Exercise 2.29 (NP-complete promise problems in coNP (following [68])) Consider the promise problem xSAT, having instances that are pairs of CNF formulae. The yes-instances consists of pairs (ϕ_1, ϕ_2) such that ϕ_1 is satisfiable and ϕ_2 is unsatisfiable, whereas the no-instances consists of pairs such that ϕ_1 is unsatisfiable and ϕ_2 is satisfiable.

1. Show that xSAT is in the intersection of (the promise problem classes that are analogous to) $\mathcal{NP}$ and co$\mathcal{NP}$.

2. Prove that any promise problem in $\mathcal{NP}$ is Cook-reducible to xSAT. In designing the reduction, recall that queries that violate the promise may be answered arbitrarily.

   Guideline: Show a reduction of SAT to xSAT. Specifically, show that the search problem associated with SAT is Cook-reducible to xSAT, by following the ideas of the proof of Proposition 2.14. Actually, we need a more careful implementation of the search process. Suppose that we know (or assume) that τ is a prefix of a satisfying assignment to ϕ, and we wish to extend τ by one bit. Then, for each $\sigma \in \{0,1\}$, we construct a formula, denoted ϕ'_σ, by setting the first $|\tau|+1$ variables of ϕ according to the values $\tau\sigma$. We query the oracle about the pair (ϕ'_1, ϕ'_0), and extend τ accordingly (i.e., we extend τ by the value 1 if and only if the answer is positive). Note that if both ϕ'_1 and ϕ'_0 are satisfiable then it does not matter which bit we use in the extension, whereas if exactly one formula is satisfiable then the oracle answer is reliable.

3. Pinpoint the source of failure of the proof of Theorem 2.33 when applied to the reduction provided in the previous item.

Chapter 3

Variations on P and NP

In this chapter we consider variations on the complexity classes P and NP. We refer specifically to the non-uniform version of P, and to the Polynomial-time Hierarchy (which extends NP). These variations are motivated by relatively technical considerations, and the resulting classes are referred to quite frequently in the literature.

Summary: Non-uniform polynomial-time (P/poly) captures efficient computations that are carried out by devices that handle specific input lengths. The basic formalism ignore the complexity of constructing such devices (i.e., a uniformity condition). A finer formalism that allows to quantify the amount of non-uniformity refers to so called "machines that take advice."

The Polynomial-time Hierarchy (PH) generalizes NP by considering statements expressed by a quantified Boolean formula with a fixed number of alternations of existential and universal quantifiers. It is widely believed that each quantifier alternation adds expressive power to the class of such formulae.

The two different classes are related by showing that if NP is contained in P/poly then the Polynomial-time Hierarchy collapses to its second level. Assuming the latter collapse does not occur, this means that there is hope to resolve the P-vs-NP Question by showing that NP is not contained in "non-uniform P" (P/poly).

3.1 Non-uniform polynomial-time (P/poly)

In this section we consider two formulations of the notion of non-uniform polynomial-time, presenting two models of non-uniform (polynomial-time) computing devices. These models are derived by specializing the treatment of non-uniform computing devices, provided in Section 1.2.4, to the case of polynomially bounded complexities. It turns out that both models allow to solve the same class of computational

problems, which is a strict superset of the class of problems solvable by polynomial-time algorithms.

The two models discussed below are the two models presented in Section 1.2.4 (i.e., Boolean circuits and "machines that take advice"). We will focus on the restriction of both models to the case of polynomial complexities, considering (non-uniform) polynomial-size circuits and polynomial-time algorithms that take (non-uniform) polynomially bounded advice.

The main motivation for considering non-uniform polynomial-size circuits is that their limitations imply analogous limitations on polynomial-time algorithms. The hope is that, as is often the case in mathematics and Science, disposing of an auxiliary condition that is not well-understood (i.e., uniformity) may turn out fruitful. In particular, it may facilitate a low-level analysis of the evolution of a computation, by using combinatorial techniques. This hope has materialized in the study of restricted classes of circuits (see Sections B.2.2 and B.2.3).

Polynomial-time algorithms that take polynomially bounded advice are useful in modeling auxiliary information available to possible efficient strategies that are of interest to us. Indeed, the typical cases are the modeling of adversaries in the context of cryptography and the modeling of arbitrary randomized algorithms in the context of derandomization. Furthermore, the model of polynomial-time algorithms that take advice allows for a quantitative study of the amount of non-uniformity, ranging from zero to polynomial.

3.1.1 Boolean Circuits

We refer the reader to §1.2.4.1 for a definition of (families of) Boolean circuits and the functions computed by them. For concreteness and simplicity, we assume throughout this section that all circuits has bounded fan-in. We highlight the following result stated in §1.2.4.1:

Theorem 3.1 (circuit evaluation): *There exists a polynomial-time algorithm that, given a circuit $C : \{0,1\}^n \to \{0,1\}^m$ and an n-bit long string x, returns $C(x)$.*

Recall that the algorithm works by performing the value-determining process described when defining the value of the circuit vertices on a given input.

Circuit size as a complexity measure. We recall the definitions of circuit complexity presented in to §1.2.4.1: The size of a circuit is defined as the number of edges, and the length of its description is almost linear in the latter; that is, a circuit of size s is commonly described by the list of its edges and the labels of its vertices, which means that its description length is $O(s \log s)$. We are interested in families of circuits that solve computational problems, and say that the circuit family $(C_n)_{n\in\mathbb{N}}$ computes the function $f : \{0,1\}^* \to \{0,1\}^*$ if for every $x \in \{0,1\}^*$ it holds that $C_{|x|}(x) = f(x)$. The size complexity of this family is the function $s : \mathbb{N} \to \mathbb{N}$ such that $s(n)$ is the size of C_n. The circuit complexity of a function f, denoted s_f, is the size complexity of the smallest family of circuits that computes f. An equivalent alternative follows.

Definition 3.2 (circuit complexity): *The* circuit complexity of $f : \{0,1\}^* \to \{0,1\}^*$ *is the function* $s_f : \mathbb{N} \to \mathbb{N}$ *such that* $s_f(n)$ *is the size of the smallest circuit that computes the restriction of* f *to* n*-bit strings.*

We stress that non-uniformity is implicit in this definition, because no conditions are made regarding the relation between the various circuits used to compute the function on different input lengths.

We will be interested in the class of problems that are solvable by families of polynomial-size circuits. That is, a problem is solvable by polynomial-size circuits if it can be solved by a function f that has polynomial circuit complexity (i.e., there exists a polynomial p such that $s_f(n) \leq p(n)$, for every $n \in \mathbb{N}$).

A detour: uniform families. A family of *polynomial-size* circuits $(C_n)_n$ is called uniform if given n one can construct the circuit C_n in poly(n)-time. More generally:

Definition 3.3 (uniformity): *A family of circuits* $(C_n)_n$ *is called* uniform *if there exists an algorithm* A *that on input* n *outputs* C_n *within a number of steps that is polynomial in the size of* C_n*.*

We note that stronger notions of uniformity have been considered. For example, one may require the existence of a polynomial-time algorithm that on input n and v, returns the label of vertex v as well as the list of its children (or an indication that v is not a vertex in C_n). For further discussion see Section 5.2.3.

Proposition 3.4 *If a problem is solvable by a uniform family of polynomial-size circuits then it is solvable by a polynomial-time algorithm.*

As was hinted in §1.2.4.1, the converse holds as well. The latter fact follows easily from the proof of Theorem 2.20 (see also the proof of Theorem 3.6).

Proof: On input x, the algorithm operates in two stages. In the first stage, it invokes the algorithm guaranteed by uniformity condition, on input $n \stackrel{\text{def}}{=} |x|$, and obtains C_n. Next, it invokes the circuit evaluation algorithm (asserted in Theorem 3.1) on input C_n and x, and obtains $C_n(x)$. Since the size and the description length of C_n are polynomial in n, it follows that each stage of our algorithm runs in polynomial time (in $n = |x|$). Thus, the algorithm emulates the computation of $C_{|x|}(x)$, and does so in time polynomial in the length of its own input (i.e., x). ■

3.1.2 Machines that take advice

General (non-uniform) families of polynomial-size circuits and uniform families of polynomial-size circuits are two extremes with respect to the "amounts of non-uniformity" in the computing device. Intuitively, in the former, non-uniformity is only bounded by the size of the device, whereas in the latter the amounts of non-uniformity is zero. Here we consider a model that allows to decouple the size

of the computing device from the amount of non-uniformity, which may indeed range from zero to the device's size. Specifically, we consider algorithms that "take a non-uniform advice" that depends only on the input length. The amount of non-uniformity will be defined to equal the length of the corresponding advice (as a function of the input length). Here we specialize Definition 1.12 to the case of polynomial-time algorithms.

Definition 3.5 (non-uniform polynomial-time and $\mathcal{P}/\text{poly}$): *We say that a function f is* computed in polynomial-time with advice of length $\ell : \mathbb{N} \to \mathbb{N}$ *if these exists a polynomial-time algorithm A and an infinite* advice sequence $(a_n)_{n\in\mathbb{N}}$ *such that*

1. *For every $x \in \{0,1\}^*$, it holds that $A(a_{|x|}, x) = f(x)$.*
2. *For every $n \in \mathbb{N}$, it holds that $|a_n| = \ell(n)$.*

We say that a computational problem can be solved in polynomial-time with advice of length ℓ if a function solving this problem can be computed within these resources. We denote by $\mathcal{P}/\ell$ the class of decision problems that can be solved in polynomial-time with advice of length ℓ, and by $\mathcal{P}/\text{poly}$ the union of $\mathcal{P}/p$ taken over all polynomials p.

Clearly, $\mathcal{P}/0 = \mathcal{P}$. But allowing some (non-empty) advice increases the power of the class (as we shall see below), and allowing advice of length comparable to the time complexity yields a formulation equivalent to circuit complexity (see Theorem 3.6). We highlight the greater flexibility available by the formalism of machines that take advice, which allows for separate specification of time complexity and advice length. (This comes at the expense of a more cumbersome formulation when we wish to focus on the case that both measures are equal.)

Relation to families of polynomial-size circuits. As hinted before, the class of problems solvable by polynomial-time algorithms with polynomially bounded advice equals the class of problems solvable by families of polynomial-size circuits. For concreteness, we state this fact for decision problems.

Theorem 3.6 *A decision problem is in $\mathcal{P}/\text{poly}$ if and only if it can solved by a family of polynomial-size circuits.*

More generally, for any function t, the following proof establishes that equivalence of the power of machines having time complexity t and taking advice of length t versus families of circuits of size polynomially related to t.

Proof: Suppose that a problem can be solved by a polynomial-time algorithm A using the polynomially bounded advice sequence $(a_n)_{n\in\mathbb{N}}$. Adapting the proof of Theorem 2.20, observe that the computation of $A(a_{|x|}, x)$ can be emulated by a circuit of $\text{poly}(|x|)$-size, which incorporates $a_{|x|}$ and is given x as input. That is, we construct a circuit C_n such that $C_n(x) = A(a_n, x)$ holds for every $x \in \{0,1\}^n$ (analogously to the way C_x was constructed in the proof of Theorem 2.20, where $C_x(y) = M_R(x, y)$).

On the other hand, given a family of polynomial-size circuits, we obtain a polynomial-time algorithm for emulating this family using advice that provide the description of the relevant circuits. Specifically, we use the evaluation algorithm asserted in Theorem 3.1, while using the circuit description as advice. We use the fact that a circuit of size s can be computed using advice of length $O(s \log s)$, where the log factor is due to the fact that a graph with v vertices and e edges can be described by a string of length $2e \log_2 v$. ■

Another perspective. A set S is called sparse if there exists a polynomial p such that for every n it holds that $|S \cap \{0,1\}^n| \leq p(n)$. We note that $\mathcal{P}/\text{poly}$ equals the class of sets that are Cook-reducible to a sparse set (see Exercise 3.1). Thus, SAT is Cook-reducible to a sparse set if and only if $\mathcal{NP} \subset \mathcal{P}/\text{poly}$. In contrast, SAT is Karp-reducible to a sparse set if and only if $\mathcal{NP} = \mathcal{P}$ (see Exercise 3.10).

The power of $\mathcal{P}/\text{poly}$. In continuation to Theorem 1.13, we prove the following result.

Theorem 3.7 (the power of advice, revisited): *The class $\mathcal{P}/1 \subseteq \mathcal{P}/\text{poly}$ contains $\mathcal{P}$ as well as some undecidable problems.*

Actually, $\mathcal{P}/1 \subset \mathcal{P}/\text{poly}$. Furthermore, by using a counting argument, one can show that for any two polynomially bounded functions $\ell_1, \ell_2 : \mathbb{N} \to \mathbb{N}$ such that $\ell_2 - \ell_1 > 0$ is unbounded, it holds that $\mathcal{P}/\ell_1$ is strictly contained in $\mathcal{P}/\ell_2$; see Exercise 3.2.

Proof: Clearly, $\mathcal{P} = \mathcal{P}/0 \subseteq \mathcal{P}/1 \subseteq \mathcal{P}/\text{poly}$. To prove that $\mathcal{P}/1$ contains some undecidable problems, we review the proof of Theorem 1.13. The latter proof established the existence of uncomputable Boolean function that only depend on their input length. That is, there exists an undecidable set $S \subset \{0,1\}^*$ such that for every pair of equal length strings (x, y) it holds that $x \in S$ if and only if $y \in S$. In other words, for every $x \in \{0,1\}^*$ it holds that $x \in S$ if and only if $1^{|x|} \in S$. But such a set is easily decidable in polynomial-time by a machine that takes one bit of advice; that is, consider the algorithm A and the advice sequence $(a_n)_{n \in \mathbb{N}}$ such that $a_n = 1$ if and only if $1^n \in S$ and $A(a, x) = a$ (for $a \in \{0,1\}$ and $x \in \{0,1\}^*$). Note that indeed $A(a_{|x|}, x) = 1$ if and only if $x \in S$. ■

3.2 The Polynomial-time Hierarchy (PH)

We start with an informal motivating discussion, which will be made formal in Section 3.2.1.

Sets in $\mathcal{NP}$ can be viewed as sets of valid assertions that can be expressed as quantified Boolean formulae using only existential quantifiers. That is, a set S is in $\mathcal{NP}$ if there is a Karp-reduction of S to the problem of deciding whether or not an existentially quantified Boolean formula is valid (i.e., an instance x is mapped by this reduction to a formula of the form $\exists y_1 \cdots \exists y_{m(x)} \phi_x(y_1, ..., y_{m(x)})$).

The conjectured intractability of $\mathcal{NP}$ seems due to the long sequence of existential quantifiers. Of course, if somebody else (i.e., a "prover") were to provide us with an adequate assignment (of the y_i's) whenever such an assignment exists then we would be in good shape. That is, we can efficiently verify proofs of validity of existentially quantified Boolean formulae.

But what if we want to verify the validity of a universally quantified Boolean formulae (i.e., formulae of the form $\forall y_1 \cdots \forall y_m \phi(y_1, ..., y_m)$). Here we seem to need the help of a totally different entity: we need a "refuter" that is guaranteed to provide us with a refutation whenever such exist, and we need to believe that if we were not presented with such a refutation then it is the case that no refutation exists (and hence the universally quantified formulae is valid). Indeed, this new setting (of a "refutation system") is fundamentally different from the setting of a proof system: In a proof system we are only convinced by proofs (to assertions) that we have verified by ourselves, whereas in the "refutation system" we trust the "refuter" to provide evidence against false assertions.[1] Furthermore, there seems to be no way of converting one setting (e.g., the proof system) into another (resp., the refutation system).

Taking an additional step, we may consider a more complicated system in which we use two agents: a "supporter" that tries to provide evidence in favor of an assertion and an "objector" that tries to refute it. These two agents conduct a debate (or an argument) in our presence, exchanging messages with the goal of making us (the referee) rule their way. The assertions that can be proven in this system take the form of general quantified formulae with alternating sequences of quantifiers, where the number of alternations equals the number of rounds of interaction in the said system. We stress that the exact length of each sequence of quantifiers of the same type does not matter, what matters is the number of alternations, denoted k.

The aforementioned system of alternations can be viewed as a two-party game, and we may ask ourselves which of the two parties has a k-move winning strategy. In general, we may consider any (0-1 zero-sum) two-party game, in which the game's position can be efficiently updated (by any given move) and evaluated. For such a fixed game, given an initial position, we may ask whether the first party has a (k-move) winning strategy. It seems that answering this question for some k does not help in answering it for $k+1$. We now turn to formalize the foregoing discussion.

3.2.1 Alternation of quantifiers

In the following definition, the aforementioned propositional formula ϕ_x is replaced by the input x itself. (Correspondingly, the combination of the Karp-reduction and a formula evaluation algorithm are replaced by the verification algorithm V (see Exercise 3.5).) This is done in order to make the comparison to the definition

[1] More formally, in proof systems the soundness condition relies only on the actions of the verifier, whereas completeness also relies on the prover using an adequate strategy. In contrast, in "refutation system" the soundness condition relies on the proper actions of the refuter, whereas completeness does not depend on the refuter's actions.

of $\mathcal{NP}$ more transparent (as well as to fit the standard presentations). We also replace a sequence of Boolean quantifiers of the same type by a single corresponding quantifier that quantifies over a string of corresponding length.

Definition 3.8 (the class Σ_k): *For a natural number k, a decision problem $S \subseteq \{0,1\}^*$ is in Σ_k if there exists a polynomial p and a polynomial time algorithm V such that $x \in S$ if and only if*

$$\exists y_1 \in \{0,1\}^{p(|x|)} \forall y_2 \in \{0,1\}^{p(|x|)} \exists y_3 \in \{0,1\}^{p(|x|)} \cdots Q_k y_k \in \{0,1\}^{p(|x|)} \text{ s.t. } V(x, y_1, ..., y_k) = 1$$

where Q_k is an existential quantifier if k is odd and is a universal quantifier otherwise.

Note that $\Sigma_1 = \mathcal{NP}$ and $\Sigma_0 = \mathcal{P}$. The Polynomial-time Hierarchy, denoted $\mathcal{PH}$, is the union of all the aforementioned classes (i.e., $\mathcal{PH} = \cup_k \Sigma_k$), and Σ_k is often referred to as the k^{th} level of $\mathcal{PH}$. The levels of the Polynomial-time Hierarchy can also be defined inductively, by defining Σ_{k+1} based on $\Pi_k \stackrel{\text{def}}{=} \text{co}\Sigma_k \stackrel{\text{def}}{=} \{\{0,1\}^* \setminus S : S \in \Sigma_k\}$.

Proposition 3.9 *The set S is in Σ_{k+1} if and only if there exists a polynomial p and a set $S' \in \Pi_k$ such that $S = \{x : \exists y \in \{0,1\}^{p(|x|)} \text{ s.t. } (x,y) \in S'\}$.*

Proof: Suppose that S is in Σ_{k+1} and let p and V be as in Definition 3.8. Then define S' as the set of pairs (x,y) such that $|y| = p(|x|)$ and

$$\forall y_1 \in \{0,1\}^{p(|x|)} \exists y_2 \in \{0,1\}^{p(|x|)} \cdots Q_k y_k \in \{0,1\}^{p(|x|)} \text{ s.t. } V(x, y, y_1, ..., y_k) = 1\,.$$

Note that $x \in S$ if and only if there exists $y \in \{0,1\}^{p(|x|)}$ such that $(x,y) \in S'$, and that $S' \in \Pi_k$ (by replacing V with $1 - V$; see Exercise 3.4).

On the other hand, suppose that for some polynomial p and a set $S' \in \Pi_k$ it holds that $S = \{x : \exists y \in \{0,1\}^{p(|x|)} \text{ s.t. } (x,y) \in S'\}$. Then, for some p' and V', it holds that $(x,y) \in S'$ if and only if $|y| = p(|x|)$ and

$$\forall y_1 \in \{0,1\}^{p'(|x|)} \exists y_2 \in \{0,1\}^{p'(|x|)} \cdots Q_k y_k \in \{0,1\}^{p'(|x|)} \text{ s.t. } V'(x, y, y_1, ..., y_k) \neq 1$$

(see Exercise 3.4 again). By suitable encoding (of y and the y_i's as strings of length $\max(p(|x|), p'(|x|))$) and a trivial modification of V', we conclude that $S \in \Sigma_{k+1}$. ■

Determining the winner in k-move games. Definition 3.8 can be interpreted as capturing the complexity of determining the winner in certain *efficient two-party game*. Specifically, we refer to two-party games that satisfy the following three conditions:

1. The parties alternate in taking moves that effect the game's (global) position, where each move has a description length that is bounded by a polynomial in the length of the current position.

2. The current position can be updated in polynomial-time based on the previous position and the current party's move.[2]

3. The winner in each position can be determined in polynomial-time.

A set $S \in \Sigma_k$ can be viewed as the set of initial positions (in a suitable game) for which the first party has a k-move winning strategy. Specifically, $x \in S$ if starting at the initial position x, there exists move y_1 for the first party, such that for every response move y_2 of the second party, there exists move y_3 for the first party, etc, such that after k moves the parties reach a position in which the first party wins, where the final position as well as which party wins in it are determined by the predicate V (in Definition 3.8). That is, $V(x, y_1, ..., y_k) = 1$ if the position that is reached when starting from position x and taking the move sequence $y_1, ..., y_k$ is a winning position for the first party.

The collapsing effect of some equalities. Extending the intuition behind the $\mathcal{NP} \neq \text{co}\mathcal{NP}$ conjecture, it is conjectured that $\Sigma_k \neq \Pi_k$ for every $k \in \mathbb{N}$. The failure of this conjecture causes the collapse of the Polynomial-time Hierarchy to the corresponding level.

Proposition 3.10 *For every $k \geq 1$, if $\Sigma_k = \Pi_k$ then $\Sigma_{k+1} = \Sigma_k$, which in turn implies $\mathcal{PH} = \Sigma_k$.*

The converse also holds (i.e., $\mathcal{PH} = \Sigma_k$ implies $\Sigma_{k+1} = \Sigma_k$ and $\Sigma_k = \Pi_k$).

Proof: Assuming that $\Sigma_k = \Pi_k$, we consider any set S in Σ_{k+1}. By Proposition 3.9, there exists a polynomial p and a set $S' \in \Pi_k$ such that $S = \{x : \exists y \in \{0,1\}^{p(|x|)} \text{ s.t. } (x,y) \in S'\}$. Using the hypothesis, we infer that $S' \in \Sigma_k$, and it follows that $S \in \Sigma_k$ (by collapsing two adjacent existential quantifiers (relying on $k \geq 1$)).[3] We also note that $\Sigma_{k+1} = \Sigma_k$ (or, equivalently, $\Pi_{k+1} = \Pi_k$) implies $\Sigma_{k+2} = \Sigma_{k+1}$ (again by using Proposition 3.9), and thus $\Sigma_{k+1} = \Sigma_k$ implies $\mathcal{PH} = \Sigma_k$. ■

Decision problems that are Cook-reductions to NP. Recall that in Section 2.2.2 we defined two types of optimization problems and showed that under some natural conditions these two types are computationally equivalent (under Cook reductions). Specifically, one type of problems referred to finding a solution that have a value exceeding some given threshold, whereas the second type called for finding optimal solutions. In Section 2.3 we presented several problems of the first type, and proved that they are NP-complete. We note that corresponding

[2] Note that, since we consider a constant number of moves, the length of all possible final positions is bounded by a polynomial in the length of the initial position, and thus all items have an equivalent form in which one refers to the complexity as a function of the length of the initial position. The latter form allows for a smooth generalization to games with a polynomial number of moves (as in Section 5.4), where it is essential to state all complexities in terms of the length of the initial position.

[3] Needless to say, Proposition 3.10 does not seem to hold for $k = 0$.

versions of the second type are believed not to be in NP. For example, we discussed the problem of deciding whether or not a given graph G has a clique of a given size K, and showed that it is NP-complete. In contract, the problem of deciding whether or not K is the maximum clique size of the graph G is not known (and quite unlikely) to be in $\mathcal{NP}$, although it is Cook-reducible to $\mathcal{NP}$. Thus, the class of decision problems that are Cook-reducible to $\mathcal{NP}$ contains many natural problems that are unlikely to be in $\mathcal{NP}$. The Polynomial-time Hierarchy contains all these problems, because Σ_2 contains all problems that are Cook-reductions to $\mathcal{NP}$ (see Exercise 3.3).

Complete problems and a relation to AC0. We note that quantified Boolean formulae with a bounded number of quantifier alternation provide complete problems for the various levels of the Polynomial-time Hierarchy (see Exercise 3.5). We also note the correspondnace between these formulae and (highly uniform) constant-depth circuits of unbounded fan-in that get as input the truth-table of the underlying (quantifier-free) formula (see Exercise 3.6).

3.2.2 Non-deterministic oracle machines

The Polynomial-time Hierarchy is commonly defined in terms of non-deterministic polynomial-time (oracle) machines that are given oracle access to a set of the lower level of the same hierarchy. Such machines are defined by combining the definitions of non-deterministic (polynomial-time) machines (cf. Definition 2.7) and oracle machines (cf. Definition 1.11). Specifically, for an oracle $f : \{0,1\}^* \to \{0,1\}^*$, a non-deterministic machine M, and a string x, one considers the question of whether or not there exists an accepting (non-deterministic) computation of M on input x and access to the oracle f. The class of sets that can be accepted by non-deterministic polynomial-time (oracle) machines with access to f is denoted $\mathcal{NP}^f$. (We note that this notation makes sense because we can associate the class $\mathcal{NP}$ with a collection of machines that lends itself to be extended to oracle machines.) For any class of decision problems $\mathcal{C}$, we denote by $\mathcal{NP}^{\mathcal{C}}$ the union of $\mathcal{NP}^f$ taken over all decision problems f in $\mathcal{C}$. The following result provides an alternative definition of the Polynomial-time Hierarchy.

Proposition 3.11 *For every $k \geq 1$, it holds that $\Sigma_{k+1} = \mathcal{NP}^{\Sigma_k}$.*

Proof: The first direction is straightforward: For any $S \in \Sigma_{k+1}$, let $S' \in \Pi_k$ and p be as in Proposition 3.9; that is, $S = \{x : \exists y \in \{0,1\}^{p(|x|)} \text{ s.t. } (x,y) \in S'\}$. Consider a non-deterministic machine that, on input x, non-deterministically generates $y \in \{0,1\}^{p(|x|)}$ and accepts if and only if $(x,y) \in S'$. Thus, $S \in \mathcal{NP}^{\Pi_k} = \mathcal{NP}^{\Sigma_k}$, where equality holds by letting the oracle machine flip each (binary) answer that is provided by the oracle.[4]

[4] Do not get confused by the fact that the class of oracles may *not* be closed under complementation. From the point of view of the oracle machine, the oracle is merely a function, and the machine may do with its answer whatever it pleases (and in particular negate it).

For the opposite direction, suppose that the non-deterministic polynomial-time oracle machine M accepts S when given oracle access to $S' \in \Sigma_k$. Note that (unlike the machine constructed in the foregoing argument) machine M may issue several queries to S', and these queries may be determined based on previous oracle answers. To simplify the argument, we assume, without loss of generality, that at the very beginning of its execution M guesses (non-deterministic) all oracle answers and accepts only if the actual answers match its guesses. Thus, M's queries to the oracle are determined by its input, denoted x, and its non-deterministic choices, denoted y. We denote by $q^{(i)}(x,y)$ the i^{th} query made by M (on input x and non-deterministic choices y), and by $a^{(i)}(x,y)$ the corresponding (a priori) guessed answer (which is a bit in y). Thus, M accepts x if and only if there exists $y \in \{0,1\}^{\text{poly}(|x|)}$ such that the following two conditions hold:

1. Machine M accepts x, on input x and non-deterministic choices y, provided that all oracle answers fit the correspondingly guessed answers $a^{(i)}(x,y)$.
2. All oracle answers fit the correspondingly guessed answers $a^{(i)}(x,y)$; that is, $a^{(i)}(x,y)=1$ if and only if $q^{(i)}(x,y) \in S'$, for every $i = 1,...,q(x,y)$, where $q(x,y) = \text{poly}(|x|)$ is the number of queries made by M.

Denoting the first event by $A(x,y)$, it follows that $x \in S$ if and only if

$$\exists y \left(A(x,y) \wedge \bigwedge_{i=1}^{q(x,y)} \left((a^{(i)}(x,y)=1) \Leftrightarrow (q^{(i)}(x,y) \in S') \right) \right).$$

Denoting the verification algorithm of S' by V', it holds that $x \in S$ if and only if

$$\exists y \left(A(x,y) \wedge \bigwedge_{i=1}^{q(x,y)} \left((a^{(i)}(x,y)=1) \Leftrightarrow \exists y_1^{(i)} \forall y_2^{(i)} \cdots Q_k y_k^{(i)} \, V'(q^{(i)}(x,y), y_1^{(i)}, ..., y_k^{(i)})=1 \right) \right).$$

The proof is completed by observing that the foregoing expression can be rearranged to fit the definition of Σ_{k+1}. Details follow.

Starting with the foregoing expression, we first pull all quantifiers outside, and obtain a quantified expression with $k+1$ alternations, starting with an existential quantifier.[5] (We get $k+1$ alternations rather than k, because $a^{(i)}(x,y)=0$ introduces an expression of the form $\neg \exists y_1^{(i)} \forall y_2^{(i)} \cdots Q_k y_k^{(i)} \, V'(q^{(i)}(x,y), y_1^{(i)}, ..., y_k^{(i)})=1$, which in turn is equivalent to the expression $\forall y_1^{(i)} \exists y_2^{(i)} \cdots \overline{Q}_k y_k^{(i)} \, \neg V'(q^{(i)}(x,y), y_1^{(i)}, ..., y_k^{(i)})=1$.) Once this is done, we may incorporate the computation of all the $q^{(i)}(x,y)$'s (and $a^{(i)}(x,y)$'s) as well as the polynomial number of invocations of V' (and other logical operations) into the new verification algorithm V. It follows that $S \in \Sigma_{k+1}$. ■

[5] For example, note that for predicates P_1 and P_2, the expression $\exists y \, (P_1(y) \Leftrightarrow \exists z \, P_2(y,z))$ is equivalent to the expression $\exists y \, ((P_1(y) \wedge \exists z \, P_2(y,z)) \vee ((\neg P_1(y) \wedge \neg \exists z \, P_2(y,z)))$, which in turn is equivalent to the expression $\exists y \exists z' \forall z'' \, ((P_1(y) \wedge P_2(y,z')) \vee ((\neg P_1(y) \wedge \neg P_2(y,z'')))$. Note that pulling the quantifiers outside in $\wedge_{i=1}^{t} \exists y^{(i)} \forall z^{(i)} P(y^{(i)}, z^{(i)})$ yields an expression of the type $\exists y^{(1)}, ..., y^{(t)} \forall z^{(1)}, ..., z^{(t)} \wedge_{i=1}^{t} P(y^{(i)}, z^{(i)})$.

A general perspective – what does ${\mathcal{C}_1}^{\mathcal{C}_2}$ mean? By the foregoing discussion it should be clear that the class ${\mathcal{C}_1}^{\mathcal{C}_2}$ can be defined for two complexity classes $\mathcal{C}_1$ and $\mathcal{C}_2$, *provided that $\mathcal{C}_1$ is associated with a class of machines that extends naturally in a way that allows for oracle access.* Actually, the class ${\mathcal{C}_1}^{\mathcal{C}_2}$ *is not defined based on the class $\mathcal{C}_1$ but rather by analogy to it.* Specifically, suppose that $\mathcal{C}_1$ is the class of sets recognizable by machines of certain type (e.g., deterministic or non-deterministic) with certain resource bounds (e.g., time and/or space bounds). Then, we consider analogous oracle machines (i.e., of the same type and with the same resource bounds), and say that $S \in {\mathcal{C}_1}^{\mathcal{C}_2}$ if there exists an adequate oracle machine M_1 (i.e., of this type and resource bounds) and a set $S_2 \in \mathcal{C}_2$ such that $M_1^{S_2}$ accepts the set S.

Decision problems that are Cook-reductions to NP, revisited. Using the foregoing notation, the class of decision problems that are Cook-reductions to $\mathcal{NP}$ is denoted $\mathcal{P}^{\mathcal{NP}}$, and thus is a subset of $\mathcal{NP}^{\mathcal{NP}} = \Sigma_2$ (see Exercise 3.7). In contrast, recall that the class of decision problems that are Karp-reductions to $\mathcal{NP}$ equals $\mathcal{NP}$.

3.2.3 The P/poly-versus-NP Question and PH

As stated in Section 3.1, a main motivation for the definition of $\mathcal{P}/\mathrm{poly}$ is the hope that it can serve to separate $\mathcal{P}$ from $\mathcal{NP}$ (by showing that $\mathcal{NP}$ is not even contained in $\mathcal{P}/\mathrm{poly}$, which is a (strict) superset of $\mathcal{P}$). In light of the fact that $\mathcal{P}/\mathrm{poly}$ extends far beyond $\mathcal{P}$ (and in particular contains undecidable problems), one may wonder if this approach does not run the risk of $\mathcal{NP}$ being in $\mathcal{P}/\mathrm{poly}$ (even if $\mathcal{P} \neq \mathcal{NP}$). Ideally, we would like to know that $\mathcal{NP} \subset \mathcal{P}/\mathrm{poly}$ may occur only if $\mathcal{P} = \mathcal{NP}$ (which means that the Polynomial-time Hierarchy collapses to its zero level). The following result may seem to get close, showing that $\mathcal{NP} \subset \mathcal{P}/\mathrm{poly}$ may occur only if the Polynomial-time Hierarchy collapses to its second level.

Theorem 3.12 *If $\mathcal{NP} \subset \mathcal{P}/\mathrm{poly}$ then $\Sigma_2 = \Pi_2$.*

Recall that $\Sigma_2 = \Pi_2$ implies $\mathcal{PH} = \Sigma_2$ (see Proposition 3.10). Thus, an unexpected behavior of the non-uniform complexity class $\mathcal{P}/\mathrm{poly}$ implies an unexpected behavior of in the world of uniform complexity (i.e., the ability to reduce any constant number of quantifier alternations into two).

Teaching note: We present the proof in terms of machines that take advice, although a presentation in terms of circuits may be somewhat less cumbersome (i.e., may use less symbols). We note that a presentation in terms of circuits tends to confuse some students, becuase these circuits appear both as strings (describing these circuits) and as computing devices (applied to other strings).

Proof: Starting with an arbitrary set $S \in \Pi_2$, we shall show that $S \in \Sigma_2$. By Proposition 3.9, there exists a polynomial p and a set $S' \in \mathcal{NP}$ such that $S = \{x : \forall y \in \{0,1\}^{p(|x|)}\ (x,y) \in S'\}$. Using the theorem's hypothesis, it follows

that $S' \in \mathcal{P}/\mathrm{poly}$, which means that there exists polynomially bounded advice that allow for deciding S' in polynomial-time. That is, there exists a polynomial-time algorithm A and an advice sequence $(a_n)_{n\in\mathbb{N}}$ such that $|a_n| = \mathrm{poly}(n)$ and $A(a_n, \cdot)$ correctly decides the membership of n-bit long strings in S'. Thus, $x \in S$ if and only if there exists a "correct advice" $a \in \{0,1\}^{\mathrm{poly}(|x|)}$ for $(|x| + p(|x|))$-bit long inputs such that for every $y \in \{0,1\}^{p(|x|)}$ it holds that $A(a,(x,y)) = 1$, where the correctness condition means that $A(a,\cdot)$ decides correctly the membership of $(|x| + p(|x|))$-bit long strings in S'. Note that the aforementioned conditions have the right form as they assert that *there exists a string a such that for all y and z some condition holds* (i.e., $A(a,(x,y)) = 1$ and $A(a,z) = \chi_{S'}(z)$, where $\chi_{S'}(z) = 1$ if $z \in S'$ and $\chi_{S'}(z) = 0$ otherwise).

The problem is that part of the latter condition (i.e., $A(a,z) = \chi_{S'}(z)$) is not checkable in polynomial-time. Specifically, verifying $A(a,z) = \chi_{S'}(z)$ requires the ability to decide membership in S', which is an arbitrary set in $\mathcal{NP}$. Still, we shall see that in some cases the condition $\forall z A(a,z) = \chi_{S'}(z)$ can be replaced by a condition of the form $\forall z B(a,z)$, where B is a polynomial-time computable predicate.

Suppose, for a moment, that S' is downwards self-reducible; that is, that deciding whether $z \in S'$ can be reduced to deciding membership in S' of shorter (than z) strings. Then, referring to the aforementioned algorithm A and polynomial p, we revise (or rather augment) the foregoing suggestion and assert that $x \in S$ if and only if there exists a sequence of $\mathrm{poly}(|x|)$-bit long strings $a_1, ..., a_m$, where $m = (|x| + p(|x|))$, such that the following two conditions hold

1. For every $y \in \{0,1\}^{p(|x|)}$, it holds that $A(a_m,(x,y)) = 1$.

2. For $i = 1, ..., m$, for every $z \in \{0,1\}^i$ it holds that $A(a_i,z) = 1$ if and only if $z \in S'$. The point is that this condition can be expressed by a condition of the form for every $i = 1, ..., m$ and $z \in \{0,1\}^i$ it holds that $B(a_1, ..., a_i, z) = 1$, where B is a polynomial-time computable predicate.

   Specifically, using downwards self-reducible, we may obtain the value of $\chi_{S'}(z)$ from the values of $A(a_j, z')$ for some $j < i$ and $z' \in \{0,1\}^j$. Thus, we check the value of $A(a_i,z)$ against the value obtained by reducing the question of whether $z \in S'$ to the membership of shorter strings in S', where the latter decisions are made by relying on the hypothesis that we have correct advice for shorter strings. (Further details are given later.)

   In summary, we verify the correctness of the advice a_i, by relying on the correctness of the advice $a_1, ..., a_{i-1}$. (Needless to say, the correctness of the advice a_1 can be verified in a constant number of steps by using a brute force algorithm for deciding membership in S'.)

There is a minor problem with this revised suggestion: we have assumed that S' is downwards self-reducible, while S' may be an arbitrary set in $\mathcal{NP}$. The solution is to reduce S' to SAT, and rely on the fact that SAT is downwards self-reducible. The latter fact is implicit in the proof of Proposition 2.14; that is, $\phi \in$ SAT if and only if either $\phi_{x_1=0} \in$ SAT or $\phi_{x_1=1} \in$ SAT, where $\phi_{x_1=\sigma}$ denotes the formula obtained

from ϕ by setting the first variable (denoted x_1) to σ and simplifying the formula in the obvious way (thus reducing its length).

For simplicity, in the rest of the proof, we rely on the fact that S' is Karp-reducible to SAT (rather than only Cook-reducible to it). Specifically, let f be a Karp-reduction of S' to SAT. Thus, $x \in S$ if and only if $\forall y \in \{0,1\}^{p(|x|)}$ $\phi_{x,y} \in$ SAT, where $\phi_{x,y} \stackrel{\text{def}}{=} f(x,y)$. Using the hypothesis, we have SAT $\in \mathcal{P}/\text{poly}$, and thus there exists a polynomial-time algorithm A that solves SAT using advice of polynomial length. Now, we assert that $x \in S$ if and only if there exists a sequence of poly($|x|$)-bit long strings $a_1, ..., a_m$, where $m = \text{poly}(|x|)$ satisfies $m \geq \max_{y \in \{0,1\}^{p(|x|)}} \{|f(x,y)|\}$, such that the following two conditions hold:

1. For every $y \in \{0,1\}^{p(|x|)}$, it holds that $A(a_{|f(x,y)|}, f(x,y)) = 1$.
2. For $i = 1, ..., m$, for every $\phi \in \{0,1\}^i$ it holds that $A(a_i, \phi) = 1$ if and only if $\phi \in$ SAT. This condition is checked as follows.
   (a) For every (non-empty) formula ϕ, we denote by ϕ_σ the formula resulting from ϕ by setting its first variable to σ and making the obvious simplifications (i.e., omitting constants from clauses and omitting empty clauses). Thus, $|\phi_\sigma| < |\phi|$.
   (b) For every $i > 1$ and $\phi \in \{0,1\}^i$, we check that $A(a_i, \phi) = 1$ if and only if either $A(a_{|\phi_0|}, \phi_0) = 1$ or $A(a_{|\phi_1|}, \phi_1) = 1$.
   For $\phi \in \{0,1\}^1$, we check that $A(a_1, \phi) = 1$ if and only if ϕ is satisfiable (which can be determined in constant time). (Indeed, for any threshold $t = O(\log |x|)$, we could have distinguished the cases $i > t$ and $i \leq t$, where the latter can be handled in time poly(2^t).)

   To summarize, we determine whether or not ϕ is in SAT by computing $A(a_{|\phi_0|}, \phi_0)$ and $A(a_{|\phi_1|}, \phi_1)$. Thus, for $i > 1$, the condition checked is

$$\forall \phi \in \{0,1\}^i \; A(a_i, \phi) = \vee_{\sigma \in \{0,1\}} A(a_{|\phi_\sigma|}, \phi_\sigma).$$

Note that the correctness of the various advice follows by induction. That is, if the advice $a_1, ..., a_{i-1}$ are correct (for input lengths 1 through $i-1$) and, for $\phi^{(i)} \in \{0,1\}^i$, it holds that $A(a_i, \phi^{(i)}) = \vee_\sigma A(a_{|\phi_\sigma^{(i)}|}, \phi_\sigma^{(i)})$ then the advice a_i is correct (for input length i).

Finally, observe that the expression obtained for membership in S (i.e., $x \in S$ if and only if $\exists a_1, ..., a_m \forall y, \phi^{(1)}, ..., \phi^{(m)}$ it holds that $A(a_{|f(x,y)|}, f(x,y)) = 1$ and for $i = 2, ..., m$ $A(a_i, \phi^{(i)}) = \vee_\sigma A(a_{|\phi_\sigma^{(i)}|}, \phi_\sigma^{(i)})$ (and $A(a_1, \phi^{(1)})$ is correct)) is indeed of the Σ_2-form. Thus, $S \in \Sigma_2$ and it follows that $\Pi_2 \subseteq \Sigma_2$. Using Exercise 3.7, the theorem follows. ■

A technical perspective. The proof of Theorem 3.12 implies that *any set $S' \in \mathcal{P}/\text{poly}$ that is downwards self-reducible is in Σ_2*. The point is that the existential quantifier selects advice that are checked for correctness using the universal quantifier, while relying on the downwards self-reducibility process.

Chapter Notes

The class $\mathcal{P}$/poly was defined by Karp and Lipton [128] as part of a general formulation of "machines which take advise" [128]. They also noted the equivalence to the traditional formulation of polynomial-size circuits as well as the effect of uniformity (Proposition 3.4).

The Polynomial-Time Hierarchy ($\mathcal{PH}$) was introduced by Stockmeyer [199]. A third equivalent formulation of $\mathcal{PH}$ (via "alternating machines") can be found in [49].

The implication of the failure of the conjecture that $\mathcal{NP}$ is not contained in $\mathcal{P}$/poly on the Polynomial-time Hierarchy (i.e., Theorem 3.12) was discovered by Karp and Lipton [128]. This interesting connection between non-uniform and uniform complexity provides the main motivation for presenting $\mathcal{P}$/poly and $\mathcal{PH}$ in the same chapter. We call the reader's attention to the proof of Theorem 3.12, and specifically to the *inspiring use of downwards self-reducibility*.

Exercises

Exercise 3.1 (sparse sets) A set $S \subset \{0,1\}^*$ is called sparse if there exists a polynomial p such that $|S \cap \{0,1\}^n| \leq p(n)$ for every n.

1. Prove that any sparse set is in $\mathcal{P}$/poly. Note that a sparse set may be undecidable.
2. Prove that a set is in $\mathcal{P}$/poly if and only if it is Cook-reducible to some sparse set.

Guideline: For the forward direction of Part 2, encode the advice sequence $(a_n)_{n\in\mathbb{N}}$ as a sparse set $\{(1^n, i, \sigma_{n,i}) : n \in \mathbb{N},\ i \leq |a_n|\}$, where $\sigma_{n,i}$ is the i^{th} bit of a_n. For the opposite direction, note that on input x the Cook-reduction makes queries of length at most poly($|x|$), and so all the relevant strings in the target of the reduction can be encoded in the n^{th} advice.

Exercise 3.2 (advise hierarchy) Prove that for any two functions $\ell, \delta : \mathbb{N} \to \mathbb{N}$ such that $\ell(n) < 2^{n-1}$ and δ is unbounded, it holds that $\mathcal{P}/\ell$ is strictly contained in $\mathcal{P}/(\ell + \delta)$.
(Hint: for every sequence $\overline{a} = (a_n)_{n\in\mathbb{N}}$ such that $|a_n| = \ell(n) + \delta(n)$, consider the set $S_{\overline{a}}$ where $x \in S$ if and only if x is the i^{th} string of length $|x|$ and $i \leq |a_{|x|}|$ and the i^{th} bit in $a_{|x|}$ is 1. For more details see Section 4.1.)

Exercise 3.3 Prove that Σ_2 contains all sets that are Cook-reducible to $\mathcal{NP}$.
(Hint: This is quite obvious when using the definition of Σ_2 as presented in Section 3.2.2; see Exercise 3.7. Alternatively, the fact can be proved by using some of the ideas of the proof of Theorem 2.33.)

Exercise 3.4 (the class Π_i) Prove that for any natural number k, a decision problem $S \subseteq \{0,1\}^*$ is in Π_k if there exists a polynomial p and a polynomial time

algorithm V such that $x \in S$ if and only if

$$\forall y_1 \in \{0,1\}^{p(|x|)} \exists y_2 \in \{0,1\}^{p(|x|)} \forall y_3 \in \{0,1\}^{p(|x|)} \cdots Q_k y_k \in \{0,1\}^{p(|x|)} \text{ s.t. } V(x, y_1, ..., y_k) = 1$$

where Q_k is a universal quantifier if k is odd and is an existential quantifier otherwise.
(Recall that Π_k was defined as $\mathrm{co}\Sigma_k$, which in turn is defined as $\{\{0,1\}^* \setminus S : S \in \Sigma_k\}$.)

Exercise 3.5 (complete problems for $\mathcal{PH}$) A k-alternating quantified Boolean formula is a quantified Boolean formula with up to k alternations between existential and universal quantifiers, starting with an existential quantifier. For example, $\exists z_1 \exists z_2 \forall z_3 \phi(z_1, z_2, z_3)$ (where the z_i's are Boolean variables) is a 2-alternating quantified Boolean formula. Prove that the problem of deciding whether or not a k-alternating quantified Boolean formula is valid is Σ_k-complete. That is, denoting the aforementioned problem by `kQBF`, prove that `kQBF` is in Σ_k and every problem in Σ_k is Karp-reducible to `kQBF`.

Exercise 3.6 (on the relation between $\mathcal{PH}$ and $\mathcal{AC}^0$) Note that there is an obvious analogy between $\mathcal{PH}$ and constant depth circuits of unbounded fan-in, where existential (resp., universal) quantifiers are represented by "large" $\bigvee$ (rep., $\bigwedge$) gates. To articulate this relationship, consider the following definitions.

- A family of circuits $\{C_N\}$ is called highly uniform if there exists a polynomial-time algorithm that answers local queries regarding the structure of the relevant circuit. Specifically, on input (N, u, v), the algorithm determines the type of gates represented by the vertices u and v in C_N as well as whether there exists a directed edge from u to v. Note that this algorithm operates in time that polylogarithmic in the size of C_N.

  We focus on family of polynomial-size circuits, meaning that the size of C_N is polynomial in N, which in turn represents the number of inputs to C_N.

- Fixing a polynomial p, a *p-succinctly represented input* $Z \in \{0,1\}^N$ is a circuit c_Z of size at most $p(\log_2 N)$ such that for every $i \in [N]$ it holds that $c_Z(i)$ equals the i^{th} bit of Z.

- For a fixed family of highly uniform circuits $\{C_N\}$ and a fixed polynomial p, the problem of evaluating a succinctly represented input is defined as follows. Given p-succinct representation of an input $Z \in \{0,1\}^N$, determine whether or not $C_N(Z) = 1$.

For every k and every $S \in \Sigma_k$, show that there exists a family of highly uniform unbounded fan-in circuits of depth k and polynomial-size such that S is Karp-reducible to evaluating a succinctly represented input (with respect to that family of circuits). That is, the reduction should map an instance $x \in \{0,1\}^n$ to a p-succinct representation of some $Z = Z_x \in \{0,1\}^N$ such that $x \in S$ if and only if

$C_N(Z) = 1$. (Note that Z is represented by a circuit c_Z of size at most $p(\log_2 N)$, and that it must hold that $|c_Z| \leq \text{poly}(n)$ and thus $N \leq \exp(\text{poly}(n))$.)[6]

Guideline: Let $S \in \Sigma_k$ and let V be the corresponding verification algorithm as in Definition 3.8. That is, $x \in S$ if and only if $\exists y_1 \forall y_2 \cdots Q_k y_k$, where each $y_i \in \{0,1\}^{\text{poly}(|x|)}$ such that $V(x, y_1, ..., y_k) = 1$. Then, for $m = \text{poly}(|x|)$ and $N = 2^{k \cdot m}$, consider the circuit $C_N(Z) = \bigvee_{i_1 \in [2^m]} \bigwedge_{i_2 \in [2^m]} \cdots Q'_{i_k \in [2^m]} Z_{i_1,i_2,...,i_k}$, and the problem of evaluating it at the input consisting of the truth-table of $V(x, \cdots)$ (i.e., when setting $Z_{i_1,i_2,...,i_k} = V(x, i_1, ..., i_k)$). Note that the size of C_N is $O(N)$.[7]

Exercise 3.7 Verify the following facts:

1. For every $k \geq 1$, it holds that $\Sigma_k \subseteq \mathcal{P}^{\Sigma_k} \subseteq \Sigma_{k+1}$.

   (Note that, for any complexity class $\mathcal{C}$, the class $\mathcal{P}^{\mathcal{C}}$ is the class of sets that are Cook-reducible to some set in $\mathcal{C}$. In particular, $\mathcal{P}^{\mathcal{P}} = \mathcal{P}$.)
2. For every $k \geq 1$, $\Pi_k \subseteq \mathcal{P}^{\Pi_k} \subseteq \Pi_{k+1}$.

   (Hint: For any complexity class $\mathcal{C}$, it holds that $\mathcal{P}^{\mathcal{C}} = \mathcal{P}^{\text{co}\mathcal{C}}$ and $\mathcal{P}^{\mathcal{C}} = \text{co}\mathcal{P}^{\mathcal{C}}$.)
3. For every $k \geq 1$, it holds that $\Sigma_k \subseteq \Pi_{k+1}$ and $\Pi_k \subseteq \Sigma_{k+1}$. Thus, $\mathcal{PH} = \cup_k \Pi_k$.
4. For every $k \geq 1$, if $\Sigma_k \subseteq \Pi_k$ (resp., $\Pi_k \subseteq \Sigma_k$) then $\Sigma_k = \Pi_k$.

   (Hint: For any $S \in \Pi_k$ (resp., $S \in \Sigma_k$), apply the hypothesis to $\{0,1\}^* \setminus S$.)

Exercise 3.8 In continuation to Exercise 3.5, prove that following claims:

1. SAT is computationally equivalent to 1QBF.
2. For every $k \geq 1$, it holds that $\mathcal{P}^{\Sigma_k} = \mathcal{P}^{\text{kQBF}}$ and $\Sigma_{k+1} = \mathcal{NP}^{\text{kQBF}}$.

   Guideline: Prove that if S is $\mathcal{C}$-complete then $\mathcal{P}^{\mathcal{C}} = \mathcal{P}^S$. Note that $\mathcal{P}^{\mathcal{C}} \subseteq \mathcal{P}^S$ uses the polynomial-time reductions of $\mathcal{C}$ to S, whereas $\mathcal{P}^S \subseteq \mathcal{P}^{\mathcal{C}}$ uses $S \in \mathcal{C}$.

Exercise 3.9 Referring to the notion of downwards self-reducibility (as defined in the proof of Theorem 3.12), prove that if $\mathcal{P} = \mathcal{NP}$ (resp., $\mathcal{NP} = \text{co}\mathcal{NP}$) then any set $S' \in \mathcal{P}/\text{poly}$ that is downwards self-reducible is in $\mathcal{P}$ (resp., in $\mathcal{NP}$).
(Hint: any set $S' \in \mathcal{P}/\text{poly}$ that is downwards self-reducible is in Σ_2.)

Exercise 3.10 In continuation to Part 2 of Exercise 3.1, we consider the class of sets that are Karp-reducible to a sparse set. It can be proved that this class contains SAT if and only if $\mathcal{P} = \mathcal{NP}$ (see [77]). Here, we only consider the special case in which the sparse set is contained in another sparse set that is in $\mathcal{P}$ (e.g., the latter set may be $\{1\}^*$, in which case the former set may be an arbitrary unary set). Specifically, we claim that

[6] Assuming $\mathcal{P} \neq \mathcal{NP}$, it cannot be that $N \leq \text{poly}(n)$ (because circuit evaluation can be performed in time polynomial in the size of the circuit).

[7] Advanced comment: the limitations of $\mathcal{AC}^0$ circuits imply limitations on the functions of the truth-table of a *generic* V that such the aforementioned circuits C_N can compute. Unfortunately, these limitations do not seem to provide useful information on the limitations of functions that are confined to truth-tables that have succinct representation (as in the case of the actual V). This fundamental problem is "resolved" in the context of "relativization" by providing V with oracle access to an arbitrary input of length N (or so); cf. [79].

if SAT *is Karp-reducible to a set* $S \subseteq G$ *such that* $G \in \mathcal{P}$ *and* G *is sparse then* $\mathtt{SAT} \in \mathcal{P}$.

Using the hypothesis, we outline a polynomial-time procedure for solving the search problem of SAT, and leave the task of providing the details as an exercise. The procedure conducts a DFS on the tree of all possible partial truth assignment to the input formula, while truncating the search at nodes that are roots of sub-trees that were already proved to contain no satisfying assignment (at the leaves).[8]

Guideline: The key observation is that each internal node (which yields a formula derived from the initial formulae by instantiating the corresponding partial truth assignment) is mapped by the reduction either to a string not in G (in which case we conclude that the sub-tree contains no satisfying assignments and backtrack from this node) or to a string in G. In the latter case, unless we already know that this string is not in S, we start a scan of the sub-tree rooted at this node. but once we backtrack from this internal node, we know that the corresponding element of G is not in S, and we will never extend a node mapped to this element again. Also note that once we reach a leaf, we can check by ourselves whether or not it corresponds to a satisfying assignment to the initial formula. (Hint: When analyzing the forgoing procedure, note that on input an n-variable formulae ϕ the number of times we start to scan a sub-tree is at most $n \cdot |\cup_{i=1}^{\mathrm{poly}(|\phi|)} \{0,1\}^i \cap (G \setminus S)|$.)

[8] For an n-variable formulae, the leaves of the tree correspond to all possible n-bit long strings, and an internal node corresponding to τ is the parent of nodes corresponding to $\tau 0$ and $\tau 1$.

Chapter 4

More Resources, More Power?

More electricity, less toil.

The Israeli Electricity Company, 1960s

Is it indeed the case that the more resources one has, the more one can achieve? The answer may seem obvious, but the obvious answer (of yes) actually presumes that the worker knows how much resources are at his/her disposal. In this case, when allocated more resources, the worker (or computation) can indeed achieve more. But otherwise, nothing may be gained by adding resources.

In the context of computational complexity, an algorithm knows the amount of resources that it is allocated if it can determine this amount without exceeding the corresponding resources. This condition is satisfies in all natural cases, but it may not hold in general. The latter fact should not be that surprising: we already know that some functions are not computable and if these functions are used to determine resources then the algorithm may be in trouble. Needless to say, this discussion requires some formalization, which is provided in the current chapter.

Summary: When using "nice" functions to determine the algorithm's resources, it is indeed the case that more resources allow for more tasks to be performed. However, when "ugly" functions are used for the same purpose, increasing the resources may have no effect. By nice functions we mean functions that can be computed without exceeding the amount of resources that they specify (e.g., $t(n) = n^2$ or $t(n) = 2^n$). Naturally, "ugly" functions do not allow to present themselves in such nice forms.

The forgoing discussion refers to a uniform model of computation and to (natural) resources such as time and space complexities. Thus, we get results asserting, for example, that there are functions computable in cubic-time but not in quadratic-time. In case of non-uniform models

of computation, the issue of "nicety" does not arise, and it is easy to establish separations between levels of circuit complexity that differ by any unbounded amount.

Results that *separate* the class of problems solvable within one resource bound from the class of problems solvable within a larger resource bound are called hierarchy theorems. Results that indicate the non-existence of such separations, hence indicating a "gap" in the growth of computing power (or a "gap" in the existence of algorithms that utilize the added resources), are called gap theorems. A somewhat related phenomenon, called speed-up theorems, refers to the inability to define the complexity of some problems.

Caveat: Uniform complexity classes based on specific resource bounds (e.g., cubic-time) are model dependent. Furthermore, the tightness of separation results (i.e., how much more time is required to solve an additional computational problem) is also model dependent. Still the existence of such separations is a phenomenon common to all reasonable and general models of computation (as referred to in the Cobham-Edmonds Thesis). In the following presentation, we will explicitly differentiate model-specific effects from generic ones.

Organization: We will first demonstrate the "more resources yield more power" phenomenon in the context of non-uniform complexity. In this case the issue of "knowing" the amount of resources allocated to the computing device does not arise, because each device is tailored to the amount of resources allowed for the input length that it handles (see Section 4.1).

4.1 Non-uniform complexity hierarchies

The model of machines that use advice (cf. §1.2.4.2 and Section 3.1.2) offers a very convenient setting for separation results. We refer specifically, to classes of the form $\mathcal{P}/\ell$, where $\ell : \mathbb{N} \to \mathbb{N}$ is an arbitrary function (see Definition 3.5). Recall that every Boolean function is in $\mathcal{P}/2^n$, by virtue of a trivial algorithm that is given as advice the truth-table of the function restricted to the relevant input length. An analogous algorithm underlies the following separation result.

Theorem 4.1 *For any two functions $\ell', \delta : \mathbb{N} \to \mathbb{N}$ such that $\ell'(n) + \delta(n) \leq 2^n$ and δ is unbounded, it holds that $\mathcal{P}/\ell'$ is strictly contained in $\mathcal{P}/(\ell' + \delta)$.*

Proof: Let $\ell \stackrel{\text{def}}{=} \ell' + \delta$, and consider the algorithm A that given advice $a_n \in \{0,1\}^{\ell(n)}$ and input $i \in \{1, ..., 2^n\}$ (viewed as an n-bit long string), outputs the i^{th} bit of a_n if $i \leq |a_n|$ and zero otherwise. Clearly, for any $\overline{a} = (a_n)_{n \in \mathbb{N}}$ such that $|a_n| = \ell(n)$, it holds that the function $f_{\overline{a}}(x) \stackrel{\text{def}}{=} A(a_{|x|}, x)$ is in $\mathcal{P}/\ell$. Furthermore, different sequences $\overline{a}$ yield different functions $f_{\overline{a}}$. We claim that some of these functions $f_{\overline{a}}$ are not in $\mathcal{P}/\ell$, thus obtaining a separation.

The claim is proved by considering all possible (polynomial-time) algorithms A' and all possible sequences $\overline{a}' = (a'_n)_{n\in\mathbb{N}}$ such that $|a'_n| = \ell'(n)$. Fixing any algorithm A', we consider the number of n-bit long functions that are correctly computed by $A'(a'_n, \cdot)$. Clearly, the number of these functions is at most $2^{\ell'(n)}$, and thus A' may account for at most $2^{-\delta(n)}$ fraction of the functions $f_{\overline{a}}$ (even when restricted to n-bit strings). This consideration holds for every n and every possible A', and thus the measure of the set of functions that are computable by algorithms that take advice of length ℓ is zero.[1] ■

A somewhat less tight bound can be obtained by using the model of Boolean circuits. In this case some slackness is needed in order to account for the gap between the upper and lower bounds regarding the number of Boolean functions over $\{0,1\}^n$ that are computed by Boolean circuits of size $s < 2^n$. Specifically (see Exercise 4.1), an obvious lower-bound on this number is $2^{s/O(\log s)}$ whereas an obvious upper-bound is $\binom{s^2}{s} \approx 2^{2s\log_2 s}$. (Compare these to the lower-bound 2^s, and the upper-bound $2^{s+((\delta(n)-2)/2)}$ used in the proof of Theorem 4.1.)

4.2 Time Hierarchies and Gaps

In this section we show that in the "natural cases" increasing time-complexity allows for more problems to be solved, whereas in "pathological cases" it may happen that even a dramatic increase in the time-complexity provides no additional computing power. As hinted in the introductory comments to the current chapter, the "natural cases" correspond to time bounds that can be determined by the algorithm itself within the specified time complexity.

4.2.1 Time Hierarchies

Note that the non-uniform computing devices considered in the previous section were explicitly given the relevant resource bounds. Actually, they were given the resources themselves and did not need to monitor their usage of these resources. In contrast, when designing algorithms of time complexity $t : \mathbb{N} \to \mathbb{N}$, we need to make sure that the algorithm does not exceed the time bound. Furthermore, when invoked on input x, the algorithm is not given the time bound $t(|x|)$ explicitly, and a reasonable design methodology (for dealing with generic algorithms as we will in the proof of the following separation results) is to let the algorithm compute this bound (i.e., $t(|x|)$). This, in turn, requires to read the entire input (see Exercise 4.2) as well as to compute $t(n)$ using $O(t(n))$ (or so) time. The latter requirement motivates the following definition (which is related to the standard definition of "fully time constructibility" (cf. [116, Sec. 12.3])).

Definition 4.2 (time constructible functions): *A function $t : \mathbb{N} \to \mathbb{N}$ is called* time constructible *if there exists an algorithm that on input n outputs $t(n)$ using at most $t(n)$ steps.*

[1] It suffices to show that this measure is strictly less than one. This is easily done by considering, for every algorithm A', the performance of A' on inputs of length n such that $\delta(n) > 2|\langle A'\rangle| + 2$.

Equivalently, we may require that the mapping $1^n \mapsto t(n)$ be computable within time complexity t. We warn that the foregoing definition is model dependent; however, typically nice functions are computable even faster (e.g., in $\text{poly}(\log t(n))$ steps), in which case the model-dependency is irrelevant (for reasonable and general models of computation, as referred to in the Cobham-Edmonds Thesis). For example, in any reasonable and general model, functions like $t_1(n) = n^2$, $t_2(n) = 2^n$, and $t_3(n) = 2^{2^n}$ are computable in $\text{poly}(\log t_i(n))$ steps.

Likewise, for a fixed model of computation (to be understood from the context) and for any function $t : \mathbb{N} \to \mathbb{N}$, we *denote by* $\text{DTIME}(t)$ *the class of decision problems that are solvable in time complexity* t.

4.2.1.1 The Time Hierarchy Theorem

In the following theorem, we refer to the model of two-tape Turing machines. In this case we obtain quite a tight hierarchy, when referring to the relation between t_1 and t_2. We stress that using the Cobham-Edmonds Thesis, this results yields (possibly less tight) hierarchy theorems for any reasonable and general model of computation.

Teaching note: The standard statement of the following result asserts that *for any time computable function t_2 and every function t_1 such that $t_2 = \omega(t_1 \log t_1)$ and $t_1(n) > n$ it holds that* $\text{DTIME}(t_1)$ *is strictly contained in* $\text{DTIME}(t_2)$. We find the proof of the current version more intuitive, and comment on the proof of the standard version after proving the current one.

Theorem 4.3 (time hierarchy for two-tape Turing machines): *For any time computable function t_1 and every function t_2 such that $t_2(n) \geq (\log t_1(n))^2 \cdot t_1(n)$ and $t_1(n) > n$ it holds that* $\text{DTIME}(t_1)$ *is strictly contained in* $\text{DTIME}(t_2)$.

As will become clear from the proof, an analogous result holds for any model in which a universal machine can emulate t steps of another machine in $O(t \log t)$ time. Before proving Theorem 4.3, we derive the following corollary.

Corollary 4.4 (time hierarchy for any reasonable and general model): *For any reasonable and general model of computation there exists a positive polynomial p such that for any time-computable function t_1 and every function t_2 such that $t_2 > p(t_1)$ and $t_1(n) > n$ it holds that* $\text{DTIME}(t_1)$ *is strictly contained in* $\text{DTIME}(t_2)$.

Proof of Corollary 4.4: Letting DTIME_2 denote the classes that correspond to two-tape Turing machines, we have $\text{DTIME}(t_1) \subseteq \text{DTIME}_2(t_1')$ and $\text{DTIME}(t_2) \supseteq \text{DTIME}_2(t_2')$, where $t_1' = \text{poly}(t_1)$ and t_2' is defined such that $t_2(n) = \text{poly}(t_2'(n))$. The latter unspecified polynomials, hereafter denoted p_1 and p_2 respectively, are the ones guaranteed by the Cobham-Edmonds Thesis. Also, the hypothesis that t_1 is time-computable implies that $t_1' = p_1(t_1)$ is time-constructible with respect to the two-tape Turing machine model. Thus, for a suitable choice of the polynomial p, it holds that

$$t_2'(n) = p_2^{-1}(t_2(n)) > p_2^{-1}(p(t_1(n))) > p_2^{-1}(p(p_1^{-1}(t_1'(n)))) > (t_1'(n))^2 .$$

Invoking Theorem 4.3, we have $\text{DTIME}_2(t_2') \supset \text{DTIME}_2(t_1')$, and the corollary follows. ■

Proof of Theorem 4.3: The idea is to construct a Boolean function f such that all machines having time complexity t_1 fail to compute f. This is done by associating each possible machine M a different input x_M (e.g., $x_M = \langle M \rangle$), and making sure that $f(x_M) \neq M'(x)$, where $M'(x)$ denotes an emulation of $M(x)$ that is suspended after $t_1(|x|)$ steps. Actually, we are going to use a mapping μ of inputs to machines (i.e., $\mu(x_M) = M$), such that each machine is in the range of μ and μ is very easy to compute.

The issue is presenting an algorithm for computing f. This algorithm is straightforward: On input x, it computes $t = t_1(|x|)$, determines the machine $M = \mu(x)$ that corresponds to x (outputting a default value of no such machine exists), emulates $M(x)$ for t steps, and returns the value $1 - M'(x)$. The question is how much time is required for this emulation. We should bear in mind that the time complexity of our algorithm needs to be evaluated in the two-tape Turing machine model, whereas M itself is a two-tape Turing machine. The obvious approach is to implement our algorithm on a three-tape Turing-machine, using two tapes for the emulation itself and another tape for the emulation procedure. Using this approach, each step of M is emulated in $O(|\langle M \rangle|)$ steps (on the three-tape machine). Emulating t' steps of a three-tape machine on a two-tape machine requires $O(t' \log t')$ steps, and so the emulation costs $O(T_M(|x|) \log T_M(|x|))$, where $T_M(n) = O(|\langle M \rangle| \cdot t_1(n))$.

It turns out that the quality of the result we obtain depends on the mapping μ of inputs to machines. Using the naive (identity) mapping of the input x_M to the machine M (such that $x_M = \langle M \rangle$ is a description of M), we can only establish the theorem for $t_2(n) = \omega(n \cdot t_1(n) \log(n \cdot t_1(n)))$. The theorem follows by associating with machine M the input $x_M = 0^m 1 \langle M \rangle$, where $m = 2^{|\langle M \rangle|}$, because in this case $|\langle M \rangle| < \log_2 |x_M| = O(\log t_1(|x_M|))$. In other words, we may use the mapping μ such that $\mu(x) = M$ if $x = 0^{2^{|\langle M \rangle|}} 1 \langle M \rangle$ and $\mu(x)$ equals some fixed machine otherwise. ■

Teaching note: Proving the standard version of Theorem 4.3 cannot be done by associating a sufficiently long input x_M with each machine M, because this does not allow to get rid from an additional unbounded factor multiplier of $t_1(n) \log t_1(n)$. Note that the latter factor needs to be computable (at the very least) and thus cannot be accounted for by the generic ω-notation that appears in the standard version (cf. [116, Thm. 12.9]). Instead, a different approach is taken (see Footnote 2).

[2] The function f is not defined with reference to $t_1(|x_M|)$ steps of $M(x_M)$, but rather with reference to the result of emulating $M(x_M)$ while using a total of $t_2(|x_M|)$ steps in the emulation process (i.e., in the algorithm used to compute f). This guarantees that f is in $\text{DTIME}(t_2)$, and pushes the problem to showing that f is not in $\text{DTIME}(t_1)$, where the problem is resolved by noting that each candidate machine having time complexity t_1 will have its executions fully emulated on any sufficiently long input. Thus, we merely need to associate with each M a disjoint set of infinitely many inputs and make sure that M errs on each of these inputs.

Comments. The proof of Theorem 4.3 associates with each potential machine an input and makes this machine err on this input. The aforementioned association is rather flexible: it should merely be efficiently computed (in the direction from the input to a possible machine) and should be sufficiently shrinking (in that direction). Specifically, we used the mapping μ such that $\mu(x) = M$ if $x = 0^{2^{|\langle M\rangle|}}1\langle M\rangle$ and $\mu(x)$ equals some fixed machine otherwise. We comment that each machine can be made to err on infinitely many inputs by redefining μ such that $\mu(x) = M$ if $0^{2^{|\langle M\rangle|}}1\langle M\rangle$ is a prefix of x (and $\mu(x)$ equals some fixed machine otherwise). We also comment that, in contrast to the proof of Theorem 4.3, the proof of Theorem 1.5 utilizes a rigid mapping of inputs to machines (i.e., there $\mu(x) = M$ if $x = \langle M\rangle$).

4.2.1.2 Impossibility of speed-up for universal computation

The Time Hierarchy Theorem (Theorem 4.3) implies that the computation of a universal machine cannot be significantly sped-up. That is, consider the function $\mathtt{u}'(\langle M\rangle, x, t) \stackrel{\text{def}}{=} y$ if on input x machine M halts within t steps and outputs the string y, and $\mathtt{u}'(\langle M\rangle, x, t) \stackrel{\text{def}}{=} \perp$ if on input x machine M makes more than t steps. Recall that the value of $\mathtt{u}'(\langle M\rangle, x, t)$ can be computed in $\widetilde{O}(|x| + |\langle M\rangle| \cdot t)$ steps. Theorem 4.3 implies that this value cannot be computed with significantly less steps.

Theorem 4.5 *There exists no two-tape Turing machine that, on input $\langle M\rangle, x$ and t, computes $\mathtt{u}'(\langle M\rangle, x, t)$ in $o((t + |x|) \cdot f(M)/\log^2(t + |x|))$ steps, where f is an arbitrary function.*

A similar result holds for any reasonable and general model of computation (cf., Corollary 4.4).

Proof: Suppose (towards the contradiction) that, for every M, given x and $t > |x|$, the value of $\mathtt{u}'(\langle M\rangle, x, t)$ can be computed in $o(t/\log^2 t)$ steps. Consider an arbitrary time computable t_1 (s.t. $t_i 1(n) > n$) and an arbitrary set $S \in \text{DTIME}(t_2)$, where $t_2(n) = t_1(n) \cdot \log^2 t_1(n)$. Let M be a machine of time complexity t_2 that decides membership in S, and consider an algorithm that, on input x, first computes $t = t_1(|x|)$, and then computes (and outputs) the value $\mathtt{u}'(\langle M\rangle, x, t\log^2 t)$. By the time constructibility of t_1, the first computation can be implemented in t steps, and by the contradiction hypothesis the same holds for the second computation. Thus, S can be decided in $\text{DTIME}(t_1)$, implying that $\text{DTIME}(t_2) = \text{DTIME}(t_1)$, which in turn contradicts Theorem 4.3. ■

4.2.1.3 Hierarchy theorem for non-deterministic time

Analogously to DTIME, for a fixed model of computation (to be understood from the context) and for any function $t : \mathbb{N} \to \mathbb{N}$, we *denote by* $\text{NTIME}(t)$ *the class of sets that are accepted by some non-deterministic machine of time complexity t.* Alternatively, analogously to the definition of $\mathcal{NP}$, a set $S \subseteq \{0,1\}^*$ is in $\text{NTIME}(t)$ if there exists a *linear-time* algorithm V such that the two conditions hold

1. For every $x \in S$ there exists $y \in \{0,1\}^{t(|x|)}$ such that $V(x,y) = 1$.

2. For every $x \notin S$ and every $y \in \{0,1\}^*$ it holds that $V(x,y) = 0$.

We warn that the two formulations are not identical, but in sufficiently strong models (e.g., two-tape Turing machines) they are related up to logarithmic factors (see Exercise 4.3). The hierarchy theorem itself is similar to the one for deterministic time, except that here we require that $t_2(n) \geq (\log t_1(n+1))^2 \cdot t_1(n+1)$ (rather than $t_2(n) \geq (\log t_1(n))^2 \cdot t_1(n)$). That is:

Theorem 4.6 (non-deterministic time hierarchy for two-tape Turing machines): *For any time computable function t_1 and every function t_2 such that $t_2(n) \geq (\log t_1(n+1))^2 \cdot t_1(n+1)$ and $t_1(n) > n$ it holds that* $\text{NTIME}(t_1)$ *is strictly contained in* $\text{NTIME}(t_2)$.

Proof Sketch: We cannot just apply the proof of Theorem 4.3, because the Boolean function f defined there requires the ability to determine whether M accepts the input x_M in $t_1(|x_M|)$ steps. In the current context, M is a non-deterministic machine and so the only way we know how to determine this questions (both for a yes and no answers) is to try all the $(2^{t_1(|x_M|)})$ relevant executions. But this would put f in $\text{DTIME}(2^{t_1})$, rather than in $\text{NTIME}(\widetilde{O}(t_1))$, and so a different approach is needed.

We associate with each machine M, a large interval of strings (viewed as integers), denoted $I_M = [\ell_M, u_M]$, such that the various intervals do not intersect and such that it is easy to determine for each string x in which interval it falls. For each $x \in [\ell_M, u_M - 1]$, we define $f(x) = 1$ if and only if there exists a non-deterministic computation of M that accepts the input $x' \stackrel{\text{def}}{=} x + 1$ in $t_1(|x'|)$ steps. As for $f(u_M)$, we define it as *zero* if and only if there exists a non-deterministic computation of M that accepts the input ℓ_M in $t_1(|\ell_M|)$ steps. This definition is coupled with a non-deterministic machine for accepting the set $\{x : f(x) = 1\}$. On input $x \in [\ell_M, u_M - 1]$, this non-deterministic machine emulates a non-deterministic computation of M, which can be done in time $(\log t_1(|x+1|))^2 \cdot t_1(|x+1|)$. On input $x = u_M$, the machine just tries all $2^{t_1(|\ell_M|)}$ ($t_1(|\ell_M|)$-step) executions of $M(\ell_M)$, which can be done in time $t_1(|u_M|) \cdot t_1(|u_M|)$ provided the interval was chosen to be large enough (i.e., $t_1(|u_M|) \geq 2^{t_1(|\ell_M|)}$, which certainly holds when $|u_M| \geq 2^{t_1(|\ell_M|)}$).

Defining f this way guarantees that it is not in $\text{NTIME}(t_1)$. Suppose on the contrary, that some non-deterministic machine M of time complexity t_1 accepts the set $\{x : f(x) = 1\}$. We define a Boolean function A_M such that $A_M(x) = 1$ if and only if there exists a non-deterministic computation of M that accepts the input x, and note that $A_M(x) = f(x)$. Focusing on the interval $[\ell_M, u_M]$, we have $A_M(x) = f(x)$ for every $x \in [\ell_M, u_M]$, which (combined with the definition of f) implies that $A_M(x) = f(x) = A_M(x+1)$ for every $x \in [\ell_M, u_M - 1]$ and $A_M(u_M) = f(u_M) = 1 - A_M(\ell_M)$. Thus, we reached a contraction (i.e., $A_M(\ell_M) = \cdots = A_M(u_M) = 1 - A_M(\ell_M)$). □

4.2.2 Time Gaps and Speed-Up

In contrast to Theorem 4.3, there exists functions $t : \mathbb{N} \to \mathbb{N}$ such that $\text{DTIME}(t) = \text{DTIME}(t^2)$ (or even $\text{DTIME}(t) = \text{DTIME}(2^t)$). Needless to say, these functions are not time-constructible (and thus the aforementioned fact does not contradict Theorem 4.3). The reason for this phenomenon is that, for such functions t, there exists not algorithms that have time complexity above t but below t^2 (resp., 2^t).

Theorem 4.7 (the time gap theorem): *For every non-decreasing computable function $g : \mathbb{N} \to \mathbb{N}$ there exists a non-decreasing computable function $t : \mathbb{N} \to \mathbb{N}$ such that $\text{DTIME}(t) = \text{DTIME}(g(t))$.*

The forgoing examples referred to $g(m) = m^2$ and $g(m) = 2^m$. Since we are mainly interested in dramatic gaps (i.e., super-polynomial functions g), the model of computation does not matter here (as long as it is reasonable and general).

Proof: Consider an enumeration of all possible algorithms (or machines) and let t_i denote the time complexity of the i^{th} algorithm. Recall that we cannot enumerate only machines that halt on every input, and that $t_i(n) = \infty$ if the i^{th} machine does not halt on some n-bit long input. The basic idea is to define t such that no t_i is "sandwiched" between t and $g(t)$, and thus no algorithm will have time complexity between t and $g(t)$.

Intuitively, if $t_i(n)$ is finite, then we may define t such that $t_i(n) < t(n)$, whereas if $t_i(n) = \infty$ then any finite value of $t(n)$ will do (because then $t_i(n) > g(t(n))$). The problem is that we want t to be computable, whereas we cannot tell whether or not $t_i(n)$ is finite. However, we do not really need to make the latter decision: for each candidate value v of $t(n)$, we should just determine whether or not $t_i(n) \in [v, g(v)]$, which can be decided by running the i^{th} machine for at most $g(v)$ steps (on each n-bit long string). Bearing in mind that we should deal with all possible machines, we obtain the following procedure for setting $t(n)$.

Let $\mu : \mathbb{N} \to \mathbb{N}$ be any unbounded and computable function (e.g., $\mu(n) = n$ will do). Starting with $v = n + 1$, we keep incrementing v until v satisfies, for every $i \in \{1, ..., \mu(n)\}$, either $t_i(n) < v$ or $t_i(n) > g(v)$. This condition can be verified by computing $\mu(n)$ and $g(v)$, and emulating the execution of of the $\mu(n)$ machines on each of the n-bit long strings for $g(v)$ steps. The procedure sets $g(n)$ to equal the first value v satisfying the aforementioned condition, and halts. To show that the procedure halts on every n, consider the set $H_n \subseteq \{1, ..., \mu(n)\}$ of indices of the relevant machines that halt on all inputs of length n. Then the procedure definitely halts before reaching the value $v = m + 2$, where $m = \max_{i \in H_n}\{t_i(n)\}$. (Indeed, the procedure may halt with $v \leq m$, provide that $g(v) < m$.)

For this function t, we claim that $\text{DTIME}(t) = \text{DTIME}(g(t))$. Indeed, let $S \in \text{DTIME}(g(t))$ and suppose that the i^{th} algorithm decides S in time at most $g(t)$; that is, for every n, it holds that $t_i(n) \leq g(t(n))$. Then (by the construction), for every n satisfying $\mu(n) \geq i$, it holds that $t_i(n) < t(n)$, and it follows that the i^{th} algorithm decides S in time at most t on all but finitely many inputs. Combining this algorithm with a "look-up table" machine that handles the exceptional inputs, the theorem follows. ■

Comment: The function t defined by the foregoing proof is computable in time that exceeds $g(t)$. Specifically, the presented procedure computes $t(n)$ (as well as $g(f(n))$) in time $\widetilde{O}(2^n \cdot g(t(n)) + T_g(t(n)))$, where $T_g(m)$ denotes the number of steps required to compute $g(m)$ on input m.

Speed-up Theorems. Theorem 4.7 can be viewed as asserting that some time complexity classes (i.e., $\text{DTIME}(g(t))$ in the theorem) collapse to lower classes (i.e., to $\text{DTIME}(t)$). A conceptually related phenomenon is of problems that have no optimal algorithm (not even in a very mild sense); that is, every algorithm for these ("pathological") problems can be drastically sped-up. It follows that the complexity of these problems can not be defined (i.e., as the complexity of the best algorithm solving this problem). The following drastic speed-up theorem should not be confused with the linear speed-up that is an artifact of the definition of a Turing machine (see Exercise 4.4).[3]

Theorem 4.8 (the time speed-up theorem): *For every computable* (and super-linear) *function g there exists a decidable set S such that if $S \in \text{DTIME}(t)$ then $S \in \text{DTIME}(t')$ for t' satisfying $g(t'(n)) < t(n)$.*

Taking $g(n) = n^2$ (or $g(n) = 2^n$), the theorem asserts that, for every t, if $S \in \text{DTIME}(t)$ then $S \in \text{DTIME}(\sqrt{t})$ (resp., $S \in \text{DTIME}(\log t)$). Note that Theorem 4.8 can be applied any number of times, which means that we cannot give a reasonable estimate to the complexity of deciding membership in S. In contrast, recall that in some important cases (see, e.g., Theorem 2.31), optimal algorithms for solving computational problems do exist. Specifically, algorithms solving (candid) search problems in NP cannot be speed-up (see Theorem 2.31), nor can the computation of a universal machine (see Theorem 4.5).

We refrain from presenting a proof of Theorem 4.8, but comment on the complexity of the sets involved in this proof. The proof (presented in [116, Sec. 12.6]) provides a construction of a set S in $\text{DTIME}(t') \setminus \text{DTIME}(t'')$ for any $t'(n) = h(n - O(1))$ and $t''(n) = h(n - \omega(1))$, where $h(n)$ denoted g iterated n times on 2 (i.e., $h(n) = g^{(n)}(2)$, where $g^{(i+1)}(m) = g(g^{(i)}(m))$ and $g^{(1)} = g$). The set S is constructed such that for every $i > 0$ there exists a $j > i$ and an algorithm that decides S in time t_i but not in time t_j, where $t_k(n) = h(n - k)$.

4.3 Space Hierarchies and Gaps

Hierarchy and Gap Theorems analogous to Theorem 4.3 and Theorem 4.7, respectively, are known for space complexity. In fact, since space-efficient emulation of space-bounded machines is simpler than time-efficient emulations of time-bounded machines, the results tend to be sharper. This is most conspicuous in the case of the separation result (stated next), which is optimal (in light of linear speed-up results; see Exercise 4.4).

[3] We note that this fact was implicitly addressed in the proof of Theorem 4.3, by allowing an emulation overhead that depends on the length of the description of the emulated machine.

Before stating the result, we need a few preliminaries. We refer the reader to §1.2.3.4 for a definition of space complexity (and to Chapter 5) for further discussion. As in case of time complexity, we consider a specific model of computation, but the results hold for any other reasonable and general model. Specifically, we consider three-tape Turing machines, because we designate two special tapes for input and output. For any function $s : \mathbb{N} \to \mathbb{N}$, we *denote by* $\text{DSPACE}(s)$ *the class of decision problems that are solvable in space complexity* s. Analogously to Definition 4.2, we call a function $s : \mathbb{N} \to \mathbb{N}$ space constructible if there exists an algorithm that on input n outputs $s(n)$ using at most $s(n)$ cells of the work-tape. Actually, functions like $s_1(n) = \log n$, $s_2(n) = (\log n)^2$, and $s_3(n) = 2^n$ are computable using $\log s_i(n)$ space.

Theorem 4.9 (space hierarchy for three-tape Turing machines): *For any space computable function* s_2 *and every function* s_1 *such that* $s_2 = \omega(s_1)$ *and* $s_1(n) > \log n$ *it holds that* $\text{DSPACE}(s_1)$ *is strictly contained in* $\text{DSPACE}(s_2)$.

Theorem 4.9 is analogous to the traditional version of Theorem 4.3 (rather to the one we presented), and is proven using the alternative approach sketched in Footnote 2. The details are left as an exercise (see Exercise 4.5).

Chapter Notes

The material presented in this chapter predates the theory of NP-completeness and the dominant stature of the P-vs-NP Question. At these early days, the field (to be known as complexity theory) did not yet develope an independent identity and its perspectives were dominated by two classical theories: the theory of computability (and recursive function) and the theory of formal languages. Nevertheless, we believe that the results presented in this chapter are interesting for two reasons. Firstly, as stated upfront, these results address the natural question of under what conditions do more resources help. Secondly, their inapplicability to questions of the type of P-vs-NP, illustrates the non-generic flavor of these questions as referring to specific (and natural) aspects of the "complexity of computation".

The hierarchy theorems (e.g., Theorem 4.3) were proved by Hartmanis and Stearns [109]. Gap theorems (e.g., Theorem 4.7, often referred to as Borodin's Gap Theorem) were proven by Borodin [44]. A axiomatic treatment of complexity measures and corresponding speed-up theorems (e.g., Theorem 4.8, often referred to as Blum's Speed-up Theorem) are due to Blum [36].

Exercises

Exercise 4.1 For $s < 2^n$, prove that the number of Boolean functions over $\{0,1\}^n$ that are computed by Boolean circuits of size s is at least $2^{s/O(\log s)}$ and at most $\binom{s^2}{s}$.

Guideline: Any Bollean function $f : \{0,1\}^\ell \to \{0,1\}$ can be computed by a circuit of size $s = O(\ell \cdot 2^\ell)$. Thus, circuits of size s may compute $2^{2^\ell} > 2^{s/O(\log s)}$ different Boolean

functions. On the other hand, the number of circuits of size s is less than $\binom{s^2}{s}$, where the upper-bound represents the number of possible choices of pair of gates that feed any gate in the circuit.

Exercise 4.2 Referring to a reasonable model of computation (and assuming that the input length is not given explicitly (e.g., as in Definition 10.10)), prove that any algorithm that has sub-linear time-complexity actually has constant time-complexity.
(Hint: Consider the question of whether or not there exists an infinite set of strings S such that when invoked on any input $x \in S$ the algorithm reads all of x. Note that if S is infinite then the algorithm cannot have sub-linear time-complexity, and prove that if S is finite then the algorithm has constant time-complexity.)

Exercise 4.3 Prove that the two definitions of NTIME, presented in §4.2.1.3, are related up to logarithmic factors. Note the importance of condition that V has linear (rather than polynomial) time-complexity.
(Hint: when emulating a non-deterministic machine by the verification procedure V, encode the non-deterministic choices in y while using sufficient padding such that t steps of the original machine can be emulated in $|y| = O(t \log t)$ steps of V.)

Exercise 4.4 Prove that any problem that can be solved by a two-tape Turing machine that has time complexity t can be solved by another two-tape Turing machine having time complexity t', where $t'(n) = O(n) + (t(n)/2)$.
(Hint: Consider a machine that uses a larger alphabet, capable of encoding a constant number of symbols of the original machine, and thus capable of emulating a constant number of steps of the original machine in a smaller constant number of steps. Note that the $O(n)$ term accounts to a preprocessing required to encode the input in the work-alphabet of the new machine, and that a similar result for one-tape Turing machine seems to require a $O(n^2)$ term.)
State and prove an analogous result for space complexity, when using the standard definition of space as recalled in Section 4.3. Note that this result does not hold with respect to "binary space complexity" as defined in Section 5.1.

Exercise 4.5 Prove Theorem 4.9. As a warm-up, assume that s_1 (rather than s_2) is space constructible.

Guideline: Note that providing a space-efficient emulation of one machine by another machine is easier than providing an analogous time-efficient emulation.

Chapter 5

Space Complexity

Open are the double doors of the horizon; unlocked are its bolts.

Philip Glass, Akhnaten, Prelude

Whereas the number of steps taken during a computation is the primary measure of its efficiency, the amount of temporary storage used by the computation is also a major concern. Furthermore, in some settings, space is even more scarce than time.

This chapter is devoted to the study of the space complexity of computations, while focusing on two rather extreme cases. The first case is that of algorithms having logarithmic space complexity. We view such algorithms as utilizing the naturally minimal amount of temporary storage, where the term "minimal" is used here in an intuitive (but somewhat inaccurate) sense, and note that logarithmic space complexity seems a more stringent requirement than polynomial time. The second case is that of algorithms having polynomial space complexity. Such algorithms can perform almost all computational tasks considered in this book (i.e., the class $\mathcal{PSPACE}$ contains almost all complexity classes considered in this book).

In addition to the intrinsic interest in space complexity, its study provides an interesting perspective on the study of time complexity. For example, in contrast to the common conjecture by which $\mathcal{NP} \neq \text{co}\mathcal{NP}$, we shall see that analogous space complexity classes (e.g., $\mathcal{NL}$) are closed under complementation (e.g., $\mathcal{NL} = \text{co}\mathcal{NL}$).

Summary: We first consider algorithms of logarithmic space complexity. Such algorithms may be used for solving various search and decision problems, providing reductions among such problems, and yielding a strong notion of uniformity for Boolean circuits. The highlight of this part is a log-space algorithm for exploring (undirected) graphs.

We then turn to non-deterministic machines, focusing on the complexity class $\mathcal{NL}$ that is captured by the problem of deciding directed connec-

tivity of (directed) graphs. The highlight of this part is a proof that $\mathcal{NL} = \text{co}\mathcal{NL}$.

We conclude with a short discussion of the class $\mathcal{PSPACE}$, proving that the set of satisfiable quantified Boolean formulae is $\mathcal{PSPACE}$-complete (under polynomial-time reductions). We mention the similarity between this proof and the proof that $\text{NSPACE}(s) \subseteq \text{DSPACE}(O(s^2))$.

We stress that, as in the case of time complexity, the main results presented in this chapter hold for any reasonable model of computation.[1] In fact, when properly defined, space complexity is even more robust than time complexity. Still, for sake of clarity, we often refer to the specific model of Turing machines.

Organization. Space complexity seems to behave quite differently from time complexity, and seems to require a different mind-set as well as auxiliary conventions. Some of these are discussed in Section 5.1. We then turn to the study of logarithmic space complexity (see Section 5.2) and the corresponding non-deterministic version (see Section 5.3). Finally, we consider polynomial space complexity (see Section 5.4).

5.1 General preliminaries

Space complexity is meant to measure the amount of temporary storage (i.e., computer's memory) used when performing a computational task. Since much of our focus will be on using an amount of memory that is sub-linear in the input length, it is important to use a model in which one can differentiate memory used for computation from memory used for storing the initial input or the final output. That is, we do not want to count the input and output themselves within the space of computation, and thus formulate that they are delivered on special devices that are not considered memory. On the other hand, we have to make sure that the input and output devices cannot be abused for providing work space (which is uncounted for). This leads to the convention by which the input device (e.g., a designated input-tape of a multi-tape Turing machine) is read-only, whereas the output device (e.g., a designated output-tape of a such machine) is write-only. Thus, space complexity accounts for the use of space on the other (storage) devices (e.g., the work-tapes of a multi-tape Turing machine)

Fixing a concrete model of computation (e.g., multi-tape Turing machines), we *denote by* $\text{DSPACE}(s)$ *the class of decision problems that are solvable in space complexity* s. The space complexity of search problems is defined analogously. Specifically, the *standard definition of space complexity* (see §1.2.3.4) *refers to the number of cells of the work-tape scanned by the machine on each input.* We prefer, however, an alternative definition, which provides a more accurate account of the

[1] The only exceptions appear in Exercises 5.3 and 5.14, which refer to the notion of a crossing sequence. The use of this notion in these proofs assumes that the machine scans its storage devices in a serial manner. We stress that the various notions of an instantaneous configuration do not assume such a machine model.

actual storage. Specifically, the binary space complexity of a computation *refers to the the number of bits that can be stored in these cells*, thus multiplying the number of cells by the logarithm of the finite set of work symbols of the machine.[2]

The difference between the two definitions is mostly immaterial, since it amounts to a constant factor and we will discard such factors. Nevertheless, aside from being conceptually right, the definition of binary space complexity will facilitate some technical details (because the number of possible configurations is explicitly upper-bounded in terms of binary space complexity whereas the relation to the standard definition depends on the machine in question). Towards such applications, one may also count the finite state of the machine in its space complexity. Furthermore, for sake of simplicity, we also assume that the machine does not scan the input-tape beyond the boundaries of the input, which are indicated by special symbols.

We stress that individual locations of the (read-only) input-tape (or device) may be read several times. This is essential for many algorithms that use a sublinear amount of space (because such algorithms may need to scan their input more than once while they cannot afford copying their input to their storage device). In contrast, rewriting on (the same loaction of) the write-only output-tape is inessential, and in fact can be eliminated at a relatively small cost (see Exercise 5.1).

5.1.1 On the minimal amount of useful computation space

Bearing in mind that one of our main objectives is identifying natural sub-classes of $\mathcal{P}$, we consider the question of what is the minimal amount of space that allows for meaningful computations. We note that regular sets [116, Chap. 2] are decidable by constant-space Turing machines and that this is all that the latter can decide (see, e.g., [116, Sec. 2.6]). It is tempting to say that sub-logarithmic space machines are not more useful than constant-space machines, because it *seems* impossible to allocate a sub-logarithmic amount of space. This wrong intuition is based on the presumption that the allocation of a non-constant amount of space requires explicitly computing the length of the input, which in turn requires logarithmic space. However, this presumption is wrong: the input itself (in case it is of a proper form) can be used to determine its length, whereas in case the input is not of the proper form then this fact may be detectable (within sub-logarithmic space). In fact, *for* $\ell(n) = \log\log n$*, the class* $\text{DSPACE}(O(\ell))$ *is a proper superset of* $\text{DSPACE}(O(1))$; see Exercise 5.2. In contrast to Exercise 5.2, double-logarithmic space is indeed the smallest amount of space that is more useful than constant space; that is, *for* $\ell(n) = \log\log n$*, it holds that* $\text{DSPACE}(o(\ell)) = \text{DSPACE}(O(1))$; see Exercise 5.3.

In spite of the fact that some non-trivial things can be done in sub-logarithmic space complexity, the lowest space complexity class that we shall study in depth is logarithmic space (see Section 5.2). As we shall see, this class is the natural habitat of several fundamental computational phenomena.

[2] We note that, unlike in the context of time-complexity, linear speed-up (as in Exercise 4.4) does not seem to represent an actual saving in space resources. Indeed, time can be sped-up by using stronger hardware (i.e., a Turing machine with a bigger work alphabet), but the actual space is not really affected by partitioning it into bigger chunks (i.e., using bigger cells).

A parenthetical comment (or a side lesson). Before proceeding let us highlight the fact that a naive presumption about generic algorithms (i.e., that the use of a non-constant amount of space requires explicitly computing the length of the input) could have led us to a wrong conclusion. This demonstrates the danger in making ("reasonably looking") presumptions about *arbitrary* algorithms, which in turn are the subject of a complexity lower-bound.

5.1.2 Time versus Space

Space complexity behaves very different from time complexity and indeed different paradigms are used in studying it. One notable example is provided by the context of algorithmic composition, discussed next.

5.1.2.1 Two composition lemmas

Unlike time, space can be re-used; but, on the other hand, intermediate results of a computation cannot be recorded for free. These two conflicting aspects are captured in the following composition lemma.

Lemma 5.1 (naive composition): *Let* $f_1 : \{0,1\}^* \to \{0,1\}^*$ *and* $f_2 : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ *be computable in space* s_1 *and* s_2*, respectively.*[3] *Then* f *defined by* $f(x) \stackrel{\text{def}}{=} f_2(x, f_1(x))$ *is computable in space* s *such that*

$$s(n) = \max(s_1(n), s_2(n + \ell(n))) + \ell(n) + O(1)\,,$$

where $\ell(n) = \max_{x \in \{0,1\}^n}\{|f_1(x)|\}$.

That is, $f(x)$ is computed by first computing and storing $f_1(x)$, and then re-using the space (used in the first computation) when computing $f_2(x, f_1(x))$. The additional term of $\ell(n)$ is due to storing the intermediate result (i.e., $f_1(x)$). Lemma 5.1 is useful when ℓ is relatively small, but in many cases $\ell \gg \max(s_1, s_2)$. In these cases, the following composition lemma is more useful.

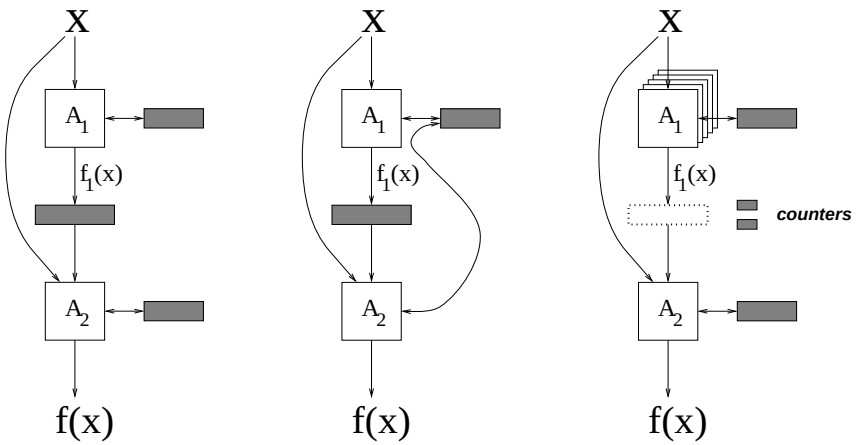
Lemma 5.2 (emulative composition): *Let* f_1, f_2, s_1, s_2, ℓ *and* f *be as in Lemma 5.1. Then* f *is computable in space* s *such that*

$$s(n) = s_1(n) + s_2(n + \ell(n)) + O(\log(n + \ell(n)))\,.$$

The alternative compositions are depicted in Figure 5.1 (which also shows the most straightforward composition of A_1 and A_2 that makes no attempt to economize space).

Proof: The idea is avoiding the storage of the temporary value of $f_1(x)$, by computing each of its bits ("on the fly") whenever it is needed for the computation of f_2. That is, we do not start by computing $f_1(x)$, but rather start by computing $f_2(x, f_1(x))$ although we do not have some of the bits of the relevant input. The

[3] Here (and throughout the chapter) we assume, for simplicity, that all complexity bounds are monotonically non-decreasing.



The leftmost figure shows the trivial composition (which just invokes A_1 and A_2 without attempt to economize storage), the middle figure shows the naive composition (of Lemma 5.1), and the rightmost figure shows the emulative composition (of Lemma 5.2). In all figures the filled rectangles represent designated storage spaces. The dotted rectangle represents a virtual storage device.

Figure 5.1: Algorithmic composition for space-bounded computation

missing bits will be computed (and re-computed) whenever we need them in the computation of $f_2(x, f_1(x))$. Details follow.

Let us assume, for simplicity, that algorithm A_1 never rewrites on (the same loaction of) its write-only output-tape. As shown in Exercise 5.1, this assumption can be justified at an additive cost of $O(\log \ell(n))$.[4]

Let A_1 and A_2 be the algorithms (for computing f_1 and f_2, respectively) guaranteed in the hypothesis. Then, on input $x \in \{0,1\}^n$, we invoke algorithm A_2 (for computing f_2). Algorithm A_2 is invoked on a virtual input, and so when emulating each of its steps we should provide it with the relevant bit. Thus, we should also keep track of the location of A_2 on the imaginary (virtual) input tape. Whenever A_2 seeks to read the i^{th} bit of its input, where $i \in [n + \ell(n)]$, we provide A_2 with this bit by reading it from x if $i \leq n$ and invoke $A_1(x)$ otherwise. When invoking $A_1(x)$ we provide it with a virtual output tape, which means that we get the bits of its output one-by-one and do not record them anywhere. Instead, we count until reaching the $(i-n)^{\text{th}}$ output bit, which we then pass to A_2 (as the i^{th} bit of $\langle x, f_1(x)\rangle$).

Note that while invoking $A_1(x)$, we suspend the execution of A_2 but keep its current configuration such that we can resume the execution (of A_2) once we get the desired bit. Thus, we need to allocate separate space for the computation of A_2

[4] Alternatively, the idea presented in Exercise 5.1 can be incorporated directly in the current proof.

and for the computation of A_1. In addition, we need to allocate separate storage for maintaining the aforementioned counters (i.e., the bit-location currently read by A_2 and the index of the bit currently produced in the current invocation of A_1). ■

Reflection. The algorithm presented in the proof of Lemma 5.2 is wasteful in terms of time: it re-computes $f_1(x)$ again and again (i.e., once per each access of A_2 to the second part of its input). Indeed, our aim was economizing on space and not on time (and the two goals may be conflicting (see, e.g., [56, Sec. 4.3])).

5.1.2.2 An obvious bound

The time complexity of an algorithm is essentially upper-bounded by an exponential in its space complexity. This is due to an upper-bound on the number of possible instantaneous "configurations" of the algorithm (as formulated in the proof of Theorem 5.3) and the fact that if the computation passes through the same configuration twice then it must loop forever.

Theorem 5.3 *If an algorithm A has binary space complexity s and halts on every input then it has time complexity t such that $t(n) = n \cdot 2^{s(n)+\log_2 s(n)}$.*

Note that for $s(n) = \Omega(\log n)$, the factor of n can be absorbed by $2^{O(s(n))}$, and so we may just write $t(n) = 2^{O(s(n))}$.

Proof: The proof refers to the notion of an *instantaneous configuration* (in a computation). Before starting, we warn the reader that this notion may be given different definitions, each tailored to the application at hand. All these definitions share the desire to specify *variable information* that together with some *fixed information* determines the next step of the computation being analyzed. In the current proof, we fix an algorithm A and an input x, and consider as variable the contents of the storage device (e.g., work-tape of a Turing machine as well as its finite state) and the machine's location on the input device and on the storage device. Thus, an instantaneous configuration of $A(x)$ consists of the latter three objects (i.e., the contents of the storage device and a pair of locations), and can be encoded by a binary string of length $\ell(|x|) = s(|x|) + \log_2 |x| + \log_2 s(|x|)$.

The key observation is that the computation $A(x)$ cannot pass through the same computation twice, because otherwise the computation $A(x)$ passes through this configuration infinitely many times, which means that it does not halt. Intuitively, the point is that the fixed information (i.e., A and x) together with the configuration, determines the next step of the computation. Thus, whatever happens (i steps) after the first time that the computation $A(x)$ passes through configuration γ, will also happen (i steps) after the second time that the computation $A(x)$ passes through γ.

By the forgoing observation, we infer that $t(|x|) < 2^{\ell(|x|)}$, and the theorem follows. ■

5.1.2.3 Subtleties regarding space-bounded reductions

Lemmas 5.1 and 5.2 suffice for the analysis of the affect of many-to-one reductions in the context of space-bounded computations. Specifically:

1. (In spirit of Lemma 5.1:)[5] If f is reducible to g via a many-to-one reduction that can be computed in space s_1, and g is computable in space s_2, then f is computable in space s such that $s(n) = \max(s_1(n), s_2(\ell(n))) + \ell(n)$, where $\ell(n)$ denotes the maximum length of the image of the reduction when applied to some n-bit string.

2. (In spirit of Lemma 5.2:) For f and g as in Item 1, it follows that f is computable in space s such that $s(n) = s_1(n) + s_2(\ell(n)) + O(\log \ell(n))$.

Note that by Theorem 5.3, it holds that $\ell(n) \leq 2^{s_1(n)+\log_2 s_1(n)} \cdot n$. We stress the fact that ℓ is not bounded by s_1 (as in the analogous case of time-bounded computation), but rather by $\exp(s_1)$.

Things gets much more complicated when we turn to general (space-bounded) reductions, especially when referring to such reductions that make a non-constant number of queries. A preliminary issue is defining the space complexity of general reductions (i.e., of oracle machines). In the standard definition, the length of the queries and answers is not counted in the space complexity, but the queries of the reduction (resp., answers given to it) are written on (resp., read from) a special device that is write-only (resp., read-only) for the reduction (and read-only (resp., write-only) for the invoked oracle). Note that these convention are analogous to the conventions regarding input and output (as well as fit the definitions of space-bounded many-to-one reductions (see Section 5.2.2)). The rest of the discussion, which is quite advanced and laconic (and is inessential to the rest of the chapter), concerns two additional issues.

Recall that the complexity of the algorithm resulting from the composition of an oracle machine and an actual algorithm depends on the length of the queries made by the oracle machine. The length of the first query is upper-bounded by an exponential function in the space complexity of the oracle machine, but the same does not necessarily hold for subsequent queries, *unless some conventions are added to enforce it*. For example, consider a reduction, that on input x and access to the oracle f such that $f(z) = 1^{2^{|z|}}$, invokes the oracle $|x|$ times, where each time it uses as a query the answer obtained to the previous query. This reduction uses constant space, but produces queries that are exponentially longer than the input, whereas the first query of any constant-space reduction has length that is linear in its input. This problem can be resolved by placing explicit bounds on the length of the queries that space-bounded reductions are allowed to make; for example, we may bound the length of all queries by the obvious (exponential in the space complexity) bound that holds for the length of the first query.

With the aforementioned convention (or restriction) in place, let us consider the composition of *general* space-bounded reductions with a space-bounded implementation of the oracle. Specifically, we say that a reduction is (ℓ, ℓ')-restricted if,

[5] Here and in the next item, we refer to the case that $f(x) = g(f_1(x))$ rather than to the more general case where $f(x) = g(x, f_1(x))$. Consequently, s_2 is applied to $\ell(n)$ rather than to $n + \ell(n)$.

on input x, all oracle queries are of length at most $\ell(|x|)$ and the corresponding oracle answers are of length at most $\ell'(|x|)$. It turns out that naive composition (in the spirit of Lemma 5.1) remains valid, whereas the emulative composition of Lemma 5.2 breaks down (in the sense that it yield very weak results).

1. Following Lemma 5.1, we claim that *if* Π *can be computed in space* s_1 *when given* (ℓ,ℓ')*-restricted oracle access to* Π' *and* Π' *is solvable is space* s_2*, then* Π *is solvable in space* s *such that* $s(n) = s_1(n) + s_2(\ell(n)) + \ell(n) + \ell'(n)$. The claim is proved by using a naive emulation that allocates separate space for the reduction (i.e., oracle machine) itself, the emulation of its query and answer devices, and the algorithm solving Π'. Note that here we cannot re-use the space of the reduction when running the algorithm that solves Π', because the reduction's computation continues after the oracle answer is obtained.

   A related composition result is presented in Exercise 5.5. It yields $s(n) = 2s_1(n) + s_2(\ell(n)) + 2\ell'(n) + \log_2 \ell(n)$, which for $\ell(n) < 2^{2s_1(n)}$ means $s(n) = 4s_1(n) + s_2(\ell(n)) + 2\ell'(n)$.

2. Turning to the approach underlying Lemma 5.2, we get into more serious trouble. Recomputing the answer of the i^{th} query requires recomputing the query itself, which unlike in Lemma 5.2 is not the input to the reduction but rather depends on the answers to prior queries, which need to be recomputed as well. Thus, the space required for such an emulation may be linear in the number of queries. In fact, we should not expect any better, because any computation of space complexity s can be performed by a constant-space $(2s, 2s)$-restricted reduction to a problem that is solvable in constant-space (see Exercise 5.6).

An alternative notion of space-bounded reductions is discussed in §5.2.4.2. This notion is more cumbersome and more restricted, but it allows recursive composition with a smaller overhead than the two options explored above.

5.1.2.4 Complexity hierarchies and gaps

Recall that more space allows for more computation (see Theorem 4.9), provided that the space-bounding function is "nice" in an adequate sense. Actually, the proofs of space-complexity hierarchies and gaps are simpler than in the analogous proofs for time-complexity, because emulations are easier in the context of space-bounded algorithms (cf. Section 4.3).

5.1.2.5 Simultaneous time-space complexity

Recall that, for space complexity that is at least logarithmic, the time of a computation is always upper-bounded by an exponential function in the space complexity (see Theorem 5.3). Thus, polylogarithmic space complexity may extend beyond polynomial-time, and it make sense to define a class like $\mathcal{SC}$ that consists of all

decision problems that may be solved by a polynomial-time algorithm of polylogarithmic space complexity. The class $\mathcal{SC}$ is indeed a natural sub-class of $\mathcal{P}$ (and contains the class $\mathcal{L}$, which is defined in Section 5.2.1).[6]

In general, one may define $\text{DTISP}(t,s)$ as the class of decision problems solvable by an algorithm that has time complexity t and space complexity s. Note that $\text{DTISP}(t,s) \subseteq \text{DTIME}(t) \cap \text{DSPACE}(s)$ and that a strict containment may hold. We mention that $\text{DTISP}(\cdot,\cdot)$ provides the arena for the only known absolute (and highly non-trivial) lower-bound regarding $\mathcal{NP}$; see [75]. We also note that lower bounds on time-space trade-offs (see, e.g., [56, Sec. 4.3]) may be stated as referring to the classes $\text{DTISP}(\cdot,\cdot)$.

5.1.3 Circuit Evaluation

Recall that Theorem 3.1 asserts the existence of a polynomial-time algorithm that, given a circuit $C : \{0,1\}^n \to \{0,1\}^m$ and an n-bit long string x, returns $C(x)$. For circuits of bounded fan-in, the space complexity of such an algorithm can be made linear in the depth of the circuit and logarithmic in its size. This is obtained by the following DFS-type algorithm.

The algorithm (recursively) determines the value of a gate in the circuit by first determining the value of its first in-coming edge and next determining the value of the second in-coming edge. Thus, the recursive procedure, started at each output terminal of the circuit, needs only store the path that leads to the currently processed vertex as well as the temporary values computed for each ancestor. Note that this path is determined by indicating, for each vertex on the path, whether we currently process its first or second in-coming edge. In case we currently process the vertex's second in-coming edge, we need also store the value computed for its first in-coming edge.

The temporary storage used by the foregoing algorithm, on input (C,x), is thus $2d_C + O(\log|x| + \log|C(x)|)$, where d_C denotes the depth of C. The first term in the space-bound accounts for the core activity of the algorithm (i.e., the recursion), whereas the other terms account for the overhead involved in assigning the bits of x to the corresponding input terminals of C and in scanning all output terminals of C.

5.2 Logarithmic Space

Although Exercise 5.2 asserts that "there is life below log-space," logarithmic space seems to be the smallest amount of space that supports interesting computational phenomena. In particular, logarithmic space is required for merely maintaining an auxiliary counter that holds a position in the input, which seems required in many computations. On the other hand, logarithmic space suffices for solving many natural computational problems, for establishing reductions among many natural computational problems, and for a stringent notion of uniformity (of families of

[6] We also mention that $\mathcal{BPL} \subseteq \mathcal{SC}$, where $\mathcal{BPL}$ is defined in §6.1.4.1 and the result is proved in Section 8.5 (see Theorem 8.23).

Boolean circuits). Indeed, an important feature of logarithmic space computations is that they are a natural subclass of the polynomial-time computations (see Theorem 5.3).

5.2.1 The class L

Focusing on decision problems, we denote by $\mathcal{L}$ the class of decision problems that are solvable by algorithms of logarithmic space complexity; that is, $\mathcal{L} = \cup_c \text{DSPACE}(\ell_c)$, where $\ell_c(n) \stackrel{\text{def}}{=} c \log_2 n$. Note that, by Theorem 5.3, $\mathcal{L} \subseteq \mathcal{P}$. As hinted, many natural computational problems are in $\mathcal{L}$ (see Exercises 5.4 and 5.7 as well as Section 5.2.4). On the other hand, *it is widely believed that* $\mathcal{L} \neq \mathcal{P}$.

5.2.2 Log-Space Reductions

Another class of important log-space computations is the class of *logarithmic space reductions*. In light of the subtleties discussed in §5.1.2.3 we confine ourselves to the case of many-to-one reductions. Analogously to the definition of Karp-reductions (Definition 2.10), we say that f is a log-space many-to-one reduction of S to S' if f is log-space computable and, for every x, it holds that $x \in S$ if and only if $f(x) \in S'$. Clearly, if S is so reducible to $S' \in \mathcal{L}$ then $S \in \mathcal{L}$. Similarly, one can define a log-space variant of Levin-reductions (Definition 2.11). Both types of reductions are transitive (see Exercise 5.8). Note that Theorem 5.3 applies in this context and implies that these reductions run in polynomial-time. Thus, the notion of a log-space many-to-one reduction is a special case of a Karp-reduction.

We observe that all known Karp-reductions establishing NP-completeness results are actually log-space reductions. This is easily verifiable in the case of the reductions presented in Section 2.3.3 (as well as in Section 2.3.2). For example, consider the generic reduction to CSAT presented in the proof of Theorem 2.20: The constructed circuit is "highly uniform" and can be easily constructed in logarithmic-space (see Section 5.2.3). A degeneration of this reduction suffices for proving that every problem in $\mathcal{P}$ is log-space reducible to the problem of evaluating a given circuit on a given input. Note that the latter problem is in $\mathcal{P}$, and thus we may say that it is *P-complete under log-space reductions*.

Theorem 5.4 (The complexity of Circuit Evaluation): *Let* CEVL *denote the set of pairs* (C, α) *such that* C *is a Boolean circuit and* $C(\alpha) = 1$. *Then* CEVL *is in* $\mathcal{P}$ *and every problem in* $\mathcal{P}$ *is log-space Karp-reducible to* CEVL.

Proof Sketch: Recall that the observation underlying the proof of Theorem 2.20 (and used also in the proof of Theorem 3.6) is that the computation of a Turing machine can be emulated by a "highly uniform" circuit. In the proof of Theorem 2.20, we hardwired the input to the reduction (denoted x) into the circuit (denoted C_x) and introduced input terminals corresponding to the bits of the NP-witness (denoted y). In the current context we leave x as an input to the circuit, while noting that the auxiliary NP-witness does not exists (or has length zero). Thus, the reduction from $S \in \mathcal{P}$ to CEVL maps the instance x (for S) to the pair

$(C_{|x|}, x)$, where $C_{|x|}$ is a circuit that emulates the computation of the machine that decides membership in S (on any $|x|$-bit long input). □

The impact of P-completeness under log-space reductions. Indeed Theorem 5.4 implies that $\mathcal{L} \neq \mathcal{P}$ if any only if $\texttt{CEVL} \notin \mathcal{L}$. Other natural problems were proved to have the same property (i.e., being P-complete under log-space reductions; cf. [57]).

Log-space reductions are used to define completeness with respect to other classes that are assumed to extend beyond $\mathcal{L}$. This restriction of the power of the reduction is definitely needed when the class of interest is contained in $\mathcal{P}$ (e.g., $\mathcal{NL}$, see Section 5.3.2). In general, we say that a problem Π is $\mathcal{C}$-complete under log-space reductions if Π is in $\mathcal{C}$ and every problem in $\mathcal{C}$ is log-space (many-to-one) reducible to Π. In such a case, if $\Pi \in \mathcal{L}$ then $\mathcal{C} \subseteq \mathcal{L}$.

As in the case of polynomial-time reductions, we wish to stress that the relevance of log-space reductions extends beyond being a tool for defining complete problems.

5.2.3 Log-Space uniformity and stronger notions

Strengthening Definition 3.3, we say that a family of circuits $(C_n)_n$ is log-space uniform if there exists an algorithm A that on input n outputs C_n while using space that is logarithmic in the size of C_n. As implied by Theorem 5.5 (and implicitly proved in Theorem 5.4), *the computation of any polynomial-time algorithm can be emulated by a log-space uniform family of* (bounded fan-in) *polynomial-size circuits.* On the other hand, in continuation to Section 5.1.3, we note that *log-space uniform circuits of bounded fan-in and logarithmic depth can be emulated by an algorithm of logarithmic space complexity* (i.e., $\mathcal{NC}^1$ is in log-space; see Exercise 5.7).

As mentioned in Section 3.1.1, stronger notions of uniformity have been considered. Specifically, in analogy to the discussion in §E.2.1.2, we say that $(C_n)_n$ has a strongly explicit construction if there exists an algorithm that runs in polynomial-time and linear-space such that, on input n and v, the algorithm returns the label of vertex v in C_n as well as the list of its children (or an indication that v is not a vertex in C_n). Note that if $(C_n)_n$ has a strongly explicit construction then it is log-space uniform, because the length of the description of a vertex in C_n is logarithmic in the size of C_n. The proof of Theorem 5.4 actually establishes the following.

Theorem 5.5 (strongly uniform circuits emulating $\mathcal{P}$): *For every polynomial-time algorithm A there exists a strongly explicit construction of a family of polynomial-size circuits $(C_n)_n$ such that for every x it holds that $C_{|x|}(x) = A(x)$.*

5.2.4 Undirected Connectivity

Exploring a graph (e.g., towards determining its connectivity) is one of the most basic and ubiquitous computational tasks regarding graphs. The standard graph

exploration algorithms (e.g., BFS and DFS) require temporary storage that is linear in the number of vertices. In contrast, the algorithm presented in this section uses temporary storage that is only logarithmic in the number of vertices. In addition to demonstrating the power of log-space computation, this algorithm (or rather its actual implementation) provides a taste of the type of issues arising in the design of sophisticated log-space algorithms.

The intuitive task of "exploring a graph" is captured by the task of *deciding whether a given graph is connected.*[7] In addition to the intrinsic interest in this natural computational problem, we mention that it is computationally equivalent (under log-space reductions) to numerous other computational problems (see, e.g., Exercise 5.12). We note that some seemingly related problems seem harder; for example, determining directed connectivity (in directed graphs) captures the essence of the class $\mathcal{NL}$ (see Section 5.3.2). In view of this situation, we emphasize the fact that the computational problem considered here refers to undirected graphs by calling it undirected connectivity.

Theorem 5.6 *Deciding undirected connectivity* (UCONN) *is in* $\mathcal{L}$

The algorithm is based on the fact that UCONN is easy in the special case that the graph consists of a collection of constant degree expanders (see Appendix E.2). In particular, if the graph has constant degree and logarithmic diameter then it can be explored using a logarithmic amount of space (which is used for determining a generic path from a fixed starting vertex).[8]

However, the input graph does not necessarily consist of a collection of constant degree expanders. The main idea is then to transform the input graph into one that does satisfy the aforementioned condition, while preserving the number of connected components of the graph. Needless to say, the key point is performing such a transformation in logarithmic space. The rest of this section is devoted to the description of such a transformation. We first present the basic approach and next turn to the highly non-trivial implementation details.

Teaching note: We recommend leaving the actual proof of Theorem 5.6 for advanced reading. The main reason is its heavy dependence on technical material that is beyond the scope of a course in complexity theory.

We first note that it is easy to transform the input graph $G_0 = (V_0, E_0)$ into a constant-degree graph G_1 that preserves the number of connected components in G_0. Specifically, each vertex $v \in V$ having degree $d(v)$ (in G_0) is represented by a cycle C_v of $d(v)$ vertices (in G_1), and each edge $\{u, v\} \in E_0$ is replaced by an edge having one end-point on the cycle C_v and the other end-point on the cycle C_u such that each vertex in G_1 has degree three (i.e., has two cycle edges and a single intra-cycle edge). This transformation can be performed using logarithmic space, and

[7] See Appendix G.1 for basic terminology.

[8] Indeed, this is analogous to the circuit evaluation algorithm of Section 5.1.3, where the circuit depth corresponds to the diameter and the bounded fan-in corresponds to the constant degree. For further details, see Exercise 5.9.

thus (relying on Lemma 5.2) we assume throughout the rest of the proof that the input graph has degree three. Our goal is to transform this graph into a collection of expanders, while maintaining the number of connected components. In fact, *we shall describe the transformation while pretending that the graph is connected, while noting that otherwise the transformation acts separately on each connected component.*

A couple of technicalities. For a constant integer $d > 2$ determined so as to satisfy some additional condition, we may assume that the input graph is actually d^2-regular (albeit is not necessarily simple). Furthermore, we shall assume that this graph is not bipartite. Both assumptions can be justified by augmenting the aforementioned construction of a 3-regular graph by adding $d^2 - 3$ self-loops to each vertex.

Prerequisites: Needless to say, the aforementioned transformation refers to the notion of an expander graph (as defined in §E.2.1.1). The transformation also relies on the *zig-zag product* defined in §E.2.2.2.

5.2.4.1 The basic approach

Recall that our goal is to transform G_1 into an expander. The transformation is gradual and consists of logarithmically many iterations, where in each iteration an adequate expansion parameter doubles while the graph becomes a constant factor larger and maintains the degree bound. The (expansion) parameter of interest is the gap between the relative second eigenvalue of the graph and 1 (see §E.2.1.1). A constant value of this parameter indicates that the graph is an expander. Initially, this parameter is lower-bounded by $1/O(n^2)$, where n is the size of the graph, and after logarithmically many iterations this parameter is lower-bounded by a constant (and the current graph is an expander).

The crux of the aforementioned gradual transformation is the transformation that takes place in each single iteration. This transformation combines the standard graph powering (to a constant power c) and the *zig-zag product* presented in §E.2.2.2. Specifically, for adequate positive integers d and c, we start with the d^2-regular graph $G_1 = (V_1, E_1)$, and go through a logarithmic number of iterations letting $G_{i+1} = G_i^c ⓩ G$ for $i = 1, ..., t-1$, where G is a fixed d-regular graph with d^{2c} vertices. That is, in each iteration, we raise the current graph (i.e., G_i) to the power c and combine the resulting graph with the fixed graph G using the zig-zag product. Thus, G_i is a d^2-regular graph with $d^{(i-1)\cdot 2c} \cdot |V_1|$ vertices, where this invariant is preserved by definition of the zig-zag product.

The analysis of the improvement in the expansion parameter, denoted $\delta_2(\cdot) \stackrel{\text{def}}{=} 1 - \bar{\lambda}_2(\cdot)$, relies on Eq. (E.9). Recall that Eq. (E.9) implies that if $\bar{\lambda}_2(G) < 1/2$ then $1 - \bar{\lambda}_2(G' ⓩ G) > (1 - \bar{\lambda}_2(G'))/3$. Thus, the fixed graph G is selected such that $\bar{\lambda}_2(G) < 1/2$, which requires a sufficiently large constant d. Thus, we have

$$\delta_2(G_{i+1}) \;=\; 1 - \bar{\lambda}_2(G_i^c ⓩ G) \;>\; \frac{1 - \bar{\lambda}_2(G_i^c)}{3} \;=\; \frac{1 - \bar{\lambda}_2(G_i)^c}{3}$$

whereas, for sufficiently large constant c, it holds that $1-\bar{\lambda}_2(G_i)^c > \max(6\cdot(1-\bar{\lambda}_2(G_i)),1/2)$. It follows that that $\delta_2(G_{i+1}) > \max(2\delta_2(G_i),1/6)$. Thus, setting $t = O(\log|V_1|)$ and using $\delta_2(G_1) = 1-\bar{\lambda}_2(G_1) = \Omega(|V_1|^{-2})$, we obtain $\delta_2(G_t) > 1/6$ as desired.

One detail of crucial importance is the ability to transform G_1 into G_t via a log-space computation. Indeed, the transformation of G_i to G_{i+1} can be performed in logarithmic space (see Exercise 5.10), but we need to compose a logarithmic number of such transformations. Unfortunately, the standard composition lemmas for space-bounded algorithms involve overhead that we cannot afford.[9] Still, taking a closer look at the transformation of G_i to G_{i+1}, one may note that it is highly structured and in some sense it can be implemented in constant space and supports a stronger composition result that incurs only a constant amount of storage per iteration. The resulting implementation (of the iterative transformation of G_1 to G_t) and the underlying formalism will be the subject of §5.2.4.2. (An alternative implementation, provided in [177], can be obtained by unraveling the composition.)

5.2.4.2 The actual implementation

The space-efficient implementation of the iterative transformation outlined in §5.2.4.1 is based on the observation that we do not need to explicitly construct the various graphs but merely provide "oracle access" to them. This observation is crucial when applied to the intermediate graphs; rather than constructing G_{i+1}, when given G_i as input, we show how to provide oracle access to G_{i+1} (i.e., answer "neighborhood queries" regarding G_{i+1}) when given oracle access to G_i (i.e., an oracle that answers neighborhood queries regarding G_i). That is, we view G_i and G_{i+1} (or rather their incidence lists) as functions (to be evaluated) rather than as strings (to be printed), and show how to reduce the task of finding neighbors in G_{i+1} (i.e., evaluating the "incidence function" at a given vertex) to the task of finding neighbors in G_i.

A clarifying discussion. Note that here we are referring to oracle machines that access a finite oracle, which represents a *finite variable object* (which in turn is an instance of some computational problem). Such a machine provides access to a complex object by using its access to a more basic object, which is represented by the oracle. Specifically, such a machine get an input, which is a "query" regarding the complex object (i.e, the object that the machine tries to emulate), and produce an output (which is the answer to the query). Analogously, these machines make queries, which are queries regarding another object (i.e., the one represented in the oracle), and obtain corresponding answers.[10]

[9] Needless to say, we cannot afford the naive composition (of Lemma 5.1), since it causes an overhead linear in the size of the intermediate output. As for the emulative composition (of Lemma 5.2), it sums up the space complexities of the composed algorithms (not to mention adding another logarithmic term), which would result in a log-squared bound on the space complexity.

[10] Indeed, the current setting (in which the oracle represents a *finite variable object*, which in turn is an instance of some computational problem) is different from the standard setting, where the oracle represents a *fixed computational problem*. Still the mechanism (and/or operations) of these two types of oracle machines is the same: They both get an input (which here is a

Like in §5.1.2.3, queries are made via a special write-only device and the answers are read from a corresponding read-only device, where the use of these devices is not charged in the space complexity. With these conventions in place, we claim that neighborhoods in the d^2-regular graph G_{i+1} can be computed by a constant-space oracle machine that is given oracle access to the d^2-regular graph G_i. That is, letting $g_i : V_i \times [d^2] \to V_i \times [d^2]$ (resp., $g_{i+1} : V_{i+1} \times [d^2] \to V_{i+1} \times [d^2]$) denote the edge rotation function[11] of G_i (resp., G_{i+1}), we have:

Claim 5.7 *There exists a constant-space oracle machine that evaluates g_{i+1} when given oracle access to g_i, where the state of the machine is counted in the space complexity.*

Proof Sketch: We first show that the two basic operation that underly the definition of G_{i+1} (i.e., powering and zig-zag product with a constant graph) can be performed in constant-space.

The edge rotation function of the square of a graph can be evaluated at $(v, \langle j_1, j_2 \rangle)$, using a constant amount of space, by evaluating the edge rotation function of the original graph twice. First, by making the query (v, j_1) we obtain the edge rotation of (v, j_1), denoted (u, k_1), and next, making the query (u, j_2), we obtain (w, k_2) and output $(w, \langle k_2, k_1 \rangle)$. We stress that we only use the temporary storage to record k_1, whereas u is directly copied from the oracle answer device to the oracle query device. Accounting also for a constant number of states needed for the various stages of the foregoing activity, we conclude that graph squaring can be performed in constant-space. The argument extends to the task of raising the graph to any constant power.

Turning to the zig-zag product (of G' with a fixed G), we note that the corresponding edge rotation function can be evaluated in constant-space (given oracle access to the edge rotation function of G'). This follows directly from Eq. (E.7), noting that the latter calls for a single evaluation of the edge rotation function of G' and two simple modifications that only depend on the constant-size graph G. Again, using the fact that it suffices to copy vertex names from the input (or oracle answer device) to the oracle query device (or output), we conclude that the aforementioned activity can be performed using constant space.

The argument extends to a sequential composition of a constant number of operations of the aforementioned type (i.e., graph squaring and zig-zag product with a constant graph). □

Recursive composition. Using Claim 5.7, we wish to obtain a log-space oracle machine that evaluates g_t by making oracle calls to g_1, where $t = O(\log |V_1|)$. Such

"query" regarding a variable object rather than an instance of a fixed computational problem), and produce an output (which here is the answer to the query rather than a "solution" for the given instance). Analogously, these machines make queries (which here are queries regarding another variable object rather than queries regarding another fixed computational problem), and obtain corresponding answers.

[11] Recall that the edge rotation function of a graph maps the pair (v, j) to the pair (u, k) if vertex u is the j^{th} neighbor of vertex v and v is the k^{th} neighbor of u (see §E.2.2.2).

an oracle machine will yield a log-space transformation of G_1 to G_t (by evaluating g_t at all possible values). It is tempting to hope that an adequate composition lemma, when applied to Claim 5.7, will yield the desired log-space oracle machine (reducing the evaluation of g_t to g_1). This is indeed the case, except that the adequate composition lemma is still to be developed (as we do next).

We first note that applying a naive composition (as in Lemma 5.1) amounts to an additive overhead of $O(\log |V_1|)$ *per each composition*. But we cannot afford more than an amortized constant additive overhead per composition. Applying the emulative composition (as in Lemma 5.2) amounts in multiplicative overhead (see §5.1.2.3), which is certainly unaffordable. The composition developed next is a variant of the naive composition, which is beneficial in the context of recursive calls. The basic idea is deviating from the paradigm that allocates separate input/output and query devices to each level in the recursion, and combining all these devices in a single ("global") device which will be used by all levels of the recursion. That is, rather than following the "structured programming" methodology of using (locally) designated space for passing information to the subroutine, we use the "bad programming" methodology of passing information through global variables. As usual, this notion is formulated by referring to the model of multi-tape Turing machine, but it can be formulated in any other reasonable model of computation.

Definition 5.8 (global-tape oracle machines): *A* global-tape oracle machine *is defined as an oracle machine* (cf. Definition 1.11)*, except that the input, output and oracle tapes are replaced by a single* global tape*. In addition, the machine has a constant number of work tapes, called the* local tapes*. The machine obtains its input from the global tape, writes each query on this very tape, obtains the corresponding answer from this tape, and writes its final output on this tape. The space complexity of such a machine is stated when referring separately to the use of the global tape and to the use of the local tapes.*

Clearly, any ordinary oracle machine can be converted into an equivalent global-tape oracle machine. The resulting machine uses a global tape of length at most $n + \ell + m$, where n denotes the length of the input, ℓ denote the length of the longest query or oracle answer, and m denotes the length of the output. However, combining the different tapes into one global tape seems to require holding separate pointers for each of the original tapes, which means that the local tape has to be used to store corresponding counters (in addition to the original work-tape). A key observation is that these counters can be avoided in the case that the original machine can be described as a sequence of transformations (of the input into the first query, and of the i^{th} answer to the $i+1^{\text{st}}$ query or the output), while maintaining auxiliary information on the work-tape. Indeed, the machine presented in the proof of Claim 5.7 has this form, and thus can be implemented by a global-tape oracle machine that uses a global-tape not longer than its input and a local-tape of constant length.

Claim 5.9 (Claim 5.7, revisited): *There exists a global-tape oracle machine that evaluates g_{i+1} when given oracle access to g_i, while using global tape of length $\log_2(d^2|V_{i+1}|)$ and a local tape of constant length.*

Proof Sketch: Following the proof of Claim 5.7, we merely indicate the exact use of the two tapes. For example, recall that the edge rotation function of the square of G_i is evaluated at $(v, \langle j_1, j_2 \rangle)$ by evaluating the edge rotation function of the original graph first at (v, j_1) and then at (u, j_2), where $(u, k_1) = g_i(v, j_1)$. This means the global-tape machine first reads $(v, \langle j_1, j_2 \rangle)$ from the global tape and replaces it by the query (v, j_1), while storing j_2 on the local tape. Thus, the machine merely deletes a constant number of bits from the global tape (and leaves its prefix intact). After invoking the oracle, the machine copies k_1 from the global tape (which currently holds (u, k_1)) to its local tape, and copies j_2 from its local tape to the global tape (such that it contains (u, j_2)). After invoking the oracle for the second time, the global tape contains $(w, k_2) = g_i(u, j_2)$, and the machine merely modifies it to $(w, \langle k_2, k_1 \rangle)$, which is the desired output.

Similarly, the edge rotation function of the zig-zag product of G' with a fixed G is evaluated at $(\langle u, i \rangle, \langle \alpha, \beta \rangle)$ by querying G' at $(u, E_\alpha(i))$ and outputting $(\langle v, E_\beta(j') \rangle, \langle \beta, \alpha \rangle)$, where (v, j') denotes the oracle answer (see Eq. (E.7)). This means that the global-tape oracle machine first copies α, β from the global-tape to the local-tape, transforms the contents of the global tape from $(\langle u, i \rangle, \langle \alpha, \beta \rangle)$ to $(u, E_\alpha(i))$, and makes an analogous transformation after the oracle is invoked. □

Composing global-tape oracle machines. In the proof of Claim 5.9, we implicitly used sequential composition of computations conducted by global-tape oracle machines.[12] In general, when sequentially composing such computations the global-tape and local-tape usage are the maximum among all composed computations; that is, the current formalism offers a tight bound on naive composition (as opposed to Lemma 5.1). Furthermore, global-tape oracle machines are beneficial in the context of recursive composition, as indicated by Lemma 5.10 (which relies on this model in a crucial way). The key observation is that all levels in the recursive composition may re-use the same global storage, and only the local storage gets added. Consequently we have the following composition lemma, where n denotes the length of the input to f_1 (and t may depend on n).

Lemma 5.10 (recursive composition in the global-tape model): *Suppose that, for every $i = 1, ..., t-1$, there exists a global-tape oracle machine that computes f_i by making oracle calls to f_{i+1} while using a global-tape of length $L(n)$ and a local-tape of length $l_i(n)$, which also accounts for the machine's state. Then f_1 can be computed by a standard oracle machine that makes calls to f_t and uses space $L(n) + \sum_{i=1}^{t-1} O(l_i(n))$.*

We shall apply this lemma with $f_i = g_{t+1-i}$, using the bounds $L(n) = O(n)$ and $l_i(n) = O(1)$ (as guaranteed by Claim 5.9), where in this application $n = \log_2(d^2|V_t|)$ is the length of the input to $f_1 = g_t$ and $t = O(\log |V_1|) = O(\log |V_t|)$.

Proof Sketch: We compute f_1 by allocating space for the emulation of the global-tape and the local-tapes of each level in the recursion. We emulate the recursive

[12] A similar composition took place in the proof of Claim 5.7, but here we assert a stronger feature of this specific computation.

computation by capitalizing on the fact that all recursive levels use the same global-tape (for making queries and receiving answers). In the actual recursion, each level may use the global-tape arbitrarily as long as when it returns control to the invoking machine the global-tape contains the right answer. Thus, the emulation may do the same, and emulate each recursive call by using the space allocated for the global-tape as well as the space designated for the local-tape of this level. □

Conclusion. Combining Claim 5.9 and Lemma 5.10, we conclude that the evaluation of $g_{O(\log |V_1|)}$ can be reduced to the evaluation of g_1 in space $O(\log |V_1|)$. Recalling that G_1 can be constructed in log-space (based on the input graph G_0), we infer that $G' = G_{O(\log |V_1|)}$ can be constructed in log-space. Theorem 5.6 follows by recalling that G' (which has constant degree and logarithmic diameter) can be tested for connectivity in log-space (see Exercise 5.9). Using a similar argument, we can test whether a given pair of vertices are connected in the input graph (see Exercise 5.11).

5.3 Non-Deterministic Space Complexity

The difference between space-complexity and time-complexity is quite striking in the context of non-deterministic computations. One aspect is the huge gap between the power of two formulation of non-deterministic space complexity (see Section 5.3.1), while in contrast the analogous formulations are equivalent in the context of time-complexity. Another aspect is the contrast between the results regarding (the standard model of) non-deterministic space-bounded computation (see Section 5.3.2) and the analogous questions in the context of time-complexity.

5.3.1 Two models

Recall that non-deterministic time-bounded computations were defined via two equivalent models. In the off-line model (underlying the definition of NP as a proof system (see Definition 2.5)) non-determinism is captured by reference to the existential choice of an auxiliary ("non-deterministic") input. In contrast, in the on-line model (underlying the traditional definition of NP (see Definition 2.7)) non-determinism is captured by reference to the non-deterministic choices of the machine itself. These models are equivalent because the latter on-line choices can be recorded (almost) for free (see the proof of Theorem 2.8). That is, while it is clear that the off-line model can emulate the on-line model (i.e., the off-line machine can emulate on-line choices by using its non-deterministic input), the emulation of the off-line model by the on-line model is enabled by the fact that an on-line machine may store (and re-use) a sequence of non-deterministic (on-line) choices.

The naive emulation of the off-line model on the on-line model is not possible in the context of space-bounded computation, because the number of non-deterministic choices is typically much larger than the space-bound. The models become equivalent only if the off-line model is restricted to access its non-

deterministic input in a uni-directional manner. Let us formulate the two models and consider them more closely.

In the standard model, called the on-line model, the machine makes non-deterministic choices "on the fly" (or alternatively reads a non-deterministic input from a special read-only tape *that can be read only in a uni-directional way*). Thus, if the machine needs to refer to such a non-deterministic choice at a latter stage in its computation, then it must store the choice on its storage device (and be charged for it). In contrast, in the so-called off-line model the non-deterministic choices (or the bits of the non-deterministic input) are read from a read-only device (or tape) *that can be scanned in both directions like the main input.*

We denote by $\text{NSPACE}_{\text{on-line}}(s)$ (resp., $\text{NSPACE}_{\text{off-line}}(s)$) the class of sets that are acceptable by an on-line (resp., off-line) non-deterministic machine having space complexity s. We stress that, as in Definition 2.7, the set accepted by a non-deterministic machine M is the set of strings x such that there exists a computation of $M(x)$ that is accepting. Clearly, $\text{NSPACE}_{\text{on-line}}(s) \subseteq \text{NSPACE}_{\text{off-line}}(s)$. On the other hand, not only that $\text{NSPACE}_{\text{on-line}}(s) \neq \text{NSPACE}_{\text{off-line}}(s)$ but rather $\text{NSPACE}_{\text{on-line}}(s) = \text{NSPACE}_{\text{off-line}}(\Theta(\log s))$, provided that s is at least linear. For details, see Exercise 5.14.

Before proceeding any further, let us justify the focus on the on-line model in the rest of this section. Indeed, the off-line model fits better the motivations to $\mathcal{NP}$ (as presented in Section 2.1.2), but the on-line model seems more adequate for the study of non-deterministic in the context of space complexity. One reason is that an off-line non-deterministic input can be used to code computations (see Exercise 5.14), and in a sense allows to "cheat" with respect to the "actual" space complexity of the computation. This is reflected in the fact that the off-line model can emulate the on-line model while using space that is logarithmic in the space used by the on-line model. A related phenomenon is that $\text{NSPACE}_{\text{off-line}}(s)$ is only known to be contained in $\text{DTIME}(2^{2^s})$, whereas $\text{NSPACE}_{\text{on-line}}(s) \subseteq \text{DTIME}(2^s)$. This fact motivates the study of $\mathcal{NL} = \text{NSPACE}_{\text{on-line}}(\texttt{log})$, as a study of a (natural) sub-class of $\mathcal{P}$. Indeed, the various results regarding $\mathcal{NL}$ justify its study in retrospect.

In light of the foregoing, we adopt the standard conventions and let $\text{NSPACE}(s) = \text{NSPACE}_{\text{on-line}}(s)$. Our main focus will be the study of $\mathcal{NL} = \text{NSPACE}(\texttt{log})$.

5.3.2 NL and directed connectivity

This section is devoted to the study of $\mathcal{NL}$, which we view as the non-deterministic analogue of $\mathcal{L}$. Specifically, $\mathcal{NL} = \cup_c \text{NSPACE}(\ell_c)$, where $\ell_c(n) = c \log_2 n$ (and the definitional issues pertaining $\text{NSPACE} = \text{NSPACE}_{\text{on-line}}$ are discussed in Section 5.3.1).

We first note that the proof of Theorem 5.3 can be easily extended to the (on-line) non-deterministic context. The reason being that moving from the deterministic model to the current model does not affect the number of instantaneous configurations (as defined in the proof of Theorem 5.3), whereas this number bounds the time complexity. Thus, $\mathcal{NL} \subseteq \mathcal{P}$.

The following problem, called directed connectivity (st-CONN), captures the essence of non-deterministic log-space computations (and, in particular, is complete for $\mathcal{NL}$ under log-space reductions). The input to st-CONN consists of a directed graph $G = (V, E)$ and a pair of vertices (s, t), and the task is to determine whether there exists a directed path from s to t (in G).[13] Indeed, the study of $\mathcal{NL}$ is often conducted via st-CONN. For example, note that $\mathcal{NL} \subseteq \mathcal{P}$ follows easily from the fact that st-CONN is in $\mathcal{P}$ (and the fact that $\mathcal{NL}$ is log-space reducible to st-CONN).

5.3.2.1 Completeness and beyond

Clearly, st-CONN is in $\mathcal{NL}$ (see Exercise 5.15). The $\mathcal{NL}$-completeness of st-CONN under log-space reductions follows by noting that the computation of any non-deterministic space-bounded machine yields a directed graph in which vertices correspond to possible configurations and edges represent the "successive" relation of the computation. In particular, for log-space computations the graph has polynomial size, but in general the relevant graph is strongly explicit (in a natural sense; see Exercise 5.16).

Theorem 5.11 *Every problem in $\mathcal{NL}$ is log-space reducible to* st-CONN (via a many-to-one reduction).

Proof Sketch: Fixing a non-deterministic (on-line) machine M and an input x, we consider the following directed graph $G_x = (V_x, E_x)$. The vertices of V_x are possible instantaneous configurations of $M(x)$, where each configuration consists of the contents of the work-tape (and the machine's finite state), the machine's location on it, and the machine's location on the input. The directed edges represent possible single moves in such a computation. We stress that such a move depends on the machine M as well as on the (single) bit of x that resides in the location specified by the first configuration (i.e., the configuration corresponding to the start-point of the potential edge).[14] Note that, given x, the graph G_x can be constructed in log-space (by scanning all pairs of vertices and outputting only the pairs that are valid edges (which, in turn, can be tested in constant-space)).

By definition, the graph G_x represents the possible computations of M on input x. In particular, there exists an accepting computation of M on input x if and only if there exists a directed path, in G_x, starting at the vertex s that corresponds to the initial configuration and ending at the vertex t that corresponds to a canonical accepting configuration. Thus, $x \in S$ if and only if (G_x, s, t) is a yes-instance of st-CONN. □

[13] See Appendix G.1 for basic graph theoretic terminology. We note that, here (and in the sequel), s stands for *start* and t stands for *terminate*.

[14] Thus, the actual input x only affects the set of edges of G_x (whereas the set of vertices is only affected by $|x|$). A related construction is obtained by incorporating in the configuration also the (single) bit of x that resides in the machine's location on the input. In the latter case, x itself also affects V_x.

Reflection. We believe that the proof of Theorem 5.11 (see also Exercise 5.16) justifies saying that `st-CONN` captures the essence of non-deterministic space-bounded computations. Note that this (intuitive and imformal) statement goes beyond saying that `st-CONN` is $\mathcal{NL}$-complete under log-space reductions.

We note the discrepancy between the status of undirected connectivity (see Theorem 5.6 and Exercise 5.11) and directed connectivity (see Theorem 5.11). In this context it is worthwhile to note that determining the existence of relatively short paths (rather than arbitrary paths) in undirected (or directed) graphs is also $\mathcal{NL}$-complete under log-space reductions; see Exercise 5.18.

5.3.2.2 Relating NSPACE to DSPACE

Recall that in the context of time complexity, converting non-deterministic computation to deterministic computation is known only when allowing an exponential blow-up in the complexity. In contrast, space complexity allows such a conversion at the cost of a polynomial blow-up in the complexity.

Theorem 5.12 (Non-deterministic versus deterministic space): *For any space-constructible $s : \mathbb{N} \to \mathbb{N}$ that is at least logarithmic, it holds that* $\text{NSPACE}(s) \subseteq \text{DSPACE}(O(s^2))$.

In particular, non-deterministic polynomial-space is contained in deterministic polynomial-space (and non-deterministic poly-logarithmic space is contained in deterministic poly-logarithmic space).

Proof Sketch: We focus on the special case of $\mathcal{NL}$ and the argument extends easily to the general case. Alternatively, the general statement can be derived from the special case by using a suitable upwards-translation lemma (see, e.g., [116, Sec. 12.5]). The special case boils down to presenting a log-square space algorithm for deciding directed connectivity.

The basic idea is that checking whether or not there is a path of length at most ℓ from u to v in G, reduces (in log-space) to checking whether there is an intermediate vertex w such that there is a path of length at most $\lceil \ell/2 \rceil$ from u to w and a path of length at most $\lfloor \ell/2 \rfloor$ from w to v. Let $\phi_G(u,v,\ell) \stackrel{\text{def}}{=} 1$ if there is a path of length at most ℓ from u to v in G and $\phi_G(u,v,\ell) \stackrel{\text{def}}{=} 0$ otherwise. Thus, $\phi_G(u,v,\ell)$ can be decided recursively by scanning all vertices w in G, and checking for each w whether both $\phi_G(u,w,\lceil \ell/2 \rceil) = 1$ and $\phi_G(w,v,\lfloor \ell/2 \rfloor) = 1$ hold. We stress that all these computations re-use the same space, and we need only store one additional bit representing the results of all prior computations.

Turning back to the origianl problem, note that, given a directed graph $G = (V,E)$ and a pair of vertices (s,t), we should merely compute $\phi_G(s,t,|V|)$. This is done by invoking a recursive procedure that computes $\phi_G(u,v,\ell)$ by scanning all vertices in G, and computing for each vertex w the values of $\phi_G(u,w,\lceil \ell/2 \rceil)$ and $\phi_G(w,v,\lfloor \ell/2 \rfloor)$. We return the value 1 if and only if for some w it holds that $\phi_G(u,w,\lceil \ell/2 \rceil) = \phi_G(w,v,\lfloor \ell/2 \rfloor) = 1$ (see Figure 5.2). Needless to say, $\phi_G(u,v,1)$ and $\phi_G(u,v,0)$ can be decided easily in logarithmic space.

Recursive computation of $\phi_G(u, v, 2\ell)$, for $\ell \geq 1$.

```
For w = 1, ..., |V| do begin          (storing the vertex name)
    Compute σ ← φ_G(u, w, ℓ)          (by a recursive call)
    Compute σ ← σ ∧ φ_G(w, v, ℓ)      (by a second recursive call)
    If σ = 1 then return 1.           (success: an intermediate vertex was found)
End                                   (of scan).
return 0.          (reached only if the scan was completed without success).
```

Figure 5.2: The recursive procedure in $\mathcal{NL} \subseteq \text{DSPACE}(O(\log^2))$.

The amount of space taken by each level of the recursion is $\log_2 |V|$ (for storing the current value of w), and the number of levels is $\log_2 |V|$. We stress that when computing $\phi_G(u, v, \ell)$, we make many recursive calls, but all these calls re-use the same work space. That is, when we compute $\phi_G(u, w, \lceil \ell/2 \rceil)$ we re-use the space that was used for computing $\phi_G(u, w', \lfloor \ell/2 \rfloor)$ for the previous w', and we re-use the same space when we compute $\phi_G(w, v, \lfloor \ell/2 \rfloor)$. The theorem follows. □

Digest. The proof of Theorem 5.12 relies on two main observations. The first observation is that an existential claim can be verifying by scanning all possible values in the relevant domain, which in terms of space complexity has a cost that is logarithmic in the size of the domain. The second observation is that a disjunction (resp., conjunction) of two Boolean conditions can be verifier using space $s + O(1)$, where s is the space complexity of verifying a single condition. This follows by applying naive composition (i.e., Lemma 5.1). The proof of Theorem 5.12 is facilitated by the fact that we may consider a concrete and simple computational problem such as `st-CONN`. Nevertheless, the same ideas can be applied directly to $\mathcal{NL}$ (or ant NSPACE class).

The simple formulation of `st-CONN` facilitates placing $\mathcal{NL}$ in complexity classes such as $\mathcal{NC}^2$. All that is needed is observing that `st-CONN` can be solved by raising the adequate matrix (i.e., the adjacency matrix of the graph augmented with 1-entries on the diagonal) to the adequate power (i.e., its dimension). Squaring a matrix can be done by a uniform family of logarithmic depth bounded fan-in circuits (i.e., in NC1), and by repeated squaring the n^{th} power of an n-by-n matrix can be computed by a uniform family of bounded fan-in circuits of polynomial size and depth $O(\log^2 n)$; thus, $\mathcal{NL} \subseteq \mathcal{NC}^2$.

5.3.2.3 Complementation or NL=coNL

Recall that (natural) non-deterministic time-complexity classes are not known to be closed under complementation. Furthermore, it is widely believed that $\mathcal{NP} \neq \text{co}\mathcal{NP}$. In contrast, (natural) non-deterministic space-complexity classes are closed under complementation, as captured by the result $\mathcal{NL} = \text{co}\mathcal{NL}$, where

$\text{co}\mathcal{NL} \stackrel{\text{def}}{=} \{\{0,1\}^* \setminus S : S \in \mathcal{NL}\}$. Before proving this result, let us take a closer look at the problem it resolves (which we rephrase as $\mathcal{NL}$ versus $\mathcal{NL} \cap \text{co}\mathcal{NL}$).

Recall that a set S is in $\mathcal{NL}$ if there exists a non-deterministic log-space machine M that accepts S, and that the acceptance condition of non-deterministic machines is asymmetric in nature. That is, $x \in S$ implies the *existence* of an accepting computation of M on input x, whereas $x \notin S$ implies that *all* computations of M on input x are non-accepting. Thus, the existence of a accepting computation of M on input x is an absolute indication for $x \in S$, but the existence of a rejecting computation of M on input x is not an absolute indication for $x \notin S$. We note that, for $S \in \mathcal{NL} \cap \text{co}\mathcal{NL}$, there exist absolute indications both for $x \in S$ and for $x \notin S$ (or, equivalently for $x \in \overline{S} \stackrel{\text{def}}{=} \{0,1\}^* \setminus S$), where each of the two types of indication is provided by a different machine (i.e., the one accepting S or the one accepting $\overline{S}$). Combining both machines, we obtain a single non-deterministic machine that, for every input, sometimes outputs the correct answer and always outputs either the correct answer or a special ("don't know") symbol. This yields the following definition, which refers to Boolean functions as a special case.

Definition 5.13 (non-deterministic computation of functions): *We say that a* non-deterministic machine M computes the function $f : \{0,1\}^* \to \{0,1\}^*$ *if for every $x \in \{0,1\}^*$ the following two conditions hold.*

1. *Every computation of M on input x yields an output in $\{f(x), \perp\}$, where $\perp \notin \{0,1\}^*$ is a special symbol* (indicating "don't know").
2. *There exists a computation of M on input x that yields the output $f(x)$.*

Note that $S \in \mathcal{NL} \cap \text{co}\mathcal{NL}$ if and only if there exists a non-deterministic log-space machine that computes χ_S, where $\chi_S(x) = 1$ if $x \in S$ and $\chi_S(x) = 0$ otherwise (see Exercise 5.20). It follows that $\mathcal{NL} = \text{co}\mathcal{NL}$ if and only if for every $S \in \mathcal{NL}$ there exists a non-deterministic log-space machine that computes χ_S.

Theorem 5.14 ($\mathcal{NL} = \text{co}\mathcal{NL}$): *For every $S \in \mathcal{NL}$ there exists a non-deterministic log-space machine that computes χ_S.*

As in the case of Theorem 5.12, the result extends to any space-constructible $s : \mathbb{N} \to \mathbb{N}$ that is at least logarithmic; that is, for such s and every $S \in \text{NSPACE}(s)$, it holds that $\{0,1\}^* \setminus S \in \text{NSPACE}(O(s))$. This extension can be proved either by generalizing the following proof or by using an adequate upwards-translation lemma.

Proof Sketch: It suffices to present a non-deterministic (on-line) log-space machine that computes the characteristic function of `st-CONN`, denoted χ (i.e., $\chi(G,s,t) = 1$ if there is a directed path from s to t in G and $\chi(G,s,t) = 0$ otherwise).

We first show that the computation of χ is log-space reducible (by two queries)[15] to determining the number of vertices that are reachable from a given vertex in

[15] We stress the fact that two queries are used in the reduction, because this avoids the difficulties (discussed in §5.1.2.3) regarding emulative composition for general space-bounded reduction. Alternatively, we may use a version of the naive composition, while relying on the fact that the oracle queries are "highly related" to the input and that the answers have logarithmic length. For details, see Exercises 5.23 and 5.24.

a given graph. On input (G, s, t), the reduction computes the number of vertices that are reachable from s in the graph G and compares it to the number of vertices reachable from s in the graph obtained by deleting t from G. (An alternative reduction that uses a single query is presented in Exercise 5.22.) Note that if computing f is log-space reducible by a constant number of queries to computing some function g and there exists a non-deterministic log-space machine that computes g, then there exists a non-deterministic log-space machine that computes f (see Exercise 5.23). Thus, we focus on providing a non-deterministic log-space machine that compute the number of vertices that are reachable from a given vertex in a given graph.

Fixing an n-vertex graph $G = (V, E)$ and a vertex v, we consider the set of vertices that are reachable from v by a path of length at most i. We denote this set by R_i, and observe that $R_0 = \{v\}$ and that for every $i = 1, 2, ...$, it holds that

$$R_i = R_{i-1} \cup \{u : \exists w \in R_{i-1} \text{ s.t. } (w, u) \in E\} \tag{5.1}$$

Our aim is to compute $|R_n|$. This will be done in n iterations such that at the i^{th} iteration we compute $|R_i|$. When computing $|R_i|$ we rely on the fact that $|R_{i-1}|$ is known to us, which means that we shall store $|R_{i-1}|$ in memory. We stress that we discard $|R_{i-1}|$ from memory as soon as we complete the computation of $|R_i|$, which we store instead. Thus, at each iteration i, our record of past iterations only contains $|R_{i-1}|$.

Computing $|R_i|$. Given $|R_{i-1}|$, we non-deterministically compute $|R_i|$ by making a guess (for $|R_i|$), denoted g, and verifying its correctness as follows:

1. We verify that $|R_i| \geq g$ in a straightforward manner. That is, scanning V in some canonical order, we verify for g vertices that they are each in R_i. That is, during the scan, we select non-deterministically g vertices, and for each selected vertex w we verify that w is reachable from v by a path of length at most i, where this verification is performed by just guessing and verifying an adequate path (see Exercise 5.15).

   We use $\log_2 n$ bits to store the currently scanned vertex (i.e., w), and another $O(\log n)$ bits for implementing the verification of the existence of a path of length at most i from v to w.

2. The verification of the condition $|R_i| \leq g$ (equivalently, $|V \setminus R_i| \geq n - g$) is the interesting part of the procedure. Indeed, as we saw, demonstrating membership in R_i is easy, but here we wish to demonstrate non-membership in R_i. We so so by relying on the fact that we know $|R_{i-1}|$, which allows for a non-deterministic enumeration of R_{i-1} itself, which in turn allows for proofs of non-membership in R_i (via the use of Eq. (5.1)). Details follows (and an even more structured description is provided in Figure 5.3).

   Scanning V (again), we verify for $n-g$ (guessed) vertices that they are *not* in R_i (i.e., are *not* reachable from v by paths of length at most i). By Eq. (5.1), verifying that $u \notin R_i$ amounts to proving that for every $w \in R_{i-1}$, it holds that $u \neq w$ and $(w, u) \notin E$. As hinted, the knowledge of $|R_{i-1}|$ allows for the

enumeration of R_{i-1}, and thus we merely check the aforementioned condition on each vertex in R_{i-1}. Thus, verifying that $u \notin R_i$ is done as follows.

(a) We scan V guessing $|R_{i-1}|$ vertices that are in R_{i-1}, and verify each such guess in the straightforward manner (i.e., as in Step 1).[16]

(b) For each $w \in R_{i-1}$ that was guessed and verified in Step 2a, we verify that both $u \neq w$ and $(w,u) \notin E$.

By Eq. (5.1), if u passes the foregoing verification then indeed $u \notin R_i$.

Note that we use $\log_2 n$ bits to store the vertex u, another $\log_2 n$ bits to count the number of vertices that are verified to be in R_{i-1}, another $\log_2 n$ bits to store such a vertex w, and another $O(\log n)$ bits for verifying that $w \in R_{i-1}$ (as in Step 1).

If any of the foregoing verifications fails, then the procedure halts outputting the "don't know" symbol $\perp$. Otherwise, it outputs g.

It can be verified that, when given the correct value of $|R_{i-1}|$, the foregoing non-deterministic log-space procedure computes the value of $|R_i|$. That is, if all verifications are satisfied then it must hold that $g = |R_i|$, and if $g = |R_i|$ then there are adequate non-deterministic choices that satisfy all verifications.

Recall that R_n is computed iteratively, starting with $|R_0| = 1$ and computing $|R_i|$ based on $|R_{i-1}|$. Each iteration $i = 1, ..., n$ is non-deterministic, and is either completed with the correct value of $|R_i|$ (at which point $|R_{i-1}|$ is discarded) or halts in failure (in which case we halt the entire process and output $\perp$). This yields a non-deterministic log-space machine for computing $|R_n|$, and the theorem follows. □

Digest. Step 2 is the heart of the proof (of Theorem 5.14). In this step a non-deterministic procedure is used to verify non-membership in an NL-type set. Indeed, verifying membership in NL or NP-type sets is the archetypical task of non-deterministic procedures (i.e., they are defined so to fit these tasks). The verification of non-membership is enabled, in the current context, by an iterative (non-deterministic) process that consumes an admissible amount of resources (i.e., a logarithmic amount of space).

5.3.3 Discussion

The current section may be viewed as a study of the "power of non-determinism in computation" (which is a somewhat contradictory term). Recall that we view non-deterministic processes as fictitious abstractions aimed at capturing fundamental phenomena such as verification of proofs (cf., Section 2.1.4). Since these fictitious abstractions are fundamental in the context of time-complexity, we may hope to

[16] Note that implicit in Step 2a is a non-deterministic procedure that computes the mapping $(G, v, i, |R_{i-1}|) \to R_{i-1}$, where R_{i-1} denotes the set of vertices that are reachable in G by a path of length at most i from v.

Given $|R_{i-1}|$ and a guess g, the claim $g \geq |R_i|$ is verified as follows.

Set $c \leftarrow 0$. (*initializing the main counter*)
For $u = 1, ..., n$ do begin (*the main scan*)
  Guess whether or not $u \in R_i$.
  For a negative guess (i.e., $u \notin R_i$), do begin
  (*Verify that $u \notin R_i$ via Eq.* (5.1).)
    Set $c' \leftarrow 0$. (*initializing a secondary counter*)
    For $w = 1, ..., n$ do begin (*the secondary scan*)
      Guess whether or not $w \in R_{i-1}$.
      For a positive guess (i.e., $w \in R_{i-1}$), do begin
        Verify that $w \in R_{i-1}$ (as in Step 1).
        Verify that $u \neq w$ and $(w, u) \notin E$.
        If some verification failed
        then halt with output $\perp$ otherwise increment c'.
      End (*of handling a positive guess for $w \in R_{i-1}$*).
    End (*of secondary scan*). (*c' vertices in R_{i-1} were checked*)
    If $c' < |R_{i-1}|$ then halt with output $\perp$.
    Otherwise ($c' = |R_{i-1}|$), increment c. (*u verified to be outside of R_i*)
  End (*of handling a negative guess for $u \notin R_i$*).
End (*of main scan*). (*c vertices were shown outside of R_i*)
If $c < n - g$ then halt with output $\perp$.
Otherwise $n - |R_i| \geq c \geq n - g$ is verified.

Figure 5.3: The main step in proving $\mathcal{NL} = \text{co}\mathcal{NL}$.

gain some understanding by a comparative study (conducted in the context of space-complexity). Furthermore, we may discover that non-deterministic space bounded machines give rise to interesting computational phenomena.

The aforementioned hopes seems to come true in the current section. For example, the fact that $\mathcal{NL} = \text{co}\mathcal{NL}$, while the common conjecture is that $\mathcal{NP} \neq \text{co}\mathcal{NP}$, indicates that the latter conjecture is less generic than sometimes stated. It is not that an existential quantifier cannot be "feasibly replaced" by a universal quantifier, but rather the feasibility of such a replacement depends very much of the type of notion of feasibility. Turning to the other type of benefits, we learned that `st-CONN` can be reduced in log-space to `st-unCONN` (i.e., the set of graphs in which there is no directed path between the two designated vertices; see Exercise 5.21).

Still, one may ask what does the class $\mathcal{NL}$ actually represent (beyond `st-CONN`, which is more than merely a complete problem for this class; see §5.3.2.1). Turning back to Section 5.3.1, we recall that the class $\text{NSPACE}_{\text{off-line}}$ captures the straightforward notion of space-bounded verification. In this model (called the off-line model), the alleged proof is written on a special device (similarly to the claim being proved by it), which is being read freely. In contrast, underlying the alternative

class $\text{NSPACE}_{\text{on-line}}$ is a notion of proofs that are verified by reading them sequentially (rather than scanning them back and forth). In this case, if the verification procedure needs to relate to the currently read part of the proof in the future, then it must store the relevant part (and is charged for this storage). Thus, the on-line model underlying $\text{NSPACE}_{\text{on-line}}$ refers to the standard process of reading proofs in a sequential manner and making side notes for future verification, rather than scanning them back and forth all the time. In retrospect (see discussion in Section 5.3.1), our feeling is that the off-line model allows for an unfair accounting of space as well as unintendedly long proofs.

5.4 PSPACE and Games

As stated up-front, we will rarely treat computational problems that require less than logarithmic space. On the other hand, we will rarely treat computational problems that require more than polynomial space. The class of decision problems that are solvable in polynomial-space is denoted $\mathcal{PSPACE} \stackrel{\text{def}}{=} \cup_c \text{DSPACE}(p_c)$, where $p_c(n) = n^c$.

To get a sense of the power of $\mathcal{PSPACE}$, we observe that $\mathcal{PH} \subseteq \mathcal{PSPACE}$; for example, a polynomial-space algorithm can easily verify the quantified condition underlying Definition 3.8. In fact, such an algorithm can handle an unbounded number of alternating quantifiers (see Theorem 5.15). On the other hand, by Theorem 5.3, $\mathcal{PSPACE} \subseteq \mathcal{EXP}$, where $\mathcal{EXP} = \cup_c \text{DTIME}(2^{p_c})$ for $p_c(n) = n^c$.

The class $\mathcal{PSPACE}$ can be interpreted as capturing the complexity of determining the winner in certain *efficient two-party game*; specifically, the very games considered in Section 3.2.1 (modulo Footnote 2 there). Recall that we refer to two-party games that satisfy the following three conditions:

1. The parties alternate in taking moves that effect the game's (global) position, where each move has a description length that is bounded by a polynomial in the length of the *initial* position.

2. The current position is updated based on the previous position and the current party's move. This updating can be performed in time that is polynomial in the length of the *initial* position. (Equivalently, we may require a polynomial-time updating procedure and postulate that the length of the current position be bounded by a polynomial in the length of the *initial* position.)

3. The winner in each position can be determined in polynomial-time.

A set $S \in \mathcal{PSPACE}$ can be viewed as the set of initial positions (in a suitable game) for which the first party has a winning strategy *consisting of a polynomial number of moves*. Specifically, $x \in S$ if starting at the initial position x, there exists move y_1 for the first party, such that for every response move y_2 of the second party, there exists move y_3 for the first party, etc, such that after $\text{poly}(|x|)$ many moves the parties reach a position in which the first party wins, where the final position

as well as which party wins in it can be computed in polynomial-time (from the initial position x and the sequence of moves $y_1, y_2, ...$). The fact that every set in $\mathcal{PSPACE}$ corresponds to such a game follows from Theorem 5.15, which refers to the satisfiability of quantified Boolean formulae (QBF).[17]

Theorem 5.15 QBF *is complete for* $\mathcal{PSPACE}$ *under polynomial-time many-to-one reductions.*

Proof: As note before, QBF is solvable by a polynomial-space algorithm that just evaluates the quantified formula. Specifically, consider a recursive procedure that eliminates a Boolean quantifier by evaluating the value of the two residual formulae, and note that the space used in the first (recursive) evaluation can be re-used in the second evaluation. (Alternatively, consider a DFS-type procedure as in Section 5.1.3.) Note that the space used is linear in the depth of the recursion, which in turn is linear in the length of the input formula.

We now turn to show that any set $S \in \mathcal{PSPACE}$ is many-to-one reducible to QBF. The proof is similar to the proof of Theorem 5.12, except that here we work with an implicit graph (rather than with an explicitly given graph). Specifically, we refer to the directed graph of configuration (of the algorithm A deciding membership in S) as defined in Exercise 5.16. Actually, here we use a different notion of a configuration that *includes also the input*. That is, in the rest of this proof, *a configuration consists of the contents of all storage devices of the algorithm* (including the input device) as well as the location of the algorithm on each device.

Recall that for a graph G, we defined $\phi_G(u, v, \ell) = 1$ if there is a path of length at most ℓ from u to v in G (and $\phi_G(u, v, \ell) = 0$ otherwise). We need to determine $\phi_G(s, t, 2^m)$ for s that encodes the initial configuration of $A(x)$ and t that encodes the canonical accepting configuration, where G depends on the algorithm A and $m = \text{poly}(|x|)$ is such that $A(x)$ uses at most m space and runs for at most 2^m steps. By the specific definition of a configuration (which contains all relevant information including the input x), the value of $\phi_G(u, v, 1)$ can be determined easily based solely on the fixed algorithm A (i.e., either $u = v$ or v is a configuration following u). Recall that $\phi_G(u, v, 2\ell) = 1$ if and only if there exists a configuration w such that both $\phi_G(u, w, \ell) = 1$ and $\phi_G(w, v, \ell) = 1$ hold. Thus, we obtain the recursion

$$\phi_G(u, v, 2\ell) = \exists w \in \{0,1\}^m \phi_G(u, w, \ell) \wedge \phi_G(w, v, \ell), \tag{5.2}$$

where the bottom of the recursion (i.e., $\phi_G(u, v, 1)$) is a simple propositional formula (see foregoing discussion). The problem with Eq. (5.2) is that the expression for $\phi_G(\cdot, \cdot, 2\ell)$ involves two occurrences of $\phi_G(\cdot, \cdot, \ell)$, which doubles the length of the recursively constructed formula (yielding an exponential blow-up).

Our aim is to express $\phi_G(\cdot, \cdot, 2\ell)$ *while using* $\phi_G(\cdot, \cdot, \ell)$ *only once*. The extra restriction, which prevents an exponential blow-up, corresponds to the *re-using of space* in the (two evaluations of $\phi_G(\cdot, \cdot, \ell)$ that take place in the) computation of $\phi_G(u, v, 2\ell)$. The main idea is replacing the condition $\phi_G(u, w, \ell) = \phi_G(w, v, \ell) = 1$ by the condition $\forall (u'v') \in \{(u, w), (w, v)\}\, \phi_G(u', v', \ell)$. Next, we reformulate the

[17] See Appendix G.2.

"non-standard quantifier" (which ranges over a specific pair of strings) by using additional quantifiers as well as some simple boolean conditions. That is, $\forall(u'v') \in \{(u,w),(w,v)\}$ is replaced by $\forall\sigma \in \{0,1\}\exists u', v' \in \{0,1\}^m$ and the auxiliary condition

$$[(\sigma=0) \Rightarrow (u'=u \wedge v'=w)] \wedge [(\sigma=1) \Rightarrow (u'=w \wedge v'=v)]. \tag{5.3}$$

Thus, $\phi_G(u,v,2\ell)$ holds if and only if there exist w such that for every σ there exists (u',v') such that both Eq. (5.3) and $\phi_G(u',v',\ell)$ hold. Note that the length of this expression for $\phi_G(\cdot,\cdot,2\ell)$ equals the length of $\phi_G(\cdot,\cdot,\ell)$ plus an additive overhead term of $O(m)$. Thus, using a recursive construction, the length of the formula grows only linearly in the number of recursion steps.

The reduction itself maps an instance x (of S) to the quantified Boolean formula $\Phi(s_x,t,2^m)$, where s_x denotes the initial configuration of $A(x)$, (t and $m=\mathrm{poly}(|x|)$ are as above), and Φ is recursively defined as follows

$$\begin{aligned}\Phi(u,v,2\ell) \stackrel{\mathrm{def}}{=} \quad & \exists w \in \{0,1\}^m \, \forall\sigma \in \{0,1\}\exists u', v' \in \{0,1\}^m \\ & [(\sigma=0) \Rightarrow (u'=u \wedge v'=w)] \\ & \wedge [(\sigma=1) \Rightarrow (u'=w \wedge v'=v)] \\ & \wedge \Phi(u',v',\ell)\end{aligned} \tag{5.4}$$

with $\Phi(u,v,1)=1$ if and only if either $u=v$ or there is an edge from u to v. Note that $\Phi(u,v,1)$ is a propositional formula with Boolean variables representing the bits of u and v such that $\Phi(u,v,1)$ is satisfies if and only if either $u=v$ or v is a configuration that follows the configuration u in a computation of A. On the other hand, note that $\Phi(s_x,t,2^m)$ is a quantified formula in which the quantified variables are not shown in the notation.

We stress that the mapping of x to $\Phi(s_x,t,2^m)$ can be computed in polynomial-time. Firstly, note that the propositional formula $\Phi(u,v,1)$, having Boolean variables representing the bits of u and v, expresses extremely simple conditions and can certainly be constructed in polynomial-time (i.e., polynomial in the number of Boolean variables, which in turn equals $2m$). Next note that, given $\Phi(u,v,\ell)$, which (for $\ell>1$) contains quantified variables that are not shown in the notation, we can construct $\Phi(u,v,2\ell)$ by merely replacing variables names and adding quantifiers and Boolean conditions as in the recursive definition of Eq. (5.4). This is certainly doable in polynomial-time. Lastly, note that the construction of $\Phi(s_x,t,2^m)$ depends mainly on the length of x, where x itself only affects s_x (and does so in a trivial manner). Recalling that $m=\mathrm{poly}(|x|)$, it follows that everything is computable in time polynomial in $|x|$. Thus, given x, the formula $\Phi(s_x,t,2^m)$ can be constructed in polynomial-time.

Finally, note that $x \in S$ if and only if the formula $\Phi(s_x,t,2^m)$ is satisfiable. The theorem follows. ■

Other $\mathcal{PSPACE}$-complete problems. Several generalizations of natural games give rise to $\mathcal{PSPACE}$-complete problems (see [194, Sec. 8.3]). This further justifies the title of the current section.

Chapter Notes

The material presented in the current chapter is based on a mix of "classical" results (proven in the 1970's if not earlier) and "modern" results (proven in the late 1980's and even later). What we wish to emphasize is the time gap between the formulation of some questions and their resolution. Details follow.

We first mention the "classical" results. These include the $\mathcal{NL}$-completeness of `st-CONN`, the emulation of non-deterministic space-bounded machines by deterministic space-bounded machines (i.e., Theorem 5.12 due to Savitch [184]), the $\mathcal{PSPACE}$-completeness of `QBF`, and the connections between circuit depth and space complexity (see Section 5.1.3 and Exercise 5.7 due to Borodin [45]).

Before turning to the "modern" results, we mention that some people tend to be discouraged by the impression that "decades of research have failed to answer any of the famous open problems of complexity theory." In addition to the fact that substantial progress towards the understanding of many fundamental issues has been achieved, people tend to forget that some famous open problems were actually resolved. Two such examples were presented in this chapter.

The question of whether $\mathcal{NL} = \text{co}\mathcal{NL}$ was a famous open problem for almost two decades. Furthermore, this question is related to an even older open problem dating to the early days of research in the area of formal languages (i.e., to the 1950's).[18] This open problem was resolved in 1988 by Immerman [118] and Szelepcsenyi [205], who (independently) proved Theorem 5.14 (i.e., $\mathcal{NL} = \text{co}\mathcal{NL}$).

For more than two decades, undirected connectivity was one of the most appealing examples of the computational power of randomness. Recall that the classical (deterministic) linear-time algorithms (e.g., BFS and DFS) require an extensive use of (extra) memory (i.e., space linear in the size of the graph). On the other hand, it was known (since 1979, see §6.1.4.2) that, with high probability, a random walk of polynomial length visits all vertices (in the corresponding connected component). Thus, the randomized algorithm requires a minimal amount of auxiliary memory (i.e., logarithmic in the size of the graph). In the early 1990's, this algorithm (as well as the entire class $\mathcal{BPL}$ (see Definition 6.9)) was derandomized in polynomial-time and poly-logarithmic space (see Theorem 8.23), but despite more than a decade of research attempts, a significant gap remained between the space complexity of randomized and deterministic polynomial-time algorithms for this natural and ubiquitous problem. This gap was closed by Reingold [177], who established Theorem 5.6 in 2004. Our presentation (in Section 5.2) follows his ideas, but the specific formulation in §5.2.4.2 is new.

Exercises

Exercise 5.1 (rewriting on the write-only output-tape) Let A be an arbitrary algorithm of space complexity s. Show that there exists a functionally

[18] Specifically, the class of sets recognized by linear-space non-deterministic machines equals the class of context-sensitive languages (see, e.g., [116, Sec. 9.3]), and thus Theorem 5.14 resolves the question of whether the latter class is closed under complementation.

equivalent algorithm A' that never rewrites on (the same location of) its output-device and has space complexity s' such that $s'(n) = s(n) + O(\log \ell(n))$, where $\ell(n) = \max_{x \in \{0,1\}^n} |A(x)|$.

Guideline: Algorithm A' proceeds in iterations, where in the i^{th} iteration it outputs the i^{th} bit of $A(x)$ by emulating the computation of A on input x. The i^{th} emulation of A avoids printing $A(x)$, but rather keeps a records of the i^{th} location of $A(x)$'s output-tape (and terminates by outputing the final value of this bit). Indeed, this emulation requires maintaining the current value of i as well as the current location of emulated machine (i.e., A) on its output-tape.

Exercise 5.2 (on the power of double-logarithmic space) For any $k \in \mathbb{N}$, let w_k denote the concatenation of all k-bit long strings (in lexicographic order) separated by $*$'s (i.e., $w_k = 0^{k-2}00 * 0^{k-2}01 * 0^{k-2}10 * 0^{k-2}11 * \cdots * 1^k$). Show that the set $S \stackrel{\text{def}}{=} \{w_k : k \in \mathbb{N}\} \subset \{0,1,*\}$ is not regular and yet is decidable in double-logarithmic space.

Guideline: The non-regularity of S can be shown using standard techniques. Towards developing an algorithm, note that $|w_k| > 2^k$, and thus $O(\log k) = O(\log \log |w_k|)$. Membership of x in S is determined by iteratively checking whether $x = w_i$, for $i = 1, 2, ...$ Note that the i^{th} iteration can be implemented in space $O(\log i)$, and that on input $x \notin S$ we halt and reject after at most $\log |x|$ iterations.

Exercise 5.3 (on the weakness of less than double-logarithmic space) Prove that for $\ell(n) = \log \log n$, it holds that $\text{DSPACE}(o(\ell)) = \text{DSPACE}(O(1))$.

Guideline: Let s denote the machine's (binary) space complexity (see Footnote 2). Assuming that s is unbounded, consider for each m the shortest string x such that on input x the machine uses space at least m. Consider, for each location on the input, the sequence of the residual configurations of the machine (i.e., the contents of its temporary storage)[19] at the times in which the machine crosses (or rather passes through) this input location. For starters, note that the length of this "crossing sequence" is upper-bounded by the number of possible configurations, which is at most $t \stackrel{\text{def}}{=} 2^{s(|x|)} \cdot s(|x|)$. Thus, the number of such crossing sequences is upper-bounded by $(t+1)! < t^t$. Now, if $t^t < |x|/2$ then there exist three input locations that have the same crossing sequence, and two of them hold the same bit value. Contracting the string at these two locations, we get a shorter input on which the machine behaves in exactly the same manner, contradicting the hypothesis that x is the shortest input on which the machine uses space at least m. We conclude that $t^t \geq |x|/2$ must hold, and $s(|x|) = \Omega(\log t) = \Omega(\log \log |x|)$ follows.

Exercise 5.4 (some log-space algorithms) Present log-space algorithms for the following computational problems.

1. Addition and multiplication of a pair of integers.

[19] Note that, unlike in the proof of Theorem 5.3, the machine's location on the input is not part of the notion of a configuration used here. On the other hand, although not stated explicitly, the configuration also encodes the machine's location on the storage tape.

Guideline: Relying on Lemma 5.2, first transform the input to a more convenient format, then perform the operation, and finally transform the result to the adequate format. For example, when adding $x = \sum_{i=0}^{n-1} x_i 2^i$ and $y = \sum_{i=0}^{n-1} y_i 2^i$ a convenient format is $((x_0, y_0), ..., (x_{n-1}, y_{n-1}))$.

2. Transforming the adjacency matrix representation of a graph to its incidence list representation, and vice versa.

3. Deciding whether the input graph is acyclic (i.e., has no simple cycles).

   Guideline: Consider a scanning of the graph that proceeds as follows. Upon entering a vertex v via the i^{th} edge incident at it, we exit this vertex using its $i+1^{\text{st}}$ if v has degree at least $i+1$ and exit via the first edge otherwise. Note that when started at any vertex of any tree, this scanning performs a DFS. On the other hand, for every cyclic graph there exists a vertex v and an edge e incident to v such that if this scanning is started by traversing the edge e from v then it returns to v via an edge different from e.

4. Deciding whether the input graph is a tree.

   Guideline: Use the fact that a graph $G = (V, E)$ is a tree if and only if it is acyclic and $|E| = |V| - 1$.

Exercise 5.5 (another composition result) In continuation to the discussion in §5.1.2.3, prove that *if* Π *can be computed in space* s_1 *when given an* (ℓ, ℓ')*-restricted oracle access to* Π' *and* Π' *is solvable is space* s_2*, then* Π *is solvable in space* s *such that* $s(n) = 2s_1(n) + s_2(\ell(n)) + 2\ell'(n) + \log_2 \ell(n)$.

Guideline: Combine the ideas underlying the proofs of Lemmas 5.1 and 5.2. Specifically, view the oracle-aided computation of Π as consisting of iterations such that in the i^{th} iteration the i^{th} query is determined based on the initial input, the $i-1^{\text{st}}$ oracle answer and the contents of the work tape at the time the $i-1^{\text{st}}$ answer was given. Composing each iteration with the computation of Π' using Lemma 5.2, we conclude that the i^{th} answer can be computed (without storing the i^{th} query) in space $s_1(n) + s_2(\ell(n)) + \log_2 \ell(n)$. Thus, we can emulate the entire computation using space $s(n)$, where the extra space $s_1(n) + 2\ell'(n)$ is used for storing the work-tape of the oracle machine and the $i-1^{\text{st}}$ and i^{th} oracle answers.

Exercise 5.6 Referring to the discussion in §5.1.2.3, prove that any problem having space complexity s can be solved by a constant-space $(2s, 2s)$-restricted reduction to a problem that is solvable in constant-space.

Guideline: The reduction is to the "next configuration function" associated with the said algorithm (of space complexity s). To facilitate the computation of this function, represent each configuration in a redundant manner (e.g., as a sequence over a 4-ary rather than a binary alphabet). The reduction consists of iteratively copying strings from the (input or) oracle-answer tape to the oracle-query (or output) tape.

Exercise 5.7 (log-space uniform $\mathcal{NC}^1$ is in $\mathcal{L}$) Suppose that a problem Π is solvable by a family of log-space uniform bounded fan-in circuits of depth d such

that $d(n) \geq \log n$. Prove that Π is solvable by an algorithm having space complexity $O(d)$.

Guideline: Combine the algorithm outlined in Section 5.1.3 with the definition of log-space uniformity (using Lemma 5.2).

Exercise 5.8 (transitivity of log-space reductions) Prove that log-space Karp-reductions are transitive. Define log-space Levin-reductions and prove that they are transitive.

Guideline: Use Lemma 5.2, noting that such reductions are merely log-space computable functions.

Exercise 5.9 (UCONN in constant degree graphs of logarithmic diameter) Present a log-space algorithm for deciding the following promise problem, which is parameterized by constants c and d. The input graph satisfies the promise if each vertex has degree at most d and every pair of vertices that reside in the same connected component are connected by a path of length at most $c \log_2 n$, where n denotes the number of vertices in the input graph. The task is to decide whether the input graph is connected.

Guideline: For every pair of vertices in the graph, we check whether these vertices are connected in the graph. (Alternatively, we may just check whether each vertex is connected to the first vertex.) Relying on the promise, it suffices to inspect all paths of length at most $\ell \stackrel{\text{def}}{=} c \log_2 n$, and these paths can be enumerated using $\ell \cdot \lceil \log_2 d \rceil$ bits of storage.

Exercise 5.10 (warm-up towards §5.2.4.2) In continuation to §5.2.4.1, present a log-space transformation of G_i to G_{i+1}.

Guideline: Given the graph G_i as input, we may construct G_{i+1} by first constructing $G' = G_i^c$ and then constructing $G' \textcircled{z} G$. To construct G', we scan all vertices of G_i (holding the current vertex in temporary storage), and for each such vertex construct its neighborhood in G' (by using $O(c)$ space for enumerating all possible neighbors). Similarly, we can construct the vertex neighborhoods in $G' \textcircled{z} G$ (by storing the current vertex name and using a constant amount of space for indicating incident edges in G).

Exercise 5.11 (st-UCONN) In continuation to Section 5.2.4, prove that the following computational problem is in $\mathcal{L}$: Given an undirected graph $G = (V, E)$ and two designated vertices, s and t, determine whether there is a path from s to t in G.

Guideline: Note that the transformation described in Section 5.2.4 can be easily extended such that it map vertices in G_0 to vertices in $G_{O(\log |V|)}$ while preserving the connectivity relation (i.e., u and v are connected in G_0 if and only if their images under the map are connected in $G_{O(\log |V|)}$).

Exercise 5.12 Prove that the problem of determining whether or not the input graph is bipartite (2-colorable) is computationally equivalent under log-space reductions to `st-UCONN` (as defined in Exercise 5.11).

Guideline: Both reductions use the mapping of a graph $G=(V,E)$ to a bipartite graph $G'=(V',E')$ such that $V'=\{v^{(1)},v^{(2)}:v\in V\}$ and $E'=\{(u^{(1)},v^{(2)}),(u^{(2)},v^{(1)}):\{u,v\}\in E\}$. When reducing to `st-UCONN` note that a vertex v resides on an odd cycle in G if and only if $v^{(1)}$ and $v^{(2)}$ are connected in G'. When reducing from `st-UCONN` note that s and t are connected in G by a path of even (resp., odd) length if and only if the graph G' ceases to be bipartite when augmented with the edge $\{s^{(1)},t^{(1)}\}$ (resp., $\{s^{(1)},t^{(2)}\}$).

Exercise 5.13 (finding paths in undirected graphs) In continuation to Exercise 5.11, present a log-space algorithm that given an undirected graph $G=(V,E)$ and two designated vertices, s and t, finds a path from s to t in G (in case such a path exists).

Guideline: In continuation to Exercise 5.11, we may finds and store a logarithmic path in $G_{O(\log|V|)}$ that connects a representative of s and a representative of t. Focusing on the task of finding a path in G_0 that corresponds to an edge in $G_{O(\log|V|)}$, we note that such a path can be found by using the reduction underlying the combination of Claim 5.9 and Lemma 5.10. (An alternative description appears in [177].)

Exercise 5.14 (relating the two models of NSPACE) Referring to the definitions in Section 5.3.1, prove that for every function s such that $\log s$ is space constructible and at least logarithmic, it holds that $\mathrm{NSPACE}_{\text{on-line}}(s)=\mathrm{NSPACE}_{\text{off-line}}(\Theta(\log s))$.

Guideline (for $\mathrm{NSPACE}_{\text{on-line}}(s)\subseteq\mathrm{NSPACE}_{\text{off-line}}(O(\log s))$): Use the non-deterministic input of the off-line machine for encoding an accepting computation of the on-line machine; that is, this input should contain a sequence of consecutive configurations leading from the initial configuration to an accepting configuration, where each configuration contains the contents of the work-tape as well as the machine's location on the work-tape and on the input tape. The emulating off-line machine (which verifies the correctness of the sequence of configurations recorded on its non-deterministic input tape) needs only store its *location within the current pair of consecutive configurations* that it examines, which requires space logarithmic in the length of a single configuration (which in turn equals $s(n)+\log_2 s(n)+\log_2 n$). (Note that this verification relies on a two-directional access to the non-deterministic input.)

Guideline (for $\mathrm{NSPACE}_{\text{off-line}}(s')\subseteq\mathrm{NSPACE}_{\text{on-line}}(\exp(s'))$): Here we refer to the notion of a crossing-sequence. Specifically, for each location on the off-line non-deterministic input, consider the sequence of the residual configurations of the machine, where such a residual configuration consists of the bit residing in this non-deterministic tape location, the contents of the machine's temporary storage and the machine's locations on the input and storage tapes (but not its location on the non-deterministic tape). Show that the length of such a crossing-sequence is exponential in the space complexity of the off-line machine, and that the time complexity of the off-line machine is at most double-exponential in its space complexity (see Exercise 5.3). The on-line machine merely generates a sequence of crossing-sequences ("on the fly") and checks that each consecutive pair of crossing-sequences is consistent. This requires holding two crossing-sequences in storage, which require space linear in the length of such sequences (which, in turn, is exponential in the space complexity of the off-line machine).

Exercise 5.15 (st-CONN and variants of it are in NL) Prove that the following computational problem is in $\mathcal{NL}$. The instances have the form (G, v, w, ℓ), where $G=(V, E)$ is a directed graph, $v, w \in V$, and ℓ is an integer, and the question is whether G contains a path of length at most ℓ from v to w.

Guideline: Consider a non-deterministic (on-line) machine that generates and verifiers an adequate path on the fly. That is, starting at $v_0 = v$, the machine proceeds in iterations, such that in the i^{th} iteration it non-deterministically generates v_i, verifiers that $(v_{i-1}, v_i) \in E$, and checks whether $i \leq \ell$ and $v_i = w$. Note that this machine need only store the last two vertices on the path (i.e., v_{i-1} and v_i) as well as the number of edges traversed so far (i.e., i). Using a careful implementation, it suffices to store only one of these vertices (as well as i).

Exercise 5.16 (NSPACE and directed connectivity) Our aim is to establish a relation between general non-deterministic space-bounded computation and directed connectivity in "strongly constructible" graphs that have size exponential in the space bound. Let s be space constructible and at least logarithmic. For every $S \in \text{NSPACE}(s)$, present a linear-time oracle machine (somewhat as in §5.2.4.2) that given oracle access to x provides oracle access to a directed graph G_x of size $\exp(s(|x|))$ such that $x \in S$ if and only if there is a directed path between the first and last vertices of G_x. That is, on input a pair (u, v) and oracle access to x, the machine decides whether or not (u, v) is a directed edge in G_x.

Guideline: Follow the proof of Theorem 5.11.

Exercise 5.17 (an alternative presentation of the proof of Theorem 5.12) We refer to directed graphs in which each vertex has a self-loop.

1. Viewing the adjacency matrices of directed graphs as oracles (cf. Exercise 5.16), present a linear space oracle machine that determines whether a given pair of vertices is connected by a directed path of length two in the input graph. Note that this machine computes the adjacency relation of the square of the graph represented in the oracle.

2. Using naive composition (as in Lemma 5.1), present a quadratic space oracle machine that determines whether a given pair of vertices is connected by a directed path in the input graph.

Note that the machine in Item 2 implies that `st-CONN` can be decided in log-square space. In particular, justify the self-loop assumption made up-front.

Exercise 5.18 (finding shortest paths in undirected graphs) Prove that the following computational problem is $\mathcal{NL}$-complete under (many-to-one) log-space reductions: Given an undirected graph $G = (V, E)$, two designated vertices, s and t, and an integer K, determine whether there is a path of length at most K from s to t in G.

Guideline (for $\mathcal{NL}$-hardness): Reduce from `st-CONN`. Specifically, given a directed graph $G = (V, E)$ and vertices s, t, consider a ("layered") graph $G' = (V', E')$ such that

$V' = \cup_{i=0}^{|V|-1}\{\langle i,v\rangle : v \in V\}$ and $E' = \cup_{i=0}^{|V|-2}\{\{\langle i,u\rangle, \langle i+1,v\rangle\} : (u,v) \in E \vee u = v\}$. Note that there exists a directed path from s to t in G if and only if there exists a path of length $|V|-1$ between $\langle s,0\rangle$ and $\langle t,|V|-1\rangle$ in G'.

Exercise 5.19 (deciding strong connectivity) A directed graph is called strongly connected if there exists a directed path between every ordered pair of vertices in the graph (or, equivalently, a directed cycle passing through every two vertices). Prove that the problem of deciding whether a directed graph is strongly connected is $\mathcal{NL}$-complete under (many-to-one) log-space reductions.

Guideline (for $\mathcal{NL}$-hardness): Reduce from `st-CONN`, noting that (G,s,t) is a yes-instance of `st-CONN`, where $G=(V,E)$, if and only if the graph $G' = (V, E \cup \{(v,s) : v \in V\} \cup \{(t,v) : v \in V\})$ is strongly connected.

Exercise 5.20 (an operational interpretation of $\mathcal{NL} \cap \mathrm{co}\mathcal{NL}$, $\mathcal{NP} \cap \mathrm{co}\mathcal{NP}$, etc) Referring to Definition 5.13, prove that $S \in \mathcal{NL} \cap \mathrm{co}\mathcal{NL}$ if and only if there exists a non-deterministic log-space machine that computes χ_S, where $\chi_S(x) = 1$ if $x \in S$ and $\chi_S(x) = 0$ otherwise. State and prove an analogous result for $\mathcal{NP} \cap \mathrm{co}\mathcal{NP}$.

Guideline: A non-deterministic machine computing any function f yields, for each value v, a machine of similar complexity that accept $\{x : f(x) = v\}$. (Extra hint: Invoke the machine M that computes f and accept if and only if M outputs v.) On the other hand, for any function f of finite range, combining machines that accept the various $S_v \stackrel{\mathrm{def}}{=} \{x : f(x) = v\}$, we obtain a machine of similar complexity that computes f. (Extra hint: On input x, the combined machine invokes each of the aforementioned machines on input x and outputs the value v if and only if the machine accepting S_v has accepted. In the case that none of the machines accepts, the combined machine outputs $\perp$.)

Exercise 5.21 (a graph algorithmic interpretation of $\mathcal{NL} = \mathrm{co}\mathcal{NL}$) Show that there exists a log-space computable function f such that for every (G,s,t) it holds that (G,s,t) is a yes-instance of `st-CONN` if and only if $(G',s',t') = f(G,s,t)$ is a no-instance of `st-CONN`.

Exercise 5.22 As an alternative to the two-query reduction presented in the proof of Theorem 5.14, show that computing the characteristic function of `st-CONN` is log-space reducible via a single query to the problem of determining the number of vertices that are reachable from a given vertex in a given graph.
(Hint: On input (G,s,t), where $G = ([N],E)$, consider the number of vertices reachable from s in the graph $G' = ([2N], E \cup \{(t,N+i) : i = 1,...,N\})$.)

Exercise 5.23 (reductions and non-deterministic computations) Suppose that computing f is log-space reducible by a constant number of queries to computing some function g. Referring to the non-deterministic computations as in Definition 5.13, prove that if there exists a non-deterministic log-space machine that computes g then there exists a non-deterministic log-space machine that computes f.

Guideline: Use the emulative composition (as in Lemma 5.2). If any of the non-deterministic computations of g returns the value $\perp$ then return $\perp$ as the value of f.

Otherwise, use the non-$\perp$ values provided by the non-deterministic computations of g to compute the value of f.

Exercise 5.24 (reductions and non-deterministic computations, revisited) Suppose that computing f is log-space reducible (by any number of queries) to computing some function g such that for every x it holds that $|g(x)| = O(\log |x|)$. Referring to the non-deterministic computations as in Definition 5.13, prove that if there exists a non-deterministic log-space machine that computes g then there exists a non-deterministic log-space machine that computes f. As a warm-up consider the special case in which every query to g is computable in log-space based on the input to f.

Guideline: As in Exercise 5.23, except that here we use different composition techniques. Specifically, in the warm-up we use the naive composition (as in Lemma 5.1), whereas in the general case we use the semi-naive composition of Exercise 5.5.

Exercise 5.25 Referring to Definition 5.13, prove that there exists a non-deterministic log-space machine that computes the distance between two given vertices in a given undirected graph.

Guideline: Relate this computational problem to the decision problem considered in Exercise 5.18, and use $\mathcal{NL} = \text{co}\mathcal{NL}$.