CHAPTER 8

Complexity Classes

In an ideal world, each computational problem would be classified at least approximately by its use of computational resources. Unfortunately, our ability to so classify some important problems is limited. We must be content to show that such problems fall into general complexity classes, such as the polynomial-time problems **P**, problems whose running time on a deterministic Turing machine is a polynomial in the length of its input, or **NP**, the polynomial-time problems on nondeterministic Turing machines.

Many complexity classes contain "complete problems," problems that are hardest in the class. If the complexity of one complete problem is known, that of all complete problems is known. Thus, it is very useful to know that a problem is complete for a particular complexity class. For example, the class of **NP**-complete problems, the hardest problems in **NP**, contains many hundreds of important combinatorial problems such as the Traveling Salesperson Problem. It is known that each **NP**-complete problem can be solved in time exponential in the size of the problem, but it is not known whether they can be solved in polynomial time. Whether **P** and **NP** are equal or not is known as the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ question. Decades of research have been devoted to this question without success. As a consequence, knowing that a problem is **NP**-complete is good evidence that it is an exponential-time problem. On the other hand, if one such problem were shown to be in **P**, all such problems would be been shown to be in **P**, a result that would be most important.

In this chapter we classify problems by the resources they use on serial and parallel machines. The serial models are the Turing and random-access machines. The parallel models are the circuit and the parallel random-access machine (PRAM). We begin with a discussion of tasks, machine models, and resource measures, after which we examine serial complexity classes and relationships among them. Complete problems are defined and the **P**-complete, **NP**-complete, and **PSPACE**-complete problems are examined. We then turn to the PRAM and circuit models and conclude by identifying important circuit complexity classes such as **NC** and **P/poly**.

8.1 Introduction

The classification of problems requires a precise definition of those problems and the computational models used. Problems are accurately classified only when we are sure that they have been well defined and that the computational models against which they are classified are representative of the computational environment in which these problems will be solved. This requires the computational models to be general. On the other hand, to be useful, problem classifications should not be overly dependent on the characteristics of the machine model used for classification purposes. For example, because of the obviously inefficient use of memory on the Turing machine, the set of problems that runs in time linear in the length of their input on a random-access machine is likely to be different from the set that runs in linear time on the Turing machine. On the other hand, the set of problems that run in polynomial time on both machines is the same.

8.2 Languages and Problems

Before formally defining decision problems, a major topic of this chapter, we give two examples of them, SATISFIABILITY and UNSATISFIABILITY. A set of clauses is **satisfiable** if values can be assigned to Boolean variables in these clauses such that each clause has at least one literal with value 1.

SATISFIABILITY
Instance: A set of literals $X = \{x_1, \overline{x}_1, x_2, \overline{x}_2, \ldots, x_n, \overline{x}_n\}$, and a sequence of clauses $C = (c_1, c_2, \ldots, c_m)$ where each clause c_i is a subset of X.
Answer: "Yes" if for some assignment of Boolean values to variables in $\{x_1, x_2, \ldots, x_n\}$, at least one literal in each clause has value 1.

The complement of the decision problem SATISFIABILITY, UNSATISFIABILITY, is defined below.

UNSATISFIABILITY
Instance: A set of literals $X = \{x_1, \overline{x}_1, x_2, \overline{x}_2, \ldots, x_n, \overline{x}_n\}$, and a sequence of clauses $C = (c_1, c_2, \ldots, c_m)$ where each clause c_i is a subset of X.
Answer: "Yes" if for all assignments of Boolean values to variables in $\{x_1, x_2, \ldots, x_n\}$, all literals in at least one clause have value 0.

The clauses $C_1 = (\{x_1, x_2, x_3\}, \{x_1, \overline{x}_2\}, \{x_2, \overline{x}_3\})$ are satisfied with $x_1 = x_2 = x_3 = 1$, whereas the clauses $C_2 = (\{x_1, x_2, x_3\}, \{x_1, \overline{x}_2\}, \{x_2, \overline{x}_3\}, \{x_3, \overline{x}_1\}, \{\overline{x}_1, \overline{x}_2, \overline{x}_3\})$ are not satisfiable. SATISFIABILITY consists of collections of satisfiable clauses. C_1 is in SATISFIABILITY. The complement of SATISFIABILITY, UNSATISFIABILITY, consists of instances of clauses not all of which can be satisfied. C_2 is in UNSATISFIABILITY.

We now introduce terminology used to classify problems. This terminology and the associated concepts are used throughout this chapter.

DEFINITION 8.2.1 *Let Σ be an arbitrary finite alphabet. A* **decision problem** *$\mathcal{P}$ is defined by a set of* **instances** *$I \subseteq \Sigma^*$ of the problem and a condition $\phi_{\mathcal{P}} : I \mapsto \mathcal{B}$ that has value 1 on "Yes" instances and 0 on "No" instances. Then $I_{\text{yes}} = \{\boldsymbol{w} \in I \mid \phi_{\mathcal{P}}(\boldsymbol{w}) = 1\}$ are the "Yes" instances. The "No" instances are $I_{\text{no}} = I - I_{\text{yes}}$.*

The **complement of a decision problem** $\mathcal{P}$*, denoted* **co**$\mathcal{P}$*, is the decision problem in which the "Yes" instances of* **co**$\mathcal{P}$ *are the "No" instances of* $\mathcal{P}$ *and vice versa.*

The "Yes" instances of a decision problem are encoded as binary strings by an **encoding function** $\sigma : \Sigma^* \mapsto \mathcal{B}^*$ *that assigns to each* $\boldsymbol{w} \in I$ *a string* $\sigma(\boldsymbol{w}) \in \mathcal{B}^*$.

With respect to σ*, the* **language** $L(\mathcal{P})$ **associated with a decision problem** $\mathcal{P}$ *is the set* $L(\mathcal{P}) = \{\sigma(\boldsymbol{w}) \,|\, \boldsymbol{w} \in I_{\rm yes}\}$*. With respect to* σ*, the language* $L($**co**$\mathcal{P})$ *associated with* **co**$\mathcal{P}$ *is the set* $L($**co**$\mathcal{P}) = \{\sigma(\boldsymbol{w}) \,|\, \boldsymbol{w} \in I_{\rm no}\}$.

The **complement of a language** L*, denoted* $\overline{L}$*, is* $\mathcal{B}^* - L$*; that is,* $\overline{L}$ *consists of the strings that are not in* L.

A decision problem can be generalized to a **problem** $\mathcal{P}$ *characterized by a function* $f : \mathcal{B}^* \mapsto \mathcal{B}^*$ *described by a set of ordered pairs* $(\boldsymbol{x}, f(\boldsymbol{x}))$*, where each string* $\boldsymbol{x} \in \mathcal{B}^*$ *appears once as the left-hand side of a pair. Thus, a* **language** *is defined by problems* $f : \mathcal{B}^* \mapsto \mathcal{B}$ *and consists of the strings on which* f *has value* 1.

SATISFIABILITY and all other decision problems in **NP** have succinct "certificates" for "Yes" instances, that is, choices on a nondeterministic Turing machine that lead to acceptance of a "Yes" instance in a number of steps that is a polynomial in the length of the instance. A certificate for an instance of SATISFIABILITY consists of values for the variables of the instance on which each clause has at least one literal with value 1. The verification of such a certificate can be done on a Turing machine in a number of steps that is quadratic in the length of the input. (See Problem 8.3.)

Similarly, UNSATISFIABILITY and all other decision problems in **coNP** can be disqualified quickly; that is, their "No" instances can be "disqualified" quickly by exhibiting certificates for them (which are certificates for the "Yes" instance of the complementary decision problem). For example, a disqualification for UNSATISFIABILITY is a satisfiable assignment for a "No" instance, that is, a satisfiable set of clauses.

It is not known how to identify a certificate for a "Yes" instance of SATISFIABILITY or any other **NP**-complete problem in time polynomial in length of the instance. If a "Yes" instance has n variables, an exhaustive search of the 2^n values for the n variables is about the best general method known to find an answer.

8.2.1 Complements of Languages and Decision Problems

There are many ways to encode problem instances. For example, for SATISFIABILITY we might represent x_i as i and $\overline{x}_i$ as $\sim i$ and then use the standard seven-bit ASCII encodings for characters. Then we would translate the clause $\{x_4, \overline{x}_7\}$ into $\{4, \sim 7\}$ and then represent it as 123 052 044 126 055 125, where each number is a decimal representing a binary 7-tuple and 4, comma, and $\sim$ are represented by 052, 044, and 126, respectively, for example.

All the instances I of decision problems $\mathcal{P}$ considered in this chapter are characterized by regular expressions. In addition, the encoding function of Definition 8.2.1 can be chosen to map strings in I to binary strings $\sigma(I)$ describable by regular expressions. Thus, a finite-state machine can be used to determine if a binary string is in $\sigma(I)$ or not. We assume that membership of a string in $\sigma(I)$ can be determined efficiently.

As suggested by Fig. 8.1, the strings in $\overline{L}(\mathcal{P})$, the complement of $L(\mathcal{P})$, are either strings in $L($**co**$\mathcal{P})$ or strings in $\sigma(\Sigma^* - I)$. Since testing of membership in $\sigma(\Sigma^* - I)$ is easy, testing for membership in $\overline{L}(\mathcal{P})$ and $L($**co**$\mathcal{P})$ requires about the same space and time. For this reason, we often equate the two when discussing the complements of languages.

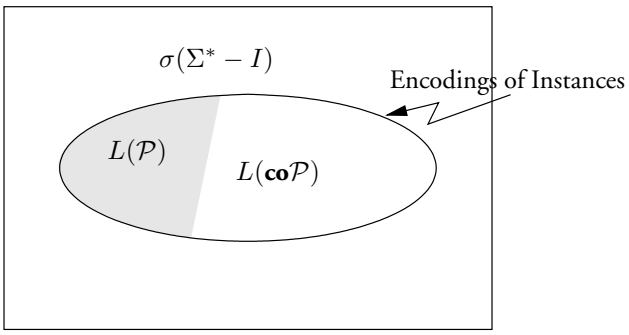


Figure 8.1 The language $L(\mathcal{P})$ of a decision problem $\mathcal{P}$ and the language of its complement $L(\mathbf{co}\mathcal{P})$. The languages $L(\mathcal{P})$ and $L(\mathbf{co}\mathcal{P})$ encode all instances of I. The complement of $L(\mathcal{P})$, $\overline{L}(\mathcal{P})$, is the union of $L(\mathbf{co}\mathcal{P})$ with $\sigma(\Sigma^* - I)$, strings that are in neither $L(\mathcal{P})$ nor $L(\mathbf{co}\mathcal{P})$.

8.3 Resource Bounds

One of the most important problems in computer science is the identification of the computationally feasible problems. Currently a problem is considered feasible if its running time on a DTM (deterministic Turing machine) is polynomial. (Stated by Edmonds [95], this is known as the **serial computation thesis**.) Note, however, that some polynomial running times, such as n^{1000}, where n is the length of a problem instance, can be enormous. In this case doubling n increases the time bound by a factor of 2^{1000}, which is approximately 10^{301}!

Since problems are classified by their use of resources, we need to be precise about **resource bounds**. These are functions $r : \mathbb{N} \mapsto \mathbb{N}$ from the natural numbers $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ to the natural numbers. The resource functions used in this chapter are:

Logarithmic function	$r(n) = O(\log n)$
Poly-logarithmic function	$r(n) = \log^{O(1)} n$
Linear function	$r(n) = O(n)$
Polynomial function	$r(n) = n^{O(1)}$
Exponential function	$r(n) = 2^{n^{O(1)}}$

A resource function that grows faster than any polynomial is called a **superpolynomial function**. For example, the function $f(n) = 2^{\log^2 n}$ grows faster than any polynomial (the ratio $\log f(n)/\log n$ is unbounded) but more slowly than any exponential (for any $k > 0$ the ratio $(\log^2 n)/n^k$ becomes vanishingly small with increasing n).

Another note of caution is appropriate here when comparing resource functions. Even though one function, $r(n)$, may grow more slowly asymptotically than another, $s(n)$, it may still be true that $r(n) > s(n)$ for very large values of n. For example, $r(n) = 10 \log^4 n > s(n) = n$ for $n \leq 1{,}889{,}750$ despite the fact that $r(n)$ is much smaller than $s(n)$ for large n.

Some resource functions are so complex that they cannot be computed in the time or space that they define. For this reason we assume throughout this chapter that all resource functions are proper. (Definitions of time and space on Turing machines are given in Section 8.4.2.)

DEFINITION 8.3.1 *A function $r : \mathbb{N} \mapsto \mathbb{N}$ is* **proper** *if it is nondecreasing $(r(n+1) \geq r(n))$ and for some tape symbol a there is a deterministic multi-tape Turing machine M that, on all*

inputs of length n in time $O(n + r(n))$ and temporary space $r(n)$, writes the string $a^{r(n)}$ (**unary notation** *for $r(n)$) on one of its tapes and halts.*

Thus, if a resource function $r(n)$ is proper, there is a DTM, M_r, that given an input of length n can write $r(n)$ markers on one of its tapes within time $O(n+r(n))$ and space $r(n)$. Another DTM, M, can use a copy of M_r to mark $r(n)$ squares on a tape that can be used to stop M after exactly $Kr(n)$ steps for some constant K. The resource function can also be used to insure that M uses no more than $Kr(n)$ cells on its work tapes.

8.4 Serial Computational Models

We consider two serial computational models in this chapter, the random-access machine (RAM) introduced in Section 3.4 and the Turing machine defined in Chapter 5.

In this section we show that, up to polynomial differences in running time, the random-access and Turing machines are equivalent. As a consequence, if the running time of a problem on one machine grows at least as fast as a polynomial in the length of a problem instance, then it grows equally fast on the other machine. This justifies using the Turing machine as basis for classifying problems by their serial complexity.

In Sections 8.13 and 8.14 we examine two parallel models of computation, the logic circuit and the parallel random-access machine (PRAM).

Before beginning our discussion of models, we note that any model can be considered either serial or parallel. For example, a finite-state machine operating on inputs and states represented by many bits is a parallel machine. On the other hand, a PRAM that uses one simple RAM processor is serial.

8.4.1 The Random-Access Machine

The random-access machine (RAM) is introduced in Section 3.4. (See Fig. 8.2.) In this section we generalize the simulation results developed in Section 3.7 by considering a RAM in which words are of potentially unbounded length. This RAM is assumed to have instructions for

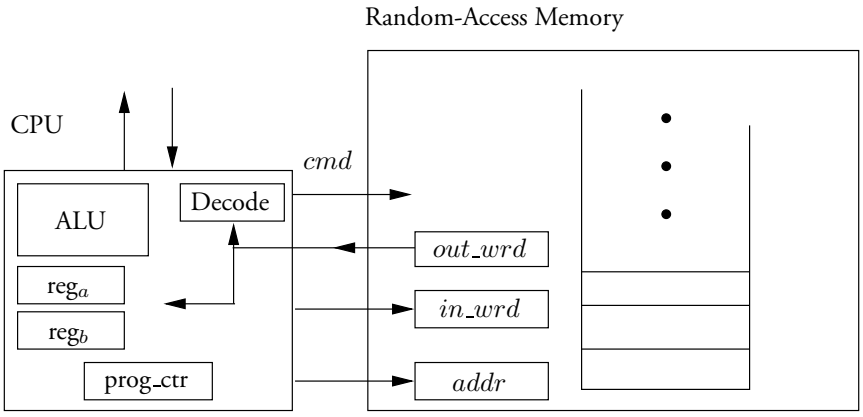


Figure 8.2 A RAM in which the number and length of words are potentially unbounded.

addition, subtraction, shifting left and right by one place, comparison of words, and Boolean operations of AND, OR, and NOT (the operations are performed on corresponding components of the source vectors), as well as conditional and unconditional jump instructions. The RAM also has load (and store) instructions that move words to (from) registers from (to) the random-access memory. Immediate and direct addressing are allowed. An immediate address contains a value, a direct address is the address of a value, and an indirect address is the address of the address of a value. (As explained in Section 3.10 and stated in Problem 3.10, indirect addressing does not add to the computing power of the RAM and is considered only in the problems.)

The **time** on a RAM is the number of steps it executes. The **space** is the maximum number of bits of storage used either in the CPU or the random-access memory during a computation.

We simplify the RAM without changing its nature by eliminating its registers, treating location 0 of the random-access memory as the accumulator, and using memory locations as registers. The RAM retains its program counter, which is incremented on each instruction execution (except for a jump instruction, when its value is set to the address supplied by the jump instruction). The word length of the RAM model is typically allowed to be unlimited, although in Section 3.4 we limited it to b bits. A RAM program is a finite sequence of RAM instructions that is stored in the random-access memory. The RAM implements the stored-program concept described in Section 3.4.

In Theorem 3.8.1 we showed that a b-bit standard Turing machine (its tape alphabet contains 2^b characters) executing T steps and using S bits of storage (S/b words) can be simulated by the RAM described above in $O(T)$ steps with $O(S)$ bits of storage. Similarly, we showed that a b-bit RAM executing T steps and using S bits of memory can be simulated by an $O(b)$-bit standard Turing machine in $O(ST \log^2 S)$ steps and $O(S \log S)$ bits of storage. As seen in Section 5.2, T-step computations on a multi-tape TM can be simulated in $O(T^2)$ steps on a standard Turing machine.

If we could insure that a RAM that executes T steps uses a highest address that is $O(T)$ and generates words of fixed length, then we could use the above-mentioned simulation to establish that a standard Turing machine can simulate an arbitrary T-step RAM computation in time $O(T^2 \log^2 T)$ and space $O(S \log S)$ measured in bits. Unfortunately, words can have length proportional to $O(T)$ (see Problem 8.4) and the highest address can be much larger than T due to the use of jumps. Nonetheless, a reasonably efficient polynomial-time simulation of a RAM computation by a DTM can be produced. Such a DTM places one (`address, contents`) pair on its tape for each RAM memory location visited by the RAM. (See Problem 8.5.)

We leave the proof of the following result to the reader. (See Problem 8.6.)

THEOREM 8.4.1 *Every computation on the RAM using time T can be simulated by a deterministic Turing machine in $O(T^3)$ steps.*

In light of the above results and since we are generally interested in problems whose time is polynomial in the length of the input, we use the DTM as our model of serial computation.

8.4.2 Turing Machine Models

The deterministic and nondeterministic Turing machines (DTM and NDTM) are discussed in Sections 3.7, 5.1, and 5.2. (See Fig. 8.3.) In this chapter we use multi-tape Turing machines to define classes of problems characterized by their use of time and space. As shown in The-

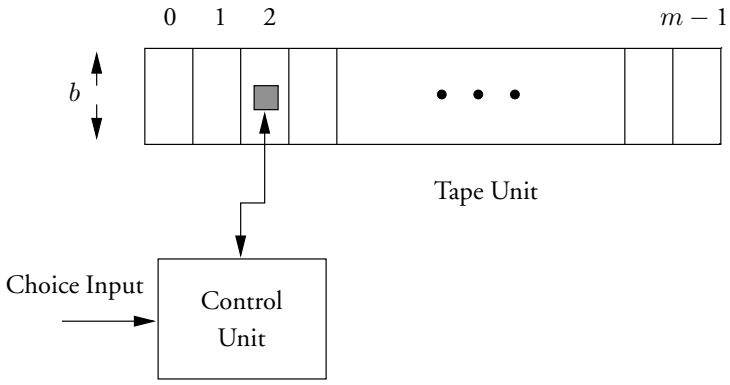


Figure 8.3 A one-tape nondeterministic Turing machine whose control unit has an external choice input that disambiguates the value of its next state.

orem 5.2.2, the general language-recognition capability of DTMs and NDTMs is the same, although, as we shall see, their ability to recognize languages within the same resource bounds is very different.

We recognize two types of Turing machine, the standard one-tape DTM and NDTM and the **multi-tape DTM and NDTM**. The multi-tape versions are defined here to have one read-only input tape, one write-only output tape, and one or more work tapes. The **space** on these machines is defined to be the number of work tape cells used during a computation. This measure allows us to classify problems by a storage that may be less than linear in the size of the input. **Time** is the number of steps they execute. It is interesting to compare these measures with those for the RAM. (See Problem 8.7.) As shown on Section 5.2, we can assume without loss of generality that each NDTM has either one or two choices for next state for any given input letters and state.

As stated in Definitions 3.7.1 and 5.1.1, a DTM M **accepts the language** L if and only if for each string in L placed left-adjusted on the otherwise blank input tape it eventually enters the accepting halt state. A language accepted by a DTM M is **recursive** if M halts on all inputs. Otherwise it is **recursively enumerable**. A DTM M **computes a partial function** f if for each input string $\boldsymbol{w}$ for which f is defined, it prints $f(\boldsymbol{w})$ left-adjusted on its otherwise blank output tape. A **complete function** is one that is defined on all points of its domain.

As stated in Definition 5.2.1, an NDTM **accepts** the language L if for each string $\boldsymbol{w}$ in L placed left-adjusted on the otherwise blank input tape there is a choice input $\boldsymbol{c}$ for M that leads to an accepting halt state. A NDTM M **computes a partial function** $f : \mathcal{B}^* \mapsto \mathcal{B}^*$ if for each input string $\boldsymbol{w}$ for which f is defined, there is a sequence of moves by M that causes it to print $f(\boldsymbol{w})$ on its output tape and enter a halt state and there is no choice input for which M prints an incorrect result.

The **oracle Turing machine** (OTM), the multi-tape DTM or NDTM with a special **oracle tape**, defined in Section 5.2.3, is used to classify problems. (See Problem 8.15.) **Time** on an OTM is the number of steps it takes, where one consultation of the oracle is one step, whereas **space** is the number of cells used on its work tapes not including the oracle tape.

A **precise Turing machine** M is a multi-tape DTM or NDTM for which there is a function $r(n)$ such that for every $n \geq 1$, every input $\boldsymbol{w}$ of length n, and every (possibly nondeterministic) computation by M, M halts after precisely $r(n)$ steps.

We now show that if a total function can be computed by a DTM, NDTM, or OTM within a proper time or space bound, it can be computed within approximately the same resource bound by a precise TM of the same type. The following theorem justifies the use of proper resource functions.

THEOREM 8.4.2 *Let $r(n)$ be a proper function with $r(n) \geq n$. Let M be a multi-tape DTM, NDTM, or OTM with k work tapes that computes a total function f in time or space $r(n)$. Then there is a constant $K > 0$ and a precise Turing machine of the same type that computes f in time and space $Kr(n)$.*

Proof Since $r(n)$ is a proper function, there is a DTM M_r that computes its value from an input of length n in time $K_1 r(n)$ for some constant $K_1 > 0$ and in space $r(n)$. We design a precise TM M_p computing the same function.

The TM M_p has an "enumeration tape" that is distinct from its work tapes. M_p initially invokes M_r to write $r(n)$ instances of the letter a on the enumeration tape in $K_1 r(n)$ steps, after which it returns the head on this tape to its initial position.

Suppose that M computes f within a time bound of $r(n)$. M_p then alternates between simulating one step of M on its work tapes and advancing its head on the enumeration tape. When M halts, M_p continues to read and advance the head on its enumeration tape on alternate steps until it encounters a blank. Clearly, M_p halts in precisely $(K_1 + 2)r(n)$ steps.

Suppose now that M computes f in space $r(n)$. M_p invokes M_r to write $r(n)$ special blank symbols on each of its work tapes. It then simulates M, treating the special blank symbols as standard blanks. Thus, M_p uses precisely $kr(n)$ cells on its k work tapes. ■

Configuration graphs, defined in Section 5.3, are graphs that capture the state of Turing machines with potentially unlimited storage capacity. Since all resource bounds are proper, as we know from Theorem 8.4.2, all DTMs and NDTMs used for decision problems halt on all inputs. Furthermore, NDTMs never give an incorrect answer. Thus, configuration graphs can be assumed to be acyclic.

8.5 Classification of Decision Problems

In this section we classify decision problems by the resources they consume on deterministic and nondeterministic Turing machines. We begin with the definition of complexity classes.

DEFINITION 8.5.1 *Let $r(n) : \mathbb{N} \mapsto \mathbb{N}$ be a proper resource function. Then* **TIME($r(n)$)** *and* **SPACE($r(n)$)** *are the* **time** *and* **space Turing complexity classes** *containing languages that can be recognized by DTMs that halt on all inputs in time and space $r(n)$, respectively, where n is the length of an input.* **NTIME($r(n)$)** *and* **NSPACE($r(n)$)** *are the* **nondeterministic time** *and* **space Turing complexity classes**, *respectively, defined for NDTMs instead of DTMs. The union of complexity classes is also a complexity class.*

Let k be a positive integer. Then **TIME**(k^n) and **NSPACE**(n^k) are examples of complexity classes. They are the decision problems solvable in deterministic time k^n and nondeterministic

space n^k, respectively, for n the length of the input. Since time and space on a Turing machine are measured by the number of steps and number of tape cells, it is straightforward to show that time and space for a given Turing machine, deterministic or not, can each be reduced by a constant factor by modifying the Turing machine description so that it acts on larger units of information. (See Problem 8.8.) Thus, for a constant $K > 0$ the following classes are the same: a) **TIME**(k^n) and **TIME**(Kk^n), b) **NTIME**(k^n) and **NTIME**(Kk^n), c) **SPACE**(n^k) and **SPACE**(Kn^k), and d) **NSPACE**(n^k) and **NSPACE**(Kn^k).

To emphasize that the union of complexity classes is another complexity class, we define as unions two of the most important Turing complexity classes, **P**, the class of deterministic polynomial-time decision problems, and **NP**, the class of nondeterministic polynomial-time decision problems.

DEFINITION 8.5.2 *The classes* **P** *and* **NP** *are sets of decision problems solvable in polynomial time on DTMs and NDTMs, respectively; that is, they are defined as follows:*

$$\mathbf{P} = \bigcup_{k \geq 0} \mathbf{TIME}(n^k)$$

$$\mathbf{NP} = \bigcup_{k \geq 0} \mathbf{NTIME}(n^k)$$

Thus, for each decision problem $\mathcal{P}$ in **P** there is a DTM M and a polynomial $p(n)$ such that M halts on each input string of length n in $p(n)$ steps, accepting this string if it is an instance $\boldsymbol{w}$ of $\mathcal{P}$ and rejecting it otherwise.

Also, for each decision problem $\mathcal{P}$ in **NP** there is an NDTM M and a polynomial $p(n)$ such that for each instance $\boldsymbol{w}$ of $\mathcal{P}$, $|\boldsymbol{w}| = n$, there is a choice input of length $p(n)$ such that M accepts $\boldsymbol{w}$ in $p(n)$ steps.

Problems in **P** are considered **feasible problems** because they can be decided in time polynomial in the length of their input. Even though some polynomial functions, such as n^{1000}, grow very rapidly in their one parameter, at the present time problems in **P** are considered feasible. Problems that require exponential time are not considered feasible.

The class **NP** includes the decision problems associated with many hundreds of important searching and optimization problems, such as TRAVELING SALESPERSON described below. (See Fig. 8.4.) If **P** is equal to **NP**, then these important problems have feasible solutions. If not, then there are problems in **NP** that require superpolynomial time and are therefore largely infeasible. Thus, it is very important to have the answer to the question $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

TRAVELING SALESPERSON
Instance: An integer k and a set of n^2 symmetric integer distances $\{d_{i,j} \mid 1 \leq i, j \leq n\}$ between n cities where $d_{i,j} = d_{j,i}$.
Answer: "Yes" if there is a **tour** (an ordering) $\{i_1, i_2, \ldots, i_n\}$ of the cities such that the length $l = d_{i_1,i_2} + d_{i_2,i_3} + \cdots + d_{i_n,i_1}$ of the tour satisfies $l \leq k$.

The TRAVELING SALESPERSON problem is in **NP** because a tour satisfying $l \leq k$ can be chosen nondeterministically in n steps and the condition $l \leq k$ then verified in a polynomial number of steps by finding the distances between successive cities on the chosen tour in the description of the problem and adding them together. (See Problem 3.24.) Many other important problems are in **NP**, as we see in Section 8.10. While it is unknown whether a deterministic polynomial-time algorithm exists for this problem, it can clearly be solved deter-

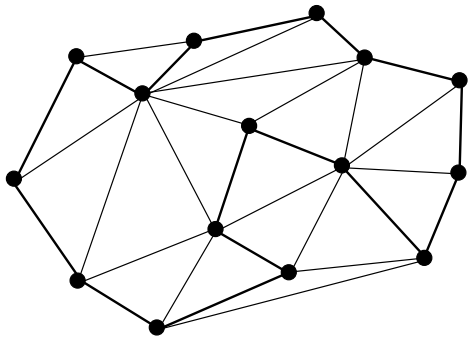


Figure 8.4 A graph on which the TRAVELING SALESPERSON problem is defined. The heavy edges identify a shortest tour.

ministically in exponential time by enumerating all tours and choosing the one with smallest length. (See Problem 8.9.)

The TRAVELING SALESPERSON decision problem is a reduction of the **traveling salesperson optimization problem**, whose goal is to find the shortest tour that visits each city once. The output of the optimization problem is an ordering of the cities that has the shortest tour. By contrast, the TRAVELING SALESPERSON decision problem reports that there is or is not a tour of length k or less. Given an algorithm for the optimization problem, the decision problem can be solved by calculating the length of an optimal tour and comparing it to the parameter k of the decision problem. Since the latter steps can be done in polynomial time, if the optimization algorithm can be done in polynomial time, so can the decision problem. On the other hand, given an algorithm for the decision problem, the optimization problem can be solved through **bisection** as follows: a) Since the length of the shortest tour is in the interval $[n \min_{i,j} d_{i,j}, n \max_{i,j} d_{i,j}]$, invoke the decision algorithm with k equal to the midpoint of this interval. b) If the instance is a "yes" instance, let k be the midpoint of the lower half of the current interval; if not, let it be the midpoint of the upper half. c) Repeat the previous step until the interval is reduced to one integer. The interval is bisected $O(\log n(\max_{i,j} d_{i,j} - \min_{i,j} d_{i,j}))$ times. Thus, if the decision problem can be solved in polynomial time, so can the optimization problem.

Whether $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ is one of the outstanding problems of computer science. The current consensus of complexity theorists is that nondeterminism is such a powerful specification device that they are not equal. We return to this topic in Section 8.8.

8.5.1 Space and Time Hierarchies

In this section we state without proof the following time and space hierarchy theorems. (See [126,127].) These theorems state that if one space (or time) resource bound grows sufficiently rapidly relative to another, the set of languages recognized within the first bound is strictly larger than the set recognized within the second bound.

THEOREM 8.5.1 (Time Hierarchy Theorem) *If $r(n) \geq n$ is a proper complexity function, then* $\mathbf{TIME}(r(n))$ *is strictly contained in* $\mathbf{TIME}(r(n) \log r(n))$.

Let $r(n)$ and $s(n)$ be proper functions. If for all $K > 0$ there exists an N_0 such that $s(n) \geq Kr(n)$ for $n \geq N_0$, we say that **$r(n)$ is little oh of $s(n)$** and write $r(n) = o(s(n))$.

THEOREM 8.5.2 (Space Hierarchy Theorem) *If $r(n)$ and $s(n)$ are proper complexity functions and $r(n) = o(s(n))$, then* **SPACE**$(r(n))$ *is strictly contained in* **SPACE**$(s(n))$.

Theorem 8.5.3 states that there is a recursive but not proper resource function $r(n)$ such that **TIME**$(r(n))$ and **TIME**$(2^{r(n)})$ are the same. That is, for some function $r(n)$ there is a gap of at least $2^{r(n)} - r(n)$ in time over which no new decision problems are encountered. This is a weakened version of a stronger result in [333] and independently reported by [51].

THEOREM 8.5.3 (Gap Theorem) *There is a recursive function $r(n) : \mathcal{B}^* \mapsto \mathcal{B}^*$ such that* **TIME**$(r(n)) =$ **TIME**$(2^{r(n)})$.

8.5.2 Time-Bounded Complexity Classes

As mentioned earlier, decision problems in **P** are considered to be feasible while the class **NP** includes many interesting problems, such as the TRAVELING SALESPERSON problem, whose feasibility is unknown. Two other important complexity classes are the deterministic and nondeterministic exponential-time problems. By the remarks on page 336, TRAVELING SALESPERSON clearly falls into the latter class.

DEFINITION 8.5.3 *The classes* **EXPTIME** *and* **NEXPTIME** *consist of those decision problems solvable in deterministic and nondeterministic exponential time, respectively, on a Turing machine. That is,*

$$\textbf{EXPTIME} = \bigcup_{k \geq 0} \textbf{TIME}(2^{n^k})$$

$$\textbf{NEXPTIME} = \bigcup_{k \geq 0} \textbf{NTIME}(2^{n^k})$$

We make the following observations concerning containment of these complexity classes.

THEOREM 8.5.4 *The following complexity class containments hold:*

$$\textbf{P} \subseteq \textbf{NP} \subseteq \textbf{EXPTIME} \subseteq \textbf{NEXPTIME}$$

However, **P** $\subset$ **EXPTIME**, *that is,* **P** *is strictly contained in* **EXPTIME**.

Proof Since languages in **P** are recognized in polynomial time by a DTM and such machines are included among the NDTMs, it follows immediately that **P** $\subseteq$ **NP**. By similar reasoning, **EXPTIME** $\subseteq$ **NEXPTIME**.

We now show that **P** is strictly contained in **EXPTIME**. **P** $\subseteq$ **TIME**(2^n) follows because **TIME**$(n^k) \subseteq$ **TIME**(2^n) for each $k \geq 0$. By the Time Hierarchy Theorem (Theorem 8.5.1), we have that **TIME**$(2^n) \subset$ **TIME**$(n2^n)$. But **TIME**$(n2^n) \subseteq$ **EXPTIME**. Thus, **P** is strictly contained in **EXPTIME**.

Containment of **NP** in **EXPTIME** is deduced from the proof of Theorem 5.2.2 by analyzing the time taken by the deterministic simulation of an NDTM. If the NDTM executes T steps, the DTM executes $O(k^T)$ steps for some constant k. ■

The relationships **P** $\subseteq$ **NP** and **EXPTIME** $\subseteq$ **NEXPTIME** are examples of a more general result, namely, **TIME**$(r(n)) \subseteq$ **NTIME**$(r(n))$, where these two classes of decision problems can respectively be solved deterministically and nondeterministically in time $r(n)$, where n is the length of the input. This result holds because every $\mathcal{P} \in$ **TIME**$(r(n))$ of length n is accepted in $r(n)$ steps by some DTM $M_{\mathcal{P}}$ and a DTM is also a NDTM. Thus, it is also true that $\mathcal{P} \in$ **NTIME**$(r(n))$.

8.5.3 Space-Bounded Complexity Classes

Many other important space complexity classes are defined by the amount of space used to recognize languages and compute functions. We highlight five of them here: the deterministic and nondeterministic logarithmic space classes **L** and **NL**, the square-logarithmic space class $\mathbf{L^2}$, and the deterministic and nondeterministic polynomial-space classes **PSPACE** and **NPSPACE**.

DEFINITION 8.5.4 **L** *and* **NL** *are the decision problems solvable in logarithmic space on a DTM and NDTM, respectively.* $\mathbf{L^2}$ *are the decision problems solvable in space* $O(\log^2 n)$ *on a DTM.* **PSPACE** *and* **NPSPACE** *are the decision problems solvable in polynomial space on a DTM and NDTM, respectively.*

Because **L** and **PSPACE** are deterministic complexity classes, they are contained in **NL** and **NPSPACE**, respectively: that is, **L** $\subseteq$ **NL** and **PSPACE** $\subseteq$ **NPSPACE**.

We now strengthen the latter result and show that **PSPACE** $=$ **NPSPACE**, which means that nondeterminism does not increase the recognition power of Turing machines if they already have access to a polynomial amount of storage space.

The REACHABILITY problem on directed acyclic graphs defined below is used to show this result. REACHABILITY is applied to configuration graphs of deterministic and nondeterministic Turing machines. Configuration graphs are introduced in Section 5.3.

REACHABILITY
Instance: A directed graph $G = (V, E)$ and a pair of vertices $u, v \in V$.
Answer: "Yes" if there is a directed path in G from u to v.

REACHABILITY can be decided by computing the transitive closure of the adjacency matrix of G in parallel. (See Section 6.4.) However, a simple serial RAM program based on depth-first search can also solve the reachability problem. **Depth-first search** (DFS) on an undirected graph G visits each edge in the forward direction once. Edges at each vertex are ordered. Each time DFS arrives at a vertex it traverses the next unvisited edge. If DFS arrives at a vertex from which there are no unvisited edges, it retreats to the previously visited vertex. Thus, after DFS visits all the descendants of a vertex, it backs up, eventually returning to the vertex from which the search began.

Since every T-step RAM computation can be simulated by an $O(T^3)$-step DTM computation (see Problem 8.6), a cubic-time DTM program based on DFS exists for REACHABILITY. Unfortunately, the space to execute DFS on the RAM and Turing machine both can be linear in the size of the graph. We give an improved result that allows us to strengthen **PSPACE** $\subseteq$ **NPSPACE** to **PSPACE** = **NPSPACE**.

Below we show that REACHABILITY can be realized in quadratic logarithmic space. This fact is then used to show that **NSPACE**$(r(n)) \subseteq$ **SPACE**$(r^2(n))$ for $r(n) = \Omega(\log n)$.

THEOREM 8.5.5 (Savitch) REACHABILITY *is in* **SPACE**$(\log^2 n)$.

Proof As mentioned three paragraphs earlier, the REACHABILITY problem on a graph $G = (V, E)$ can be solved with depth-first search. This requires storing data on each vertex visited during a search. This data can be as large as $O(n)$, $n = |V|$. We exhibit an algorithm that uses much less space.

Given an instance of REACHABILITY defined by $G = (V, E)$ and $u, v \in V$, for each pair of vertices (a, b) and integer $k \leq \lceil \log_2 n \rceil$ we define predicates PATH$\left(a, b, 2^k\right)$ whose value is true if there exists a path from a to b in G whose length is at most 2^k and false otherwise. Since no path has length more than n, the solution to the REACHABILITY problem is the value of PATH$\left(u, v, 2^{\lceil \log_2 n \rceil}\right)$. The predicates PATH$\left(a, b, 2^0\right)$ are true if either $a = b$ or there is a path of length 1 (an edge) between the vertices a and b. Thus, PATH$\left(a, b, 2^0\right)$ can be evaluated directly by consulting the problem instance on the input tape.

The algorithm that computes PATH$\left(u, v, 2^{\lceil \log_2 n \rceil}\right)$ with space $O(\log^2 n)$ uses the fact that any path of length at most 2^k can be decomposed into two paths of length at most 2^{k-1}. Thus, if PATH$\left(a, b, 2^k\right)$ is true, then there must be some vertex z such that PATH$\left(a, z, 2^{k-1}\right)$ and PATH$\left(z, b, 2^{k-1}\right)$ are both true. The truth of PATH$\left(a, b, 2^k\right)$ can be established by searching for a z such that PATH$\left(a, z, 2^{k-1}\right)$ is true. Upon finding one, we determine the truth of PATH$\left(z, b, 2^{k-1}\right)$. Failing to find such a z, PATH$\left(a, b, 2^k\right)$ is declared to be false. Each evaluation of a predicate is done in the same fashion, that is, recursively. Because we need evaluate only one of PATH$\left(a, z, 2^{k-1}\right)$ and PATH$\left(z, b, 2^{k-1}\right)$ at a time, space can be reused.

We now describe a deterministic Turing machine with an input tape and two work tapes computing PATH$\left(u, v, 2^{\lceil \log_2 n \rceil}\right)$. The input tape contains an instance of REACHABILITY, which means it has not only the vertices u and v but also a description of the graph G. The first work tape will contain triples of the form (a, b, k), which are called **activation records**. This tape is initialized with the activation record $(u, v, \lceil \log_2 n \rceil)$. (See Fig. 8.5.)

The DTM evaluates the last activation record, (a, b, k), on the first work tape as described above. There are three kinds of activation records, **complete records** of the form (a, b, k), **initial segments** of the form $(a, z, k-1)$, and **final segments** of the form $(z, b, k-1)$. The first work tape is initialized with the complete record $(u, v, \lceil \log_2 n \rceil)$.

An initial segment is created from the current complete record (a, b, k) by selecting a vertex z to form the record $(a, z, k-1)$, which becomes the current complete record. If it evaluates to true, it can be determined to be an initial or final segment by examining the previous record (a, b, k). If it evaluates to false, $(a, z, k-1)$ is erased and another value of z, if any, is selected and another initial segment placed on the work tape for evaluation. If no other z exists, $(a, z, k-1)$ is erased and the expression PATH$\left(a, b, 2^k\right)$ is declared false. If $(a, z, k-1)$ evaluates to true, the final record $(z, b, k-1)$ is created, placed on the work tape, and evaluated in the same fashion. As mentioned in the second paragraph of this

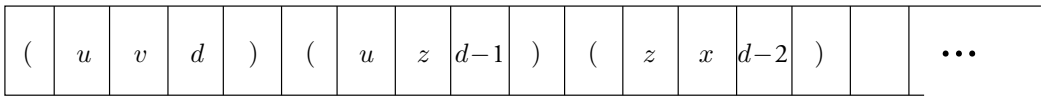


Figure 8.5 A snapshot of the stack used by the REACHABILITY algorithm in which the components of an activation record (a, b, k) are distributed over several cells.

proof, $(a, b, 0)$ is evaluated by consulting the description of the graph on the input tape. The second work tape is used for bookkeeping, that is, to enumerate values of z and determine whether a segment is initial or final.

The second work tape uses space $O(\log n)$. The first work tape contains at most $\lceil \log_2 n \rceil$ activation records. Each activation record (a, b, k) can be stored in $O(\log n)$ space because each vertex can be specified in $O(\log n)$ space and the depth parameter k can be specified in $O(\log k) = O(\log \log n)$ space. It follows that the first work tape uses at most $O(\log^2 n)$ space. ■

The following general result, which is a corollary of Savitch's theorem, demonstrates that nondeterminism does not enlarge the space complexity classes if they are defined by space bounds that are at least logarithmic. In particular, it implies that **PSPACE** = **NPSPACE**.

COROLLARY 8.5.1 *Let $r(n)$ be a proper Turing computable function $r : \mathbb{N} \mapsto \mathbb{N}$ satisfying $r(n) = \Omega(\log n)$. Then* **NSPACE**$(r(n)) \subseteq$ **SPACE**$(r^2(n))$.

Proof Let $M_{\rm ND}$ be an NDTM with input and output tapes and s work tapes. Let it recognize a language $L \in$ **NSPACE**$(r(n))$. For each input string $\boldsymbol{w}$, we generate a configuration graph $G(M_{\rm ND}, \boldsymbol{w})$ of $M_{\rm ND}$. (See Fig. 8.6.) We use this graph to determine whether or not $\boldsymbol{w} \in L$. $M_{\rm ND}$ has at most $|Q|$ states, each tape cell can have at most c values (there are $c^{(s+2)r(n)}$ configurations for the $s + 2$ tapes), the s work tape heads and the output tape head can assume values in the range $1 \leq h_j \leq r(n)$, and the input head h_{s+1} can assume one of n positions (there are $nr(n)^{s+1}$ configurations for the tape heads). It follows that $M_{\rm ND}$ has at most $|Q|c^{(s+2)r(n)}(n\, r(n)^{s+1}) \leq k^{\log n + r(n)}$ configurations. $G(M_{\rm ND}, \boldsymbol{w})$ has the same number of vertices as there are configurations and a number of edges at most the square of its number of vertices.

Let $L \in$ **NSPACE**$(r(n))$ be recognized by an NDTM $M_{\rm ND}$. We describe a deterministic $r^2(n)$-space Turing machine $M_{\rm D}$ recognizing L. For input string $\boldsymbol{w} \in L$ of length n, this machine solves the REACHABILITY problem on the configuration graph $G(M_{\rm ND}, \boldsymbol{w})$ of $M_{\rm ND}$ described above. However, instead of placing on the input tape the entire configuration graph, we place the input string $\boldsymbol{w}$ and the description of $M_{\rm ND}$. We keep configurations on the work tape as part of activation records (they describe vertices of $G(M_{\rm ND}, \boldsymbol{w})$).

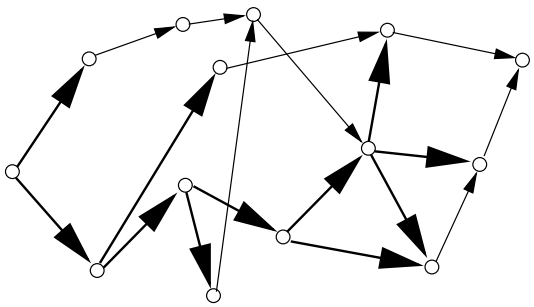


Figure 8.6 The acyclic configuration graph $G(M_{\rm ND}, \boldsymbol{w})$ of a nondeterministic Turing machine $M_{\rm ND}$ on input $\boldsymbol{w}$ has one vertex for each configuration of $M_{\rm ND}$. Here heavy edges identify the nondeterministic choices associated with a configuration.

Each of the vertices (configurations) adjacent to a particular vertex can be deduced from the description of M_{ND}.

Since the number of configurations of M_{ND} is $N = O\left(k^{\log n + r(n)}\right)$, each configuration or activation record can be stored as a string of length $O(r(n))$.

From Theorem 8.5.5, the reachability in $G(M_{\text{ND}}, \boldsymbol{w})$ of the final configuration from the initial one can be determined in space $O(\log^2 N)$. But $N = O\left(k^{\log n + r(n)}\right)$, from which it follows that **NSPACE**$(r(n)) \subseteq$ **SPACE**$(r^2(n))$. ■

The classes **NL**, $\mathbf{L^2}$ and **PSPACE** are defined as unions of deterministic and nondeterministic space-bounded complexity classes. Thus, it follows from this corollary that **NL** $\subseteq \mathbf{L^2} \subseteq$ **PSPACE**. However, because of the space hierarchy theorem (Theorem 8.5.2), it follows that $\mathbf{L^2}$ is contained in but not equal to **PSPACE**, denoted $\mathbf{L^2} \subset$ **PSPACE**.

8.5.4 Relations Between Time- and Space-Bounded Classes

In this section we establish a number of complexity class containment results involving both space- and time-bounded classes. We begin by proving that the nondeterministic $O(r(n))$-space class is contained within the deterministic $O\left(k^{r(n)}\right)$-time class. This implies that **NL** $\subseteq$ **P** and **NPSPACE** $\subseteq$ **EXPTIME**.

THEOREM 8.5.6 *The classes* **NSPACE**$(r(n))$ *and* **TIME**$(r(n))$ *of decision problems solvable in nondeterministic space and deterministic time* $r(n)$*, respectively, satisfy the following relation for some constant* $k > 0$*:*

$$\textbf{NSPACE}(r(n)) \subseteq \textbf{TIME}(k^{\log n + r(n)})$$

Proof Let M_{ND} accept a language $L \in$ **NSPACE**$(r(n))$ and let $G(M_{\text{ND}}, \boldsymbol{w})$ be the configuration graph for M_{ND} on input $\boldsymbol{w}$. To determine if $\boldsymbol{w}$ is accepted by M_{ND} and therefore in L, it suffices to determine if there is a path in $G(M_{\text{ND}}, \boldsymbol{w})$ from the initial configuration of M_{ND} to the final configuration. This is the REACHABILITY problem, which, as stated in the proof of Theorem 8.5.5, can be solved by a DTM in time polynomial in the length of the input. When this algorithm needs to determine the descendants of a vertex in $G(M_{\text{ND}}, \boldsymbol{w})$, it consults the definition of M_{ND} to determine the configurations reachable from the current configuration. It follows that membership of $\boldsymbol{w}$ in L can be determined in time $O\left(k^{\log n + r(n)}\right)$ for some $k > 1$ or that L is in **TIME**$\left(k^{\log n + r(n)}\right)$. ■

COROLLARY 8.5.2 **NL** $\subseteq$ **P** *and* **NPSPACE** $\subseteq$ **EXPTIME**

Later we explore the polynomial-time problems by exhibiting other important complexity classes that reside inside **P**. (See Section 8.15.) We now show containment of the nondeterministic time complexity classes in deterministic space classes.

THEOREM 8.5.7 *The following containment holds:*

$$\textbf{NTIME}(r(n)) \subseteq \textbf{SPACE}(r(n))$$

Proof We use the construction of Theorem 5.2.2. Let L be a language in **NTIME**$(r(n))$. We note that the choice string on the enumeration tape converts the nondeterministic recognition of L into deterministic recognition. Since L is recognized in time $r(n)$ for some accepting computation, the deterministic enumeration runs in time $r(n)$ for each choice

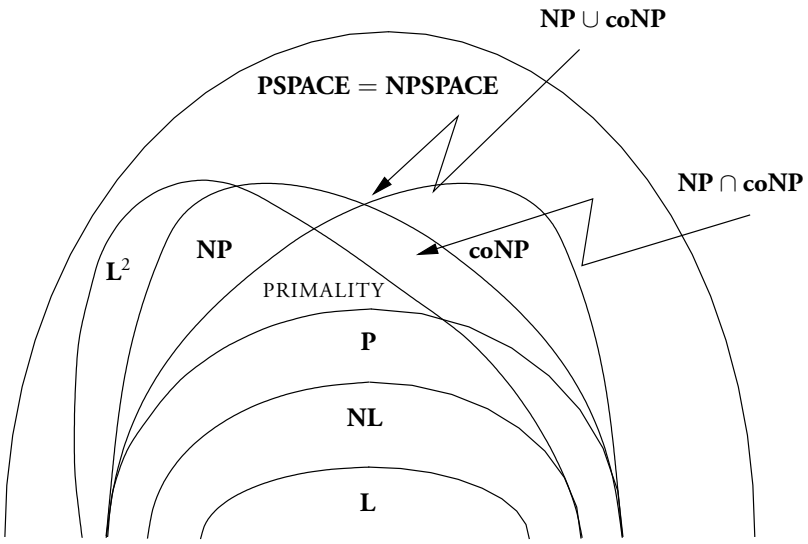


Figure 8.7 The relationships among complexity classes derived in this section. Containment is indicated by arrows.

string. Thus, $O(r(n))$ cells are used on the work and enumeration tapes in this deterministic simulation and L is in **PSPACE**. ■

An immediate corollary to this theorem is that **NP** $\subseteq$ **PSPACE**. This implies that **P** $\subseteq$ **EXPTIME**. However, as mentioned above, **P** is strictly contained within **EXPTIME**.

Combining these results, we have the following complexity class inclusions:

$$\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXPTIME} \subseteq \mathbf{NEXPTIME}$$

where **PSPACE** $=$ **NPSPACE**. We also have $\mathbf{L^2} \subset$ **PSPACE**, and **P** $\subset$ **EXPTIME**, which follow from the space and time hierarchy theorems. These inclusions and those derived below are shown in Fig. 8.7.

In Section 8.6 we develop refinements of this partial ordering of complexity classes by using the complements of complexity classes.

We now digress slightly to discuss space-bounded functions.

8.5.5 Space-Bounded Functions

We digress briefly to specialize Theorem 8.5.6 to log-space computations, not just log-space language recognition. As the following demonstrates, log-space computable functions are computable in polynomial time.

THEOREM 8.5.8 *Let M be a DTM that halts on all inputs using space $O(\log n)$ to process inputs of length n. Then M executes a polynomial number of steps.*

Proof In the proof of Corollary 8.5.1 the number of configurations of a Turing machine M with input and output tapes and s work tapes is counted. We repeat this analysis. Let $r(n)$

be the maximum number of tape cells used and let c be the maximal size of a tape alphabet. Then, M can be in one of at most $\chi \leq c^{(s+2)r(n)}(n\,r(n)^{s+1}) = O(k^{r(n)})$ configurations for some $k \geq 1$. Since M always halts, by the pigeonhole principle, it passes through at most χ configurations in at most χ steps. Because $r(n) = O(\log n)$, $\chi = O(n^d)$ for some integer d. Thus, M executes a polynomial number of steps. ■

8.6 Complements of Complexity Classes

As seen in Section 4.6, the regular languages are closed under complementation. However, we have also seen in Section 4.13 that the context-free languages are not closed under complementation. Thus, complementation is a way to develop an understanding of the properties of a class of languages. In this section we show that the nondeterministic space classes are closed under complements. The complements of languages and decision problems were defined at the beginning of this chapter.

Consider REACHABILITY. Its complement $\overline{\text{REACHABILITY}}$ is the set of directed graphs $G = (V, E)$ and pairs of vertices $u, v \in V$ such that there are no directed paths between u and v. It follows that the union of these two problems is not the entire set of strings over $\mathcal{B}^*$ but the set of all instances consisting of a directed graph $G = (V, E)$ and a pair of vertices $u, v \in V$. This set is easily detected by a DTM. It must only verify that the string describing a putative graph is in the correct format and that the representations for u and v are among the vertices of this graph.

Given a complexity class, it is natural to define the complement of the class.

DEFINITION 8.6.1 *The* **complement of a complexity class of decision problems $\mathcal{C}$,** *denoted* **co$\mathcal{C}$,** *is the set of decision problems that are complements of decision problems in* $\mathcal{C}$.

Our first result follows from the definition of the recognition of languages by DTMs.

THEOREM 8.6.1 *If $\mathcal{C}$ is a deterministic time or space complexity class, then* **co$\mathcal{C}$** $= \mathcal{C}$.

Proof Every $L \in \mathcal{C}$ is recognized by a DTM M that halts within the resource bound of $\mathcal{C}$ for every string, whether in L or $\overline{L}$, the complement of L. Create $\overline{M}$ from M by complementing the accept/reject status of states of M's control unit. Thus, $\overline{L}$, which by definition is in **co$\mathcal{C}$**, is also in $\mathcal{C}$. That is, **co$\mathcal{C}$** $\subseteq \mathcal{C}$. Similarly, $\mathcal{C} \subseteq$ **co$\mathcal{C}$**. Thus, **co$\mathcal{C}$** $= \mathcal{C}$. ■

In particular, this result says that the class **P** is closed under complements. That is, if the "yes" instances of a decision problem can be answered in deterministic polynomial time, then so can the "No" instances.

We use the above theorem and Theorem 5.7.6 to give another proof that there are problems that are not in **P**.

COROLLARY 8.6.1 *There are languages not in* **P**, *that is, languages that cannot be recognized deterministically in polynomial time.*

Proof Since every language in **P** is recursive and $\mathcal{L}_1$ defined in Section 5.7.2 is not recursive, it follows that $\mathcal{L}_1$ is not in **P**. ■

We now show that all nondeterministic space classes with a sufficiently large space bound are also closed under complements. This leaves open the question whether the nondetermin-

istic time classes are closed under complement. As we shall see, this is intimately related to the question $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

As stated in Definition 5.2.1, for no choices of moves is an NDTM allowed to produce an answer for which it is not designed. In particular, when computing a function it is not allowed to give a false answer for any set of nondeterministic choices.

THEOREM 8.6.2 (Immerman-Szelepscényi) *Given a graph $G = (V, E)$ and a vertex v, the number of vertices reachable from v can be computed by an NDTM in space $O(\log n)$, $n = |V|$.*

Proof Let $V = \{1, 2, \ldots, n\}$. Any node reachable from a vertex v must be reachable via a path of length (number of edges) of at most $n - 1$, $n = |V|$. Let $R(k, u)$ be the number of vertices of G reachable from u by paths of length k or less. The goal is to compute $R(n-1, u)$. A deterministic program for this purpose could be based on the predicate PATH(u, v, k) that has value 1 if there is a path of length k or less from vertex u to vertex v and 0 otherwise and the predicate ADJACENT-OR-IDENTICAL(x, v) that has value 1 if $x = v$ or there is an edge in G from x to v and 0 otherwise. (See Fig. 8.8.) If we let the vertices be associated with the integers in the interval $[1, \ldots, n]$, then $R(n-1, u)$ can be evaluated as follows:

$$\begin{aligned} R(n-1, u) &= \sum_{1 \le v \le n} \text{PATH}(u, v, n-1) \\ &= \bigvee_{1 \le v \le n} \sum_{1 \le x \le n} \text{PATH}(u, x, n-2)\text{ADJACENT-OR-EQUAL}(x, v) \end{aligned}$$

When this description of $R(n-1, u)$ is converted to a program, the amount of storage needed grows more rapidly than $O(\log n)$. However, if the inner use of PATH$(u, x, n-2)$ is replaced by the nonrecursive and nondeterministic test EXISTS-PATH-FROM-u-TO-v-$\le$ LENGTH of Fig. 8.9 for a path from u to x of length $n-2$, then the space can be kept to $O(\log n)$. This test nondeterministically guesses paths but verifies deterministically that all paths have been explored.

The procedure COUNTING-REACHABILITY of Fig. 8.9 is a nondeterministic program computing $R(n-1, u)$. It uses the procedure #-VERTICES-AT-$\le$-DISTANCE-FROM-u to compute the number of vertices at distance $dist$ or less from u in order of increasing values of $dist$. (It computes $dist$ correctly or fails.) This procedure has $prev\_num\_dist$ as a parameter, which is the number of vertices at distance $dist - 1$ or less. It passes this

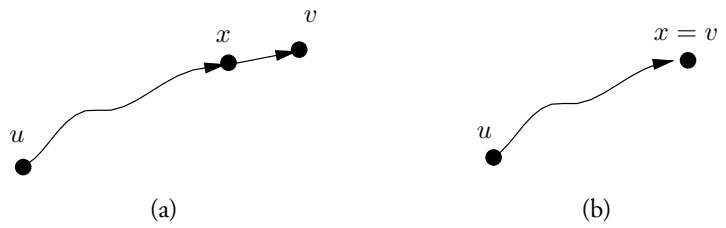


Figure 8.8 Paths explored by the REACHABILITY algorithm. Case (a) applies when x and v are different and (b) when they are the same.

```
COUNTING-REACHABILITY(u)
{R(k,u) = number of vertices at distance ≤ k from u in G = (V,E)}
  prev_num_dist := 1; {num_dist = R(0,u)}
  for dist := 1 to n − 1
    num_dist := #-VERTICES-AT-≤-DIST-FROM-u(dist, u, prev_num_dist)
    prev_num_dist := num_dist
      {num_dist = R(dist,u)}
  return(num_dist)

#-VERTICES-AT-≤-DISTANCE-FROM-u(dist, u, prev_num_dist)
{Returns R(dist,u) given prev_num_dist = R(dist − 1,u) or fails}
  num_nodes := 0
  for last_node := 1 to n
    if IS-NODE-AT-≤-DIST-FROM-u(dist, u, last_node, prev_num_dist) then
      num_nodes := num_nodes + 1
  return (num_nodes)

IS-NODE-AT-≤-DIST-FROM-u(dist, u, last_node, prev_num_dist)
  {num_node = number of vertices at distance ≤ dist from u found so far}
  num_node := 0;
  reply := false
  for next_to_last_node := 1 to n
    if EXISTS-PATH-FROM-u-TO-v-≤-LENGTH(u, next_to_last_node, dist − 1) then
      num_node := num_node + 1 {count number of next-to-last nodes or fail}
      if ADJACENT-OR-IDENTICAL(next_to_last_node, last_node) then
        reply := true
  if num_node < prev_num_dist then
    fail
  else return(reply)

EXISTS-PATH-FROM-u-TO-v-≤-LENGTH(u, v, dist)
  {nondeterministically choose at most dist vertices, fail if they don't form a path}
  node_1 := u
  for count := 1 to dist
    node_2 := NONDETERMINISTIC-GUESS([1, .., n])
    if not ADJACENT-OR-IDENTICAL(node_1, node_2) then
      fail
    else node_1 := node_2
  if node_2 = v then
    return(true)
  else
    return(false)
```

Figure 8.9 A nondeterministic program counting vertices reachable from u. Comments are enclosed in braces {, }.

value to the procedure IS-NODE-AT-$\leq$-DIST-FROM-u, which examines and counts all possible $next\_to\_last\_nodes$ reachable from u. #-VERTICES-AT-$\leq$-DISTANCE-FROM-u either fails to find all possible vertices at distance $dist - 1$, in which case it fails, or finds all such vertices. Thus, it nondeterministically verifies that all possible paths from u have been explored. IS-NODE-AT-$\leq$-DIST-FROM-u uses the procedure EXISTS-PATH-FROM-u-TO-v-$\leq$-LENGTH that either correctly verifies that a path of length $dist - 1$ exists from u to $next\_to\_last\_node$ or fails. In turn, EXISTS-PATH-FROM-u-TO-v-$\leq$-LENGTH uses the command NONDETERMINISTIC-GUESS$([1, .., n])$ to nondeterministically choose nodes on a path from u to v.

Since this program is not recursive, it uses a fixed number of variables. Because these variables assume values in the range $[1, 2, 3, \ldots, n]$, it follows that space $O(\log n)$ suffices to implement it on an NDTM. ■

We now extend this result to nondeterministic space computations.

COROLLARY 8.6.2 *If $r(n) = \Omega(\log n)$ is proper,* **NSPACE**$(r(n)) =$ **coNSPACE**$(r(n))$.

Proof Let $L \in$ **NSPACE**$(r(n))$ be decided by an $r(n)$-space bounded NDTM M. We show that the complement of L can be decided by a nondeterministic $r(n)$-space bounded Turing machine $\overline{M}$, stopping on all inputs. We modify slightly the program of Fig. 8.9 for this purpose. The graph G is the configuration graph of M. Its initial state is determined by the string $\boldsymbol{w}$ that is initially written on M's input tape. To determine adjacency between two vertices in the configuration graph, computations of M are simulated on one of $\overline{M}$'s work tapes.

$\overline{M}$ computes a slightly modified version of COUNTING-REACHABILITY. First, if the procedure IS-NODE-AT-LENGTH-$\leq$-DIST-FROM-u returns **true** for a vertex u that is a halting accepting configuration of M, then $\overline{M}$ halts and rejects the string. If the procedure COUNTING-REACHABILITY completes successfully without rejecting any string, then $\overline{M}$ halts and accepts the input string because every possible accepting computation for the input string has been examined and none of them is accepting. This computation is nondeterministic.

The space used by $\overline{M}$ is the space needed for COUNTING-REACHABILITY, which means it is $O(\log N)$, where N is the number of vertices in the configuration graph of M plus the space for a simulation of M, which is $O(r(n))$. Since $N = O(k^{\log n + r(n)})$ (see the proof of Theorem 8.5.6), the total space for this computation is $O(\log n + r(n))$, which is $O(r(n))$ if $r(n) = \Omega(\log n)$. By definition $\overline{L} \in$ **coNSPACE**$(r(n))$. From the above construction $\overline{L} \in$ **NSPACE**$(r(n))$. Thus, **coNSPACE**$(r(n)) \subseteq$ **NSPACE**$(r(n))$.

By similar reasoning, if $L \in$ **coNSPACE**$(r(n))$, then $\overline{L} \in$ **NSPACE**$(r(n))$, which implies that **NSPACE**$(r(n)) \subseteq$ **coNSPACE**$(r(n))$; that is, they are equal. ■

The lowest class in the space hierarchy that is known to be closed under complements is the class **NL**; that is, **NL** = **coNL**. This result is used in Section 8.11 to show that the problem 2-SAT, a specialization of the **NP**-complete problem 3-SAT, is in **P**.

From Theorem 8.6.1 we know that all deterministic time and space complexity classes are closed under complements. From Corollary 8.6.2 we also know that all nondeterministic space complexity classes with space $\Omega(\log n)$ are closed under complements. However, we do not yet know whether the nondeterministic time complexity classes are closed under complements.

This important question is related to the question whether $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$, because if $\mathbf{NP} \neq \mathbf{coNP}$, then $\mathbf{P} \neq \mathbf{NP}$ because **P** is closed under complements but **NP** is not.

8.6.1 The Complement of NP

The class **coNP** is the class of decision problems whose complements are in **NP**. That is, **coNP** is the language of "No" instances of problems in **NP**. The decision problem VALIDITY defined below is an example of a problem in **coNP**. In fact, it is log-space complete for **coNP**. (See Problem 8.10.) VALIDITY identifies SOPEs (the sum-of-products expansion, defined in Section 2.3) that can have value 1.

VALIDITY
Instance: A set of literals $X = \{x_1, \overline{x}_1, x_2, \overline{x}_2, \ldots, x_n, \overline{x}_n\}$, and a sequence of products $P = (p_1, p_2, \ldots, p_m)$, where each product p_i is a subset of X.
Answer: "Yes" if for all assignments of Boolean values to variables in $\{x_1, x_2, \ldots, x_n\}$ every literal in at least one product has value 1.

Given a language L in **NP**, a string in L has a certificate for its membership in L consisting of the set of choices that cause its recognizing Turing machine to accept it. For example, a certificate for SATISFIABILITY is a set of values for its variables satisfying at least one literal in each sum. For an instance of a problem in **coNP**, a **disqualification** is a certificate for the complement of the instance. An instance in **co**VALIDITY is disqualified by an assignment that causes all products to have value 0. Thus, each "Yes" instance in VALIDITY is disqualified by an assignment that prevents the expression from being valid. (See Problem 8.11.)

As mentioned just before the start of this section, if $\mathbf{NP} \neq \mathbf{coNP}$, then $\mathbf{P} \neq \mathbf{NP}$ because **P** is closed under complements. Because we know of no way to establish $\mathbf{NP} \neq \mathbf{coNP}$, we try to identify a problem that is in **NP** but is not known to be in **P**. A problem that is **NP** and **coNP** simultaneously (the class $\mathbf{NP} \cap \mathbf{coNP}$) is a possible candidate for a problem that is in **NP** but not **P**, which would show that $\mathbf{P} \neq \mathbf{NP}$. We show that PRIMALITY is in $\mathbf{NP} \cap \mathbf{coNP}$. (It is straightforward to show that $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{coNP}$. See Problem 8.12.)

PRIMALITY
Instance: An integer n written in binary notation.
Answer: "Yes" if n is a prime.

A disqualification for PRIMALITY is an integer that is a factor of n. Thus, the complement of PRIMALITY is in **NP**, so PRIMALITY is in **coNP**. We now show that PRIMALITY is also in **NP** or that it is in $\mathbf{NP} \cap \mathbf{coNP}$. To prove the desired result we need the following result from number theory, which we do not prove (see [234, p. 222] for a proof).

THEOREM 8.6.3 *An integer $p > 2$ is prime if and only if there is an integer $1 < r < p$ such that $r^{p-1} = 1 \bmod p$ and for all prime divisors q of $p-1$, $r^{(p-1)/q} \neq 1 \bmod p$.*

As a consequence, to give evidence of primality of an integer $p > 1$, we need only provide an integer r, $1 < r < p$, and the prime divisors $\{q_1, \ldots, q_k\}$ other than 1 of $p-1$ and then show that $r^{p-1} = 1 \bmod p$ and $r^{(p-1)/q} \neq 1 \bmod p$ for $q \in \{q_1, \ldots, q_k\}$. By the theorem, such integers exist if and only if p is prime. In turn, we must give evidence that the integers $\{q_1, \ldots, q_k\}$ are prime divisors of $p-1$, which requires showing that they divide $p-1$ and are prime. We must also show that k is small and that the recursive check of the primes does

not grow exponentially. Evidence of the primality of the divisors can be given in the same way, that is, by exhibiting an integer r_j for each prime as well as the prime divisors of $q_j - 1$ for each prime q_j. We must then show that all of this evidence can be given succinctly and verified deterministically in time polynomial in the length n of p.

THEOREM 8.6.4 PRIMALITY *is in* **NP** $\cap$ **coNP**.

Proof We give an inductive proof that PRIMALITY is in **NP**. For a prime p we give its evidence $E(p)$ as $(p; r, E(q_1), \ldots, E(q_k))$, where $E(q_j)$ is evidence for the prime q_j. We let the evidence for the base case $p = 2$ be $E(2) = (2)$. Then, $E(3) = (3; 2, (2))$ because $r = 2$ works for this case and 2 is the only prime divisor of $3 - 1$, and (2) is the evidence for it. Also, $E(5) = (5; 3, (2))$. The **length** $|E(p)|$ **of the evidence** $E(p)$ on p is the number of parentheses, commas and bits in integers forming part of the evidence.

We show by induction that $|E(p)|$ is at most $4 \log_2^2 p$. The base case satisfies the hypothesis because $|E(2)| = 4$.

Because the prime divisors $\{q_1, \ldots, q_k\}$ satisfy $q_i \geq 2$ and $q_1 q_2 \cdots q_k \leq p-1$, it follows that $k \leq \lfloor \log_2 p \rfloor \leq n$. Also, since p is prime, it is odd and $p - 1$ is divisible by 2. Thus, the first prime divisor of $p - 1$ is 2.

Let $E(p) = (p; r, E(2), E(q_2), \ldots, E(q_k))$. Let the inductive hypothesis be that $|E(p)| \leq 4 \log_2^2 p$. Let $n_j = \log_2 q_j$. From the definition of $E(p)$ we have that $|E(p)|$ satisfies the following inequality because at most n bits are needed for p and r, there are $k - 1 \leq n - 1$ commas and three other punctuation marks, and $|E(2)| = 4$.

$$|E(p)| \leq 3n + 6 + 4 \sum_{2 \leq j \leq k} n_j^2$$

Since the q_j are the prime divisors of $p - 1$ and some primes may be repeated in $p - 1$, their product (which includes $q_1 = 2$) is at most $p - 1$. It follows that $\sum_{2 \leq j \leq k} n_j \leq \log_2 \Pi_{2 \leq j \leq k} q_j \leq \log((p-1)/2)$. Since the sum of the squares of n_j is less than or equal to the square of the sum of n_j, it follows that the sum in the above expression is at most $(\log_2 p - 1)^2 \leq (n-1)^2$. But $3n + 6 + 4(n-1)^2 = 4n^2 - 5n + 10 \leq 4n^2$ when $n \geq 2$. Thus, the description of a certificate for the primality of p is polynomial in the length n of p.

We now show by induction that a prime p can be verified in $O(n^4)$ steps on a RAM. Assume that the divisors $q_1, \ldots, q_k$ for $p - 1$ have been verified. To verify p, we compute $r^{p-1} \bmod p$ from r and p as well as $r^{(p-1)/q} \bmod p$ for each of the prime divisors q of $p - 1$ and compare the results with 1. The integers $(p - 1)/q$ can be computed through subtraction of n-bit numbers in $O(n^2)$ steps on a RAM. To raise r to an exponent e, represent e as a binary number. For example, if $e = 7$, write it as $p = 2^2 + 2^1 + 2^0$. If t is the largest such power of 2, $t \leq \log_2(p - 1) \leq n$. Compute $r^{2^j} \bmod p$ by squaring r j times, each time reducing it by p through division. Since each squaring/reduction step takes $O(n^2)$ RAM steps, at most $O(jn^2)$ RAM steps are required to compute r^{2^j}. Since this may be done for $2 \leq j \leq t$ and $\sum_{2 \leq j \leq t} j = O(t^2)$, at most $O(n^3)$ RAM steps suffice to compute one of $r^{p-1} \bmod p$ or $r^{(p-1)/q} \bmod p$ for a prime divisor q. Since there are at most n of these quantities to compute, $O(n^4)$ RAM steps suffice to compute them.

To complete the verification of the prime p, we also need to verify the divisors $q_1, \ldots, q_k$ of $p - 1$. We take as our inductive hypothesis that an arbitrary prime q of n bits can be verified in $O(n^5)$ steps. Since the sum of the number of bits in $q_2, \ldots, q_k$ is $(\log_2(p-1)/2 - 1)$ and the sum of the kth powers is no more than the kth power of the sum, it follows that

$O(n^5)$ RAM steps suffice to verify p. Since a polynomial number of RAM steps can be executed in a polynomial number of Turing machine steps, PRIMALITY is in **NP**. ■

Since **NP** ∩ **coNP** ⊆ **NP** and **NP** ∩ **coNP** ⊆ **coNP** as well as **NP** ⊆ **NP** ∪ **coNP** and **coNP** ⊆ **NP** ∪ **coNP**, we begin to have the makings of a hierarchy. If we add that **coNP** ⊆ **PSPACE** (see Problem 8.13), we have the relationships between complexity classes shown schematically in Fig. 8.7.

8.7 Reductions

In this section we specialize the reductions introduced in Section 2.4 and use them to classify problems into categories. We show that if problem A is reduced to problem B by a function in the set R and A is hard relative to R, then B cannot be easy relative to R because A can be solved easily by reducing it to B and solving B with an easy algorithm, contradicting the fact that A is hard. On the other hand, if B is easy to solve relative to R, then A must be easy to solve. Thus, reductions can be used to show that some problems are hard or easy. Also, if A can be reduced to B by a function in R and vice versa, then A and B have the same complexity relative to R.

Reductions are widely used in computer science; we use them whenever we specialize one procedure to realize another. Thus, reductions in the form of simulations are used throughout Chapter 3 to exhibit circuits that compute the same functions that are computed by finite-state, random-access, and Turing machines, with and without nondeterminism. Simulations prove to be an important type of reduction. Similarly, in Chapter 10 we use simulation to show that any computation done in the pebble game can be simulated by a branching program.

Not only did we simulate machines with memory by circuits in Chapter 3, but we demonstrated in Sections 3.9.5 and 3.9.6 that the languages CIRCUIT VALUE and CIRCUIT SAT describing circuits are **P**-complete and **NP**-complete, respectively. We demonstrated that each string $\boldsymbol{x}$ in an arbitrary language in **P** (**NP**) could be translated into a string in CIRCUIT VALUE (respectively, CIRCUIT SAT) by a program whose running time is polynomial in the length of $\boldsymbol{x}$ and whose space is logarithmic in its length.

In this chapter we extend these results. We consider primarily transformations (also called **many-one reductions** and just **reductions** in Section 5.8.1), a type of reduction in which an instance of one decision problem is translated to an instance of a second problem such that the former is a "yes" instance if and only if the latter is a "yes" instance. A **Turing reduction** is a second type of reduction that is defined by an oracle Turing machine. (See Section 8.4.2 and Problem 8.15.) In this case the Turing machine may make more than one call to the second problem (the oracle). A transformation is equivalent to an oracle Turing reduction that makes one call to the oracle. Turing reductions subsume all previous reductions used elsewhere in this book. (See Problems 8.15 and 8.16.) However, since the results of this section can be derived with the weaker transformations, we limit our attention to them.

DEFINITION 8.7.1 *If L_1 and L_2 are languages, a* **transformation** *h from L_1 to L_2 is a DTM-computable function $h : \mathcal{B}^* \mapsto \mathcal{B}^*$ such that $\boldsymbol{x} \in L_1$ if and only if $h(\boldsymbol{x}) \in L_2$. A* **resource-bounded transformation** *is a transformation that is computed under a resource bound such as deterministic logarithmic space or polynomial time.*

The classification of problems is simplified by considering classes of transformations. These classes will be determined by bounds on resources such as space and time on a Turing machine or circuit size and depth.

DEFINITION 8.7.2 *For decision problems $\mathcal{P}_1$ and $\mathcal{P}_2$, the notation $\mathcal{P}_1 \leq_R \mathcal{P}_2$ means that $\mathcal{P}_1$ can be transformed to $\mathcal{P}_2$ by a transformation in the class R.*

Compatibility among transformation classes and complexity classes helps determine conditions under which problems are hard.

DEFINITION 8.7.3 *Let C be a complexity class, R a class of resource-bounded transformations, and $\mathcal{P}_1$ and $\mathcal{P}_2$ decision problems. A set of transformations R is* **compatible** *with C if $\mathcal{P}_1 \leq_R \mathcal{P}_2$ and $\mathcal{P}_2 \in C$, then $\mathcal{P}_1 \in C$.*

It is easy to see that the **polynomial-time transformations** (denoted $\leq_{\mathrm{p}}$) are compatible with **P**. (See Problem 8.17.) Also compatible with **P** are the **log-space transformations** (denoted $\leq_{\text{log-space}}$) associated with transformations that can be computed in logarithmic space. Log-space transformations are also polynomial transformations, as shown in Theorem 8.5.8.

8.8 Hard and Complete Problems

Classes of problems are defined above by their use of space and time. We now set the stage for the identification of problems that are hard relative to members of these classes. A few more definitions are needed before we begin this task.

DEFINITION 8.8.1 *A class R of transformations is* **transitive** *if the composition of any two transformations in R is also in R and for all problems $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$, $\mathcal{P}_1 \leq_R \mathcal{P}_2$ and $\mathcal{P}_2 \leq_R \mathcal{P}_3$ implies that $\mathcal{P}_1 \leq_R \mathcal{P}_3$.*

If a class R of transformations is transitive, then we can compose any two transformations in the class and obtain another transformation in the class. Transitivity is used to define hard and complete problems.

The transformations $\leq_p$ and $\leq_{\text{log-space}}$ described above are transitive. Below we show that $\leq_{\text{log-space}}$ is transitive and leave to the reader the proof of transitivity of $\leq_p$ and the polynomial-time Turing reductions. (See Problem 8.19.)

THEOREM 8.8.1 *Log-space transformations are transitive.*

Proof A log-space transformation is a DTM that has a read-only input tape, a write-only output tape, and a work tape or tapes on which it uses $O(\log n)$ cells to process an input string $\boldsymbol{w}$ of length n. As shown in Theorem 8.5.8, such DTMs halt within polynomial time. We now design a machine T that composes two log-space transformations in logarithmic space. (See Fig. 8.10.)

Let M_1 and M_2 denote the first and second log-space DTMs. When M_1 and M_2 are composed to form T, the output tape of M_1, which is also the input tape of M_2, becomes a work tape of T. Since M_1 may execute a polynomial number of steps, we cannot store all its output before beginning the computation by M_2. Instead we must be more clever. We keep the contents of the work tapes of both machines as well as (and this is where we are

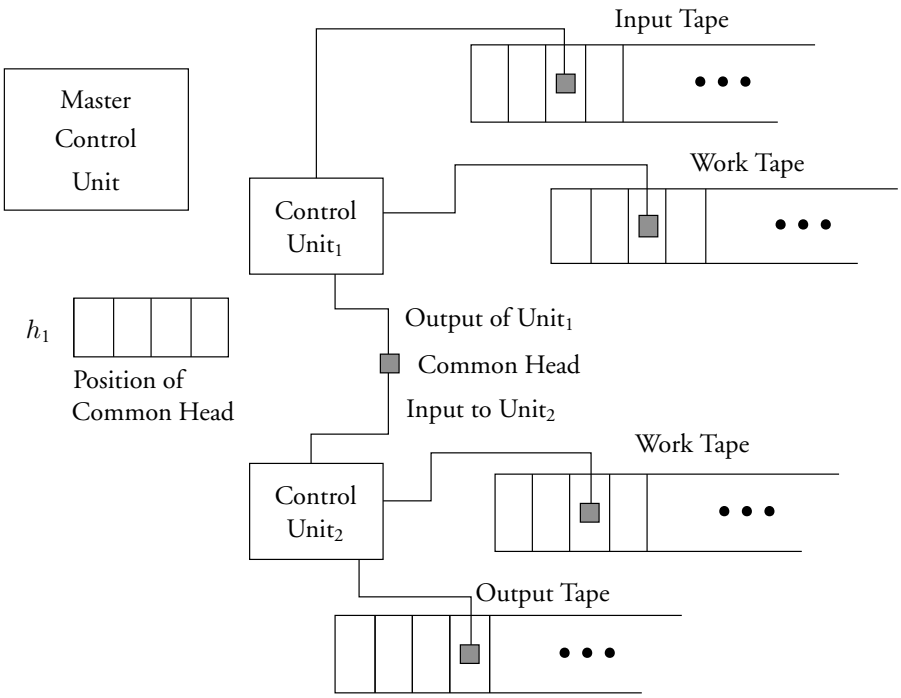


Figure 8.10 The composition of two deterministic log-space Turing machines.

clever) an integer h_1 recording the position of the input head of M_2 on the output tape of M_1. If M_2 moves its input head right by one step, M_1 is simulated until one more output is produced. If its head moves left, we decrement h_1, restart M_1, and simulate it until h_1 outputs are produced and then supply this output as an input to M_2.

The space used by this simulation is the space used by M_1 and M_2 plus the space for h_1, the value under the input head of M_2 and some temporary space. The total space is logarithmic in n since h_1 is at most a polynomial in n. ■

We now apply transitivity of reductions to define hard and complete problems.

DEFINITION 8.8.2 *Let R be a class of reductions, let C be a complexity class, and let R be compatible with C. A problem $\mathcal{Q}$ is* **hard for C under R-reductions** *if for every problem $\mathcal{P} \in C$, $\mathcal{P} \leq_R \mathcal{Q}$. A problem $\mathcal{Q}$ is* **complete for C under R-reductions** *if it is hard for C under R-reductions and is a member of C.*

Problems are hard for a class if they are as hard to solve as any other problem in the class. Sometimes problems are shown hard for a class without showing that they are members of that class. Complete problems are members of the class for which they are hard. Thus, complete problems are the hardest problems in the class. We now define three important classes of complete problems.

DEFINITION 8.8.3 *Problems in* **P** *that are hard for* **P** *under log-space reductions are called* **P-complete**. *Problems in* **NP** *that are hard for* **NP** *under polynomial-time reductions are called* **NP-complete**. *Problems in* **PSPACE** *that are hard for* **PSPACE** *under polynomial-time reductions are called* **PSPACE-complete**.

We state Theorem 8.8.2, which follows directly from Definition 8.7.3 and transitivity of log-space and polynomial-time reductions, because it incorporates as conditions the goals of the study of **P**-complete, **NP**-complete, and **PSPACE**-complete problems, namely, to show that all problems in **P** can be solved in log-space and all problems in **NP** and **PSPACE** can be solved in polynomial time. It is unlikely that any of these goals can be reached.

THEOREM 8.8.2 *If a* **P***-complete problem can be solved in log-space, then all problems in* **P** *can be solved in log-space. If an* **NP***-complete problem is in* **P**, *then* **P** = **NP**. *If a* **PSPACE***-complete problem is in* **P**, *then* **P** = **PSPACE**.

In Theorem 8.14.2 we show that if a **P**-complete problem can be solved in poly-logarithmic time with polynomially many processors on a CREW PRAM (they are fully parallelizable), then so can all problems in **P**. It is considered unlikely that all languages in **P** can be fully parallelized. Nonetheless, the question of the parallelizability of **P** is reduced to deciding whether **P**-complete problems are parallelizable.

8.9 P-Complete Problems

To show that a problem $\mathcal{P}$ is **P**-complete we must show that it is in **P** and that all problems in **P** can be reduced to $\mathcal{P}$ via a log-space reduction. (See Section 3.9.5.) The task of showing this is simplified by the knowledge that log-space reductions are transitive: if another problem $\mathcal{Q}$ has already been shown to be **P**-complete, to show that $\mathcal{P}$ is **P**-complete it suffices to show there is a log-space reduction from $\mathcal{Q}$ to $\mathcal{P}$ and that $\mathcal{P} \in$ **P**.

CIRCUIT VALUE
Instance: A circuit description with fixed values for its input variables and a designated output gate.
Answer: "Yes" if the output of the circuit has value 1.

In Section 3.9.5 we show that the CIRCUIT VALUE problem described above is **P**-complete by demonstrating that for every decision problem $\mathcal{P}$ in **P** an instance $\boldsymbol{w}$ of $\mathcal{P}$ and a DTM M that recognizes "Yes" instances of $\mathcal{P}$ can be translated by a log-space DTM into an instance $\boldsymbol{c}$ of CIRCUIT VALUE such that $\boldsymbol{w}$ is a "Yes" instance of $\mathcal{P}$ if and only if $\boldsymbol{c}$ is a "Yes" instance of CIRCUIT VALUE.

Since **P** is closed under complements (see Theorem 8.6.1), it follows that if the "Yes" instances of a decision problem can be determined in polynomial time, so can the "No" instances. Thus, the CIRCUIT VALUE problem is equivalent to determining the value of a circuit from its description. Note that for CIRCUIT VALUE the values of all variables of a circuit are included in its description.

CIRCUIT VALUE is in **P** because, as shown in Theorem 8.13.2, a circuit can be evaluated in a number of steps proportional at worst to the square of the length of its description. Thus, an instance of CIRCUIT VALUE can be evaluated in a polynomial number of steps.

Monotone circuits are constructed of AND and OR gates. The functions computed by monotone circuits form an asymptotically small subset of the set of Boolean functions. Also, many important Boolean functions are not monotone, such as binary addition. But even though monotone circuits are a very restricted class of circuits, the monotone version of CIRCUIT VALUE, defined below, is also **P**-complete.

MONOTONE CIRCUIT VALUE
Instance: A description for a monotone circuit with fixed values for its input variables and a designated output gate.
Answer: "Yes" if the output of the circuit has value 1.

CIRCUIT VALUE is a starting point to show that many other problems are **P**-complete. We begin by reducing it to MONOTONE CIRCUIT VALUE.

THEOREM 8.9.1 MONOTONE CIRCUIT VALUE *is* **P***-complete.*

Proof As shown in Problem 2.12, every Boolean function can be realized with just AND and OR gates (this is known as dual-rail logic) if the values of input variables and their complements are made available. We reduce an instance of CIRCUIT VALUE to an instance of MONOTONE CIRCUIT VALUE by replacing each gate with the pair of monotone gates described in Problem 2.12. Such descriptions can be written out in log-space if the gates in the monotone circuit are numbered properly. (See Problem 8.20.) The reduction must also write out the values of variables of the original circuit and their complements. ■

The class of **P**-complete problems is very rich. Space limitations require us to limit our treatment of this subject to two more problems. We now show that LINEAR INEQUALITIES described below is **P**-complete. LINEAR INEQUALITIES is important because it is directly related to LINEAR PROGRAMMING, which is widely used to characterize optimization problems. The reader is asked to show that LINEAR PROGRAMMING is **P**-complete. (See Problem 8.21.)

LINEAR INEQUALITIES
Instance: An integer-valued $m \times n$ matrix A and column m-vector $\boldsymbol{b}$.
Answer: "Yes" if there is a rational column n-vector $\boldsymbol{x} > \mathbf{0}$ (all components are non-negative and at least one is non-zero) such that $A\boldsymbol{x} \leq \boldsymbol{b}$.

We show that LINEAR INEQUALITIES is **P**-hard, that is, that every problem in **P** can be reduced to it in log-space. The proof that LINEAR INEQUALITIES is in **P**, an important and difficult result in its own right, is not given here. (See [164].)

THEOREM 8.9.2 LINEAR INEQUALITIES *is* **P***-hard.*

Proof We give a log-space reduction of CIRCUIT VALUE to LINEAR INEQUALITIES. That is, we show that in log-space an instance of CIRCUIT VALUE can be transformed to an instance of LINEAR INEQUALITIES so that an instance of CIRCUIT VALUE is a "Yes" instance if and only if the corresponding instance of LINEAR INEQUALITIES is a "Yes" instance.

The log-space reduction that we use converts each gate and input in an instance of a circuit into a set of inequalities. The inequalities describing each gate are shown below. (An equality relation $a = b$ is equivalent to two inequality relations, $a \leq b$ and $b \leq a$.) The reduction also writes the equality $z = 1$ for the output gate z. Since each variable must be non-negative, this last condition insures that the resulting vector of variables, $\boldsymbol{x}$, satisfies $\boldsymbol{x} > \mathbf{0}$.

	Input		*Gates*		
Type	TRUE	FALSE	NOT	AND	OR
Function	$x_i = 1$	$x_i = 0$	$w = \neg u$	$w = u \wedge v$	$w = u \vee v$
Inequalities	$x_i = 1$	$x_i = 0$	$0 \leq w \leq 1$	$0 \leq w \leq 1$	$0 \leq w \leq 1$
			$w = 1 - u$	$w \leq u$	$u \leq w$
				$w \leq v$	$v \leq w$
				$u + v - 1 \leq w$	$w \leq u + v$

Given an instance of CIRCUIT VALUE, each assignment to a variable is translated into an equality statement of the form $x_i = 0$ or $x_i = 1$. Similarly, each AND, OR, and NOT gate is translated into a set of inequalities of the form shown above. Logarithmic temporary space suffices to hold gate numbers and to write these inequalities because the number of bits needed to represent each gate number is logarithmic in the length of an instance of CIRCUIT VALUE.

To see that an instance of CIRCUIT VALUE is a "Yes" instance if and only if the instance of LINEAR INEQUALITIES is also a "Yes" instance, observe that inputs of 0 or 1 to a gate result in the correct output if and only if the corresponding set of inequalities forces the output variable to have the same value. By induction on the size of the circuit instance, the values computed by each gate are exactly the same as the values of the corresponding output variables in the set of inequalities. ■

We give as our last example of a **P**-complete problem DTM ACCEPTANCE, the problem of deciding if a string is accepted by a deterministic Turing machine in a number of steps specified as a unary number. (The integer k is represented as a unary number by a string of k characters.) For this problem it is more convenient to give a direct reduction from all problems in **P** to DTM ACCEPTANCE.

DTM ACCEPTANCE
Instance: A description of a DTM M, a string $\boldsymbol{w}$, and an integer n written in unary.
Answer: "Yes" if and only if M, when started with input $\boldsymbol{w}$, halts with the answer "Yes" in at most n steps.

THEOREM 8.9.3 DTM ACCEPTANCE *is* **P***-complete.*

Proof To show that DTM ACCEPTANCE is log-space complete for **P**, consider an arbitrary problem $\mathcal{P}$ in **P** and an arbitrary instance of $\mathcal{P}$, namely $\boldsymbol{x}$. There is some Turing machine, say $M_{\mathcal{P}}$, that accepts instances $\boldsymbol{x}$ of $\mathcal{P}$ of length n in time $p(n)$, p a polynomial. We assume that p is included with the specification of $M_{\mathcal{P}}$. For example, if $p(y) = 2y^4 + 3y^2 + 1$, we can represent it with the string $((2, 4), (3, 2), (1, 0))$. The log-space Turing machine that translates $M_{\mathcal{P}}$ and $\boldsymbol{x}$ into an instance of DTM ACCEPTANCE writes the description of $M_{\mathcal{P}}$ together with the input $\boldsymbol{x}$ and the value of $p(n)$ in unary. Constant temporary space suffices to move the descriptions of $M_{\mathcal{P}}$ and $\boldsymbol{x}$ to the output tape. To complete the proof we need only show that $O(\log n)$ temporary space suffices to write the value in $p(n)$ in unary, where n is the length of $\boldsymbol{x}$.

Since the length of the input $\boldsymbol{x}$ is provided in unary, that is, by the number of characters it contains, its length n can be written in binary on a work tape in space $O(\log n)$ by counting the number of characters in $\boldsymbol{x}$. Since it is not difficult to show that any power of a k-bit binary number can be computed by a DTM in work space $O(k)$, it follows that any fixed polynomial in n can be computed by a DTM in work space $O(k) = O(\log n)$. (See Problem 8.18.)

To show that DTM ACCEPTANCE is in **P**, we design a Turing machine that accepts the "Yes" instances in polynomial time. This machine copies the unary string of length n to one of its work tapes. Given the description of the DTM M, it simulates M with a universal Turing machine on input $\boldsymbol{w}$. When it completes a step, it advances the head on the work tape containing n in unary, declaring the instance of DTM ACCEPTANCE accepted if M terminates without using more than n steps. By definition, it will complete its simulation of M in at most n of M's steps each of which uses a constant number of steps on the simulating machine. That is, it accepts a "Yes" instance of DTM ACCEPTANCE in time polynomial in the length of the input. ■

8.10 NP-Complete Problems

As mentioned above, the **NP**-complete problems are the problems in **NP** that are the most difficult to solve. We have shown that **NP** $\subseteq$ **PSPACE** $\subseteq$ **EXPTIME** or that every problem in **NP**, including the **NP**-complete problems, can be solved in exponential time. Since the **NP**-complete problems are the hardest problems in **NP**, each of these is at worst an exponential-time problem. Thus, we know that the **NP**-complete problems require either polynomial or exponential time, but we don't know which.

The CIRCUIT SAT problem is to determine from a description of a circuit whether it can be **satisfied**; that is, whether values can be assigned to its inputs such that the circuit output has value 1. As mentioned above, this is our canonical **NP**-complete problem.

CIRCUIT SAT
Instance: A circuit description with n input variables $\{x_1, x_2, \ldots, x_n\}$ for some integer n and a designated output gate.
Answer: "Yes" if there is an assignment of values to the variables such that the output of the circuit has value 1.

As shown in Section 3.9.6, CIRCUIT SAT is an **NP**-complete problem. The goal of this problem is to recognize the "Yes" instances of CIRCUIT SAT, instances for which there are values for the input variables such that the circuit has value 1.

In Section 3.9.6 we showed that CIRCUIT SAT described above is **NP**-complete by demonstrating that for every decision problem $\mathcal{P}$ in **NP** an instance $\boldsymbol{w}$ of $\mathcal{P}$ and an NDTM M that accepts "Yes" instances of $\mathcal{P}$ can be translated by a polynomial-time (actually, a log-space) DTM into an instance $\boldsymbol{c}$ of CIRCUIT SAT such that $\boldsymbol{w}$ is a "Yes" instance of $\mathcal{P}$ if and only if $\boldsymbol{c}$ is a "Yes" instance of CIRCUIT SAT.

Although it suffices to reduce problems in **NP** via a polynomial-time transformation to an **NP**-complete problem, each of the reductions given in this chapter can be done by a log-space transformation. We now show that a variety of other problems are **NP**-complete.

8.10.1 **NP**-Complete Satisfiability Problems

In Section 3.9.6 we showed that SATISFIABILITY defined below is **NP**-complete. In this section we demonstrate that two variants of this language are **NP**-complete by simple extensions of the basic proof that CIRCUIT SAT is **NP**-complete.

SATISFIABILITY
Instance: A set of **literals** $X = \{x_1, \overline{x}_1, x_2, \overline{x}_2, \ldots, x_n, \overline{x}_n\}$ and a sequence of **clauses** $C = (c_1, c_2, \ldots, c_m)$, where each clause c_i is a subset of X.
Answer: "Yes" if there is a (satisfying) assignment of values for the variables $\{x_1, x_2, \ldots, x_n\}$ over the set $\mathcal{B}$ such that each clause has at least one literal whose value is 1.

The two variants of SATISFIABILITY are 3-SAT, which has at most three literals in each clause, and NAESAT, in which not all literals in each clause have the same value.

3-SAT
Instance: A set of literals $X = \{x_1, \overline{x}_1, x_2, \overline{x}_2, \ldots, x_n, \overline{x}_n\}$, and a sequence of clauses $C = (c_1, c_2, \ldots, c_m)$, where each clause c_i is a subset of X containing at most three literals.
Answer: "Yes" if there is an assignment of values for variables $\{x_1, x_2, \ldots, x_n\}$ over the set $\mathcal{B}$ such that each clause has at least one literal whose value is 1.

THEOREM 8.10.1 3-SAT *is* **NP***-complete.*

Proof The proof that SATISFIABILITY is **NP**-complete also applies to 3-SAT because each of the clauses produced in the transformation of instances of CIRCUIT SAT has at most three literals per clause. ■

NAESAT
Instance: An instance of 3-SAT.
Answer: "Yes" if each clause is satisfiable when not all literals have the same value.

NAESAT contains as its "Yes" instances those instances of 3-SAT in which the literals in each clause are not all equal.

THEOREM 8.10.2 NAESAT *is* **NP***-complete.*

Proof We reduce CIRCUIT SAT to NAESAT using almost the same reduction as for 3-SAT. Each gate is replaced by a set of clauses. (See Fig. 8.11.) The only difference is that we add the new literal y to each two-literal clause associated with AND and OR gates and to the clause associated with the output gate. Clearly, this reduction can be performed in deterministic log-space. Since a "Yes" instance of NAESAT can be verified in nondeterministic polynomial time, NAESAT is in **NP**. We now show that it is **NP**-hard.

Given a "Yes" instance of CIRCUIT SAT, we show that the instance of 3-SAT is a "Yes" instance. Since every clause is satisfied in a "Yes" instance of CIRCUIT SAT, every clause of the corresponding instance of NAESAT has at least one literal with value 1. The clauses that don't contain the literal y by their nature have not all literals equal. Those containing y can be made to satisfy this condition by setting y to 0, thereby providing a "Yes" instance of NAESAT.

Now consider a "Yes" instance of NAESAT produced by the mapping from CIRCUIT SAT. Replacing every literal by its complement generates another "Yes" instance of NAESAT

Step Type				*Corresponding Clauses*		
$(i$	READ	$x)$		$(\overline{g}_i \vee x)$	$(g_i \vee \overline{x})$	
$(i$	NOT	$j)$		$(\overline{g}_i \vee \overline{g}_j)$	$(g_i \vee g_j)$	
$(i$	OR	j	$k)$	$(g_i \vee \overline{g}_j \vee y)$	$(g_i \vee \overline{g}_k \vee y)$	$(\overline{g}_i \vee g_j \vee g_k)$
$(i$	AND	j	$k)$	$(\overline{g}_i \vee g_j \vee y)$	$(\overline{g}_i \vee g_k \vee y)$	$(g_i \vee \overline{g}_j \vee \overline{g}_k)$
$(i$	OUTPUT	$j)$		$(g_j \vee y)$		

Figure 8.11 A reduction from CIRCUIT SAT to NAESAT is obtained by replacing each gate in a "Yes" instance of CIRCUIT SAT by a set of clauses. The clauses used in the reduction from CIRCUIT SAT to 3-SAT (see Section 3.9.6) are those shown above with the literal y removed. In the reduction to NAESAT the literal y is added to the 2-literal clauses used for AND and OR gates and to the output clause.

since the literals in each clause are not all equal, a property that applies before and after complementation. In one of these "Yes" instances y is assigned the value 0. Because this is a "Yes" instance of NAESAT, at least one literal in each clause has value 1; that is, each clause is satisfiable. This implies that the original CIRCUIT SAT problem is satisfiable. It follows that an instance of CIRCUIT SAT has been translated into an instance of NAESAT so that the former is a "Yes" instance if and only if the latter is a "Yes" instance. ■

8.10.2 Other NP-Complete Problems

This section gives a sampling of additional **NP**-complete problems. Following the format of the previous section, we present each problem and then give a proof that it is **NP**-complete. Each proof includes a reduction of a problem previously shown **NP**-complete to the current problem. The succession of reductions developed in this book is shown in Fig. 8.12.

INDEPENDENT SET
Instance: A graph $G = (V, E)$ and an integer k.
Answer: "Yes" if there is a set of k vertices of G such that there is no edge in E between them.

THEOREM 8.10.3 INDEPENDENT SET *is* **NP***-complete.*

Proof INDEPENDENT SET is in **NP** because an NDTM can propose and then verify in polynomial time a set of k independent vertices. We show that INDEPENDENT SET is **NP**-hard by reducing 3-SAT to it. We begin by showing that a restricted version of 3-SAT, one in which each clause contains exactly three literals, is also **NP**-complete. If for some variable x, both x and $\overline{x}$ are in the same clause, we eliminate the clause since it is always satisfied. Second, we replace each 2-literal clause $(a \vee b)$ with the two 3-literal clauses $(a \vee b \vee z)$ and $(a \vee b \vee \overline{z})$, where z is a new variable. Since z is either 0 or 1, if all clauses are satisfied then $(a \vee b)$ has value 1 in both causes. Similarly, a clause with a single literal can be transformed to one containing three literals by introducing two new variables and replacing the clause containing the single literal with four clauses each containing three literals. Since adding

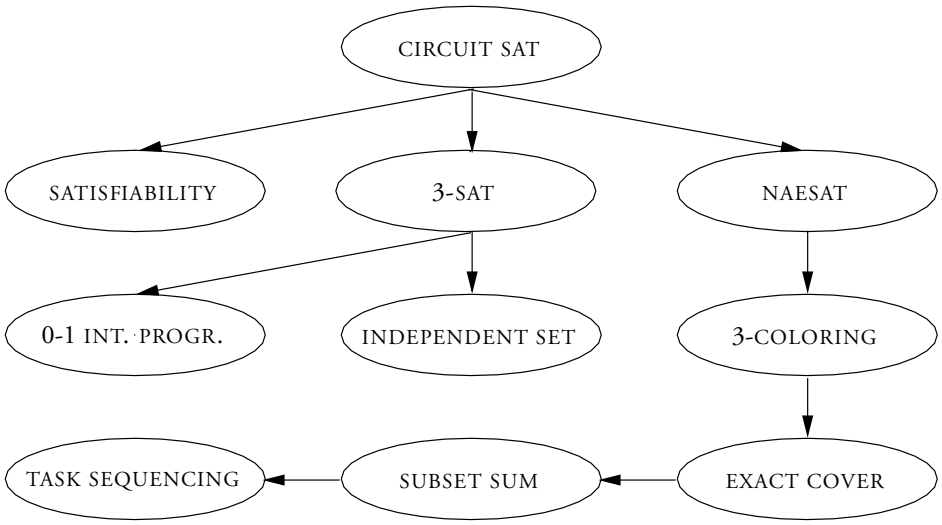


Figure 8.12 The succession of reductions used in this chapter.

distinct new variables to each clause that contains fewer than three literals can be done in log-space, this new problem, which we also call 3-SAT, is also **NP**-complete.

We now construct an instance of INDEPENDENT SET from this new version of 3-SAT in which k is equal to the number of clauses. (See Fig. 8.13.) Its graph G has one triangle for each clause and vertices carry the names of the three literals in a clause. G also has an edge between vertices carrying the labels of complementary literals.

Consider a "Yes" instance of 3-SAT. Pick one literal with value 1 from each clause. This identifies k vertices, one per triangle, and no edge exists between these vertices. Thus, the instance of INDEPENDENT SET is a "Yes" instance. Conversely, a "Yes" instance of INDEPENDENT SET on G has k vertices, one per triangle, and no two vertices carry the label of a variable and its complement because all such vertices have an edge between them. The literals associated with these independent vertices are assigned value 1, causing each clause to be satisfied. Variables not so identified are assigned arbitrary values. ■

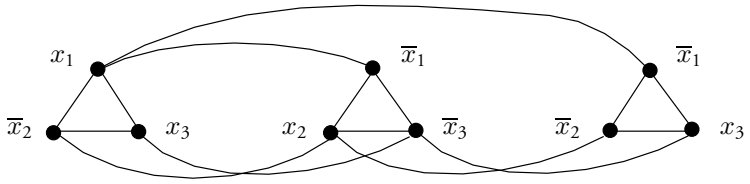


Figure 8.13 A graph for an instance of INDEPENDENT SET constructed from the following instance of 3-SAT: $(x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_3)$.

3-COLORING
Instance: The description of a graph $G = (V, E)$.
Answer: "Yes" if there is an assignment of three colors to vertices such that adjacent vertices have different colors.

THEOREM 8.10.4 3-COLORING *is* **NP***-complete.*

Proof To show that 3-COLORING is in **NP**, observe that a three-coloring of a graph can be proposed in nondeterministic polynomial time and verified in deterministic polynomial time.

We reduce NAESAT to 3-COLORING. Recall that an instance of NAESAT is an instance of 3-SAT. A "Yes" instance of NAESAT is one for which each clause is satisfiable with not all literals equal. Let an instance of NAESAT consist of m clauses $C = (c_1, c_2, \ldots, c_m)$ containing exactly three literals from the set $X = \{x_1, \overline{x}_1, x_2, \overline{x}_2, \ldots, x_n, \overline{x}_n\}$ of literals in n variables. (Use the technique introduced in the proof of Theorem 8.10.3 to insure that each clause in an instance of 3-SAT has exactly three literals per clause.)

Given an instance of NAESAT, we construct a graph G in log-space and show that this graph is three-colorable if and only if the instance of NAESAT is a "Yes" instance.

The graph G has a set of n **variable triangles**, one per variable. The vertices of the triangle associated with variable x_i are $\{\nu, x_i, \overline{x}_i\}$. (See Fig. 8.14.) Thus, all the variable triangles have one vertex in common. For each clause containing three literals we construct one **clause triangle** per clause. If clause c_j contains literals λ_{j_1}, λ_{j_2}, and λ_{j_3}, its associated clause triangle has vertices labeled (j, λ_{j_1}), (j, λ_{j_2}), and (j, λ_{j_3}). Finally, we add an edge between the vertex (j, λ_{j_k}) and the vertex associated with the literal λ_{j_k}.

We now show that an instance of NAESAT is a "Yes" instance if and only if the graph G is three-colorable. Suppose the graph is three-colorable and the colors are $\{0, 1, 2\}$. Since

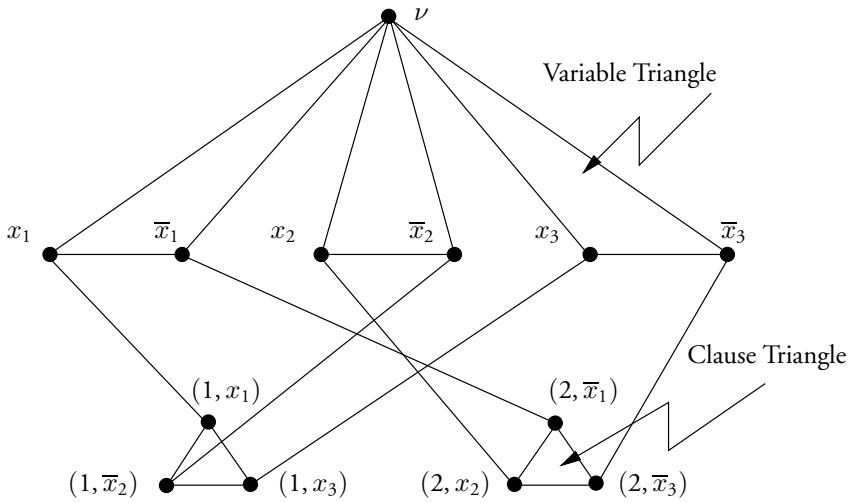


Figure 8.14 A graph G corresponding to the clauses $c_1 = \{x_1, \overline{x}_2, x_3\}$ and $c_2 = \{\overline{x}_1, x_2, \overline{x}_3\}$ in an instance of NAESAT. It has one variable triangle for each variable and one clause triangle for each clause.

three colors are needed to color the vertices of a triangle and the variable triangles have a vertex labeled ν in common, assume without loss of generality that this common vertex has color 2. The other two vertices in each variable triangle are assigned value 0 or 1, values we give to the associated variable and its complement.

Consider now the coloring of clause triangles. Since three colors are needed to color vertices of a clause triangle, consider vertices with colors 0 and 1. The edges between these clause vertices and the corresponding vertices in variable triangles have different colors at each end. Let the literals in the clause triangles be given values that are the Boolean complement of their colors. This provides values for literals that are consistent with the values of variables and insures that not all literals in a clause have the same value. The third vertex in each triangle has color 2. Give its literal a value consistent with the value of its variable. It follows that the clauses are a "Yes" instance of NAESAT.

Suppose, on the other hand, that a set of clauses is a "Yes" instance of NAESAT. We show that the graph G is three-colorable. Assign color 2 to vertex ν and colors 0 and 1 to vertices labeled x_i and $\overline{x}_i$ based on the values of these literals in the "Yes" instance. Consider two literals in clause c_j that are not both satisfied. If x_i ($\overline{x}_i$) is one of these, give the vertex labeled (j, x_i) ($(j, \overline{x}_i)$) the value that is the Boolean complement of the color of x_i ($\overline{x}_i$) in its variable triangle. Do the same for the other literal. Since the third literal has the same value as one of the other two literals (they have different values), let its vertex have color 2. Then G is three-colorable. Thus, G is a "Yes" instance of 3-COLORING if and only if the corresponding set of clauses is a "Yes" instance of NAESAT. ■

EXACT COVER
Instance: A set $S = \{u_1, u_2, \ldots, u_p\}$ and a family $\{S_1, S_2, \ldots, S_n\}$ of subsets of S.
Answer: "Yes" if there are disjoint subsets $S_{j_1}, S_{j_2}, \ldots, S_{j_t}$ such that $\cup_{1 \le i \le t} S_{j_i} = S$.

THEOREM 8.10.5 EXACT COVER *is* **NP***-complete.*

Proof It is straightforward to show that EXACT COVER is in **NP**. An NDTM can simply select the subsets and then verify in time polynomial in the length of the input that these subsets are disjoint and that they cover the set S.

We now give a log-space reduction from 3-COLORING to EXACT COVER. Given an instance of 3-COLORING, that is, a graph $G = (V, E)$, we construct an instance of EXACT COVER, namely, a set S and a family of subsets of S such that G is a "Yes" instance of 3-COLORING if and only if the family of sets is a "Yes" instance of EXACT COVER.

As the set S we choose $S = V \cup \{< e, i > \mid e \in E, 0 \le i \le 2\}$ and as the family of subsets of S we choose the sets $S_{v,i}$ and $R_{e,i}$ defined below for $v \in V$, $e \in E$ and $0 \le i \le 2$:

$$S_{v,i} = \{v\} \cup \{< e, i > \mid e \text{ is incident on } v \in V\}$$
$$R_{e,i} = \{< e, i >\}$$

Let G be three-colorable. Then let c_v, an integer in $\{0, 1, 2\}$, be the color of vertex v. We show that the subsets S_{v,c_v} for $v \in V$ and $R_{e,i}$ for $< e, i > \notin S_{v,c_v}$ for any $v \in V$ are an exact cover. If $e = (v, w) \in E$, then $c_v \neq c_w$ and S_{v,c_v} and S_{w,c_w} are disjoint. By definition the sets $R_{e,i}$ are disjoint from the other sets. Furthermore, every element of S is in one of these sets.

On the other hand, suppose that S has an exact cover. Then, for each $v \in V$, there is a unique c_v, $0 \le c_v \le 2$, such that $v \in S_{v,c_v}$. To show that G has a three-coloring, assume

that it doesn't and establish a contradiction. Since G doesn't have a three-coloring, there is an edge $e = (v, w)$ such that $c_v = c_w$, which contradicts the assumption that S has an exact cover. It follows that G has a three-coloring if and only if S has an exact cover. ■

SUBSET SUM
Instance: A set $Q = \{a_1, a_2, \ldots, a_n\}$ of positive integers and a positive integer d.
Answer: "Yes" if there is a subset of Q that adds to d.

THEOREM 8.10.6 SUBSET SUM *is* **NP***-complete.*

Proof SUBSET SUM is in **NP** because a subset can be nondeterministically chosen in time equal to n and an accepting choice verified in a polynomial number of steps by adding up the chosen elements of the subset and comparing the result to d.

To show that SUBSET SUM is **NP**-hard, we give a log-space reduction of EXACT COVER to it. Given an instance of EXACT COVER, namely, a set $S = \{u_1, u_2, \ldots, u_p\}$ and a family $\{S_1, S_2, \ldots, S_n\}$ of subsets of S, we construct the instance of SUBSET SUM characterized as follows. We let $\beta = n + 1$ and $d = \beta^{n-1} + \beta^{n-2} + \cdots + \beta^0 = (\beta^n - 1)/(\beta - 1)$. We represent the element $u_i \in S$ by the integer β^{i-1}, $1 \leq i \leq n$, and represent the set S_j by the integer a_j that is the sum of the integers associated with the elements contained in S_j. For example, if $p = n = 3$, $S_1 = \{u_1, u_3\}$, $S_2 = \{u_1, u_2\}$, and $S_3 = \{u_2\}$, we represent S_1 by $a_1 = \beta^2 + \beta^0$, S_2 by $a_2 = \beta + \beta^0$, and S_3 by $a_3 = \beta$. Since S_1 and S_3 forms an exact cover of S, $a_1 + a_3 = \beta^2 + \beta + 1 = d$.

Thus, given an instance of EXACT COVER, this polynomial-time transformation produces an instance of SUBSET SUM. We now show that the instance of the former is a "Yes" instance if and only if the instance of the latter is a "Yes" instance. To see this, observe that in adding the integers corresponding to the sets in an EXACT COVER in base β there is no carry from one power of β to the next. Thus the coefficient of β^k is exactly the number of times that u_{k+1} appears in each of the sets corresponding to a set of subsets of S. The subsets form a "Yes" instance of EXACT COVER exactly when the corresponding integers contain each power of β exactly once, that is, when the integers sum to d. ■

TASK SEQUENCING
Instance: Positive integers $t_1, t_2, \ldots, t_r$, which are **execution times**, $d_1, d_2, \ldots, d_r$, which are **deadlines**, $p_1, p_2, \ldots, p_r$, which are **penalties**, and integer $k \geq 1$.
Answer: "Yes" if there is a permutation π of $\{1, 2, \ldots, r\}$ such that

$$\left(\sum_{j=1}^{r} [\textbf{ if } t_{\pi(1)} + t_{\pi(2)} + \cdots + t_{\pi(j)} > d_{\pi(j)} \textbf{ then } p_{\pi(j)} \textbf{ else } 0] \right) \leq k$$

THEOREM 8.10.7 TASK SEQUENCING *is* **NP***-complete.*

Proof TASK SEQUENCING is in **NP** because a permutation π for a "Yes" instance can be verified as a satisfying permutation in polynomial time. We now give a log-space reduction of SUBSET SUM to TASK SEQUENCING.

An instance of SUBSET SUM is a positive integer d and a set $Q = \{a_1, a_2, \ldots, a_n\}$ of positive integers. A "Yes" instance is one such that a subset of Q adds to d. We translate an instance of SUBSET SUM to an instance of TASK SEQUENCING by setting $r = n$, $t_i = p_i = a_i$, $d_i = d$, and $k = (\sum_i a_i) - d$. Consider a "Yes" instance of this TASK

SEQUENCING problem. Then the following inequality holds:

$$\left(\sum_{j=1}^{r}[\ \textbf{if}\ a_{\pi(1)} + a_{\pi(2)} + \cdots + a_{\pi(j)} > d,\ \textbf{then}\ a_{\pi(j)}\ \textbf{else}\ 0]\right) \leq k$$

Let q be the expression in parentheses in the above inequality. Then $q = a_{\pi(l+1)} + a_{\pi(l+2)} + \cdots + a_{\pi(n)}$, where l is the integer for which $p = a_{\pi(1)} + a_{\pi(2)} + \cdots + a_{\pi(l)} \leq d$ and $p + a_{\pi(l+1)} > d$. By definition $p + q = \sum_i a_i$. It follows that $q \geq \sum_i a_i - d$. Since $q \leq k = \sum_i a_i - d$, we conclude that $p = d$ or that the instance of TASK SEQUENCING corresponds to a "Yes" instance of SUBSET SUM. Similarly, consider a "Yes" instance of SUBSET SUM. It follows from the above argument that there is a permutation such that the instance of TASK SEQUENCING is a "Yes" instance. ■

The following **NP**-complete problem is closely related to the **P**-complete problem LINEAR INEQUALITIES. The difference is that the vector $\boldsymbol{x}$ must be a 0-1 vector in the case of 0-1 INTEGER PROGRAMMING, whereas in LINEAR INEQUALITIES it can be a vector of rationals. Thus, changing merely the conditions on the vector $\boldsymbol{x}$ elevates the problem from **P** to **NP** and makes it **NP**-complete.

0-1 INTEGER PROGRAMMING
Instance: An $n \times m$ matrix A and a column n-vector $\boldsymbol{b}$, both over the ring of integers for integers n and m.
Answer: "Yes" if there is a column m-vector $\boldsymbol{x}$ over the set $\{0, 1\}$ such that $A\boldsymbol{x} = \boldsymbol{b}$.

THEOREM 8.10.8 0-1 INTEGER PROGRAMMING *is* **NP***-complete.*

Proof To show that 0-1 INTEGER PROGRAMMING is in **NP**, we note that a 0-1 vector $\boldsymbol{x}$ can be chosen nondeterministically in n steps, after which verification that it is a solution to the problem can be done in $O(n^2)$ steps on a RAM and $O(n^4)$ steps on a DTM.

To show that 0-1 INTEGER PROGRAMMING is **NP**-hard we give a log-space reduction of 3-SAT to it. Given an instance of 3-SAT, namely, a set of literals $X = (x_1, \overline{x}_1, x_2, \overline{x}_2, \ldots, x_n, \overline{x}_n)$ and a sequence of clauses $C = (c_1, c_2, \ldots, c_m)$, where each clause c_i is a subset of X containing at most three literals, we construct an $m \times p$ matrix $A = [B \mid C]$, where $B = [b_{i,j}]$ for $1 \leq i, j \leq n$ and $C = [c_{r,s}]$ for $1 \leq r \leq n$ and $1 \leq s \leq pm$. We also construct a column p-vector $\boldsymbol{d}$ as shown below, where $p = (m+1)n$. The entries of B and C are defined below.

$$b_{i,j} = \begin{cases} 1 & \text{if } x_j \in c_i \text{ for } 1 \leq j \leq n \\ -1 & \text{if } \overline{x}_j \in c_i \text{ for } 1 \leq j \leq n \end{cases}$$

$$c_{r,s} = \begin{cases} -1 & \text{if } (r-1)n + 1 \leq s \leq rn \\ 0 & \text{otherwise} \end{cases}$$

Since no one clause contains both x_j and $\overline{x}_j$, this definition of $a_{i,j}$ is consistent.

We also let d_i, the ith component of $\boldsymbol{d}$, satisfy $d_i = 1 - q_i$, where q_i is the number of complemented variables in c_i. Thus, the matrix A has the form given below, where B is an $m \times n$ matrix and each row of A contains n instances of –1 outside of B in non-overlapping

columns:

$$A = \left[\begin{array}{c|cccccccccc} & -1 & -1 & \ldots & -1 & 0 & \ldots & 0 & 0 & \ldots & 0 \\ & 0 & 0 & & 0 & -1 & \ldots & -1 & \vdots & & 0 \\ B & \vdots & & & \vdots & & & \ddots & 0 & \ldots & 0 \\ & 0 & 0 & \ldots & 0 & 0 & \ldots & 0 & -1 & \ldots & -1 \end{array}\right]$$

We show that the instance of 3-SAT is a "Yes" instance if and only if this instance of 0-1 INTEGER PROGRAMMING is a "Yes" instance, that is, if and only if $A\boldsymbol{x} = \boldsymbol{d}$.

We write the column p-vector $\boldsymbol{x}$ as the concatenation of the column m-vector $\boldsymbol{u}$ and the column mn-vector $\boldsymbol{v}$. It follows that $A\boldsymbol{x} = \boldsymbol{b}$ if and only if $A\boldsymbol{u} \geq \boldsymbol{b}$. Now consider the ith component of $A\boldsymbol{u}$. Let $\boldsymbol{u}$ select k_i uncomplemented and l_i complemented variables of clause c_i. Then, $A\boldsymbol{u} \geq \boldsymbol{b}$ if and only if $k_i - l_i \geq d_i = 1 - q_i$ or $k_i + (q_i - l_i) \geq 1$ for all i. Now let $x_i = u_i$ for $1 \leq i \leq n$. Then k_i and $q_i - l_i$ are the numbers of uncomplemented and complemented variables in c_i that are set to 1 and 0, respectively. Since $k_i + (q_i - l_i) \geq 1$, c_i is satisfied, as are all clauses, giving us the desired result. ■

8.11 The Boundary Between P and NP

It is important to understand where the boundary lies between problems in **P** and the **NP**-complete problems. While this topic is wide open, we shed a modest amount of light on it by showing that 2-SAT, the version of 3-SAT in which each clause has at most two literals, lies on the **P**-side of this boundary, as shown below. In fact, it is in **NL**, which is in **P**.

THEOREM 8.11.1 2-SAT *is in* **NL**.

Proof Given an instance I of 2-SAT, we first insure that each clause has exactly two distinct literals by adding to each one-literal clause a new literal z that is not used elsewhere. We then construct a directed graph $G = (V, E)$ with vertices V labeled by the literals x and $\overline{x}$ for each variable x appearing in I. This graph has an edge (α, β) in E directed from vertex α to vertex β if the clause $(\overline{\alpha} \vee \beta)$ is in I. If $(\overline{\alpha} \vee \beta)$ is in I, so is $(\beta \vee \overline{\alpha})$ because of commutativity of $\vee$. Thus, if $(\alpha, \beta) \in E$, then $(\overline{\beta}, \overline{\alpha}) \in E$ also. (See Fig. 8.15.) Note that $(\alpha, \beta) \neq (\overline{\beta}, \overline{\alpha})$ because this requires that $\beta = \overline{\alpha}$, which is not allowed. Let $\alpha \neq \overline{\gamma}$. It follows that if there is a path from α to γ in G, there is a distinct path from $\overline{\gamma}$ to $\overline{\alpha}$ obtained by reversing the directions of each edge on the path and replacing the literals by their complements.

To understand why these edges are chosen, note that if all clauses of I are satisfied and $(\overline{\alpha} \vee \beta)$ is in I, then $\alpha = 1$ implies that $\beta = 1$. This implication relation, denoted $\alpha \Rightarrow \beta$, is transitive. If there is a path $(\alpha_1, \alpha_2, \ldots, \alpha_k)$ in G, then there are clauses $(\overline{\alpha}_1 \vee \alpha_2)$, $(\overline{\alpha}_2 \vee \alpha_3), \ldots, (\overline{\alpha}_{k-1} \vee \alpha_k)$ in I. If all clauses are satisfied and if the literal $\alpha_1 = 1$, then each un-negated literal on this path must have value 1.

We now show that an instance I is a "No" instance if and only if there is a variable x such that there is a path in G from x to $\overline{x}$ and one from $\overline{x}$ to x.

If there is a variable x such that such paths exists, this means that $x \Rightarrow \overline{x}$ and $\overline{x} \Rightarrow x$ which is a logical contradiction. This implies that the instance I is a "No" instance.

Conversely, suppose I is a "No" instance. To prove there is a variable x such that there are paths from vertex x to vertex $\overline{x}$ and from $\overline{x}$ to x, assume that for no variable x does this

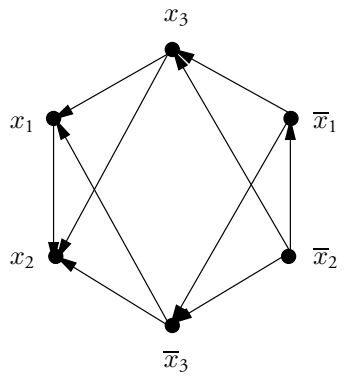


Figure 8.15 A graph capturing the implications associated with the following satisfiable instance of 2-SAT: $(x_3 \vee x_2) \wedge (x_3 \vee x_1) \wedge (\overline{x}_3 \vee x_2) \wedge (\overline{x}_1 \vee x_2) \wedge (\overline{x}_3 \vee x_1)$.

condition hold and show that I is a "Yes" instance, that is, every clause is satisfied, which contradicts the assumption that I is a "No" instance.

Identify a variable that has not been assigned a value and let α be one of the two corresponding literals such that there is no directed path in G from the vertex α to $\overline{\alpha}$. (By assumption, this must hold for at least one of the two literals associated with x.) Assign value 1 to α and each literal λ reachable from it. (This assigns values to the variables identified by these literals.) If these assignments can be made without assigning a variable both values 0 and 1, each clause can be satisfied and I is "Yes" instance rather than a "No" one, as assumed. To show that each variable is assigned a single value, we assume the converse and show that the conditions under which values are assigned to variables by this procedure are contradicted. A variable can be assigned contradictory values in two ways: a) on the current step the literals λ and $\overline{\lambda}$ are both reachable from α and assigned value 1, and b) a literal λ is reachable from α on the current step that was assigned value 0 on a previous step. For the first case to happen, there must be a path from α to vertices λ and $\overline{\lambda}$. By design of the graph, if there is a path from α to $\overline{\lambda}$, there is a path from λ to $\overline{\alpha}$. Since there is a path from α to λ, there must be a path from α to $\overline{\alpha}$, contradicting the assumption that there are no such paths. In the second case, let a λ be assigned 1 on the current step that was assigned 0 on a previous step. It follows that $\overline{\lambda}$ was given value 1 on that step. Because there is a path from α to λ, there is one from $\overline{\lambda}$ to $\overline{\alpha}$ and our procedure, which assigned $\overline{\lambda}$ value 1 on the earlier step, must have assigned $\overline{\alpha}$ value 1 on that step also. Thus, α had the value 0 before the current step, contradicting the assumption that it was not assigned a value.

To show that 2-SAT is in **NL**, recall that **NL** is closed under complements. Thus, it suffices to show that "No" instances of 2-SAT can be accepted in nondeterministic logarithmic space. By the above argument, if I is a "No" instance, there is a variable x such that there is a path in G from x to $\overline{x}$ and from $\overline{x}$ to x. Since the number of vertices in G is at most linear in n, the length of I (it may be as small as $O(\sqrt{n})$), an NDTM can propose and then verify in space $O(\log n)$ a path in G from x to $\overline{x}$ and back by checking that the putative edges are edges of G, that x is the first and last vertex on the path, and that $\overline{x}$ is encountered before the end of the path. ■

8.12 PSPACE-Complete Problems

PSPACE is the class of decision problems that are decidable by a Turing machine in space polynomial in the length of the input. Problems in **PSPACE** are potentially much more complex than problems in **P**.

The hardest problems in **PSPACE** are the **PSPACE**-complete problems. (See Section 8.8.) Such problems have two properties: a) they are in **PSPACE** and b) every problem in **PSPACE** can be reduced to them by a polynomial-time Turing machine. The **PSPACE**-complete problems are the hardest problems in **PSPACE** in the sense that if they are in **P**, then so are all problems in **PSPACE**, an unlikely prospect.

We now establish that QUANTIFIED SATISFIABILITY defined below is **PSPACE**-complete. We also show that GENERALIZED GEOGRAPHY, a game played on a graph, is **PSPACE**-complete by reducing QUANTIFIED SATISFIABILITY to it. A characteristic shared by many important **PSPACE**-complete problems and these two problems is that they are equivalent to games on graphs.

8.12.1 A First PSPACE-Complete Problem

Quantified Boolean formulas use existential quantification, denoted $\exists$, and universal quantification, denoted $\forall$. **Existential quantification** on variable x_1, denoted $\exists x_1$, means "there exists a value for the Boolean variable x_1," whereas **universal quantification** on variable x_2, denoted $\forall x_2$, means "for all values of the Boolean variable x_2." Given a Boolean formula such as $(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$, a quantification of it is a collection of universal or existential quantifiers, one per variable in the formula, followed by the formula. For example,

$$\forall x_1 \exists x_2 \forall x_3 [(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)]$$

is a quantified formula. Its meaning is "for all values of x_1, does there exist a value for x_2 such that for all values of x_3 the formula $(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$ is satisfied?" In this case the answer is "No" because for $x_1 = 1$, the function is not satisfied with $x_3 = 0$ when $x_2 = 0$ and is not satisfied with $x_3 = 1$ when $x_2 = 1$. However, if the third quantifier is changed from universal to existential, then the quantified formula is satisfied. Note that the order of the quantifiers is important. To see this, observe that under the quantification order $\forall x_1 \forall x_3 \exists x_2$ that the quantified formula is satisfied.

QUANTIFIED SATISFIABILITY consists of satisfiable instances of quantified Boolean formulas in which each formula is expressed as a set of clauses.

QUANTIFIED SATISFIABILITY
Instance: A set of literals $X = \{x_1, \overline{x}_1, x_2, \overline{x}_2, \ldots, x_n, \overline{x}_n\}$, a sequence of clauses $C = (c_1, c_2, \ldots, c_m)$, where each clause c_i is a subset of X, and a sequence of quantifiers $(Q_1, Q_2, \ldots, Q_n)$, where $Q_j \in \{\forall, \exists\}$.
Answer: "Yes" if under the quantifiers $Q_1 x_1 Q_2 x_2 \cdots Q_n x_n$, the clauses $c_1, c_2, \ldots, c_m$ are satisfied, denoted

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \ [\phi]$$

where the formula $\phi = c_1 \wedge c_2 \wedge \cdots \wedge c_m$ is in the product-of-sums form. (See Section 2.2.)

In this section we establish the following result, stronger than **PSPACE**-completeness of QUANTIFIED SATISFIABILITY: we show it is complete for **PSPACE** under log-space transformations. Reductions of this type are potentially stronger than polynomial-time reductions because the transformation is executed in logarithmic space, not polynomial time. While it is true that every log-space transformation is a polynomial-time transformation (see Theorem 8.5.8), it is not known if the reverse is true. We prove this result in two stages: we first show that QUANTIFIED SATISFIABILITY is in **PSPACE** and then that it is hard for **PSPACE**.

LEMMA 8.12.1 QUANTIFIED SATISFIABILITY *is in* **PSPACE**.

Proof To show that QUANTIFIED SATISFIABILITY is in **PSPACE** we evaluate in polynomial space a circuit, C_{qsat}, whose value is 1 if and only if the instance of QUANTIFIED SATISFIABILITY is a "Yes" instance. The circuit C_{qsat} is a tree all of whose paths from the inputs to the output (root of the tree) have the same length, each vertex is either an AND gate or an OR gate, and each input has value 0 or 1. (See Fig. 8.16.) The gate at the root of the tree is associated with the variable x_1, the gates at the next level are associated with x_2, etc. The type of gate at the jth level is determined by the jth quantifier Q_j and is AND if $Q_j = \forall$ and OR if $Q_j = \exists$. The leaves correspond to all 2^n the values of the n variables: at each level of the tree the left and right branches correspond to the values 0 and 1 for the corresponding quantified variable. Each leaf of the tree contains the value of the formula ϕ for the values of the variables leading to that leaf. In the example of Fig. 8.16 the leftmost leaf has value 1 because on input $x_1 = x_2 = x_3 = 0$ each of the three clauses $\{x_1, x_2, \overline{x}_3\}$, $\{\overline{x}_1, x_2, x_3\}$ and $\{\overline{x}_1, \overline{x}_2, \overline{x}_3\}$ is satisfied.

It is straightforward to see that the value at the root of the tree is 1 if all clauses are satisfied under the quantifiers $Q_1 x_1 Q_2 x_2 \cdots Q_n x_n$ and 0 otherwise. Thus, the circuit solves the QUANTIFIED SATISFIABILITY problem and its complement. (Note that **PSPACE** = **coPSPACE**, as shown in Theorem 8.6.1.)

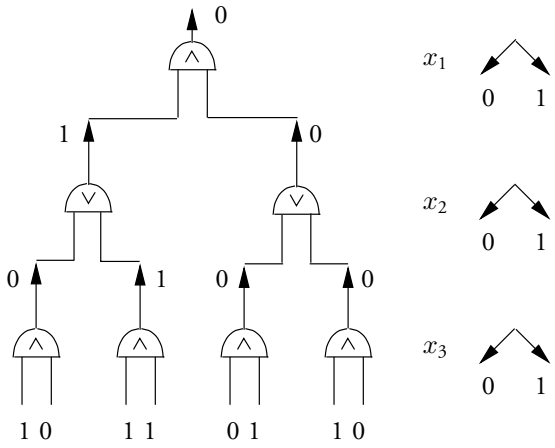


Figure 8.16 A tree circuit constructed from the instance $\forall x_1 \exists x_2 \forall x_3 \phi$ for $\phi = (x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$ of QUANTIFIED SATISFIABILITY. The eight values at the leaves of the tree are the values of ϕ on the eight different assignments to (x_1, x_2, x_3).

```
tree_eval(n, φ, Q, d, w);
  if d = n then
    return(evaluate(φ, w));
  else
    if first(Q) = ∃ then
      return(tree_eval(n, φ, rest(Q), d + 1, w0)
        OR tree_eval(n, φ, rest(Q), d + 1, w1));
    else
      return(tree_eval(n, φ, rest(Q), d + 1, w0)
        AND tree_eval(n, φ, rest(Q), d + 1, w1));
```

Figure 8.17 A program for the recursive procedure `tree_eval`$(n, \phi, \boldsymbol{Q}, d, \boldsymbol{w})$. The tuple $\boldsymbol{w}$ keeps track of the path taken into the tree.

The circuit C_{qsat} has size exponential in n because there are 2^n values for the n variables. However, it can be evaluated in polynomial space, as we show. For this purpose consider the recursive procedure `tree_eval`$(n, \phi, \boldsymbol{Q}, d, \boldsymbol{w})$ in Fig. 8.17 that evaluates C_{qsat}. Here n is the number of variables in the quantization, d is the depth of recursion, ϕ is the expression over which quantification is done, $\boldsymbol{Q}$ is a sequence of quantifiers, and $\boldsymbol{w}$ holds the values for d variables. Also, `first(`$\boldsymbol{Q}$`)` and `rest(`$\boldsymbol{Q}$`)` are the first and all but the first components of $\boldsymbol{Q}$, respectively. When $d = 0$, $\boldsymbol{Q} = (Q_1, Q_2, \ldots, Q_n)$ and $Q_1x_1Q_2x_2 \cdots Q_nx_n\ \phi$ is the expression to evaluate. We show that `tree_eval`$(n, \phi, \boldsymbol{Q}, 0, \epsilon)$ can be computed in space quadratic in the length of an instance of QUANTIFIED SATISFIABILITY.

When $d = n$, the procedure has reached a leaf of the tree and the string $\boldsymbol{w}$ contains values for the variables $x_1, x_2, \ldots, x_n$, in that order. Since all variables of ϕ are known when $d = n$, ϕ can be evaluated. Let `evaluate`$(\phi, \boldsymbol{w})$ be the function that evaluates ϕ with values specified by $\boldsymbol{w}$. Clearly `tree_eval`$(n, \phi, \boldsymbol{Q}, 0, \epsilon)$ is the value of $Q_1x_1Q_2x_2 \cdots Q_nx_n\ \phi$.

We now determine the work space needed to compute `tree_eval`$(n, \phi, \boldsymbol{Q}, d, \boldsymbol{w})$ on a DTM. (The discussion in the proof of Theorem 8.5.5 is relevant.) Evaluation of this procedure amounts to a depth-first traversal of the tree. An activation record is created for each call to the procedure and is pushed onto a stack. Since the depth of the tree is n, at most $n + 1$ records will be on the stack. Since each activation record contains a string of length at most $O(n)$, the total space used is $O(n^2)$. And the length of $Q_1x_1Q_2x_2 \cdots Q_nx_n\ \phi$ is at least n, the space is polynomial in the length of this formula. ■

LEMMA 8.12.2 QUANTIFIED SATISFIABILITY *is log-space hard for* **PSPACE**.

Proof Our goal is to show that every decision problem $\mathcal{P} \in$ **PSPACE** can be reduced in log-space to an instance of QUANTIFIED SATISFIABILITY. Instead, we show that every such $\mathcal{P}$ can be reduced in log-space to a "No" instance of QUANTIFIED SATISFIABILITY (we call this QUANTIFIED UNSATISFIABILITY). But a "No" instance is one for which the formula ϕ, which is in product-of-sums form, is not satisfied under the specified quantification or that its Boolean complement, which is in sum-of-products expansion (SOPE) form, is satisfied under a quantification in which $\forall$ is replaced by $\exists$ and vice versa. Exchanging "Yes" and "No" instances of decision problems (which we can do since **PSPACE** is closed un-

der complements), we have that every problem in **coPSPACE** can be reduced in log-space to QUANTIFIED SATISFIABILITY. However, since **PSPACE** = **coPSPACE**, we have the desired result.

Our task now is to show that every problem $\mathcal{P} \in$ **PSPACE** can be reduced in log-space to an instance of QUANTIFIED UNSATISFIABILITY. Let $L \in$ **PSPACE** be the language of "Yes" instances of $\mathcal{P}$ and let M be the DTM deciding L. Instances of QUANTIFIED UNSATISFIABILITY will be quantified formulas in SOPE form that describe conditions on the configuration graph $G(M, \boldsymbol{w})$ of M on input $\boldsymbol{w}$. We associate a Boolean vector with each vertex in $G(M, \boldsymbol{w})$ and assume that $G(M, \boldsymbol{w})$ has one initial and final vertex associated with the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, respectively. (We can make the last assumption because M can be designed to enter a cleanup phase in which it prints blanks in all non-blank tape cells.)

Let $\boldsymbol{c}$ and $\boldsymbol{d}$ be vector encodings of arbitrary configurations c and d of $G(M, \boldsymbol{w})$. We construct formulas $\psi_i(\boldsymbol{c}, \boldsymbol{d})$, $0 \leq i \leq k$, in SOPE form that are satisfied if and only if there exists a path from c to d in $G(M, \boldsymbol{w})$ of length at most 2^i (it computes the predicate PATH$(c, d, 2^i)$ introduced in the proof of Theorem 8.5.5). Then a "Yes" instance of QUANTIFIED UNSATISFIABILITY is the formula $\psi_k(\boldsymbol{a}, \boldsymbol{b})$, where $\boldsymbol{a}$ and $\boldsymbol{b}$ are encodings of the initial and final vertices of $G(M, \boldsymbol{w})$ for k sufficiently large that a polynomial-space computation can be done in time 2^k. Since, as seen in Theorem 8.5.6, a deterministic computation in space S is done in time $O(2^S)$, it suffices for k to be polynomial in the length of the input.

The formula $\psi_0(\boldsymbol{c}, \boldsymbol{d})$ is satisfiable if either $\boldsymbol{c} = \boldsymbol{d}$ or $\boldsymbol{d}$ follows from $\boldsymbol{c}$ in one step. Such formulas are easily computed from the descriptions of M and $\boldsymbol{w}$. $\psi_i(\boldsymbol{c}, \boldsymbol{d})$ can be expressed as shown below, where the existential quantification is over all possible intermediate configurations $\boldsymbol{e}$ of M. (See the proof of Theorem 8.5.5 for the representation of PATH$(c, d, 2^i)$ in terms of PATH$(c, e, 2^{i-1})$ and PATH$(e, d, 2^{i-1})$.)

$$\psi_i(\boldsymbol{c}, \boldsymbol{d}) = \exists \boldsymbol{e} \, [\psi_{i-1}(\boldsymbol{c}, \boldsymbol{e}) \wedge \psi_{i-1}(\boldsymbol{e}, \boldsymbol{d})] \tag{8.1}$$

Note that $\exists \boldsymbol{e}$ is equivalent to $\exists e_1 \exists e_2 \cdots \exists e_q$, where q is the length of $\boldsymbol{e}$. Universal quantification over a vector is expanded in a similar fashion.

Unfortunately, for $i = k$ this recursively defined formula requires space exponential in the size of the input. Fortunately, we can represent $\psi_i(\boldsymbol{c}, \boldsymbol{d})$ more succinctly using the implication operator $x \Rightarrow y$, as shown below, where $x \Rightarrow y$ is equivalent to $\overline{x} \vee y$. Note that if $x \Rightarrow y$ is TRUE, then either x is FALSE or x and y are both TRUE.

$$\psi_i(\boldsymbol{c}, \boldsymbol{d}) = \exists \boldsymbol{e} \, [\forall \boldsymbol{x} \forall \boldsymbol{y} \;\; [(\boldsymbol{x} = \boldsymbol{c} \wedge \boldsymbol{y} = \boldsymbol{e}) \vee (\boldsymbol{x} = \boldsymbol{e} \wedge \boldsymbol{y} = \boldsymbol{d})] \Rightarrow \psi_{i-1}(\boldsymbol{x}, \boldsymbol{y})] \tag{8.2}$$

Here $\boldsymbol{x} = \boldsymbol{y}$ denotes $(x_1 = y_1) \wedge (x_2 = y_2) \wedge \cdots \wedge (x_q = y_q)$, where $(x_i = y_i)$ denotes $x_i y_i \vee \overline{x}_i \overline{y}_i$. Then, the formula in the outer square brackets of (8.2) is true when either $(\boldsymbol{x} = \boldsymbol{c} \wedge \boldsymbol{y} = \boldsymbol{e}) \vee (\boldsymbol{x} = \boldsymbol{e} \wedge \boldsymbol{y} = \boldsymbol{d})$ is FALSE or this expression is TRUE and $\psi_{i-1}(\boldsymbol{x}, \boldsymbol{y})$ is also TRUE. Because the contents of the outer square brackets are TRUE, the quantization on $\boldsymbol{x}$ and $\boldsymbol{y}$ requires that $\psi_{i-1}(\boldsymbol{c}, \boldsymbol{e})$ and $\psi_{i-1}(\boldsymbol{e}, \boldsymbol{d})$ both be TRUE or that the formula given in (8.1) be satisfied.

It remains to convert the expression for $\psi_i(\boldsymbol{c}, \boldsymbol{d})$ given above to SOPE form in log-space. But this is straightforward. We replace $g \Rightarrow h$ by $\overline{g} \vee h$, where $g = (r \wedge s) \vee (t \wedge u)$ and $r = (\boldsymbol{x} = \boldsymbol{c})$, $s = (\boldsymbol{y} = \boldsymbol{e})$, $t = (\boldsymbol{x} = \boldsymbol{e})$, and $u = (\boldsymbol{y} = \boldsymbol{d})$. It follows that

$$\begin{aligned} \overline{g} &= (\overline{r} \vee \overline{s}) \wedge (\overline{t} \vee \overline{u}) \\ &= (\overline{r} \wedge \overline{t}) \vee (\overline{r} \wedge \overline{u}) \vee (\overline{s} \wedge \overline{t}) \vee (\overline{s} \wedge \overline{u}) \end{aligned}$$

Since each of r, s, t, and u can be expressed as a conjunction of q terms of the form $(x_j = y_j)$ and $\overline{(x_j = y_j)} = (\overline{x}_j y_j \vee x_j \overline{y}_j)$, $1 \leq i \leq q$, it follows that $\overline{r}$, $\overline{s}$, $\overline{t}$, and $\overline{u}$ can each be expressed as a disjunction of $2q$ terms. Each of the four terms of the form $(\overline{r} \wedge \overline{t})$ consists of $4q^2$ terms, each of which is a conjunction of four literals. Thus, $\overline{g}$ is the disjunction of $16q^2$ terms of four literals each.

Given the regular structure of this formula for ψ_i, it can be generated from a formula for ψ_{i-1} in space $O(\log q)$. Since $0 \leq i \leq k$ and k is polynomial in the length of the input, all the formulas, including that for ψ_k, can be generated in log-space. By the above reasoning, this formula is a "Yes" instance of QUANTIFIED UNSATISFIABILITY if and only if there is a path in the configuration graph $G(M, \boldsymbol{w})$ between the initial and final states. ■

Combining the two results, we have the following theorem.

THEOREM 8.12.1 QUANTIFIED SATISFIABILITY *is log-space complete for* **PSPACE**.

8.12.2 Other **PSPACE**-Complete Problems

An important version of QUANTIFIED SATISFIABILITY is ALTERNATING QUANTIFIED SATISFIABILITY.

ALTERNATING QUANTIFIED SATISFIABILITY
Instance: Instances of QUANTIFIED SATISFIABILITY that have an even number of quantifiers that alternate between $\exists$ and $\forall$, with $\exists$ the first quantifier.
Answer: "Yes" if the instance is a "Yes" instance of QUANTIFIED SATISFIABILITY.

THEOREM 8.12.2 ALTERNATING QUANTIFIED SATISFIABILITY *is log-space complete for* **PSPACE**.

Proof ALTERNATING QUANTIFIED SATISFIABILITY is in **PSPACE** because it is a special case of QUANTIFIED SATISFIABILITY. We reduce QUANTIFIED SATISFIABILITY to ALTERNATING QUANTIFIED SATISFIABILITY in log-space as follows. If two universal quantifiers appear in succession, we add an existential quantifier between them in a new variable, say x_l, and add the new clause $\{x_l, \overline{x}_l\}$ at the end of the formula ϕ. If two existential quantifiers appear in succession, add universal quantification over a new variable and a clause containing it and its negation. If the number of quantifiers is not even, repeat one or the other of the above steps. This transformation at most doubles the number of variables and clauses and can be done in log-space. The instance of ALTERNATING QUANTIFIED SATISFIABILITY is a "Yes" instance if and only if the instance of QUANTIFIED SATISFIABILITY is a "Yes" instance. ■

The new version of QUANTIFIED SATISFIABILITY is akin to a game in which universal and existential players alternate. The universal player attempts to show a fact for all values of its Boolean variable, whereas the existential player attempts to deny that fact by the choice of its existential variable. It is not surprising, therefore, that many games are **PSPACE**-complete. The geography game described below is of this type.

The **geography game** is a game for two players. They alternate choosing names of cities in which the first letter of the next city is the last letter of the previous city until one of the two players (the losing player) cannot find a name that has not already been used. (See Fig. 8.18.) This game is modeled by a graph in which each vertex carries the name of a city and there is

an edge from vertex u_1 to vertex u_2 if the last letter in the name associated with u_1 is the first letter in the name associated with u_2. In general this graph is directed because an edge from u_1 to u_2 does not guarantee an edge from u_2 to u_1.

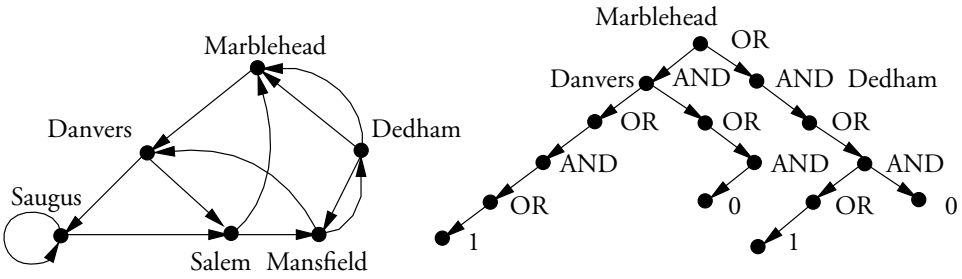
GENERALIZED GEOGRAPHY
Instance: A directed graph $G = (V, E)$ and a vertex v.
Answer: "Yes" if there is a sequence of (at most $|V|$) alternating vertex selections by two players such that vertex v is the first selection by the first player and for each selection of the first player and all selections of the second player of vertices adjacent to the previous selection, the second player arrives at a vertex from which it cannot select a vertex not previously selected.

THEOREM 8.12.3 GENERALIZED GEOGRAPHY *is log-space complete for* **PSPACE**.

Proof To show that GENERALIZED GEOGRAPHY is log-space complete for **PSPACE**, we show that it is in **PSPACE** and that QUANTIFIED SATISFIABILITY can be reduced to it in log-space. To establish the first result, we show that the outcome of GENERALIZED GEOGRAPHY can be determined by evaluating a graph similar to the binary tree used to show that QUANTIFIED SATISFIABILITY is realizable in **PSPACE**.

Given the graph $G = (V, E)$ (see Fig. 8.18(a)), we construct a search graph (see Fig. 8.18(b)) by performing a variant of depth-first search of G from v. At each vertex we visit the next unvisited descendant, continuing until we encounter a vertex on the current path, at which point we backtrack and try the next sibling of the current vertex, if any. In depth-first search if a vertex has been visited previously, it is not visited again. In this variant of the algorithm, however, a vertex is revisited if it is not on the current path. The length of the longest path in this tree is at most $|V| - 1$ because each path can contain no more than $|V|$ vertices. The tree may have a number of vertices exponential in $|V|$.

At a leaf vertex a player has no further moves. The first player wins if it is the second player's turn at a leaf vertex and loses otherwise. Thus, a leaf vertex is labeled 1 (0) if the first player wins (loses). To insure that the value at a vertex u is 1 if the two players reach u and the first player wins, we assign OR operators to vertices at which the first player makes selections and AND operators otherwise. (The output of a one-input AND or OR gate is the



(a) (b)

Figure 8.18 (a) A graph for the generalized geography game and (b) the search tree associated with the game in which the start vertex is Marblehead.

value of its input.) This provides a circuit that can be evaluated just as was the circuit C_{qsat} used in the proof of Theorem 8.12.1. The "Yes" instances of GENERALIZED GEOGRAPHY are such that the first player can win by choosing a first city. In Fig. 8.18 the value of the root vertex is 0, which means that the first player loses by choosing to start with Marblehead as the first city.

Vertices labeled AND or OR in the tree generated by depth-first search can have arbitrary in-degree because the number of vertices that can be reached from a vertex in the original graph is not restricted. The procedure `tree_eval` described in the proof of Theorem 8.12.1 can be modified to apply to the evaluation of this DAG whose vertex in-degree is potentially unbounded. (See Problem 8.30.) This modified procedure runs in space polynomial in the size of the graph G.

We now show that ALTERNATING QUANTIFIED SATISFIABILITY (abbreviated AQSAT) can be reduced in log-space to GENERALIZED GEOGRAPHY. Given an instance of AQSAT such as that shown below, we construct an instance of GENERALIZED GEOGRAPHY, as shown in Fig. 8.19. We assume without loss of generality that the number of quantifiers is even. If not, add a dummy variable and quantify on it:

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 [(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (x_4 \vee \overline{x}_4)]$$

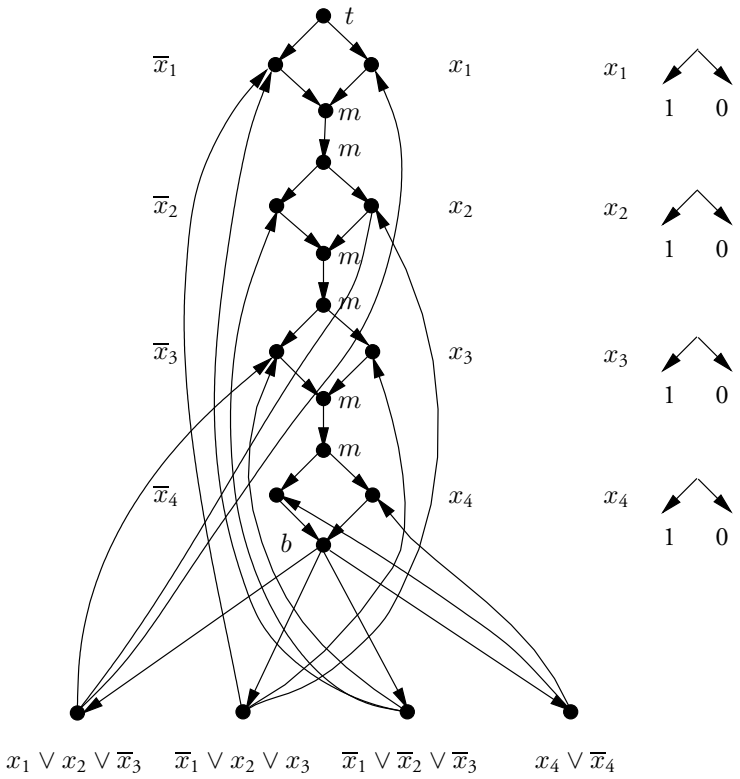


Figure 8.19 An instance of GENERALIZED GEOGRAPHY corresponding to an instance of ALTERNATING QUANTIFIED SATISFIABILITY.

The instance of GENERALIZED GEOGRAPHY corresponding to an instance of AQSAT is formed by cascading a set of diamond-shaped subgraphs, one per variable (see Fig. 8.19), and connecting the bottom vertex b in the last diamond to a set of vertices, one per clause. An edge is drawn from a clause to a vertex associated with a literal (x_i or $\overline{x}_i$) if that literal is in the clause. The literal x_i ($\overline{x}_i$) is associated with the middle vertex on the right-hand (left-hand) side of a diamond. Thus, in the example, there is an edge from the leftmost clause vertex to the left-hand vertex in the diamond for x_3 and to the right-hand vertices in diamonds for x_1 and x_2.

Let the geography game be played on this graph starting with the first player from the topmost vertex labeled t. The first player can choose either the left or right path. The second player has only one choice, taking it to the bottom of the first diamond, and the first player now has only one choice, taking it to the top of the second diamond. The second player now can choose a path to follow. Continuing in this fashion, we see that the first (second) player can exercise a choice on the odd- (even-) numbered diamonds counting from the top. Since the number of quantifiers is even, the choice at the bottom vertex labeled b belongs to the second player. Observe that whatever choices are made within the diamonds, the vertices labeled m and b are visited.

Because the goal of each player is to force the other player into a position from which it has no moves, at vertex b the second player attempts to choose a clause vertex such that the first player has no moves: that is, every vertex reachable from the clause vertex chosen by the second player has already been visited. On the other hand, if all clauses are satisfiable, then for every clause chosen by the second player there should be an edge from its vertex to a diamond vertex that has not been previously visited. To insure that the first player wins if and only if the instance of AQSAT used to construct this graph is a "Yes" instance, the first player always chooses an edge according to the directions in Fig. 8.19. For example, it visits the vertex labeled $\overline{x}_1$ if it wishes to set $x_1 = 1$ because this means that the vertex labeled x_1 is not visited on the path from t to b and can be visited by the first player on the last step of the game. Since each vertex labeled m and b is visited before a clause vertex is visited, the second player does not have a move and loses. ■

8.13 The Circuit Model of Computation

The complexity classes seen so far in this chapter are defined in terms of the space and time needed to recognize languages with deterministic and nondeterministic Turing machines. These classes generally help us to understand the complexity of serial computation. Circuit complexity classes, studied in this section, help us to understand parallel computation.

Since a circuit is a fixed interconnection of gates, each circuit computes a single Boolean function on a fixed number of inputs. Thus, to compute the unbounded set of functions computed by a Turing machine, a family of circuits is needed. In this section we investigate uniform and non-uniform circuit families. A uniform family of circuits is a potentially unbounded set of circuits for which there is a Turing machine that, given an integer n in unary notation, writes a description of the nth circuit. We show that uniform circuits compute the same functions as Turing machines.

As mentioned below, non-uniform families of circuits are so powerful that they can compute functions not computed by Turing machines. Given the Church-Turing thesis, it doesn't make sense to assume non-uniform circuits as a model of computation. On the other hand, if

we can develop large lower bounds on the size or depth of circuits without regard to whether or not they are drawn from a uniform family, then such lower bounds apply to uniform families as well and, in particular, to other models of computation, such as Turing machines. For this reason non-uniform circuits are important.

A circuit is a form of unstructured parallel machine, since its gates can operate in parallel. The parallel random-access machine (PRAM) introduced in Chapter 1 and examined in Chapter 7 is another important parallel model of computation in terms of which the performance of many other parallel computational models can be measured. In Section 8.14 we show that circuit size and depth are related to number of processors and time on the PRAM. These results emphasize the important role of circuits not only in the construction of machines, but also in measuring the serial and parallel complexity of computational problems.

Throughout the following sections we assume that circuits are constructed from gates chosen from the **standard basis** $\Omega_0 = \{\text{AND}, \text{OR}, \text{NOT}\}$.

We now explore uniform and non-uniform circuit families, thereby setting the stage for the next chapter, in which methods for deriving lower bounds on the size of circuits are developed. After introducing uniform circuits we show that uniform families of circuits and Turing machines compute the same functions. We then introduce a number of languages defined in terms of the properties of families of circuits that recognize them.

8.13.1 Uniform Families of Circuits

Families of circuits are useful in characterizing decision problems in which the set of instances is unbounded. One circuit in each family is associated with the "Yes" instances of each length: it has value 1 on the "Yes" instances and value 0 otherwise.

Families of circuits are designed in Chapter 3 to simulate computations by finite-state, random-access, and Turing machines on arbitrary numbers of inputs. For each machine M of one of these types, there is a DTM $S(M)$ such that on an input of length n, $S(M)$ can produce as output the description of a circuit on n inputs that computes exactly the same function as does M on n inputs. (See the program in Fig. 3.27.) These circuits are generated in a uniform fashion.

On the other hand, non-uniform circuit families can be used to define non-computable languages. For example, consider the family in which the nth circuit, C_n, is designed to have value 1 on those strings $\boldsymbol{w}$ of length n in the language $\mathcal{L}_1$ defined in Section 5.7 and value 0 otherwise. Such a circuit realizes the minterm defined by $\boldsymbol{w}$. As shown in Theorem 5.7.4, $\mathcal{L}_1$ is not recursively enumerable; that is, there is no Turing machine that can recognize it.

This example motivates the need to identify families of circuits that compute functions computable by Turing machines, that is, uniform families of circuits.

DEFINITION 8.13.1 *A* **circuit family** $\mathcal{C} = \{C_1, C_2, C_3, \ldots\}$ *is a collection of logic circuits in which* C_n *has* n *inputs and* $m(n)$ *outputs for some function* $m : \mathbb{N} \mapsto \mathbb{N}$.

A time-$r(n)$ *(space-*$r(n)$*)* **uniform circuit family** *is a circuit family for which there is a deterministic Turing machine* M *such that for each integer* n *supplied in unary notation, namely* 1^n*, on its input tape,* M *writes the description of* C_n *on its output tape using time (space)* $r(n)$.

A **log-space uniform circuit family** *is one for which the temporary storage space used by a Turing machine that generates it is* $O(\log n)$*, where* n *is the length of the input. The* **function** $f : \mathcal{B}^* \mapsto \mathcal{B}^*$ **is computed by** $\mathcal{C}$ *if for each* $n \geq 1$*,* f *restricted to* n *inputs is the function computed by* C_n.

8.13.2 Uniform Circuits Are Equivalent to Turing Machines

We now show that the functions computed by log-space uniform families of circuits and by polynomial-time DTMs are the same. Since the family of functions computed by one-tape and multi-tape Turing machines are the same (see Theorem 5.2.1), we prove the result only for the standard one-tape Turing machine and proper resource functions (see Section 8.3).

THEOREM 8.13.1 *Let $p(n)$ be a polynomial and a proper function. Then every total function $f : \mathcal{B}^* \mapsto \mathcal{B}^*$ computed by a DTM in time $p(n)$ on inputs of length n can be computed by a log-space uniform circuit family $\mathcal{C}$.*

Proof Let $f_n : \mathcal{B}^n \mapsto \mathcal{B}^*$ be the restriction to inputs of length n of the function $f : \mathcal{B}^* \mapsto \mathcal{B}^*$ computed by a DTM M in time $p(n)$. It follows that the number of bits in the word $f_n(\boldsymbol{w})$ is at most $p(n)$. Since the function computed by a circuit has a fixed-length output and the length of $f_n(\boldsymbol{w})$ may vary for different inputs $\boldsymbol{w}$ of length n, we show how to create a DTM M^*, a modified version of M, that computes f_n^*, a function that contains all the information in the function f_n. The value of f_n^* has at most $2p(n)$ bits on inputs of length n. We show that M^* produces its output in time $O(p^2(n))$.

Let M^* place a mark in the $2p(n)$th cell on its tape (a cell beyond any reached during a computation). Let it now simulate M, which is assumed to print its output in the first k locations on the tape, $k \leq p(n)$. M^* now recodes and expands this binary string into a longer string. It does so by marking k cells to right of the output string (in at most k^2 steps), after which it writes every letter in the output string twice. That is, 0 appears as 00 and 1 as 11. Finally, the remaining $2(p(n) - k)$ cells are filled with alternating 0s and 1s. Clearly, the value of f_n can be readily deduced from the output, but the length of the value f_n^* is the same on all inputs of length n.

A Turing machine $M_{\mathcal{C}}$ that constructs the nth circuit from n represented in unary and a description of M^* invokes a slightly revised version of the program of Fig. 3.27 to construct the circuit computing f_n. This revised circuit contains placeholders for the values of the n letters representing the input to M. The program uses space $O(\log p^2(n))$, which is logarithmic in n. ■

We now show that the function computed by a log-space uniform family of circuits can be computed by a polynomial-time Turing machine.

THEOREM 8.13.2 *Let $\mathcal{C}$ be a log-space uniform circuit family. Then there exists a polynomial-time Turing machine M that computes the same set of functions computed by the circuits in $\mathcal{C}$.*

Proof Let $M_{\mathcal{C}}$ be the log-space TM that computes the circuit family $\mathcal{C}$. We design the TM M to compute the same set of functions on an input $\boldsymbol{w}$ of length n. M uses $\boldsymbol{w}$ to obtain a unary representation for the input $M_{\mathcal{C}}$. It uses $M_{\mathcal{C}}$ to write down a description of the nth circuit on its work tape. It then computes the outputs of this circuit in time quadratic in the length of the circuit. Since the length of the circuit is a polynomial in n because the circuit is generated by a log-space TM (see Theorem 8.5.8), the running time of M is polynomial in the length of $\boldsymbol{w}$. ■

These two results can be generalized to uniform circuit families and Turing machines that use more than logarithmic space and polynomial time, respectively. (See Problem 8.32.)

In the above discussion we examine functions computed by Turing machines. If these functions are **characteristic functions**, $f : \mathcal{B}^* \mapsto \mathcal{B}$; that is, they have value 0 or 1, then those strings for which f has value 1 define a language L_f. Also, associated with each language $L \subseteq \mathcal{B}^*$ is a characteristic function $f_L : \mathcal{B}^* \mapsto \mathcal{B}$ that has value 1 on only those strings in L.

Consider now a language $L \subseteq \mathcal{B}^*$. For each $n \geq 1$ a circuit can be constructed whose value is 1 on binary strings in $L \cap \mathcal{B}^n$ and 0 otherwise. Similarly, given a family $\mathcal{C}$ of circuits such that for each natural number $n \geq 1$ the nth circuit, C_n, computes a Boolean function on n inputs, the language L associated with this circuit family contains only those strings of length n for which C_n has value 1. We say that ***L* is recognized by *C***. At the risk of confusion, we use the same name for a circuit family and the languages they define.

In Theorem 8.5.6 we show that $\mathbf{NSPACE}(r(n)) \subseteq \mathbf{TIME}(k^{\log n + r(n)})$. We now use the ideas of that proof together with the parallel algorithm for transitive closure given in Section 6.4 to show that languages in $\mathbf{NSPACE}(r(n))$, $r(n) \geq \log n$, are recognized by a uniform family of circuits in which the nth circuit has size $O(k^{\log n + r(n)})$ and depth $O(r^2(n))$. When $r(n) = O(\log n)$, the circuit family in question is contained in the class $\mathbf{NC^2}$ introduced in the next section.

THEOREM 8.13.3 *If language $L \subseteq \mathcal{B}^*$ is in $\mathbf{NSPACE}(r(n))$, $r(n) \geq \log n$, there exists a time-$r(n)$ uniform family of circuits recognizing L such that the nth circuit has size $O(k^{\log n + r(n)})$ and depth $O(r^2(n))$ for some constant k.*

Proof We assume without loss of generality that the NDTM accepting L has one accepting configuration. We then construct the adjacency matrix for the configuration graph of M. This matrix has a 1 entry in row i, column j if there is a transition from the ith to the jth configuration. All other entries are 0. From the analysis of Corollary 8.5.1, this graph has $O(k^{\log n + r(n)})$ configurations. The initial configuration is determined by the word $\boldsymbol{w}$ written initially on the tape of the NDTM accepting L. If the transitive closure of this matrix has a 1 in the row and column corresponding to the initial and final configurations, respectively, then the word $\boldsymbol{w}$ is accepted.

From Theorem 6.4.1 the transitive closure of a Boolean $p \times p$ matrix A can be computed by computing $(I + A)^q$ for $q \geq p - 1$. This can be done by squaring A s times for $s \geq \log_2 p$. From this we conclude that the transitive closure can be computed by a circuit of depth $O(\log^2 m)$, where m is the number of configurations. Since $m = O(k^{\log n + r(n)})$, we have the desired circuit size and depth bounds.

A program to compute the dth power of an $p \times p$ matrix A is shown in Fig. 8.20. This program can be converted to one that writes the description of a circuit for this purpose, and both the original and converted programs can be realized in space $O(d \log p)$. (See

```
trans(A, n, d, i, j)
  if d = 0 then
    return(a_{i,j})
  else
    return(∑_{k=1}^{n} trans(A, n, d − 1, i, k) * trans(A, n, d − 1, k, j))
```

Figure 8.20 A recursive program to compute the dth power of an $n \times n$ matrix A.

Problem 8.33.) Invoking this procedure to write a program for the above problem, we see that an $O(r^2(n))$-depth circuit recognizing L can be written by an $O(r^2(n))$-time DTM.■

8.14 The Parallel Random-Access Machine Model

The PRAM model, introduced in Section 7.9, is an abstraction of realistic parallel models that is sufficiently rich to permit the study of parallel complexity classes. (See Fig. 7.21, repeated as Fig. 8.21.) The PRAM consists of a set of RAM processors with a bounded number of memory locations and a common memory. The words of the common memory are allowed to be of unlimited size, but the instructions that the RAM processors can apply to them are restricted. These processors can perform addition, subtraction, vector comparison operations, conditional branching, and shifts by fixed amounts. We also allow load and store instructions for moving words between registers, local memories, and the common memory. These instructions are sufficiently rich to compute all computable functions.

In the next section we show that the CREW (concurrent read/exclusive write) PRAM that runs in polynomial time and the log-space uniform circuits characterize the same complexity classes. We then go on to explore the parallel complexity thesis, which states that sequential space and parallel time are polynomially related.

8.14.1 Equivalence of the CREW PRAM and Circuits

Because a parallel machine with p processors can provide a speedup of at most a factor of p over a comparable serial machine (see Theorem 7.4.1), problems that are computationally infeasible on serial machines are computationally infeasible on parallel machines with a reasonable number of processors. For this reason the study of parallelism is usually limited to feasible problems, that is, problems that can be solved in serial polynomial time (the class **P**). We limit our attention to such problems here.

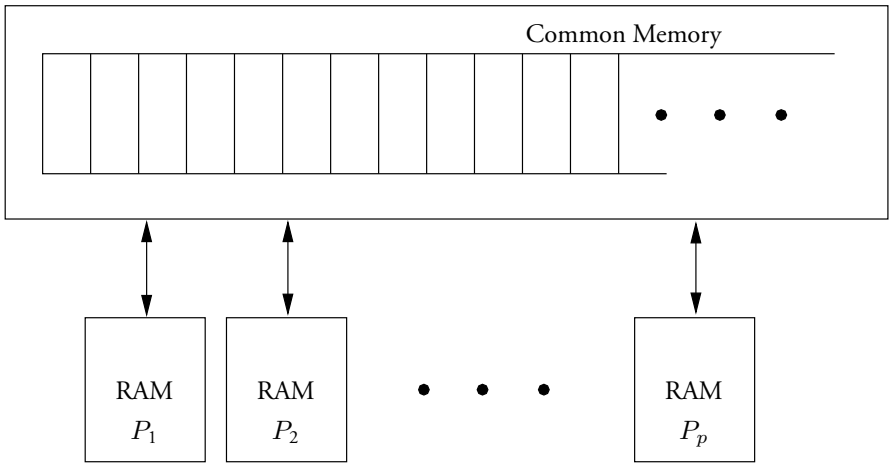


Figure 8.21 The PRAM consists of synchronous RAMs accessing a common memory.

Connections between PRAMs and circuits can be derived that are similar to those stated for Turing machines and circuits in Section 8.13.2. In this section we consider only log-space uniform families of circuits.

Given a PRAM, we now construct a circuit simulating it. This construction is based on that given in Section 3.4. With a suitable definition of **log-space uniform family of PRAMs** the circuits described in the following lemma constitute a log-space uniform family of circuits. (See Problem 8.35.) Also, this theorem can be extended to PRAMs that access memory locations with addresses much larger than $O(p(n)t(n))$, perhaps through indirect addressing. (See Problem 8.37.)

LEMMA 8.14.1 *Consider a function on input words of total length n bits computed by a CREW PRAM P in time $t(n)$ with a polynomial number of processors $p(n)$ in which the largest common memory address is $O(p(n)t(n))$. This function can be computed by a circuit of size $O(p^2(n)t(n) + p(n)t^2(n))$ and depth $O(\log(p(n)t(n)))$.*

Proof Since P executes at most $t(n)$ steps, by a simple extension to Problem 8.4 (only one RAM CPU at a time writes a word), we know that after $t(n)$ steps each word in the common memory of the PRAM has length at most $b = t(n) + n + K$ for some constant $K \geq 0$, because the PRAM can only compare or add numbers or shift them left by one position on each time step. This follows because each RAM CPU uses integers of fixed length and the length of the longest word in the common memory is initially n.

We exhibit a circuit for the computation by P by modifying and extending the circuit sketched in Section 3.4 to simulate one RAM CPU. This circuit uses the next-state/output circuit for the RAM CPU together with the next-state/output circuit for the random-access memory of Fig. 3.21 (repeated in Fig. 8.22). The circuit of Fig. 8.22(a) either writes a new value d_j for $w^*_{i,j}$, the jth component of the ith memory word of the random-access memory, or it writes the old value $w_{i,j}$. The circuit simulating the common memory of the PRAM is obtained by replacing the three gates at the output of the circuit in Fig. 8.22(a) with a subcircuit that assigns to $w^*_{i,j}$ the value of $w_{i,j}$ if $c_i = 0$ for each RAM CPU and the OR of the values of d_j supplied by each RAM CPU if $c_i = 1$ for some CPU. Here we count on the fact that at most one CPU addresses a given location for writing. Thus, if a CPU writes to a location, all other CPUs cannot do so. Concurrent reading is simulated by allowing every component of every memory cell to be used as input by every CPU.

Since the longest word that can be constructed by the CREW PRAM has length $b = t(n)+n+K$, it follows from Lemma 3.5.1 that the next-state/output circuit for the random-access memory designed for one CPU has size $O(p(n)t^2(n))$ and depth $O(\log(p(n)t(n)))$. The modifications described in the previous paragraph add size $O(p^2(n)t(n))$ (each of the $p(n)t(n)$ memory words has $O(p(n))$ new gates) and depth $O(\log p(n))$ (each OR tree has $p(n)$ inputs) to this circuit. As shown at the end of Section 3.10, the size and depth of a circuit for the next-state/output circuit of the CPU are $O(t(n) + \log(p(n)t(n)))$ and $O(\log t(n) + \log\log(p(n)t(n)))$, respectively. Since these sizes and depths add to those for the common memory, the total size and depth for the next-state/output circuit for the PRAM are $O(p^2(n)t(n) + p(n)t^2(n))$ and $O(\log(p(n)t(n)))$, respectively. ■

We now show that the function computed by a log-space uniform circuit family can be computed in poly-logarithmic time on a PRAM.

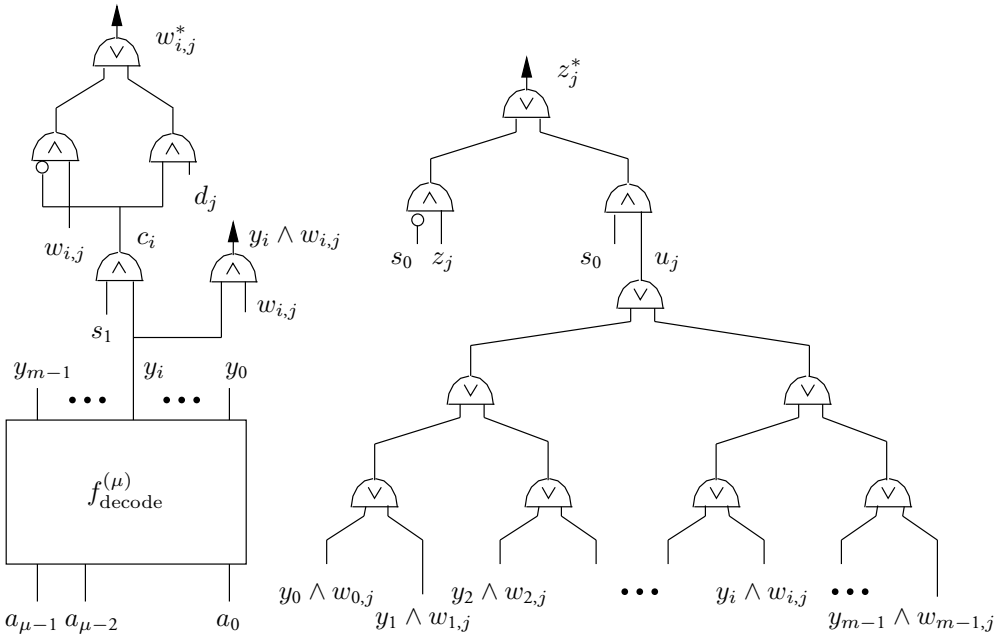


Figure 8.22 A circuit for the next-state and output function of the random-access memory. The circuit in (a) computes the next values for components of memory words, whereas that in (b) computes components of the output word. This circuit is modified to generate a circuit for the PRAM.

LEMMA 8.14.2 *Let $\mathcal{C} = (C_1, C_2, \ldots\}$ be a log-space uniform family of circuits. There exists a CREW PRAM that computes in poly-logarithmic time and a polynomial number of processors the function $f : \mathcal{B}^* \mapsto \mathcal{B}^*$ computed by $\mathcal{C}$.*

Proof The CREW PRAM is given a string $\boldsymbol{w}$ on which to compute the function f. First it computes the length n of $\boldsymbol{w}$. Second it invokes the CREW PRAM described below to simulate with a polynomial number of processors in poly-logarithmic time the log-space DTM M that writes a description of the nth circuit, $C(M, n)$. Finally we show that the value of $C(M, n)$ can be evaluated from this description by a CREW PRAM in $O(\log^2 n)$ steps with polynomially many processors.

Let M be a three-tape DTM that realizes a log-space transformation. This DTM has a read-only input tape, a work tape, and a write-only output tape. Given a string $\boldsymbol{w}$ on its input tape, it provides on its output tape the result of the transformation. Since M uses $O(\log n)$ cells on its work tape on inputs of length n, it can be modeled by a finite-state machine with $2^{O(\log n)}$ states. The circuit $C(M, n)$ described in Theorem 3.2.2 for the simulation of the FSM M is constructed to simulate M on inputs of length n. We show that $C(M, n)$ has size and depth that are polynomial and poly-logarithmic in n, respectively. We then demonstrate that a CREW PRAM can simulate $C(M, n)$ (and write its output into its common memory) in $O(\log^2 n)$ steps with a polynomial number of processors.

From Theorem 8.5.8 we know that the log-space DTM M generating $C(M,n)$ does not execute more than $p(n)$ steps, $p(n)$ a polynomial in n. Since $p(n)$ is assumed proper, we can assume without loss of generality that M executes $p(n)$ steps on all inputs of length n. Thus, M has exactly $|Q| = O(p(n))$ configurations.

The input string $\boldsymbol{w}$ is placed in the first n locations of the otherwise blank common memory. To determine the length of the input, for each i the ith CREW PRAM processor examines the words in locations i and $i+1$. If location $i+1$ is blank but location i is not, $i = n$. The nth processor then computes $p(n)$ in $O(\log^2 n)$ serial steps (see Problem 8.2) and places it in common memory.

The circuit $C(M,n)$ is constructed from representations of next-state mappings, one mapping for every state transition. Since there are no external inputs to M (all inputs are recorded on the input tape before the computation begins), all next-state mappings are the same. As shown in Section 3.2, let this one mapping be defined by a Boolean $|Q| \times |Q|$ matrix M_Δ whose rows and columns are indexed by configurations of M. A configuration of M is a tuple $(q, h_1, h_2, h_3, \boldsymbol{x})$ in which q is the current state, h_1, h_2, and h_3 are the positions of the heads on the input, output, and work tapes, respectively, and $\boldsymbol{x}$ is the current contents of the work tape. Since M computes a log-space transformation, it executes a polynomial number of steps. Thus, each configuration has length $O(\log n)$. Consequently, a single CREW PRAM can determine in $O(\log n)$ time whether an entry in row r and column c, where r and c are associated with configurations, has value 0 or 1. For concreteness, assign PRAM processor i to row r and column c of M_Δ, where $r = \lceil i/p(n) \rceil$ and $c = i - r \times p(n)$, quantities that can be computed in $O(\log^2 n)$ steps.

The circuit $C(M,n)$ simulating M is obtained via a prefix computation on $p(n)$ copies of the matrix M_Δ using matrix multiplication as the associative operator. (See Section 3.2.)

Once $C(M,n)$ has been written into the common memory, it can be evaluated by assigning one processor per gate and then computing its value as many times as the depth of $C(M,n)$. This involves a four-phase operation in which the jth processor reads each of the at most two arguments of the jth gate in the first two phases, computes its value in the third, and then writes it to common memory in the fourth. This process is repeated as many times as the depth of the circuit $C(M,n)$, thereby insuring that correct values for gates propagate throughout the circuit. Again concurrent reads and exclusive writes suffice. ■

These two results (and Problem 8.37) imply the result stated below, namely, that the binary functions computed by circuits with polynomial size and poly-logarithmic depth are the same as those computed by the CREW PRAM with polynomially many processors and poly-logarithmic time.

THEOREM 8.14.1 *The functions $f : \mathcal{B}^* \mapsto \mathcal{B}^*$ computed by circuits of polynomial-size and poly-logarithmic depth are the same as those computed by the CREW PRAM with a polynomial number of processors and poly-logarithmic time.*

8.14.2 The Parallel Computation Thesis

A deep connection exists between serial space and parallel time. The **parallel computation thesis** states that sequential space and parallel time are polynomially related; that is, if there exists a sequential algorithm that uses space S, then there exists a parallel algorithm using time $p(S)$ for some polynomial p and vice versa. There is strong evidence that this hypothesis holds.

In this section we set the stage for discussing the parallel computation thesis in a limited way by showing that every log-space reduction (on a Turing machine) can be realized by a CREW PRAM in time $O\left(\log^2 n\right)$ with polynomially many processors. This implies that if a **P**-complete problem can be solved on a PRAM with polynomially many processors in poly-logarithmic time, then so can every problem in **P**, an unlikely prospect.

LEMMA 8.14.3 *Log-space transformations can be realized by CREW PRAMs with polynomially many processors in time* $O(\log^2 n)$.

Proof We use the CREW PRAM described in the proof of Lemma 8.14.2. The processors in this PRAM are then assigned to perform the matrix operations in the order required for a parallel prefix computation. (See Section 2.6.) If we assign $|Q(n)|^2$ processors per matrix multiplication operation, each operation can be done in $O(\log |Q(n)|^2) = O(\log n)$ steps. Since the prefix computation has depth $O(\log n)$, the PRAM can perform the prefix computation in time $O(\log^2 n)$. The number of processors used is $p(n) \cdot O(|Q(n)|^2)$, which is a polynomial in n. Concurrent reads and exclusive writes suffice for these operations. ■

Since a log-space transformation can be realized in poly-logarithmic time with polynomially many processors on a CREW PRAM, if a CREW PRAM solves a **P**-complete problem in poly-logarithmic time, we can compose such machines to form a CREW PRAM with poly-logarithmic time and polynomially many processors to solve an arbitrary problem in **P**.

THEOREM 8.14.2 *If a* **P***-complete problem can be solved in poly-logarithmic time with polynomially many processors on a CREW PRAM, then so can all problems in* **P** *and all problems in* **P** *are fully parallelizable.*

8.15 Circuit Complexity Classes

In this section we introduce several important circuit complexity classes including **NC**, the languages recognized by uniform families of circuits whose size and depth are polynomial and poly-logarithmic in n, respectively, and **P/poly**, the largest set of languages $L \subset \mathcal{B}^*$ with the property that L is recognized by a (non-uniform) circuit family of polynomial size. We also derive relationships among these classes and previously defined classes.

8.15.1 Efficiently Parallelizable Languages

DEFINITION 8.15.1 *The class* $\mathbf{NC}^k$ *contains those languages* L *recognized by a uniform family of Boolean circuits of polynomial size and depth* $O(\log^k n)$ *in* n*, the length of an input. The class* **NC** *is the union of the classes* $\mathbf{NC}^k$*,* $k \geq 1$*; that is,*

$$\mathbf{NC} = \bigcup_{k \geq 1} \mathbf{NC}^k$$

In Section 8.14 we explored the connection between circuit size and depth and PRAM time and number of processors and concluded that circuits having polynomial size and poly-logarithmic depth compute the same languages as do PRAMs with a polynomial number of processors and poly-logarithmic parallel time.

The class **NC** is considered to be the largest feasibly parallelizable class of languages. By **feasible** we mean that the number of gates (equivalently processors) is no more than polynomial in the length n of the input and by **parallelizable** we mean that circuit depth (equivalently computation time) must be no more than poly-logarithmic in n. Feasibly parallelizable languages meet both requirements.

The prefix circuits introduced in Section 2.6 belong to $\mathbf{NC^1}$, as do circuits constructed with prefix operations, such as binary addition and subtraction (see Section 2.7) and the circuits for solutions of linear recurrences (see Problem 2.24). (Strictly speaking, these functions are not predicates and do not define languages. However, comparisons between their values and a threshold converts them to predicates. In this section we liberally mix functions and predicates.) The class $\mathbf{NC^1}$ also contains functions associated with integer multiplication and division.

The fast Fourier transform (see Section 6.7.3) and merging networks (see Section 6.8) can both be realized by algebraic and combinatorial circuits of depth $O(\log n)$, where n is the number of circuit inputs. If the additions and multiplications of the FFT are done over a ring of integers modulo m for some m, the FFT can be realized by a circuit of depth $O(\log^2 n)$. If the items to be merged are represented in binary, a comparison operator can be realized with depth $O(\log n)$ and merging can also be done with a circuit of depth $O(\log^2 n)$. Thus, both problems are in $\mathbf{NC^2}$.

When matrices are defined over a field of characteristic zero, the inverse of invertible matrices (see Section 6.5.5) can be computed by an algebraic circuit of depth $O(\log^2 n)$. If the matrix entries when represented as binary numbers have size n, the ring operations may be realized in terms of binary addition and multiplication, and matrix inversion is in $\mathbf{NC^3}$.

Also, it follows from Theorem 8.13.3 that the nth circuit in the log-space uniform families of circuits has polynomial size and depth $O(\log^2 n)$; that is, it is contained in $\mathbf{NC^2}$. Also contained in this set is the transitive closure of a Boolean matrix (see Section 6.4). Since the circuits constructed in Chapter 3 to simulate finite-state machines as well as polynomial-time Turing machines are log-space uniform (see Theorem 8.13.1), each of these circuit families is in $\mathbf{NC^2}$.

We now relate these complexity classes to one another and to **P**.

THEOREM 8.15.1 *For $k \geq 2$, $\mathbf{NC^1} \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{NC^2} \subseteq \mathbf{NC^k} \subseteq \mathbf{NC} \subseteq \mathbf{P}$.*

Proof The containment $\mathbf{L} \subseteq \mathbf{NL}$ is obvious. The containment $\mathbf{NL} \subseteq \mathbf{NC^2}$ is a restriction of the result of Theorem 8.13.3 to $r(n) = O(\log n)$. The containments $\mathbf{NC^2} \subseteq \mathbf{NC^k} \subseteq \mathbf{NC}$ follow from the definitions. The last containment, $\mathbf{NC} \subseteq \mathbf{P}$, is a consequence of the fact that the circuit on n inputs in a log-space uniform family of circuits, call it C_n, can be generated in polynomial time by a Turing machine that can then evaluate C_n in a time quadratic in its length, that is, in polynomial time. (Theorems 8.5.8 and 8.13.2 apply.)

The first containment, namely $\mathbf{NC^1} \subseteq \mathbf{L}$, is slightly more difficult to establish. Given a language $L \in \mathbf{NC^1}$, consider the problem of recognizing whether or not a string $\boldsymbol{w}$ is in L. This recognition task is done in log-space by invoking two log-space transformations, as is now explained.

The first log-space transformation generates the nth circuit, C_n, in the family recognizing L. C_n has value 1 if $\boldsymbol{w}$ is in L and 0 otherwise. By definition, C_n has size polynomial in n. Also, each circuit is described by a straight-line program, as explained in Section 2.2.

The second log-space transformation evaluates the circuit with temporary work space proportional to the maximal length of such strings. If the strings identifying gates have larger length, their transformation would use more space. (Note that it is easy to identify gates with an $O(\log^2 n)$-length string(s) by concatenating the number of each gate on the path to it, including itself.) For this reason we give an efficient encoding of gate locations.

The gates of circuits in $\mathbf{NC^1}$ generally have fan-out exceeding 1. That is, they have more than one parent gate in the circuit. We describe how to identify gates with strings that may associate multiple strings with a gate. We walk the graph, which is the circuit, starting from the output vertex and moving toward input vertices. The output gate is identified with the empty string string ϵ. If we reach a gate g via a parent whose string is $\boldsymbol{p}$, g is identified by $\boldsymbol{p}0$ or $\boldsymbol{p}1$. If the parent has only one descendant, as would be the case for NOT gates and inputs, we represent g by $\boldsymbol{p}0$. If it has two descendants, as would be the case for AND and OR, and g has the smaller gate number, its string is $\boldsymbol{p}0$; otherwise it is $\boldsymbol{p}1$.

The algorithm to produce each of these binary strings can be executed in logarithmic space because one need only walk each path in the circuit from the output to inputs. The tuple defining each gate contains the gate numbers of its predecessors, $O(\log n)$-length numbers, and the algorithm need only carry one such number at a time in its working memory to find the location of a predecessor gate in the input string containing the description of the circuit.

The second log-space transformation evaluates the circuit using the binary strings describing the circuit. It visits the input vertex with the lexicographically smallest string and determines its value. It then evaluates the gate whose string is that of the input vertex minus the last bit. Even though it may have to revisit all gates on the path to this vertex to do this, $O(\log n)$ space is used. If this gate is either a) AND and the input has value 0, b) OR and the input has value 1, or c) NOT, the value of the gate is decided. If the gate has more than one input and its value is not decided, the other input to it is evaluated (the one with suffix 1). Because the second input to the gate is evaluated only if needed, its value determines the value of the gate. This process is repeated at each gate in the circuit until the output gate is reached and its value computed. Since this procedure keeps only one path of length $O(\log n)$ active at a time, the algorithm uses space $O(\log n)$. ■

An important open question is whether the complexity hierarchy of this theorem collapses and, if so, where. For example, is it true that a problem in **P** is also in **NC**? If so, all serial polynomial-time problems are parallelizable with a number of processing elements polynomial in the length of the input and poly-logarithmic time, an unlikely prospect.

8.15.2 Circuits of Polynomial Size

We now examine the class of languages **P/poly** and show that they are exactly the languages recognized by Boolean circuits of polynomial size. To set the stage we introduce advice and pairing functions.

DEFINITION 8.15.2 *An* **advice function** $a : \mathbb{N} \mapsto \mathcal{B}^*$ *maps natural numbers to binary strings. A* **polynomial advice function** *is an advice function for which* $|a(n)| \leq p(n)$ *for* $p(n)$ *a polynomial function in* n.

DEFINITION 8.15.3 *A* **pairing function** $<,>: \mathcal{B}^* \times \mathcal{B}^* \mapsto \mathcal{B}^*$ *encodes pairs of binary strings* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *with two end markers and a separator (a comma) into the binary string* $< \boldsymbol{x}, \boldsymbol{y} >$.

Pairing functions can be very easy to describe and compute. For example, $< \boldsymbol{x}, \boldsymbol{y} >$ can be implemented by representing 0 by 01, 1 by 10, both $<$ and $>$ by 11, and , (comma) by 00. Thus, $< 0010, 110 >$ is encoded as 11010110010010100111. It is clearly trivial to identify, extract, and decode each component of the pair. We are now prepared to define **P/poly**.

DEFINITION 8.15.4 *Let $a : \mathbb{N} \mapsto \mathcal{B}^*$ be a polynomial advice function.* **P/poly** *is the set of languages $L = \{\boldsymbol{w} \mid < \boldsymbol{w}, a(|\boldsymbol{w}|) > \in A\}$ for which there is a language A in* **P**.

The **advice** $a(|\boldsymbol{w}|)$ given on a string $\boldsymbol{w}$ in a language $L \in$ **P/poly** is the same for all strings of the same length. Furthermore, $< \boldsymbol{w}, a(|\boldsymbol{w}|) >$ must be easy to recognize, namely, recognizable in polynomial time.

The subset of the languages in **P/poly** for which the advice function is the empty string is exactly the languages in **P**, that is, **P** $\subseteq$ **P/poly**.

The following result is the principal result of this section. It gives two different interpretations of the advice given on strings.

THEOREM 8.15.2 *A language L is recognizable by a family of circuits of polynomial size if and only if $L \in$* **P/poly**.

Proof Let L be recognizable by a family $\mathcal{C}$ of circuits of polynomial size. We show that it is in **P/poly**.

Let $\overline{C}_n$ be an encoding of the circuit C_n in $\mathcal{C}$ that recognizes strings in $L \cap \mathcal{B}^n$. Let the advice function $a(n) = \overline{C}_n$ and let $\boldsymbol{w} \in \mathcal{B}^*$ have length n. Then, $\boldsymbol{w} \in \mathcal{B}^n$ if and only if the value of C_n on $\boldsymbol{w}$ is 1. Since $\boldsymbol{w}$ has length polynomial in n, $\boldsymbol{w} \in \mathcal{B}^n$ if and only if the pairing function $< \boldsymbol{w}, a(|\boldsymbol{w}|) >$ is an instance of CIRCUIT SAT, which has been shown to be in **P**. (See Theorem 8.13.2.)

On the other hand, suppose that $L \in$ **P/poly**. We show that L is recognizable by circuits of polynomial size. By definition there is an advice function $a : \mathbb{N} \mapsto \mathcal{B}^*$ and a language $A \in$ **P** for L such that for all $\boldsymbol{w} \in L$, $< \boldsymbol{w}, a(|\boldsymbol{w}|) > \in A$. Since $A \in$ **P**, there is a polynomial-time DTM that accepts $< \boldsymbol{w}, a(|\boldsymbol{w}|) >$. By Theorem 8.13.1 there is a circuit of polynomial size that recognizes $< \boldsymbol{w}, a(|\boldsymbol{w}|) >$. The string $a(|\boldsymbol{w}|)$ is constant for strings $\boldsymbol{w}$ of length n. Thus, the circuit for $A \cap \mathcal{B}^n$ to which is supplied the constant string $a(|\boldsymbol{w}|)$ is a circuit of length polynomial in n that accepts strings $\boldsymbol{w}$ in L. ■

Problems

MATHEMATICAL PRELIMINARIES

8.1 Show that if strings over an alphabet $\mathcal{A}$ with at least two letters are encoded over a one-letter alphabet (a **unary encoding**), then strings of length n over $\mathcal{A}$ require strings of length exponential in n in the unary encoding.

8.2 Show that the polynomial function $p(n) = K_1 n^k$ can be computed in $O(\log^2 n)$ serial steps from n and for constants $K_1 \geq 1$ and $k \geq 1$ on a RAM when additions require one unit of time.

SERIAL COMPUTATIONAL MODELS

8.3 Given an instance of satisfiability, namely, a set of clauses over a set of literals and values for the variables, show that the clauses can be evaluated in time quadratic in the length of the instance.

8.4 Consider the RAM of Section 8.4.1. Let $l(\mathcal{I})$ be the length, measured in bits, of the contents $\mathcal{I}$ of the RAMs input registers. Similarly, let $l(v)$ be the maximal length of any integer addressed by an instruction in the RAMs program. Show that after k steps the contents of any RAM memory location is at most $k + l(\mathcal{I}) + l(v)$.

Given an example of a computation that produces a word of length k.

Hint: Consider which instructions have the effect of increasing the length of an integer used or produced by the RAM program.

8.5 Consider the RAM of Section 8.4.1. Assume the RAM executes T steps. Describe a Turing-machine simulation of this RAM that uses space proportional to T^2 measured in bits.

Hint: Represent each RAM memory location visited during a computation by an (`address`, `contents`) pair. When a RAM location is updated, fill the cells on the second tape containing the old (`address`, `contents`) pair with a special "blank" character and add the new (`address`, `contents`) pair to the end of the list of such pairs. Use the results of Problem 8.4 to bound the length of individual words.

8.6 Consider the RAM of Section 8.4.1. Using the result of Problem 8.5, describe a multitape Turing machine that simulates in $O(T^3)$ steps a T-step computation by the RAM.

Hint: Let your machine have seven tapes: one to hold the input, a second to hold the contents of RAM memory recorded as (`address`, `contents`) pairs separated and terminated by appropriate markers, a third to hold the current value of the program counter, a fourth to hold the memory address being sought, and three tapes for operands and results. On the input tape place the program to be executed and the input on which it is to be executed. Handle the second tape as suggested in Problem 8.5. When performing an operation that has two operands, place them on the fifth and sixth tapes and the result on the seventh tape.

8.7 Justify using the number of tape cells as a measure of space for the Turing machine when the more concrete measure of bits is used for the space measure for the RAM.

CLASSIFICATION OF DECISION PROBLEMS

8.8 Given a Turing machine, deterministic or not, show that there exists another Turing machine with a larger tape alphabet that performs the same computation but in a number of steps and number of tape cells that are smaller by constant factors.

8.9 Show that strings in TRAVELING SALESPERSON can be accepted by a deterministic Turing machine in an exponential number of steps.

COMPLEMENTS OF COMPLEXITY CLASSES

8.10 Show that VALIDITY is log-space complete for **coNP**.

8.11 Prove that the complements of **NP**-complete problems are **coNP**-complete.

8.12 Show that the complexity class **P** is contained in the intersection of **NP** and **coNP**.

8.13 Demonstrate that **coNP** $\subseteq$ **PSPACE**.

8.14 Prove that if a **coNP**-complete problem is in **NP**, then **NP** $=$ **coNP**.

REDUCTIONS

8.15 If $\mathcal{P}_1$ and $\mathcal{P}_2$ are decision problems, a **Turing reduction** from $\mathcal{P}_1$ to $\mathcal{P}_2$ is any OTM that solves $\mathcal{P}_1$ given an oracle for $\mathcal{P}_2$. Show that the reductions of Section 2.4 are Turing reductions.

8.16 Prove that the reduction given in Section 10.9.1 of a pebble game to a branching computation is a Turing reduction. (See Problem 8.15.)

8.17 Show that if a problem $\mathcal{P}_1$ can be Turing-reduced to problem $\mathcal{P}_2$ by a polynomial-time OTM and $\mathcal{P}_2$ is in **P**, then $\mathcal{P}_1$ is also in **P**.

Hint: Since each invocation of the oracle can be done deterministically in polynomial time in the length of the string written on the oracle tape, show that it can be done in time polynomial in the length of the input to the OTM.

8.18 a) Show that every fixed power of an integer written as a binary k-tuple can be computed by a DTM in space $O(k)$.

b) Show that every fixed polynomial in an integer written as a binary k-tuple can be computed by a DTM in space $O(k)$.

Hint: Show that carry-save addition can be used to multiply two k-bit integers with work space $O(k)$.

HARD AND COMPLETE PROBLEMS

8.19 The class of polynomial-time Turing reductions are Turing reductions in which the OTM runs in time polynomial in the length of its input. Show that the class of Turing reductions is transitive.

P-COMPLETE PROBLEMS

8.20 Show that numbers can be assigned to gates in an instance of MONOTONE CIRCUIT VALUE that corresponds to an instance of CIRCUIT VALUE in Theorem 8.9.1 so that the reduction from it to MONOTONE CIRCUIT VALUE can be done in logarithmic space.

8.21 Prove that LINEAR PROGRAMMING described below is **P**-complete.

LINEAR PROGRAMMING
Instance: Integer-valued $m \times n$ matrix A and column m-vectors $\boldsymbol{b}$ and $\boldsymbol{c}$.
Answer: "Yes" if there is a rational column n-vector $\boldsymbol{x} > 0$ such that $A\boldsymbol{x} < \boldsymbol{b}$ and $\boldsymbol{x}$ maximizes $\boldsymbol{c}^T\boldsymbol{x}$.

NP-COMPLETE PROBLEMS

8.22 A **Horn clause** has at most one **positive literal** (an instance of x_i). Every other literal in a Horn clause is a **negative literal** (an instance of $\overline{x}_i$). HORN SATISFIABILITY is an

instance of SATISFIABILITY in which each clause is a Horn clause. Show that HORN SATISFIABILITY is in **P**.

Hint: If all literals in a clause are negative, the clause is satisfied only if some associated variables have value 0. If a clause has one positive literal, say y, and negative literals, say $\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_k$, then the clause is satisfied if and only if the implication $x_1 \wedge x_2 \wedge \cdots \wedge x_k \Rightarrow y$ is true. Thus, y has value 1 when each of these variables has value 1. Let $\mathcal{T}$ be a set variables that must have value 1. Let $\mathcal{T}$ contain initially all positive literals that appear alone in a clause. Cycle through all implications and for each implication all of whose left-hand side variables appear in $\mathcal{T}$ but whose right-hand side variable does not, add this variable to $\mathcal{T}$. Since $\mathcal{T}$ grows until all left-hand sides are satisfied, this procedure terminates. Show that all satisfying assignments contain $\mathcal{T}$.

8.23 Describe a polynomial-time algorithm to determine whether an instance of CIRCUIT SAT is a "yes" instance when the circuit in question consists of a layer of AND gates followed by a layer of OR gates. Inputs are connected to AND gates and the output gate is an OR gate.

8.24 Prove that the CLIQUE problem defined below is **NP**-complete.

CLIQUE
Instance: The description of an undirected graph $G = (V, E)$ and an integer k.
Answer: "Yes" if there is a set of k vertices of G such that all vertices are adjacent.

8.25 Prove that the HALF CLIQUE problem defined below is **NP**-complete.

HALF CLIQUE
Instance: The description of an undirected graph $G = (V, E)$ in which $|V|$ is even and an integer k.
Answer: "Yes" if G contains a clique on $|V|/2$ vertices or has more than k edges.

Hint: Try reducing an instance of CLIQUE on a graph with m vertices and a clique of size k to this problem by expanding the number of vertices and edges to create a graph that has $|V| \geq m$ vertices and a clique of size $|V|/2$. Show that a test for the condition that G contains more than k edges can be done very efficiently by counting the number of bits among the variables describing edges.

8.26 Show that the NODE COVER problem defined below is **NP**-complete.

NODE COVER
Instance: The description of an indirected graph $G = (V, E)$ and an integer k.
Answer: "Yes" if there is a set of k vertices such that every edge contains at least one of these vertices.

8.27 Prove that the HAMILTONIAN PATH decision problem defined below is **NP**-complete.

HAMILTONIAN PATH
Instance: The description of an undirected graph G.
Answer: "Yes" if there is a path visiting each node once.

Hint: 3-SAT can be reduced to HAMILTONIAN PATH, but the construction is challenging. First, add literals to clauses in an instance of 3-SAT so that each clause has

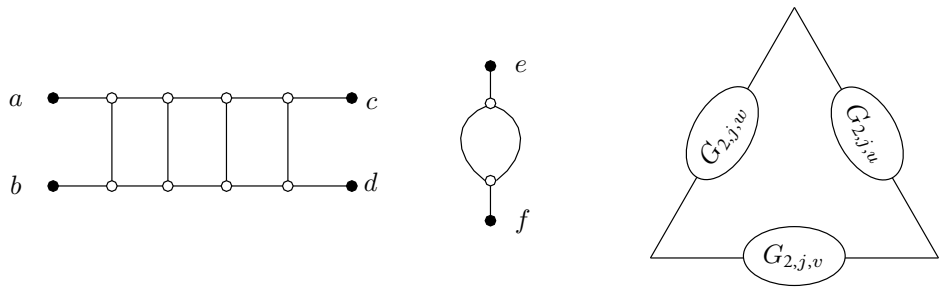


Figure 8.23 Gadgets used to reduce 3-SAT to HAMILTONIAN PATH.

three literals. Second, construct and interconnect three types of subgraphs (gadgets). Figures 8.23(a) and (b) show the first and second of theses gadgets, G_1 and G_2.

There is one first gadget for each variable x_i, $1 \leq i \leq n$, denoted $G_{1,i}$. The left path between the two middle vertices in $G_{1,i}$ is associated with the value $x_i = 1$ and the right path is associated with the complementary value, $x_i = 0$. Vertex f of $G_{1,i}$ is identified with vertex e of $G_{1,i+1}$ for $1 \leq i \leq n-1$, vertex e of $G_{1,1}$ is connected only to a vertex in $G_{1,1}$, and vertex f of $G_{1,n}$ is connected to the clique described below.

There is one second gadget for each literal in each clause. Thus, if x_i ($\overline{x}_i$) is a literal in clause c_j, then we create a gadget $G_{2,j,i,1}$ ($G_{2,j,i,0}$).

Since a HAMILTONIAN PATH touches every vertex, a path through $G_{2,j,i,v}$ for $v \in \{0,1\}$ passes either from a to c or from b to d.

For each $1 \leq i \leq n$ the two parallel edges of $G_{1,i}$ are broken open and two vertices appear in each of them. For each instance of the literal x_i ($\overline{x}_i$), connect the vertices a and c of $G_{2,j,i,1}$ ($G_{2,j,i,0}$) to the pair of vertices on the left (right) that are created in $G_{1,i}$. Connect the b vertex of one literal in clause c_j to the d vertex of another one, as suggested in Fig. 8.23(c).

The third gadget has vertices g and h and a connecting edge. One of these two vertices, h, is connected in a clique with the b and d vertices of the gadgets $G_{2,j,i,v}$ and the f vertex of $G_{1,n}$.

This graph has a Hamiltonian path between g and the e vertex of $G_{1,1}$ if and only if the instance of 3-SAT is a "yes" instance.

8.28 Show that the TRAVELING SALESPERSON decision problem defined below is **NP**-complete.

TRAVELING SALESPERSON
Instance: An integer k and a set of $n(n-1)/2$ distances $\{d_{1,2}, d_{1,3}, \ldots, d_{1,n}, d_{2,3}, \ldots, d_{2,n}, \ldots, d_{n-1,n}\}$ between n cities.
Answer: "Yes" if there is a **tour** (an ordering) $\{i_1, i_2, \ldots, i_n\}$ of the cities such that the length $l = d_{i_1,i_2} + d_{i_2,i_3} + \cdots + d_{i_n,i_1}$ of the tour satisfies $l \leq k$.

Hint: Try reducing HAMILTONIAN PATH to TRAVELING SALESPERSON.

8.29 Give a proof that the PARTITION problem defined below is **NP**-complete.

PARTITION
Instance: A set $Q = \{a_1, a_2, \ldots, a_n\}$ of positive integers.
Answer: "Yes" if there is a subset of Q that adds to $\frac{1}{2}\sum_{1 \le i \le n} a_i$.

PSPACE-COMPLETE PROBLEMS

8.30 Show that the procedure `tree_eval` described in the proof of Theorem 8.12.1 can be modified slightly to apply to the evaluation of the trees generated in the proof of Theorem 8.12.3.

Hint: A vertex of in-degree k can be replaced by a binary tree of k leaves and depth $\log_2 k$.

THE CIRCUIT MODEL OF COMPUTATION

8.31 Prove that the class of circuits described in Section 3.1 that simulate a finite-state machine are uniform.

8.32 Generalize Theorems 8.13.1 and 8.13.2 to uniform circuit families and Turing machines that use more than logarithmic space and polynomial time, respectively.

8.33 Write a $O(\log^2 n)$-space program based on the one in Fig. 8.20 to describe a circuit for the transitive closure of an $n \times n$ matrix based on matrix squaring.

THE PARALLEL RANDOM-ACCESS MACHINE MODEL

8.34 Complete the proof of Lemma 8.14.2 by making specific assignments of data to memory locations. Also, provide formulas for the assignment of processors to tasks.

8.35 Give a definition of a **log-space uniform family of PRAMs** for which Lemma 8.14.1 can be extended to show that the function $f : \mathcal{B}^* \mapsto \mathcal{B}^*$ computed by a log-space family of PRAMs can also be computed by a log-space uniform family of circuits satisfying the conditions of Lemma 8.14.1.

8.36 Exhibit a non-uniform family of PRAMs that can solve problems that are not recursively enumerable.

8.37 Lemma 8.14.1 is stated for PRAMs in which the CPU does not access a common memory address larger than $O(p(n)t(n))$. In particular, this model does not permit indirect addressing. Show that this theorem can be extended to RAM CPUs that do allow indirect addressing by using the representation for memory accesses in Problem 8.6.

Chapter Notes

The classification of languages by the resources needed for their recognition is a very large subject capable of book-length study. The reader interested in going beyond the introduction given here is advised to consult one of the readily available references. The *Handbook of Theoretical Computer Science* contains three survey articles on this subject by van Embde Boas [349], Johnson [150], and Karp and Ramachandran [160]

The first examines simulation of one computational model by another for a large range of models. The second provides a large catalog of complexity classes and relationships between them. The third examines parallel algorithms and complexity. Other sources for more information on this topic are the books by Hopcroft and Ullman [140], Lewis and Papadimitriou [199], Balcázar, Díaz, and Gabarrò on structural complexity [27], Garey and Johnson [108] on the theory of **NP**-completeness, Greenlaw, Hoover, and Ruzzo [119] on **P**-completeness, and Papadimitriou [234] on computational complexity.

The Turing machine was defined by Alan Turing in 1936 [337], as was the oracle Turing machine. Random-access machines were introduced by Shepherdson and Sturgis [307] and the performance of RAMs was analyzed by Cook and Reckhow [77].

Hartmanis, Lewis, and Stearns [126,127] gave the study of time and space complexity classes its impetus. Their papers contain many of the basic theorems on complexity classes, including the space and time hierarchy theorems stated in Section 8.5.1. The gap theorem was obtained by Trakhtenbrot [333] and rediscovered by Borodin [51]. Blum [46] developed machine-independent complexity measures and established a speedup theorem showing that for some languages there is no single fastest recognition algorithm [47].

Many individuals identified and recognized the importance of the classes **P** and **NP**. Cook [74] formalized **NP**, emphasized the importance of polynomial-time reducibility, and exhibited the first **NP**-complete problem, SATISFIABILITY. Karp [158] then demonstrated that a number of other combinatorial problems, including TRAVELING SALESPERSON, are **NP**-complete. Cook used Turing reductions in his classification whereas Karp used polynomial-time transformations. Independently and almost simultaneously Levin [198] (see also [334]) was led to concepts similar to the above.

The relationship between nondeterministic and deterministic space (Theorem 8.5.5 and Corollary 8.5.1) was established by Savitch [296]. The proof that nondeterministic space classes are closed under complementation (Theorem 8.6.2 and Corollary 8.6.2) is independently due to Szelepscényi [321] and Immerman [144].

Theorem 8.6.4, showing that PRIMALITY is in $\mathbf{NP} \cap \mathbf{coNP}$, is due to Pratt [256].

Cook [75] defined the concept of a **P**-complete problem and exhibited the first such problem. He was followed quickly by Jones and Laaser [152] and Galil [107]. Ladner [184] showed that circuits simulating Turing machines (see [285]) could be constructed in logarithmic space, thereby establishing that CIRCUIT VALUE is **P**-complete. Goldschlager [116] demonstrated that MONOTONE CIRCUIT VALUE is **P**-complete. Valiant [344] and Cook established that LINEAR INEQUALITIES is **P**-hard, and Khachian [164] showed that this problem is in **P**. The proof that DTM ACCEPTANCE is **P**-complete is due to Johnson [150].

Cook [74] gave the first proof that SATISFIABILITY is **NP**-complete and also gave the reduction to 3-SAT. Independently, Levin [198] (see also [334]) was led to similar concepts for combinatorial problems. Schäfer [298] showed that NAESAT is **NP**-complete. Karp [158] established that 0-1 INTEGER PROGRAMMING, 3-COLORING, EXACT COVER, SUBSET SUM, TASK SEQUENCING, and INDEPENDENT SET are **NP**-complete.

The proof that 2-SAT is in **NL** (Theorem 8.11.1) is found in Papadimitriou [234].

Karp [158] exhibited a **PSPACE**-complete problem, Meyer and Stockmeyer [315] demonstrated that QUANTIFIED SATISFIABILITY is **PSPACE**-complete and Schäfer established that GENERALIZED GEOGRAPHY is **PSPACE**-complete [298].

The notion of a uniform circuit was introduced by Borodin [52] and has been examined by many others. (See [119].) Borodin [52] established the connection between nondeterministic

space and circuit depth stated in Theorem 8.13.3. Stockmeyer and Vishkin [316] show how to simulate efficiently the PRAM with circuits and vice versa. (See also [160].) The class **NC** was defined by Cook [76]. Theorem 8.15.2 is due to Pippenger [248]. The class **P/poly** and Theorem 8.15.2 are due to Karp and Lipton [159].

A large variety of parallel computational models have been developed. (See van Embde Boas [349] and Greenlaw, Hoover, and Ruzzo [119].) The PRAM was introduced by Fortune and Wyllie [102] and Goldschlager [117,118].

Several problems on the efficient simulation of RAMs are from Papadimitriou [234].