Appendix A
Mathematical Machinery

A.1 Basic Notation

Let A, B be sets. We write $\emptyset$ for the *empty set* and $\mathrm{card}(A)$ to denote the *cardinality of the set* A and $\mathcal{P}(B)$ for the *power set* of B. The operations $\cup$, $\cap$, $-$, and the relation $\subseteq$ on sets are defined as usual. The set $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$ denotes the *Cartesian product* of A and B (also called the cross-product), where (a, b) is the ordered pair. If R is a relation, then by R^{-1} we denote its inverse. *The set of natural numbers* is denoted by ω. If $k \in \omega$, then A^k denotes

$$\underbrace{A \times \ldots \times A}_{k}.$$

We will write $\mathbb{Q}$ for *the set of rational numbers*. If $q \in \mathbb{Q}$, we write $\lceil q \rceil$ for the ceiling function of q, i.e., the smallest integer greater than q.

A.2 Computability

Throughout the book we use the general notions of computability theory (see, e.g., Hopcroft et al. 2000; Cooper 2003). In particular, we refer to the basic methods and notions of complexity theory (see, e.g., Papadimitriou 1993; Kozen 2006). Below we review some of them briefly to keep the book self-contained.

A.2.1 Languages and Automata

Formal language theory—which we briefly survey below—is an important part of logic, computer science, and linguistics (see, e.g., Hopcroft et al. 2000 for a complete treatment). Historically speaking, formal language theory forms the foundation of modern (mathematical) linguistics and its connection with psycholinguistics (see, e.g., Partee et al. 1990).

Languages

By an *alphabet* we mean any nonempty finite set of symbols. For example, $A = \{a, b\}$ and $B = \{0, 1\}$ are two different binary alphabets.

A *word (string)* is a finite sequence of symbols from a given alphabet, e.g., '1110001110' is a word over the alphabet B.

The *empty word* is a sequence without symbols. It is needed mainly for technical reasons and is written ε.

The *length of a word* is the number of symbols occurring in it. We write $lh()$ for length, e.g., $lh(111) = 3$ and $lh(\varepsilon) = 0$.

If Γ is an alphabet, then by Γ^k we mean the *set of all words of length* k over Γ. For instance, $A^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$. For every alphabet Γ we have $\Gamma^0 = \{\varepsilon\}$.

For any letter a and a natural number n by a^n we denote a string of length n consisting only of the letter a.

The set of all words over alphabet Γ is denoted by Γ^*, e.g., $\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. In other words, $\Gamma^* = \bigcup_{n \in \omega} \Gamma^n$. Γ^* is almost always infinite, except for two cases: $\Gamma = \emptyset$ and $\Gamma = \{\varepsilon\}$.

By xy we mean the *concatenation* of the word x with the word y, i.e., the new word xy is built from x followed by y. If $x = a_1 \ldots a_i$ and $y = b_1 \ldots b_n$, then xy is of length $i + n$ and $xy = a_1 \ldots a_i b_1 \ldots b_n$. For instance, if $x = 101$ and $y = 00$, then $xy = 10100$. For any string α the following holds: $\varepsilon\alpha = \alpha\varepsilon = \alpha$. Hence, ε is the neutral element for concatenation.

Any set of words, a subset of Γ^*, will be called a *language*. If Γ is an alphabet and $L \subseteq \Gamma^*$, then we say that L is a language over Γ. For instance, the set $L \subseteq A^*$ such that $L = \{\alpha \mid \text{the number of occurrences of } b \text{ in } \alpha \text{ is even}\}$ is a language over the alphabet A.

Finite Automata

A finite-state automaton is a model of computation consisting of a finite number of states and transitions between those states. We give a formal definition below.

Definition A.1 A *nondeterministic finite automaton* (FA) is a tuple (A, Q, q_s, F, δ), where:

- A is an input alphabet;
- Q is a finite set of states;
- $q_s \in Q$ is an initial state;
- $F \subseteq Q$ is a set of accepting states;
- $\delta : Q \times A \longrightarrow \mathcal{P}(Q)$ is a transition function.

If $H = (A, Q, q_s, F, \delta)$ is an FA such that for every $a \in A$ and $q \in Q$ we have $\text{card}(\delta(q, a)) \leq 1$, then H is a *deterministic* automaton. In this case we can describe a transition function as a partial function: $\delta : Q \times A \longrightarrow Q$.

Finite automata are often presented as graphs, where vertices (circles) symbolize internal states, the initial state is marked by an arrow, an accepting state is double circled, and arrows between nodes describe a transition function on letters given by the labels of these arrows. We will give a few examples in what follows.

Definition A.2 Let us first define the *generalized transition function* $\bar{\delta}$, which describes the behavior of an automaton reading a string w from the initial state q:

$$\bar{\delta} : Q \times A^* \longrightarrow \mathcal{P}(Q), \text{ where:}$$

$$\bar{\delta}(q, \varepsilon) = \{q\}$$

$$\text{and for each } w \in A^* \text{ and } a \in A, \ \bar{\delta}(q, wa) = \bigcup_{q' \in \bar{\delta}(q,w)} \delta(q', a).$$

Definition A.3 *The language accepted (recognized) by some FA H* is the set of all words over the alphabet A which are accepted by H, that is:

$$L(H) = \{w \in A^* : \bar{\delta}(q_s, w) \cap F \neq \emptyset\}.$$

Definition A.4 We say that a *language* $L \subseteq A^*$ *is regular* if and only if there exists some FA H such that $L = L(H)$.

The following equivalence is a well-known fact.

Theorem A.1 *Deterministic and nondeterministic finite automata recognize the same class of languages, i.e., regular languages.*

Proof First of all notice that every deterministic FA is a nondeterministic FA. Then we only have to show that every nondeterministic FA can be simulated by some deterministic FA. The proof goes through the so-called subset construction. It involves constructing all subsets of the set of states of the nondeterministic FA and using them as states of a new, deterministic FA. The new transition function is defined naturally (see Hopcroft et al. 2000 for details). Notice that in the worst case the new deterministic automaton can have 2^n states, where n is the number of states of the corresponding nondeterministic automaton. □

Example A.1 Let us give a few simple examples of regular languages together with the corresponding accepting automata.

Let $A = \{a, b\}$ and consider the language $L_1 = A^*$. $L_1 = L(H_1)$, where $H_1 = (Q_1, q_1, F_1, \delta_1)$, such that: $Q_1 = \{q_1\}$, $F_1 = \{q_1\}$, $\delta_1(q_1, a) = q_1$ and $\delta_1(q_1, b) = q_1$. The automaton is shown in Fig. A.1.

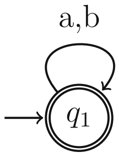

Fig. A.1 Finite automaton recognizing language $L_1 = A^*$

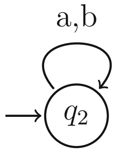

Fig. A.2 Finite automaton recognizing language $L_2 = \emptyset$

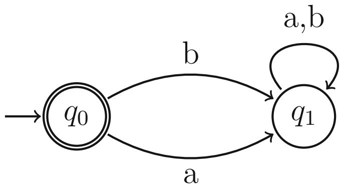

Fig. A.3 Finite automaton recognizing language $L_3 = \{\varepsilon\}$

Now let $L_2 = \emptyset$; then $L_2 = L(H_2)$, where $H_2 = (Q_2, q_2, F_2, \delta_2)$ such that: $Q_2 = \{q_2\}$, $F_2 = \emptyset$, $\delta_2(q_2, a) = q_2$, and $\delta_2(q_2, b) = q_2$. The automaton is depicted in Fig. A.2.

Finally, let $L_3 = \{\varepsilon\}$. $L_3 = L(H_3)$, where $H_3 = (Q_3, q_0, F_3, \delta_3)$ such that: $Q_3 = \{q_0, q_1\}$, $F_3 = \{q_0\}$, $\delta_3(q_0, i) = q_1$, and $\delta_3(q_1, i) = q_1$, for $i = a, b$.

The finite automaton accepting this language is presented in Fig. A.3.

Now let us review some useful closure results for regular languages.

Theorem A.2 *Regular languages are closed under the Boolean operations of union, intersection, and complementation.*

Proof

- For union closure, connecting a new start state s' to the start state s_1 of A_1 and s_2 of A_2 by ϵ-transitions yields an NFA recognizing $L_1 \cup L_2$.
- Intersection closure is shown by the *product construction*. Taking $Q_1 \times Q_2$ as the set of states, $F_1 \times F_2$ as the set of accepting states, and transitioning from $\langle q, p\rangle$ to $\langle q', p'\rangle$ on symbol x if $\delta_1(q, x) = q'$ and $\delta_2(p, x) = p'$ yields a DFA that recognizes $L_1 \cap L_2$.
- For complementation closure, reversing the accepting and rejecting states (F_1 and $Q_1 - F_1$) of A_1 yields a DFA recognizing $\overline{L_1}$. □

Theorem A.3 *Regular languages are closed under concatenation.*

Proof Connecting final states of A_1 to the start state of A_2 by ϵ-transitions yields an NFA recognizing L_1L_2. □

A *substitution* s on L with alphabet Σ is a mapping of each $a \in \Sigma$ to a language L_a. For $w = a_1 \cdots a_n \in L$, $s(w)$ is the language of the concatenation $s(a_1) \cdots s(a_n)$. Then $s(L)$ is the union of $s(w)$ for all $w \in L$.

Theorem A.4 *Regular languages are closed under regular substitution.*[1]

Proof The basic idea of the automaton construction is to replace every a-transition in A by an ϵ-transition to a distinct copy of A_a, and for every final state of A_a add an ϵ-transition to the target of the original a-transition.[2] □

Beyond Finite Automata

It is well known that not every formal language is regular, i.e., recognized by a finite automaton. For example, the language $L_{ab} = \{a^n b^n : n \geq 1\}$ cannot be recognized by any finite automaton. Why is this? Strings from this language can be arbitrarly long and to recognize them a machine needs to count whether the number of letters 'a' is equal to the number of letters 'b'. A string from L_{ab} can start with any number of letters 'a' so the corresponding machine needs to be able to memorize an arbitrarily large natural number. To do this a machine has to be equipped with an unbounded internal memory. However, a finite automaton with k states can remember only numbers smaller than k. This claim is precisely formulated in the following lemma, which implies that the language L_{ab} is not regular.

Theorem A.5 (Pumping Lemma for Regular Languages) *For any infinite regular language $L \subseteq A^*$ there exists a natural number n such that for every word $\alpha \in L$, if $lh(\alpha) \geq n$, then there are $x, y, z \in A^*$ such that:*

1. $\alpha = xyz$;
2. $y \neq \varepsilon$;
3. $lh(xz) \leq n$;
4. *For every* $k \geq 0$ *the string* xy^kz *is in* L.

Push-down Automata

To account for languages which are not regular we need to extend the concept of a finite automaton. A push-down automaton (PDA) is a finite automaton that can make use of a stack (internal memory). The definition follows.

[1] A substitution is regular if the substituted languages are regular.

[2] The interested reader can see Algorithm 4.2.7 of Meduna 2000 for a complete description (apparently the only automaton proof for regular substitution in the literature at all). The work is a comprehensive dictionary of proof by automaton.

Definition A.5 A nondeterministic push-down automaton (PDA) is a tuple $(A, \Gamma, \#, Q, q_s, F, \delta)$, where:

- A is an input alphabet;
- Γ is a stack alphabet;
- $\# \notin \Gamma$ is a stack-initial symbol, an empty stack consists only of it;
- Q is a finite set of states;
- $q_s \in Q$ is an initial state;
- $F \subseteq Q$ is a set of accepting states;
- $\delta : Q \times (A \cup \{\varepsilon\}) \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma^*)$ is a transition function. We denote a single transition by: $(q, a, n) \xrightarrow{H} (p, \gamma)$, if $(p, \gamma) \in \delta(q, a, n)$, where $q, p \in Q, a \in A, n \in \Gamma, \gamma \in \Gamma^*$.

If $H = (A, \Gamma, \#, Q, q_s, q_a, \delta)$ is a PDA and for every $a \in A$, $q \in Q$, and $\gamma \in \Gamma$, $\mathrm{card}(\delta(q, a, \gamma)) \leq 1$ and $\delta(q, \varepsilon, \gamma) = \emptyset$, then H is a *deterministic push-down automaton (DPDA)*. Graphically, we extend the notion of finite automata by labeling each transition: with x, y/w, where x is the current input the machine reads (i.e., the element under consideration), y is the top element of the stack, and w is the element which will be put on the top of the stack next.

The language recognized by a PDA H is the set of strings accepted by H. A string w is accepted by H if and only if starting in the initial state q_0 with the empty stack and reading the string w, the automaton H terminates in an accepting state $p \in F$.

Definition A.6 We say that a language $L \subseteq A^*$ is context-free if and only if there is a PDA H such that $L = L(H)$.

Observe that (nondeterministic) PDAs accept a larger class of languages than DPDAs. For instance, the language consisting of palindromes is context-free but cannot be recognized by any DPDA as a machine needs to 'guess' which is the middle letter of every string.

Example A.2 Obviously, the class of all context-free languages is larger than the class of all regular languages. For instance, the language $L_{ab} = \{a^n b^n : n \geq 1\}$, which we argued to be nonregular, is context-free. To show this we will construct a PDA H such that $L_{ab} = L(H)$. H recognizes L_{ab} reading every string from left to right and pushes every occurrence of the letter 'a' to the top of the stack. After finding the first occurrence of the letter 'b', the automaton H pops an 'a' off the stack when reading each 'b'. H accepts a string if after processing all of it the stack is empty. See Fig. A.4.

Formally, let $H = (A, \Gamma, \#, Q, q_s, F, \delta)$, where $A = \{a, b\} = \Gamma$, $Q = \{q_s, q_1, q_2, q_a\}$, $F = \{q_a\}$, and the transition function is specified in the following way:

- $(q_s, a, \#) \xrightarrow{H} (q_s, \#a)$;
- $(q_s, a, a) \xrightarrow{H} (q_s, aa)$;
- $(q_s, b, a) \xrightarrow{H} (q_1, \varepsilon)$;
- $(q_s, b, \#) \xrightarrow{H} (q_2, \#)$;

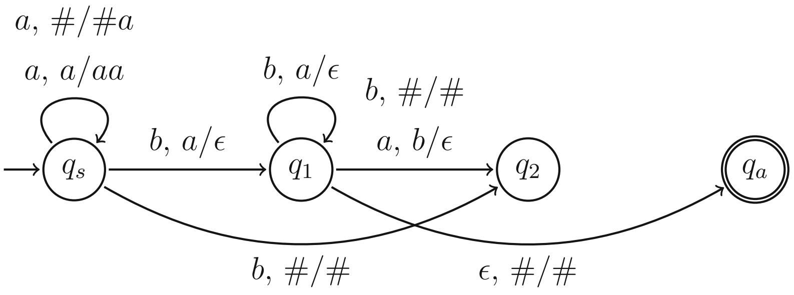

Fig. A.4 PDA recognizing language L_{ab}

- $(q_1, \varepsilon, \#) \xrightarrow{H} (q_a, \#)$;
- $(q_1, b, a) \xrightarrow{H} (q_1, \varepsilon)$;
- $(q_1, b, \#) \xrightarrow{H} (q_2, \#)$;
- $(q_1, a, \#) \xrightarrow{H} (q_2, \varepsilon)$.

Theorem A.6 *Context-free languages are closed under substitution (with both regular and context-free languages).*

Proof The proof idea is the same as with regular languages: given a context-free language L with alphabet Σ and a substitution s mapping $a \in \Sigma$ to a context-free language L_a take the context-free grammar, CFG, G generating L and replace each a (each terminal in the production rules) with the start symbol S_a of G_a generating L_a. This yields a CFG generating $s(L)$. The equivalent PDA construction is also very similar to the construction for finite automata.[3] □

Theorem A.7 *Context-free languages are closed under concatenation.*

Context-free languages, CFLs, are also closed under union, but are not closed under the remaining Boolean operations of intersection and complementation (thus, sometimes performing these operations with CFLs produces languages in yet stronger classes higher up the hierarchy with *context sensitivity*, which we do not discuss further here). Suppose CFLs were closed under intersection. Let $L_1 = \{a^n b^n c^m\}$ and $L_2 = \{a^m b^n c^n\}$. These are both context-free since in each one only two numbers must be matched. But their intersection $L = \{a^n b^n c^n\}$ is canonically non-context-free. Since closure under complementation and union together yield intersection closure by De Morgan's laws, we know the former also cannot hold.

Context-free languages also have restricted descriptive power. For example, the language $L_{abc} = \{a^k b^k c^k : k \geq 1\}$ is not context-free. This fact follows from the extended version of the pumping lemma.

[3] And it can be found (uniquely, as far as we know) in Meduna 2000.

Theorem A.8 (Pumping Lemma for Context-Free Languages) *For every context-free language $L \subseteq A^*$ there is a natural number k such that for each $w \in L$, if $lh(w) \geq k$, then there are $\beta_1, \beta_2, \gamma_1, \gamma_2, \eta$ such that:*

- $\gamma_1 \neq \varepsilon$ *or* $\gamma_2 \neq \varepsilon$*;*
- $w = \beta_1 \gamma_1 \eta \gamma_2 \beta_2$*;*
- *for every* $m \in \omega$*:* $\beta_1 \gamma_1^m \eta \gamma_2^m \beta_2 \in L$.

By extending push-down automata with more memory (e.g., one additional stack), we reach the realm of Turing machines.

A.2.2 Turing Machines

The basic device of computation in this book is a multitape Turing (1936) machine. Most of the particularities of Turing machines are not of direct interest to us. Nevertheless, we will review the basic idea. A *multitape Turing machine* consists of a read-only *input tape*, a read and write *working tape*, and a write-only *output tape*. Every tape is divided into cells scanned by the *read-write head* of the machine. Each cell contains a symbol from some finite alphabet. The tapes are assumed to be arbitrarily extendable to the right. At any time the machine is in one of a finite number of *states*. The actions of a Turing machine are determined by a finite *program* which determines, according to the current *configuration* (i.e., the state of the machine and the symbols in the cells being scanned), which action should be executed next. A *computation* of a Turing machine thus consists of a series of successive configurations. A Turing machine is *deterministic* if its state transitions are uniquely defined, otherwise it is *nondeterministic*. Therefore, a deterministic Turing machine has a single computation path (for any particular input), and a nondeterministic Turing machine has a computation tree. A Turing machine *accepts an input* if its computation on that input halts after a finite time in an accepting state. It *rejects an input* if it halts in a rejecting state.

Definition A.7 Let Γ be some finite *alphabet* and $L \subseteq \Gamma^*$ a language. We say that a *deterministic Turing machine, M*, *decides* L if for every $x \in \Gamma^*$ M halts in the accepting state on x whenever $x \in L$ and in the rejecting state otherwise. A *nondeterministic Turing machine, M*, *recognizes* L if for every $x \in L$ there is a computation of M which halts in the accepting state and there is no such computation for any $x \notin L$.

It is important to notice that nondeterministic Turing machines recognize the same class of languages as deterministic ones. This means that for every problem which can be recognized by a nondeterministic Turing machine there exists a deterministic Turing machine deciding it.

Theorem A.9 *If there is a nondeterministic Turing machine N recognizing a language L, then there exists a deterministic Turing machine M for language L.*

Proof The basic idea for simulating N is as follows. Machine M considers all computation paths of N and simulates N on each of them. If N halts on a given computation path in an accepting state, then M also accepts. Otherwise, M moves to consider the next computation path of N. M rejects the input if machine N does not halt in an accepting state at any computation path. □

The length of an accepting computation of a deterministic Turing machine is, in general, exponential in the length of the shortest accepting computation of the nondeterministic Turing machine as a deterministic machine has to simulate all possible computation paths of the nondeterministic machine.[4] The question whether or not this simulation can be done without exponential growth in computation time leads us to computational complexity theory.

A.2.3 Complexity Classes

Let us start our complexity considerations with the notation used for comparing the growth rates of functions.

Definition A.8 Let $f, g : \omega \longrightarrow \omega$ be any functions. We say that $f = O(g)$ if there exists a constant $c > 0$ such that $f(n) \leq cg(n)$ for almost all (i.e., all but finitely many) n.

Let $f : \omega \longrightarrow \omega$ be a natural number function. TIME(f) is the class of languages (problems) which can be recognized by a deterministic Turing machine in a time bounded by f with respect to the length of the input. In other words, $L \in$ TIME(f) if there exists a deterministic Turing machine such that for every $x \in L$, the computation path of M on x is shorter than $f(n)$, where n is the length of x. TIME(f) is called a *deterministic computational complexity class*. A *nondeterministic complexity class*, NTIME(f), is the class of languages L for which there exists a nondeterministic Turing machine M such that for every $x \in L$ all branches in the computation tree of M on x are bounded by $f(n)$ and moreover M decides L. One way of thinking about a nondeterministic Turing machine bounded by f is that it first guesses the right answer and then deterministically in a time bounded by f checks if the guess is correct.

[4] In general, the simulation outlined above leads to a deterministic Turing machine working in time $O(c^{f(n)})$, where $f(n)$ is the time used by a nondeterministic Turing machine solving the problem and $c > 1$ is a constant depending on that machine (see, e.g., Papadimitriou 1993, p.49 for details).

SPACE(f) is the class of languages which can be recognized by a deterministic machine using at most $f(n)$ cells of the working tape. NSPACE(f) is defined analogously.

Below we define the most important and well-known complexity classes, i.e., the sets of languages of related complexity (see Fig. A.5 for illustration). In other words, we can say that a complexity class is the set of problems that can be solved by a Turing machine using $O(f(n))$ of a time or space resource, where n is the size of the input. To estimate these resources mathematically, natural functions have been chosen, like logarithmic, polynomial, and exponential functions. It is well known that polynomial functions grow faster than any logarithmic functions and exponential functions dominate polynomial functions. Therefore, it is commonly believed that problems belonging to logarithmic classes need essentially fewer resources to be solved than problems from the polynomial classes and likewise that polynomial problems are easier than exponential problems.

Definition A.9

- LOGSPACE $= \bigcup_{k\in\omega}$ SPACE($k \log n$)
- NLOGSPACE $= \bigcup_{k\in\omega}$ NSPACE($k \log n$)
- PTIME $= \bigcup_{k\in\omega}$ TIME(n^k)
- NP $= \bigcup_{k\in\omega}$ NTIME(n^k)
- PSPACE $= \bigcup_{k\in\omega}$ SPACE(n^k)
- NPSPACE $= \bigcup_{k\in\omega}$ NSPACE(n^k)
- EXPTIME $= \bigcup_{k\in\omega}$ TIME(k^n)
- NEXPTIME $= \bigcup_{k\in\omega}$ NTIME(k^n)

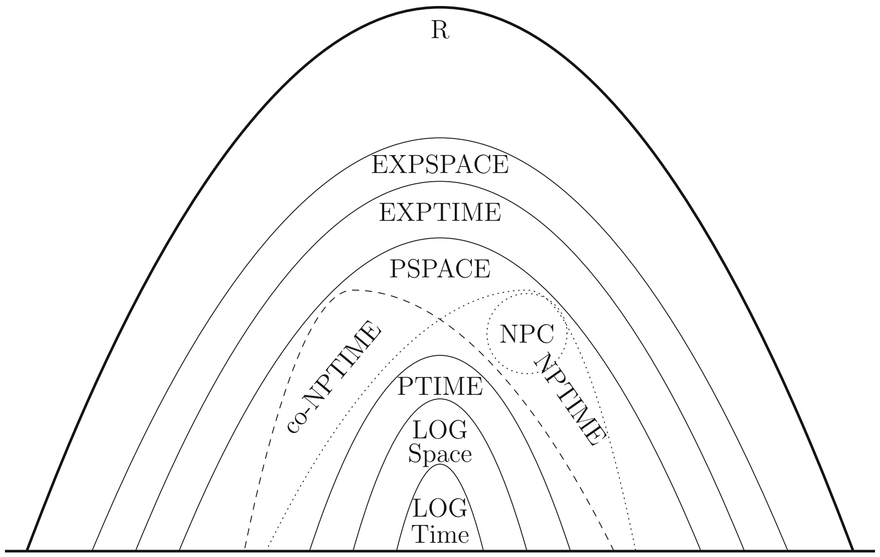

Fig. A.5 Computational complexity classes

If $L \in$ NP, then we say that L *is decidable (computable, solvable) in nondeterministic polynomial time* and likewise for other complexity classes.

It is obvious that for any pair of the complexity classes presented above, the lower one includes the upper one. However, when it comes to the strictness of these inclusions not much is known. One instance that has been proven is for LOGSPACE and PSPACE (see, e.g., Papadimitriou 1993 for so-called Hierarchy Theorems).

The complexity class of all regular languages, i.e., languages recognized by finite automata, is sometimes referred to as REG and equals SPACE(O(1)), the decision problems that can be solved in constant space (the space used is independent of the input size). The complexity class of all languages recognized by push-down automata (i.e., context-free languages) is contained in LOGSPACE.

The question whether PTIME is strictly contained in NP is the famous Millennium Problem—one of the most fundamental problems in theoretical computer science, and in mathematics in general. The importance of this problem reaches well outside the theoretical sciences as the problems in NP are usually taken to be *intractable* or *not efficiently computable* as opposed to the problems in P, which are conceived of as *efficiently solvable*. In this book we take this distinction for granted and investigate semantic constructions in natural language from this perspective (see Chap. 1 for a discussion of this claim).

Moreover, it has been shown by Savitch (1970) that if a nondeterministic Turing machine can solve a problem using $f(n)$ space, an ordinary deterministic Turing machine can solve the same problem in the square of the space. Although it seems that nondeterminism may produce exponential gains in time, this theorem shows that it has a markedly more limited effect on space requirements.

Theorem A.10 (Savitch 1970) *For any function* $f(n) \geq \log(n)$*:*

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

Corollary A.1 PSPACE = NPSPACE

Definition A.10 For any computation class $\mathcal{C}$ we will denote by co-$\mathcal{C}$ the class of complements of languages in $\mathcal{C}$.

Every deterministic complexity class coincides with its complement. It is enough to change accepting states into rejecting states to get a machine computing the complement L from a deterministic machine deciding L itself. However, it is unknown whether NP = co-NP. This is a very important question, as P = NP would imply that NP = co-NP.

A.2.4 Oracle Machines

An *oracle machine* can be described as a Turing machine with a black box, called an oracle, which is able to decide certain decision problems in a single step. More precisely, an oracle machine has a separate write-only oracle tape for writing down queries for the oracle. In a single step, the oracle computes the query, erases its input, and writes its output to the tape.

Definition A.11 If $\mathcal{B}$ and $\mathcal{C}$ are complexity classes, then $\mathcal{B}$ *relativized to* $\mathcal{C}$, $\mathcal{B}^{\mathcal{C}}$, is the class of languages recognized by oracle machines which obey the bounds defining $\mathcal{B}$ and use an oracle for problems belonging to $\mathcal{C}$.

A.2.5 The Polynomial Hierarchy

The Polynomial Hierarchy, PH, is a very well-known hierarchy of classes above NP. It is usually defined inductively using oracle machines and relativization (see, e.g., Papadimitriou 1993) as below.

Definition A.12

(1) $\Sigma_1^P = \mathrm{NP}$;
(2) $\Sigma_{n+1}^P = \mathrm{NP}^{\Sigma_n^P}$;
(3) $\Pi_n^P = \text{co-}\Sigma_n^P$;
(4) $\mathrm{PH} = \bigcup_{i\geq 1} \Sigma_i^P$.

It is known that PH $\subseteq$ PSPACE (see, e.g., Papadimitriou 1993).

However, PH can also be defined using alternating computations. The computation with n-alternation is equivalent to the n-th level of the polynomial hierarchy. As this alternative definition explains some intuitions and also fits nicely in the context of descriptive complexity theory, we explain it below.

Apart from the usual deterministic and nondeterministic Turing machines, one can also be interested in so-called Alternating Turing machines (see, e.g., Chandra et al. 1981).

An Alternating Turing machine is a nondeterministic Turing machine in which the set of nonfinal states is divided into two subsets, existential, E, and universal, A. Let x be an input and consider a computation tree of an Alternating Turing machine with k-alternations on input x. For 3 alternations the exemplar tree would look like the one in Fig. A.6.

Every node of the tree contains a configuration of the machine. Now we have to recursively define the set of accepting configurations. The final configuration (the leaf of the tree) is accepting if and only if it contains the accepting state. Configuration c with the universal state is accepting if and only if all configurations which can be reached from c (that are below the c in the tree) are accepting. Configuration c with the existential state is accepting if and only if at least one configuration which is reachable from c is accepting. We say that Alternating Turing machine accepts x if and only if its starting configuration is an accepting one.

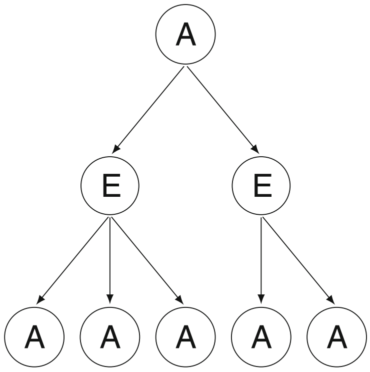

Fig. A.6 Computation tree for k = 3

A Σ_m *machine* is an alternating machine which starts in an existential state and switches between existential and universal states at most $m-1$ times on a single computation path. Observe that a Σ_1 machine is just a nondeterministic Turing machine. A Π_m machine is defined dually, i.e., it starts in a universal state.

Now we can define nonintermediate levels of the polynomial hierarchy as polynomial time computations on a corresponding alternating machine as follows:

$$\Sigma_m^P = \bigcup_{k\in\omega} \Sigma_m TIME(n^k);$$

$$\Pi_n^P = \bigcup_{k\in\omega} \Pi_m TIME(n^k).$$

A.2.6 Stockmeyer Theorem

Theorem A.11 (Stockmeyer 1976) *For any m, Σ_m^1 captures Σ_m^P.*

Recall that Fagin's theorem establishes a correspondence between existential second-order logic and NP. Stockmeyer extends it for the hierarchy of second-order formulae and the polynomial hierarchy. There are many other logical characterizations of complexity classes known (see, e.g., Immerman 1998), for instance that first-order logic is contained in LOGSPACE (see Immerman 1998 Theorem3.1). One of the famous results is the characterization of PTIME over ordered graph structures in terms of fixed-point logic, due to Immerman (1982) and Vardi (1982). Namely, in the presence of a linear ordering of the universe it is possible to use tuples of nodes to build a model of a Turing machine inside the graph and imitate the polynomial time property by a suitable fixed-point sentence (see, e.g., Immerman 1998). One of the most important open problems is the question of which logic $\mathcal{L}$ captures PTIME on graphs if we do not have an ordering of the vertices. Knowing $\mathcal{L}$ one could try to show that $\mathcal{L} \neq \Sigma_1^1$, from which it would follow P $\neq$ NP. Studying the computational complexity of quantifiers can contribute to this question. For instance, Hella et al. (1996) have proven that there is a representation of PTIME queries in terms of fixed-point logic enriched by the quantifier which holds on a randomly chosen finite structure with a probability approaching one as the size of the structure increases. However, Hella (1996) has shown that on unordered finite models, PTIME is not the extension of fixed-point logic by finitely many generalized quantifiers.

A.2.7 Reductions and Complete Problems

The intuition that some problems are more difficult than others is formalized in complexity theory by the notion of a *reduction*. We will use only polynomial time many-one Karp 1972 reductions.

Definition A.13 We say that a function $f : A \longrightarrow A$ is a *polynomial-time computable function* iff there exits a deterministic Turing machine computing $f(w)$ for every $w \in A$ in polynomial time.

Definition A.14 A *problem* $L \subseteq \Gamma^*$ *is polynomial reducible to a problem* $L' \subseteq \Gamma^*$ if there is a polynomial-time computable function $f : \Gamma^* \longrightarrow \Gamma^*$ from strings to strings, such that

$$w \in L \iff f(w) \in L'.$$

We will call such a function f a *polynomial time reduction* of L to L'.

Definition A.15 A language L is complete for a complexity class $\mathcal{C}$ if $L \in \mathcal{C}$ and every language in $\mathcal{C}$ is reducible to L.

Intuitively, if L is complete for a complexity class $\mathcal{C}$, then it is among the hardest problems in this class. The theory of complete problems was initiated by a seminal result of Cook (1971), who proved that the satisfiability problem for propositional formulae, SAT, is complete for NP. Many other now famous problems were then proven to be NP-complete by Karp (1972)—including some versions of satisfiability, like 3SAT (the restriction of SAT to formulae in conjunctional normal form such that every clause contains 3 literals), as well as some graph problems, e.g., CLIQUE, which we define below. The book of Garey and Johnson (1990) contains a list of NP-complete problems.

Example A.3 Let us give an example of a polynomial reduction. We will prove that the problems INDEPENDENT SET and CLIQUE are NP-complete by reducing 3SAT to the first one and then showing that they are equivalent. We define other versions of the CLIQUE problem and use them to prove some complexity results for quantifiers in Chap. 7.

Definition A.16 Let $G = (V, E)$ be a graph and take a set $Cl \subseteq V$. We say that Cl is *independent* if there is no $(i, j) \in E$ for every $i, j \in Cl$.

Definition A.17 The *problem INDEPENDENT SET* can now be formulated as follows. Given a graph $G = (V, E)$ and a natural number k, determine whether there is an independent set in G of cardinality at least k.

Definition A.18 Let $G = (V, E)$ be a graph and take a set $Cl \subseteq V$. We say that Cl is a *clique* if there is $(i, j) \in E$ for every $i, j \in Cl$.

Definition A.19 The *problem CLIQUE* can now be formulated as follows. Given a graph $G = (V, E)$ and a natural number k, determine whether there is a clique in G of cardinality at least k.

Theorem A.12 *INDEPENDENT SET is NP-complete.*

Proof First we have to show that INDEPENDENT SET belongs to NP. Once we have located k or more vertices which form an independent set, it is trivial to verify that they do; this is why the clique problem is in NP.

To show NP-hardness, we will reduce 3SAT to INDEPENDENT SET. Assume that our input is a set of clauses in the form of 3SAT:
$Z = \{(\ell_1^1 \vee \ell_2^1 \vee \ell_3^1), \ldots, (\ell_1^m \vee \ell_2^m \vee \ell_3^m)\}$, where ℓ_i^j is a literal. We construct (G, k) such that:

- $k = m$;
- $G = (V, E)$, where:
  - $V = \{v_{ij} \mid i = 1, \ldots, m;\ j = 1, 2, 3\}$;
  - $E = \{(v_{ij}, v_{\ell k}) \mid i = \ell;\ \ell_i^j = \neg \ell_\ell^k\}$.

To complete the proof it suffices to observe that in graph G there is an independent set of cardinality k if and only if the set Z is satisfiable. □

Theorem A.13 *CLIQUE is NP-complete.*

Proof By the reduction which maps (G, k) to $(\bar{G}, k)$, where $\bar{G}$ is the complement of graph G. □

A.2.8 Intermediate Problems

There are problems in NP that are neither in P nor NP-complete. Such problems are called NP-intermediate, and the class of such problems is called NPI. Ladner (1975) proved the following seminal result:

Theorem A.14 *If P $\neq$ NP, then NPI is not empty.*

Therefore, P = NP if and only if NPI is empty.

Assuming P $\neq$ NP Ladner constructed an artificial NPI problem. Schaefer (1978) proved a dichotomy theorem for Boolean satisfiability, therefore providing conditions under which classes of constrained Boolean satisfiability problems cannot be in NPI. It remains an interesting open question whether there are some natural problems in NPI (see, e.g., Grädel et al. 2007).

A.2.9 The Exponential Time Hypothesis

Definition A.20 (Flum and Grohe 2006) Let $f, g : \omega \to \omega$ be computable functions. Then $f \in o(g)$ (also denoted $f(n) \in o(g(n))$) if there is a computable function h such that for all $\ell \geq 1$ and $n \geq h(\ell)$, we have:

$$f(n) \leq \frac{g(n)}{\ell}.$$

Alternatively, the following definition is equivalent. We have that $f \in o(g)$ if there exists $n_0 \in \omega$ and a computable function $\iota : \omega \to \omega$ that is nondecreasing and unbounded such that for all $n \geq n_0$:

$$f(n) \leq \frac{g(n)}{\iota(n)}.$$

Exponential Time Hypothesis:

3-SAT cannot be solved in time $2^{o(n)}$, where n denotes the number of variables in the input formula.

The following result, which we use to prove the existence of intermediate Ramsey quantifier, is an example of a lower bound based on the ETH.

Theorem A.15 (Chen et al. 2005) *Assuming the ETH, there is no $f(k)m^{o(k)}$ time algorithm for k-CLIQUE, where m is the size of the input graph and where f is a computable function.*

References

Chandra, A. K., Kozen, D. C., & Stockmeyer, L. J. (1981). Alternation. *Journal of the ACM*, *28*(1), 114–133.

Chen, J., Chor, B., Fellows, M., Huang, X., Juedes, D., Kanj, I. A., & Xia, G. (2005). Tight lower bounds for certain parameterized NP-hard problems. *Information and Computation*, *201*(2), 216–231.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the Third Annual ACM Symposium on Theory of Computing* (pp. 151–158). New York: ACM Press.

Cooper, B. S. (2003). *Computability Theory*. Chapman Hall/CRC Mathematics Series. Chapman & Hall/CRC.

Flum, J., & Grohe, M. (2006). *Parameterized Complexity Theory*. Berlin: Springer.

Grädel, E., Kolaitis, P. G., Libkin, L., Marx, M., Spencer, J., Vardi, M. Y., et al. (2007). *Finite Model Theory and Its Applications*. An EATCS Series. Springer: Texts in Theoretical Computer Science.

Garey, M. R., & Johnson, D. S. (1990). *Computers and Intractability. A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman.

Hella, L. (1996). Logical hierarchies in PTIME. *Information and Computation*, *129*(1), 1–19.

Hella, L., Kolaitis, P. G., & Luosto, K. (1996). Almost everywhere equivalence of logics in finite model theory. *Bulletin of Symbolic Logic*, *2*(4), 422–443.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2000). *Introduction to Automata Theory, Languages, and Computation* (2nd ed.). Addison Wesley.

Immerman, N. (1982). Relational queries computable in polynomial time (extended abstract). In *STOC '82: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing* (pp. 147–152). New York: ACM Press.

Immerman, N. (1998). *Descriptive Complexity*. Texts in Computer Science. New York: Springer.

Karp, R. M. (1972). Reducibility among combinatorial problems. In R. E. Miller & J. W. Thatcher (Eds.), *Complexity of Computer Computations* (pp. 85–103). Plenum Press.

Kozen, D. C. (2006). *Theory of Computation: Classical and Contemporary Approaches*. Texts in Computer Science. London: Springer.

Ladner, R. E. (1975). On the structure of polynomial time reducibility. *Journal of the ACM*, *22*(1), 155–171.

Meduna, A. (2000). *Automata and Languages: Theory and Applications*. Springer.

Papadimitriou, C. H. (1993). *Computational Complexity*. Addison Wesley.

Partee, B., Meulen, A., & Wall, R. (1990). *Mathematical Methods in Linguistics*. Studies in Linguistics and Philosophy. Springer.

Savitch, W. (1970). Relationship between nondeterministic and deterministic tape classes. *Journal of Computer and System Sciences*, *4*, 177–192.

Schaefer, T. J. (1978). The complexity of satisfiability problems. In *STOC '78: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing* (pp. 216–226). New York: ACM Press.

Stockmeyer, L. J. (1976). The polynomial-time hierarchy. *Theoretical Computer Science, 3*(1), 1–22.

Turing, A. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, *42*(2), 230–265.

Vardi, M. Y. (1982). The complexity of relational query languages. In *STOC'82: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing* (pp. 137–146). New York: ACM Press.

Index

A
Arithmetic of addition, *see* Presburger Arithmetic

B
van Benthem problem, 170

C
CLIQUE, 112, 131, 204
co-NP, 201
Cobham-Edmonds Thesis, 11
Collective lifts
  determiner fitting operator, 171
  existential modifier, 169
  neutral modifier, 171
Collective properties, 167
Combined complexity, 104
Complexity class, 199
  captured by logic, 103
  relativized, 202
Context-free language, 196, 201

D
Data complexity, *see* model-checking
Descriptive complexity, 102

E
Ehrenfeucht sentence, 108
Ehrenfeucht-Fraïssé games, 28
Exponential Time Hypothesis, 118
Expression complexity, 104
EXPTIME, 200

F
Fagin's Theorem, 13, 103
Finite automata, 47, 193
  acyclic, 47
  deterministic, 193
  nondeterministic, 192
F-large, 115
Frege boundary, 95
Frege Thesis, 96

G
Graph, 104

H
Hierarchy of second-order formulae, 103
Hintikka's sentence, 111, 144, 146
  strong reading, 145
  two-way reading, 146
  weak readings, 145
Hintikka's Thesis, 145

I
Isomorphism, 25

L
Logic
  equivalent, 29
  extended by quantifier, 27
  first-order, 27
  second-order, 29
LOGSPACE, 200

M
Marr's levels of explanation, 52
Meaning
  inferential, 15
  referential, 6, 7
Model, 25
  finite, 104
    encoding, 104
Model-checking, 103

N
Negation normal sentence, 151
NEXPTIME, 200
NLOGSPACE, 200
NP, 200
NP-complete, 10
NP-hard, 10
NPSPACE, 200

P
P-cognition Thesis, 137
P=NP, 10, 201
Polynomial Hierarchy, 202
Polynomial time
  computable function, 204
  many-one reduction, 204
Presburger Arithmetic, 47
PSPACE, 200
PTIME, 10, 200
Pumping Lemma
  context-free languages, 198
  regular languages, 195
Push-down automata, 47
  deterministic, 196
  nondeterministic, 196

Q
Q-large, 113
Quantifier
  additive, *see* semi-linear
  Aristotelian, 44
  automata, 44
  Boolean combinations, 30, 106
  bounded, 115
  branching, 105, 108, 111, 117, 144
    proportional, 109
  cardinal, 44
  CE, 33
  collective reading, 166
  complexity, 105
  conservativity, 32, 172
    collective, 173
  constant-log bounded, 118
  cumulation, 90, 106
  definability, 27, 29, 177
  distributive reading, 166
  divisibility, 45, 47
  domain independent, 31
  first-order, 61
  generalized, 26
    satisfaction relation, 27
    second-order, 174
  Henkin, *see* branching
  higher-order, 61
  iteration, 88, 106, 148
  Lindström, 25, 26
  monadic, 25
  monotonicity, 36
  NP-complete, 106
  NP-hard, 106
  partially ordered, *see* branching
  polyadic, 25, 88
  polyadic lift, 88
  proportional, 45, 110, 132, 149
  Ramsey, 111, 128
    counting, 112, 131
    proportional, 113, 132
  relativization, 31, 132, 134
  resumption, 90, 106, 117
  semi-linear, 47
  topic neutral, 26
  vectorization, *see* resumption

R
Reciprocal expressions, 123, 160
  intermediate alternative reading, 125
  intermediate reading, 125
  strong alternative reading, 125
  strong reading, 125
  weak alternative reading, 125
  weak reading, 125

Reciprocal lift, 129
  intermediate, 129
  strong, 128
  weak, 129
Regular language, 193, 201
Ristad's Thesis, 14, 182

S
3SAT, 204
SAT, 204
Skolem function, 144
Stockmeyer's Theorem, 203
Strong Meaning Hypothesis, 127, 136
Substitution property, 29

T
Turing machine, 198
  oracle, 201
Type-theory, 168