8

SPACE COMPLEXITY

In this chapter we consider the complexity of computational problems in terms of the amount of space, or memory, that they require. Time and space are two of the most important considerations when we seek practical solutions to many computational problems. Space complexity shares many of the features of time complexity and serves as a further way of classifying problems according to their computational difficulty.

As we did with time complexity, we need to select a model for measuring the space used by an algorithm. We continue with the Turing machine model for the same reason that we used it to measure time. Turing machines are mathematically simple and close enough to real computers to give meaningful results.

DEFINITION 8.1

Let M be a deterministic Turing machine that halts on all inputs. The ***space complexity*** of M is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of tape cells that M scans on any input of length n. If the space complexity of M is $f(n)$, we also say that M runs in space $f(n)$.

If M is a nondeterministic Turing machine wherein all branches halt on all inputs, we define its space complexity $f(n)$ to be the maximum number of tape cells that M scans on any branch of its computation for any input of length n.

We typically estimate the space complexity of Turing machines by using asymptotic notation.

DEFINITION 8.2

Let $f: \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. The ***space complexity classes***, $\mathbf{SPACE}(f(n))$ and $\mathbf{NSPACE}(f(n))$, are defined as follows.

$$\text{SPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } O(f(n)) \text{ space deterministic Turing machine}\}.$$

$$\text{NSPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } O(f(n)) \text{ space nondeterministic Turing machine}\}.$$

EXAMPLE 8.3

In Chapter 7 we introduced the NP-complete problem *SAT*. Here, we show that *SAT* can be solved with a linear space algorithm. We believe that *SAT* cannot be solved with a polynomial time algorithm, much less with a linear time algorithm, because *SAT* is NP-complete. Space appears to be more powerful than time because space can be reused, whereas time cannot.

M_1 = "On input $\langle \phi \rangle$, where ϕ is a Boolean formula:

1. For each truth assignment to the variables $x_1, \ldots, x_m$ of ϕ:
2. Evaluate ϕ on that truth assignment.
3. If ϕ ever evaluated to 1, *accept*; if not, *reject*."

Machine M_1 clearly runs in linear space because each iteration of the loop can reuse the same portion of the tape. The machine needs to store only the current truth assignment and that can be done with $O(m)$ space. The number of variables m is at most n, the length of the input, so this machine runs in space $O(n)$.

EXAMPLE 8.4

Here, we illustrate the nondeterministic space complexity of a language. In the next section we show how determining the nondeterministic space complexity can be useful in determining its deterministic space complexity. Consider

the problem of testing whether a nondeterministic finite automaton accepts all strings. Let

$$ALL_{\text{NFA}} = \{\langle A \rangle \mid A \text{ is a NFA and } L(A) = \Sigma^*\}.$$

We give a nondeterministic linear space algorithm that decides the complement of this language, $\overline{ALL_{\text{NFA}}}$. The idea behind this algorithm is to use nondeterminism to guess a string that is rejected by the NFA and to use linear space to keep track of which states the NFA could be in at a particular time. Note that this language is not known to be in NP or in coNP.

N = "On input $\langle M \rangle$ where M is an NFA:

1. Place a marker on the start state of the NFA.
2. Repeat 2^q times, where q is the number of states of M:
3. Nondeterministically select an input symbol and change the positions of the markers on M's states to simulate reading that symbol.
4. *Accept* if Stages 2 and 3 reveal some string that M rejects, that is, if at some point none of the markers lie on accept states of M. Otherwise, *reject*."

If M rejects any strings, it must reject one of length at most 2^q because in any longer string that is rejected the locations of the markers described in the preceding algorithm would repeat. The section of the string between the repetitions can be removed to obtain a shorter rejected string. Hence N decides $\overline{ALL_{\text{NFA}}}$. (Note that N accepts improperly formed inputs, too.)

The only space needed by this algorithm is for storing the location of the markers and the repeat loop counter, and that can be done with linear space. Hence the algorithm runs in nondeterministic space $O(n)$. Next, we prove a theorem that provides information about the deterministic space complexity of ALL_{NFA}.

8.1 SAVITCH'S THEOREM

Savitch's theorem is one of the earliest results concerning space complexity. It shows that deterministic machines can simulate nondeterministic machines by using a surprisingly small amount of space. For time complexity, such a simulation seems to require an exponential increase in time. For space complexity, Savitch's theorem shows that any nondeterministic TM that uses $f(n)$ space can be converted to a deterministic TM that uses only $f^2(n)$ space.

THEOREM 8.5

Savitch's theorem For any[1] function $f: \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq n$,

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

PROOF IDEA We need to simulate an $f(n)$ space NTM deterministically. A naive approach is to proceed by trying all the branches of the NTM's computation, one by one. The simulation needs to keep track of which branch it is currently trying so that it is able to go on to the next one. But a branch that uses $f(n)$ space may run for $2^{O(f(n))}$ steps, and each step may be a nondeterministic choice. Exploring the branches sequentially would require recording all the choices used on a particular branch in order to be able to find the next branch. Therefore this approach may use $2^{O(f(n))}$ space, exceeding our goal of $O(f^2(n))$ space.

Instead, we take a different approach by considering the following more general problem. We are given two configurations of the NTM, c_1 and c_2, together with a number t, and we must test whether the NTM can get from c_1 to c_2 within t steps. We call this problem the ***yieldability problem***. By solving the yieldability problem, where c_1 is the start configuration, c_2 is the accept configuration, and t is the maximum number of steps that the nondeterministic machine can use, we can determine whether the machine accepts its input.

We give a deterministic, recursive algorithm that solves the yieldability problem. It operates by searching for an intermediate configuration c_m, and recursively testing whether (1) c_1 can get to c_m within $t/2$ steps, and (2) whether c_m can get to c_2 within $t/2$ steps. Reusing the space for each of the two recursive tests allows a significant saving of space.

This algorithm needs space for storing the recursion stack. Each level of the recursion uses $O(f(n))$ space to store a configuration. The depth of the recursion is $\log t$, where t is the maximum time that the nondeterministic machine may use on any branch. We have $t = 2^{O(f(n))}$, so $\log t = O(f(n))$. Hence the deterministic simulation uses $O(f^2(n))$ space.

PROOF Let N be an NTM deciding a language A in space $f(n)$. We construct a deterministic TM M deciding A. Machine M uses the procedure CANYIELD, which tests whether one of N's configurations can yield another within a specified number of steps. This procedure solves the yieldability problem described in the proof idea.

Let w be a string considered as input to N. For configurations c_1 and c_2 of N on w, and integer t, CANYIELD(c_1, c_2, t) outputs *accept* if N can go from configuration c_1 to configuration c_2 in t or fewer steps along some nondeterministic path. If not, CANYIELD outputs *reject*. For convenience, we assume that t is a power of 2.

---

[1] On page 323, we show that Savitch's theorem also holds whenever $f(n) \geq \log n$.

CANYIELD = "On input c_1, c_2, and t:

1. If $t = 1$, then test directly whether $c_1 = c_2$ or whether c_1 yields c_2 in one step according to the rules of N. *Accept* if either test succeeds; *reject* if both fail.
2. If $t > 1$, then for each configuration c_m of N on w using space $f(n)$:
3. Run CANYIELD$(c_1, c_m, \frac{t}{2})$.
4. Run CANYIELD$(c_m, c_2, \frac{t}{2})$.
5. If steps 3 and 4 both accept, then *accept*.
6. If haven't yet accepted, *reject*."

Now we define M to simulate N as follows. We first modify N so that when it accepts it clears its tape and moves the head to the leftmost cell, thereby entering a configuration called c_{accept}. We let c_{start} be the start configuration of N on w. We select a constant d so that N has no more than $2^{df(n)}$ configurations using $f(n)$ tape, where n is the length of w. Then we know that $2^{df(n)}$ provides an upper bound on the running time of any branch of N on w.

M = "On input w:

1. Output the result of CANYIELD$(c_{\text{start}}, c_{\text{accept}}, 2^{df(n)})$."

Algorithm CANYIELD obviously solves the yieldability problem, and hence M correctly simulates N. We need to analyze it to verify that M works within $O(f^2(n))$ space.

Whenever CANYIELD invokes itself recursively, it stores the current stage number and the values of c_1, c_2, and t on a stack so that these values may be restored upon return from the recursive invocation. Each level of the recursion thus uses $O(f(n))$ additional space. Furthermore, each level of the recursion divides the size of t in half. Initially t starts out equal to $2^{df(n)}$, so the depth of the recursion is $O(\log 2^{df(n)})$ or $O(f(n))$. Therefore the total space used is $O(f^2(n))$, as claimed.

One technical difficulty arises in this argument because algorithm M needs to know the value of $f(n)$ when it calls CANYIELD. We can handle this difficulty by modifying M so that it tries $f(n) = 1, 2, 3, \ldots$ For each value $f(n) = i$, the modified algorithm uses CANYIELD to determine whether the accept configuration is reachable. In addition, it uses CANYIELD to determine whether N uses at least space $i + 1$ by testing whether N can reach any of the configurations of length $i+1$ from the start configuration. If the accept configuration is reachable, M accepts; if no configuration of length $i + 1$ is reachable, M rejects; and otherwise M continues with $f(n) = i + 1$. (We could have handled this difficulty in another way by assuming that M can compute $f(n)$ within $O(f(n))$ space, but then we would need to add that assumption to the statement of the theorem).

8.2

THE CLASS PSPACE

By analogy with the class P, we define the class PSPACE for space complexity.

DEFINITION 8.6

PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine. In other words,

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k).$$

We define NPSPACE, the nondeterministic counterpart to PSPACE, in terms of the NSPACE classes. However, PSPACE = NPSPACE by virtue of Savitch's theorem, because the square of any polynomial is still a polynomial.

In Examples 8.3 and 8.4 we showed that SAT is in SPACE(n) and that ALL_{NFA} is in coNSPACE(n) and hence, by Savitch's theorem, in SPACE(n^2), because the deterministic space complexity classes are closed under complement. Therefore both languages are in PSPACE.

Let's examine the relationship of PSPACE with P and NP. We observe that P $\subseteq$ PSPACE because a machine that runs quickly cannot use a great deal of space. More precisely, for $t(n) \geq n$, any machine that operates in time $t(n)$ can use at most $t(n)$ space because a machine can explore at most one new cell at each step of its computation. Similarly, NP $\subseteq$ NPSPACE, and so NP $\subseteq$ PSPACE.

Conversely, we can bound the time complexity of a Turing machine in terms of its space complexity. For $f(n) \geq n$, a TM that uses $f(n)$ space can have at most $f(n)\, 2^{O(f(n))}$ different configurations, by a simple generalization of the proof of Lemma 5.8 on page 194. A TM computation that halts may not repeat a configuration. Therefore a TM[2] that uses space $f(n)$ must run in time $f(n)\, 2^{O(f(n))}$, so PSPACE $\subseteq$ EXPTIME $= \bigcup_k \text{TIME}(2^{n^k})$.

We summarize our knowledge of the relationships among the complexity classes defined so far in the series of containments

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE} \subseteq \text{EXPTIME}.$$

We don't know whether any of these containments is actually an equality. Someone may yet discover a simulation like the one in Savitch's theorem that merges some of these classes into the same class. However, in Chapter 9 we prove that P $\neq$ EXPTIME. Therefore at least one of the preceding containments is proper, but we are unable to say which! Indeed, most researchers

[2]The requirement here that $f(n) \geq n$ is generalized later to $f(n) \geq \log n$, when we introduce TMs that use sublinear space on page 322.

believe that all the containments are proper. The following diagram depicts the relationships among these classes, assuming that all are different.

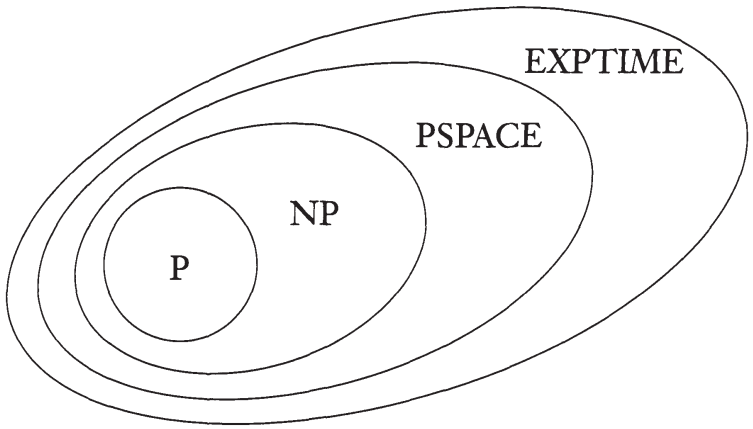


FIGURE 8.7
Conjectured relationships among P, NP, PSPACE, and EXPTIME

8.3 PSPACE-COMPLETENESS

In Section 7.4 we introduced the category of NP-complete languages as representing the most difficult languages in NP. Demonstrating that a language is NP-complete provides strong evidence that the language is not in P. If it were, P and NP would be equal. In this section we introduce the analogous notion, PSPACE-completeness, for the class PSPACE.

DEFINITION 8.8

A language B is ***PSPACE-complete*** if it satisfies two conditions:

1. B is in PSPACE, and
2. every A in PSPACE is polynomial time reducible to B.

If B merely satisfies condition 2, we say that it is ***PSPACE-hard***.

In defining PSPACE-completeness, we use polynomial time reducibility as given in Definition 7.29. Why don't we define a notion of polynomial *space* reducibility and use that instead of polynomial *time* reducibility? To understand the answer to this important question, consider our motivation for defining complete problems in the first place.

Complete problems are important because they are examples of the most difficult problems in a complexity class. A complete problem is most difficult because any other problem in the class is easily reduced into it, so if we find an easy way to solve the complete problem, we can easily solve all other problems in the class. The reduction must be *easy*, relative to the complexity of typical problems in the class, for this reasoning to apply. If the reduction itself were difficult to compute, an easy solution to the complete problem wouldn't necessarily yield an easy solution to the problems reducing to it.

Therefore the rule is: Whenever we define complete problems for a complexity class, the reduction model must be more limited than the model used for defining the class itself.

THE TQBF PROBLEM

Our first example of a PSPACE-complete problem involves a generalization of the satisfiability problem. Recall that a ***Boolean formula*** is an expression that contains Boolean variables, the constants 0 and 1, and the Boolean operations $\wedge$, $\vee$, and $\neg$. We now introduce a more general type of Boolean formula.

The ***quantifiers*** $\forall$ (for all) and $\exists$ (there exists) make frequent appearances in mathematical statements. Writing the statement $\forall x\, \phi$ means that, for *every* value for the variable x, the statement ϕ is true. Similarly, writing the statement $\exists x\, \phi$ means that, for *some* value of the variable x, the statement ϕ is true. Sometimes, $\forall$ is referred to as the ***universal quantifier*** and $\exists$ as the ***existential quantifier***. We say that the variable x immediately following the quantifier is ***bound*** to the quantifier.

For example, considering the natural numbers, the statement $\forall x\, [x + 1 > x]$ means that the successor $x + 1$ of every natural number x is greater than the number itself. Obviously, this statement is true. However, the statement $\exists y\, [y + y = 3]$ obviously is false. When interpreting the meaning of statements involving quantifiers, we must consider the ***universe*** from which the values are drawn. In the preceding cases the universe comprised the natural numbers, but if we took the real numbers instead, the existentially quantified statement would become true.

Statements may contain several quantifiers, as in $\forall x\, \exists y\, [y > x]$. For the universe of the natural numbers, this statement says that every natural number has another natural number larger than it. The order of the quantifiers is important. Reversing the order, as in the statement $\exists y\, \forall x\, [y > x]$, gives an entirely different meaning—namely, that some natural number is greater than all others. Obviously, the first statement is true and the second statement is false.

A quantifier may appear anywhere in a mathematical statement. It applies to the fragment of the statement appearing within the matched pair of parentheses or brackets following the quantified variable. This fragment is called the ***scope*** of the quantifier. Often, it is convenient to require that all quantifiers appear at the beginning of the statement and that each quantifier's scope is everything following it. Such statements are said to be in ***prenex normal form***. Any statement may be put into prenex normal form easily. We consider statements in this form only, unless otherwise indicated.

Boolean formulas with quantifiers are called ***quantified Boolean formulas***. For such formulas, the universe is $\{0, 1\}$. For example,

$$\phi = \forall x \, \exists y \, [\, (x \vee y) \wedge (\overline{x} \vee \overline{y}) \,]$$

is a quantified Boolean formula. Here, ϕ is true, but it would be false if the quantifiers $\forall x$ and $\exists y$ were reversed.

When each variable of a formula appears within the scope of some quantifier, the formula is said to be ***fully quantified***. A fully quantified Boolean formula is sometimes called a ***sentence*** and is always either true or false. For example, the preceding formula ϕ is fully quantified. However, if the initial part, $\forall x$, of ϕ were removed, the formula would no longer be fully quantified and would be neither true nor false.

The *TQBF* problem is to determine whether a fully quantified Boolean formula is true or false. We define the language

$$TQBF = \{\langle \phi \rangle \mid \phi \text{ is a true fully quantified Boolean formula}\}.$$

THEOREM 8.9

TQBF is PSPACE-complete.

PROOF IDEA To show that *TQBF* is in PSPACE we give a straightforward algorithm that assigns values to the variables and recursively evaluates the truth of the formula for those values. From that information the algorithm can determine the truth of the original quantified formula.

To show that every language A in PSPACE reduces to *TQBF* in polynomial time, we begin with a polynomial space-bounded Turing machine for A. Then we give a polynomial time reduction that maps a string to a quantified Boolean formula ϕ that encodes a simulation of the machine on that input. The formula is true iff the machine accepts.

As a first attempt at this construction, let's try to imitate the proof of the Cook–Levin theorem, Theorem 7.37. We can construct a formula ϕ that simulates M on an input w by expressing the requirements for an accepting tableau. A tableau for M on w has width $O(n^k)$, the space used by M, but its height is exponential in n^k because M can run for exponential time. Thus, if we were to represent the tableau with a formula directly, we would end up with a formula of exponential size. However, a polynomial time reduction cannot produce an exponential-size result, so this attempt fails to show that $A \leq_{\mathrm{P}} TQBF$.

Instead, we use a technique related to the proof of Savitch's theorem to construct the formula. The formula divides the tableau into halves and employs the universal quantifier to represent each half with the same part of the formula. The result is a much shorter formula.

PROOF First, we give a polynomial space algorithm deciding *TQBF*.

T = "On input $\langle \phi \rangle$, a fully quantified Boolean formula:

1. If ϕ contains no quantifiers, then it is an expression with only constants, so evaluate ϕ and *accept* if it is true; otherwise, *reject*.
2. If ϕ equals $\exists x\ \psi$, recursively call T on ψ, first with 0 substituted for x and then with 1 substituted for x. If either result is accept, then *accept*; otherwise, *reject*.
3. If ϕ equals $\forall x\ \psi$, recursively call T on ψ, first with 0 substituted for x and then with 1 substituted for x. If both results are accept, then *accept*; otherwise, *reject*."

Algorithm T obviously decides *TQBF*. To analyze its space complexity we observe that the depth of the recursion is at most the number of variables. At each level we need only store the value of one variable, so the total space used is $O(m)$, where m is the number of variables that appear in ϕ. Therefore T runs in linear space.

Next, we show that *TQBF* is PSPACE-hard. Let A be a language decided by a TM M in space n^k for some constant k. We give a polynomial time reduction from A to *TQBF*.

The reduction maps a string w to a quantified Boolean formula ϕ that is true iff M accepts w. To show how to construct ϕ we solve a more general problem. Using two collections of variables denoted c_1 and c_2 representing two configurations and a number $t > 0$, we construct a formula $\phi_{c_1,c_2,t}$. If we assign c_1 and c_2 to actual configurations, the formula is true iff M can go from c_1 to c_2 in at most t steps. Then we can let ϕ be the formula $\phi_{c_{\text{start}},c_{\text{accept}},h}$, where $h = 2^{df(n)}$ for a constant d, chosen so that M has no more than $2^{df(n)}$ possible configurations on an input of length n. Here, let $f(n) = n^k$. For convenience, we assume that t is a power of 2.

The formula encodes the contents of tape cells as in the proof of the Cook–Levin theorem. Each cell has several variables associated with it, one for each tape symbol and state, corresponding to the possible settings of that cell. Each configuration has n^k cells and so is encoded by $O(n^k)$ variables.

If $t = 1$, we can easily construct $\phi_{c_1,c_2,t}$. We design the formula to say that either c_1 equals c_2, or c_2 follows from c_1 in a single step of M. We express the equality by writing a Boolean expression saying that each of the variables representing c_1 contains the same Boolean value as the corresponding variable representing c_2. We express the second possibility by using the technique presented in the proof of the Cook–Levin theorem. That is, we can express that c_1 yields c_2 in a single step of M by writing Boolean expressions stating that the contents of each triple of c_1's cells correctly yields the contents of the corresponding triple of c_2's cells.

If $t > 1$, we construct $\phi_{c_1,c_2,t}$ recursively. As a warmup let's try one idea that doesn't quite work and then fix it. Let

$$\phi_{c_1,c_2,t} = \exists m_1 \left[\phi_{c_1,m_1,\frac{t}{2}} \wedge \phi_{m_1,c_2,\frac{t}{2}}\right].$$

The symbol m_1 represents a configuration of M. Writing $\exists m_1$ is shorthand for $\exists x_1, \ldots, x_l$, where $l = O(n^k)$ and $x_1, \ldots, x_l$ are the variables that encode m_1. So this construction of $\phi_{c_1,c_2,t}$ says that M can go from c_1 to c_2 in at most t steps if some intermediate configuration m_1 exists, whereby M can go from c_1 to m_1 in at most $\frac{t}{2}$ steps and then from m_1 to c_2 in at most $\frac{t}{2}$ steps. Then we construct the two formulas $\phi_{c_1,m_1,\frac{t}{2}}$ and $\phi_{m_1,c_2,\frac{t}{2}}$ recursively.

The formula $\phi_{c_1,c_2,t}$ has the correct value; that is, it is TRUE whenever M can go from c_1 to c_2 within t steps. However, it is too big. Every level of the recursion involved in the construction cuts t in half but roughly doubles the size of the formula. Hence we end up with a formula of size roughly t. Initially $t = 2^{df(n)}$, so this method gives an exponentially large formula.

To reduce the size of the formula we use the $\forall$ quantifier in addition to the $\exists$ quantifier. Let

$$\phi_{c_1,c_2,t} = \exists m_1 \, \forall (c_3,c_4) \in \{(c_1,m_1),(m_1,c_2)\} \, [\phi_{c_3,c_4,\frac{t}{2}}].$$

The introduction of the new variables representing the configurations c_3 and c_4 allows us to "fold" the two recursive subformulas into a single subformula, while preserving the original meaning. By writing $\forall (c_3,c_4) \in \{(c_1,m_1),(m_1,c_2)\}$, we indicate that the variables representing the configurations c_3 and c_4 may take the values of the variables of c_1 and m_1 or of m_1 and c_2, respectively, and that the resulting formula $\phi_{c_3,c_4,\frac{t}{2}}$ is true in either case. We may replace the construct $\forall x \in \{y,z\} \, [\ldots]$ by the equivalent construct $\forall x \, [(x = y \vee x = z) \rightarrow \ldots]$ to obtain a syntactically correct quantified Boolean formula. Recall that in Section 0.2 we showed that Boolean implication ($\rightarrow$) and Boolean equality ($=$) can be expressed in terms of AND and NOT. Here, for clarity, we use the symbol $=$ for Boolean equality instead of the equivalent symbol $\leftrightarrow$ used in Section 0.2.

To calculate the size of the formula $\phi_{c_{\text{start}},c_{\text{accept}},h}$, where $h = 2^{df(n)}$, we note that each level of the recursion adds a portion of the formula that is linear in the size of the configurations and is thus of size $O(f(n))$. The number of levels of the recursion is $\log(2^{df(n)})$, or $O(f(n))$. Hence the size of the resulting formula is $O(f^2(n))$.

WINNING STRATEGIES FOR GAMES

For the purposes of this section, a ***game*** is loosely defined to be a competition in which opposing parties attempt to achieve some goal according to prespecified rules. Games appear in many forms, from board games such as chess to economic and war games that model corporate or societal conflict.

Games are closely related to quantifiers. A quantified statement has a corresponding game; conversely, a game often has a corresponding quantified statement. These correspondences are helpful in several ways. For one, expressing a mathematical statement that uses many quantifiers in terms of the corresponding game may give insight into the statement's meaning. For another, expressing a game in terms of a quantified statement aids in understanding the complexity of the game. To illustrate the correspondence between games and quantifiers, we turn to an artificial game called the ***formula game***.

Let $\phi = \exists x_1 \forall x_2 \exists x_3 \cdots \mathrm{Q} x_k \, [\psi]$ be a quantified Boolean formula in prenex normal form. Here Q represents either a $\forall$ or an $\exists$ quantifier. We associate a game with ϕ as follows. Two players, called Player A and Player E, take turns selecting the values of the variables $x_1, \ldots, x_k$. Player A selects values for the variables that are bound to $\forall$ quantifiers and player E selects values for the variables that are bound to $\exists$ quantifiers. The order of play is the same as that of the quantifiers at the beginning of the formula. At the end of play we use the values that the players have selected for the variables and declare that Player E has won the game if ψ, the part of the formula with the quantifiers stripped off, is now TRUE. Player A has won if ψ is now FALSE.

EXAMPLE 8.10

Say that ϕ_1 is the formula

$$\exists x_1 \forall x_2 \exists x_3 \left[(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})\right].$$

In the formula game for ϕ_1, Player E picks the value of x_1, then Player A picks the value of x_2, and finally Player E picks the value of x_3.

To illustrate a sample play of this game, we begin by representing the Boolean value TRUE with 1 and FALSE with 0, as usual. Let's say that Player E picks $x_1 = 1$, then Player A picks $x_2 = 0$, and finally Player E picks $x_3 = 1$. With these values for x_1, x_2, and x_3, the subformula

$$(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})$$

is 1, so Player E has won the game. In fact, Player E may always win this game by selecting $x_1 = 1$ and then selecting x_3 to be the negation of whatever Player A selects for x_2. We say that Player E has a ***winning strategy*** for this game. A player has a winning strategy for a game if that player wins when both sides play optimally.

Now let's change the formula slightly to get a game in which Player A has a winning strategy. Let ϕ_2 be the formula

$$\exists x_1 \forall x_2 \exists x_3 \left[(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3})\right].$$

Player A now has a winning strategy because, no matter what Player E selects for x_1, Player A may select $x_2 = 0$, thereby falsifying the part of the formula appearing after the quantifiers, whatever Player E's last move may be. ▒

We next consider the problem of determining which player has a winning strategy in the formula game associated with a particular formula. Let

$$\textit{FORMULA-GAME} = \{\langle \phi \rangle \mid \text{Player E has a winning strategy in the formula game associated with } \phi\}.$$

THEOREM 8.11

FORMULA-GAME is PSPACE-complete

PROOF IDEA *FORMULA-GAME* is PSPACE-complete for a simple reason. It is the same as *TQBF*. To see that *FORMULA-GAME* = *TQBF*, observe that a formula is TRUE exactly when Player E has a winning strategy in the associated formula game. The two statements are different ways of saying the same thing.

PROOF The formula $\phi = \exists x_1 \forall x_2 \exists x_3 \cdots [\psi]$ is TRUE when some setting for x_1 exists such that, for any setting of x_2, a setting of x_3 exists such that, and so on ..., where ψ is TRUE under the settings of the variables. Similarly, Player E has a winning strategy in the game associated with ϕ when Player E can make some assignment to x_1 such that, for any setting of x_2, Player E can make an assignment to x_3 such that, and so on ..., ψ is TRUE under these settings of the variables.

The same reasoning applies when the formula doesn't alternate between existential and universal quantifiers. If ϕ has the form $\forall x_1, x_2, x_3 \exists x_4, x_5 \forall x_6 [\psi]$, Player A would make the first three moves in the formula game to assign values to x_1, x_2, and x_3; then Player E would make two moves to assign x_4 and x_5; and finally Player A would assign a value x_6.

Hence $\phi \in$ *TQBF* exactly when $\phi \in$ *FORMULA-GAME*, and the theorem follows from Theorem 8.9.

GENERALIZED GEOGRAPHY

Now that we know that the formula game is PSPACE-complete, we can establish the PSPACE-completeness or PSPACE-hardness of some other games more easily. We'll begin with a generalization of the game geography and later discuss games such as chess, checkers, and GO.

Geography is a child's game in which players take turns naming cities from anywhere in the world. Each city chosen must begin with the same letter that ended the previous city's name. Repetition isn't permitted. The game starts with some designated starting city and ends when some player loses because he or she is unable to continue. For example, if the game starts with Peoria, then Amherst might legally follow (because Peoria ends with the letter *a*, and Amherst begins with the letter *a*), then Tucson, then Nashua, and so on until one player gets stuck and thereby loses.

We can model this game with a directed graph whose nodes are the cities of the world. We draw an arrow from one city to another if the first can lead to the second according to the game rules. In other words, the graph contains an edge from a city X to a city Y if city X ends with the same letter that begins city Y. We illustrate a portion of the geography graph in Figure 8.12.

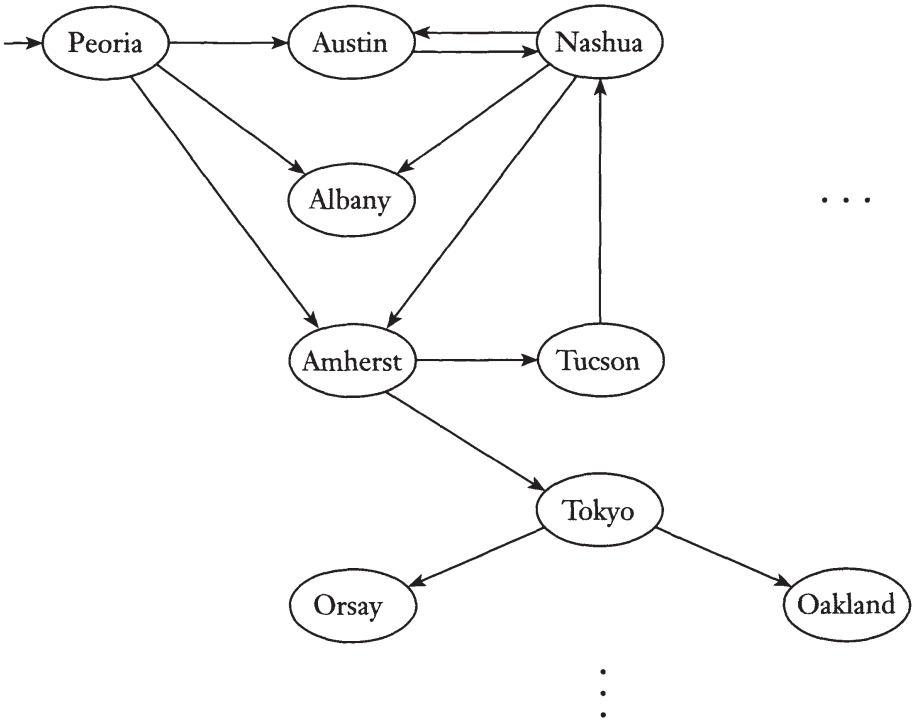


FIGURE 8.12
Portion of the graph representing the geography game

When the rules of geography are interpreted for this graphic representation, one player starts by selecting the designated start node and then the players take turns alternately by picking nodes that form a simple path in the graph. The requirement that the path be simple (i.e., doesn't use any node more than once) corresponds to the requirement that a city may not be repeated. The first player unable to extend the path loses the game.

In ***generalized geography*** we take an arbitrary directed graph with a designated start node instead of the graph associated with the actual cities. For example, the following graph is an example of a generalized geography game.

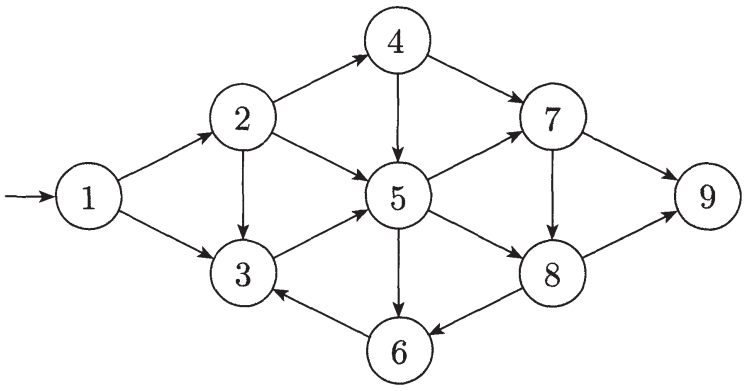


FIGURE 8.13
A sample generalized geography game

Say that Player I is the one who moves first and Player II second. In this example, Player I has a winning strategy as follows. Player I starts at node 1, the designated start node. Node 1 points only at nodes 2 and 3, so Player I's first move must be one of these two choices. He chooses 3. Now Player II must move, but node 3 points only to node 5, so she is forced to select node 5. Then Player I selects 6, from choices 6, 7, and 8. Now Player II must play from node 6, but it points only to node 3, and 3 was previously played. Player II is stuck, and thus Player I wins.

If we change the example by reversing the direction of the edge between nodes 3 and 6, Player II has a winning strategy. Can you see it? If Player I starts out with node 3 as before, Player II responds with 6 and wins immediately, so Player I's only hope is to begin with 2. In that case, however, Player II responds with 4. If Player I now takes 5, Player II wins with 6. If Player I takes 7, Player II wins with 9. No matter what Player I does, Player II can find a way to win, so Player II has a winning strategy.

The problem of determining which player has a winning strategy in a generalized geography game is PSPACE-complete. Let

$$GG = \{\langle G, b\rangle \mid \text{Player I has a winning strategy for the generalized geography game played on graph } G \text{ starting at node } b\}.$$

THEOREM 8.14

GG is PSPACE-complete.

PROOF IDEA A recursive algorithm similar to the one used for *TQBF* in Theorem 8.9 determines which player has a winning strategy. This algorithm runs in polynomial space and so $GG \in$ PSPACE.

To prove that *GG* is PSPACE-hard, we give a polynomial time reduction from *FORMULA-GAME* to *GG*. This reduction converts a formula game to a generalized geography graph so that play on the graph mimics play in the formula game. In effect, the players in the generalized geography game are really playing an encoded form of the formula game.

PROOF The following algorithm decides whether Player I has a winning strategy in instances of generalized geography; in other words, it decides *GG*. We show that it runs in polynomial space.

M = "On input $\langle G, b\rangle$, where G is a directed graph and b is a node of G:

1. If b has outdegree 0, *reject*, because Player I loses immediately.
2. Remove node b and all connected arrows to get a new graph G_1.
3. For each of the nodes $b_1, b_2, \ldots, b_k$ that b originally pointed at, recursively call M on $\langle G_1, b_i\rangle$.
4. If all of these accept, Player II has a winning strategy in the original game, so *reject*. Otherwise, Player II doesn't have a winning strategy, so Player I must; therefore *accept*."

The only space required by this algorithm is for storing the recursion stack. Each level of the recursion adds a single node to the stack, and at most m levels occur, where m is the number of nodes in G. Hence the algorithm runs in linear space.

To establish the PSPACE-hardness of *GG*, we show that *FORMULA-GAME* is polynomial time reducible to *GG*. The reduction maps the formula

$$\phi = \exists x_1 \, \forall x_2 \, \exists x_3 \cdots \mathsf{Q} x_k \, [\, \psi \,]$$

to an instance $\langle G, b \rangle$ of generalized geography. Here we assume for simplicity that ϕ's quantifiers begin and end with $\exists$ and that they strictly alternate between $\exists$ and $\forall$. A formula that doesn't conform to this assumption may be converted to a slightly larger one that does by adding extra quantifiers binding otherwise unused or "dummy" variables. We assume also that ψ is in conjunctive normal form (see Problem 8.12).

The reduction constructs a geography game on a graph G where optimal play mimics optimal play of the formula game on ϕ. Player I in the geography game takes the role of Player E in the formula game, and Player II takes the role of Player A.

The structure of graph G is partially shown in the following figure. Play starts at node b, which appears at the top left-hand side of G. Underneath b, a sequence of diamond structures appears, one for each of the variables of ϕ. Before getting to the right-hand side of G, let's see how play proceeds on the left-hand side.

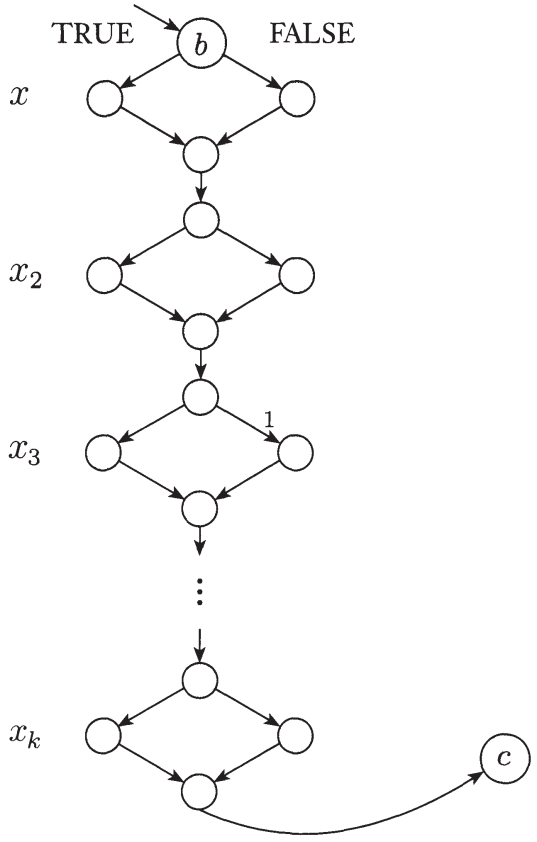


FIGURE **8.15**
Partial structure of the geography game simulating the formula game

Play starts at b. Player I must select one of the two edges going from b. These edges correspond to Player E's possible choices at the beginning of the formula game. The left-hand choice for Player I corresponds to TRUE for Player E in the formula game and the right-hand choice to FALSE. After Player I has selected one of these edges—say, the left-hand one—Player II moves. Only one outgoing edge is present, so this move is forced. Similarly, Player I's next move is forced and play continues from the top of the second diamond. Now two edges again are present, but Player II gets the choice. This choice corresponds to Player A's first move in the formula game. As play continues in this way, Players I and II choose a rightward or leftward path through each of the diamonds.

After play passes through all the diamonds, the head of the path is at the bottom node in the last diamond, and it is Player I's turn because we assumed that the last quantifier is $\exists$. Player I's next move is forced. Then they are at node c in Figure 8.15 and Player II makes the next move.

This point in the geography game corresponds to the end of play in the formula game. The chosen path through the diamonds corresponds to an assignment to ϕ's variables. Under that assignment, if ψ is TRUE, Player E wins the formula game, and if ψ is FALSE, Player A wins. The structure on the right-hand side of the following figure guarantees that Player I can win if Player E has won and that Player II can win if Player A has won.

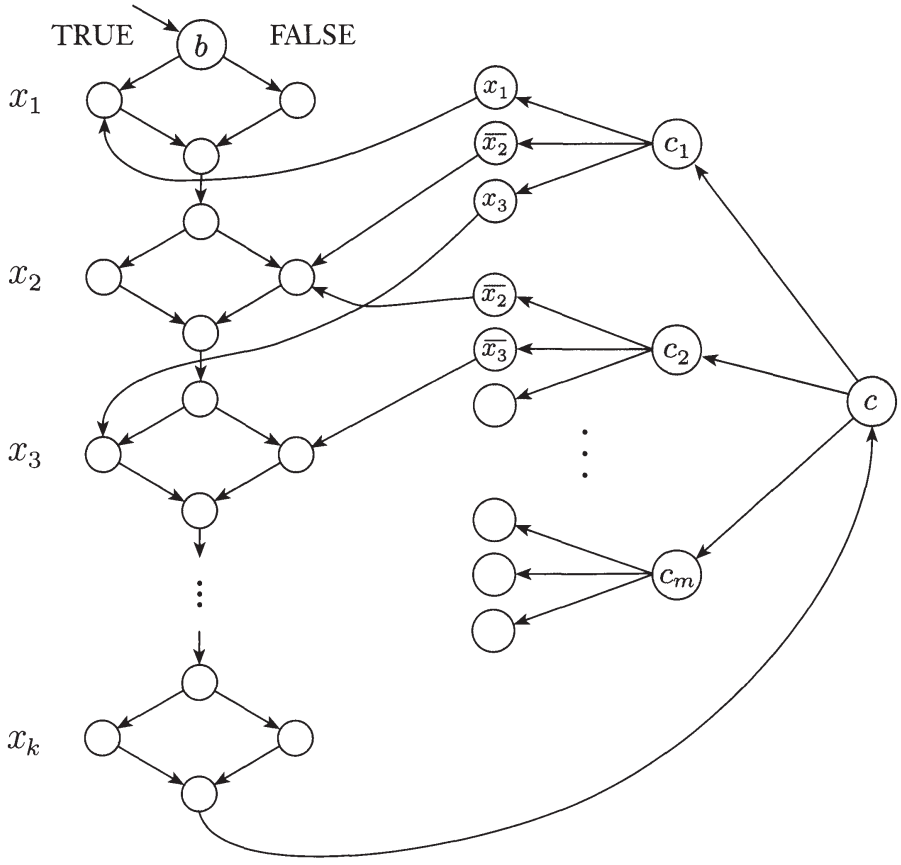


FIGURE 8.16
Full structure of the geography game simulating the formula game, where $\phi = \exists x_1 \forall x_2 \cdots \mathsf{Q} x_k\, [(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3} \vee \cdots) \wedge \cdots \wedge (\quad)]$

At node c, Player II may choose a node corresponding to one of ψ's clauses. Then Player I may choose a node corresponding to a literal in that clause. The nodes corresponding to unnegated literals are connected to the left-hand (TRUE) sides of the diamond for associated variables, and similarly for negated literals and right-hand (FALSE) sides as shown in Figure 8.16.

If ϕ is FALSE, Player II may win by selecting the unsatisfied clause. Any literal that Player I may then pick is FALSE and is connected to the side of the diamond that hasn't yet been played. Thus Player II may play the node in the diamond, but then Player I is unable to move and loses. If ϕ is TRUE, any clause that Player II picks contains a TRUE literal. Player I selects that literal after Player II's move. Because the literal is TRUE, it is connected to the side of the diamond that has already been played, so Player II is unable to move and loses.

In Theorem 8.14 we showed that no polynomial time algorithm exists for optimal play in generalized geography unless P = PSPACE. We'd like to prove a similar theorem regarding the difficulty of computing optimal play in board games such as chess, but an obstacle arises. Only a finite number of different game positions may occur on the standard 8×8 chess board. In principle, all these positions may be placed in a table, along with the best move in each position. The table would be too large to fit inside our galaxy but, being finite, could be stored in the control of a Turing machine (or even that of a finite automaton!). Thus the machine would be able to play optimally in linear time, using table lookup. Perhaps at some time in the future, methods that can quantify the complexity of finite problems will be developed, but current methods are asymptotic and hence apply only to the rate of growth of the complexity as the problem size increases—not to any fixed size. Nevertheless, we can give some evidence for the difficulty of computing optimal play for many board games by generalizing them to an $n \times n$ board. Such generalizations of chess, checkers, and GO have been shown to be PSPACE-hard or hard for even larger complexity classes, depending on the details of the generalization.

8.4 THE CLASSES L AND NL

Until now, we have considered only time and space complexity bounds that are at least linear—that is, bounds where $f(n)$ is at least n. Now we examine smaller, *sublinear* space bounds. In time complexity, sublinear bounds are insufficient for reading the entire input, so we don't consider them here. In sublinear space complexity the machine is able to read the entire input but it doesn't have enough space to store the input. To consider this situation meaningfully, we must modify our computational model.

We introduce a Turing machine with two tapes: a read-only input tape and a read/write work tape. On the read-only tape the input head can detect symbols but not change them. We provide a way for the machine to detect when the head is at the left-hand and right-hand ends of the input. The input head must remain on the portion of the tape containing the input. The work tape may be read and written in the usual way. Only the cells scanned on the work tape contribute to the space complexity of this type of Turing machine.

Think of a read-only input tape as a CD-ROM, a device used for input on many personal computers. Often, the CD-ROM contains more data than the computer can store in its main memory. Sublinear space algorithms allow the computer to manipulate the data without storing all of it in main memory.

For space bounds that are at least linear, the two-tape TM model is equivalent to the standard one-tape model (see Exercise 8.1). For sublinear space bounds, we use only the two-tape model.

DEFINITION 8.17

L is the class of languages that are decidable in logarithmic space on a deterministic Turing machine. In other words,

$$\mathrm{L} = \mathrm{SPACE}(\log n).$$

NL is the class of languages that are decidable in logarithmic space on a nondeterministic Turing machine. In other words,

$$\mathrm{NL} = \mathrm{NSPACE}(\log n).$$

We focus on $\log n$ space instead of, say, $\sqrt{n}$ or $\log^2 n$ space, for several reasons that are similar to those for our selection of polynomial time and space bounds. Logarithmic space is just large enough to solve a number of interesting computational problems, and it has attractive mathematical properties such as robustness even when machine model and input encoding method change. Pointers into the input may be represented in logarithmic space, so one way to think about the power of log space algorithms is to consider the power of a fixed number of input pointers.

EXAMPLE 8.18

The language $A = \{0^k 1^k \mid k \geq 0\}$ is a member of L. In Section 7.1 on page 247 we described a Turing machine that decides A by zigzagging back and forth across the input, crossing off the 0s and 1s as they are matched. That algorithm uses linear space to record which positions have been crossed off, but it can be modified to use only log space.

The log space TM for A cannot cross off the 0s and 1s that have been matched on the input tape because that tape is read-only. Instead, the machine counts the number of 0s and, separately, the number of 1s in binary on the work tape. The only space required is that used to record the two counters. In binary, each counter uses only logarithmic space, and hence the algorithm runs in $O(\log n)$ space. Therefore $A \in$ L. ▒

EXAMPLE 8.19

Recall the language

$$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$$

defined in Section 7.2. Theorem 7.14 shows that *PATH* is in P but that the algorithm given uses linear space. We don't know whether *PATH* can be solved in logarithmic space deterministically, but we do know a nondeterministic log space algorithm for *PATH*.

The nondeterministic log space Turing machine deciding *PATH* operates by starting at node s and nondeterministically guessing the nodes of a path from s to t. The machine records only the position of the current node at each step on the work tape, not the entire path (which would exceed the logarithmic space requirement). The machine nondeterministically selects the next node from among those pointed at by the current node. Then it repeats this action until it reaches node t and *accepts*, or until it has gone on for m steps and *rejects*, where m is the number of nodes in the graph. Thus *PATH* is in NL. ▒

Our earlier claim that any $f(n)$ space bounded Turing machine also runs in time $2^{O(f(n))}$ is no longer true for very small space bounds. For example, a Turing machine that uses $O(1)$ (i.e., constant) space may run for n steps. To obtain a bound on the running time that applies for every space bound $f(n)$ we give the following definition.

DEFINITION 8.20

If M is a Turing machine that has a separate read-only input tape and w is an input, a ***configuration of M on w*** is a setting of the state, the work tape, and the positions of the two tape heads. The input w is not a part of the configuration of M on w.

If M runs in $f(n)$ space and w is an input of length n, the number of configurations of M on w is $n2^{O(f(n))}$. To explain this result, let's say that M has c states and g tape symbols. The number of strings that can appear on the work tape is $g^{f(n)}$. The input head can be in one of n positions and the work tape head can

be in one of $f(n)$ positions. Therefore the total number of configurations of M on w, which is an upper bound on the running time of M on w, is $cnf(n)g^{f(n)}$, or $n2^{O(f(n))}$.

We focus almost exclusively on space bounds $f(n)$ that are at least $\log n$. Our earlier claim that the time complexity of a machine is at most exponential in its space complexity remains true for such bounds because $n2^{O(f(n))}$ is $2^{O(f(n))}$ when $f(n) \geq \log n$.

Recall that Savitch's theorem shows that we can convert nondeterministic TMs to deterministic TMs and increase the space complexity $f(n)$ by only a squaring, provided that $f(n) \geq n$. We can extend Savitch's theorem to hold for sublinear space bounds down to $f(n) \geq \log n$. The proof is identical to the original one we gave on page 305, except that we use Turing machines with a read-only input tape and instead of referring to configurations of N we refer to configurations of N on w. Storing a configuration of N on w uses $\log(n2^{O(f(n))}) = \log n + O(f(n))$ space. If $f(n) \geq \log n$, the storage used is $O(f(n))$ and the remainder of the proof remains the same.

8.5 NL-COMPLETENESS

As we mentioned in Example 8.19, the *PATH* problem is known to be in NL but isn't known to be in L. We believe that *PATH* doesn't belong to L, but we don't know how to prove this conjecture. In fact, we don't know of any problem in NL that can be proven to be outside L. Analogous to the question of whether P = NP we have the question of whether L = NL.

As a step toward resolving the L versus NL question, we can exhibit certain languages that are NL-complete. As with complete languages for other complexity classes, the NL-complete languages are examples of languages that are, in a certain sense, the most difficult languages in NL. If L and NL are different, all NL-complete languages don't belong to L.

As with our previous definitions of completeness, we define an NL-complete language to be one which is in NL and to which any other language in NL is reducible. However, we don't use polynomial time reducibility here because, as you will see, all problems in NL are solvable in polynomial time. Therefore every two problems in NL except $\emptyset$ and Σ^* are polynomial time reducible to one another (see the discussion of polynomial time reducibility in the definition of PSPACE-completeness on page 309). Hence polynomial time reducibility is too strong to differentiate problems in NL from one another. Instead we use a new type of reducibility called *log space reducibility*.

DEFINITION 8.21

A ***log space transducer*** is a Turing machine with a read-only input tape, a write-only output tape, and a read/write work tape. The work tape may contain $O(\log n)$ symbols. A log space transducer M computes a function $f\colon \Sigma^* \longrightarrow \Sigma^*$, where $f(w)$ is the string remaining on the output tape after M halts when it is started with w on its input tape. We call f a ***log space computable function***. Language A is ***log space reducible*** to language B, written $A \leq_{\mathrm{L}} B$, if A is mapping reducible to B by means of a log space computable function f.

Now we are ready to define NL-completeness.

DEFINITION 8.22

A language B is ***NL-complete*** if

1. $B \in$ NL, and
2. every A in NL is log space reducible to B.

If one language is log space reducible to another language already known to be in L, the original language is also in L, as the following theorem demonstrates.

THEOREM 8.23

If $A \leq_{\mathrm{L}} B$ and $B \in$ L, then $A \in$ L.

PROOF A tempting approach to the proof of this theorem is to follow the model presented in Theorem 7.31, the analogous result for polynomial time reducibility. In that approach, a log space algorithm for A first maps its input w to $f(w)$, using the log space reduction f, and then applies the log space algorithm for B. However, the storage required for $f(w)$ may be too large to fit within the log space bound, so we need to modify this approach.

Instead, A's machine M_A computes individual symbols of $f(w)$ as requested by B's machine M_B. In the simulation, M_A keeps track of where M_B's input head would be on $f(w)$. Every time M_B moves, M_A restarts the computation of f on w from the beginning and ignores all the output except for the desired location of $f(w)$. Doing so may require occasional recomputation of parts of $f(w)$ and so is inefficient in its time complexity. The advantage of this method is that only a single symbol of $f(w)$ needs to be stored at any point, in effect trading time for space.

COROLLARY 8.24

If any NL-complete language is in L, then L = NL.

SEARCHING IN GRAPHS

THEOREM 8.25

PATH is NL-complete.

PROOF IDEA Example 8.19 shows that *PATH* is in NL, so we only need to show that *PATH* is NL-hard. In other words, we must show that every language A in NL is log space reducible to *PATH*.

The idea behind the log space reduction from A to *PATH* is to construct a graph that represents the computation of the nondeterministic log space Turing machine for A. The reduction maps a string w to a graph whose nodes correspond to the configurations of the NTM on input w. One node points to a second node if the corresponding first configuration can yield the second configuration in a single step of the NTM. Hence the machine accepts w whenever some path from the node corresponding to the start configuration leads to the node corresponding to the accepting configuration.

PROOF We show how to give a log space reduction from any language A in NL to *PATH*. Let's say that NTM M decides A in $O(\log n)$ space. Given an input w, we construct $\langle G, s, t\rangle$ in log space, where G is a directed graph that contains a path from s to t if and only if M accepts w.

The nodes of G are the configurations of M on w. For configurations c_1 and c_2 of M on w, the pair (c_1, c_2) is an edge of G if c_2 is one of the possible next configurations of M starting from c_1. More precisely, if M's transition function indicates that c_1's state together with the tape symbols under its input and work tape heads can yield the next state and head actions to make c_1 into c_2, then (c_1, c_2) is an edge of G. Node s is the start configuration of M on w. Machine M is modified to have a unique accepting configuration, and we designate this configuration to be node t.

This mapping reduces A to *PATH* because, whenever M accepts its input, some branch of its computation accepts, which corresponds to a path from the start configuration s to the accepting configuration t in G. Conversely, if some path exists from s to t in G, some computation branch accepts when M runs on input w, and M accepts w.

To show that the reduction operates in log space, we give a log space transducer which, on input w, outputs a description of G. This description comprises two lists: G's nodes and G's edges. Listing the nodes is easy because each node is a configuration of M on w and can be represented in $c \log n$ space for some constant c. The transducer sequentially goes through all possible strings of length

$c \log n$, tests whether each is a legal configuration of M on w, and outputs those that pass the test. The transducer lists the edges similarly. Log space is sufficient for verifying that a configuration c_1 of M on w can yield configuration c_2 because the transducer only needs to examine the actual tape contents under the head locations given in c_1 to determine that M's transition function would give configuration c_2 as a result. The transducer tries all pairs (c_1, c_2) in turn to find which qualify as edges of G. Those that do are added to the output tape.

One immediate spinoff of Theorem 8.25 is the following corollary which states that NL is a subset of P.

COROLLARY 8.26

NL $\subseteq$ P.

PROOF Theorem 8.25 shows that any language in NL is log space reducible to *PATH*. Recall that a Turing machine that uses space $f(n)$ runs in time $n\,2^{O(f(n))}$, so a reducer that runs in log space also runs in polynomial time. Therefore any language in NL is polynomial time reducible to *PATH*, which in turn is in P, by Theorem 7.14. We know that every language that is polynomial time reducible to a language in P is also in P, so the proof is complete.

Though log space reducibility appears to be highly restrictive, it is adequate for most reductions in complexity theory, because these are usually computationally simple. For example, in Theorem 8.9 we showed that every PSPACE problem is polynomial time reducible to *TQBF*. The highly repetitive formulas that these reductions produce may be computed using only log space, and therefore we may conclude that *TQBF* is PSPACE-complete with respect to log space reducibility. This conclusion is important because Corollary 9.6 shows that NL $\subsetneq$ PSPACE. This separation and log space reducibility implies that *TQBF* $\notin$ NL.

8.6 NL EQUALS CONL

This section contains one of the most surprising results known concerning the relationships among complexity classes. The classes NP and coNP are generally believed to be different. At first glance, the same appears to hold for the classes NL and coNL. The fact that NL equals coNL, as we are about to prove, shows that our intuition about computation still has many gaps in it.

THEOREM 8.27

NL = coNL.

PROOF IDEA We show that $\overline{PATH}$ is in NL, and thereby establish that every problem in coNL is also in NL, because *PATH* is NL-complete. The NL algorithm M that we present for $\overline{PATH}$ must have an accepting computation whenever the input graph G does *not* contain a path from s to t.

First, let's tackle an easier problem. Let c be the number of nodes in G that are reachable from s. We assume that c is provided as an input to M and show how to use c to solve $\overline{PATH}$. Later we show how to compute c.

Given G, s, t, and c, the machine M operates as follows. One by one, M goes through all the m nodes of G and nondeterministically guesses whether each one is reachable from s. Whenever a node u is guessed to be reachable, M attempts to verify this guess by guessing a path of length m or less from s to u. If a computation branch fails to verify this guess, it rejects. In addition, if a branch guesses that t is reachable, it rejects. Machine M counts the number of nodes that have been verified to be reachable. When a branch has gone through all of G's nodes, it checks that the number of nodes that it verified to be reachable from s equals c, the number of nodes that actually are reachable, and rejects if not. Otherwise, this branch accepts.

In other words, if M nondeterministically selects exactly c nodes reachable from s, not including t, and proves that each is reachable from s by guessing the path, M knows that the remaining nodes, including t, are *not* reachable, so it can accept.

Next, we show how to calculate c, the number of nodes reachable from s. We describe a nondeterministic log space procedure whereby at least one computation branch has the correct value for c and all other branches reject.

For each i from 0 to m, we define A_i to be the collection of nodes that are at a distance of i or less from s (i.e., that have a path of length at most i from s). So $A_0 = \{s\}$, each $A_i \subseteq A_{i+1}$, and A_m contains all nodes that are reachable from s. Let c_i be the number of nodes in A_i. We next describe a procedure that calculates c_{i+1} from c_i. Repeated application of this procedure yields the desired value of $c = c_m$.

We calculate c_{i+1} from c_i, using an idea similar to the one presented earlier in this proof sketch. The algorithm goes through all the nodes of G, determines whether each is a member of A_{i+1}, and counts the members.

To determine whether a node v is in A_{i+1}, we use an inner loop to go through all the nodes of G and guess whether each node is in A_i. Each positive guess is verified by guessing the path of length at most i from s. For each node u verified to be in A_i, the algorithm tests whether (u, v) is an edge of G. If it is an edge, v is in A_{i+1}. Additionally, the number of nodes verified to be in A_i is counted. At the completion of the inner loop, if the total number of nodes verified to be in A_i is not c_i, all A_i have not been found, so this computation branch rejects. If the count equals c_i and v has not yet been shown to be in A_{i+1}, we conclude that it isn't in A_{i+1}. Then we go on to the next v in the outer loop.

PROOF Here is an algorithm for $\overline{PATH}$. Let m be the number of nodes of G.

M = "On input $\langle G, s, t \rangle$:

1. Let $c_0 = 1$. [[$A_0 = \{s\}$ has 1 node]]
2. For $i = 0$ to $m - 1$: [[compute c_{i+1} from c_i]]
3. Let $c_{i+1} = 1$. [[c_{i+1} counts nodes in A_{i+1}]]
4. For each node $v \neq s$ in G: [[check if $v \in A_{i+1}$]]
5. Let $d = 0$. [[d re-counts A_i]]
6. For each node u in G: [[check if $u \in A_i$]]
7. Nondeterministically either perform or skip these steps:
8. Nondeterministically follow a path of length at most i from s and *reject* if it doesn't end at u.
9. Increment d. [[verified that $u \in A_i$]]
10. If (u, v) is an edge of G, increment c_{i+1} and go to Stage 5 with the next v. [[verified that $v \in A_{i+1}$]]
11. If $d \neq c_i$, then *reject*. [[check whether found all A_i]]
12. Let $d = 0$. [[c_m now known; d re-counts A_m]]
13. For each node u in G: [[check if $u \in A_m$]]
14. Nondeterministically either perform or skip these steps:
15. Nondeterministically follow a path of length at most m from s and *reject* if it doesn't end at u.
16. If $u = t$, then *reject*. [[found path from s to t]]
17. Increment d. [[verified that $u \in A_m$]]
18. If $d \neq c_m$, then *reject*. [[check that found all of A_m]]
    Otherwise, *accept*."

This algorithm only needs to store u, v, c_i, c_{i+1}, d, i, and a pointer to the head of a path, at any given time. Hence it runs in log space. (Note that M accepts improperly formed inputs, too.)

We summarize our present knowledge of the relationships among several complexity classes as follows:

$$\text{L} \subseteq \text{NL} = \text{coNL} \subseteq \text{P} \subseteq \text{PSPACE}.$$

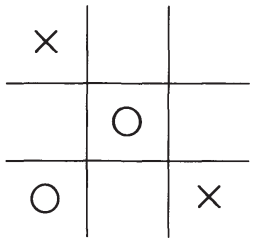
We don't know whether any of these containments are proper, although we prove NL $\subsetneq$ PSPACE[3] in Corollary 9.6. Consequently, either coNL $\subsetneq$ P or P $\subsetneq$ PSPACE must hold, but we don't know which one does! Most researchers conjecture that all these containments are proper.

[3]We write $A \subsetneq B$ to mean that A is a proper (i.e., not equal) subset of B.

EXERCISES

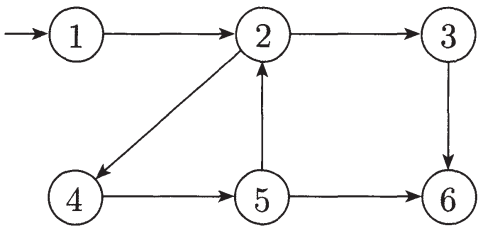
8.1 Show that for any function $f: \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq n$, the space complexity class SPACE($f(n)$) is the same whether you define the class by using the single-tape TM model or the two tape read-only input TM model.

8.2 Consider the following position in the standard tic-tac-toe game.



Let's say that it is the X-player's turn to move next. Describe the winning strategy for this player. (Recall that a winning strategy isn't merely the best move to make in the current position. It also includes all the responses that this player must make in order to win, however the opponent moves.)

8.3 Consider the following generalized geography game wherein the start node is the one with the arrow pointing in from nowhere. Does Player I have a winning strategy? Does Player II? Give reasons for your answers.



8.4 Show that PSPACE is closed under the operations union, complementation, and star.

A8.5 Show that NL is closed under the operations union, intersection, and star.

8.6 Show that any PSPACE-hard language is also NP-hard.

A8.7 Show that $A_{\text{DFA}} \in$ L.

PROBLEMS

8.8 Let $EQ_{\text{REX}} = \{\langle R, S\rangle \mid R \text{ and } S \text{ are equivalent regular expressions}\}$. Show that $EQ_{\text{REX}} \in$ PSPACE.

8.9 A *ladder* is a sequence of strings $s_1, s_2, \ldots, s_k$, wherein every string differs from the preceding one in exactly one character. For example the following is a ladder of English words, starting with "head" and ending with "free":

head, hear, near, fear, bear, beer, deer, deed, feed, feet, fret, free.

Let $LADDER_{\mathsf{DFA}} = \{\langle M, s, t\rangle | \ M \text{ is a DFA and } L(M) \text{ contains a ladder of strings, starting with } s \text{ and ending with } t\}$. Show that $LADDER_{\mathsf{DFA}}$ is in PSPACE.

8.10 The Japanese game *go-moku* is played by two players, "X" and "O," on a 19×19 grid. Players take turns placing markers, and the first player to achieve 5 of his markers consecutively in a row, column, or diagonal, is the winner. Consider this game generalized to an $n \times n$ board. Let

$$GM = \{\langle B\rangle | \ B \text{ is a position in generalized go-moku, where player "X" has a winning strategy}\}.$$

By a *position* we mean a board with markers placed on it, such as may occur in the middle of a play of the game, together with an indication of which player moves next. Show that $GM \in$ PSPACE.

8.11 Show that, if every NP-hard language is also PSPACE-hard, then PSPACE = NP.

8.12 Show that *TQBF* restricted to formulas where the part following the quantifiers is in conjunctive normal form is still PSPACE-complete.

8.13 Define $A_{\mathsf{LBA}} = \{\langle M, w\rangle | \ M \text{ is an LBA that accepts input } w\}$. Show that A_{LBA} is PSPACE-complete.

8.14 Consider the following two-person version of the language *PUZZLE* that was described in Problem 7.26. Each player starts with an ordered stack of puzzle cards. The players take turns placing the cards in order in the box and may choose which side faces up. Player I wins if, in the final stack, all hole positions are blocked, and Player II wins if some hole position remains unblocked. Show that the problem of determining which player has a winning strategy for a given starting configuration of the cards is PSPACE-complete.

***8.15** The cat-and-mouse game is played by two players, "Cat" and "Mouse," on an arbitrary undirected graph. At a given point each player occupies a node of the graph. The players take turns moving to a node adjacent to the one that they currently occupy. A special node of the graph is called "Hole." Cat wins if the two players ever occupy the same node. Mouse wins if it reaches the Hole before the preceding happens. The game is a draw if a situation repeats (i.e., the two players simultaneously occupy positions that they simultaneously occupied previously and it is the same player's turn to move).

$$HAPPY\text{-}CAT = \{\langle G, c, m, h\rangle | \ G, c, m, h, \text{ are respectively a graph, and positions of the Cat, Mouse, and Hole, such that Cat has a winning strategy if Cat moves first}\}.$$

Show that *HAPPY-CAT* is in P. (Hint: The solution is not complicated and doesn't depend on subtle details in the way the game is defined. Consider the entire game tree. It is exponentially big, but you can search it in polynomial time.)

8.16 Read the definition of *MIN-FORMULA* in Problem 7.44.

a. Show that *MIN-FORMULA* $\in$ PSPACE.

b. Explain why this argument fails to show that *MIN-FORMULA* $\in$ coNP: If $\phi \notin$ *MIN-FORMULA*, then ϕ has a smaller equivalent formula. An NTM can verify that $\phi \in$ *MIN-FORMULA* by guessing that formula.

8.17 Let A be the language of properly nested parentheses. For example, `(())` and `((()(()))()` are in A, but `)(` is not. Show that A is in L.

***8.18** Let B be the language of properly nested parentheses and brackets. For example, `([()()]()[])` is in B but `([)]` is not. Show that B is in L.

8.19** The game of ***Nim is played with a collection of piles of sticks. In one move a player may remove any nonzero number of sticks from a single pile. The players alternately take turns making moves. The player who removes the very last stick loses. Say that we have a game position in Nim with k piles containing $s_1, \ldots, s_k$ sticks. Call the position ***balanced*** if, when each of the numbers s_i is written in binary and the binary numbers are written as rows of a matrix aligned at the low order bits, each column of bits contains an even number of 1s. Prove the following two facts.

a. Starting in an unbalanced position, a single move exists that changes the position into a balanced one.

b. Starting in a balanced position, every single move changes the position into an unbalanced one.

Let $NIM = \{\langle s_1, \ldots, s_k\rangle |$ each s_i is a binary number and Player I has a winning strategy in the Nim game starting at this position$\}$. Use the preceding facts about balanced positions to show that $NIM \in$ L.

8.20 Let $MULT = \{a\#b\#c |$ where a, b, c are binary natural numbers and $a \times b = c\}$. Show that $MULT \in$ L.

8.21 For any positive integer x, let $x^{\mathcal{R}}$ be the integer whose binary representation is the reverse of the binary representation of x. (Assume no leading 0s in the binary representation of x.) Define the function $\mathcal{R}^+ : \mathcal{N} \longrightarrow \mathcal{N}$ where $\mathcal{R}^+(x) = x + x^{\mathcal{R}}$.

a. Let $A_2 = \{\langle x, y\rangle | \ \mathcal{R}^+(x) = y\}$. Show $A_2 \in$ L.

b. Let $A_3 = \{\langle x, y\rangle | \ \mathcal{R}^+(\mathcal{R}^+(x)) = y\}$. Show $A_3 \in$ L.

8.22

a. Let $ADD = \{\langle x, y, z\rangle | \ x, y, z > 0$ are binary integers and $x + y = z\}$. Show that $ADD \in$ L.

b. Let *PAL-ADD* $= \{\langle x, y\rangle | \ x, y > 0$ are binary integers where $x + y$ is an integer whose binary representation is a palindrome$\}$. (Note that the binary representation of the sum is assumed not to have leading zeros. A palindrome is a string that equals its reverse). Show that *PAL-ADD* $\in$ L.

***8.23** Define $UCYCLE = \{\langle G\rangle |$ G is an undirected graph that contains a simple cycle$\}$. Show that $UCYCLE \in$ L. (Note: G may be a graph that is not connected.)

***8.24** For each n, exhibit two regular expressions, R and S, of length $poly(n)$, where $L(R) \neq L(S)$, but where the first string on which they differ is exponentially long. In other words, $L(R)$ and $L(S)$ must be different, yet agree on all strings of length $2^{\epsilon n}$ for some constant $\epsilon > 0$.

8.25 An undirected graph is ***bipartite*** if its nodes may be divided into two sets so that all edges go from a node in one set to a node in the other set. Show that a graph is bipartite if and only if it doesn't contain a cycle that has an odd number of nodes. Let $BIPARTITE = \{\langle G \rangle | \ G \text{ is a bipartite graph}\}$. Show that $BIPARTITE \in$ NL.

8.26 Define *UPATH* to be the counterpart of *PATH* for undirected graphs. Show that $BIPARTITE \leq_L UPATH$. (Note: As this edition was going to press, O. Reingold [60] announced that $UPATH \in$ L. Consequently, $BIPARTITE \in$ L, but the algorithm is somewhat complicated.)

8.27 Recall that a directed graph is ***strongly connected*** if every two nodes are connected by a directed path in each direction. Let

$$STRONGLY\text{-}CONNECTED = \{\langle G \rangle | \ G \text{ is a strongly connected graph}\}.$$

Show that *STRONGLY-CONNECTED* is NL-complete.

8.28 Let $BOTH_{\mathsf{NFA}} = \{\langle M_1, M_2 \rangle | \ M_1 \text{ and } M_2 \text{ are NFAs where } L(M_1) \cap L(M_2) \neq \emptyset\}$. Show that $BOTH_{\mathsf{NFA}}$ is NL-complete.

8.29 Show that A_{NFA} is NL-complete.

8.30 Show that E_{DFA} is NL-complete.

*__8.31__ Show that *2SAT* is NL-complete.

*__8.32__ Give an example of an NL-complete context-free language.

A*__8.33__ Define $CYCLE = \{\langle G \rangle | \ G \text{ is a directed graph that contains a directed cycle}\}$. Show that *CYCLE* is NL-complete.

SELECTED SOLUTIONS

8.5 Let A_1 and A_2 be languages that are decided by NL-machines N_1 and N_2. Construct three Turing machines: $N_\cup$ deciding $A_1 \cup A_2$; $N_\circ$ deciding $A_1 \circ A_2$; and N_* deciding A_1^*. Each of these machines receives input w.

Machine $N_\cup$ nondeterministically branches to simulate N_1 or to simulate N_2. In either case, $N_\cup$ accepts if the simulated machine accepts.

Machine $N_\circ$ nondeterministically selects a position on the input to divide it into two substrings. Only a pointer to that position is stored on the work tape—insufficient space is available to store the substrings themselves. Then $N_\circ$ simulates N_1 on the first substring, branching nondeterministically to simulate N_1's nondeterminism. On any branch that reaches N_1's accept state, $N_\circ$ simulates N_2 on the second substring. On any branch that reaches N_2's accept state, $N_\circ$ accepts.

Machine N_* has a more complex algorithm, so we describe its stages.

N_* = "On input w:

1. Initialize two input position pointers p_1 and p_2 to 0, the position immediately preceding the first input symbol.
2. *Accept* if no input symbols occur after p_2.
3. Move p_2 forward to a nondeterministically selected input position.

4. Simulate N_1 on the substring of w from the position following p_1 to the position at p_2, branching nondeterministically to simulate N_1's nondeterminism.
5. If this branch of the simulation reaches N_1's accept state, copy p_2 to p_1 and go to stage 2."

8.7 Construct a TM M to decide A_{DFA}. When M receives input $\langle A, w\rangle$, a DFA and a string, M simulates A on w by keeping track of A's current state and its current head location, and updating them appropriately. The space required to carry out this simulation is $O(\log n)$ because M can record each of these values by storing a pointer into its input.

8.33 Reduce *PATH* to *CYCLE*. The idea behind the reduction is to modify the *PATH* problem instance $\langle G, s, t\rangle$ by adding an edge from t to s in G. If a path exists from s to t in G, a directed cycle will exist in the modified G. However, other cycles may exist in the modified G because they may already be present in G. To handle that problem, first change G so that it contains no cycles. A ***leveled directed graph*** is one where the nodes are divided into groups, $A_1, A_2, \ldots, A_k$, called *levels*, and only edges from one level to the next higher level are permitted. Observe that a leveled graph is acyclic. The *PATH* problem for leveled graphs is still NL-complete, as the following reduction from the unrestricted *PATH* problem shows. Given a graph G with two nodes s and t, and m nodes in total, produce the leveled graph G' whose levels are m copies of G's nodes. Draw an edge from node i at each level to node j in the next level if G contains an edge from i to j. Additionally, draw an edge from node i in each level to node i in the next level. Let s' be the node s in the first level and let t' be the node t in the last level. Graph G contains a path from s to t iff G' contains a path from s' to t'. If you modify G' by adding an edge from t' to s', you obtain a reduction from *PATH* to *CYCLE*. The reduction is computationally simple, and its implementation in logspace is routine. Furthermore, a straightforward procedure shows that $CYCLE \in$ NL. Hence *CYCLE* is NL-complete.