Texts in Computational Complexity: Space Complexity

Oded Goldreich
Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Rehovot, Israel.

December 17, 2005

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

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

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

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

Organization. Space complexity seems to behave quite differently from time complexity, and seems to require a different mind-set as well as auxiliary conventions. Some of these are discussed in Section 1. We then turn to the study of logarithmic space complexity (see Section 2) and the corresponding non-deterministic version (see Section 3). Finally, we consider polynomial space complexity (see Section 4).

1 General preliminaries

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

Fixing a concrete model of computation (e.g., multi-tape Turing machines), we *denote by* $\text{DSPACE}(s)$ *the class of decision problems that are solvable in space complexity* s. The space complexity of search problems is defined analogously. Specifically, the *standard definition of space complexity 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 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.[1]

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

1.1 On the minimal amount of useful computation space

Bearing in mind that one of our main objectives is identifying natural sub-classes of $\mathcal{P}$, we consider the question of what is the minimal amount of space that allows for meaningful computations. We note that regular sets [5, Chap. 2] are decidable by constant-space Turing machines and that this is all that the latter can decide (see, e.g., [5, Sec. 2.6]). It is tempting to say that sub-logarithmic space machines are not more useful than constant-space machines, because it *seems* impossible to allocate a sub-logarithmic amount of space. This wrong intuition is based on the presumption that the allocation of a non-constant amount of space requires explicitly computing the length of the input, which in turn requires logarithmic space. However, this presumption is wrong: the input itself (in case it is of a proper form) can be used to determine its length, whereas in case the input is not of the proper form then this fact may be detectable (within sub-logarithmic space). In fact, *for* $\ell(n) = \log\log n$*, the class* $\text{DSPACE}(O(\ell))$ *is a proper superset of* $\text{DSPACE}(O(1))$; see Exercise 12. In contrast to Exercise 12, double-logarithmic space is indeed the smallest amount of space that is more useful than constant space; that is, *for* $\ell(n) = \log\log n$*, it holds that* $\text{DSPACE}(o(\ell)) = \text{DSPACE}(O(1))$; see Exercise 13.

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 2). As we shall see, this class is the natural habitat of several fundamental computational phenomena.

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

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

1.2 Time versus Space

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

1.2.1 Two composition lemmas

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

Lemma 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.*[2] *Then f defined by $f(x) \stackrel{\rm def}{=} f_2(x, f_1(x))$ 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 1 is useful when ℓ is relatively small, but in many cases $\ell \gg \max(s_1, s_2)$. In these cases, the following composition lemma is more useful.

Lemma 2 (emulative composition): *Let f_1, f_2, s_1, s_2, ℓ and f be as in Lemma 1. Then f is computable in space s such that*

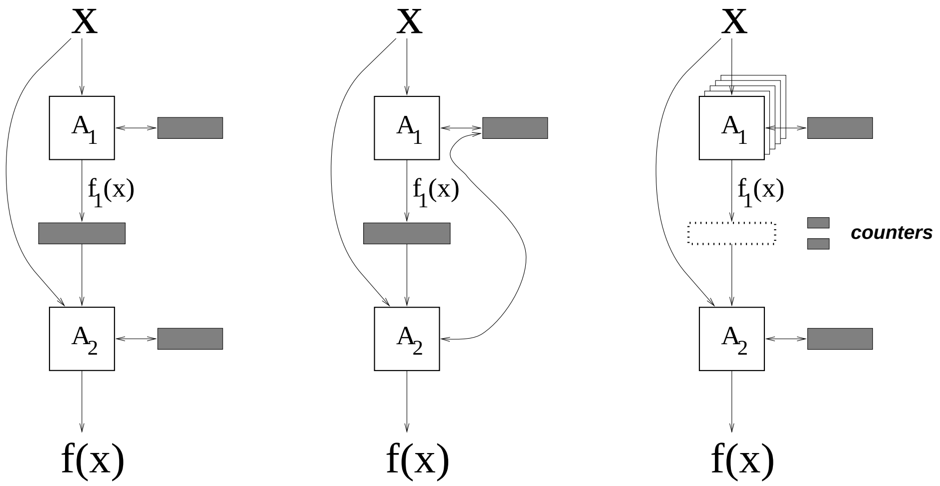
$$s(n) = s_1(n) + s_2(n + \ell(n)) + O(\log(n + \ell(n)))\,.$$

The alternative compositions are depicted in Figure 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 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)

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



The leftmost figure shows the trivial composition (which just invokes A_1 and A_2 without attempt to economize storage), the middle figure shows the naive composition (of Lemma 1), and the rightmost figure shows the emulative composition (of Lemma 2). In all figures the filled rectangles represent designated storage spaces. The dotted rectangle represents a virtual storage device.

Figure 1: Algorithmic composition for space-bounded computation

input tape. Whenever A_2 seeks to read the $i^{\rm th}$ 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 $(i-n)^{\rm th}$ output bit, which we then pass to A_2 (as the $i^{\rm th}$ 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., the bit-location currently read by A_2 and the index of the bit currently produced in the current invocation of A_1). ■

1.2.2 An obvious bound

The time complexity of an algorithm is essentially upper-bounded by an exponential in its space complexity. This is due to an upper-bound on the number of possible instantaneous "configurations" of the algorithm (as formulated in the proof of Theorem 3) and the fact that if the computation passes through the same configuration twice then it must loop forever.

Theorem 3 *If an algorithm A has binary space complexity s and halts on every input then it has time complexity t such that $t(n) = n \cdot 2^{s(n)+\log_2 s(n)}$.*

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

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

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

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

1.2.3 Subtleties regarding space-bounded reductions

Lemmas 1 and 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 1:)[3] 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 2:) For f and g as in Item 1, it follows that f is computable in space s such that $s(n) = s_1(n) + s_2(\ell(n)) + O(\log \ell(n))$.

Note that by Theorem 3, it holds that $\ell(n) \leq 2^{s_1(n)+\log_2 s_1(n)} \cdot n$. We stress the fact that ℓ is not bounded by s_1 (as in the analogous case of time-bounded computation), but rather by $\exp(s_1)$.

Things gets much more complicated when we turn to general (space-bounded) reductions, especially when referring to such reductions that make a non-constant number of queries. A preliminary issue is defining the space complexity of general reductions (i.e., of oracle machines). In the standard definition, the length of the queries and answers is not counted in the space complexity, but the queries of the reduction (resp., answers given to it) are written on (resp., read from) a special device that is write-only (resp., read-only) for the reduction (and read-only (resp., write-only) for the invoked oracle). Note that these convention are analogous to the conventions regarding input and output (as well as fit the definitions of space-bounded many-to-one reductions (see Section 2.2)). The rest of the discussion, which is quite advanced and laconic (and is inessential to the rest of the chapter), concerns two additional issues.

Recall that the complexity of the algorithm resulting from the composition of an oracle machine and an actual algorithm depends on the length of the queries made by the oracle machine. The length of the first query is upper-bounded by an exponential function in the space complexity of

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

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

With the aforementioned convention (or restriction) in place, let us consider the composition of *general* space-bounded reductions with a space-bounded implementation of the oracle. Specifically, we say that a reduction is (ℓ, ℓ')-restricted if, 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 1) remains valid, whereas the emulative composition of Lemma 2 breaks down (in the sense that it yield very weak results).

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

   A related composition result is presented in Exercise 15. It yields $s(n) = 2s_1(n) + s_2(\ell(n)) + 2\ell'(n) + \log_2 \ell(n)$, which for $\ell(n) < 2^{2s_1(n)}$ means $s(n) = 4s_1(n) + s_2(\ell(n)) + 2\ell'(n)$.

2. Turning to the approach underlying Lemma 2, we get into more serious trouble. Recomputing the answer of the i^{th} query requires recomputing the query itself, which unlike in Lemma 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 16).

An alternative notion of space-bounded reductions is presented in [4]. This notion is more cumbersome and more restricted, but it allows recursive composition with a smaller overhead than the two options explored above.

1.2.4 Complexity hierarchies and gaps

Recall that more space allows for more computation, 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.

1.2.5 Simultaneous time-space complexity

Recall that, for space complexity that is at least logarithmic, the time of a computation is always upper-bounded by an exponential function in the space complexity (see Theorem 3). Thus, polylogarithmic space complexity may extend beyond polynomial-time, and it make sense to define a class like $\mathcal{SC}$ that consists of all decision problems that may be solved by a polynomial-time algorithm of polylogarithmic space complexity. The class $\mathcal{SC}$ is indeed a natural sub-class of $\mathcal{P}$ (and contains the class $\mathcal{L}$, which is defined in Section 2.1).[4]

In general, one may define $\mathrm{DTiSp}(t,s)$ as the class of decision problems solvable by an algorithm that has time complexity t and space complexity s. Note that $\mathrm{DTiSp}(t,s) \subseteq \mathrm{Dtime}(t) \cap \mathrm{Dspace}(s)$ and that a strict containment may hold. Lower bounds on time-space trade-offs (see, e.g., [2, Sec. 4.3]) may be stated as referring to the classes $\mathrm{DTiSp}(\cdot,\cdot)$.

1.3 Circuit Evaluation

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

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

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

2 Logarithmic Space

Although Exercise 12 asserts that "there is life below log-space," logarithmic space seems to be the smallest amount of space that supports interesting computational phenomena. In fact, 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 3).

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, $\mathcal{L} = \cup_c \mathrm{Dspace}(\ell_c)$, where $\ell_c(n) \stackrel{\mathrm{def}}{=} c\log_2 n$. Note that, by Theorem 3, $\mathcal{L} \subseteq \mathcal{P}$. As hinted, many natural computational problems are in $\mathcal{L}$ (see Exercises 14 and 17 as well as Section 2.4). On the other hand, *it is widely believed that $\mathcal{L} \neq \mathcal{P}$.*

[4]We also mention that $\mathcal{BPL} \subseteq \mathcal{SC}$, where $\mathcal{BPL}$ is the "BPP analogue" of $\mathcal{L}$.

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 §1.2.3 we confine ourselves to the case of many-to-one reductions. Analogously to the definition of Karp-reductions, 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. Both types of reductions are transitive (see Exercise 18). Note that Theorem 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. For example, consider the generic reduction to CSAT presented in the proof of the NP-completeness of CSAT: The constructed circuit is "highly uniform" and can be easily constructed in logarithmic-space (see Section 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 4 (The complexity of Circuit Evaluation): *Let* CEVL *denote the set of pairs* (C, α) *such that* C *is a Boolean circuit and* $C(\alpha) = 1$. *Then* CEVL *is in* $\mathcal{P}$ *and every problem in* $\mathcal{P}$ *is log-space Karp-reducible to* CEVL.

Proof: Recall that the observation underlying the proof of the NP-completeness of CSAT is that the computation of a Turing machine can be emulated by a "highly uniform" circuit. In that proof one hardwires the input to the reduction (denoted x) into the circuit (denoted C_x) and introduced input terminals corresponding to the bits of the NP-witness (denoted y). In the current context we leave x as an input to the circuit, while noting that the auxiliary NP-witness does not exists (or has length zero). Thus, the reduction from $S \in \mathcal{P}$ to CEVL maps the instance x (for S) to the pair $(C_{|x|}, x)$, where $C_{|x|}$ is a circuit that emulates the computation of the machine that decides membership in S (on any $|x|$-bit long input). ■

The impact of P-completeness under log-space reductions. Indeed Theorem 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. [3]).

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 3.2). In general, we say that a problem Π is $\mathcal{C}$-complete under log-space reductions if Π is in $\mathcal{C}$ and every problem in $\mathcal{C}$ is log-space (many-to-one) reducible to Π. In such a case, if $\Pi \in \mathcal{L}$ then $\mathcal{C} \subseteq \mathcal{L}$.

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

2.3 Log-Space uniformity and stronger notions

Strengthening the standard definition of (polynomial-time) uniformity, 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 (and implicitly proved in Theorem 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 1.3, we note that *log-space uniform circuits of bounded fan-in and logarithmic depth can be emulated by an algorithm of logarithmic space complexity* (i.e., $\mathcal{NC}^1$ is in log-space; see Exercise 17).

Stronger notions of uniformity have been considered. Specifically, 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 4 actually establishes the following.

Theorem 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 $C_{|x|}(x) = A(x)$.*

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. In addition to the intrinsic interest in this natural computational problem, we note that related versions of the problem seem harder. For example, determining directed connectivity (in directed graphs) captures the essence of the class $\mathcal{NL}$ (see Section 3.2). In view of this situation, we emphasize the fact that the computational problem considered here refers to undirected graphs by calling it undirected connectivity.

Theorem 6 *Deciding undirected connectivity* (UCONN) *is in $\mathcal{L}$*

The starting point of the algorithm is the fact that any expander is easy to traverse in deterministic logarithmic-space, and thus the algorithm gradually transforms any graph into an expander, while maintaining the ability to map a traversal of the latter into a traversal of the former. Thus, the algorithm traverses a virtual graph, which being an expander is easy to traverse in deterministic logarithmic-space, and maps the virtual traversal of the virtual graph to a real traversal of the actual input graph. The virtual graph is constructed in (logarithmically many) iterations, where in each iteration the graph becomes easier to traverse. Specifically, in each iteration the expansion property of the graph improves by a constant factor, while the graph itself only grows by a constant factor, and each iteration can be performed (or rather emulated) in constant space. Since each graph has some noticeable expansion (i.e., expansion inversely related to the size of the graph), after logarithmically many steps this process yields a good expander (i.e., constant expansion). The details are presented in a seperate text [4], which uses a more abstract presentation than the one in the original work [7].

3 Non-Deterministic Space Complexity

The difference between space-complexity and time-complexity is quite striking in the context of non-deterministic computations. One aspect is the huge gap between the power of two formulation of non-deterministic space complexity (see Section 3.1), while in contrast the analogous formulations are equivalent in the context of time-complexity. Another aspect is the contrast between the results regarding (the standard model of) non-deterministic space-bounded computation (see Section 3.2) and the analogous questions in the context of time-complexity.

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), non-determinism is captured by reference to the existential choice of the contents of the auxiliary ("non-deterministic") input tape. In the on-line model (underlying the traditional definition of NP, non-determinism is captured by reference to the non-deterministic choices of the machine itself. These models are equivalent because the latter on-line choices can be recorded (almost) for free. That is, while it is clear that the off-line model can emulate the on-line model (i.e., the off-line machine can emulate on-line choices by using the contents of its non-deterministic input tape), the emulation of the off-line model by the on-line model is enabled by the fact that an on-line machine may store (and re-use) a sequence of non-deterministic (on-line) choices.

The naive emulation of the off-line model on the on-line model is not possible in the context of space-bounded computation, because the number of non-deterministic choices is typically much larger than the space-bound. The models become equivalent only if the off-line model is restricted to access its non-deterministic input tape in a uni-directional manner. Let us formulate the two models and consider them more closely.

In the standard model, called the on-line model, the machine makes non-deterministic choices "on the fly" (or alternatively reads a non-deterministic input from a 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 the so-called off-line model, the non-deterministic choices (or the bits of the non-deterministic input) are read from a special read-only device (or tape) *that can be scanned in both directions like the main input.*

We denote by $\textsc{Nspace}_{\text{on-line}}(s)$ (resp., $\textsc{Nspace}_{\text{off-line}}(s)$) 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 the traditional definition of $\mathcal{NP}$, 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, $\textsc{Nspace}_{\text{on-line}}(s) \subseteq \textsc{Nspace}_{\text{off-line}}(s)$. On the other hand, not only that $\textsc{Nspace}_{\text{on-line}}(s) \neq \textsc{Nspace}_{\text{off-line}}(s)$ but rather $\textsc{Nspace}_{\text{on-line}}(s) = \textsc{Nspace}_{\text{off-line}}(\Theta(\log s))$, provided that s is at least linear. For details, see Exercise 19.

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 $\mathcal{NP}$, 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 tape can be used to code computations (see Exercise 19), 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 $\textsc{Nspace}_{\text{off-line}}(s)$ is only known to be contained in $\textsc{Dtime}(2^{2^s})$, whereas $\textsc{Nspace}_{\text{on-line}}(s) \subseteq \textsc{Dtime}(2^s)$. 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 $\mathcal{NL} = \text{NSPACE}(\texttt{log})$.

3.2 NL and directed connectivity

This section is devoted to the study of $\mathcal{NL}$, which we view as the non-deterministic analogue of $\mathcal{L}$. Specifically, $\mathcal{NL} = \cup_c \text{NSPACE}(\ell_c)$, where $\ell_c(n) = c \log_2 n$ (and the definitional issues pertaining $\text{NSPACE} = \text{NSPACE}_{\text{on-line}}$ are discussed in Section 3.1).

We first note that the proof of Theorem 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 3), whereas this number bounds the time complexity. Thus, $\mathcal{NL} \subseteq \mathcal{P}$.

The following problem, called `directed connectivity` (`st-CONN`), captures the essence of non-deterministic log-space computations (and, in particular, is complete for $\mathcal{NL}$ under log-space reductions). The input to `st-CONN` consists of a directed graph $G = (V, E)$ and a pair of vertices (s, t), and the task is to determine whether there exists a directed path from s to t (in G).[5] 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`).

3.2.1 Completeness and beyond

Clearly, `st-CONN` is in $\mathcal{NL}$ (see Exercise 20). 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 21).

Theorem 7 *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, 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. Note 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 one corresponding to the start-point of the potential edge). Thus, the question of whether $x \in S$ is represented by the existence of a directed path from the vertex that corresponds to the initial configuration to the vertex that corresponds to a canonical accepting configuration. □

We believe that the proof of Theorem 7 (see also Exercise 21) justifies our choice to say that `st-CONN` captures the essence of non-deterministic log-space computations (rather than merely saying that `st-CONN` is $\mathcal{NL}$-complete under log-space reductions).

We note the discrepancy between the status of undirected connectivity (see Theorem 6) and directed connectivity (see Theorem 7). In this context it is worthwhile to note that determining the

[5] We note that, here (and in the sequel), s stands for *start* and t stands for *terminate*.

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 23.

3.2.2 Relating NSPACE to DSPACE

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

Theorem 8 (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., [5, Sec. 12.5]). The special case boils down to presenting a log-square space algorithm for deciding directed connectivity.

The basic idea is that checking whether or not there is a path of length at most ℓ from u to v in G, reduces (in log-space) to checking whether there is an intermediate vertex w such that there is a path of length at most $\lceil \ell/2 \rceil$ from u to w and a path of length at most $\lfloor \ell/2 \rfloor$ from w to v. Let $\phi_G(u,v,\ell) \stackrel{\text{def}}{=} 1$ if there is a path of length at most ℓ from u to v in G and $\phi_G(u,v,\ell) \stackrel{\text{def}}{=} 0$ otherwise. Thus, $\phi_G(u,v,\ell)$ can be decided recursively by scanning all vertices w in G, and checking for each w whether both $\phi_G(u,w,\lceil \ell/2 \rceil) = 1$ and $\phi_G(w,v,\lfloor \ell/2 \rfloor) = 1$ hold.

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

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

Placing NL in NC2. The simple formulation of `st-CONN` facilitates placing $\mathcal{NL}$ in complexity classes such as $\mathcal{NC}^2$. All that is needed is observing that `st-CONN` can be solved by raising the adequate matrix (i.e., the adjacency matrix of the graph augmented with 1-entries on the diagonal) to the adequate power (i.e., its dimension). Squaring a matrix can be done by a uniform family of logarithmic depth bounded fan-in circuits (i.e., in NC1), and by repeated squaring the n^{th} power of an n-by-n matrix can be computed by a uniform family of bounded fan-in circuits of polynomial size and depth $O(\log^2 n)$; thus, $\mathcal{NL} \subseteq \mathcal{NC}^2$.

3.2.3 Complementation or NL=coNL

Recall that (natural) non-deterministic time-complexity classes are not known to be closed under complementation. Furthermore, it is widely believed that $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$. In contrast, (natural) non-deterministic space-complexity classes are closed under complementation, as captured by the result $\mathcal{NL} = \mathrm{co}\mathcal{NL}$, where $\mathrm{co}\mathcal{NL} \stackrel{\rm def}{=} \{\{0,1\}^* \setminus S : S \in \mathcal{NL}\}$. Before proving this result, let us take a closer look at the problem it resolves (which we rephrase as $\mathcal{NL}$ versus $\mathcal{NL} \cap \mathrm{co}\mathcal{NL}$).

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

Definition 9 (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 $S \in \mathcal{NL} \cap \mathrm{co}\mathcal{NL}$ if and only if there exists a non-deterministic log-space machine that computes χ_S, where $\chi_S(x) = 1$ if $x \in S$ and $\chi_S(x) = 0$ otherwise (see Exercise 25). It follows that $\mathcal{NL} = \mathrm{co}\mathcal{NL}$ if and only if for every $S \in \mathcal{NL}$ there exists a non-deterministic log-space machine that computes χ_S.

Theorem 10 ($\mathcal{NL} = \mathrm{co}\mathcal{NL}$): *For every $S \in \mathcal{NL}$ there exists a non-deterministic log-space machine that computes χ_S.*

As in the case of Theorem 8, 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 $S \in \textsc{Nspace}(s)$, it holds that $\{0,1\}^* \setminus S \in \textsc{Nspace}(O(s))$. This extension can be proved either by generalizing the following proof or by using an adequate upwards-translation lemma.

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

We first show that the computation of χ is log-space reducible (by two queries)[6] to determining the number of vertices that are reachable from a given vertex in a given graph. On input (G,s,t),

[6] We stress the fact that two queries are used in the reduction, because this avoids the difficulties (discussed in §1.2.3) regarding emulative composition for general space-bounded reduction. Alternatively, we may use a version of the naive composition, while relying on the fact that the oracle queries are "highly related" to the input and that the answers have logarithmic length. For details, see Exercises 28 and 29.

the reduction computes the number of vertices that are reachable from s in the graph G and compares it to the number of vertices reachable from s in the graph obtained by deleting t from G. (An alternative reduction that uses a single query is presented in Exercise 27.) 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 28). 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

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

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

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

1. We verify that $|R_i| \geq g$ in a straightforward manner. That is, scanning V in some canonical order, we verify for g vertices that they are each in R_i. That is, during the scan, we select non-deterministically g vertices, and for each selected vertex w we verify that w is reachable from v by a path of length at most i, where this verification is performed by just guessing and verifying an adequate path (see Exercise 20).

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

2. The verification of the condition $|R_i| \leq g$ is the interesting part of the procedure. Here we rely 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. (1)). Details follows (and an even more structured description is provided in Figure 2).

   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. (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).[7]

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

[7]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.

By Eq. (1), if u passes the foregoing verification then indeed $u \notin R_i$.

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

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

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

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

Figure 2: The main step in proving $\mathcal{NL} = \mathrm{co}\mathcal{NL}$.

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

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

4 PSPACE and Games

As stated up-front, we will rarely treat computational problems that require less than logarithmic space. On the other hand, we will rarely treat computational problems that require more than polynomial space. The class of decision problems that are solvable in polynomial-space is denoted $\mathcal{PSPACE} \stackrel{\rm def}{=} \cup_c \mathrm{DSPACE}(p_c)$, 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 the definition of Σ_k. In fact, such an algorithm can handle an unbounded number of alternating quantifiers (see Theorem 11). On the other hand, by Theorem 3, $\mathcal{PSPACE} \subseteq \mathcal{EXP}$, where $\mathcal{EXP} = \cup_c \mathrm{DTIME}(2^{p_c})$ 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 the context of the Polynomial-Time hierarchy. Recall that we refer to two-party games that satisfy the following three conditions:

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

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

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

A set $S \in \mathcal{PSPACE}$ can be viewed as the set of initial positions (in a suitable game) for which the first party has a winning strategy *consisting of a polynomial number of moves*. Specifically, $x \in S$ if starting at the initial position x, there exists move y_1 for the first party, such that for every response move y_2 of the second party, there exists move y_3 for the first party, etc, such that after $\mathrm{poly}(|x|)$ 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 11, which refers to the satisfiability of quantified Boolean formulae (QBF).

Theorem 11 QBF *is complete for $\mathcal{PSPACE}$ under polynomial-time many-to-one reductions.*

Proof: As note before, QBF is solvable by a polynomial-space algorithm that just evaluates the quantified formula. Specifically, consider a recursive procedure that eliminates a Boolean quantifier by evaluating the value of the two residual formulae, and note that the space used in the first (recursive) evaluation can be re-used in the second evaluation. (Alternatively, consider a DFS-type procedure as in Section 1.3.) Note that the space used is linear in the depth of the recursion, which in turn is linear in the length of the input formula.

We now turn to show that any set $S \in \mathcal{PSPACE}$ is many-to-one reducible to QBF. The proof is similar to the proof of Theorem 8, 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 21. Actually, here we use a different notion of a configuration that *includes also the input*. That is, in the rest of this proof, *a configuration consists of the contents of all storage devices of the algorithm* (including the input device) as well as the location of the algorithm on each device.

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

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

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. (2) is that the expression for $\phi_G(\cdot,\cdot,2\ell)$ involves two occurrences of $\phi_G(\cdot,\cdot,\ell)$, which doubles the length of the recursively constructed formula (yielding an exponential blow-up).

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

$$[(\sigma=0)\Rightarrow(u'=u\wedge v'=w)]\ \wedge\ [(\sigma=1)\Rightarrow(u'=w\wedge v'=v)]. \tag{3}$$

Thus, $\phi_G(u,v,2\ell)$ holds if and only if there exist w such that for every σ there exists (u',v') such that both Eq. (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 $m=\text{poly}(|x|)$ are as above), and Φ is recursively defined as follows

$$\Phi(u,v,2\ell)\ \stackrel{\text{def}}{=}\ \begin{array}{l}\exists w\in\{0,1\}^m\ \forall\sigma\in\{0,1\}\exists u',v'\in\{0,1\}^m\\ \qquad[(\sigma=0)\Rightarrow(u'=u\wedge v'=w)]\\ \qquad\wedge[(\sigma=1)\Rightarrow(u'=w\wedge v'=v)]\\ \qquad\wedge\ \Phi(u',v',\ell)\end{array} \tag{4}$$

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

We stress that the mapping of x to $\Phi(s_x,t,2^m)$ can be computed in polynomial-time. Firstly, note that the propositional formula $\Phi(u,v,1)$, having Boolean variables representing the bits of u

and v, expresses extremely simple conditions and can certainly be constructed in polynomial-time (i.e., polynomial in the number of Boolean variables, which in turn equals $2m$). Next note that, given $\Phi(u, v, \ell)$, which (for $\ell > 1$) contains quantified variables that are not shown in the notation, we can construct $\Phi(u, v, 2\ell)$ by merely replacing variables names and adding quantifiers and Boolean conditions as in the recursive definition of Eq. (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 $m = \text{poly}(|x|)$, 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 [9, Sec. 8.3]). This further justifies the title of the current section.

Notes

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

We first mention the "classical" results. These include the $\mathcal{NL}$-completeness of `st-CONN`, the emulation of non-deterministic space-bounded machines by deterministic space-bounded machines (i.e., Theorem 8 due to Savitch [8]), the $\mathcal{PSPACE}$-completeness of `QBF`, and the connections between circuit depth and space complexity (see Section 1.3 and Exercise 17 due to Borodin [1]).

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

The question of whether $\mathcal{NL} = \text{co}\mathcal{NL}$ was a famous open problem for almost two decades. Furthermore, this question is related to an even older open problem dating to the early days of research in the area of formal languages (i.e., to the 1950's).[8] This open problem was resolved in 1988 by Immerman [6] and Szelepcsenyi [10], who (independently) proved Theorem 10 (i.e., $\mathcal{NL} = \text{co}\mathcal{NL}$).

For more than two decades, undirected connectivity was one of the most appealing examples of the computational power of randomness. Recall that the classical (deterministic) linear-time algorithms (e.g., BFS and DFS) require an extensive use of (extra) memory (i.e., space linear in the size of the graph). On the other hand, it was known (since 1979) that, with high probability, a random walk of polynomial length visits all vertices (in the corresponding connected component). Thus, the randomized algorithm requires a minimal amount of auxiliary memory (i.e., logarithmic in the size of the graph). In the early 1990's, this algorithm (as well as the entire class $\mathcal{BPL}$) was

[8] Specifically, the class of sets recognized by linear-space non-deterministic machines equals the class of context-sensitive languages (see, e.g., [5, Sec. 9.3]), and thus Theorem 10 resolves the question of whether the latter class is closed under complementation.

derandomized in polynomial-time and poly-logarithmic space, 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 [7], who established Theorem 6 in 2004.

Exercises

Exercise 12 (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 $S \stackrel{\rm def}{=} \{w_k : k \in \mathbb{N}\} \subset \{0,1,*\}$ is not regular and yet is decidable in double-logarithmic space.

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

Exercise 13 (on the weakness of less than double-logarithmic space) Prove that for $\ell(n) = \log\log n$, it holds that $\textsc{Dspace}(o(\ell)) = \textsc{Dspace}(O(1))$.

Guideline: Let s denote the machine's (binary) space complexity (see Footnote 1). Assuming that s is unbounded, consider for each m the shortest string x such that on input x the machine uses space at least m. Consider, for each location on the input, the sequence of the residual configurations of the machine (i.e., the contents of its temporary storage)[9] at the times in which the machine crosses (or rather passes through) this input location. For starters, note that the length of this "crossing sequence" is upper-bounded by the number of possible configurations, which is at most $t \stackrel{\rm def}{=} 2^{s(|x|)} \cdot s(|x|)$. Thus, the number of such crossing sequences is upper-bounded by $(t+1)! < t^t$. Now, if $t^t < |x|/2$ then there exist three input locations that have the same crossing sequence, and two of them hold the same bit value. Contracting the string at these two locations, we get a shorter input on which the machine behaves in exactly the same manner, contradicting the hypothesis that x is the shortest input on which the machine uses space at least m. We conclude that $t^t \geq |x|/2$ must hold, and $s(|x|) = \Omega(\log t) = \Omega(\log\log|x|)$ follows.

Exercise 14 (some log-space algorithms) Present log-space algorithms for the following computational problems.

1. Addition and multiplication of a pair of integers.

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

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

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

   Guideline: Consider a scanning of the graph that proceeds as follows. Upon entering a vertex v via the $i^{\rm th}$ edge incident at it, we exit this vertex using its $i+1^{\rm st}$ if v has degree at least $i+1$ and exit

[9] Note that, unlike in the proof of Theorem 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.

via the first edge otherwise. Note that when started at any vertex of any tree, this scanning performs a DFS. On the other hand, for every cyclic graph there exists a vertex v and an edge e incident to v such that if this scanning is started by traversing the edge e from v then it returns to v via an edge different from e.

4. Deciding whether the input graph is a tree.

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

Exercise 15 (another composition result) In continuation to the discussion in §1.2.3, prove that *if* Π *can be computed in space* s_1 *when given an* (ℓ,ℓ')*-restricted oracle access to* Π' *and* Π' *is solvable is space* s_2*, then* Π *is solvable in space* s *such that* $s(n)=2s_1(n)+s_2(\ell(n))+2\ell'(n)+\log_2\ell(n)$.

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

Exercise 16 Referring to the discussion in §1.2.3, prove that any problem having space complexity s can be solved by a constant-space $(2s,2s)$-restricted reduction to a problem that is solvable in constant-space.

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

Exercise 17 (log-space uniform $\mathcal{NC}^1$ is in $\mathcal{L}$) Suppose that a problem Π is solvable by a family of log-space uniform bounded fan-in circuits of depth d such that $d(n)\geq\log n$. Prove that Π is solvable by an algorithm having space complexity $O(d)$.

Guideline: Combine the algorithm outlined in Section 1.3 with the definition of log-space uniformity (using Lemma 2).

Exercise 18 (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 2, noting that such reductions are merely log-space computable functions.

Exercise 19 (relating the two models of NSPACE) Referring to the definitions in Section 3.1, prove that for every function s such that $\log s$ is space contructible 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 tape of the off-line machine for encoding an accepting computation of the on-line machine; that is, this tape should contain a sequence of consecutive configurations leading from the initial configuration to an accepting configuration, where each configuration contains the contents of the work-tape as well as the machine's location on the work-tape and on the input tape. The emulating off-line machine (which verifies the correctness of the sequence of configurations recorded on its non-deterministic input tape) needs only store its *location within the current pair of consecutive configurations* that it examines, which requires space logarithmic in the length of a single configuration (which in turn equals $s(n) + \log_2 s(n) + \log_2 n$). (Note that this verification relies on a two-directional access to the non-deterministic input tape.)

Guideline (for $\text{NSPACE}_{\text{off-line}}(s') \subseteq \text{NSPACE}_{\text{on-line}}(\exp(s'))$): Here we refer to the notion of a crossing-sequence. Specifically, for each location on the off-line non-deterministic tape, 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 13). 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 20 (st-CONN and variants of it are in NL) Prove that the following computational problem is in $\mathcal{NL}$. The instances have the form (G, v, w, ℓ), where $G = (V, E)$ is a directed graph, $v, w \in V$, and ℓ is an integer, and the question is whether G contains a path of length at most ℓ from v to w.

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

Exercise 21 (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 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 7.

Exercise 22 (an alternative presentation of the proof of Theorem 8) 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 21), 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 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 23 (finding shortest paths in undirected graphs) Prove that the following computational problem is $\mathcal{NL}$-complete under (many-to-one) log-space reductions: Given an undirected graph $G = (V, E)$, two designated vertices, s and t, and an integer K, determine whether there is a path of length at most K from s to t in G.

Guideline (for $\mathcal{NL}$-hardness): Reduce from `st-CONN`. Specifically, given a directed graph $G = (V, E)$ and vertices s, t, consider a ("layered") graph $G' = (V', E')$ such that $V' = \cup_{i=0}^{|V|-1}\{\langle i, v\rangle : v \in V\}$ and $E' = \cup_{i=0}^{|V|-2}\{\{\langle i, u\rangle, \langle i+1, v\rangle\} : (u, v) \in E \vee u = v\}$. Note that there exists a directed path from s to t in G if and only if there exists a path of length $|V| - 1$ between $\langle s, 0\rangle$ and $\langle t, |V| - 1\rangle$ in G'.

Exercise 24 (deciding strong connectivity) A directed graph is called strongly connected if there exists a directed path between every ordered pair of vertices in the graph (or, equivalently, a directed cycle passing through every two vertices). Prove that the problem of deciding whether a directed graph is strongly connected is $\mathcal{NL}$-complete under (many-to-one) log-space reductions.

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

Exercise 25 (an operational interpretation of $\mathcal{NL} \cap \mathrm{co}\mathcal{NL}$, $\mathcal{NP} \cap \mathrm{co}\mathcal{NP}$, etc) Referring to Definition 9, prove that $S \in \mathcal{NL} \cap \mathrm{co}\mathcal{NL}$ if and only if there exists a non-deterministic log-space machine that computes χ_S, where $\chi_S(x) = 1$ if $x \in S$ and $\chi_S(x) = 0$ otherwise. State and prove an analogous result for $\mathcal{NP} \cap \mathrm{co}\mathcal{NP}$.

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

Exercise 26 (a graph algorithmic interpretation of $\mathcal{NL} = \mathrm{co}\mathcal{NL}$) Show that there exists a log-space computable function f such that for every (G, s, t) it holds that (G, s, t) is a yes-instance of `st-CONN` if and only if $(G', s', t') = f(G, s, t)$ is a no-instance of `st-CONN`.

Exercise 27 As an alternative to the two-query reduction presented in the proof of Theorem 10, 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 28 (reductions and non-deterministic computations) Suppose that computing f is log-space reducible by a constant number of queries to computing some function g. Referring to the non-deterministic computations as in Definition 9, 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 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 29 (reductions and non-deterministic computations, revisited) Suppose that computing f is log-space reducible (by any number of queries) to computing some function g such that for every x it holds that $|g(x)| = O(\log|x|)$. Referring to the non-deterministic computations as in Definition 9, prove that if there exists a non-deterministic log-space machine that computes g then there exists a non-deterministic log-space machine that computes f. As a warm-up consider the special case in which every query to g is computable in log-space based on the input to f.

Guideline: As in Exercise 28, except that here we use different composition techniques. Specifically, in the warm-up we use the naive composition (as in Lemma 1), whereas in the general case we use the semi-naive composition of Exercise 15.

Exercise 30 Prove that the problem of determining whether or not the input graph is bipartite (2-colorable) is in $\mathcal{NL}$.

Guideline: A graph is bipartite if and only if it contains no odd-length cycles. (Extra hint: Use $\mathcal{NL} = \mathrm{co}\mathcal{NL}$.)

Exercise 31 Referring to Definition 9, 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 23, and use $\mathcal{NL} = \mathrm{co}\mathcal{NL}$.

References

[1] A. Borodin. On Relating Time and Space to Size and Depth. *SIAM Journal on Computing*, Vol. 6 (4), pages 733–744, 1977.

[2] S.A. Cook. A overview of Computational Complexity. Turing Award Lecture. *CACM*, Vol. 26 (6), pages 401–408, 1983.

[3] S.A. Cook. A Taxonomy of Problems with Fast Parallel Algorithms. *Information and Control*, Vol. 64, pages 2–22, 1985.

[4] O. Goldreich. Proving that Undirected Connectivity is in $\mathcal{L}$ (with a long appendix on expander graphs). Unpublished note, December 2005. Availabe from the webpage `http://www.wisdom.weizmann.ac.il/~oded/cc-texts.html`

[5] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[6] N. Immerman. Nondeterministic Space is Closed Under Complementation. *SIAM Journal on Computing*, Vol. 17, pages 760–778, 1988.

[7] O. Reingold. Undirected ST-Connectivity in Log-Space. In *37th ACM Symposium on the Theory of Computing*, pages 376–385, 2005.

[8] W.J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *JCSS*, Vol. 4 (2), pages 177-192, 1970.

[9] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[10] R. Szelepcsenyi. A Method of Forced Enumeration for Nondeterministic Automata. *Acta Informatica*, Vol. 26, pages 279–284, 1988.