e., E is a TM with $L = \text{Lang}(E) \in \mathcal{E}$. We give a reduction from L_{U} to $\text{TM}(\mathcal{E})$; due to Theorem 4.13, the claim then follows. For now, let us assume that $\emptyset \notin \mathcal{E}$. Consider an input $\text{Code}(M)111w$ of L_{U}. We design a TM M' that has $\text{Code}(E)$, $\text{Code}(M)$, and w built into its transitions, and, at the beginning of its computation, writes $\text{Code}(M)$ and w (encoded as in the proof of Theorem 3.4) onto its tapes. Let w' denote the input of M'; M' works as follows.

- First, M' simulates M on w while completely ignoring its own input w'. If M rejects w, then M' also rejects; if M runs forever on w, then M' also runs forever.
- If M accepts w, then M' simulates E on the input w'. If E accepts w', then M' also accepts; if E rejects w', then M' rejects; if E runs forever, then M' also runs forever.

We consequently get the following implications.

- If $w \notin \text{Lang}(M)$, then the simulation of M on w either does not halt or M rejects w. In this case, M' does not halt or it rejects. This means that M' does not accept its input w' (independently of which word w' actually is). Thus, $\text{Lang}(M') = \emptyset$, which is not in $\mathcal{E}$.
- If $w \in \text{Lang}(M)$, then M accepts w, and thus M' continues with simulating E on its input w'. Thus, in this case, M' accepts exactly the language L of E.

This yields the equivalence

$$w \in \mathrm{Lang}(M) \iff \mathrm{Lang}(M') = L$$

and since M' is a TM for $L \in \mathcal{E}$,

$$w \in \mathrm{Lang}(M) \iff \mathrm{Code}(M') \in \mathrm{TM}(\mathcal{E}) \; .$$

For a contradiction, now suppose $\mathrm{TM}(\mathcal{E})$ were decidable and consider a TM M^* which decides $\mathrm{TM}(\mathcal{E})$. We supply the encoding of M' as input to M^*. If M^* answers that $\mathrm{Code}(M') \in \mathrm{TM}(\mathcal{E})$, we know that the simulation of M on w, which was carried out by M', must have resulted in M accepting w; thus $w \in \mathrm{Lang}(M)$ and hence $\mathrm{Code}(M)111w \in L_{\mathrm{U}}$. Conversely, if M^* answers that $\mathrm{Code}(M') \notin \mathrm{TM}(\mathcal{E})$, it immediately follows that M does not accept w and therefore $\mathrm{Code}(M)111w \notin L_{\mathrm{U}}$.

Now we assume that $\emptyset \in \mathcal{E}$. In this case, consider the property $\overline{\mathcal{E}}$; obviously, $\emptyset \notin \overline{\mathcal{E}}$, and hence the above proof shows that $\mathrm{TM}(\overline{\mathcal{E}})$ is not recursive. The language of any given TM is either in $\mathcal{E}$ or in $\overline{\mathcal{E}}$. Therefore, if the encoding of any TM is not in $\mathrm{TM}(\mathcal{E})$, it has to be in $\mathrm{TM}(\overline{\mathcal{E}})$, which means that $\overline{\mathrm{TM}}(\mathcal{E}) = \mathrm{TM}(\overline{\mathcal{E}})$. From $\overline{\mathrm{TM}}(\mathcal{E})$ not being recursive, it immediately follows that $\mathrm{TM}(\mathcal{E})$ is also not recursive due to Theorem 4.9. □

A consequence of Theorem 4.17 is that it is impossible in general for a given TM M to decide whether the language $\mathrm{Lang}(M)$ of M

- is regular (or context-free or context-sensitive),
- is empty,
- is finite,
- contains a given word (or does not contain it), or
- only contains words that start with a given prefix (or end with a given suffix).

As stated earlier, Rice's theorem speaks about the semantics of TMs, i.e., about the languages they accept. Syntactical questions such as "does a given TM have more than five states?" are decidable; "does a given TM reach its fifth state with a given word as input?" is not.

4.7 Historical and Bibliographical Notes

As already mentioned, Turing machines were introduced by Turing [27] in 1936. In this publication, he also showed that there is a language that is not recursively enumerable (Theorem 4.8). As also mentioned earlier, Gödel proved his incompleteness theorem in 1931 [10], using an approach similar to that in the proof of that theorem, also applying Cantor's diagonal argument. In the original proof, Turing assumed the machines to accept words by halting; thus there was no distinction between halting in an accepting or nonaccepting state; we described that both models are equivalent in Section 4.5. It is a different formulation to the halting problem as defined in Section 4.5.

Rice's theorem (Theorem 4.17) was proven by Rice in 1953 [26].

5 Intractability

In the previous chapter, we have learned about decidable problems, i.e., those problems which can be solved automatically. A problem is called decidable (or recursive) if it can, in principle, be solved by an algorithm. Now we would like to take a closer look at this "in principle." Indeed, we have so far neglected completely how hard it is to compute a solution in terms of the resources used; such resources are, e.g., time and space. In this chapter, we will focus on the former one, i.e., we will investigate how much time is used in order to solve particular decision problems. Depending on the concrete bounds, we will distinguish between problems that are **tractable** (which means solvable in practice) and those that are **intractable** (which means decidable, thus solvable theoretically, but unsolvable in practice since this would require too much time also for reasonably small instances).

We have so far not considered the running time of TMs at all. We start by defining the **running time of a TM M on a word x**, denoted by $\mathrm{Time}_M(x)$, which is simply the number of moves M makes before halting; the TM M_{ab} (see Figure 16), e.g., has a running time of 13 on the word *aabb*. In a way, it is sufficient to count the $\vdash s$ in a given computation (if we are able to write it down explicitly). What we are particularly interested in is the worst-case running time of M, i.e., a guarantee on the number of moves M makes on a given word. For sure, the running time will usually increase with the length of the input word (there are exceptions, such as deciding whether the given word starts with a particular letter), but it makes a huge difference how large this growth is. We define the **time complexity of M** with input alphabet Σ as a function that gives, for every $n \in \mathbb{N}$, the maximum number of moves M makes on words of length n, i.e.,

$$\mathrm{Time}_M(n) := \max\{\mathrm{Time}_M(x) \mid x \in \Sigma^n\} .$$

To classify TMs with respect to their running time, we use the **big $\mathcal{O}$ notation**; i.e., we speak about their **asymptotic time complexities**. Let f and g be two positive functions in a variable n; to make our arguments simpler, we will write $f(n)$ ($g(n)$, respectively) instead of f (g, respectively). Recall that $f(n)$ is in $\mathcal{O}(g(n))$ if, for values larger than some fixed threshold, $f(n)$ grows faster than $g(n)$ only by a fixed constant; more formally

$$f(n) \in \mathcal{O}(g(n)) \iff \exists n_0, k \text{ such that } \forall n \geq n_0 \text{ we have } f(n) \leq k \cdot g(n) .$$

A few examples are $n^2 \in \mathcal{O}(n^3)$, $3n^2 + 5n \in \mathcal{O}(n^3)$, $n^3 \in \mathcal{O}(n!)$, $n^4 \in \mathcal{O}(n^4)$, $\sqrt{n} \in \mathcal{O}(n)$, $10\,000 \cdot n^{100} \in \mathcal{O}(2^n)$, and $\log_2 n \in \mathcal{O}(n)$. We then say that a TM M has a time complexity in $\mathcal{O}(g(n))$ if $\mathrm{Time}_M(n) \in \mathcal{O}(g(n))$; of course, we are interested in functions g that give a bound which is as tight as possible, i.e., the smaller g grows, the more meaningful the statement becomes.

5.1 The Class $\mathcal{P}$

In particular, we are interested in TMs with polynomial time complexities, i.e., those TMs M with $\mathrm{Time}_M(n) \in \mathcal{O}(n^d)$ for a fixed (i.e., independent of n) $d \in \mathbb{N}$. Decision problems that can be decided by such TMs are called **tractable**, **efficiently solvable**, or simply **solvable in polynomial time**.

Definition 5.1 (The Class $\mathcal{P}$). *A language L is called* ***tractable*** *if there is a TM M with $\text{Lang}(M) = L$ and $\text{Time}_M(n) \in \mathcal{O}(n^d)$ for some fixed $d \in \mathbb{N}$, and we say that L is solvable in polynomial time. The class of tractable problems is*

$$\mathcal{P} = \{L \mid L \text{ is solvable in polynomial time}\} .$$

Note that, by definition, $\mathcal{P} \subseteq \mathcal{L}_{\text{R}}$ since our definition of "time complexity" assumes that M always halts after $\text{Time}_M(n)$ moves, independent of whether the input is accepted or not. As an alternative, we could have defined that M has time complexity $\text{Time}_M(n)$ if it makes that many moves on all words in its language; in this case, M could make more moves if it rejects or it may even not halt at all. However, for such a TM M, we could then design an MTM M' that simulates M on any given input x and keeps track of the number of moves M makes on x. As soon as M' detects that M would make more than $\text{Time}_M(n)$ steps on x, it "knows" that M rejects x or runs forever; in this case, M' rejects x. We can then convert M' to a TM M'' that has a time complexity which is polynomial in that of M; thus $\mathcal{P}$ contains exactly the same decision problems with this definition.

One could argue that allowing for any polynomial function in order to call a problem tractable is very coarse and unsatisfying; e.g., it seems very doubtful why we should call a TM with a time complexity of n^{1000} efficient. Why not bound the time complexity from above more concretely, e.g., by defining tractable problems to be those that admit TMs with a time complexity in $\mathcal{O}(n^4)$? There are good reasons why this is not done.

- **Equivalence of different TM models.** In Section 3.1, we considered different restrictions and extensions of Turing machines, and argued that the expressive power of all of them is the same. However, converting, e.g., an MTM into a TM with a single tape both increases the size of the TM and its time complexity.

  It can be shown that converting any model of a TM to any other model only increases the time complexity polynomially. This allows us to switch between them in order to prove lower and upper bounds on the time complexity of TMs for a given decision problem L. If, e.g., we want to show that L is tractable, it is sufficient to design an MTM M for L with polynomial time complexity; it then follows that there is a TM M' for L with polynomial time complexity, as well. The degree of the time complexity of M' may, however, be a lot larger than that of M. Conversely, if we want to show that L does not admit a efficient TM, it is sufficient prove that there is no such TM according to the most simple model; this directly implies that there neither is any efficient more general TM.

  Actually, we already made use of this fact when we discussed the alternative definition of time complexity above.

- **Equivalence to real computers.** With arguments more involved than those above, it can be argued that a TM can also simulate a "real computer." Each computational step of the computer takes a polynomial number of moves of the TM. We will not describe the details of the proof here.

To prove for a given decision problem L that it is contained in $\mathcal{P}$, it thus suffices to design an algorithm for L in some fixed programming language (or even pseudocode).

- **Closure properties.** The class of polynomial functions is closed under many important operations; we say that polynomials are "robust." This is crucial to build up the complexity theory which we will introduce in Section 5.3.
  - The sum $p_1 + p_2$ of two polynomials p_1 and p_2 yields another polynomial; e.g., $(2n^3 + n^2) + (4n^5 + 7n^2 + 11) = 4n^5 + 2n^3 + 8n^2 + 11$.
  - The multiplication $p_1 \cdot p_2$ of two polynomials p_1 and p_2 also yields a polynomial; e.g., $(6n^2 + 3n) \cdot (n^3 + 2) = 6n^5 + 3n^4 + 12n^2 + 6n$.
  - Plugging a polynomial p_1 into a polynomial p_2 (thus replacing every variable n of p_2 by p_1) also yields a polynomial $p_2(p_1)$; e.g., $p_1 = n^2+3n$ and $p_2 = 5n^3+7n^2+3n$ leads to $5(n^2 + 3n)^3 + 7(n^2 + 3n)^2 + 3(n^2 + 3n) = 5n^6 + 45n^5 + 142n^4 + 177n^3 + 66n^2 + 9n$.
- **Practical relevance.** Experience of many decades has shown that it is indeed reasonable to call problems tractable if there are algorithms (likewise, TMs that always halt) for them which work in polynomial time. Usually, if a problem arising from a practical application admits an algorithm with polynomial time complexity, we are also able to find such an algorithm with a rather small exponent such as n^3 or n^4. Thus, problems in $\mathcal{P}$ are those that we can handle well with computers. Conversely, problems that do not admit polynomial-time algorithms (or at least do not seem to) and thus (seem to) require algorithms with a time complexity asymptotically larger than a polynomial function cannot be solved in meaningful time.

5.2 Nondeterministic Turing Machines and the Class $\mathcal{NP}$

We have encountered the principle of nondeterminism in the context of finite automata (Section 1.3) and pushdown automata (Section 2.6); now we introduce nondeterminism for TMs. **Nondeterministic TMs** (NTMs for short) are defined in the obvious way, i.e., the components Q, Σ, Γ, δ, q_0, $\sqcup$, and F are defined as in the deterministic case (i.e., as in Definition 3.1) with the only difference that δ is not a function that maps tuples of states and letters to single triples of states, letters, and directions, but to sets of such triples. Such a set may again be empty, contain a single triple, or all of them. Generally, NTMs do not have to halt on all computations, but only on accepting ones. Analogously to MTMs, there are NMTMs, which can be simulated by NTMs with a single tape.

For an NTM $N = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$, we can define the $\vdash$-relation similar to deterministic TMs; however, a move $\alpha q \beta \vdash \gamma p \mu$ with $p, q \in Q$ and $\alpha, \beta, \gamma, \mu \in \Gamma^*$ now incorporates a nondeterministic guess in the transition from the first configuration to the second one; the $\vdash^*$-relation is defined accordingly. The language of N is then defined analogously to that of deterministic TMs as

$$\mathrm{Lang}(N) = \{w \in \Sigma^* \mid q_0 w \vdash^* \alpha p \beta \text{ with } \alpha, \beta \in \Gamma^* \text{ and } p \in F\} .$$

We have seen that DFAs and NFAs are equivalent with respect to their expressive power. Now we prove that the same is true for TMs and NTMs.

Theorem 5.2. *Every TM can be converted into an equivalent NTM.*

Proof. We can use the exact same arguments as in the proof of Theorem 1.5, and regard any given TM as an NTM that makes no nondeterministic guesses at all. □

Theorem 5.3. *Every NTM can be converted into an equivalent TM.*

Proof. This implication is again not as straightforward as the opposite direction. Similarly to the proof of Theorem 1.6, let $N = (Q_N, \Sigma_N, \Gamma_N, \delta_N, q_{0,N}, \sqcup, F_N)$ be an NTM with $\text{Lang}(N) = L$, and let us design a TM $M = (Q_M, \Sigma_M, \Gamma_M, \delta_M, q_{0,M}, \sqcup, F_M)$ with $\Sigma_M = \Sigma_N$ and $\text{Lang}(M) = L$. Without loss of generality, we assume that the input alphabets are binary. As opposed to converting NFAs to DFAs, it is now not sufficient to mimic being in a number of states at the same time. We can, however, follow an idea that is still somewhat similar to the powerset construction; the key is to consider the different configurations in which N can be after a fixed number of moves.

M is a 4-MTM which, as described in Section 4.3, first writes the encoding of N onto its second working tape. M then copies its input x from the input tape to its third working tape, converting it to the same encoding as in the encoding of N (i.e., 0 is encoded as 0 and 1 as 00, and 1s separate single letters). The fourth tape is used for auxiliary computations. M now simulates N on x as follows.

- First, M writes the initial configuration of N on x on its first tape. By concatenating a $*$ to this configuration it is marked as the current configuration. It positions the head on the third working tape on the first letter of x.

- Depending on the current state of N (indicated on the first tape), and the currently scanned letter (indicated on the third tape), M searches a corresponding transition on the first tape.

  If the current state is an accepting state, M also moves to an accepting state.

  If no transition is found, N gets stuck, and therefore also M gets stuck by entering a state without outgoing transitions.

  If a transition is found, let r denote the number of different choices, out of which one is chosen nondeterministically. M writes all possible successor configurations to the first working tape. This is done by copying the current configuration r times to the end of the tape and then modifying them to realize the r choices.

- After that, M moves the tape head of its first working tape back to the current configuration, removes the $*$ and writes it behind the next configuration.

- This is iterated until either an accepting state is found in the current configuration, in which case M moves to an accepting state, or N gets stuck, in which case M gets stuck as well.

Suppose that N accepts x after n moves, and suppose that, for any transition, N has a choice between at most m possibilities. This means that there are m different configurations in which N can be after one move, m^2 configurations in which it can be after two moves, and so on. In the sum this makes

$$\sum_{i=1}^{n} m^i \leq nm^n$$

configurations which M needs to write down. However, this means that M accepts after finite time. Essentially, M performs a breadth-first search on the configuration tree of M on x. It follows that M accepts x if and only there is an accepting computation of N of x, i.e., if and only if N accepts x. □

Theorem 5.3 proves that NTMs and TMs have the same expressive power. Then again, we note that M may be a lot larger than N, and furthermore n moves of N translate to at most nm^n moves of M, which may thus take exponentially more time. Of course, this may be a very pessimistic bound.

Definition 5.4 (The Class $\mathcal{NP}$). *A language L is called **solvable in nondeterministic polynomial time** if there is an NTM N with $\mathrm{Lang}(N) = L$ and $\mathrm{Time}_N(n) \in \mathcal{O}(n^d)$ for some fixed $d \in \mathbb{N}$ and N always halts after at most $\mathrm{Time}_N(n)$ moves. We say that L is solvable in nondeterministic polynomial time, and the corresponding class is*

$$\mathcal{NP} = \{L \mid L \text{ is solvable in nondeterministic polynomial time}\} .$$

Note that, as the NTM in the above definition is guaranteed to halt, we trivially have $\mathcal{NP} \subseteq \mathcal{L}_{\mathrm{R}}$, i.e., all problems in $\mathcal{NP}$ are decidable.

Some of the problems we study in what follows are defined on graphs, for which we want to decide whether they have some certain properties. At all times, with "graph" we mean both an undirected and unweighted graph.

Definition 5.5 (Independent Set). *Let $G = (V, E)$ be a graph. An **independent set** in G is any set of vertices I such that no vertices from I are connected by an edge.*

Figure 26a shows an independent set of size four in a graph with nine vertices. The counterpart of independent sets are called cliques, which are set of vertices that are all pairwise connected; a clique of size four is shown in Figure 26b.

Definition 5.6 (Clique). *Let $G = (V, E)$ be a graph. A **clique** in G is any set of vertices C such that any two vertices from C are connected by an edge.*

The independent set problem is, given a graph G and an integer k, to find out whether G contains an independent set of size k (or larger). As for the Hamiltonian cycle problem (we simply write "HC" instead of L_{HC}), which we defined at the very beginning in Section 1.1, G is encoded by its adjacency matrix which is a word over Σ_{graph}. The value of k is written

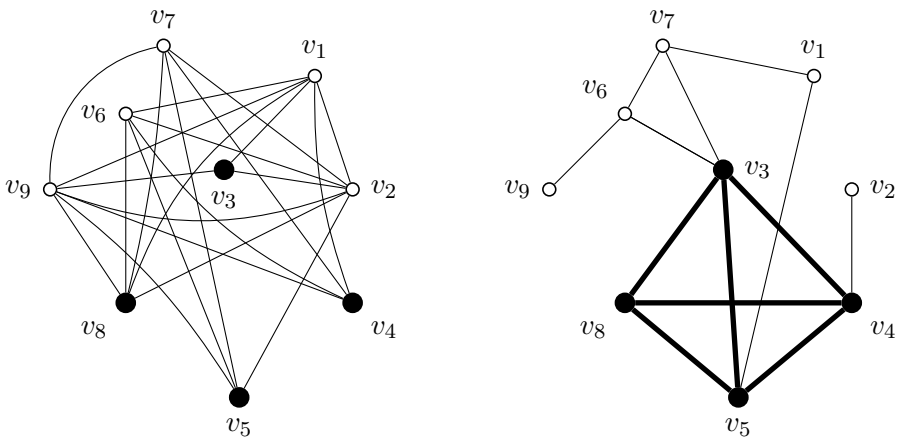


Figure 26.

down in binary as a word over Σ_{bin}. The graph from Figure 26a and the integer 4 would then be encoded as

$$x = 011101011\#101011111\#110000001\#100001101\#010001101\#110110010\#$$
$$010110011\#110001101\#111110110\#\#11 \ .$$

In order to keep the notation simple, we will simply write "(G, k)" as an input of IND-SET and similar problems, just as we wrote "$(\text{Code}(M), w)$" instead of $\text{Code}(M)111w$ as inputs of L_{U}. Formalized as a decision problem (we again write "IND-SET" instead of $L_{\text{IND-SET}}$), we get

$$\text{IND-SET} = \{(G, k) \mid G \text{ contains an independent set of size } k\} \ .$$

Theorem 5.7. IND-SET $\in \mathcal{NP}$.

Proof. We show IND-SET $\in \mathcal{NP}$ constructively by designing an NTM $N_{\text{IND-SET}}$ that decides, for a given input $x = (G, k)$ to IND-SET, whether $x \in$ IND-SET or $x \notin$ IND-SET and that has a polynomial time complexity. Let m be the number of vertices of G, i.e., G is represented by an adjacency matrix of size $m \times m$. Thus, the encoding of G has a length of $m^2 + (m-1)$, accounting for the #s, then there are two #s and the binary encoding of k, which has length $\lceil \log_2(k+1) \rceil$; hence $|x| = m^2 + m + 1 + \lceil \log_2(k+1) \rceil$. Obviously, $k \leq m$ for any feasible input.

- $N_{\text{IND-SET}}$ first scans the input until it encounters two consecutive #s. It then reads the succeeding bit string and interprets it as an integer k. This needs $\mathcal{O}(|x|)$ moves.
- Next, $N_{\text{IND-SET}}$ writes down k distinct integers between 1 and m in binary. This is done nondeterministically, i.e., $N_{\text{IND-SET}}$ is designed such that there is a run for every possible sequence of k such numbers. Every number can be encoded with $\lceil \log_2(m+1) \rceil$ bits and can thus be written down with $\mathcal{O}(\log m)$ moves. Writing down all numbers can therefore be done within $\mathcal{O}(k \log m) \subseteq \mathcal{O}(m^2) \subseteq \mathcal{O}(|x|)$ moves.

- The k integers are now interpreted as the indices of k vertices of G. Due to the design of $N_{\text{IND-SET}}$, there may be any such sequence written down on $N_{\text{IND-SET}}$'s tape. If G indeed has an independent set I of size k, there is a run of $N_{\text{IND-SET}}$ such that I is also written down; if there is no such independent set, every run leads to a sequence of indices such that at least two of the corresponding vertices are connected by an edge.

  Therefore, $N_{\text{IND-SET}}$ now checks whether the k indices indeed encode an independent set. It does so by looking up the corresponding entries in the adjacency matrix of G. For a single index, it needs to consider $k-1$ cells, which can be done with $\mathcal{O}(m^2)$ moves. If, at any time a 1 is found in a cell that is checked, $N_{\text{IND-SET}}$ immediately halts and rejects. Checking all cells can be done within $\mathcal{O}(km^2) \subseteq \mathcal{O}(|x|^2)$ moves. If all cells contain a 0, $N_{\text{IND-SET}}$ finally accepts.

Since $N_{\text{IND-SET}}$ only accepts x if it guesses nondeterministically a sequence of m vertices that form an independent set in G, we have the equivalence

$$x \in \text{IND-SET} \iff \text{there is a run of } N_{\text{IND-SET}} \text{ in which it accepts } x\ ,$$

and therefore $L(N_{\text{IND-SET}}) = \text{IND-SET}$. Since $N_{\text{IND-SET}}$ makes a number of moves that is polynomial in $|x|$ and always halts, we have $\text{Time}_N(n) \in \mathcal{O}(n^d)$, for some $d \in \mathbb{N}$, and the claim follows. □

By arguments very similar to those used in the proof of Theorem 5.7, we can also show that HC $\in \mathcal{NP}$. Here, an NTM first nondeterministically guesses a permutation of the vertices of the input graph and then checks whether it encodes a Hamiltonian cycle; if so, it accepts, otherwise it rejects.

5.3 Polynomial-Time Reductions and $\mathcal{NP}$-Completeness

The class $\mathcal{NP}$ contains all decision problems L for which we can nondeterministically decide in polynomial time whether $x \in L$ or $x \notin L$ for any given x. Another point of view is that we can **verify** in polynomial time that an input x is indeed a "yes" instance of L. Suppose that we are dealing with IND-SET, but instead of only receiving an input x to the problem, we also obtain a so-called **witness** as to whether $x \in$ IND-SET, which is a binary word that can be used to verify this fact. In this case, the witness is simply an independent set of size k in G. We can then design a "verifier" $M_{\text{IND-SET}}$ that works similarly to $N_{\text{IND-SET}}$ from the proof of Theorem 5.7. Instead of making nondeterministic guesses, $M_{\text{IND-SET}}$ uses the witness to verify that an independent set of size m exists. Since $M_{\text{IND-SET}}$ works in polynomial time, we call it a **polynomial-time verifier**. This can be done for all problems in $\mathcal{NP}$, i.e., $\mathcal{NP}$ contains exactly those decision problems for which we can verify in polynomial time whether a given instance is a "yes" instance.

It makes sense to restrict ourselves to such decision problems if we are looking for those that are solvable in polynomial time. In other words, if there is no NTM (or equivalently, polynomial-time verifier) that solves a problem in polynomial time, it is hopeless to search for a TM with polynomial time complexity for this problem. But are TMs and NTMs even

equally powerful not only with respect to their expressive power but also with respect to the time they take (i.e., only different by a polynomial)?

Giving an answer to this question seems far from trivial. Up to today we do not know any nontrivial (i.e., beyond linear) lower bound on the time complexity necessary to solve any of the problems in $\mathcal{NP}$. This means that we have no clue whether any of them need, e.g., exponential running time. An idea to shed some light on this question is to identify a class of "hardest" problems within $\mathcal{NP}$. Also for those we do not have any nontrivial bounds so far, but at least we know that these problems are good candidates to search among. This approach leads to a concept of "relative hardness" in that the other problems in $\mathcal{NP}$ are not harder than those. This is formalized by the fact that, if at any point someone would find a polynomial-time algorithm for any of these hard problems, we would immediately have polynomial-time algorithms for all problems in $\mathcal{NP}$. The hardness of solving problems in $\mathcal{NP}$ efficiently can thus be "reduced" to efficiently solving the hard problems. To prove that a problem is hard in this sense, we therefore use a special kind of reduction. A **polynomial-time reduction** establishes a connection between L_1 and a second language L_2 implying that

> if L_2 were decidable in polynomial time,
> then L_1 would be decidable in polynomial time, too.

The formal definition of such a reduction is very similar to that of Definition 4.11 with the only difference that we demand that it can be carried out in polynomial time.

Definition 5.8 (Polynomial-Time Reduction). *Let L_1 and L_2 be two decision problems. If there is a TM P with $\mathrm{Time}_P(n) \in \mathcal{O}(n^d)$, for some $d \in \mathbb{N}$, which converts inputs x of L_1 to inputs $y = P(x)$ of L_2 such that*

$$x \in L_1 \iff y \in L_2 \,,$$

*we say that there is a **polynomial-time reduction from L_1 to L_2**, denoted by $L_1 \leq_{\mathsf{p}} L_2$.*

With the concept of a polynomial-time reduction, we have a formal way to express that a given decision problem is "as hard as" another given decision problem with respect to the ability to solve the problems in polynomial time.

Theorem 5.9. *Let L_1 and L_2 be two languages, and suppose $L_1 \leq_{\mathsf{p}} L_2$. If L_1 is not decidable in polynomial time, then neither is L_2.*

Proof. The idea of the proof is essentially that of the proof of Theorem 4.12; there is only a new aspect, namely the time it takes to perform the reduction. Since there is a polynomial-time reduction from L_1 to L_2, there is a TM P with polynomial time complexity that converts inputs to L_1 to inputs to L_2 as in Definition 5.8.

For a contradiction, suppose L_2 were decidable in polynomial time (but not L_1), and let M_2 be a TM for this task. A TM M_1 for L_1 takes its input x and converts it to an

input y to L_2 using P; this can be done in time $p_1(|x|)$, where p_1 is a polynomial function. Since P made a polynomial number of moves, the output y it created also has a length polynomial in $|x|$; specifically, $|y| \leq p_1(|x|)$. M_1 now simulates M_2 on y, and since M_2 works in polynomial time with respect to its input, it also works in polynomial time with respect to $|x|$. More formally, there is a polynomial function p_2 such that M_2 works in time $p_2(|y|) \leq p_2(p_1(|x|))$. M_2's answer as to whether $y \in L_2$ or $y \notin L_2$ is then also correct for the question whether $x \in L_1$ or $x \notin L_1$. Consequently, M_1 decides L_1 in polynomial time (bounded from above by $p_1(|x|) + p_2(p_1(|x|))$), which is a contradiction to our assumptions, and thus L_2 cannot be decidable in polynomial time. □

Now we can use this kind of reduction in order to define the aforementioned hardest class of problems (within $\mathcal{NP}$) such that, if one of them can be solved in polynomial time, all problems in $\mathcal{NP}$ can.

Definition 5.10 ($\mathcal{NP}$-Hardness and $\mathcal{NP}$-Completeness). *A decision problem L is called **$\mathcal{NP}$-hard** if, for every $L' \in \mathcal{NP}$, we have $L' \leq_\mathsf{p} L$. L is called **$\mathcal{NP}$-complete** if*

- *$L \in \mathcal{NP}$ and*
- *L is $\mathcal{NP}$-hard.*

The concept of polynomial-time reductions allows us to show that a problem is $\mathcal{NP}$-complete by reducing it to another $\mathcal{NP}$-complete problem. However, we need a first problem to start with.

5.4 Cook's Theorem

Similar to L_{diag}, which we used as a first problem that is not recursive, and which then allowed us to prove other problems not to be recursive as well by reductions, we now need an $\mathcal{NP}$-complete problem that plays the same role in complexity theory. In other words, for this particular problem, we need to show that every problem in $\mathcal{NP}$ reduces to it; i.e., if this one problem can be solved in polynomial time, then all problems in $\mathcal{NP}$ can, and thus $\mathcal{P} = \mathcal{NP}$ would follow.

Inputs of the problem are given as specific formulae.

Definition 5.11 (Boolean Expression and Conjunctive Normal Form). *Boolean variables are variables that are either 0 or 1. A **literal** is either a Boolean variable x or its negation (denoted by $\overline{x}$). The "or" (denoted by $\vee$) of one or more literals is called a **clause**; the "and" (denoted by $\wedge$) of one or more clauses is called a **Boolean expression in conjunctive normal form**.*

For simplicity, we abbreviate conjunctive normal form by CNF, not to be confused ChNF, which is short for Chomsky normal form. A **truth assignment φ for E** assigns a value of 0 or 1 to each variable of a Boolean expression E; by $\varphi(E)$ we denote the value of

E if the values of E's variables are assigned according to φ. As an example, consider the Boolean expression

$$E = (x_1 \vee \overline{x}_3) \wedge (\overline{x}_2 \vee x_3 \vee x_4) \wedge (\overline{x}_1 \vee \overline{x}_3 \vee \overline{x}_4) \wedge (x_2 \vee \overline{x}_3) \wedge (x_4)$$

in CNF together with the truth assignment $\varphi(x_1) = 1$, $\varphi(x_2) = 0$, $\varphi(x_3) = 0$, and $\varphi(x_4) = 1$, which yields

$$\varphi(E) = (1 \vee 1) \wedge (1 \vee 0 \vee 1) \wedge (0 \vee 1 \vee 0) \wedge (0 \vee 1) \wedge (1) \ ,$$

and hence $\varphi(E) = 1$ or "φ makes E true." If such a truth assignment satisfying all clauses exists for E, we call E **satisfiable**. The **satisfiability problem**, SAT for short, is, given a Boolean expression E in CNF, to decide whether it is satisfiable, i.e.,

$$\text{SAT} = \{E \text{ is a Boolean expression in CNF} \mid \exists\, \varphi \text{ with } \varphi(E) = 1\} \ .$$

An input for SAT can have an arbitrary number of literals. Since the alphabet size needs to be fixed and independent of the input, we need to be careful with the encoding at this point. An easy way is to consider the alphabet $\Sigma_{\text{Bool}} = \{x, 0, 1, (,), \neg, \wedge, \vee\}$ and encode the variable x_i by "$x\text{Bin}(i)$," where $\text{Bin}(i)$ is the binary encoding of i. The expression $(x_1 \vee \overline{x}_2 \vee x_5) \wedge (x_2 \vee x_3 \vee \overline{x}_6) \wedge (\overline{x}_3 \vee \overline{x}_5 \vee x_7)$ is then encoded as

$$(x1 \vee \neg x10 \vee x101) \wedge (x10 \vee x11 \vee \neg x110) \wedge (\neg x11 \vee \neg x101 \vee x111) \ .$$

With SAT, we now have a first $\mathcal{NP}$-complete problem, i.e., SAT is very general in the sense that every problem in $\mathcal{NP}$ reduces to it; it is consequently our basic building block for the theory of $\mathcal{NP}$-completeness.

Theorem 5.12 (Cook's Theorem). SAT *is $\mathcal{NP}$-complete.*

Proof. According to Definition 5.10, proving a problem to be $\mathcal{NP}$-complete means to show both that it is in $\mathcal{NP}$, and that every problem in $\mathcal{NP}$ reduces to it in polynomial time. Showing the former is rather simple. An NTM M_{SAT} for SAT simply guesses a truth assignment φ of the given Boolean expression E and then checks whether $\varphi(E) = 1$. This can obviously be done in time polynomial in the length of E.

Now it remains to show that every problem in $\mathcal{NP}$ reduces to SAT in polynomial time. Let L be any decision problem in $\mathcal{NP}$; we show $L \leq_{\mathsf{p}} \text{SAT}$. Since $L \in \mathcal{NP}$, according to Definition 5.4, there is an NTM $N = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$ with $\text{Lang}(N) = L$ and $\text{Time}_N(|x|) \leq p(|x|)$ for a polynomial function $p(|x|)$ and any word $x \in \Sigma^*$. To keep our arguments simple, we assume that N with $Q = \{q_0, q_1, \ldots, q_{l-1}\}$ and $\Gamma = \{X_1, X_2, \ldots, X_z\}$ is an NTM that

- has one semi-infinite tape,
- has a tape alphabet with $X_z = \sqcup$,
- is not allowed to make stationary moves with its tape head, and

- has one accepting state; without loss of generality, we assume that this is state q_{l-1}.

We now show how to convert N and a given input word $x \in \Sigma^*$ to a Boolean expression $E_N(x)$ in CNF such that the equivalence

$$x \in \mathrm{Lang}(N) \iff E_N(x) \text{ is satisfiable}$$

holds.

The initial configuration of N on $x = x_1 x_2 \dots x_m$ is $q_0 x_1 x_2 \dots x_m$, and generally a configuration is given by

$$Y_1 Y_2 \dots Y_{i-1} q Y_i \dots Y_n \ .$$

Since the running time of N on x is bounded from above by $p(|x|)$ and in one move of N its tape head can move at most one cell, we immediately get[3]

$$n \leq \max\{|x|+1, p(|x|)+1\} = p(|x|)+1 \ ,$$

and consequently at most $p(|x|)+1$ cells of N's tape contain a symbol different from $\sqcup$ when N is given x. To make the following arguments simpler, we pad all configurations by blanks such that each of them has a length of exactly $p(|x|)+2$, i.e., exactly the first $p(|x|)+1$ cells of the tape are considered. Furthermore, we modify N such that it loops in q_{l-1} whenever it reaches this state. This way, we get

$$N \text{ accepts } x \iff q_{l-1} \text{ appears in the } (p(|x|)+1)\text{th configuration} \ .$$

A configuration of N on x is uniquely defined by three parameters, namely the current state, head position, and tape content. For each of them, there will be a class of Boolean variables that make up $E_N(x)$.

- **Current state.** There are $l \cdot (p(|x|)+1)$ Boolean variables $s_{k,t}$ with $0 \leq k \leq l-1$ and $0 \leq t \leq p(|x|)$ such that

  $$s_{k,t} = 1 \iff N \text{ is in state } q_k \text{ after its } t\text{th move.}$$

  Here and subsequently, "after the 0th move" means that N made no move yet.

- **Current head position.** There are $(p(|x|)+1) \cdot (p(|x|)+1)$ Boolean variables $p_{i,t}$ with $1 \leq i \leq p(|x|)+1$ and $0 \leq t \leq p(|x|)$ such that

  $$p_{i,t} = 1 \iff N'\text{s head is scanning the } i\text{th tape cell after its } t\text{th move.}$$

- **Current tape content.** There are $(p(|x|)+1) \cdot z \cdot (p(|x|)+1)$ Boolean variables $c_{i,j,t}$ with $1 \leq i \leq p(|x|)+1$, $1 \leq j \leq z$ and $0 \leq t \leq p(|x|)$ such that

  $$c_{i,j,t} = 1 \iff N'\text{s } i\text{th tape cell contains } X_j \text{ after its } t\text{th move.}$$

[3] Without loss of generality, we assume $p(|x|) \geq |x|$.

$E_N(x)$ is composed of four subformulae that ensure that $E_N(x)$ is only satisfiable if there is an accepting computation of N on x, i.e., a sequence of configurations that we consider to be of length $p(|x|)+2$ each.

- **Uniqueness.** At first, we have to make sure that $E_N(x)$ indeed corresponds to a sequence of valid configurations of N on x. To this end, we have to ensure that in each of the $p(|x|)+1$ time steps (i.e., after each number of possible moves), N is in exactly one state, scans exactly one tape cell, and on each cell there is written exactly one letter from Γ; it must, e.g., be forbidden that $s_{0,3}$ and $s_{1,3}$ are both true since N cannot be in the states q_0 and q_1 simultaneously after it made three moves.

  For a fixed t with $0 \leq t \leq p(|x|)$, exactly one of the variables $s_{k,t}$ has to be 1 for exactly one k with $0 \leq k \leq l-1$; i.e., $E_N(x)$ will not be satisfiable if all such $s_{k,t}$ are 0 or if two of them are 1. The first point can be ensured by

  $$s_{0,t} \vee s_{1,t} \vee \cdots \vee s_{l-1,t}$$

  for every t. Forcing that two Boolean variables a and b are not true at the same time can be achieved by $\overline{a \wedge b}$, or equivalently (using de Morgan's laws to ensure CNF), $\overline{a} \vee \overline{b}$; the second part is thus ensured by

  $$\begin{aligned}&(\overline{s}_{0,t} \vee \overline{s}_{1,t}) \wedge (\overline{s}_{0,t} \vee \overline{s}_{2,t}) \wedge \cdots \wedge (\overline{s}_{0,t} \vee \overline{s}_{l-1,t})\\ \wedge\ &(\overline{s}_{1,t} \vee \overline{s}_{2,t}) \wedge (\overline{s}_{1,t} \vee \overline{s}_{3,t}) \wedge \cdots \wedge (\overline{s}_{1,t} \vee \overline{s}_{l-1,t})\\ \wedge\ &\ldots\\ \wedge\ &(\overline{s}_{l-2,t} \vee \overline{s}_{l-1,t})\ .\end{aligned}$$

  Summing up, we obtain

  $$U_{\text{state},N,t}(x) = \Big(\bigvee_{k=0}^{l-1} s_{k,t}\Big) \wedge \bigwedge_{0 \leq k < k' \leq l-1} (\overline{s}_{k,t} \vee \overline{s}_{k',t})$$

  as the subformula of $E_N(x)$ that ensures that N is in exactly one state after making t moves.

  With the same construction, we can make sure that, for any fixed t, N's head scans exactly one cell of the tape and the tape has a unique content. This yields two subformulae

  $$U_{\text{position},N,t}(x) = \Big(\bigvee_{i=1}^{p(|x|)+1} p_{i,t}\Big) \wedge \bigwedge_{0 \leq i < i' \leq p(|x|)+1} (\overline{p}_{i,t} \vee \overline{p}_{i',t})$$

  and

  $$U_{\text{content},N,t}(x) = \Big(\bigwedge_{i=1}^{p(|x|)+1} \Big(\bigvee_{j=1}^{z} c_{i,j,t}\Big)\Big) \wedge \bigwedge_{0 \leq i \leq p(|x|)+1} \Big(\bigwedge_{1 \leq j < j' \leq z} (\overline{c}_{i,j,t} \vee \overline{c}_{i,j',t})\Big)\ ,$$

  and the conjunction of all three subformulae for all t with $0 \leq t \leq p(|x|)$

  $$\text{Unique}_N(x) = \bigwedge_{t=0}^{p(|x|)} \Big(U_{\text{state},N,t}(x) \wedge U_{\text{position},N,t}(x) \wedge U_{\text{content},N,t}(x)\Big)$$

ensures that all configurations are unique.

The total number of literals in this subformula is polynomial in $|x|$; in particular, it is in $\mathcal{O}(p(|x|)^3)$, where the $\mathcal{O}$-notation hides a constant depending on the number l of states and the size z of Γ.

- ***N* starts in initial configuration.** Second, we ensure that N starts in the initial configuration, which we represent as

$$q_0 x_1 x_2 \dots x_m \underbrace{\sqcup\sqcup\dots\sqcup}_{p(|x|)+1-m} .$$

Thus, we have to make sure that, in time step $t = 0$, N is in state q_0 and N's head scans the first cell of its tape. Furthermore, for i with $1 \leq i \leq m$, the ith letter x_i of x is written on the ith cell; for i with $m + 1 \leq i \leq p(|x|) + 1$, the ith cell contains a blank $\sqcup$. Suppose that x_i is the letter $X_{j_i} \in \Gamma$ with $1 \leq j_i \leq z - 1$, and thus $x = X_{j_1} X_{j_2} \dots X_{j_m}$; recall that $X_z = \sqcup$. Then we obtain the subformula

$$\mathrm{Start}_N(x) = s_{0,0} \wedge p_{1,0} \wedge \underbrace{c_{1,j_1,0} \wedge c_{2,j_2,0} \wedge \dots \wedge c_{m,j_m,0}}_{\text{the input } x} \wedge \underbrace{c_{m+1,z,0} \wedge \dots \wedge c_{p(|x|)+1,z,0}}_{\text{the blanks at the end}} ,$$

with $p(|x|) + 2$ literals.

- ***N* ends in accepting configuration.** Third, we make sure that N accepts x. Recall that we assume that N does not leave its accepting state q_{l-1} once it is encountered. If x is accepted, this state therefore has to appear in the last configuration (i.e., the $p(|x|)$th configuration) of N on x, which is taken care of by the subformula

$$\mathrm{Accept}_N(x) = s_{l-1,p(|x|)} ,$$

which consists of a single literal.

- ***N* makes valid moves.** Last, we have to ensure that $E_N(x)$ can only be satisfied if there is a sequence of configurations of N on x that constitute a valid computation. To this end, we first have to make sure that, if N scans the ith tape cell after t moves, only this cell is allowed to be changed after the $(t + 1)$th move.

For every i and j with $1 \leq i \leq p(|x|) + 1$ and $1 \leq j \leq z$, $c_{i,j,t} \leftrightarrow c_{i,j,t+1}$ ensures that the content of the ith cell does not change from time step t to $t + 1$. It is only allowed to be changed if in time step t this ith cell is scanned, which leads to $(c_{i,j,t} \leftrightarrow c_{i,j,t+1}) \vee p_{i,t}$. Since $a \leftrightarrow b$ is equivalent to $(\overline{a} \vee b) \wedge (a \vee \overline{b})$ for two Boolean variables a and b, we obtain the subformula

$$\begin{aligned} M_{\mathrm{head},N,t}(x) &= \bigwedge_{i=1}^{p(|x|)+1} \left(\bigwedge_{j=1}^{z} \left((\overline{c}_{i,j,t} \vee c_{i,j,t+1}) \wedge (c_{i,j,t} \vee \overline{c}_{i,j,t+1}) \right) \vee p_{i,t} \right) \\ &= \bigwedge_{i=1}^{p(|x|)+1} \left(\bigwedge_{j=1}^{z} \left((\overline{c}_{i,j,t} \vee c_{i,j,t+1} \vee p_{i,t}) \wedge (c_{i,j,t} \vee \overline{c}_{i,j,t+1} \vee p_{i,t}) \right) \right) \end{aligned}$$

in CNF.

Of course, $E_N(x)$ has to mimic the transition function δ of N. Consider the transition

$$(q_{k_u}, X_{j_u}, D_{p_u}) \in \delta(q_k, X_j)$$

with $D_1 = \mathrm{L}$ and $D_2 = \mathrm{R}$; we set $d_1 = -1$ and $d_2 = 1$. Suppose that $|\delta(q_k, X_j)| = r$, i.e., there is a nondeterministic guess between r transitions, and thus $1 \leq u \leq r$. We thus design a subformula that can be made true if and only if the $(t+1)$th move is in accord with δ (and any corresponding nondeterministic choice). This is achieved by

$$M_{\mathrm{trans},N,t}(x) = \bigwedge_{i=1}^{p(|x|)} \Bigg(\bigwedge_{j=1}^{z} \Bigg(\bigwedge_{k=0}^{l-1} \Big(\underbrace{\Big(\overline{s}_{k,t} \vee \overline{p}_{i,t} \vee \overline{c}_{i,j,t} \Big) \vee \bigvee_{u=1}^{r} \Big(s_{k_u,t+1} \wedge p_{i+d_{p_u},t+1} \wedge c_{i,j_u,t+1} \Big) \Big)}_{F_{N,t}(i,j,k)} \Bigg) \Bigg) ,$$

which, for any fixed k, i, and j, is true if N is not in state q_k, or the head is not positioned on the ith tape cell, or the ith letter on this cell is not X_j; conversely, if the state, tape head position, and tape content are according to the transition, the subformula can only be made true by following a transition defined by δ.

Note that the functions $F_{N,t}(i,j,k)$ with $0 \leq k \leq l-1$ and $1 \leq i \leq p(|x|)$ are not in CNF.[4] However, converting these parts into CNF leads to an exponential growth with respect to r only, and the resulting formula still has a length polynomial in $|x|$.

The conjunction of the two subformulae for all t with $0 \leq t \leq p(|x|) - 1$ yields

$$\mathrm{Move}_N(x) = \bigwedge_{t=0}^{p(|x|)-1} \Big(M_{\mathrm{head},N,t}(x) \wedge M_{\mathrm{trans},N,t}(x) \Big) .$$

The total number of literals in this subformula is in $\mathcal{O}(p(|x|)^2)$; here the $\mathcal{O}$-notation hides a constant depending on l, z, and r.

Finally, we set

$$E_N(x) = \mathrm{Unique}_N(x) \wedge \mathrm{Start}_N(x) \wedge \mathrm{Accept}_N(x) \wedge \mathrm{Move}_N(x) ,$$

and due to the construction it follows that this Boolean expression is satisfiable if and only if there is a computation of N that accepts x.

It remains to show that the construction can be performed in time polynomial with respect to the input size $|x|$. The number of literals in $E_N(x)$ is in $\mathcal{O}(p(|x|)^3)$; encoding the index of a single literal can hence be done with $\mathcal{O}(\log(|x|))$, where the $\mathcal{O}$-notation hides a constant that depends on the degree of the polynomial that bounds the time complexity of N. The length of $E_N(x)$ is therefore also polynomial in $|x|$, and the construction from N and x can be performed in polynomial time. □

[4] Observe that, here, we cannot simply apply de Morgan's law in order to ensure CNF.

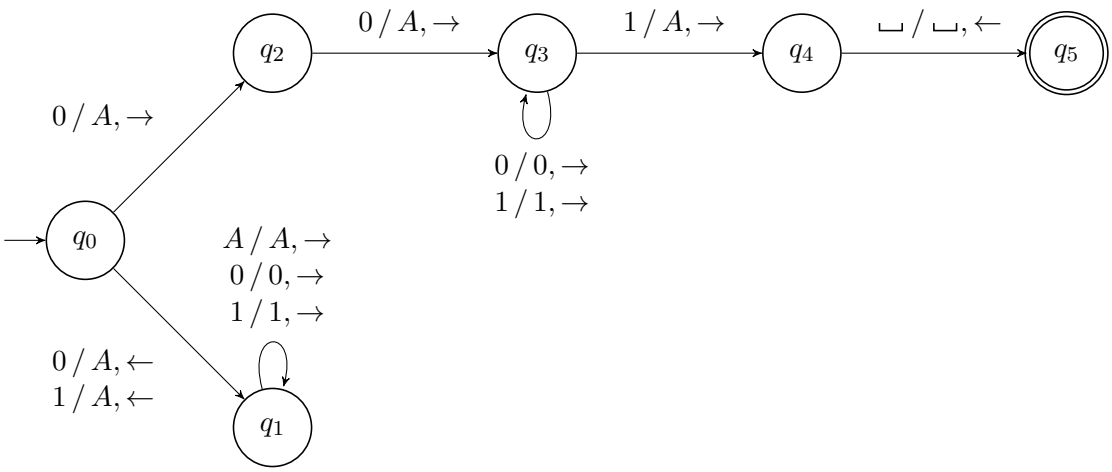


Figure 27.

In order to demonstrate how the construction used in the proof of Theorem 5.12 works, let us consider an example with the NTM $N_{00,1} = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$ with $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, A, \sqcup\}$, and $F = \{q_5\}$, which is shown in Figure 27. $N_{00,1}$ accepts the language

$$L_{00,1} = \{w \in \{0,1\}^* \mid w \text{ starts with 00 and ends with 1}\} \,,$$

and to this end it reads a given input one time from left to right until it encounters the first blank; thus $\mathrm{Time}_{N_{00,1}}(n) = n$. As an example, consider the input word 00101, which is accepted by $N_{00,1}$ after six moves and hence involves seven configurations.

We show the main points of constructing the Boolean expression $E_{N_{00,1}}(00101)$. Since there are six states, four tape letters, and seven configurations, the construction from the proof of Theorem 5.12 yields

- $6 \cdot 7 = 42$ Boolean variables $s_{0,0}, s_{1,0}, \ldots, s_{5,6}$,
- $7 \cdot 7 = 49$ Boolean variables $p_{1,0}, p_{2,0}, \ldots, p_{7,6}$, and
- $7 \cdot 4 \cdot 7 = 196$ Boolean variables $c_{1,1,0}, c_{1,1,1}, \ldots, c_{7,4,7}$.

We enumerate the letters from Γ as $0, 1, A, \sqcup$. Designing the first part $\mathrm{Unique}_{N_{00,1}}(00101)$ of the expression $E_{N_{00,1}}(00101)$ is a straightforward application of what is done in the proof of Theorem 5.12.

Making sure that $N_{00,1}$ has to start with an initial configuration $q_0 00101 \sqcup\sqcup$ is done with the subexpression

$$\mathrm{Start}_{N_{00,1}}(00101) = s_{0,0} \wedge p_{1,0} \wedge c_{1,1,0} \wedge c_{2,1,0} \wedge c_{3,2,0} \wedge c_{4,1,0} \wedge c_{5,2,0} \wedge c_{6,4,0} \wedge c_{7,4,0} \,,$$

where the last two variables model that there are two blanks right of the input word such that the string has a length of seven.

That $E_{N_{00,1}}(00101)$ can be satisfied if and only if $N_{00,1}(x)$ ends in an accepting state is taken care of by the simple expression

$$\mathrm{Accept}_{N_{00,1}}(00101) = s_{5,6} \ ,$$

which just means that $N_{00,1}$ is in the accepting state q_5 after six moves.

The subformulae $M_{\mathrm{head},N_{00,1},t}(00101)$ of $\mathrm{Move}_{N_{00,1}}(00101)$ do not depend on the transitions of the NTM, as they simply state that only correct cells of the tape can be modified. It is more interesting to have a look at the subformulae $M_{\mathrm{trans},N_{00,1},t}(00101)$. Consider reading a 0 in state q_0; there is a nondeterministic guess involved, and we have $\delta(q_0, 0) = \{(q_1, A, \mathrm{L}), (q_2, A, \mathrm{R})\}$. Let us look at the first option, which involves q_1. For a fixed tape position i and time step t, the subformula

$$s_{1,t+1} \wedge p_{i-1,t+1} \wedge c_{i,3,t+1}$$

can only be made true if in the next time step (i.e., after the $(t+1)$th move) the state q_1 is entered, the cell left of i is scanned, and cell i contains the third letter of Γ, which is A. Likewise, for the second option, we obtain

$$s_{2,t+1} \wedge p_{i+1,t+1} \wedge c_{i,3,t+1} \ .$$

We consider the "or" of these two expressions, and further combine them with expressions (the first part of $F_{N_{00,1},t}(i,j,k)$) that make sure that this formula only needs to be true if and only if in time step t the current state is is q_0 and the symbol read is 0. Then the "and" over all possible states, tape positions, tape letters, and time steps, ensures that only valid transitions can be followed in order to satisfy $E_{N_{00,1}}(00101)$.

The accepting computation of $N_{00,1}$ on 00101 is

$$q_0 00101 \vdash A q_2 0101 \vdash AA q_3 101 \vdash AA1 q_3 01 \vdash AA10 q_3 1 \vdash AA10A q_4 \text{␣} \vdash AA10 q_5 A \ .$$

The part of $E_{N_{00,1}}(00101)$ mimicking this sequence of seven configurations is

$$\begin{aligned}
&\mathrm{Start}_{N_{00,1}}(00101) && \{\text{state: } q_0,\ \text{position: } 1,\ \text{read: } 0\} \\
&\quad \wedge s_{2,1} \wedge p_{2,1} \wedge c_{1,3,1} && \{\text{state: } q_2,\ \text{position: } 2,\ \text{written: } A,\ \text{read: } 0\} \\
&\quad \wedge s_{3,2} \wedge p_{3,2} \wedge c_{2,3,2} && \{\text{state: } q_3,\ \text{position: } 3,\ \text{written: } A,\ \text{read: } 1\} \\
&\quad \wedge s_{3,3} \wedge p_{4,3} \wedge c_{3,2,3} && \{\text{state: } q_3,\ \text{position: } 4,\ \text{written: } 1,\ \text{read: } 0\} \\
&\quad \wedge s_{3,4} \wedge p_{5,4} \wedge c_{4,1,4} && \{\text{state: } q_3,\ \text{position: } 5,\ \text{written: } 0,\ \text{read: } 1\} \\
&\quad \wedge s_{4,5} \wedge p_{6,5} \wedge c_{5,3,5} && \{\text{state: } q_4,\ \text{position: } 6,\ \text{written: } A,\ \text{read: } \text{␣}\} \\
&\quad \wedge s_{5,6} \wedge p_{5,6} \wedge c_{6,4,6} && \{\text{state: } q_5,\ \text{position: } 5,\ \text{written: } \text{␣},\ \text{read: } A\} \\
&\quad \wedge \mathrm{Accept}_{N_{00,1}}(00101) \ .
\end{aligned}$$

5.5 Other $\mathcal{NP}$-Complete Problems

Our goal is to prove other problems to be $\mathcal{NP}$-complete by polynomial-time reductions. According to Definition 5.10, we have to show that every problem in $\mathcal{NP}$ reduces to the

given problem $L \in \mathcal{NP}$ in order to show its $\mathcal{NP}$-completeness; this is what we did in the proof of Theorem 5.12. Having a first $\mathcal{NP}$-complete problem at hand, however, it is sufficient to reduce this single problem to L, since this implies that all problems in $\mathcal{NP}$ reduce to L in polynomial time.

Theorem 5.13. *Let $L_1, L_2 \in \mathcal{NP}$ be two languages, and suppose $L_1 \leq_\mathsf{p} L_2$. If L_1 is $\mathcal{NP}$-complete, then L_2 is $\mathcal{NP}$-complete, too.*

Proof. First, since L_1 is $\mathcal{NP}$-complete, due to Definition 5.10 there is a polynomial-time reduction from L to L_1 for every $L \in \mathcal{NP}$, i.e., $L \leq_\mathsf{p} L_1$. There is thus a TM P_1 that converts any input x for L to an input y for L_1 in time $p_1(|x|)$, where p_1 is a polynomial function. Second, since $L_1 \leq_\mathsf{p} L_2$, there is a TM P_2 that converts inputs y for L_1 to inputs z for L_2 in time $p_2(|y|)$ for a polynomial function p_2.

Using P_1 and P_2 consecutively, the input x for L can thus be converted into an input y for L_1 in time at most $p_1(|x|)$ with $|y| \leq p_1(|x|)$, and then y can be converted to an input z for L_2 in time at most $p_2(|y|) \leq p_2(p_1(|x|))$. As a consequence, x is converted to z in time at most $p_1(|x|) + p_2(p_1(|x|))$. This constitutes a polynomial-time reduction from L to L_2, and therefore L_2 is $\mathcal{NP}$-complete. □

Note that, if we drop the assumption that L_1 and L_2 are both in $\mathcal{NP}$, the same proof can be used to show L_2 to be $\mathcal{NP}$-hard if L_1 is $\mathcal{NP}$-hard.

By now, there are thousands of $\mathcal{NP}$-complete problems, and so far no one was able to either show that any of them allows for a polynomial-time algorithm (in which case $\mathcal{P} = \mathcal{NP}$) or that it cannot be solved in polynomial time (in which case $\mathcal{P} \neq \mathcal{NP}$). Here, we will only look at four specific problems. The first one is a special variant of SAT, and the other three are all defined on graphs; as before, by "graph" we always refer to undirected and unweighted graphs.

5.5.1 The Satisfiability Problem with Clauses of Length Three

An input for SAT is by definition given in CNF; now we consider a subproblem which only allows clauses with exactly three literals; we call this a 3CNF, and the corresponding problem is called 3SAT. An input for 3SAT is, e.g.,

$$(x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_3 \vee x_4) \wedge (\overline{x}_3 \vee x_4 \vee \overline{x}_5) \ ,$$

and even though the restriction to 3CNF seems to remove quite some of the complexity of the problem, it is still $\mathcal{NP}$-complete.

Theorem 5.14. 3SAT *is $\mathcal{NP}$-complete.*

Proof. Clearly, 3SAT $\in \mathcal{NP}$ by the same arguments we used to argue that SAT $\in \mathcal{NP}$. We now prove that 3SAT is $\mathcal{NP}$-hard by SAT $\leq_\mathsf{p}$ 3SAT, which is sufficient due to Theorem 5.13. Let E be a Boolean expression in CNF that is an input for SAT. We show how to convert E to a Boolean expression E' in 3CNF that is satisfiable if and only if E is satisfiable. We consider each of the clauses $C_1, C_2, \ldots, C_n$ of E separately, and describe how each C_i with $1 \leq i \leq n$ is converted to a sequence of clauses each with three literals that are satisfiable if and only if C_i is satisfiable.

- If $C_i = (l_i)$, i.e., it consists of a single literal, then E' contains four clauses

$$E'_i = (l_i \vee y_1 \vee y_2) \wedge (l_i \vee y_1 \vee \overline{y}_2) \wedge (l_i \vee \overline{y}_1 \vee y_2) \wedge (l_i \vee \overline{y}_1 \vee \overline{y}_2)$$

that replace C_i, where y_1 and y_2 are "new" Boolean variables (i.e., variables that are not contained in E).

If a truth assignment φ satisfies E, then it has to set C_i and thus l_i to 1. If there is such a φ, then there is also an truth assignment φ' for E'_i that extends φ; φ' satisfies E'_i by setting l_i to 1 and y_1 and y_2 arbitrarily.

Conversely, if a truth assignment φ sets l_i to 0, there is no possibility to extend φ such that E'_i is satisfied, independent of how the values of y_1 and y_2 are chosen.

- If $C_i = (l_{i,1} \vee l_{i,2})$, i.e., it consists of two literals, then E' contains two clauses

$$E'_i = (l_{i,1} \vee l_{i,2} \vee y) \wedge (l_{i,1} \vee l_{i,2} \vee \overline{y})$$

instead of C_i, where y is a new variable. Similarly as in the first case, if a truth assignment φ makes C_i true, it can be extended to a truth assignment φ' that makes E'_i true; if there is no such φ, then E'_i cannot be satisfied.

- If $C_i = (l_{i,1} \vee l_{i,2} \vee l_{i,3})$, i.e., it consists of three literals, then E' contains $E'_i = C_i$.

- If $C_i = (l_{i,1} \vee l_{i,2} \vee \cdots \vee l_{i,m})$ with $m \geq 4$, i.e., it consists of more than three literals, then let $y_1, y_2, \ldots, y_{m-3}$ be $m-3$ new variables. C_i is replaced by

$$\begin{aligned} E'_i = {} & (l_{i,1} \vee l_{i,2} \vee y_1) \wedge (\overline{y}_1 \vee l_{i,3} \vee y_2) \wedge (\overline{y}_2 \vee l_{i,4} \vee y_3) \wedge (\overline{y}_3 \vee l_{i,5} \vee y_4) \wedge \ldots \\ & \cdots \wedge (\overline{y}_{m-4} \vee l_{i,m-2} \vee y_{m-3}) \wedge (\overline{y}_{m-3} \vee l_{i,m-1} \vee l_{i,m}) \ , \end{aligned}$$

which consists of $m-2$ clauses.

Now consider a truth assignment φ that satisfies C_i by setting one of its literals to 1, say $l_{i,k}$. In this case, there is a truth assignment φ' that makes E'_i true, by also setting $l_{i,k}$ to 1 together with $y_1, y_2, \ldots, y_{k-2}$; furthermore, $y_{k-1}, y_k, \ldots, y_{m-3}$ are set to 0. This leads to

$$\begin{aligned} \varphi'(E'_i) = {} & (l_{i,1} \vee l_{i,2} \vee 1) \wedge (0 \vee l_{i,3} \vee 1) \wedge (0 \vee l_{i,4} \vee 1) \wedge \ldots \\ & \cdots \wedge (0 \vee l_{i,k} \vee 0) \wedge \ldots \\ & \cdots \wedge (1 \vee l_{i,m-3} \vee 0) \wedge (1 \vee l_{i,m-2} \vee 0) \wedge (1 \vee l_{i,m-1} \vee l_{i,m}) \ , \end{aligned}$$

which is true since the kth clause of E'_i is true due to $l_{i,k} = 1$.

Conversely, any truth assignment that does not make C_i true cannot be extended to a truth assignment that makes E'_i true. To see this, note that all literals of C_i are set to false by such an assignment. It is not possible to make all clauses of E'_i true by choosing the values of the new variables accordingly, because there are $m-2$ clauses but only $m-3$ new variables, and each variable can only make at most one clause true.

Finally, we set $E' = E'_1 \wedge E'_2 \wedge \cdots \wedge E'_n$. Due to the construction of E' and the considerations above, if there is a truth assignment that makes E true, then there is truth assignment that makes E' true. If, however, there is no truth assignment that makes E true, there is no truth assignment that makes E' true. Hence, we get the equivalence

$$E \in \text{SAT} \iff E' \in \text{3SAT} .$$

It follows that 3SAT is $\mathcal{NP}$-complete. □

We see that 1SAT is decidable in polynomial time; easily, if all clauses of a Boolean expression are of size one, this expression is satisfiable if and only if no literal is the negation of another one. 2SAT is also in $\mathcal{P}$ although proving this is not as straightforward.

5.5.2 The Independent Set Problem

We have already described the decision problem IND-SET in Section 5.2; it turns out that this problem is also $\mathcal{NP}$-complete, which can be shown by a polynomial-time reduction from 3SAT.

Theorem 5.15. IND-SET *is $\mathcal{NP}$-complete.*

Proof. We already know that IND-SET $\in \mathcal{NP}$, thus it remains to show that it is also $\mathcal{NP}$-hard. To this end, we show $\text{3SAT} \leq_{\mathsf{p}} \text{IND-SET}$. Let E be a Boolean expression as input for 3SAT, and suppose that E consists of n clauses $C_1, C_2, \ldots, C_n$; the literals of C_i are denoted by $l_{i,1}$, $l_{i,2}$, and $l_{i,3}$. We construct an input (G, k) to IND-SET as follows. G consists of n cliques $K_1, K_2, \ldots, K_n$ that are each of size three; the three vertices of K_i are called $v_{i,1}$, $v_{i,2}$, and $v_{i,3}$ with $1 \leq i \leq n$. The clique K_i corresponds to the clause C_i (which is why we will refer to it as a "clause clique"), and more specifically the jth vertex $v_{i,j}$ of K_i corresponds to the jth literal $l_{i,j}$ of C_i. All vertices $v_{i,j}$ and $v_{i',j'}$ with $i \neq i'$ are connected by an edge if and only if $l_{i,j}$ is the negation of $l_{i',j'}$.

Now we prove the equivalence

$$E \in \text{3SAT} \iff (G, n) \in \text{IND-SET} .$$

Suppose that $E \in \text{3SAT}$; hence, E is satisfiable and there is a truth assignment φ with $\varphi(E) = 1$. For every clause C_i of E, there is consequently at least one literal $l_{i,j}$ with $1 \leq i \leq n$ and $1 \leq j \leq 3$ that is made true by φ; if there are more than one for some clause, we pick one of them arbitrarily. Now consider the set I of the corresponding n vertices $v_{i,j}$ of G. Clearly, $|I| = n$, and thus it only remains to show that I is indeed an independent set, i.e., that there are no edges between any two vertices in I. Note that no two vertices from I are from the same clause clique, because the vertices are all from different clauses. Now let v and w be two vertices from I. By the construction of G, the only way that there is an edge between them is that v corresponds to a literal l and w corresponds to its negation $\bar{l}$. However, this would mean that φ sets both l and $\bar{l}$ to one, which is impossible. Therefore, there is no edge between v and w. It follows that I is an independent set of size n in G and thus $(G, n) \in \text{IND-SET}$.

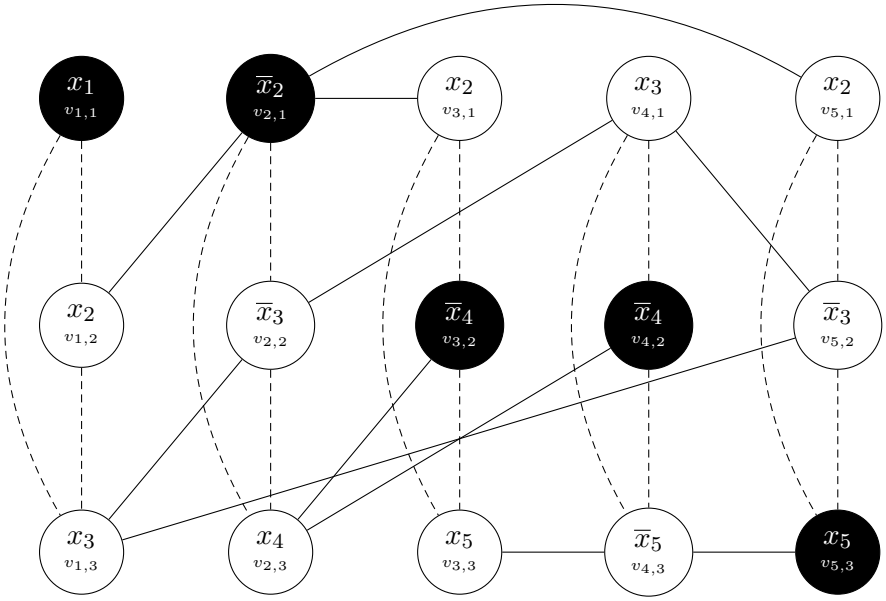


Figure 28.

Now suppose that $(G, n) \in$ IND-SET; hence there is an independent set I with $|I| = n$ in G. We first note that no two vertices from I can be part of the same clause clique, and thus there has to be exactly one vertex per clause in I. Consider the truth assignment φ that sets the literal $l_{i,j}$ to 1 if $v_{i,j} \in I$. All variables that are not assigned a value as a consequence get an arbitrary value, say 1. We argue that φ satisfies E. Since I contains one vertex per clause clique, at least one literal per clause is set to 1 by φ. Moreover, no two of these literals are the negation of one another, since then the corresponding vertices were connected by an edge in G and thus would not both be part of I. It follows that $\varphi(E) = 1$ and thus $E \in$ 3SAT. □

Let us apply the construction from the proof of Theorem 5.15 to the Boolean expression

$$E = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_2 \vee \overline{x}_3 \vee x_4) \wedge (x_2 \vee \overline{x}_4 \vee x_5) \wedge (x_3 \vee \overline{x}_4 \vee \overline{x}_5) \wedge (x_2 \vee \overline{x}_3 \vee x_5) \ ,$$

which consists of five clauses C_1, C_2, C_3, C_4, and C_5. The corresponding graph is shown in Figure 28 and consists of five cliques K_1, K_2, K_3, K_4, and K_5, such that K_i corresponds to the clause C_i; the threshold k which is part of the input is 5. An independent set of size 5 is given by the vertices

$$v_{1,1}, \quad v_{2,1}, \quad v_{3,2}, \quad v_{4,2}, \quad \text{and } v_{5,3} \ ,$$

and this corresponds to a truth assignment φ that assigns the variables

$$x_1 = 1, \quad x_2 = 0, \quad x_4 = 0, \quad x_4 = 0, \quad \text{and } x_5 = 1 \ .$$

This leads to

$$\varphi(E) = (1 \vee 0 \vee x_3) \wedge (1 \vee \overline{x}_3 \vee 0) \wedge (0 \vee 1 \vee 1) \wedge (x_3 \vee 1 \vee 0) \wedge (0 \vee \overline{x}_3 \vee 1) \ ,$$

which evaluates to true independent of the value of x_3. It follows that E is satisfiable and thus $E \in$ SAT, while at the same time $(G, 5) \in$ IND-SET.

5.5.3 The Clique Problem

One of the most straightforward insights obtainable once the $\mathcal{NP}$-completeness of IND-SET is understood is that also CLIQUE is $\mathcal{NP}$-complete. Indeed, compared to the preceding ones, the subsequent reduction is as simple as it gets.

Theorem 5.16. CLIQUE *is $\mathcal{NP}$-complete.*

Proof. It is easy to see that CLIQUE $\in \mathcal{NP}$. In order to show that it is also $\mathcal{NP}$-hard, we prove IND-SET $\leq_{\mathsf{p}}$ CLIQUE. The idea behind the following proof is already hinted in Figure 26. Let (G, k) be an input for IND-SET. The input for CLIQUE is $(\overline{G}, k)$, where $\overline{G}$ is the complement of G, i.e., the graph that is obtained by removing all edges from G and adding edges between all vertices that are not connected by an edge in the original graph. Converting (G, k) to $(\overline{G}, k)$ can be done in polynomial time; all that needs to be done is to complement the adjacency matrix of G (by replacing 0s by 1s and vice versa, except for the 0s on the main diagonal).

Any set of vertices in G in which no two vertices are connected by an edge corresponds to a set of vertices in $\overline{G}$ in which every pair of vertices is connected by an edge. In other words, we have the equivalence

$$G \text{ has an independent set of size } k \iff \overline{G} \text{ has a clique of size } k\ ,$$

and thus CLIQUE is $\mathcal{NP}$-complete. □

5.5.4 The Vertex Cover Problem

Let us investigate yet another problem on graphs.

Definition 5.17 (Vertex Cover). *Let $G = (V, E)$ be a graph. A **vertex cover** in G is any set of vertices N such that every edge in G has at least one of its vertices in N.*

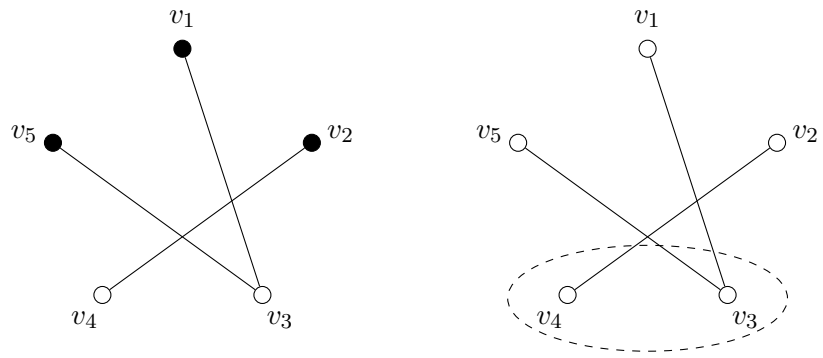
The vertex cover problem, VC for short, is defined analogously to IND-SET and CLIQUE, i.e., we are given a graph and a threshold and want to decide whether the graph has a vertex cover of the given size.

Theorem 5.18. VC *is $\mathcal{NP}$-complete.*

Proof. As for IND-SET and CLIQUE, it is easy to see that VC $\in \mathcal{NP}$. Now we show that VC is $\mathcal{NP}$-hard by proving IND-SET $\leq_{\mathsf{p}}$ VC. Let (G, k) be any input for IND-SET, and suppose that G has n vertices. We construct an input $(G, n - k)$ to VC; the conversion can clearly be done in polynomial time. We now show the equivalence

$$G \text{ has an independent set of size } k \iff G \text{ has a vertex cover of size } n - k\ .$$

Suppose there is a vertex cover in G of size $n - k$. Then there is a set N in G such that every edge from G has at least one of its endpoints in N. Let N' denote the other



(a) Independent set N' of size 3 (b) Vertex cover N of size 2

Figure 29.

vertices from G that are not in N. Since N is a vertex cover, there are no edges between the vertices in N'. Thus N' is an independent set in G.

Now suppose there is no vertex cover of size $n-k$ in G. Then there is accordingly no set of size $n-k$ such that all edges have endpoints in this set. It follows that there is no set of size k in G such that there is no edge between any two vertices in this set, which means that there is no independent set of size k in G.

It follows that VC is $\mathcal{NP}$-complete. □

The idea used in the proof of Theorem 5.18 is shown in Figure 29. Here, G has 5 vertices and k is 3. G contains an independent set N' of size 3 (see Figure 29a) and at the same time a vertex cover of size $5-3=2$ (see Figure 29b). The independent set $N' = \{v_1, v_2, v_5\}$ of G is marked by filled vertices; the vertex cover $N = \{v_3, v_4\}$ is circled with a dashed line.

Note that the constructions from Theorems 5.16 and 5.18 can be combined in a straightforward fashion to give, e.g., a reduction from CLIQUE to VC.

5.6 An $\mathcal{NP}$-Hard Problem Outside $\mathcal{NP}$

Definition 5.10 defines the two terms "$\mathcal{NP}$-hard" and "$\mathcal{NP}$-complete," but so far all $\mathcal{NP}$-hard problems we studied are also contained in $\mathcal{NP}$. So are there actually problems that are $\mathcal{NP}$-hard, but not in $\mathcal{NP}$? The answer is positive, and thus there indeed exist problems that are $\mathcal{NP}$-hard but not $\mathcal{NP}$-complete; these problems are "harder," and the halting problem L_{H} is one of them. Obviously, it cannot be in $\mathcal{NP}$, because it is not even decidable as we know from Theorem 4.15, but all problems in $\mathcal{NP}$ admit TMs that eventually halt. We now show that L_{H} is $\mathcal{NP}$-hard by giving a polynomial-time reduction from SAT.

Theorem 5.19. *L_{H} is $\mathcal{NP}$-hard.*

Proof. Suppose there is a TM H^* for L_{H} that always halts and that runs in polynomial time. We design a TM M_{SAT} that decides whether a given instance of SAT is satisfiable in polynomial time using H^*, thus proving $\mathrm{SAT} \leq_{\mathsf{p}} L_{\mathrm{H}}$. Let w be the input for M_{SAT}. M_{SAT} first checks whether w is a valid input for SAT, that is, a formula in CNF. If not,

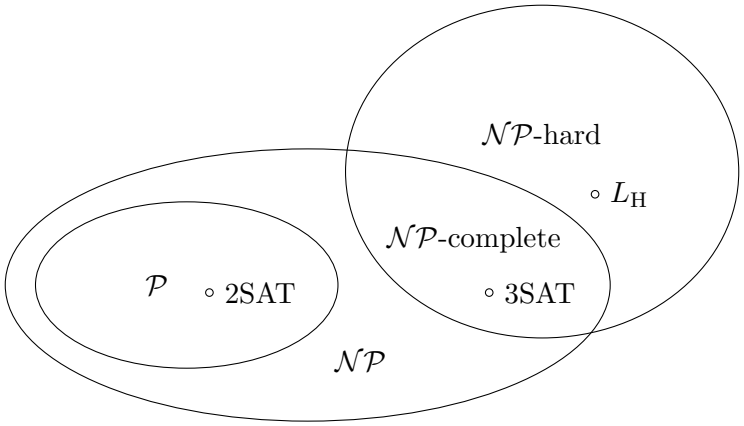


Figure 30.

M_{SAT} constructs an output that is for sure not accepted by H^*, for instance, the word $\mathrm{Code}(M')111\varepsilon$ with M' being a TM that runs forever on ε. If it is, M_{SAT} constructs a TM M that works as follows. For a given input E, which is a formula in CNF, M tries all possible assignments of the given variables and checks whether E is true for any of them.

- If M finds such an assignment, it halts.
- Otherwise, there is no such assignment; in this case, M enters some state in which it loops forever, that is, never halts.

For the given instance w of SAT, we therefore have the equivalence

$$w \text{ is satisfiable} \iff M \text{ halts on } w\ .$$

For sure, the running time of M on w is exponential in the length of w, but this does not matter. What does matter is that M can be constructed from w in polynomial time.

Then, $\mathrm{Code}(M)111w$ is given as input to H^*. By definition, H^* will output the correct answer while working in polynomial time. The answer "yes" is given if and only if w is satisfiable. Thus, H^* can be used to solve SAT in polynomial time, which contradicts that SAT is $\mathcal{NP}$-hard and $\mathcal{P} \neq \mathcal{NP}$. □

It is easy to see that a slight modification of the proof of Theorem 5.19 leads to an analogous statement about L_{U}. Our current assumptions on the relationship between $\mathcal{P}$ and $\mathcal{NP}$ is depicted in Figure 30. Finally, let us remark that, in the case that $\mathcal{P} \neq \mathcal{NP}$, there are problems in $\mathcal{NP}$ that are neither in $\mathcal{P}$ nor $\mathcal{NP}$-complete; the corresponding problems are called **$\mathcal{NP}$-intermediate** .

5.7 Historical and Bibliographical Notes

Cobham [5] and Edmonds were the first to argue that problems that are solvable in polynomial-time are those that are solvable "efficiently" in practice, and this has been commonly known as the **Cobham-Edmonds thesis**. In 1971, Stephen Cook gave rise to

the theory of $\mathcal{NP}$-completeness by proving Cook's theorem [6] (Theorem 5.12), and one year later, Richard Karp showed the $\mathcal{NP}$-completeness of 21 other decision problems [14]; "Karp's 21 $\mathcal{NP}$-complete problems" include the independent set problem (Theorem 5.15), the clique problem (Theorem 5.16), and the vertex cover problem (Theorem 5.18). Independent results were proven by Levin in 1973 [18]. By now, we know thousands of $\mathcal{NP}$-complete problems, and it is quite remarkable that so far no one found either an efficient algorithm or a strong lower bound for any of them [8, 9].

We assumed SAT to be defined on Boolean expressions in CNF, while in the literature it is sometimes defined on general Boolean expressions and our definition of SAT is called CSAT [11]. Moreover, we defined 3CNF such that all clauses contain exactly three literals. It is sometimes assumed that they have at most three literals; in this case, our 3SAT is referred to as E3SAT [12].

The fact that $\mathcal{NP}$-intermediate problems exist is known as Ladner's theorem and was shown by Ladner in 1975 [17].

References

[1] Yehoshua Bar-Hillel, Micha Perles, and Eli Shamir. On formal properties of simple phrase-structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft, und Kommunikationsforschung*, 14(2):143–172, 1961.

[2] Noam Chomsky. Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3):113–124, 1956.

[3] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[4] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[5] Alan Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the 2nd International Congress for Logic, Methodology and Philosophy of Science*, pages 24–30. North-Holland, 1965.

[6] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971)*, pages 151–158. Association for Computing Machinery, 1971.

[7] Wikipedia — The Free Encyclopedia. Cross-serial dependencies. https://en.wikipedia.org/wiki/Cross-serial_dependencies.

[8] Wikipedia — The Free Encyclopedia. List of $\mathcal{NP}$-complete problems. https://en.wikipedia.org/wiki/List_of_NP-complete_problems.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability*. Freeman, 1979.

[10] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[11] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley Pub Co Inc, 3rd edition, 2006.

[12] Juraj Hromkovič. *Theoretical Computer Science*. Springer-Verlag, 2004.

[13] David A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(3):161–190, 1954.

[14] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a Symposium on the Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[15] Stephen C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112:727–742, 1936.

[16] Stephen C. Kleene. Representation of events in nerve nets and finite automata. In Claude E. Shannon and John McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, 1956.

[17] Richard Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, 22(1):155–171, 1975.

[18] Leonid A. Levin. Universal search problems. *Problems of Information Transmission*, 9(3):115–116, 1973.

[19] Robert McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9(1):39–47, 1960.

[20] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[21] Edward F. Moore. Gedanken-Experiments on sequential machines. In Claude E. Shannon and John McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, 1956.

[22] Anil Nerode. Linear automaton transformations. *Proceedings of the AMS*, 9(4):541–544, 1958.

[23] William F. Ogden. A helpful result for proving inherent ambiguity. *Mathematical Systems Theory*, 2(3):191–194, 1956.

[24] Emil Post. Finite combinatory process-formulation. *Journal of Symbolic Logic*, 1:103–105, 1936.

[25] Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[26] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of AMS*, 89:25–59, 1953.

[27] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[28] Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.