# 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.

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:** 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, which seems a strictly more liberal restriction than polynomial time complexity. Indeed, algorithms utilizing polynomial space can perform almost all the computational tasks considered in this book (e.g., the class $\mathcal{PSPACE}$ contains almost all complexity classes considered in this book).

We first consider algorithms of logarithmic space complexity. Such algorithms may be used for solving various natural search and decision

problems, for providing reductions among such problems, and for 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 computations, focusing on the complexity class $\mathcal{NL}$ that is captured by the problem of deciding directed connectivity of (directed) graphs. The highlight of this part is a proof that $\mathcal{NL} = \text{co}\mathcal{NL}$, which may be paraphrased as a log-space reduction of directed unconnectivity to directed connectivity.

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 the relevant issues 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 and issues

We start by discussing several very important conventions regarding space complexity (see Section 5.1.1). Needless to say, reading Section 5.1.1 is essential for the understanding of the rest of this chapter. We then discuss a variety of issues, highlighting the differences between space-complexity and time-complexity. In particular, we call the reader's attention to the composition lemmas (§5.1.3.1) and related reductions (§5.1.3.3) as well as to the obvious simulation result presented in §5.1.3.2 (i.e., $\text{DSPACE}(s) \subseteq \text{DTIME}(2^{O(s)})$). Lastly, in Section 5.1.4 we relate circuit size to space complexity by considering the space-complexity of circuit evaluation (see also §5.3.2.2).

### 5.1.1 Important conventions

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

[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 presumes that the machine scans its storage devices in a serial manner. In contrast, we stress that the various notions of an instantaneous configuration do not assume such a machine model.

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 actual storage. Specifically, the binary space complexity of a computation *refers to 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, using the definition of *binary space complexity* facilitates 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, *we 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 sub-linear 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 location of) the write-only output-tape is inessential, and in fact can be eliminated at a relatively small cost (see Exercise 5.1).

**Summary.** Let us compile a list of the foregoing conventions. As stated, the first two items on the list are of crucial importance, while the rest are of technical nature (but do facilitate our exposition).

1. Space complexity discards the use of the input and output devices.

[2]We note that, unlike in the context of time-complexity, linear speed-up (as in Exercise 4.7) 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).

2. The input device is read-only and the output device is write-only.

3. We will usually refer to the binary space complexity of algorithms, where the binary space complexity of a machine $M$ that uses the alphabet $\Sigma$, finite state set $Q$, and has standard space complexity $S_M$ is defined as $(\log_2 |Q|) + (\log_2 |\Sigma|) \cdot S_M$. (Recall that $S_M$ measures the number of cells of the temporary storage device that are used by $M$ during the computation.)

4. We will assume that the machine does not scan the input-device beyond the boundaries of the input.

5. We will assume that the machine does not rewrite to locations of its output-device (i.e., it write to each cell of the output-device at most once).

### 5.1.2 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 [119, Chap. 2] are decidable by constant-space Turing machines and that this is all that the latter can decide (see, e.g., [119, 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 (and/or the allowed amount of space).[3] 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. On the other hand, it turns out that double-logarithmic space is indeed the smallest amount of space that is more useful than constant space (see Exercise 5.3); that is, *for* $\ell(n) = \log\log n$*, it holds that* $\text{DSPACE}(o(\ell)) = \text{DSPACE}(O(1))$.

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.

**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. We need to be fully aware of this danger whenever we seek impossibility results and/or complexity lower-bounds.

[3] Indeed, for this approach to work, we should be able to detect the case that the input is not of the proper form (and do so within sub-logarithmic space).

### 5.1.3 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.

#### 5.1.3.1 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.*[4] *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 $\ell$ 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* $f_1, f_2, s_1, s_2, \ell$ *and* $f$ *be as in Lemma 5.1. Then* $f$ *is computable in space* $s$ *such that*

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

*where* $\delta(n) = O(\log(s_1(n) + s_2(n + \ell(n)))) = o(s(n))$.

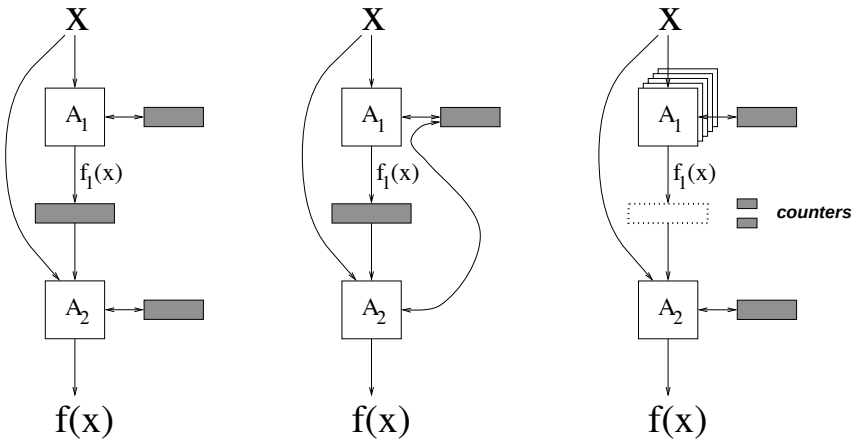
The alternative compositions are depicted in Figure 5.1 (which also shows the most straightforward composition of $A_1$ and $A_2$ that makes no attempt to economize space).

**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 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 location of) its write-only output-tape. As shown in Exercise 5.1, this assumption can be justified at an additive cost of $O(\log \ell(n))$.[5]

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

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




*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

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 $i^{\text{th}}$ 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 $i^{\text{th}}$ 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 $A_2$ and for the computation of $A_1$. In addition, we need to allocate separate storage for maintaining the aforementioned counters (i.e., we use $\log_2(n+\ell(n))$ bits to hold the location of the input-bit currently read by $A_2$, and $\log_2 \ell(n)$ bits to hold the index of the output-bit currently produced in the current invocation of $A_1$).[6] ■

[6] The additional $\delta(n)$ term takes care of the following issue. Our description of the composed algorithm refers to two storage devices, one for the computation of $A_1$ and the other for the computation of $A_2$. Indeed, we can obtain an algorithm that uses a single storage device and a

**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])).

#### 5.1.3.2 An obvious bound

The time complexity of an algorithm is essentially upper-bounded by an exponential function 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 to 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 $t(n) \leq n \cdot 2^{s(n)+\log_2 s(n)}$.*

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 $\ell(|x|) = s(|x|) + \log_2 |x| + \log_2 s(|x|)$.[7]

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 $\gamma$, will also happen ($i$ steps) after the second time that the computation $A(x)$ passes through $\gamma$.

By the forgoing observation, we infer that the number of steps taken by $A$ on input $x$ is at most $2^{\ell(|x|)}$, because otherwise the same configuration will appear twice in the computation (which contradicts the halting hypothesis). The theorem follows. ■

---

single pointer to locations on this device, but this requires holding the two original pointers in memory.

[7] Here we rely on the fact that $s$ is the binary space complexity (and not the standard space complexity).

#### 5.1.3.3 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:)[8] 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 $s(n) = s_1(n) + s_2(\ell(n)) + O(\log \ell(n)) + \delta(n)$, where $\delta(n) = O(\log(s_1(n) + s_2(\ell(n)))) = o(s(n))$.

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 $\ell$ is not bounded by $s_1$ itself (as in the analogous case of time-bounded computation), but rather by $\exp(s_1)$.

Things get much more complicated when we turn to general (space-bounded) reductions, especially when referring to general 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)). This suffices for general reductions that make a single query, but more difficulties arise when the reduction makes several adaptive queries (i.e., queries that depend on the answers to prior queries).

**Teaching note:** The rest of the discussion is quite advanced and laconic (but is inessential to the rest of the chapter).

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

[8] 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 $n + \ell(n)$.

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 bound that holds for the length of the first query (i.e., a reduction of space complexity $s$ is allowed to make queries of length at most $2^{s(n)+\log_2 s(n)} \cdot n$).

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 $(\ell, \ell')$-restricted if, 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* $\Pi$ *can be computed in space* $s_1$ *when given* $(\ell, \ell')$*-restricted oracle access to* $\Pi'$ *and* $\Pi'$ *is solvable is space* $s_2$*, then* $\Pi$ *is solvable in space* $s$ *such that* $s(n) = s_1(n) + s_2(\ell(n)) + \ell(n) + \ell'(n) + \delta(n)$*, where* $\delta(n) = O(\log(\ell(n) + \ell'(n) + s_1(n) + s_2(\ell(n)))) = o(s(n))$. The claim is proved by using a naive emulation that allocates separate space for the reduction (i.e., oracle machine) itself, for the emulation of its query and answer devices, and for the algorithm solving $\Pi'$. Note that here we cannot re-use the space of the reduction when running the algorithm that solves $\Pi'$, because the reduction's computation continues after the oracle answer is obtained. The additional $\delta(n)$ term accounts for the various pointers of the oracle machine, which need to be stored when algorithm that solves $\Pi'$ is invoked (see also Footnote 6).

   A related composition result is presented in Exercise 5.5. It yields $s(n) = 2s_1(n) + s_2(\ell(n)) + 2\ell'(n) + O(\log(\ell(n) + s_1(n) + s_2(\ell(n))))$, which for $\ell(n) < 2^{O(s_1(n))}$ means $s(n) = O(s_1(n)) + (1 + o(1))s_2(\ell(n)) + 2\ell'(n)$.

2. Turning to the approach underlying the proof of Lemma 5.2, we get into more serious trouble. Specifically, note that recomputing the answer of the $i^{\text{th}}$ 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.

#### 5.1.3.4 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).

#### 5.1.3.5 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 that consists of all decision problems that may be solved by a polynomial-time algorithm of polylogarithmic space complexity. This class, denoted $\mathcal{SC}$, is indeed a natural sub-class of $\mathcal{P}$ (and contains the class $\mathcal{L}$, which is defined in Section 5.2.1).[9]

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 $\text{DTISP}(\cdot,\cdot)$.

### 5.1.4 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 (which may be 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 manipulating the initial input and final output (i.e., assigning the bits of $x$ to the corresponding input terminals of $C$ and scanning all output terminals of $C$).

[9] 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 8.4 (see Theorem 8.23).

## 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 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 §5.1.3.3, 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 also 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* $(C, \alpha)$ *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 (as well as the proof of Theorem 3.6) is that the computation of a Turing machine can be emulated by a ("highly uniform") family of circuits. 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). For the sake of future use (in Section 5.2.3), we highlight the fact that $C_{|x|}$ can be constructed by a log-space machine that is given the input $1^{|x|}$. □

**The impact of P-completeness under log-space reductions.** Indeed, Theorem 5.4 implies that $\mathcal{L} \neq \mathcal{P}$ if any only if 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 $\Pi$ is $\mathcal{C}$-complete under log-space reductions if $\Pi$ is in $\mathcal{C}$ and every problem in $\mathcal{C}$ is log-space (many-to-one) reducible to $\Pi$. 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 5.1.4, 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 $C_{|x|}(x) = A(x)$.*

**Proof Sketch:** As noted already, the circuits $(C_{|x|})_{|x|}$ are highly uniform. In particular, the underlying digraph consists of constant-size gadgets that are arranged in an array and are only connected to adjacent gadgets (see the proof of Theorem 2.20). □

### 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*.[10] 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 related computational problems seem actually 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 state of affairs, 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).[11]

Needless to say, 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

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

[11] Indeed, this is analogous to the circuit evaluation algorithm of Section 5.1.4, 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.

of connected components of the graph. Furthermore, 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 (i.e., the rest of this section) 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 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:** Evidently, the notion of an expander graph plays a key role in the aforementioned transformation. In particular, we assume familiarity with the algebraic definition of expanders (as presented in §E.2.1.1). The transformation relies heavily on the *zig-zag product*, defined in §E.2.2.2, and the following exposition assume familiarity with this definition.

#### 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.10). Recall that Eq. (E.10) 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.

Needless to say, a "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.[12] 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 [183], 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

[12] We cannot afford the naive composition (of Lemma 5.1), because 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.

when applied to the intermediate graphs; that is, 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 $G_i$). This means that 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.[13]

Like in §5.1.3.3, 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[14] 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 $G_i^2$ (i.e., the square of the graph $G_i$) can be evaluated at any desired pair, by evaluating the edge rotation function of $G_i$ twice, and using a constant amount of space. Specifically, given $v \in V_i$ and $j_1, j_2 \in [d^2]$, we compute $g_i(g_i(v, j_1), j_2)$, which is the edge rotation of $(v, \langle j_1, j_2 \rangle)$ in $G_i^2$, as

[13] 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 "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.

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

follows. First, making the query $(v, j_1)$, we obtain the edge rotation of $(v, j_1)$, denoted $(u, k_1)$. Next, making the query $(u, j_2)$, we obtain $(w, k_2)$, and finally we 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 an arbitrary regular graph $G'$ with a fixed graph $G$), 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.8), 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 $G$ (and affect a constant number of bits of the relevant strings). Again, using the fact that it suffices to copy vertex names from the input to the oracle query device (or from the oracle answer device to the 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 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) causes a multiplicative overhead per each composition, 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*[15]*, 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, $\ell$ denote the length of the longest query or oracle answer, and $m$ denotes the length of the output. However, combining these three 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 store three corresponding counters (in addition to storing the original work-tape). Thus, the resulting machine uses a local-tape of length $w + \log_2 n + \log_2 \ell + \log_2 m$, where $w$ denotes the space complexity of the original machine and the additional logarithmic terms (which are logarithmic in the length of the global-tape) account for the aforementioned counters.

Fortunately, the aforementioned counters can be avoided in the case that the original oracle machine can be described as an iterative sequence of transformations (i.e., the input is transformed to the first query, and the $i^{\text{th}}$ answer is transformed to the $i+1^{\text{st}}$ query or to the output, all 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 (rather than logarithmic in the length of the global-tape).

**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 $\log_2(d^2 \cdot |V_{i+1}|)$ 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.

[15] This means that as a result of invoking the oracle $f$, the contents of the global-tape changes from $q$ to $f(q)$. We stress that the prior contents of the global-tape (i.e., the query $q$) is lost (i.e., it is replaced by the answer $f(q)$).

Similarly, note that the edge rotation function of the zig-zag product of the variable graph $G'$ with the fixed graph $G$ 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.8)). This means that the global-tape oracle machine first copies $\alpha,\beta$ 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.[16] In general, when sequentially composing such computations the length of the global-tape (resp., local-tape) is the maximum among all composed computations; that is, the current formalism offers a tight bound on naive *sequential 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.

**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* $f_{i+1}$ *by making oracle calls to* $f_i$ *while using a global-tape of length* $L$ *and a local-tape of length* $l_i$, *which also accounts for the machine's state. Then* $f_t$ *can be computed by a* standard oracle machine *that makes calls to* $f_1$ *and uses space* $L+2\sum_{i=1}^{t-1}l_i$.

We shall apply this lemma with $f_i=g_i$ and $t=O(\log|V_1|)=O(\log|V_t|)$, using the bounds $L=\log_2(d^2\cdot|V_t|)$ and $l_i=O(1)$ (as guaranteed by Claim 5.9). Indeed, in this application $L$ equals the length of the input to $f_t=g_t$.

**Proof Sketch:** We compute $f_t$ by allocating space for the emulation of the global-tape and the local-tapes of each level in the recursion. We emulate the recursive computation by capitalizing on the fact that all recursive levels use the same global-tape (for making queries and receiving answers). Recall that 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. The emulation should also store the locations of the other levels of the recursion on the corresponding local-tapes, but the space needed for this is clearly smaller than the length of the various local-tapes. □

**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 $O(\log|V_1|)$.

[16] A similar composition took place in the proof of Claim 5.7, but in Claim 5.9 we asserted a stronger feature of this specific computation.

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 phenomenon is the huge gap between the power of two formulation of non-deterministic space-complexity (see Section 5.3.1), which stands in contrast to the fact that the analogous formulations are equivalent in the context of time-complexity. We also highlight the contrast between various 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; for example, consider the question of complementation (cf. §5.3.2.3).

### 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. In the context of time-complexity, these models are equivalent because the latter on-line choices can be recorded (almost) for free (see the proof of Theorem 2.8). However, such a recording is not free of charge in the context of space-complexity.

Let us take a closer look at the relation between the off-line and on-line models. The fact that the off-line model can emulate the on-line model is almost generic; that is, it holds for any reasonable notion of complexity, because it is based on the fact that the off-line machine can emulate on-line choices by using its non-deterministic input (and without significantly effecting the complexity measure). In contrast, the emulation of the off-line model by the on-line model is enabled by the fact that *in the context of time-complexity* an on-line machine may store (and re-use) a sequence of non-deterministic (on-line) choices without significantly effecting the running-time (i.e., almost "free of charge"). This naive emulation (of the off-line model on the on-line model) is not free of charge in the context of space-bounded computation. Furthermore, typically the number of non-deterministic choices is much larger than the space-bound, and thus the naive emulation is not possible *in the context of space-complexity* (because it is prohibitively expensive in terms of space-complexity). Let us formulate the two models and consider the

relation between them in the context of space-complexity.

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 $\ell_c(n) = c \log_2 n$. (We refer the reader to the definitional issues pertaining $\text{NSPACE} = \text{NSPACE}_{\text{on-line}}$, which 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 com-

plete 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 $G$).[17] 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).[18] Note that (for a fixed machine $M$), 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. □

**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

[17] 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*.

[18] 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 $V_x$.

computations. Note that this (intuitive and informal) 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 and Exercise 5.18). 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 5.19.

#### 5.3.2.2 Relating NSPACE to DSPACE

Recall that in the context of time-complexity, the only known conversion of non-deterministic computation to deterministic computation comes at the cost of 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., [119, Sec. 12.5]). The special case boils down to presenting an algorithm for deciding directed connectivity that has log-square space-complexity.

The basic idea is that checking whether or not there is a path of length at most $2\ell$ 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 $\ell$ from $u$ to $w$ and a path of length at most $\ell$ from $w$ to $v$. That is, let $\phi_G(u,v,\ell) \stackrel{\text{def}}{=} 1$ if there is a path of length at most $\ell$ from $u$ to $v$ in $G$, and $\phi_G(u,v,\ell) \stackrel{\text{def}}{=} 0$ otherwise. Then $\phi_G(u,v,2\ell)$ can be computed by scanning all vertices $w$ in $G$, and checking for each $w$ whether both $\phi_G(u,w,\ell) = 1$ and $\phi_G(w,v,\ell) = 1$ hold.[19] Hence, we can compute $\phi_G(u,v,2\ell)$ by a log-space algorithm that makes oracle calls to $\phi_G(\cdot,\cdot,\ell)$, which in turn can be computed recursively in the same manner. Note that the original computational problem (i.e., st-CONN) can be cast as *computing* $\phi_G(s,t,|V|)$ (or $\phi_G(s,t,2^{\lceil \log_2 |V| \rceil})$) *for a given directed graph* $G = (V,E)$ *and a given pair of vertices* $(s,t)$. Thus, the foregoing recursive procedure yields the theorem's claim, provided that we use adequate composition results. We take a technically different approach by directly analyzing the recursive procedure at hand.

Recall that given a directed graph $G = (V,E)$ and a pair of vertices $(s,t)$, we should merely compute $\phi_G(s,t,2^{\lceil \log_2 |V| \rceil})$. This is done by invoking a recursive

[19] Similarly, $\phi_G(u,v,2\ell+1)$ can be computed by scanning all vertices $w$ in $G$, and checking for each $w$ whether both $\phi_G(u,w,\ell+1) = 1$ and $\phi_G(w,v,\ell) = 1$ hold.

procedure that computes $\phi_G(u,v,2\ell)$ by scanning all vertices in $G$, and computing for each vertex $w$ the values of $\phi_G(u,w,\ell)$ and $\phi_G(w,v,\ell)$. The punch-line is that all these computations may re-use the same space, while we need only store one additional bit representing the results of all prior computations. We return the value 1 if and only if for some $w$ it holds that $\phi_G(u,w,\ell) = \phi_G(w,v,\ell) = 1$ (see Figure 5.2). Needless to say, $\phi_G(u,v,1)$ 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 $\sigma \leftarrow \phi_G(u,w,\ell)$ (*by a recursive call*)
  Compute $\sigma \leftarrow \sigma \wedge \phi_G(w,v,\ell)$ (*by a second recursive call*)
  If $\sigma = 1$ 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 $\mathcal{NL} \subseteq \text{DSPACE}(O(\log^2))$.

We consider an implementation of the foregoing procedure (of Figure 5.2) in which each level of the recursion uses a designated portion of the entire storage for maintaining the local variables (i.e., $w$ and $\sigma$). The amount of space taken by each level of the recursion is essentially $\log_2 |V|$ (for storing the current value of $w$), and the number of levels is $\log_2 |V|$. We stress that when computing $\phi_G(u,v,2\ell)$, we make many recursive calls, but all these calls re-use the same work space (i.e., the portion that is designated to that level). That is, when we compute $\phi_G(u,w,\ell)$ we re-use the space that was used for computing $\phi_G(u,w',\ell)$ for the previous $w'$, and we re-use the same space when we compute $\phi_G(w,v,\ell)$. Thus, the space-complexity of our algorithm is merely the sum of the amount of space used by all recursion levels. It follows that `st-CONN` has log-square (deterministic) space-complexity, and the same follows for all of $\mathcal{NL}$ (either by noting that `st-CONN` actually represents any $\mathcal{NL}$ computation or by using the log-space reductions of $\mathcal{NL}$ to `st-CONN`). □

**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 verified 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 any NSPACE class).

The simple formulation of `st-CONN` facilitates placing $\mathcal{NL}$ in complexity classes such as $\mathcal{NC}^2$ (i.e., decidability by uniform families of circuits of log-square depth

and bounded fan-in). 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 circuits of logarithmic depth and bounded fan-in (i.e., in NC1), and by repeated squaring the $n^{\text{th}}$ 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, st-CONN $\in \mathcal{NC}^2$. Indeed, $\mathcal{NL} \subseteq \mathcal{NC}^2$ follows by noting that st-CONN actually represents any $\mathcal{NL}$ computation (or by noting that any log-space reduction can be computed by a uniform family of logarithmic depth and bounded fan-in circuits).

#### 5.3.2.3 Complementation or NL=coNL

Recall that (reasonable) 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, (reasonable) 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 that $\mathcal{NL} = \text{co}\mathcal{NL}$, we note that proving this result is equivalent to presenting a log-space Karp-reduction of st-CONN to its complement (or, equivalently, a reduction in the opposite direction, see Exercise 5.21). Our proof utilizes a different perspective on the NL-vs-coNL question, by rephrasing this question as referring to the relation between $\mathcal{NL}$ and $\mathcal{NL} \cap \text{co}\mathcal{NL}$, and by offering an "operational interpretation" of the class $\mathcal{NL} \cap \text{co}\mathcal{NL}$.

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 $x \notin S$. In contrast, 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 non-deterministic 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 the characteristic function of $S$ (see Exercise 5.20). Recall that the characteristic function of $S$, denoted $\chi_S$, is the Boolean function satisfying $\chi_S(x) = 1$ if $x \in S$ and $\chi_S(x) = 0$ otherwise. 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 $\chi_S$.

**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 $\chi_S$.*

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:** As in the proof of Theorem 5.12, it suffices to present a non-deterministic (on-line) log-space machine that computes the characteristic function of `st-CONN`, denoted $\chi$ (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 $\chi$ is log-space reducible (by two queries)[20] to determining the number of vertices that are reachable (via a directed path) from a given vertex in 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 this number to the number of vertices reachable from $s$ in the graph obtained by deleting $t$ from $G$. Clearly, the two numbers are different if and only if vertex $t$ is reachable from vertex $v$ (in the graph $G$). (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 $i^{\text{th}}$ 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 $|R_i|$,

[20] We stress the fact that only two queries are used in the reduction, because this avoids the difficulties (discussed in §5.1.3.3) 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 answers have logarithmic length. For details, see Exercises 5.23 and 5.24.

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 number of vertices that were already verified to be in $R_i$, another $\log_2 n$ 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 do 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).[21]

   (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 $u \notin R_i$.

   We use $\log_2 n$ bits to store the number of vertices that were already verified to be in $V \setminus R_i$, another $\log_2 n$ bits to store the current vertex $u$, another $\log_2 n$ bits to count the number of vertices that are currently 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 $g$.

[21] 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 $\mathcal{NL} = \text{co}\mathcal{NL}$.

It can be verified that, when given the correct value of $|R_{i-1}|$, the foregoing non-deterministic procedure uses a logarithmic amount of space and 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-type sets is the archetypical task of non-deterministic procedures (i.e., they are defined so to fit these tasks), and thus Step 1 is straightforward. In contrast, non-deterministic verification of non-membership

is not a common phenomenon, and thus Step 2 is not straightforward at all. In the current context (of Step 2), the verification of non-membership is performed 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 the verification of proofs (cf., Section 2.1.4). Since these fictitious abstractions are fundamental in the context of time-complexity, we may hope to gain some understanding by a comparative study; specifically, a study of non-deterministic 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 on the type of the notion of feasibility. Turning to the other type of benefits, we learned that `st-CONN` can be Karp-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 seems actually 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 established 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 be 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 taking notes for future verification, rather than scanning them back and forth all the time. Thus, the on-line model reflects the true space-complexity of taking such notes and hence of sequential verification of proofs. Indeed (as stated in Section 5.3.1), our feeling is that the off-line model allows for an unfair accounting of temporary space as well as for unintendedly long proofs.

## 5.4 PSPACE and Games

As stated up-front, we rarely encounter 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 4 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 a move $y_1$ for the first party, such that for every response move $y_2$ of the second party, there exists a 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).[22]

**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

[22] See Appendix G.2.

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 5.1.4.) 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 $\ell$ 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 $m = \mathrm{poly}(|x|)$ 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 "$\forall (u'v') \in \{(u,w),(w,v)\}\ \phi_G(u',v',\ell)$" (where we quantify over a two-element set that is not the Boolean set $\{0,1\}$). Next, we reformulate the 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 $\sigma$ 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 $\Phi$ is recursively defined as follows

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

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 (fixed) *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 [200, 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). We wish to emphasize 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 [190]), the

$\mathcal{PSPACE}$-completeness of QBF, and the connections between circuit depth and space complexity (see Section 5.1.4 and Exercise 5.7 due to Borodin [45]).

Before turning to the "modern" results, we mention that some researchers 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 our opinion this impression is fundamentally mistaken. Specifically, in addition to the fact that substantial progress towards the understanding of many fundamental issues has been achieved, these researchers 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).[23] This open problem was resolved in 1988 by Immerman [121] and Szelepcsenyi [211], who (independently) proved Theorem 5.14 (i.e., $\mathcal{NL} = \text{co}\mathcal{NL}$).

For more than two decades, undirected connectivity (UCONN) was one of the most appealing examples of the computational power of randomness. Recall that the classical linear-time (deterministic) algorithms (e.g., BFS and DFS) require an extensive use of temporary storage (i.e., 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 resulting randomized algorithm for UCONN uses a minimal amount of temporary storage (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 6.11)) 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 [183], who established Theorem 5.6 in 2004.[24] Our presentation (in Section 5.2.4) follows Reingold's ideas, but the specific formulation in §5.2.4.2 does not appear in [183].

## 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 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 $i^{\text{th}}$ iteration it outputs the $i^{\text{th}}$ bit of $A(x)$ by emulating the computation of $A$ on input $x$. The $i^{\text{th}}$ emulation of $A$

[23] Specifically, the class of sets recognized by linear-space non-deterministic machines equals the class of context-sensitive languages (see, e.g., [119, Sec. 9.3]), and thus Theorem 5.14 resolves the question of whether the latter class is closed under complementation.

[24] We mention that an almost-logarithmic space algorithm was discovered independently and concurrently by Trifonov [215], using a very different approach.

avoids printing $A(x)$, but rather keeps a records of the $i^{\text{th}}$ location of $A(x)$'s output-tape (and terminates by outputting 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 $i = 1, 2, ...$, while stopping when detecting an obvious case (i.e., either verifying that $x = w_i$ or detecting evidence that $x \neq w_k$ for every $k \geq i$). By taking advantage of the $*$'s (in $w_i$), the $i^{\text{th}}$ iteration can be implemented in space $O(\log i)$. Furthermore, on input $x \notin S$, we halt and reject after at most $\log|x|$ iterations. Actually, it is slightly simpler to handle the related set $\{w_1 **w_2 ** \cdots ** w_k : k \in \mathbb{N}\}$; moreover, in this case the $*$'s can be omitted from the $w_i$'s (as well as from between them).

**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. Show that if $s$ is unbounded then it must hold that $s(n) = \Omega(\log\log n)$ infinitely often. Specifically, for each $m$, consider a 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)[25] such that the $i^{\text{th}}$ element in the sequence represents the residual configuration of the machine at the $i^{\text{th}}$ time that 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 residual 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 $t^t$. 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 $s(|x|) = \Omega(\log\log|x|)$ holds for infinitely many $x$'s.

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

1. Addition and multiplication of a given pair of integers.

[25] 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. Deciding whether two given strings are identical.

3. Finding occurrences of a given pattern $p \in \{0,1\}^*$ in a given string $s \in \{0,1\}^*$.

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

5. 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 $i^{\text{th}}$ 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 $e$.

6. 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 §5.1.3.3, prove that *if* $\Pi$ *can be computed in space* $s_1$ *when given an* $(\ell, \ell')$*-restricted oracle access to* $\Pi'$ *and* $\Pi'$ *is solvable is space* $s_2$*, then* $\Pi$ *is solvable in space* $s$ *such that* $s(n) = 2s_1(n) + s_2(\ell(n)) + 2\ell'(n) + \delta(n)$*, where* $\delta(n) = O(\log(\ell(n) + s_1(n) + s_2(\ell(n))))$. In particular, if $s_1, s_2$ and $\ell'$ are at most logarithmic, then $s(n) = O(\log n)$.

**Guideline:** View the oracle-aided computation of $\Pi$ as consisting of iterations such that in the $i^{\text{th}}$ iteration the $i^{\text{th}}$ query (denoted $q_i$) is determined based on the initial input (denoted $x$), the $i-1^{\text{st}}$ oracle answer (denoted $a_{i-1}$), and the contents of the work tape at the time the $i-1^{\text{st}}$ answer was given (denoted $w_{i-1}$). Note that the mapping $(x, a_{i-1}, w_{i-1}) \to (q_i, w_i)$ can be computed using $s(|x|)$ bits of temporary storage. Composing each iteration with the computation of $\Pi'$ (using Lemma 5.2), we conclude that the mapping $(x, a_{i-1}, w_{i-1}) \to (a_i, w_i)$ can be computed (without storing the intermediate $q_i$) in space $s_1(n) + s_2(\ell(n)) + O(\log(\ell(n) + s_1(n) + s_2(\ell(n))))$. Thus, we can emulate the entire computation using space $s(n)$, where the extra space of $s_1(n) + 2\ell'(n)$ bits is used for storing the work-tape of the oracle machine and the $i-1^{\text{st}}$ and $i^{\text{th}}$ oracle answers.

**Exercise 5.6** Referring to the discussion in §5.1.3.3, 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 $s$), where here the configuration contains also the

single bit of the input that the machine currently examines (i.e., the value of bit at the machine's location on the input device). 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 string (with minor modification) from the (input or) oracle-answer tape to the oracle-query (or output) tape.

**Exercise 5.7 (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 5.8 (log-space uniform $\mathcal{NC}^1$ is in $\mathcal{L}$)** Suppose that a problem $\Pi$ is solvable by a family of log-space uniform circuits of bounded fan-in and depth $d$ such that $d(n) \geq \log n$. Prove that $\Pi$ is solvable by an algorithm having space complexity $O(d)$.

**Guideline:** Combine the algorithm outlined in Section 5.1.4 with the definition of log-space uniformity (using Lemma 5.2).

**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 is 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 maps 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 (Bipartiteness)** 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., with the edges $\{s^{(1)},x\}$ and $\{x,t^{(2)}\}$, where $x\notin V'$ is an auxiliary vertex).

**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 find and (implicitly) 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 [183].)

**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 state and its locations 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 $s(n)+\log_2 s(n)+\log_2 n+O(1)$). (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 $(G,v,w,\ell)$, where $G=(V,E)$ is a directed graph, $v,w \in V$, and $\ell$ is an integer, and the question is whether $G$ contains a path of length at most $\ell$ 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 $i^{\text{th}}$ 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., $i$). (Actually, using a careful implementation, it suffices to store only one of these two vertices (as well as the current $i$).)

**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 (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. Note that, for any graph $G = (V, E)$, it holds that $(G, s, t)$ is a yes-instance of st-CONN 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 5.19 (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 (resp., exactly) $K$ 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 at most (resp., exactly) $|V| - 1$ between $\langle 0, s\rangle$ and $\langle |V| - 1, t\rangle$ in $G'$.

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

**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 sets $S_v \stackrel{\text{def}}{=} \{x : f(x) = v\}$, 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 $\mathcal{NL} = \text{co}\mathcal{NL}$)** 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 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 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 only 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 (in the spirit of Lemma 5.1), whereas in the general case we apply the semi-naive composition result 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 5.19, and use $\mathcal{NL} = \text{co}\mathcal{NL}$.