Computational Complexity: A Modern Approach

Draft of a book: Dated January 2007
`Comments welcome!`

Sanjeev Arora and Boaz Barak
Princeton University
complexitybook@gmail.com

This is an Internet draft. Some chapters are more finished than others. References and attributions are very preliminary and we apologize in advance for any omissions (but hope you will nevertheless point them out to us).

Please send us bugs, typos, missing references or general comments to complexitybook@gmail.com — **Thank You!!**

Chapter 4

Space complexity

"(our) construction... also suggests that what makes "games" harder than "puzzles" (e.g. **NP***-complete problems) is the fact that the initiative ("the move") can shift back and forth between the players."*
Shimon Even and Robert Tarjan, 1976

In this chapter we will study the memory requirements of computational tasks. To do this we define *space-bounded computation,* which has to be performed by the TM using a restricted number of tape cells, the number being a function of the input size. We also study *nondeterministic space-bounded TMs.* As in the chapter on **NP**, our goal in introducing a complexity class is to "capture" interesting computational phenomena— in other words, identify an interesting set of computational problems that lie in the complexity class and are *complete* for it. One phenomenon we will "capture" this way (see Section 4.3.2) concerns computation of winning strategies in 2-person games, which seems inherently different from (and possibly more difficult than) solving **NP** problems such as SAT, as alluded to in the above quote. The formal definition of deterministic and non-deterministic space bounded computation is as follows (see also Figure 4.1):

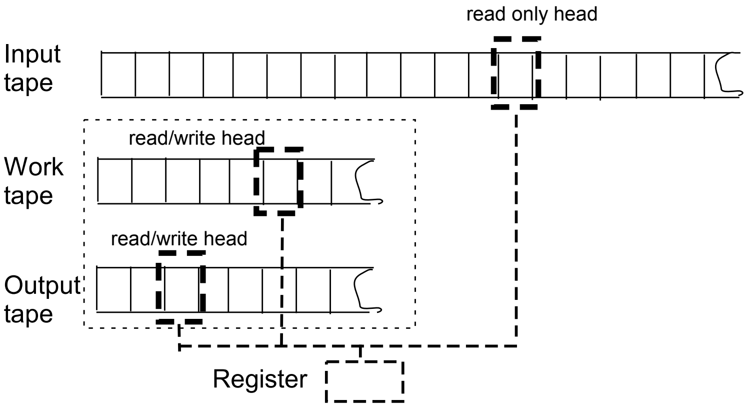

Figure 4.1: Space bounded computation. Only cells used in the read/write tapes count toward the space bound.

Definition 4.1 (Space-bounded computation.)
Let $S : \mathbb{N} \to \mathbb{N}$ and $L \subseteq \{0,1\}^*$. We say that $L \in \mathbf{SPACE}(s(n))$ (resp. $L \in \mathbf{NSPACE}(s(n))$) if there is a constant c and TM (resp. NDTM) M deciding L such that on every input $x \in \{0,1\}^*$, the total number of locations that are at some point non-blank during M's execution on x is at most $c \cdot s(|x|)$. (Non-blank locations in the read-only input tape do not count.)

As in our definitions of all nondeterministic complexity classes, we require all branches of nondeterministic machines to always halt.

Remark 4.2
Analogously to time complexity, we will restrict our attention to space bounds $S : \mathbb{N} \to \mathbb{N}$ that are *space-constructible* functions, by which we mean that there is a TM that computes $S(n)$ in $O(S(n))$ space when given 1^n as input. (Intuitively, if S is space-constructible, then the machine “knows” the space bound it is operating under.) This is a very mild restriction since functions of interest, including $\log n$,n and 2^n, are space-constructible.

Also, realize that since the work tape is separated from the input tape, it makes sense to consider space-bounded machines that use space less than the input length, namely, $S(n) < n$. (This is in contrast to time-bounded computation, where $\mathbf{DTIME}(T(n))$ for $T(n) < n$ does not make much sense since the TM does not have enough time to read the entire input.) We will assume however that $S(n) > \log n$ since the work tape has length n, and we would like the machine to at least be able to “remember” the index of the cell of the input tape that it is currently reading. (One of the exercises explores classes that result when $S(n) \ll \log n$.)

Note that $\mathbf{DTIME}(S(n)) \subseteq \mathbf{SPACE}(S(n))$ since a TM can access only one tape cell per step. Also, notice that space can be *reused*: a cell on the work tape can be overwritten an arbitrary number of times. A space $S(n)$ machine can easily run for as much as $2^{\Omega(S(n))}$ steps —think for example of the machine that uses its work tape of size $S(n)$ to maintain a counter which it increments from 1 to $2^{S(n)-1}$. The next easy theorem (whose proof appears a little later) shows that this is tight in the sense that any languages in $\mathbf{SPACE}(S(n))$ (and even $\mathbf{NSPACE}(S(n))$) is in $\mathbf{DTIME}(2^{O(S(n))})$. Surprisingly enough, up to logarithmic terms, this theorem contains the only relationships we know between the power of space-bounded and time-bounded computation. Improving this would be a major result.

Theorem 4.3
For every space constructible $S : \mathbb{N} \to \mathbb{N}$,

$$\mathbf{DTIME}(S(n)) \subseteq \mathbf{SPACE}(S(n)) \subseteq \mathbf{NSPACE}(S(n)) \subseteq \mathbf{DTIME}(2^{O(S(n))})$$

4.1 Configuration graphs.

To prove Theorem 4.3 we use the notion of a *configuration graph* of a Turing machine. This notion will also be quite useful for us later in this chapter and the book. Let M be a (deterministic or

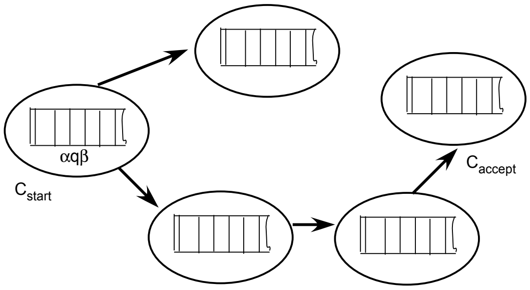

Figure 4.2: The configuration graph $G_{M,x}$ is the graph of all configurations of M's execution on x where there is an edge from a configuration C to a configuration C' if C' can be obtained from C in one step. It has out-degree one if M is deterministic and out-degree at most two if M is non-deterministic.

non-deterministic) TM. A *configuration* of a TM M consists of the contents of all non-blank entries of M's tapes, along with its state and head position, at a particular point in its execution. For every TM M and input $x \in \{0,1\}^*$, the *configuration graph of M on input x*, denoted $G_{M,x}$, is a directed graph whose nodes correspond to possible configurations that M can reach from the starting configuration C^x_{start} (where the input tape is initialized to contain x). The graph has a directed edge from a configuration C to a configuration C' if C' can be reached from C in one step according to M's transition function (see Figure 4.2). Note that if M is deterministic then the graph has out-degree one, and if M is non-deterministic then it has an out-degree at most two. Also note that we can assume that M's computation on x does not repeat the same configuration twice (as otherwise it will enter into an infinite loop) and hence that the graph is a directed acyclic graph (DAG). By modifying M to erase all its work tapes before halting, we can assume that there is only a single configuration C_{accept} on which M halts and outputs 1. This means that M accepts the input x iff there exists a (directed) path in $G_{M,x}$ from C_{start} to C_{accept}. We will use the following simple claim about configuration graphs:

Claim 4.4
Let $G_{M,x}$ be the configuration graph of a space-$S(n)$ machine M on some input x of length n. Then,

1. *Every vertex in $G_{M,x}$ can be described using $cS(n)$ bits for some constant c (depending on M's alphabet size and number of tapes) and in particular, $G_{M,x}$ has at most $2^{cS(n)}$ nodes.*

2. *There is an $O(S(n))$-size CNF formula $\varphi_{M,x}$ such that for every two strings C, C', $\varphi_{M,x}(C,C') = 1$ if and only if C, C' encode two neighboring configuration in $G_{M,x}$.*

Proof sketch: Part 1 follows from observing that a configuration is completely described by giving the contents of all work tapes, the position of the head, and the state that the TM is in (see Section 1.2.1). We can encode a configuration by first encoding the snapshot (i.e., state and current symbol read by all tapes) and then encoding in sequence the non-blank contents of all the work-tape, inserting a special "marker" symbol, to denote the locations of the heads.

Part 2 follows using similar ideas as in the proof of the Cook-Levin theorem (Theorem 2.10). There we showed that deciding whether two configurations are neighboring can be expressed as the AND of many checks, each depending on only a constant number of bits, where such checks can be expressed by constant-sized CNF formulae by Claim 2.14. ■

Now we can prove Theorem 4.3.

Proof of Theorem 4.3: Clearly $\mathbf{SPACE}(S(n)) \subseteq \mathbf{NSPACE}(S(n))$ and so we just need to show $\mathbf{NSPACE}(S(n)) \subseteq \mathbf{DTIME}(2^{O(S(n))})$. By enumerating over all possible configurations we can construct the graph $G_{M,x}$ in $2^{O(S(n))}$-time and check whether C_{start} is connected to C_{accept} in $G_{M,x}$ using the standard (linear in the size of the graph) breadth-first search algorithm for connectivity (e.g., see [**?**]). ■

We also note that there exists a universal TM for space bounded computation analogously to Theorems 1.13 and **??** for deterministic and non-deterministic time bounded computation, see Section **??** below.

4.2 Some space complexity classes.

Now we define some complexity classes, where **PSPACE**, **NPSPACE** are analogs of **P** and **NP** respectively.

Definition 4.5
$\mathbf{PSPACE} = \cup_{c>0}\mathbf{SPACE}(n^c)$
$\mathbf{NPSPACE} = \cup_{c>0}\mathbf{NSPACE}(n^c)$
$\mathbf{L} = \mathbf{SPACE}(\log n)$
$\mathbf{NL} = \mathbf{NSPACE}(\log n)$

Example 4.6
We show how $\mathsf{3SAT} \in \mathbf{PSPACE}$ by describing a TM that decides $\mathsf{3SAT}$ in linear space (that is, $O(n)$ space, where n is the size of the $\mathsf{3SAT}$ instance). The machine just uses the linear space to cycle through all 2^k assignments in order, where k is the number of variables. Note that once an assignment has been checked it can be erased from the worktape, and the worktape then reused to check the next assignment. A similar idea of cycling through all potential certificates applies to any **NP** language, so in fact $\mathbf{NP} \subseteq \mathbf{PSPACE}$.

Example 4.7
The reader should check (using the gradeschool method for arithmetic) that the following languages are in **L**:

$$\mathsf{EVEN} = \{x : \ x \text{ has an even number of 1s}\}\,.$$

$$\mathsf{MULT} = \{(\llcorner n \lrcorner, \llcorner m \lrcorner, \llcorner nm \lrcorner) : n \in \mathbb{N}\}\,.$$

Its seems difficult to conceive of any complicated computations apart from arithmetic that use only $O(\log n)$ space. Nevertheless, we cannot currently even rule out that 3SAT $\in$ **L** (in other words —see the exercises— it is open whether **NP** $\neq$ **L**). Space-bounded computations with space $S(n) \ll n$ seem relevant to computational problems such as *web crawling*. The world-wide-web may be viewed crudely as a directed graph, whose nodes are webpages and edges are hyperlinks. Webcrawlers seek to explore this graph for all kinds of information. The following problem PATH is natural in this context:

$$\mathsf{PATH} = \{\langle G, s, t\rangle : \ G \text{ is a directed graph in which there is a path from } s \text{ to } t\} \qquad (1)$$

We claim that PATH $\in$ **NL**. The reason is that a nondeterministic machine can take a "nondeterministic walk" starting at s, always maintaining the index of the vertex it is at, and using nondeterminism to select a neighbor of this vertex to go to next. The machine accepts iff the walk ends at t in at most n steps, where n is the number of nodes. If the nondeterministic walk has run for n steps already and t has not been encountered, the machine rejects. The work tape only needs to hold $O(\log n)$ bits of information at any step, namely, the number of steps that the walk has run for, and the identity of the current vertex.

Is PATH in **L** as well? This is an open problem, which, as we will shortly see, is equivalent to whether or not **L** = **NL**. That is, PATH captures the "essence" of **NL** just as 3SAT captures the "essence" of **NP**. (Formally, we will show that PATH is **NL**-complete.) A recent surprising result shows that the restriction of PATH to *undirected* graphs is in **L**; see Chapters 7 and 16.

4.3 PSPACE completeness

As already indicated, we do not know if **P** $\neq$ **PSPACE**, though we strongly believe that the answer is YES. Notice, **P** = **PSPACE** implies **P** = **NP**. Since complete problems can help capture the essence of a complexity class, we now present some complete problems for **PSPACE**.

Definition 4.8
A language A is **PSPACE**-*hard* if for every $L \in$ **PSPACE**, $L \leq_p A$. If in addition $A \in$ **PSPACE** then A is **PSPACE**-*complete*.

Using our observations about polynomial-time reductions from Chapter **??** we see that if any **PSPACE**-complete language is in **P** then so is every other language in **PSPACE**. Viewed contrapostively, if **PSPACE** $\neq$ **P** then a **PSPACE**-complete language is not in **P**. Intuitively, a **PSPACE**-complete language is the "most difficult" problem of **PSPACE**. Just as **NP** trivially contains **NP**-complete problems, so does **PSPACE**. The following is one (Exercise 3):

$$\mathsf{SPACETM} = \{\langle M, w, 1^n\rangle : \text{DTM } M \text{ accepts } w \text{ in space } n\}\,. \qquad (2)$$

Now we see some more interesting **PSPACE**-complete problems. We use the notion of a *quantified boolean formula*, which is a boolean formula in which variables are *quantified* using $\exists$

and $\forall$ which have the usual meaning "there exists" and "for all" respectively. It is customary to also specify the universe over which these signs should be interpreted, but in our case the universe will always be the truth values $\{0,1\}$. Thus a *quantified boolean formula* has the form $Q_1x_1Q_2x_2\cdots Q_nx_n\varphi(x_1,x_2,\ldots,x_n)$ where each Q_i is one of the two quantifiers $\forall$ or $\exists$ and φ is an (unquantified) boolean formula[1].

If all variables in the formula are quantified (in other words, there are no free variables) then a moment's thought shows that such a formula is either *true* or *false* —there is no "middle ground". We illustrate the notion of truth by an example.

EXAMPLE 4.9

Consider the formula $\forall x\exists y\ (x\wedge y)\vee(\overline{x}\wedge\overline{y})$ where $\forall$ and $\exists$ quantify over the universe $\{0,1\}$. Some reflection shows that this is saying "for every $x\in\{0,1\}$ there is a $y\in\{0,1\}$ that is different from it", which we can also informally represent as $\forall x\exists y(x\neq y)$. This formula is *true.* (Note: the symbols $=$ and $\neq$ are not logical symbols per se, but are used as informal shorthand to make the formula more readable.)

However, switching the second quantifier to $\forall$ gives $\forall x\forall y\ (x\wedge y)\vee(\overline{x}\wedge\overline{y})$, which is *false.*

EXAMPLE 4.10

Recall that the SAT problem is to decide, given a Boolean formula φ that has n free variables $x_1,\ldots,x_n$, whether or not φ has a satisfying assignment $x_1,\ldots,x_n\in\{0,1\}^n$ such that $\varphi(x_1,\ldots,x_n)$ is true. An equivalent way to phrase this problem is to ask whether the *quantified* Boolean formula $\psi=\exists x_1,\ldots,x_n\varphi(x_1,\ldots,x_n)$ is true.

The reader should also verify that the *negation* of the formula
$Q_1x_1Q_2x_2\cdots Q_nx_n\varphi(x_1,x_2,\ldots,x_n)$ is the same as

$$Q'_1x_1Q'_2x_2\cdots Q'_nx_n\neg\varphi(x_1,x_2,\ldots,x_n),$$

where Q'_i is $\exists$ if Q_i was $\forall$ and vice versa. The switch of $\exists$ to $\forall$ in case of SAT gives instances of TAUTOLOGY, the **coNP**-complete language encountered in Chapter **??**.

We define the language TQBF to be the set of quantified boolean formulae that are true.

[1] We are restricting attention to quantified boolean formulae which are in *prenex normal form*, i.e., all quantifiers appear to the left. However, this is without loss of generality since we can transform a general formula into an equivalent formula in prenex form in polynomial time using identities such as $p\vee\exists x\varphi(x)=\exists xp\vee\varphi(x)$ and $\neg\forall x\phi(x)=\exists x\neg\phi(x)$. Also note that unlike in the case of the SAT and 3SAT problems, we do not require that the inner unquantified formula φ is in CNF or 3CNF form. However this choice is also not important, since using auxiliary variables in a similar way to the proof of the Cook-Levin theorem, we can in polynomial-time transform a general quantified Boolean formula to an equivalent formula where the inner unquantified formula is in 3CNF form.

Theorem 4.11
*TQBF is **PSPACE**-complete.*

Proof: First we show that TQBF $\in$ **PSPACE**. Let

$$\psi = Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \varphi(x_1, x_2, \ldots, x_n) \tag{3}$$

be a quantified Boolean formula with n variables, where we denote the size of φ by m. We show a simple recursive algorithm A that can decide the truth of ψ in $O(n+m)$ space. We will solve the slightly more general case where, in addition to variables and their negations, φ may also include the constants 0 (i.e., "false") and 1 (i.e., "true"). If $n = 0$ (there are no variables) then the formula contains only constants and can be evaluated in $O(m)$ time and space. Let $n > 0$ and let ψ be as in (3). For $b \in \{0,1\}$, denote by $\psi_{\restriction x_1 = b}$ the modification of ψ where the first quantifier Q_1 is dropped and all occurrences of x_1 are replaced with the constant b. Algorithm A will work as follows: if $Q_1 = \exists$ then output 1 iff *at least one* of $A(\psi_{\restriction x_1=0})$ and $A(\psi_{\restriction x_1=1})$ returns 1. If $Q_1 = \forall$ then output 1 iff *both* $A(\psi_{\restriction x_1=0})$ and $A(\psi_{\restriction x_1=1})$. By the definition of $\exists$ and $\forall$, it is clear that A does indeed return the correct answer on any formula ψ.

Let $s_{n,m}$ denote the space A uses on formulas with n variables and description size m. The crucial point is —and here we use the fact that space can be *reused*—that both recursive computations $A(\psi_{\restriction x_1=0})$ and $A(\psi_{\restriction x_1=1})$ can run in the same space. Specifically, after computing $A(\psi_{\restriction x_1=0})$, the algorithm A needs to retain only the single bit of output from that computation, and can *reuse* the rest of the space for the computation of $A(\psi_{\restriction x_1=1})$. Thus, assuming that A uses $O(m)$ space to write $\psi\restriction x_1 = b$ for its recursive calls, we'll get that $s_{n,m} = s_{n-1,m} + O(m)$ yielding $s_{n,m} = O(n \cdot m)$. [2]

We now show that $L \leq_p$ TQBF for every $L \in$ **PSPACE**. Let M be a machine that decides L in $S(n)$ space and let $x \in \{0,1\}^n$. We show how to construct a quantified Boolean formula ψ of size $O(S(n)^2)$ that is true iff M accepts x. Recall that by Claim 4.4, there is a Boolean formula $\varphi_{M,x}$ such that for every two strings $C, C' \in \{0,1\}^m$ (where $m = O(S(n))$ is the number of bits require to encode a configuration of M), $\varphi_M(C, C') = 1$ iff C and C' are valid encodings of two adjacent configurations in the configuration graph $G_{M,x}$. We will use $\varphi_{M,x}$ to come up with a polynomial-sized quantified Boolean formula ψ' that has polynomially many Boolean variables bound by quantifiers and additional $2m$ unquantified Boolean variables $C_1, \ldots, C_m, C'_1, \ldots, C'_m$ (or, equivalently, two variables C, C' over $\{0,1\}^m$) such that for every $C, C' \in \{0,1\}^m$, $\psi(C, C')$ is true iff C has a directed path to C' in $G_{M,x}$. By plugging in the values C_{start} and C_{accept} to ψ' we get a quantified Boolean formula ψ that is true iff M accepts x.

We define the formula ψ' inductively. We let $\psi_i(C, C')$ be true if and only if there is a path of length at most 2^i from C to C' in $G_{M,x}$. Note that $\psi' = \psi_m$ and $\psi_0 = \varphi_{M,x}$. The crucial observation is that there is a path of length at most 2^i from C to C' if and only if there is a configuration C''

[2] The above analysis already suffices to show that TQBF is in **PSPACE**. However, we can actually show that the algorithm runs in linear space, specifically, $O(m+n)$ space. Note that algorithm always works with restrictions of the same formula ψ. So it can keep a global partial assignment array that for each variable x_i will contain either $0, 1$ or `'q'` (if it's quantified and not assigned any value). Algorithm A will use this global space for its operation, where in each call it will find the first quantified variable, set it to 0 and make the recursive call, then set it to 1 and make the recursive call, and then set it back to `'q'`. We see that A's space usage is given by the equation $s_{n,m} = s_{n-1,m} + O(1)$ and hence it uses $O(n+m)$ space.

with such that there are paths of length at most 2^{i-1} path from C to C'' and from C'' to C'. Thus the following formula suggests itself: $\psi_i(C, C') = \exists C'' \;\; \psi_{i-1}(C, C') \wedge \psi_{i-1}(C'', C)$.

However, this formula is no good. It implies that ψ_i's is twice the size of ψ_{i-1}, and a simple induction shows that ψ_m has size about 2^m, which is too large. Instead, we use additional quantified variables to save on description size, using the following more succinct definition for $\psi_i(C, C')$:

$$\exists C'' \forall D^1 \forall D^2 \big((D^1 = C \wedge D^2 = C') \vee (D^1 = C' \wedge D^2 = C'') \big) \Rightarrow \psi_{i-1}(D^1, D^2)$$

(Here, as in Example 4.9, $=$ and $\Rightarrow$ are convenient shorthands, and can be replaced by appropriate combinations of the standard Boolean operations $\wedge$ and $\neg$.) Note that $size(\psi_i) \leq size(\psi_{i-1}) + O(m)$ and hence $size(\psi_m) \leq O(m^2)$. We leave it to the reader to verify that the two definitions of ψ_i are indeed logically equivalent. As noted above we can convert the final formula to prenex form in polynomial time. ■

4.3.1 Savitch's theorem.

The astute reader may notice that because the above proof uses the notion of a configuration graph and does not require this graph to have out-degree one, it actually yields a stronger statement: that TQBF is not just hard for **PSPACE** but in fact even for **NPSPACE**!. Since TQBF $\in$ **PSPACE** this implies that **PSPACE** = **NSPACE**, which is quite surprising since our intuition is that the corresponding classes for time (**P** and **NP**) are different. In fact, using the ideas of the above proof, one can obtain the following theorem:

THEOREM 4.12 (SAVITCH [**?**])
For any space-constructible $S : \mathbb{N} \to \mathbb{N}$ *with* $S(n) \geq logn$, **NSPACE**$(S(n)) \subseteq$ **SPACE**$(S(n)^2)$

We remark that the running time of the algorithm obtained from this theorem can be as high as $2^{\Omega(s(n)^2)}$.

PROOF: The proof closely follows the proof that TQBF is **PSPACE**-complete. Let $L \in$ **NSPACE**$(S(n))$ be a language decided by a TM M such that for every $x \in \{0,1\}^n$, the configuration graph $G = G_{M,x}$ has at most $M = 2^{O(S(n))}$ vertices. We describe a recursive procedure REACH?(u, v, i) that returns "YES" if there is a path from u to v of length at most 2^i and "NO" otherwise. Note that REACH?$(s, t, \lceil \log M \rceil)$ is "YES" iff t is reachable from s. Again, the main observation is that there is a path from u to v of length at most 2^i iff there's a vertex z with paths from u to z and from z to v of lengths at most 2^{i-1}. Thus, on input u, v, i, REACH? will enumerate over all vertices z (at a cost of $O(\log M)$ space) and output "YES" if it finds one z such that REACH?$(u, z, i-1)$="YES" and REACH?$(z, v, i-1)$="YES". Once again, the crucial observation is that although the algorithm makes n recursive invocations, it can reuse the space in each of these invocations. Thus, if we let $s_{M,i}$ be the space complexity of REACH?(u, v, i) on an M-vertex graph we get that $s_{M,i} = s_{M,i-1} + O(\log M)$ and thus $s_{M,\log M} = O(\log^2 M) = O(S(n)^2)$. ■

4.3.2 The essence of PSPACE: optimum strategies for game-playing.

Recall that the central feature of **NP**-complete problems is that a yes answer has a short certificate. The analogous unifying concept for **PSPACE**-complete problems seems to be that of a winning strategy for a 2-player game with perfect information. A good example of such a game is Chess: two players alternately make moves, and the moves are made on a board visible to both. Thus moves have no hidden side effects; hence the term "perfect information." What does it mean for a player to have a "winning strategy?" The first player has a winning strategy iff there is a 1st move for player 1 such that for every possible 1st move of player 2 there is a 2nd move of player 1 such that.... (and so on) such that at the end player 1 wins. Thus deciding whether or not the first player has a winning strategy seems to require searching the tree of all possible moves. This is reminiscent of **NP**, for which we also seem to require exponential search. But the crucial difference is the lack of a short "certificate" for the statement "Player 1 has a winning strategy," since the only certificate we can think of is the winning strategy itself, which as noticed, requires exponentially many bits to even *describe*. Thus we seem to be dealing with a fundamentally different phenomenon than the one captured by **NP**.

The interplay of existential and universal quantifiers in the description of the the winning strategy motivates us to invent the following game.

Example 4.13 (The QBF game)
The "board" for the QBF game is a Boolean formula φ whose free variables are $x_1, x_2, \ldots, x_{2n}$. The two players alternately make moves, which involve picking values for $x_1, x_2, \ldots,$ in order. Thus player 1 will pick values for the odd-numbered variables $x_1, x_3, x_5, \ldots$ (in that order) and player 2 will pick values for the even-numbered variables $x_2, x_4, x_6, \ldots,$. We say player 1 wins iff at the end φ becomes true.

Clearly, player 1 has a winning strategy iff

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \cdots \forall x_{2n} \varphi(x_1, x_2, \ldots, x_{2n}),$$

namely, iff this quantified boolean formula is true.

Thus deciding whether player 1 has a winning strategy for a given board in the QBF game is **PSPACE**-complete.

At this point, the reader is probably thinking of familiar games such as Chess, Go, Checkers etc. and wondering whether complexity theory may help differentiate between them—for example, to justify the common intuition that Go is more difficult than Chess. Unfortunately, formalizing these issues in terms of asymptotic complexity is tricky because these are finite games, and as far as the existence of a winning strategy is concerned, there are at most three choices: Player 1 has has a winning strategy, Player 2 does, or neither does (they can play to a draw). However, one can study generalizations of these games to an $n \times n$ board where n is arbitrarily large —this may involve stretching the rules of the game since the definition of chess seems tailored to an 8×8 board— and then complexity theory can indeed by applied. For most common games, including chess, determining which player has a winning strategy in the $n \times n$ version is **PSPACE**-complete (see [?]or [?]). Note that if $\mathbf{NP} \neq \mathbf{PSPACE}$ then in general there is no short certificate for exhibiting that either player in the TQBF game has a winning strategy, which is alluded to in Evens and Tarjan's quote at the start of the chapter.

Proving **PSPACE**-completeness of games may seem like a frivolous pursuit, but similar ideas lead to **PSPACE**-completeness of some practical problems. Usually, these involve repeated moves that are in turn counteracted by an adversary. For instance, many computational problems of robotics are **PSPACE**-complete: the "player" is the robot and the "adversary" is the environment. (Treating the environment as an adversary may appear unduly pessimistic; but unfortunately even assuming a benign or "indifferent" environment still leaves us with a **PSPACE**-complete problem; see the Chapter notes.)

4.4 NL completeness

Now we consider problems that form the "essence" of non-deterministic logarithmic space computation, in other words, problems that are *complete* for **NL**. What kind of reduction should we use? We cannot use the polynomial-time reduction since $\mathbf{NL} \subseteq \mathbf{P}$. Thus every language in **NL** is polynomial-time reducible to the trivial language $\{1\}$ (reduction: "decide using polynomial time whether or not the input is in the **NL** language, and then map to 1 or 0 accordingly"). Intuitively, such trivial languages should not be the "hardest" languages of **NL**.

When choosing the type of reduction to define completeness for a complexity class, we must keep in mind the complexity phenomenon we seek to understand. In this case, the complexity question is whether or not $\mathbf{NL} = \mathbf{L}$. The reduction should not be more powerful than the weaker class, which is **L**. For this reason we use *logspace* reductions —for further, justification, see part (b) of Lemma 4.15 below). To define such reductions we must tackle the tricky issue that a reduction typically maps instances of size n to instances of size at least n, and so a logspace machine computing such a reduction does not have even the memory to write down its output. The way out is to require that the reduction should be able to compute any desired bit of the output in logarithmic space. In other words, if the reduction were given a separate output tape, it could in principle write out the entire new instance by first computing the first bit, then the second bit, and so on. (Many texts define such reductions using a "write-once" output tape.) The formal definition is as follows.

Definition 4.14 (logspace reduction)
Let $f : \{0,1\}^* \to \{0,1\}^*$ be a polynomially-bounded function (i.e., there's a constant $c > 0$ such that $f(x) \leq |x|^c$ for every $x \in \{0,1\}^*$). We say that f is *implicitly logspace computable*, if the languages $L_f = \{\langle x, i\rangle \mid f(x)_i = 1\}$ and $L'_f = \{\langle x, i\rangle \mid i \leq |f(x)|\}$ are in **L**.

Informally, we can think of a *single* $O(\log|x|)$-space machine that given input (x, i) outputs $f(x)|_i$ provided $i \leq |f(x)|$.

Language A is *logspace reducible* to language B, denoted $A \leq_l B$, if there is a function $f : \{0,1\}^* \to \{0,1\}^*$ that is implicitly logspace computable and $x \in A$ iff $f(x) \in B$ for every $x \in \{0,1\}^*$.

Logspace reducibility satisfies usual properties one expects.

Lemma 4.15
(a) If $A \leq_l B$ and $B \leq_l C$ then $A \leq_l C$. (b) If $A \leq_l B$ and $B \in \mathbf{L}$ then $A \in \mathbf{L}$.

Proof: We prove that if f, g are two functions that are logspace implicitly computable, then so is the function h where $h(x) = g(f(x))$. Then part (a) of the Lemma follows by letting f be the reduction from A to B and g be the reduction from B to C. Part (b) follows by letting f be the reduction from A to B and g be the characteristic function of B (i.e. $g(y) = 1$ iff $y \in B$).

So let M_f, M_g be the logspace machines that compute the mappings $x, i \mapsto f(x)_i$ and $y, j \mapsto g(y)_j$ respectively. We construct a machine M_h that computes the mapping $x, j \mapsto g(f(x))_j$, in other words, given input x, j outputs $g(f(x))_j$ provided $j \leq |g(f(x))|$. Machine M_h will pretend that it has an additional (fictitious) input tape on which $f(x)$ is written, and it is merely simulating M_g on this input (see Figure 4.3). Of course, the true input tape has x, j written on it. To maintain its fiction, M_h always maintains on its worktape the index, say i, of the cell on the fictitious tape that M_g is currently reading; this requires only $\log |f(x)|$ space. To compute for one step, M_g needs to know the contents of this cell, in other words, $f(x)|_i$. At this point M_h temporarily suspends its simulation of M_g (copying the contents of M_g's worktape to a safe place on its own worktape) and invokes M_f on inputs x, i to get $f(x)|_i$. Then it resumes its simulation of M_g using this bit. The total space M_h uses is $O(\log |g(f(x))| + s(|x|) + s'(|f(x)|)) = O(\log |x|)$. ■

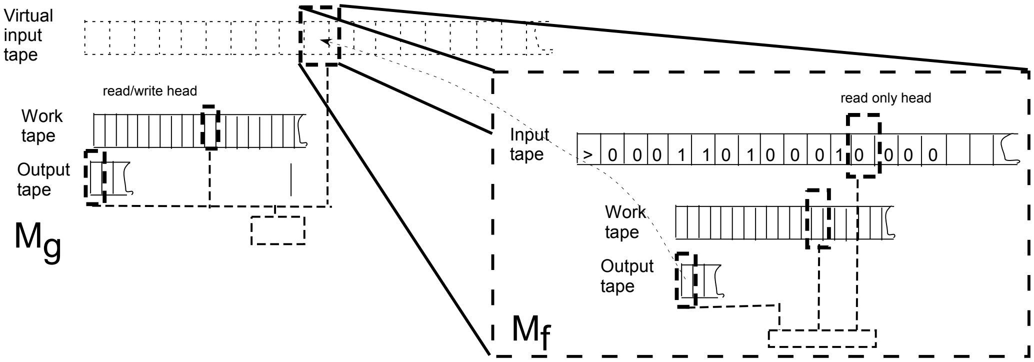

Figure 4.3: Composition of two implicitly logspace computable functions f, g. The machine M_g uses calls to f to implement a "virtual input tape". The overall space used is the space of M_f + the space of M_g + $O(\log |f(x)|) = O(log|x|)$.

We say that A is **NL**-*complete* if it is in **NL** and for every $B \in$ **NL**, $A \leq_l B$. Note that an **NL**-complete language is in **L** iff **NL** =**L**.

THEOREM 4.16
PATH *is* **NL**-*complete.*

PROOF: We have already seen that PATH is in **NL**. Let L be any language in **NL** and M be a machine that decides it in space $O(\log n)$. We describe a logspace implicitly computable function f that reduces L to PATH. For any input x of size n, $f(x)$ will be the configuration graph $G_{M,x}$ whose nodes are all possible $2^{O(\log n)}$ configurations of the machine on input x, along with the start configuration C_{start} and the accepting configuration C_{acc}. In this graph there is a path from C_{start} to C_{acc} iff M accepts x. The graph is represented as usual by an *adjacency matrix* that contain 1 in the $\langle C, C' \rangle^{th}$ position (i.e., in the C^{th} row and C'^{th} column if we identify the configurations with numbers between 0 and $2^{O(\log n)}$) iff there's an edge C from C' in $G_{M,x}$. To finish the proof we need to show that this adjacency matrix can be computed by a logspace reduction. This is easy since given a $\langle C, C' \rangle$ a deterministic machine can in space $O(|C| + |C'|) = O(\log |x|)$ examine C, C' and check whether C' is one of the (at most two) configurations that can follow C according to the transition function of M. ■

4.4.1 Certificate definition of NL: read-once certificates

In Chapter 2 we gave two equivalent definitions of **NP**— one using non-deterministic TM's and another using the notion of a *certificate*. The idea was that the nondeterministic choices of the NDTM that lead it to accept can be viewed as a "certificate" that the input is in the language, and vice versa. We can give a certificate-based definition also for **NL**, but only after addressing one tricky issue: a certificate may be polynomially long, and a logspace machine does not have the space to store it. Thus, the certificate-based definition of **NL** assumes that the logspace machine on a separate read-only tape. Furthermore, on each step of the machine the machine's head on that tape can either stay in place or move to the right. In particular, it cannot reread any bit to the left of where the head currently is. (For this reason the this kind of tape is called "*read once*".) It is easily seen that the following is an alternative definition of **NL** (see also Figure 4.4):

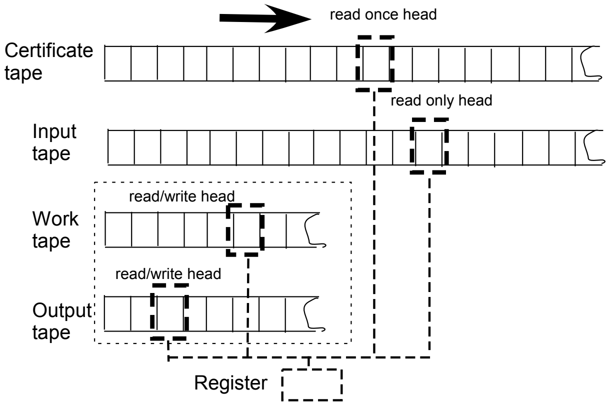

Figure 4.4: Certificate view of **NL**. The certificate for input x is placed on a special "read-once" tape on which the machine's head can never move to the left.

Definition 4.17 (**NL**- alternative definition.)
A language L is in **NL** if there exists a deterministic TM M and a with an additional special read-once input tape polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$,

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x,u) = 1$$

where by $M(x,u)$ we denote the output of M where x is placed on its input tape and u is placed on its special read-once tape, and M uses at most $O(\log |x|)$ space on its read/write tapes for every input x.

4.4.2 NL = coNL

Consider the problem $\overline{\mathsf{PATH}}$, i.e., the complement of PATH. A decision procedure for this language must accept when there is no path from s to t in the graph. Unlike in the case of PATH, there is no natural certificate for the *non-existence* of a path from s to t and thus it seemed "obvious" to researchers that $\overline{\mathsf{PATH}} \notin \mathbf{NL}$, until the discovery of the following theorem in the 1980s proved them wrong.

Theorem 4.18 (Immerman-Szlepcsenyi)
$\overline{\mathsf{PATH}} \in \mathbf{NL}$.

Proof: As we saw in Section 4.4.1, we need to show an $O(\log n)$-space algorithm A such that for every n-vertex graph G and vertices s and t, there exists a polynomial certificate u such that $A(\langle G, s, t\rangle, u) = 1$ if and only if t is not reachable from u in G, where A has only read-once access to u.

What can we certify to an $O(\log n)$-space algorithm? Let C_i be the set of vertices that are reachable from s in G within at most i steps. For every $i \in [n]$ and vertex v, we can easily certify that v is in C_i. The certificate simply contains the labels $v_0, v_1, \ldots, v_k$ of the vertices along the path from s to v (we can assume without loss of generality that vertices are labeled by the numbers 1 to n and hence the labels can be described by $\log n$ bit strings). The algorithm can check the certificate using read-once access by verifying that **(1)** $v_0 = s$, **(2)** for $j > 0$, there is an edge from v_{j-1} to v_j, **(3)** $v_k = v$ and (using a counter) that **(4)** the path ends within at most i steps. Note that the certificate is indeed of size at most polynomial in n.

Our algorithm uses the following two procedures:

1. Procedure to certify that a vertex v is not in C_i given the size of C_i.
2. Procedure to certify that $|C_i| = c$ for some number c, given the size of C_{i-1}.

Since $C_0 = \{s\}$ and C_n contains all the vertices reachable from s, we can apply the second procedure iteratively to learn the sizes of the sets $C_1, \ldots, C_n$, and then use the first procedure to certify that $t \notin C_n$.

Certifying that v is not in C_i, given $|C_i|$. The certificate is simply the list of certificates that u is in C_i for every $u \in C_i$ sorted in ascending order of labels (recall that we identify labels with numbers in $[n]$). The algorithm checks that **(1)** each certificate is valid, **(2)** the label of a vertex u for which a certificate is given is indeed larger than the label of the previous vertex, **(3)** no certificate is provided for v, and **(4)** the total number of certificates provided is exactly $|C_i|$. If $v \notin C_i$ then the algorithm will accept the above certificate, but if $v \in C_i$ there will not exist $|C_i|$ certificates that vertices $u_1 < u_2 < \ldots < u_{|C_i|}$ are in C_i where $u_j \neq v$ for every j.

Certifying that v is not in C_i, given $|C_{i-1}|$. Before showing how we certify that $|C_i| = c$ given $|C_{i-1}|$, we show how to certify that $v \notin C_i$ with this information. This is very similar to the above procedure: the certificate is the list of $|C_{i-1}|$ certificates that $u \in C_{i-1}$ for every $u \in C_{i-1}$ in ascending order. The algorithm checks everything as before except that in step **(3)** it verifies that no certificate is given for v *or for a neighbor of* v. Since $v \in C_i$ if and only if there exists $u \in C_{i-1}$ such that $u = v$ or u is a neighbor of v in G, the procedure will not accept a false certificate by the same reasons as above.

Certifying that $|C_i| = c$ given $|C_{i-1}|$. For every vertex v, if $v \in C_i$ then there is a certificate for this fact, and by the above procedure, given $|C_{i-1}|$, if $v \notin C_i$ then there is a certificate for this fact as well. The certificate that $|C_i| = c$ will consist of n certificates for each of the vertices 1 to n in ascending order. For every vertex u, there will be an appropriate certificate depending on whether $u \in C_i$ or not. The algorithm will verify all the certificate and count the number of certificate that a vertex is in C_i. It accepts if this count is equal to c. ■

Using the notion of the configuration graph we can modify the proof of Theorem 4.18 to prove the following:

COROLLARY 4.19
For every space constructible $S(n) > \log n$, $\mathbf{NSPACE}(S(n)) = \mathbf{coNSPACE}(S(n))$.

Our understanding of space-bounded complexity.
The following is our understanding of space-bounded complexity.

$$\mathbf{DTIME}(s(n)) \subseteq \mathbf{SPACE}(s(n)) \subseteq \mathbf{NSPACE}(s(n)) = \mathbf{coNSPACE}(s(n)) \subseteq \mathbf{DTIME}(2^{O(s(n))}).$$

None of the inclusions are known to be strict though we believe they all are.

Chapter notes and history

The concept of space complexity had already been explored in the 1960s; in particular, Savitch's theorem predates the Cook-Levin theorem. Stockmeyer and Meyer proved the **PSPACE**-completeness of TQBF soon after Cook's paper appeared. A few years later Even and Tarjan pointed out the connection to game-playing and proved the **PSPACE**-completeness of a game called Generalized Hex. Papadimitriou's book gives a detailed account of **PSPACE**-completeness. He also shows **PSPACE**-completeness of several *Games against nature* first defined in [?]. Unlike the TQBF game, where one player is *Existential* and the other *Universal*, here the second player chooses moves randomly. The intention is to model games played against nature—where "nature" could mean not just weather for example, but also large systems such as the stock market that are presumably "indifferent" to the fate of individuals. Papadimitriou gives an alternative characterization **PSPACE** using such games. A stronger result, namely, a characterization of **PSPACE** using interactive proofs, is described in Chapter 8.

Exercises

§1 Show that $\mathbf{SPACE}(S(n)) = \mathbf{SPACE}(0)$ when $S(n) = \log\log n$.

§2 Prove the existence of a universal TM for space bounded computation (analogously to the deterministic universal TM of Theorem 1.13). That is, prove that there exists a a TM $\mathcal{SU}$ such that for every string α, and input x, if the TM M_α represented by α halts on x before using t cells of its work tapes then $\mathcal{SU}(\alpha, t, x) = M_\alpha(x)$, and moreover, $\mathcal{SU}$ uses at most Ct cells of its work tapes, where C is a constant depending only on M_α. (Despite the fact that the bound here is better than the bound of Theorem 1.13, the proof of this statement is actually easier than the proof of Theorem 1.13.)

§3 Prove that the language $\mathsf{SPACETM}$ of (2) is **PSPACE**-complete.

§4 Show that the following language is **NL**-complete:

$$\{ \llcorner G \lrcorner : G \text{ is a strongly connected digraph}\} .$$

§5 Show that 2SAT is in **NL**.

§6 Suppose we define **NP**-completeness using logspace reductions instead of polynomial-time reductions. Show (using the proof of the Cook-Levin Theorem) that SAT and $\mathsf{3SAT}$ continue to be **NP**-complete under this new definition. Conclude that $\mathsf{SAT} \in \mathbf{L}$ iff $\mathbf{NP} = \mathbf{L}$.

§7 Show that TQBF is complete for **PSPACE** also under logspace reductions.

§8 Show that in every finite 2-person game with perfect information (by *finite* we mean that there is an *a priori* upperbound n on the number of moves after which the game is over and one of the two players is declared the victor —there are no draws) one of the two players has a winning strategy.

§9 Define **polyL** to be $\cup_{c>0}\mathbf{SPACE}(\log^c n)$. Steve's Class SC (named in honor of Steve Cook) is defined to be the set of languages that can be decided by deterministic machines that run in polynomial time and $\log^c n$ space for some $c > 0$.

It is an open problem whether $\mathsf{PATH} \in \mathsf{SC}$. Why does Savitch's Theorem not resolve this question?

Is SC the same as $\mathbf{polyL} \cap \mathbf{P}$?