- A finite transition table δ, which comprises tuples of the form

$$(q, s_0, s_1, \ldots, s_k, q', s'_1, \ldots, s'_k, d_0, d_1, \ldots, d_k)$$

where $q, q' \in Q$, each $s_i, s'_i \in \Sigma \cup \{\Box\}$, and each $d_i \in \{-1, 0, +1\}$.

A tuple specifies a step of M: if the current state is q, and $s_0, s_1, \ldots, s_k$ are the symbols in the cells scanned by the access heads, then M replaces s_i by s'_i for $i = 1, \ldots, k$ simultaneously, changes state to q', and moves the head on tape i one cell to the left ($d_i = -1$) or right ($d_i = +1$) or not at all ($d_i = 0$) for $i = 0, \ldots, k$. Note that M cannot write on tape 0, that is, M can write only on the worktapes, not on the input tape.

- In a tuple, no s'_i can be the blank symbol $\Box$. Because M may not write a blank, the worktape cells that its access heads previously visited are nonblank.
- No tuple contains q_A or q_R as its first component. Thus, once M enters state q_A or state q_R, it stops.
- Initially, M is in state q_0, an input word in Σ^* is inscribed on contiguous cells of the input tape, the access head on the input tape is on the leftmost symbol of the input word, and all other cells of all tapes contain the blank symbol $\Box$.

The Turing machine M that we have defined is *nondeterministic:* δ may have several tuples with the same combination of state q and symbols $s_0, s_1, \ldots, s_k$ as the first $k+2$ components, so that M may have several possible next steps. A machine M is *deterministic* if for every combination of state q and symbols $s_0, s_1, \ldots, s_k$, at most one tuple in δ contains the combination as its first $k+2$ components. A deterministic machine always has at most one possible next step.

A *configuration* of a Turing machine M specifies the current state, the contents of all tapes, and the positions of all access heads.

A *computation path* is a sequence of configurations $C_0, C_1, \ldots, C_t, \ldots$, where C_0 is the initial configuration of M, and each C_{j+1} follows from C_j in one step by applying the changes specified by a tuple in δ. If no tuple is applicable to C_t, then C_t is *terminal*, and the computation path is *halting*. If M has no infinite computation paths, then M *always halts*.

A halting computation path is *accepting* if the state in the last configuration C_t is q_A; otherwise it is *rejecting*. By adding tuples to the program if needed, we can ensure that every rejecting computation ends in state q_R. This leaves the question of computation paths that do not halt. In complexity theory, we rule this out by considering only machines whose computation paths *always halt*. M *accepts* an input word x if there exists an accepting computation path that starts from the initial configuration in which x is on the input tape. For nondeterministic M, it does not matter if some other computation paths end at q_R. If M is deterministic, then there is at most one halting computation path, hence at most one accepting path.

The *language accepted by* M, written $L(M)$, is the set of words accepted by M. If $A = L(M)$, and M always halts, then M *decides* A.

In addition to deciding languages, deterministic Turing machines can compute functions. Designate tape 1 to be the *output tape*. If M halts on input word x, then the nonblank word on tape 1 in the final configuration is the output of M. A function f is *total recursive* if there exists a deterministic Turing machine M that always halts such that for each input word x, the output of M is the value of $f(x)$.

Almost all results in complexity theory are insensitive to minor variations in the underlying computational models. For example, we could have chosen Turing machines whose tapes are restricted to be only one-way infinite or whose alphabet is restricted to $\{\mathbf{0}, \mathbf{1}\}$. It is straightforward to simulate a Turing machine as defined by one of these restricted Turing machines, one step at a time: each step of the original machine can be simulated by $O(1)$ steps of the restricted machine.

5.2.3 Universal Turing Machines

Chapter 6 states that there exists a *universal Turing machine* U, which takes as input a string $\langle M, x \rangle$ that encodes a Turing machine M and a word x, and simulates the operation of M on x, and U accepts $\langle M, x \rangle$ if and only if M accepts x. A theorem of Hennie and Stearns [1966] implies that the machine U can be

constructed to have only two worktapes, such that U can simulate any t steps of M in only $O(t \log t)$ steps of its own, using only $O(1)$ times the worktape cells used by M. The constants implicit in these big-O bounds may depend on M.

We can think of U with a fixed M as a machine U_M and define $L(U_M) = \{x : U \text{ accepts } \langle M, x\rangle\}$. Then $L(U_M) = L(M)$. If M always halts, then U_M always halts; and if M is deterministic, then U_M is deterministic.

5.2.4 Alternating Turing Machines

By definition, a nondeterministic Turing machine M accepts its input word x if there exists an accepting computation path, starting from the initial configuration with x on the input tape. Let us call a *configuration* C accepting if there is a computation path of M that starts in C and ends in a configuration whose state is q_A. Equivalently, a configuration C is accepting if either the state in C is q_A or there exists an accepting configuration C' reachable from C by one step of M. Then M accepts x if the initial configuration with input word x is accepting.

The *alternating Turing machine* generalizes this notion of acceptance. In an alternating Turing machine M, each state is labeled either existential or universal. (Do not confuse the universal state in an alternating Turing machine with the universal Turing machine.) A nonterminal configuration C is existential (respectively, universal) if the state in C is labeled existential (universal). A terminal configuration is accepting if its state is q_A. A nonterminal existential configuration C is accepting if there *exists* an accepting configuration C' reachable from C by one step of M. A nonterminal universal configuration C is accepting if for *every* configuration C' reachable from C by one step of M, the configuration C' is accepting. Finally, M accepts x if the initial configuration with input word x is an accepting configuration.

A nondeterministic Turing machine is thus a special case of an alternating Turing machine in which every state is existential.

The computation of an alternating Turing machine M alternates between existential states and universal states. Intuitively, from an existential configuration, M guesses a step that leads toward acceptance; from a universal configuration, M checks whether each possible next step leads toward acceptance — in a sense, M checks all possible choices in parallel. An alternating computation captures the essence of a two-player game: player 1 has a winning strategy if there exists a move for player 1 such that for every move by player 2, there exists a subsequent move by player 1, etc., such that player 1 eventually wins.

5.2.5 Oracle Turing Machines

Some computational problems remain difficult even when solutions to instances of a particular, different decision problem are available for free. When we study the complexity of a problem *relative* to a language A, we assume that answers about membership in A have been precomputed and stored in a (possibly infinite) table and that there is no cost to obtain an answer to a membership query: Is w in A? The language A is called an **oracle**. Conceptually, an algorithm queries the oracle whether a word w is in A, and it receives the correct answer in one step.

An *oracle Turing machine* is a Turing machine M with a special *oracle tape* and special states QUERY, YES, and NO. The computation of the oracle Turing machine M^A, with oracle language A, is the same as that of an ordinary Turing machine, except that when M enters the QUERY state with a word w on the oracle tape, in one step, M enters either the YES state if $w \in A$ or the NO state if $w \notin A$. Furthermore, during this step, the oracle tape is erased, so that the time for setting up each query is accounted for separately.

5.3 Resources and Complexity Classes

In this section, we define the measures of difficulty of solving computational problems. We introduce complexity classes, which enable us to classify problems according to the difficulty of their solution.

5.3.1 Time and Space

We measure the difficulty of a computational problem by the running time and the space (memory) requirements of an algorithm that solves the problem. Clearly, in general, a finite algorithm cannot have a table of all answers to infinitely many instances of the problem, although an algorithm could look up precomputed answers to a finite number of instances; in terms of Turing machines, the finite answer table is built into the set of states and the transition table. For these instances, the running time is negligible — just the time needed to read the input word. Consequently, our complexity measure should consider a whole problem, not only specific instances.

We express the complexity of a problem, in terms of the growth of the required time or space, as a function of the length n of the input word that encodes a problem instance. We consider the worst-case complexity, that is, for each n, the maximum time or space required among all inputs of length n.

Let M be a Turing machine that always halts. The *time* taken by M on input word x, denoted by $\text{Time}_M(x)$, is defined as follows:

- If M accepts x, then $\text{Time}_M(x)$ is the number of steps in the shortest accepting computation path for x.
- If M rejects x, then $\text{Time}_M(x)$ is the number of steps in the longest computation path for x.

For a deterministic machine M, for every input x, there is at most one halting computation path, and its length is $\text{Time}_M(x)$. For a nondeterministic machine M, if $x \in L(M)$, then M can guess the correct steps to take toward an accepting configuration, and $\text{Time}_M(x)$ measures the length of the path on which M always makes the best guess.

The *space* used by a Turing machine M on input x, denoted by $\text{Space}_M(x)$, is defined as follows. The space used by a halting computation path is the number of nonblank worktape cells in the last configuration; this is the number of different cells ever written by the worktape heads of M during the computation path, since M never writes the blank symbol. Because the space occupied by the input word is not counted, a machine can use a sublinear ($o(n)$) amount of space.

- If M accepts x, then $\text{Space}_M(x)$ is the minimum space used among all accepting computation paths for x.
- If M rejects x, then $\text{Space}_M(x)$ is the maximum space used among all computation paths for x.

The **time complexity** of a machine M is the function

$$t(n) = \max\{\text{Time}_M(x) : |x| = n\}$$

We assume that M reads all of its input word, and the blank symbol after the right end of the input word, so $t(n) \geq n + 1$. The **space complexity** of M is the function

$$s(n) = \max\{\text{Space}_M(x) : |x| = n\}$$

Because few interesting languages can be decided by machines of sublogarithmic space complexity, we henceforth assume that $s(n) \geq \log n$.

A function $f(x)$ is *computable in polynomial time* if there exists a deterministic Turing machine M of polynomial time complexity such that for each input word x, the output of M is $f(x)$.

5.3.2 Complexity Classes

Having defined the time complexity and space complexity of individual Turing machines, we now define classes of languages with particular complexity bounds. These definitions will lead to definitions of P and NP.

Let $t(n)$ and $s(n)$ be numeric functions. Define the following classes of languages:

- $\mathsf{DTIME}[t(n)]$ is the class of languages decided by deterministic Turing machines of time complexity $O(t(n))$.
- $\mathsf{NTIME}[t(n)]$ is the class of languages decided by nondeterministic Turing machines of time complexity $O(t(n))$.
- $\mathsf{DSPACE}[s(n)]$ is the class of languages decided by deterministic Turing machines of space complexity $O(s(n))$.
- $\mathsf{NSPACE}[s(n)]$ is the class of languages decided by nondeterministic Turing machines of space complexity $O(s(n))$.

We sometimes abbreviate $\mathsf{DTIME}[t(n)]$ to $\mathsf{DTIME}[t]$ (and so on) when t is understood to be a function, and when no reference is made to the input length n.

The following are the **canonical complexity classes**:

- $\mathsf{L} = \mathsf{DSPACE}[\log n]$ (deterministic log space)
- $\mathsf{NL} = \mathsf{NSPACE}[\log n]$ (nondeterministic log space)
- $\mathsf{P} = \mathsf{DTIME}[n^{O(1)}] = \bigcup_{k\geq 1} \mathsf{DTIME}[n^k]$ (polynomial time)
- $\mathsf{NP} = \mathsf{NTIME}[n^{O(1)}] = \bigcup_{k\geq 1} \mathsf{NTIME}[n^k]$ (nondeterministic polynomial time)
- $\mathsf{PSPACE} = \mathsf{DSPACE}[n^{O(1)}] = \bigcup_{k\geq 1} \mathsf{DSPACE}[n^k]$ (polynomial space)
- $\mathsf{E} = \mathsf{DTIME}[2^{O(n)}] = \bigcup_{k\geq 1} \mathsf{DTIME}[k^n]$
- $\mathsf{NE} = \mathsf{NTIME}[2^{O(n)}] = \bigcup_{k\geq 1} \mathsf{NTIME}[k^n]$
- $\mathsf{EXP} = \mathsf{DTIME}[2^{n^{O(1)}}] = \bigcup_{k\geq 1} \mathsf{DTIME}[2^{n^k}]$ (deterministic exponential time)
- $\mathsf{NEXP} = \mathsf{NTIME}[2^{n^{O(1)}}] = \bigcup_{k\geq 1} \mathsf{NTIME}[2^{n^k}]$ (nondeterministic exponential time)

The space classes L and PSPACE are defined in terms of the DSPACE complexity measure. By Savitch's Theorem (see Theorem 5.2), the NSPACE measure with polynomial bounds also yields PSPACE.

The class P contains many familiar problems that can be solved efficiently, such as (decision problem versions of) finding shortest paths in networks, parsing for context-free languages, sorting, matrix multiplication, and linear programming. Consequently, P has become accepted as representing the set of computationally feasible problems. Although one could legitimately argue that a problem whose best algorithm has time complexity $\Theta(n^{99})$ is really infeasible, in practice, the time complexities of the vast majority of known polynomial-time algorithms have low degrees: they run in $O(n^4)$ time or less. Moreover, P is a robust class: although defined by Turing machines, P remains the same when defined by other models of sequential computation. For example, random access machines (RAMs) (a more realistic model of computation defined in Chapter 6) can be used to define P because Turing machines and RAMs can simulate each other with polynomial-time overhead.

The class NP can also be defined by means other than nondeterministic Turing machines. NP equals the class of problems whose solutions can be *verified* quickly, by deterministic machines in polynomial time. Equivalently, NP comprises those languages whose membership proofs can be checked quickly.

For example, one language in NP is the set of satisfiable Boolean formulas, called SAT. A Boolean formula ϕ is satisfiable if there exists a way of assigning `true` or `false` to each variable such that under this truth assignment, the value of ϕ is `true`. For example, the formula $x \wedge (\overline{x} \vee y)$ is satisfiable, but $x \wedge \overline{y} \wedge (\overline{x} \vee y)$ is not satisfiable. A nondeterministic Turing machine M, after checking the syntax of ϕ and counting the number n of variables, can nondeterministically write down an n-bit 0-1 string a on its tape, and then deterministically (and easily) evaluate ϕ for the truth assignment denoted by a. The computation path corresponding to each individual a accepts if and only if $\phi(a) =$ `true`, and so M itself accepts ϕ if and only if ϕ is satisfiable; that is, $L(M) = \mathrm{SAT}$. Again, this checking of given assignments differs significantly from trying to *find* an accepting assignment.

Another language in NP is the set of undirected graphs with a *Hamiltonian circuit*, that is, a path of edges that visits each vertex exactly once and returns to the starting point. If a solution exists and is given, its

correctness can be verified quickly. Finding such a circuit, however, or proving one does not exist, appears to be computationally difficult.

The characterization of NP as the set of problems with easily verified solutions is formalized as follows: $A \in \mathsf{NP}$ if and only if there exist a language $A' \in \mathsf{P}$ and a polynomial p such that for every x, $x \in A$ if and only if there exists a y such that $|y| \leq p(|x|)$ and $(x, y) \in A'$. Here, whenever x belongs to A, y is interpreted as a positive solution to the problem represented by x, or equivalently, as a proof that x belongs to A. The difference between P and NP is that between solving and checking, or between finding a proof of a mathematical theorem and testing whether a candidate proof is correct. In essence, NP represents all sets of theorems with proofs that are short (i.e., of polynomial length) and checkable quickly (i.e., in polynomial time), while P represents those statements that can proved or refuted quickly from scratch.

Further motivation for studying L, NL, and PSPACE comes from their relationships to P and NP. Namely, L and NL are the largest space-bounded classes known to be contained in P, and PSPACE is the smallest space-bounded class known to contain NP. (It is worth mentioning here that NP does not stand for "non-polynomial time"; the class P is a subclass of NP.) Similarly, EXP is of interest primarily because it is the smallest deterministic time class known to contain NP. The closely related class E is not known to contain NP.

5.4 Relationships between Complexity Classes

The P versus NP question asks about the relationship between these complexity classes: Is P a proper subset of NP, or does $\mathsf{P} = \mathsf{NP}$? Much of complexity theory focuses on the relationships between complexity classes because these relationships have implications for the difficulty of solving computational problems. In this section, we summarize important known relationships. We demonstrate two techniques for proving relationships between classes: diagonalization and padding.

5.4.1 Constructibility

The most basic theorem that one should expect from complexity theory would say, "If you have more resources, you can do more." Unfortunately, if we are not careful with our definitions, then this claim is false:

Theorem 5.1 (Gap Theorem) *There is a computable, strictly increasing time bound $t(n)$ such that* $\mathsf{DTIME}[t(n)] = \mathsf{DTIME}[2^{2^{t(n)}}]$ [Borodin, 1972].

That is, there is an empty gap between time $t(n)$ and time doubly-exponentially greater than $t(n)$, in the sense that anything that can be computed in the larger time bound can already be computed in the smaller time bound. That is, even with much more time, you can not compute more. This gap can be made much larger than doubly-exponential; for any computable r, there is a computable time bound t such that $\mathsf{DTIME}[t(n)] = \mathsf{DTIME}[r(t(n))]$. Exactly analogous statements hold for the NTIME, DSPACE, and NSPACE measures.

Fortunately, the gap phenomenon cannot happen for time bounds t that anyone would ever be interested in. Indeed, the proof of the Gap Theorem proceeds by showing that one can define a time bound t such that no machine has a running time that is between $t(n)$ and $2^{2^{t(n)}}$. This theorem indicates the need for formulating only those time bounds that actually describe the complexity of some machine.

A function $t(n)$ is **time-constructible** if there exists a deterministic Turing machine that halts after exactly $t(n)$ steps for every input of length n. A function $s(n)$ is **space-constructible** if there exists a deterministic Turing machine that uses exactly $s(n)$ worktape cells for every input of length n. (Most authors consider only functions $t(n) \geq n + 1$ to be time-constructible, and many limit attention to $s(n) \geq \log n$ for space bounds. There do exist sub-logarithmic space-constructible functions, but we prefer to avoid the tricky theory of $o(\log n)$ space bounds.)

For example, $t(n) = n + 1$ is time-constructible. Furthermore, if $t_1(n)$ and $t_2(n)$ are time-constructible, then so are the functions $t_1 + t_2$, $t_1 t_2$, $t_1^{t_2}$, and c^{t_1} for every integer $c > 1$. Consequently, if $p(n)$ is a polynomial, then $p(n) = \Theta(t(n))$ for some time-constructible polynomial function $t(n)$. Similarly, $s(n) = \log n$ is space-constructible, and if $s_1(n)$ and $s_2(n)$ are space-constructible, then so are the functions $s_1 + s_2$, $s_1 s_2$, $s_1^{s_2}$, and c^{s_1} for every integer $c > 1$. Many common functions are space-constructible: for example, $n \log n$, n^3, 2^n, $n!$.

Constructibility helps eliminate an arbitrary choice in the definition of the basic time and space classes. For general time functions t, the classes $\mathsf{DTIME}[t]$ and $\mathsf{NTIME}[t]$ may vary depending on whether machines are required to halt within t steps on all computation paths, or just on those paths that accept. If t is time-constructible and s is space-constructible, however, then $\mathsf{DTIME}[t]$, $\mathsf{NTIME}[t]$, $\mathsf{DSPACE}[s]$, and $\mathsf{NSPACE}[s]$ can be defined without loss of generality in terms of Turing machines that always halt.

As a general rule, any function $t(n) \geq n + 1$ and any function $s(n) \geq \log n$ that one is interested in as a time or space bound, is time- or space-constructible, respectively. As we have seen, little of interest can be proved without restricting attention to constructible functions. This restriction still leaves a rich class of resource bounds.

5.4.2 Basic Relationships

Clearly, for all time functions $t(n)$ and space functions $s(n)$, $\mathsf{DTIME}[t(n)] \subseteq \mathsf{NTIME}[t(n)]$ and $\mathsf{DSPACE}[s(n)] \subseteq \mathsf{NSPACE}[s(n)]$ because a deterministic machine is a special case of a nondeterministic machine. Furthermore, $\mathsf{DTIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$ and $\mathsf{NTIME}[t(n)] \subseteq \mathsf{NSPACE}[t(n)]$ because at each step, a k-tape Turing machine can write on at most $k = O(1)$ previously unwritten cells. The next theorem presents additional important relationships between classes.

Theorem 5.2 *Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function, $s(n) \geq \log n$.*

(a) $\mathsf{NTIME}[t(n)] \subseteq \mathsf{DTIME}[2^{O(t(n))}]$
(b) $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DTIME}[2^{O(s(n))}]$
(c) $\mathsf{NTIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$
(d) **(Savitch's Theorem)** $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DSPACE}[s(n)^2]$ [Savitch, 1970]

As a consequence of the first part of this theorem, $\mathsf{NP} \subseteq \mathsf{EXP}$. No better general upper bound on deterministic time is known for languages in NP, however. See Figure 5.2 for other known inclusion relationships between canonical complexity classes.

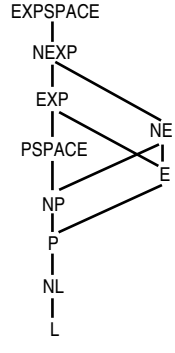


FIGURE 5.2 Inclusion relationships between the canonical complexity classes.

Although we do not know whether allowing nondeterminism strictly increases the class of languages decided in polynomial time, Savitch's Theorem says that for space classes, nondeterminism does not help by more than a polynomial amount.

5.4.3 Complementation

For a language A over an alphabet Σ, define $\overline{A}$ to be the complement of A in the set of words over Σ: that is, $\overline{A} = \Sigma^* - A$. For a class of languages $\mathcal{C}$, define co-$\mathcal{C} = \{ \overline{A} : A \in \mathcal{C} \}$. If $\mathcal{C} =$ co-$\mathcal{C}$, then $\mathcal{C}$ is **closed under complementation.**

In particular, co-NP is the class of languages that are complements of languages in NP. For the language SAT of satisfiable Boolean formulas, $\overline{\text{SAT}}$ is essentially the set of unsatisfiable formulas, whose value is `false` for every truth assignment, together with the syntactically incorrect formulas. A closely related language in co-NP is the set of Boolean tautologies, namely, those formulas whose value is `true` for every truth assignment. The question of whether NP equals co-NP comes down to whether every tautology has a short (i.e., polynomial-sized) proof. The only obvious general way to prove a tautology ϕ in m variables is to verify all 2^m rows of the truth table for ϕ, taking exponential time. Most complexity theorists believe that there is no general way to reduce this time to polynomial, hence that NP $\neq$ co-NP.

Questions about complementation bear directly on the P vs. NP question. It is easy to show that P is closed under complementation (see the next theorem). Consequently, if NP $\neq$ co-NP, then P $\neq$ NP.

Theorem 5.3 (Complementation Theorems) *Let t be a time-constructible function, and let s be a space-constructible function, with $s(n) \geq \log n$ for all n. Then,*

1. DTIME$[t]$ *is closed under complementation.*
2. DSPACE$[s]$ *is closed under complementation.*
3. NSPACE$[s]$ *is closed under complementation* [Immerman, 1988; Szelepcsényi, 1988].

The Complementation Theorems are used to prove the Hierarchy Theorems in the next section.

5.4.4 Hierarchy Theorems and Diagonalization

A hierarchy theorem is a theorem that says, "If you have more resources, you can compute more." As we saw in Section 5.4.1, this theorem is possible only if we restrict attention to constructible time and space bounds. Next, we state hierarchy theorems for deterministic and nondeterministic time and space classes. In the following, $\subset$ denotes *strict* inclusion between complexity classes.

Theorem 5.4 (Hierarchy Theorems) *Let t_1 and t_2 be time-constructible functions, and let s_1 and s_2 be space-constructible functions, with $s_1(n), s_2(n) \geq \log n$ for all n.*

(a) *If* $t_1(n) \log t_1(n) = o(t_2(n))$, *then* DTIME$[t_1] \subset$ DTIME$[t_2]$.
(b) *If* $t_1(n+1) = o(t_2(n))$, *then* NTIME$[t_1] \subset$ NTIME$[t_2]$ [Seiferas et al., 1978].
(c) *If* $s_1(n) = o(s_2(n))$, *then* DSPACE$[s_1] \subset$ DSPACE$[s_2]$.
(d) *If* $s_1(n) = o(s_2(n))$, *then* NSPACE$[s_1] \subset$ NSPACE$[s_2]$.

As a corollary of the Hierarchy Theorem for DTIME,

$$\text{P} \subseteq \text{DTIME}[n^{\log n}] \subset \text{DTIME}[2^n] \subseteq \text{E};$$

hence, we have the strict inclusion P $\subset$ E. Although we do not know whether P $\subset$ NP, there exists a problem in E that cannot be solved in polynomial time. Other consequences of the Hierarchy Theorems are NE $\subset$ NEXP and NL $\subset$ PSPACE.

In the Hierarchy Theorem for DTIME, the hypothesis on t_1 and t_2 is $t_1(n)\log t_1(n) = o(t_2(n))$, instead of $t_1(n) = o(t_2(n))$, for technical reasons related to the simulation of machines with multiple worktapes by a single universal Turing machine with a fixed number of worktapes. Other computational models, such as random access machines, enjoy tighter time hierarchy theorems.

All proofs of the Hierarchy Theorems use the technique of **diagonalization**. For example, the proof for DTIME constructs a Turing machine M of time complexity t_2 that considers all machines $M_1, M_2, \ldots$ whose time complexity is t_1; for each i, the proof finds a word x_i that is accepted by M if and only if $x_i \notin L(M_i)$, the language decided by M_i. Consequently, $L(M)$, the language decided by M, differs from each $L(M_i)$, hence $L(M) \notin \mathsf{DTIME}[t_1]$. The diagonalization technique resembles the classic method used to prove that the real numbers are uncountable, by constructing a number whose j^{th} digit differs from the j^{th} digit of the j^{th} number on the list. To illustrate the diagonalization technique, we outline the proof of the Hierarchy Theorem for DSPACE. In this subsection, $\langle i, x\rangle$ stands for the string $0^i 1x$, and $zeroes(y)$ stands for the number of 0's that a given string y starts with. Note that $zeroes(\langle i, x\rangle) = i$.

Proof (of the DSPACE Hierarchy Theorem)
We construct a deterministic Turing machine M that decides a language A such that $A \in \mathsf{DSPACE}[s_2] - \mathsf{DSPACE}[s_1]$.

Let U be a deterministic universal Turing machine, as described in Section 5.2.3. On input x of length n, machine M performs the following:

1. Lay out $s_2(n)$ cells on a worktape.
2. Let $i = zeroes(x)$.
3. Simulate the universal machine U on input $\langle i, x\rangle$. Accept x if U tries to use more than s_2 worktape cells. (We omit some technical details, and the way in which the constructibility of s_2 is used to ensure that this process halts.)
4. If U accepts $\langle i, x\rangle$, then reject; if U rejects $\langle i, x\rangle$, then accept.

Clearly, M always halts and uses space $O(s_2(n))$. Let $A = L(M)$.

Suppose $A \in \mathsf{DSPACE}[s_1(n)]$. Then there is some Turing machine M_j accepting A using space at most $s_1(n)$. Since the space used by U is $O(1)$ times the space used by M_j, there is a constant k depending only on j (in fact, we can take $k = |j|$), such that U, on inputs z of the form $z = \langle j, x\rangle$, uses at most $ks_1(|x|)$ space.

Since $s_1(n) = o(s_2(n))$, there is an n_0 such that $ks_1(n) \leq s_2(n)$ for all $n \geq n_0$. Let x be a string of length greater than n_0 such that the first $j+1$ symbols of x are $0^j 1$. Note that the universal Turing machine U, on input $\langle j, x\rangle$, simulates M_j on input x and uses space at most $ks_1(n) \leq s_2(n)$. Thus, when we consider the machine M defining A, we see that on input x the simulation does not stop in step 3, but continues on to step 4, and thus $x \in A$ if and only if U rejects $\langle j, x\rangle$. Consequently, M_j does not accept A, contrary to our assumption. Thus, $A \notin \mathsf{DSPACE}[s_1(n)]$. □

Although the diagonalization technique successfully separates some pairs of complexity classes, diagonalization does not seem strong enough to separate P from NP. (See Theorem 5.10 below.)

5.4.5 Padding Arguments

A useful technique for establishing relationships between complexity classes is the **padding argument.** Let A be a language over alphabet Σ, and let # be a symbol not in Σ. Let f be a numeric function. The **f-padded version of** L is the language

$$A' = \{x\#^{f(n)} : x \in A \text{ and } n = |x|\}.$$

That is, each word of A' is a word in A concatenated with $f(n)$ consecutive # symbols. The padded version A' has the same information content as A, but because each word is longer, the computational complexity of A' is smaller.

The proof of the next theorem illustrates the use of a padding argument.

Theorem 5.5 *If* $\mathsf{P} = \mathsf{NP}$, *then* $\mathsf{E} = \mathsf{NE}$ [Book, 1974].

Proof Since $\mathsf{E} \subseteq \mathsf{NE}$, we prove that $\mathsf{NE} \subseteq \mathsf{E}$.
Let $A \in \mathsf{NE}$ be decided by a nondeterministic Turing machine M in at most $t(n) = k^n$ time for some constant integer k. Let A' be the $t(n)$-padded version of A. From M, we construct a nondeterministic Turing machine M' that decides A' in linear time: M' checks that its input has the correct format, using the time-constructibility of t; then M' runs M on the prefix of the input preceding the first # symbol. Thus, $A' \in \mathsf{NP}$.

If $\mathsf{P} = \mathsf{NP}$, then there is a deterministic Turing machine D' that decides A' in at most $p'(n)$ time for some polynomial p'. From D', we construct a deterministic Turing machine D that decides A, as follows. On input x of length n, since $t(n)$ is time-constructible, machine D constructs $x\#^{t(n)}$, whose length is $n + t(n)$, in $O(t(n))$ time. Then D runs D' on this input word. The time complexity of D is at most $O(t(n)) + p'(n + t(n)) = 2^{O(n)}$. Therefore, $\mathsf{NE} \subseteq \mathsf{E}$. □

A similar argument shows that the $\mathsf{E} = \mathsf{NE}$ question is equivalent to the question of whether $\mathsf{NP} - \mathsf{P}$ contains a subset of 1^*, that is, a language over a single-letter alphabet.

5.5 Reducibility and Completeness

In this section, we discuss relationships between problems: informally, if one problem reduces to another problem, then in a sense, the second problem is harder than the first. The hardest problems in NP are the NP-complete problems. We define NP-completeness precisely, and we show how to prove that a problem is NP-complete. The theory of NP-completeness, together with the many known NP-complete problems, is perhaps the best justification for interest in the classes P and NP. All of the other canonical complexity classes listed above have natural and important problems that are complete for them; we give some of these as well.

5.5.1 Resource-Bounded Reducibilities

In mathematics, as in everyday life, a typical way to solve a new problem is to reduce it to a previously solved problem. Frequently, an instance of the new problem is expressed completely in terms of an instance of the prior problem, and the solution is then interpreted in the terms of the new problem. For example, the maximum weighted matching problem for bipartite graphs (also called the assignment problem) reduces to the network flow problem (see Chapter 7). This kind of reduction is called **many-one reducibility**, and is defined below.

A different way to solve the new problem is to use a subroutine that solves the prior problem. For example, we can solve an optimization problem whose solution is feasible and maximizes the value of an objective function g by repeatedly calling a subroutine that solves the corresponding decision problem of whether there exists a feasible solution x whose value $g(x)$ satisfies $g(x) \geq k$. This kind of reduction is called **Turing reducibility**, and is also defined below.

Let A_1 and A_2 be languages. A_1 is many-one reducible to A_2, written $A_1 \leq_m A_2$, if there exists a total recursive function f such that for all x, $x \in A_1$ if and only if $f(x) \in A_2$. The function f is called the **transformation function.** A_1 is Turing reducible to A_2, written $A_1 \leq_T A_2$, if A_1 can be decided by a deterministic oracle Turing machine M using A_2 as its oracle, that is, $A_1 = L(M^{A_2})$. (Total recursive functions and oracle Turing machines are defined in Section 5.2). The oracle for A_2 models a hypothetical efficient subroutine for A_2.

If f or M above consumes too much time or space, the reductions they compute are not helpful. To study complexity classes defined by bounds on time and space resources, it is natural to consider resource-bounded reducibilities. Let A_1 and A_2 be languages.

- A_1 is **Karp reducible** to A_2, written $A_1 \leq_m^p A_2$, if A_1 is many-one reducible to A_2 via a transformation function that is computable deterministically in polynomial time.
- A_1 is **log-space reducible** to A_2, written $A_1 \leq_m^{\log} A_2$, if A_1 is many-one reducible to A_2 via a transformation function that is computable deterministically in $O(\log n)$ space.
- A_1 is **Cook reducible** to A_2, written $A_1 \leq_T^p A_2$, if A_1 is Turing reducible to A_2 via a deterministic oracle Turing machine of polynomial time complexity.

The term "polynomial-time reducibility" usually refers to Karp reducibility. If $A_1 \leq_m^p A_2$ and $A_2 \leq_m^p A_1$, then A_1 and A_2 are **equivalent** under Karp reducibility. Equivalence under Cook reducibility is defined similarly.

Karp and Cook reductions are useful for finding relationships between languages of high complexity, but they are not at all useful for distinguishing between problems in P, because all problems in P are equivalent under Karp (and hence Cook) reductions. (Here and later we ignore the special cases $A_1 = \emptyset$ and $A_1 = \Sigma^*$, and consider them to reduce to any language.)

Log-space reducibility [Jones, 1975] is useful for complexity classes within P, such as NL, for which Karp reducibility allows too many reductions. By definition, for every nontrivial language A_0 (i.e., $A_0 \neq \emptyset$ and $A_0 \neq \Sigma^*$) and for every A in P, necessarily $A \leq_m^p A_0$ via a transformation that simply runs a deterministic Turing machine that decides A in polynomial time. It is not known whether log-space reducibility is different from Karp reducibility, however; all transformations for known Karp reductions can be computed in $O(\log n)$ space. Even for decision problems, L is not known to be a proper subset of P.

Theorem 5.6 *Log-space reducibility implies Karp reducibility, which implies Cook reducibility:*

1. *If* $A_1 \leq_m^{\log} A_2$, *then* $A_1 \leq_m^p A_2$.
2. *If* $A_1 \leq_m^p A_2$, *then* $A_1 \leq_T^p A_2$.

Theorem 5.7 *Log-space reducibility, Karp reducibility, and Cook reducibility are transitive:*

1. *If* $A_1 \leq_m^{\log} A_2$ *and* $A_2 \leq_m^{\log} A_3$, *then* $A_1 \leq_m^{\log} A_3$.
2. *If* $A_1 \leq_m^p A_2$ *and* $A_2 \leq_m^p A_3$, *then* $A_1 \leq_m^p A_3$.
3. *If* $A_1 \leq_T^p A_2$ *and* $A_2 \leq_T^p A_3$, *then* $A_1 \leq_T^p A_3$.

The key property of Cook and Karp reductions is that they preserve polynomial-time feasibility. Suppose $A_1 \leq_m^p A_2$ via a transformation f. If M_2 decides A_2, and M_f computes f, then to decide whether an input word x is in A_1, we can use M_f to compute $f(x)$, and then run M_2 on input $f(x)$. If the time complexities of M_2 and M_f are bounded by polynomials t_2 and t_f, respectively, then on each input x of length $n = |x|$, the time taken by this method of deciding A_1 is at most $t_f(n) + t_2(t_f(n))$, which is also a polynomial in n. In summary, if A_2 is feasible, and there is an efficient reduction from A_1 to A_2, then A_1 is feasible. Although this is a simple observation, this fact is important enough to state as a theorem (Theorem 5.8). First, however, we need the concept of "closure."

A class of languages $\mathcal{C}$ is **closed under a reducibility** $\leq_r$ if for all languages A_1 and A_2, whenever $A_1 \leq_r A_2$ and $A_2 \in \mathcal{C}$, necessarily $A_1 \in \mathcal{C}$.

Theorem 5.8

1. P *is closed under log-space reducibility, Karp reducibility, and Cook reducibility.*
2. NP *is closed under log-space reducibility and Karp reducibility.*
3. L *and* NL *are closed under log-space reducibility.*

We shall see the importance of closure under a reducibility in conjunction with the concept of completeness, which we define in the next section.

5.5.2 Complete Languages

Let $\mathcal{C}$ be a class of languages that represent computational problems. A language A_0 is **$\mathcal{C}$-hard** under a reducibility $\leq_r$ if for all A in $\mathcal{C}$, $A \leq_r A_0$. A language A_0 is **$\mathcal{C}$-complete** under $\leq_r$ if A_0 is $\mathcal{C}$-hard and $A_0 \in \mathcal{C}$. Informally, if A_0 is $\mathcal{C}$-hard, then A_0 represents a problem that is at least as difficult to solve as any problem in $\mathcal{C}$. If A_0 is $\mathcal{C}$-complete, then in a sense, A_0 is one of the most difficult problems in $\mathcal{C}$.

There is another way to view completeness. Completeness provides us with tight lower bounds on the complexity of problems. If a language A is complete for complexity class $\mathcal{C}$, then we have a lower bound on its complexity. Namely, A is as hard as the most difficult problem in $\mathcal{C}$, assuming that the complexity of the reduction itself is small enough not to matter. The lower bound is tight because A is in $\mathcal{C}$; that is, the upper bound matches the lower bound.

In the case $\mathcal{C} = \mathsf{NP}$, the reducibility $\leq_r$ is usually taken to be Karp reducibility unless otherwise stated. Thus, we say

- A language A_0 is **NP-hard** if A_0 is NP-hard under Karp reducibility.
- A_0 is **NP-complete** if A_0 is NP-complete under Karp reducibility.

However, many sources take the term "NP-hard" to refer to Cook reducibility.

Many important languages are now known to be NP-complete. Before we get to them, let us discuss some implications of the statement "A_0 is NP-complete," and also some things this statement does not mean.

The first implication is that *if* there exists a deterministic Turing machine that decides A_0 in polynomial time — that is, if $A_0 \in \mathsf{P}$ — then because P is closed under Karp reducibility (Theorem 5.8 in Section 5.5.1), it would follow that $\mathsf{NP} \subseteq \mathsf{P}$, hence $\mathsf{P} = \mathsf{NP}$. In essence, the question of whether P is the same as NP comes down to the question of whether any particular NP-complete language is in P. Put another way, *all* of the NP-complete languages stand or fall together: if one is in P, then all are in P; if one is not, then all are not. Another implication, which follows by a similar closure argument applied to co-NP, is that if $A_0 \in \text{co-NP}$, then $\mathsf{NP} = \text{co-NP}$. It is also believed unlikely that $\mathsf{NP} = \text{co-NP}$, as was noted in connection with whether all tautologies have short proofs in Section 5.4.3.

A common misconception is that the above property of NP-complete languages is actually their definition, namely: if $A \in \mathsf{NP}$ and $A \in \mathsf{P}$ implies $\mathsf{P} = \mathsf{NP}$, then A is NP-complete. This "definition" is wrong if $\mathsf{P} \neq \mathsf{NP}$. A theorem due to Ladner [1975] shows that $\mathsf{P} \neq \mathsf{NP}$ if and only if there exists a language A' in $\mathsf{NP} - \mathsf{P}$ such that A' is not NP-complete. Thus, if $\mathsf{P} \neq \mathsf{NP}$, then A' is a counterexample to the "definition."

Another common misconception arises from a misunderstanding of the statement "If A_0 is NP-complete, then A_0 is one of the most difficult problems in NP." This statement is true on one level: if there is any problem at all in NP that is not in P, then the NP-complete language A_0 is one such problem. However, note that there are NP-complete problems in $\mathsf{NTIME}[n]$ — and these problems are, in some sense, much *simpler* than many problems in $\mathsf{NTIME}[n^{10^{500}}]$.

5.5.3 Cook-Levin Theorem

Interest in NP-complete problems started with a theorem of Cook [1971] that was proved independently by Levin [1973]. Recall that SAT is the language of Boolean formulas $\phi(z_1, \ldots, z_r)$ such that there exists a truth assignment to the variables $z_1, \ldots, z_r$ that makes ϕ true.

Theorem 5.9 (Cook-Levin Theorem) SAT *is NP-complete.*

Proof We know already that SAT is in NP, so to prove that SAT is NP-complete, we need to take an arbitrary given language A in NP and show that $A \leq_m^p$ SAT. Take N to be a nondeterministic Turing

machine that decides A in polynomial time. Then the relation $R(x, y) =$ "y is a computation path of N that leads it to accept x" is decidable in deterministic polynomial time depending only on $n = |x|$. We can assume that the length m of possible y's encoded as binary strings depends only on n and not on a particular x.

It is straightforward to show that there is a polynomial p and for each n a Boolean circuit C_n^R with $p(n)$ wires, with $n + m$ input wires labeled $x_1, \ldots, x_n, y_1, \ldots, y_m$ and one output wire w_0, such that $C_n^R(x, y)$ outputs 1 if and only if $R(x, y)$ holds. (We describe circuits in more detail below, and state a theorem for this principle as part 1. of Theorem 5.14.) Importantly, C_n^R itself can be designed in time polynomial in n, and by the universality of NAND, may be composed entirely of binary NAND gates. Label the wires by variables $x_1, \ldots, x_n, y_1, \ldots, y_m, w_0, w_1, \ldots, w_{p(n)-n-m-1}$. These become the variables of our Boolean formulas. For each NAND gate g with input wires u and v, and for each output wire w of g, write down the subformula

$$\phi_{g,w} = (u \vee w) \wedge (v \vee w) \wedge (\bar{u} \vee \bar{v} \vee \bar{w})$$

This subformula is satisfied by precisely those assignments to u, v, w that give $w = u \,\text{NAND}\, v$. The conjunction ϕ_0 of $\phi_{g,w}$ over the polynomially many gates g and their output wires w thus is satisfied only by assignments that set every gate's output correctly given its inputs. Thus, for any binary strings x and y of lengths n, m, respectively, the formula $\phi_1 = \phi_0 \wedge w_0$ is satisfiable by a setting of the wire variables $w_0, w_1, \ldots, w_{p(n)-n-m-1}$ if and only if $C_n^R(x, y) = 1$ — that is, if and only if $R(x, y)$ holds.

Now given any fixed x and taking $n = |x|$, the Karp reduction computes ϕ_1 via C_n^R and ϕ_0 as above, and finally outputs the Boolean formula ϕ obtained by substituting the bit-values of x into ϕ_1. This ϕ has variables $y_1, \ldots, y_m, w_0, w_1, \ldots, w_{p(n)-n-m-1}$, and the computation of ϕ from x runs in deterministic polynomial time. Then $x \in A$ if and only if N accepts x, if and only if there exists y such that $R(x, y)$ holds, if and only if there exists an assignment to the variables $w_0, w_1, \ldots, w_{p(n)-n-m-1}$ *and* $y_1, \ldots, y_m$ that satisfies ϕ, if and only if $\phi \in$ SAT. This shows $A \leq_m^p$ SAT. □

We have actually proved that SAT remains NP-complete even when the given instances ϕ are *restricted* to Boolean formulas that are a conjunction of *clauses*, where each clause consists of (here, at most three) disjuncted literals. Such formulas are said to be in *conjunctive normal form*. Theorem 5.9 is also commonly known as Cook's Theorem.

5.5.4 Proving NP-Completeness

After one language has been proved complete for a class, others can be proved complete by constructing transformations. For NP, if A_0 is NP-complete, then to prove that another language A_1 is NP-complete, it suffices to prove that $A_1 \in$ NP, and to construct a polynomial-time transformation that establishes $A_0 \leq_m^p A_1$. Since A_0 is NP-complete, for every language A in NP, $A \leq_m^p A_0$, hence, by transitivity (Theorem 5.7), $A \leq_m^p A_1$.

Beginning with Cook [1971] and Karp [1972], hundreds of computational problems in many fields of science and engineering have been proved to be NP-complete, almost always by reduction from a problem that was previously known to be NP-complete. The following NP-complete decision problems are frequently used in these reductions — the language corresponding to each problem is the set of instances whose answers are `yes`.

- 3-SATISFIABILITY (3SAT)

  Instance: A Boolean expression ϕ in conjunctive normal form with three literals per clause [e.g., $(w \vee x \vee \overline{y}) \wedge (\overline{x} \vee y \vee z)$].

  Question: Is ϕ satisfiable?

- VERTEX COVER

  Instance: A graph G and an integer k.

  Question: Does G have a set W of k vertices such that every edge in G is incident on a vertex of W?
- CLIQUE

  Instance: A graph G and an integer k.

  Question: Does G have a set K of k vertices such that every two vertices in K are adjacent in G?
- HAMILTONIAN CIRCUIT

  Instance: A graph G.

  Question: Does G have a circuit that includes every vertex exactly once?
- THREE-DIMENSIONAL MATCHING

  Instance: Sets W, X, Y with $|W| = |X| = |Y| = q$ and a subset $S \subseteq W \times X \times Y$.

  Question: Is there a subset $S' \subseteq S$ of size q such that no two triples in S' agree in any coordinate?
- PARTITION

  Instance: A set S of positive integers.

  Question: Is there a subset $S' \subseteq S$ such that the sum of the elements of S' equals the sum of the elements of $S - S'$?

Note that our ϕ in the above proof of the Cook-Levin Theorem already meets a form of the definition of 3SAT relaxed to allow "at most 3 literals per clause." Padding ϕ with some extra variables to bring up the number in each clause to exactly three, while preserving whether the formula is satisfiable or not, is not difficult, and establishes the NP-completeness of 3SAT. Here is another example of an NP-completeness proof, for the following decision problem:

- TRAVELING SALESMAN PROBLEM (TSP)

  Instance: A set of m "cities" $C_1, \ldots, C_m$, with an integer distance $d(i, j)$ between every pair of cities C_i and C_j, and an integer D.

  Question: Is there a tour of the cities whose total length is at most D, that is, a permutation $c_1, \ldots, c_m$ of $\{1, \ldots, m\}$, such that

$$d(c_1, c_2) + \cdots + d(c_{m-1}, c_m) + d(c_m, c_1) \leq D?$$

First, it is easy to see that TSP is in NP: a nondeterministic Turing machine simply guesses a tour and checks that the total length is at most D.

Next, we construct a reduction from Hamiltonian Circuit to TSP. (The reduction goes from the known NP-complete problem, Hamiltonian Circuit, to the new problem, TSP, not vice versa.)

From a graph G on m vertices $v_1, \ldots, v_m$, define the distance function d as follows:

$$d(i, j) = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge in } G \\ m + 1 & \text{otherwise.} \end{cases}$$

Set $D = m$. Clearly, d and D can be computed in polynomial time from G. Each vertex of G corresponds to a city in the constructed instance of TSP.

If G has a Hamiltonian circuit, then the length of the tour that corresponds to this circuit is exactly m. Conversely, if there is a tour whose length is at most m, then each step of the tour must have distance 1, not $m + 1$. Thus, each step corresponds to an edge of G, and the corresponding sequence of vertices in G is a Hamiltonian circuit.

5.5.5 Complete Problems for Other Classes

Besides NP, the following canonical complexity classes have natural complete problems. The three problems now listed are complete for their respective classes under log-space reducibility.

- NL: GRAPH ACCESSIBILITY PROBLEM

  Instance: A directed graph G with nodes $1, \ldots, N$.

  Question: Does G have a directed path from node 1 to node N?
- P: CIRCUIT VALUE PROBLEM

  Instance: A Boolean circuit (see Section 5.9) with output node u, and an assignment I of $\{\mathbf{0}, \mathbf{1}\}$ to each input node.

  Question: Is $\mathbf{1}$ the value of u under I?
- PSPACE: QUANTIFIED BOOLEAN FORMULAS

  Instance: A Boolean expression with all variables quantified with either $\forall$ or $\exists$ [e.g., $\forall x \forall y \exists z(x \wedge (\overline{y} \vee z))$].

  Question: Is the expression `true`?

These problems can be used to prove other problems are NL-complete, P-complete, and PSPACE-complete, respectively.

Stockmeyer and Meyer [1973] defined a natural decision problem that they proved to be complete for NE. If this problem were in P, then by closure under Karp reducibility (Theorem 5.8), we would have $\mathsf{NE} \subseteq \mathsf{P}$, a contradiction of the hierarchy theorems (Theorem 5.4). Therefore, this decision problem is infeasible: it has no polynomial-time algorithm. In contrast, decision problems in $\mathsf{NEXP} - \mathsf{P}$ that have been constructed by diagonalization are artificial problems that nobody would want to solve anyway. Although diagonalization produces unnatural problems by itself, the combination of diagonalization and completeness shows that *natural* problems are intractable.

The next section points out some limitations of current diagonalization techniques.

5.6 Relativization of the P vs. NP Problem

Let A be a language. Define P^A (respectively, NP^A) to be the class of languages accepted in polynomial time by deterministic (nondeterministic) oracle Turing machines with oracle A.

Proofs that use the diagonalization technique on Turing machines without oracles generally carry over to oracle Turing machines. Thus, for instance, the proof of the DTIME hierarchy theorem also shows that, for *any* oracle A, $\mathsf{DTIME}^A[n^2]$ is properly contained in $\mathsf{DTIME}^A[n^3]$. This can be seen as a *strength* of the diagonalization technique because it allows an argument to "relativize" to computation carried out relative to an oracle. In fact, there are examples of lower bounds (for deterministic, "unrelativized" circuit models) that make crucial use of the fact that the time hierarchies relativize in this sense.

But it can also be seen as a weakness of the diagonalization technique. The following important theorem demonstrates why.

Theorem 5.10 *There exist languages A and B such that $\mathsf{P}^A = \mathsf{NP}^A$, and $\mathsf{P}^B \neq \mathsf{NP}^B$* [Baker et al., 1975].

This shows that resolving the P vs. NP question requires techniques that do not relativize, that is, that do not apply to oracle Turing machines too. Thus, diagonalization as we currently know it is unlikely to succeed in separating P from NP because the diagonalization arguments we know (and in fact *most* of the arguments we know) relativize. Important non-relativizing proof techniques have appeared only recently, in connection with interactive proof systems (Section 5.11.1).

5.7 The Polynomial Hierarchy

Let $\mathcal{C}$ be a class of languages. Define:

- $\mathsf{NP}^{\mathcal{C}} = \bigcup_{A \in \mathcal{C}} \mathsf{NP}^A$
- $\Sigma_0^P = \Pi_0^P = \mathsf{P}$

and for $k \geq 0$, define:

- $\Sigma_{k+1}^P = \mathsf{NP}^{\Sigma_k^P}$
- $\Pi_{k+1}^P = \text{co-}\Sigma_{k+1}^P$.

Observe that $\Sigma_1^P = \mathsf{NP}^{\mathsf{P}} = \mathsf{NP}$ because each of polynomially many queries to an oracle language in P can be answered directly by a (nondeterministic) Turing machine in polynomial time. Consequently, $\Pi_1^P = \text{co-NP}$. For each k, $\Sigma_k^P \cup \Pi_k^P \subseteq \Sigma_{k+1}^P \cap \Pi_{k+1}^P$, but this inclusion is not known to be strict. See Figure 5.3.

The classes Σ_k^P and Π_k^P constitute the **polynomial hierarchy**. Define:

$$\mathsf{PH} = \bigcup_{k \geq 0} \Sigma_k^P.$$

It is straightforward to prove that $\mathsf{PH} \subseteq \mathsf{PSPACE}$, but it is not known whether the inclusion is strict. In fact, if $\mathsf{PH} = \mathsf{PSPACE}$, then the polynomial hierarchy collapses to some level, that is, $\mathsf{PH} = \Sigma_m^P$ for some m. In the next section, we define the polynomial hierarchy in two other ways, one of which is in terms of alternating Turing machines.

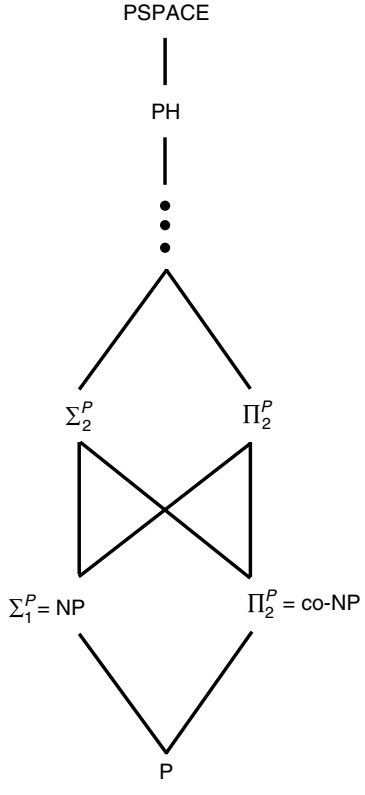


FIGURE 5.3 The polynomial hierarchy.

5.8 Alternating Complexity Classes

In this section, we define time and space complexity classes for alternating Turing machines, and we show how these classes are related to the classes introduced already. The possible computations of an alternating Turing machine M on an input word x can be represented by a tree T_x in which the root is the initial configuration, and the children of a nonterminal node C are the configurations reachable from C by one step of M. For a word x in $L(M)$, define an **accepting subtree** S of T_x to be a subtree of T_x with the following properties:

- S is finite.
- The root of S is the initial configuration with input word x.
- If S has an existential configuration C, then S has exactly one child of C in T_x; if S has a universal configuration C, then S has all children of C in T_x.
- Every leaf is a configuration whose state is the accepting state q_A.

Observe that each node in S is an accepting configuration.

We consider only alternating Turing machines that always halt. For $x \in L(M)$, define the time taken by M to be the height of the shortest accepting tree for x, and the space to be the maximum number of non-blank worktape cells among configurations in the accepting tree that minimizes this number. For $x \notin L(M)$, define the time to be the height of T_x, and the space to be the maximum number of non-blank worktape cells among configurations in T_x.

Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function. Define the following complexity classes:

- $\mathsf{ATIME}[t(n)]$ is the class of languages decided by alternating Turing machines of time complexity $O(t(n))$.
- $\mathsf{ASPACE}[s(n)]$ is the class of languages decided by alternating Turing machines of space complexity $O(s(n))$.

Because a nondeterministic Turing machine is a special case of an alternating Turing machine, for every $t(n)$ and $s(n)$, $\mathsf{NTIME}[t] \subseteq \mathsf{ATIME}[t]$ and $\mathsf{NSPACE}[s] \subseteq \mathsf{ASPACE}[s]$. The next theorem states further relationships between computational resources used by alternating Turing machines, and resources used by deterministic and nondeterministic Turing machines.

Theorem 5.11 (Alternation Theorems) [Chandra et al., 1981]. *Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function, $s(n) \geq \log n$.*

(a) $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{ATIME}[s(n)^2]$
(b) $\mathsf{ATIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$
(c) $\mathsf{ASPACE}[s(n)] \subseteq \mathsf{DTIME}[2^{O(s(n))}]$
(d) $\mathsf{DTIME}[t(n)] \subseteq \mathsf{ASPACE}[\log t(n)]$

In other words, space on deterministic and nondeterministic Turing machines is polynomially related to time on alternating Turing machines. Space on alternating Turing machines is exponentially related to time on deterministic Turing machines. The following corollary is immediate.

Theorem 5.12

(a) $\mathsf{ASPACE}[O(\log n)] = \mathsf{P}$
(b) $\mathsf{ATIME}[n^{O(1)}] = \mathsf{PSPACE}$
(c) $\mathsf{ASPACE}[n^{O(1)}] = \mathsf{EXP}$

In Section 5.7, we defined the classes of the polynomial hierarchy in terms of oracles, but we can also define them in terms of alternating Turing machines with restrictions on the number of alternations

between existential and universal states. Define a *k-alternating Turing machine* to be a machine such that on every computation path, the number of changes from an existential state to universal state, or from a universal state to an existential state, is at most $k - 1$. Thus, a nondeterministic Turing machine, which stays in existential states, is a 1-alternating Turing machine.

Theorem 5.13 [Stockmeyer, 1976; Wrathall, 1976]. *For any language A, the following are equivalent:*

1. $A \in \Sigma_k^P$.
2. *A is decided in polynomial time by a k-alternating Turing machine that starts in an existential state.*
3. *There exists a language B in* P *and a polynomial p such that for all x, $x \in A$ if and only if*

$$(\exists y_1 : |y_1| \leq p(|x|))(\forall y_2 : |y_2| \leq p(|x|)) \cdots (Qy_k : |y_k| \leq p(|x|))[(x, y_1, \ldots, y_k) \in B]$$

where the quantifier Q is $\exists$ *if k is odd,* $\forall$ *if k is even.*

Alternating Turing machines are closely related to Boolean circuits, which are defined in the next section.

5.9 Circuit Complexity

The hardware of electronic digital computers is based on digital logic gates, connected into combinational circuits (see Chapter 16). Here, we specify a model of computation that formalizes the combinational circuit.

A *Boolean circuit* on n input variables $x_1, \ldots, x_n$ is a directed acyclic graph with exactly n input nodes of indegree 0 labeled $x_1, \ldots, x_n$, and other nodes of indegree 1 or 2, called *gates*, labeled with the Boolean operators in $\{\wedge, \vee, \neg\}$. One node is designated as the output of the circuit. See Figure 5.4. Without loss of generality, we assume that there are no extraneous nodes; there is a directed path from each node to the output node. The indegree of a gate is also called its *fan-in.*

An *input assignment* is a function I that maps each variable x_i to either 0 or 1. The value of each gate g under I is obtained by applying the Boolean operation that labels g to the values of the immediate predecessors of g. The function computed by the circuit is the value of the output node for each input assignment.

A Boolean circuit computes a finite function: a function of only n binary input variables. To decide membership in a language, we need a circuit for each input length n.

A *circuit family* is an infinite set of circuits $C = \{c_1, c_2, \ldots\}$ in which each c_n is a Boolean circuit on n inputs. C *decides* a language $A \subseteq \{\mathbf{0},\mathbf{1}\}^*$ if for every n and every assignment $a_1, \ldots, a_n$ of $\{\mathbf{0},\mathbf{1}\}$ to the n inputs, the value of the output node of c_n is **1** if and only if the word $a_1 \cdots a_n \in A$. The *size complexity* of C is the function $z(n)$ that specifies the number of nodes in each c_n. The *depth complexity* of C is the function $d(n)$ that specifies the length of the longest directed path in c_n. Clearly, since the fan-in of each

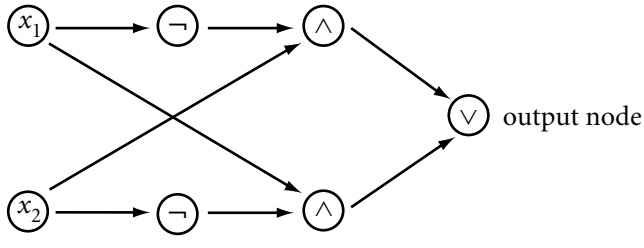


FIGURE 5.4 A Boolean circuit.

gate is at most 2, $d(n) \geq \log z(n) \geq \log n$. The class of languages decided by polynomial-size circuits is denoted by $\mathsf{P/poly}$.

With a different circuit for each input length, a circuit family could solve an undecidable problem such as the halting problem (see Chapter 6). For each input length, a table of all answers for machine descriptions of that length could be encoded into the circuit. Thus, we need to restrict our circuit families. The most natural restriction is that all circuits in a family should have a concise, uniform description, to disallow a different answer table for each input length. Several uniformity conditions have been studied, and the following is the most convenient.

A circuit family $\{c_1, c_2, \ldots\}$ of size complexity $z(n)$ is *log-space uniform* if there exists a deterministic Turing machine M such that on each input of length n, machine M produces a description of c_n, using space $O(\log z(n))$.

Now we define complexity classes for uniform circuit families and relate these classes to previously defined classes. Define the following complexity classes:

- $\mathsf{SIZE}[z(n)]$ is the class of languages decided by log-space uniform circuit families of size complexity $O(z(n))$.
- $\mathsf{DEPTH}[d(n)]$ is the class of languages decided by log-space uniform circuit families of depth complexity $O(d(n))$.

In our notation, $\mathsf{SIZE}[n^{O(1)}]$ equals P, which is a proper subclass of $\mathsf{P/poly}$.

Theorem 5.14

1. *If $t(n)$ is a time-constructible function, then* $\mathsf{DTIME}[t(n)] \subseteq \mathsf{SIZE}[t(n)\log t(n)]$ [Pippenger and Fischer, 1979].
2. $\mathsf{SIZE}[z(n)] \subseteq \mathsf{DTIME}[z(n)^{O(1)}]$.
3. *If $s(n)$ is a space-constructible function and $s(n) \geq \log n$, then* $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DEPTH}[s(n)^2]$ [Borodin, 1977].
4. *If $d(n) \geq \log n$, then* $\mathsf{DEPTH}[d(n)] \subseteq \mathsf{DSPACE}[d(n)]$ [Borodin, 1977].

The next theorem shows that size and depth on Boolean circuits are closely related to space and time on alternating Turing machines, provided that we permit sublinear running times for alternating Turing machines, as follows. We augment alternating Turing machines with a random-access input capability. To access the cell at position j on the input tape, M writes the binary representation of j on a special tape, in $\log j$ steps, and enters a special reading state to obtain the symbol in cell j.

Theorem 5.15 [Ruzzo, 1979]. *Let $t(n) \geq \log n$ and $s(n) \geq \log n$ be such that the mapping $n \mapsto (t(n), s(n))$ (in binary) is computable in time $s(n)$.*

1. *Every language decided by an alternating Turing machine of simultaneous space complexity $s(n)$ and time complexity $t(n)$ can be decided by a log-space uniform circuit family of simultaneous size complexity $2^{O(s(n))}$ and depth complexity $O(t(n))$.*
2. *If $d(n) \geq (\log z(n))^2$, then every language decided by a log-space uniform circuit family of simultaneous size complexity $z(n)$ and depth complexity $d(n)$ can be decided by an alternating Turing machine of simultaneous space complexity $O(\log z(n))$ and time complexity $O(d(n))$.*

In a sense, the Boolean circuit family is a model of parallel computation, because all gates compute independently, in parallel. For each $k \geq 0$, NC^k denotes the class of languages decided by log-space uniform bounded fan-in circuits of polynomial size and depth $O((\log n)^k)$, and AC^k is defined analogously for unbounded fan-in circuits. In particular, AC^k is the same as the class of languages decided by a parallel machine model called the CRCW PRAM with polynomially many processors in parallel time $O((\log n)^k)$ [Stockmeyer and Vishkin, 1984].

5.10 Probabilistic Complexity Classes

Since the 1970s, with the development of randomized algorithms for computational problems (see Chapter 12). Complexity theorists have placed randomized algorithms on a firm intellectual foundation. In this section, we outline some basic concepts in this area.

A **probabilistic Turing machine** M can be formalized as a nondeterministic Turing machine with exactly two choices at each step. During a computation, M chooses each possible next step with independent probability 1/2. Intuitively, at each step, M flips a fair coin to decide what to do next. The probability of a computation path of t steps is $1/2^t$. The probability that M accepts an input string x, denoted by $p_M(x)$, is the sum of the probabilities of the accepting computation paths.

Throughout this section, we consider only machines whose time complexity $t(n)$ is time-constructible. Without loss of generality, we can assume that every computation path of such a machine halts in exactly t steps.

Let A be a language. A probabilistic Turing machine M decides A with

		for all $x \in A$	for all $x \notin A$
unbounded two-sided error	if	$p_M(x) > 1/2$	$p_M(x) \leq 1/2$
bounded two-sided error	if	$p_M(x) > 1/2 + \epsilon$	$p_M(x) < 1/2 - \epsilon$
		for some positive constant ϵ	
one-sided error	if	$p_M(x) > 1/2$	$p_M(x) = 0$

Many practical and important probabilistic algorithms make one-sided errors. For example, in the primality testing algorithm of Solovay and Strassen [1977], when the input x is a prime number, the algorithm *always* says "prime"; when x is composite, the algorithm *usually* says "composite," but may occasionally say "prime." Using the definitions above, this means that the Solovay-Strassen algorithm is a one-sided error algorithm for the set A of composite numbers. It also is a bounded two-sided error algorithm for $\overline{A}$, the set of prime numbers.

These three kinds of errors suggest three complexity classes:

1. PP is the class of languages decided by probabilistic Turing machines of polynomial time complexity with unbounded two-sided error.
2. BPP is the class of languages decided by probabilistic Turing machines of polynomial time complexity with bounded two-sided error.
3. RP is the class of languages decided by probabilistic Turing machines of polynomial time complexity with one-sided error.

In the literature, RP is also called R.

A probabilistic Turing machine M is a **PP-machine** (respectively, a **BPP-machine**, an **RP-machine**) if M has polynomial time complexity, and M decides with two-sided error (bounded two-sided error, one-sided error).

Through repeated Bernoulli trials, we can make the error probabilities of BPP-machines and RP-machines arbitrarily small, as stated in the following theorem. (Among other things, this theorem implies that RP $\subseteq$ BPP.)

Theorem 5.16 *If $A \in$ BPP, then for every polynomial $q(n)$, there exists a* BPP*-machine M such that $p_M(x) > 1 - 1/2^{q(n)}$ for every $x \in A$, and $p_M(x) < 1/2^{q(n)}$ for every $x \notin A$.*

If $L \in$ RP, then for every polynomial $q(n)$, there exists an RP*-machine M such that $p_M(x) > 1 - 1/2^{q(n)}$ for every x in L.*

It is important to note just how minuscule the probability of error is (provided that the coin flips are truly random). If the probability of error is less than $1/2^{5000}$, then it is less likely that the algorithm produces an incorrect answer than that the computer will be struck by a meteor. An algorithm whose probability of

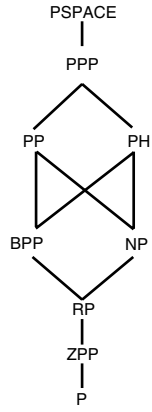


FIGURE 5.5 Probabilistic complexity classes.

error is $1/2^{5000}$ is essentially as good as an algorithm that makes no errors. For this reason, many computer scientists consider BPP to be the class of practically feasible computational problems.

Next, we define a class of problems that have probabilistic algorithms that make no errors. Define:

$$\mathsf{ZPP} = \mathsf{RP} \cap \text{co-RP}$$

The letter Z in ZPP is for zero probability of error, as we now demonstrate. Suppose $A \in \mathsf{ZPP}$. Here is an algorithm that checks membership in A. Let M be an RP-machine that decides A, and let M' be an RP-machine that decides $\overline{A}$. For an input string x, alternately run M and M' on x, repeatedly, until a computation path of one machine accepts x. If M accepts x, then accept x; if M' accepts x, then reject x. This algorithm works correctly because when an RP-machine accepts its input, it does not make a mistake. This algorithm might not terminate, but with very high probability, the algorithm terminates after a few iterations.

The next theorem expresses some known relationships between probabilistic complexity classes and other complexity classes, such as classes in the polynomial hierarchy. See Section 5.7 and Figure 5.5.

Theorem 5.17

(a) $\mathsf{P} \subseteq \mathsf{ZPP} \subseteq \mathsf{RP} \subseteq \mathsf{BPP} \subseteq \mathsf{PP} \subseteq \mathsf{PSPACE}$ [Gill, 1977]
(b) $\mathsf{RP} \subseteq \mathsf{NP} \subseteq \mathsf{PP}$ [Gill, 1977]
(c) $\mathsf{BPP} \subseteq \Sigma_2^P \cap \Pi_2^P$ [Lautemann, 1983; Sipser, 1983]
(d) $\mathsf{BPP} \subset \mathsf{P/poly}$
(e) $\mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}}$ [Toda, 1991]

An important recent research area called **de-randomization** studies whether randomized algorithms can be converted to deterministic ones of the same or comparable efficiency. For example, if there is a language in E that requires Boolean circuits of size $2^{\Omega(n)}$ to decide it, then $\mathsf{BPP} = \mathsf{P}$ [Impagliazzo and Wigderson, 1997].

5.11 Interactive Models and Complexity Classes

5.11.1 Interactive Proofs

In Section 5.3.2, we characterized NP as the set of languages whose membership proofs can be checked quickly, by a deterministic Turing machine M of polynomial time complexity. A different notion of proof involves interaction between two parties, a prover P and a verifier V, who exchange messages. In an **interactive proof system** [Goldwasser et al., 1989], the prover is an all-powerful machine, with

unlimited computational resources, analogous to a teacher. The verifier is a computationally limited machine, analogous to a student. Interactive proof systems are also called "Arthur-Merlin games": the wizard Merlin corresponds to P, and the impatient Arthur corresponds to V [Babai and Moran, 1988].

Formally, an **interactive proof system** comprises the following:

- A read-only input tape on which an input string x is written.
- A *verifier* V, which is a probabilistic Turing machine augmented with the capability to send and receive messages. The running time of V is bounded by a polynomial in $|x|$.
- A *prover* P, which receives messages from V and sends messages to V.
- A tape on which V writes messages to send to P, and a tape on which P writes messages to send to V. The length of every message is bounded by a polynomial in $|x|$.

A computation of an interactive proof system (P, V) proceeds in rounds, as follows. For $j = 1, 2, \ldots$, in round j, V performs some steps, writes a message m_j, and temporarily stops. Then P reads m_j and responds with a message m'_j, which V reads in round $j + 1$. An interactive proof system (P, V) **accepts** an input string x if the probability of acceptance by V satisfies $p_V(x) > 1/2$.

In an interactive proof system, a prover can convince the verifier about the truth of a statement without exhibiting an entire proof, as the following example illustrates.

Consider the graph non-isomorphism problem: the input consists of two graphs G and H, and the decision is `yes` if and only if G is not isomorphic to H. Although there is a short proof that two graphs *are* isomorphic (namely: the proof consists of the isomorphism mapping G onto H), nobody has found a general way of proving that two graphs are *not* isomorphic that is significantly shorter than listing all $n!$ permutations and showing that each fails to be an isomorphism. (That is, the graph non-isomorphism problem is in co-NP, but is not known to be in NP.) In contrast, the verifier V in an interactive proof system is able to take statistical evidence into account, and determine "beyond all reasonable doubt" that two graphs are non-isomorphic, using the following protocol.

In each round, V randomly chooses either G or H with equal probability; if V chooses G, then V computes a random permutation G' of G, presents G' to P, and asks P whether G' came from G or from H (and similarly if V chooses H). If P gave an erroneous answer on the first round, and G is isomorphic to H, then after k subsequent rounds, the probability that P answers all the subsequent queries correctly is $1/2^k$. (To see this, it is important to understand that the prover P does not see the coins that V flips in making its random choices; P sees only the graphs G' and H' that V sends as messages.) V accepts the interaction with P as "proof" that G and H are non-isomorphic if P is able to pick the correct graph for 100 consecutive rounds. Note that V has ample grounds to accept this as a convincing demonstration: if the graphs are indeed isomorphic, the prover P would have to have an incredible streak of luck to fool V.

It is important to comment that de-randomization techniques applied to these proof systems have shown that under plausible hardness assumptions, proofs of non-isomorphism of sub-exponential length (or even polynomial length) do exist [Klivans and van Melkebeek, 2002]. Thus, many complexity theoreticians now conjecture that the graph isomorphism problem lies in $\mathsf{NP} \cap \text{co-}\mathsf{NP}$.

The complexity class IP comprises the languages A for which there exists a verifier V and a positive ϵ such that

- There exists a prover $\hat{P}$ such that for all x in A, the interactive proof system $(\hat{P}, V)$ accepts x with probability greater than $1/2 + \epsilon$; and
- For every prover P and every $x \notin A$, the interactive proof system (P, V) rejects x with probability greater than $1/2 + \epsilon$.

By substituting random choices for existential choices in the proof that $\mathsf{ATIME}(t) \subseteq \mathsf{DSPACE}(t)$ (Theorem 5.11), it is straightforward to show that $\mathsf{IP} \subseteq \mathsf{PSPACE}$. It was originally believed likely that IP was a small subclass of PSPACE. Evidence supporting this belief was the construction of an oracle language B for which $\text{co-}\mathsf{NP}^B - \mathsf{IP}^B \neq \emptyset$ [Fortnow and Sipser, 1988], so that IP^B is strictly included in PSPACE^B. Using a proof technique that does not relativize, however, Shamir [1992] proved that, in fact, IP and PSPACE are the same class.

Theorem 5.18 IP = PSPACE. [Shamir, 1992].

If NP is a proper subset of PSPACE, as is widely believed, then Theorem 5.18 says that interactive proof systems can decide a larger class of languages than NP.

5.11.2 Probabilistically Checkable Proofs

In an interactive proof system, the verifier does not need a complete conventional proof to become convinced about the membership of a word in a language, but uses random choices to query parts of a proof that the prover may know. This interpretation inspired another notion of "proof": a proof consists of a (potentially) large amount of information that the verifier need only inspect in a few places in order to become convinced. The following definition makes this idea more precise.

A language A has a **probabilistically checkable proof** if there exists an oracle BPP-machine M such that:

- For all $x \in A$, there exists an oracle language B_x such that M^{B_x} accepts x with probability 1.
- For all $x \notin A$, and for every language B, machine M^B accepts x with probability strictly less than $1/2$.

Intuitively, the oracle language B_x represents a proof of membership of x in A. Notice that B_x can be finite since the length of each possible query during a computation of M^{B_x} on x is bounded by the running time of M. The oracle language takes the role of the prover in an interactive proof system — but in contrast to an interactive proof system, the prover cannot change strategy adaptively in response to the questions that the verifier poses. This change results in a potentially stronger system, since a machine M that has bounded error probability relative to all languages B might not have bounded error probability relative to some adaptive prover. Although this change to the proof system framework may seem modest, it leads to a characterization of a class that seems to be much larger than PSPACE.

Theorem 5.19 *A has a probabilistically checkable proof if and only if $A \in$ NEXP* [Babai et al., 1991].

Although the notion of probabilistically checkable proofs seems to lead us away from feasible complexity classes, by considering natural restrictions on how the proof is accessed, we can obtain important insights into familiar complexity classes.

Let $\mathsf{PCP}[r(n), q(n)]$ denote the class of languages with probabilistically checkable proofs in which the probabilistic oracle Turing machine M makes $O[r(n)]$ random binary choices, and queries its oracle $O[q(n)]$ times. (For this definition, we assume that M has either one or two choices for each step.) It follows from the definitions that $\mathsf{BPP} = \mathsf{PCP}(n^{O(1)}, 0)$, and $\mathsf{NP} = \mathsf{PCP}(0, n^{O(1)})$.

Theorem 5.20 (The PCP Theorem) $\mathsf{NP} = \mathsf{PCP}[\emptyset \log n, \emptyset(1)]$ [Arora et al., 1998].

Theorem 5.20 asserts that for every language A in NP, a proof that $x \in A$ can be encoded so that the verifier can be convinced of the correctness of the proof (or detect an incorrect proof) by using only $O(\log n)$ random choices, and inspecting only a *constant* number of bits of the proof.

5.12 Kolmogorov Complexity

Until now, we have considered only dynamic complexity measures, namely, the time and space used by Turing machines. Kolmogorov complexity is a static complexity measure that captures the difficulty of describing a string. For example, the string consisting of three million zeroes can be described with fewer than three million symbols (as in this sentence). In contrast, for a string consisting of three million randomly generated bits, with high probability there is no shorter description than the string itself.

Let U be a universal Turing machine (see Section 5.2.3). Let λ denote the empty string. The **Kolmogorov complexity** of a binary string y with respect to U, denoted by $K_U(y)$, is the length of the shortest binary string i such that on input $\langle i, \lambda \rangle$, machine U outputs y. In essence, i is a description of y, for it tells U how to generate y.

The next theorem states that different choices for the universal Turing machine affect the definition of Kolmogorov complexity in only a small way.

Theorem 5.21 (Invariance Theorem) *There exists a universal Turing machine U such that for every universal Turing machine U', there is a constant c such that for all y, $K_U(y) \leq K_{U'}(y) + c$.*

Henceforth, let K be defined by the universal Turing machine of Theorem 5.21. For every integer n and every binary string y of length n, because y can be described by giving itself explicitly, $K(y) \leq n + c'$ for a constant c'. Call y **incompressible** if $K(y) \geq n$. Since there are 2^n binary strings of length n and only $2^n - 1$ possible shorter descriptions, there exists an incompressible string for every length n.

Kolmogorov complexity gives a precise mathematical meaning to the intuitive notion of "randomness." If someone flips a coin 50 times and it comes up "heads" each time, then intuitively, the sequence of flips is not random — although from the standpoint of probability theory, the all-heads sequence is precisely as likely as any other sequence. Probability theory does not provide the tools for calling one sequence "more random" than another; Kolmogorov complexity theory does.

Kolmogorov complexity provides a useful framework for presenting combinatorial arguments. For example, when one wants to prove that an object with some property P exists, then it is sufficient to show that any object that does *not* have property P has a short description; thus, any incompressible (or "random") object must have property P. This sort of argument has been useful in proving lower bounds in complexity theory.

5.13 Research Issues and Summary

The core research questions in complexity theory are expressed in terms of separating complexity classes:

- Is L different from NL?
- Is P different from RP or BPP?
- Is P different from NP?
- Is NP different from PSPACE?

Motivated by these questions, much current research is devoted to efforts to understand the power of nondeterminism, randomization, and interaction. In these studies, researchers have gone well beyond the theory presented in this chapter:

- Beyond Turing machines and Boolean circuits, to restricted and specialized models in which nontrivial lower bounds on complexity can be proved
- Beyond deterministic reducibilities, to nondeterministic and probabilistic reducibilities, and refined versions of the reducibilities considered here
- Beyond worst-case complexity, to average-case complexity

Recent research in complexity theory has had direct applications to other areas of computer science and mathematics. Probabilistically checkable proofs were used to show that obtaining approximate solutions to some optimization problems is as difficult as solving them exactly. Complexity theory has provided new tools for studying questions in finite model theory, a branch of mathematical logic. Fundamental questions in complexity theory are intimately linked to practical questions about the use of cryptography for computer security, such as the existence of one-way functions and the strength of public key cryptosystems.

This last point illustrates the urgent practical need for progress in computational complexity theory. Many popular cryptographic systems in current use are based on unproven assumptions about the difficulty

of computing certain functions (such as the factoring and discrete logarithm problems). All of these systems are thus based on wishful thinking and conjecture. Research is needed to resolve these open questions and replace conjecture with mathematical certainty.

Acknowledgments

Donna Brown, Bevan Das, Raymond Greenlaw, Lane Hemaspaandra, John Jozwiak, Sung-il Pae, Leonard Pitt, Michael Roman, and Martin Tompa read earlier versions of this chapter and suggested numerous helpful improvements. Karen Walny checked the references.

Eric W. Allender was supported by the National Science Foundation under Grant CCR-0104823. Michael C. Loui was supported by the National Science Foundation under Grant SES-0138309. Kenneth W. Regan was supported by the National Science Foundation under Grant CCR-9821040.

Defining Terms

Complexity class: A set of languages that are decided within a particular resource bound. For example, $\mathsf{NTIME}(n^2 \log n)$ is the set of languages decided by nondeterministic Turing machines within $O(n^2 \log n)$ time.

Constructibility: A function $f(n)$ is time (respectively, space) constructible if there exists a deterministic Turing machine that halts after exactly $f(n)$ steps (after using exactly $f(n)$ worktape cells) for every input of length n.

Diagonalization: A technique for constructing a language A that differs from every $L(M_i)$ for a list of machines $M_1, M_2, \ldots$.

NP-complete: A language A_0 is NP-complete if $A_0 \in \mathsf{NP}$ and $A \leq_m^p A_0$ for every A in NP; that is, for every A in NP, there exists a function f computable in polynomial time such that for every x, $x \in A$ if and only if $f(x) \in A_0$.

Oracle: An oracle is a language A to which a machine presents queries of the form "Is w in A" and receives each correct answer in one step.

Padding: A technique for establishing relationships between complexity classes that uses padded versions of languages, in which each word is padded out with multiple occurrences of a new symbol — the word x is replaced by the word $x\#^{f(|x|)}$ for a numeric function f — in order to artificially reduce the complexity of the language.

Reduction: A language A reduces to a language B if a machine that decides B can be used to decide A efficiently.

Time and space complexity: The time (respectively, space) complexity of a deterministic Turing machine M is the maximum number of steps taken (nonblank cells used) by M among all input words of length n.

Turing machine: A Turing machine M is a model of computation with a read-only input tape and multiple worktapes. At each step, M reads the tape cells on which its access heads are located, and depending on its current state and the symbols in those cells, M changes state, writes new symbols on the worktape cells, and moves each access head one cell left or right or not at all.

References

Allender, E., Loui, M.C., and Regan, K.W. 1999. Chapter 27: Complexity classes, Chapter 28: Reducibility and completeness, Chapter 29: Other complexity classes and measures. In *Algorithms and Theory of Computation Handbook*, Ed. M. J. Atallah, CRC Press, Boca Raton, FL.

Arora, S., Lund, C., Motwani, R., Sudan, M., and Szegedy, M. 1998. Proof verification and hardness of approximation problems. *J. ACM*, 45(3):501–555.

Babai, L. and Moran, S. 1988. Arthur-Merlin games: a randomized proof system, and a hierarchy of complexity classes. *J. Comput. Sys. Sci.*, 36(2):254–276.

Babai, L., Fortnow, L., and Lund, C. 1991. Nondeterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1:3–40.

Baker, T., Gill, J., and Solovay, R. 1975. Relativizations of the P = NP? question. *SIAM J. Comput.*, 4(4): 431–442.

Balcázar, J.L., Díaz, J., and Gabarró, J. 1990. *Structural Complexity II*. Springer-Verlag, Berlin.

Balcázar, J.L., Díaz, J., and Gabarró, J. 1995. *Structural Complexity I*. 2nd ed. Springer-Verlag, Berlin.

Book, R.V. 1974. Comparing complexity classes. *J. Comp. Sys. Sci.*, 9(2):213–229.

Borodin, A. 1972. Computational complexity and the existence of complexity gaps. *J. Assn. Comp. Mach.*, 19(1):158–174.

Borodin, A. 1977. On relating time and space to size and depth. *SIAM J. Comput.*, 6(4):733–744.

Bovet, D.P. and Crescenzi, P. 1994. *Introduction to the Theory of Complexity*. Prentice Hall International Ltd; Hertfordshire, U.K.

Chandra, A.K., Kozen, D.C., and Stockmeyer, L.J. 1981. Alternation. *J. Assn. Comp. Mach.*, 28(1):114–133.

Cook, S.A. 1971. The complexity of theorem-proving procedures. In *Proc. 3rd Annu. ACM Symp. Theory Comput.*, pp. 151–158. Shaker Heights, OH.

Du, D-Z. and Ko, K.-I. 2000. *Theory of Computational Complexity*. Wiley, New York.

Fortnow, L. and Sipser, M. 1988. Are there interactive protocols for co-NP languages? *Inform. Process. Lett.*, 28(5):249–251.

Garey, M.R. and Johnson, D.S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, San Francisco.

Gill, J. 1977. Computational complexity of probabilistic Turing machines. *SIAM J. Comput.*, 6(4):675–695.

Goldwasser, S., Micali, S., and Rackoff, C. 1989. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208.

Hartmanis, J., Ed. 1989. *Computational Complexity Theory.* American Mathematical Society, Providence, RI.

Hartmanis, J. 1994. On computational complexity and the nature of computer science. *Commun. ACM*, 37(10):37–43.

Hartmanis, J. and Stearns, R.E. 1965. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, 117:285–306.

Hemaspaandra, L.A. and Ogihara, M. 2002. *The Complexity Theory Companion*. Springer, Berlin.

Hemaspaandra, L.A. and Selman, A.L., Eds. 1997. *Complexity Theory Retrospective II*. Springer, New York.

Hennie, F. and Stearns, R.A. 1966. Two–way simulation of multitape Turing machines. *J. Assn. Comp. Mach.*, 13(4):533–546.

Immerman, N. 1988. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938.

Impagliazzo, R. and Wigderson, A. 1997. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. *Proc. 29th Annu. ACM Symp. Theory Comput.*, ACM Press, pp. 220–229. El Paso, TX.

Jones, N.D. 1975. Space-bounded reducibility among combinatorial problems. *J. Comp. Sys. Sci.*, 11(1):68–85. Corrigendum *J. Comp. Sys. Sci.*, 15(2):241, 1977.

Karp, R.M. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations.* R.E. Miller and J.W. Thatcher, Eds., pp. 85–103. Plenum Press, New York.

Klivans, A.R. and van Melkebeek, D. 2002. Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses. *SIAM J. Comput.*, 31(5):1501–1526.

Ladner, R.E. 1975. On the structure of polynomial-time reducibility. *J. Assn. Comp. Mach.*, 22(1):155–171.

Lautemann, C. 1983. BPP and the polynomial hierarchy. *Inf. Proc. Lett.*, 17(4):215–217.

Levin, L. 1973. Universal search problems. *Problems of Information Transmission*, 9(3):265–266 (in Russian).

Li, M. and Vitányi, P.M.B. 1997. *An Introduction to Kolmogorov Complexity and Its Applications*. 2nd ed. Springer-Verlag, New York.

Papadimitriou, C.H. 1994. *Computational Complexity*. Addison-Wesley, Reading, MA.

Pippenger, N. and Fischer, M. 1979. Relations among complexity measures. *J. Assn. Comp. Mach.*, 26(2):361–381.
Ruzzo, W.L. 1981. On uniform circuit complexity. *J. Comp. Sys. Sci.*, 22(3):365–383.
Savitch, W.J. 1970. Relationship between nondeterministic and deterministic tape complexities. *J. Comp. Sys. Sci.*, 4(2):177–192.
Seiferas, J.I., Fischer, M.J., and Meyer, A.R. 1978. Separating nondeterministic time complexity classes. *J. Assn. Comp. Mach.*, 25(1):146–167.
Shamir, A. 1992. IP = PSPACE. *J. ACM* 39(4):869–877.
Sipser, M. 1983. Borel sets and circuit complexity. In *Proc. 15th Annual ACM Symposium on the Theory of Computing*, pp. 61–69.
Sipser, M. 1992. The history and status of the P versus NP question. In *Proc. 24th Annu. ACM Symp. Theory Comput.*, ACM Press, pp. 603–618. Victoria, B.C., Canada.
Solovay, R. and Strassen, V. 1977. A fast Monte-Carlo test for primality. *SIAM J. Comput.*, 6(1):84–85.
Stearns, R.E. 1990. Juris Hartmanis: the beginnings of computational complexity. In *Complexity Theory Retrospective.* A.L. Selman, Ed., pp. 5–18, Springer-Verlag, New York.
Stockmeyer, L.J. 1976. The polynomial time hierarchy. *Theor. Comp. Sci.*, 3(1):1–22.
Stockmeyer, L.J. 1987. Classifying the computational complexity of problems. *J. Symb. Logic,* 52:1–43.
Stockmeyer, L.J. and Chandra, A.K. 1979. Intrinsically difficult problems. *Sci. Am.*, 240(5):140–159.
Stockmeyer, L.J. and Meyer, A.R. 1973. Word problems requiring exponential time: preliminary report. In *Proc. 5th Annu. ACM Symp. Theory Comput.*, ACM Press, pp. 1–9. Austin, TX.
Stockmeyer, L.J. and Vishkin, U. 1984. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422.
Szelepcsényi, R. 1988. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284.
Toda, S. 1991. PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20(5):865–877.
van Leeuwen, J. 1990. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier Science, Amsterdam, and M.I.T. Press, Cambridge, MA.
Wagner, K. and Wechsung, G. 1986. *Computational Complexity.* D. Reidel, Dordrecht, The Netherlands.
Wrathall, C. 1976. Complete sets and the polynomial-time hierarchy. *Theor. Comp. Sci.*, 3(1):23–33.

Further Information

This chapter is a short version of three chapters written by the same authors for the *Algorithms and Theory of Computation Handbook* [Allender et al., 1999].

The formal theoretical study of computational complexity began with the paper of Hartmanis and Stearns [1965], who introduced the basic concepts and proved the first results. For historical perspectives on complexity theory, see Hartmanis [1994], Sipser [1992], and Stearns [1990].

Contemporary textbooks on complexity theory are by Balcázar et al. [1990, 1995], Bovet and Crescenzi [1994], Du and Ko [2000], Hemaspaandra and Ogihara [2002], and Papadimitriou [1994]. Wagner and Wechsung [1986] is an exhaustive survey of complexity theory that covers work published before 1986. Another perspective of some of the issues covered in this chapter can be found in the survey by Stockmeyer [1987].

A good general reference is the *Handbook of Theoretical Computer Science* [van Leeuwen, 1990], Volume A. The following chapters in that *Handbook* are particularly relevant: "Machine Models and Simulations," by P. van Emde Boas, pp. 1–66; "A Catalog of Complexity Classes," by D.S. Johnson, pp. 67–161; "Machine-Independent Complexity Theory," by J.I. Seiferas, pp. 163–186; "Kolmogorov Complexity and its Applications," by M. Li and P.M.B. Vitányi, pp. 187–254; and "The Complexity of Finite Functions," by R.B. Boppana and M. Sipser, pp. 757–804, which covers circuit complexity.

A collection of articles edited by Hartmanis [1989] includes an overview of complexity theory, and chapters on sparse complete languages, on relativizations, on interactive proof systems, and on applications of complexity theory to cryptography. A collection edited by Hemaspaandra and Selman [1997] includes chapters on quantum and biological computing, on proof systems, and on average case complexity.

For specific topics in complexity theory, the following references are helpful. Garey and Johnson [1979] explain NP-completeness thoroughly, with examples of NP-completeness proofs, and a collection of hundreds of NP-complete problems. Li and Vitányi [1997] provide a comprehensive, scholarly treatment of Kolmogorov complexity, with many applications.

Surveys and lecture notes on complexity theory that can be obtained via the Web are maintained by A. Czumaj and M. Kutylowski at:

`http://www.uni-paderborn.de/fachbereich/AG/agmadh/WWW/english/scripts.html`

As usual with the Web, such links are subject to change. Two good stem pages to begin searches are the site for SIGACT (the ACM Special Interest Group on Algorithms and Computation Theory) and the site for the annual IEEE Conference on Computational Complexity:

`http://sigact.acm.org/`
`http://www.computationalcomplexity.org/`

The former site has a pointer to a "Virtual Address Book" that indexes the personal Web pages of over 1000 computer scientists, including all three authors of this chapter. Many of these pages have downloadable papers and links to further research resources. The latter site includes a pointer to the *Electronic Colloquium on Computational Complexity* maintained at the University of Trier, Germany, which includes downloadable prominent research papers in the field, often with updates and revisions.

Research papers on complexity theory are presented at several annual conferences, including the annual ACM Symposium on Theory of Computing; the annual International Colloquium on Automata, Languages, and Programming, sponsored by the European Association for Theoretical Computer Science (EATCS); and the annual Symposium on Foundations of Computer Science, sponsored by the IEEE. The annual Conference on Computational Complexity (formerly Structure in Complexity Theory), also sponsored by the IEEE, is entirely devoted to complexity theory. Research articles on complexity theory regularly appear in the following journals, among others: *Chicago Journal on Theoretical Computer Science, Computational Complexity, Information and Computation, Journal of the ACM, Journal of Computer and System Sciences, SIAM Journal on Computing, Theoretical Computer Science,* and *Theory of Computing Systems* (formerly *Mathematical Systems Theory*). Each issue of *ACM SIGACT News* and *Bulletin of the EATCS* contains a column on complexity theory.

6

Formal Models and Computability

Tao Jiang
University of California

Ming Li
University of Waterloo

Bala Ravikumar
University of Rhode Island

6.1 Introduction

The concept of **algorithms** is perhaps almost as old as human civilization. The famous Euclid's algorithm is more than 2000 years old. Angle trisection, solving diophantine equations, and finding polynomial roots in terms of radicals of coefficients are some well-known examples of algorithmic questions. However, until the 1930s the notion of algorithms was used informally (or rigorously but in a limited context). It was a major triumph of logicians and mathematicians of this century to offer a rigorous definition of this fundamental concept. The revolution that resulted in this triumph was a collective achievement of many mathematicians, notably Church, Gödel, Kleene, Post, and Turing. Of particular interest is a machine model proposed by Turing in 1936, which has come to be known as a **Turing machine** [Turing 1936].

This particular achievement had numerous significant consequences. It led to the concept of a general-purpose computer or universal computation, a revolutionary idea originally anticipated by Babbage in the 1800s. It is widely acknowledged that the development of a universal Turing machine was prophetic of the modern all-purpose digital computer and played a key role in the thinking of pioneers in the development of modern computers such as von Neumann [Davis 1980]. From a mathematical point of view, however, a more interesting consequence was that it was now possible to show the *nonexistence* of algorithms, hitherto impossible due to their elusive nature. In addition, many apparently different definitions of an algorithm proposed by different researchers in different continents turned out to be equivalent (in a precise technical sense, explained later). This equivalence led to the widely held hypothesis known as the *Church–Turing thesis* that mechanical solvability is the same as solvability on a Turing machine.

Formal languages are closely related to algorithms. They were introduced as a way to convey mathematical proofs without errors. Although the concept of a formal language dates back at least to the time of Leibniz, a systematic study of them did not begin until the beginning of this century. It became a vigorous field of study when Chomsky formulated simple grammatical rules to describe the syntax of a language

[Chomsky 1956]. **Grammars** and **formal languages** entered into computability theory when Chomsky and others found ways to use them to classify algorithms.

The main theme of this chapter is about formal models, which include Turing machines (and their variants) as well as grammars. In fact, the two concepts are intimately related. Formal computational models are aimed at providing a framework for computational problem solving, much as electromagnetic theory provides a framework for problems in electrical engineering. Thus, formal models guide the way to build computers and the way to program them. At the same time, new models are motivated by advances in the technology of computing machines. In this chapter, we will discuss only the most basic computational models and use these models to classify problems into some fundamental classes. In doing so, we hope to provide the reader with a conceptual basis with which to read other chapters in this Handbook.

6.2 Computability and a Universal Algorithm

Turing's notion of mechanical computation was based on identifying the basic steps of such computations. He reasoned that an operation such as multiplication is not primitive because it can be divided into more basic steps such as digit-by-digit multiplication, shifting, and adding. Addition itself can be expressed in terms of more basic steps such as add the lowest digits, compute, carry, and move to the next digit, etc. Turing thus reasoned that the most basic features of mechanical computation are the abilities to read and write on a storage medium (which he chose to be a linear tape divided into cells or squares) and to make some simple logical decisions. He also restricted each tape cell to hold only one among a finite number of symbols (which we call the *tape alphabet*).* The decision step enables the computer to control the sequence of actions. To make things simple, Turing restricted the next action to be performed on a cell neighboring the one on which the current action occurred. He also introduced an instruction that told the computer to stop. In summary, Turing proposed a model to characterize mechanical computation as being carried out as a sequence of instructions of the form: write a symbol (such as 0 or 1) on the tape cell, move to the next cell, observe the symbol currently scanned and choose the next step accordingly, or stop.

These operations define a language we call the GOTO language.** Its instructions are

PRINT i (i is a tape symbol)
GO RIGHT
GO LEFT
GO TO STEP j IF i IS SCANNED
STOP

A **program** in this language is a sequence of instructions (written one per line) numbered $1 - k$. To run a program written in this language, we should provide the *input*. We will assume that the input is a string of symbols from a finite input alphabet (which is a subset of the tape alphabet), which is stored on the tape before the computation begins. How much memory should we allow the computer to use? Although we do not want to place any bounds on it, allowing an infinite tape is not realistic. This problem is circumvented by allowing *expandable memory*. In the beginning, the tape containing the input defines its boundary. When the machine moves beyond the current boundary, a new memory cell will be attached with a special symbol B (blank) written on it. Finally, we define the result of computation as the contents of the tape when the computer reaches the STOP instruction.

We will present an example program written in the GOTO language. This program accomplishes the simple task of doubling the number of 1s (Figure 6.1). More precisely, on the input containing k 1s, the

*This bold step of using a discrete model was perhaps the harbinger of the digital revolution that was soon to follow.

**Turing's original formulation is closer to our presentation in Section 6.5. But the GOTO language presents an equivalent model.

```
 1  PRINT 0
 2  GO LEFT
 3  GO TO STEP 2 IF 1 IS SCANNED
 4  PRINT 1
 5  GO RIGHT
 6  GO TO STEP 5 IF 1 IS SCANNED
 7  PRINT 1
 8  GO RIGHT
 9  GO TO STEP 1 IF 1 IS SCANNED
10  STOP
```

FIGURE 6.1 The doubling program in the GOTO language.

program produces $2k$ 1s. Informally, the program achieves its goal as follows. When it reads a 1, it changes the 1 to 0, moves left looking for a new cell, writes a 1 in the cell, returns to the starting cell and rewrites as 1, and repeats this step for each 1. Note the way the GOTO instructions are used for repetition. This feature is the most important aspect of programming and can be found in all of the imperative style programming languages.

The simplicity of the GOTO language is rather deceptive. There is strong reason to believe that it is powerful enough that any mechanical computation can be expressed by a suitable program in the GOTO language. Note also that the programs written in the GOTO language may not always halt, that is, on certain inputs, the program may never reach the STOP instruction. In this case, we say that the output is undefined.

We can now give a precise definition of what an algorithm is. An algorithm is any program written in the GOTO language with the additional property that it halts on all inputs. Such programs will be called *halting programs*. Throughout this chapter, we will be interested mainly in computational problems of a special kind called *decision problems* that have a yes/no answer. We will modify our language slightly when dealing with decision problems. We will augment our instruction set to include ACCEPT and REJECT (and omit STOP). When the ACCEPT (REJECT) instruction is reached, the machine will output yes or 1 (no or 0) and halt.

6.2.1 Some Computational Problems

We will temporarily shift our focus from the tool for problem solving (the computer) to the problems themselves. Throughout this chapter, a computational problem refers to an input/output relationship. For example, consider the problem of squaring an integer input. This problem assigns to each integer (such as 22) its square (in this case 484). In technical terms, this input/output relationship defines a function. Therefore, solving a computational problem is the same as computing the function defined by the problem. When we say that an algorithm (or a program) solves a problem, what we mean is that, for all inputs, the program halts and produces the correct output. We will allow inputs of arbitrary size and place no restrictions. A reader with primary interest in software applications is apt to question the validity (or even the meaningfulness) of allowing inputs of arbitrary size because it makes the set of all *possible* inputs infinite, and thus unrealistic, in real-world programming. But there are no really good alternatives. Any finite bound is artificial and is likely to become obsolete as the technology and our requirements change. Also, in practice, we do not know how to take advantage of restrictions on the size of the inputs. (See the discussion about nonuniform models in Section 6.5.) Problems (functions) that can be solved by an algorithm (or a halting GOTO program) are called *computable*.

As already remarked, we are interested mainly in decision problems. A decision problem is said to be decidable if there is a halting GOTO program that solves it correctly on all inputs. An important class of problems called **partially decidable decision problems** can be defined by relaxing our requirement a little bit; a decision problem is partially decidable if there is a GOTO program that halts and outputs 1 on all inputs for which the output should be 1 and either halts and outputs 0 or loops forever on the other inputs.

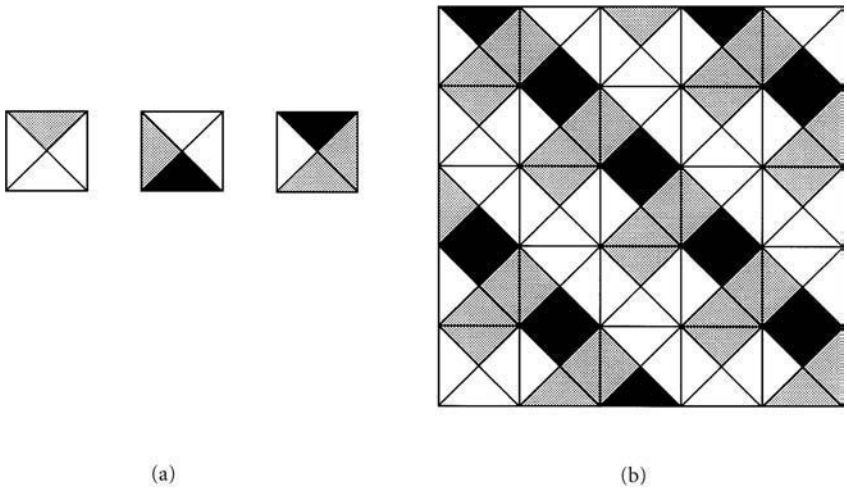


FIGURE 6.2 An example of tiling.

This means that the program may never give a wrong answer but is not required to halt on negative inputs (i.e., inputs with 0 as output).

We now list some problems that are fundamental either because of their inherent importance or because of their historical roles in the development of computation theory:

Problem 1 (halting problem). The input to this problem is a program P in the GOTO language and a binary string x. The expected output is 1 (or yes) if the program P halts when run on the input x, 0 (or no) otherwise.

Problem 2 (universal computation problem). A related problem takes as input a program P and an input x and produces as output what (if any) P would produce on input x. (Note that this is a decision problem if P is restricted to a yes/no program.)

Problem 3 (string compression). For a string x, we want to find the shortest program in the GOTO language that when started with the empty tape (i.e., tape containing one B symbol) halts and prints x. Here shortest means the total number of symbols in the program is as small as possible.

Problem 4 (tiling). A tile* is a square card of unit size (i.e., 1×1) divided into four quarters by two diagonals, each quarter colored with some color (selected from a finite set of colors). The tiles have fixed orientation and cannot be rotated. Given some finite set T of such tiles as input, the program is to determine if finite rectangular areas of all sizes (i.e., $k \times m$ for all positive integers k and m) can be tiled using only the given tiles such that the colors on any two touching edges are the same. It is assumed that an unlimited number of cards of each type is available. Figure 6.2(b) shows how the base set of tiles given in Figure 6.2(a) can be used to tile a 5×5 square area.

Problem 5 (linear programming). Given a system of linear inequalities (called constraints), such as $3x - 4y \leq 13$ with integer coefficients, the goal is to find if the system has a solution satisfying all of the constraints.

Some remarks must be made about the preceding problems. The problems in our list include nonnumerical problems and *meta problems*, which are problems about other problems. The first two problems are motivated by a quest for reliable program design. An algorithm for problem 1 (if it exists) can be used to test if a program contains an infinite loop. Problem 2 is motivated by an attempt to design a **universal**

*More precisely, a Wang tile, after Hao Wang, who wrote the first research paper on it.

algorithm, which can simulate any other. This problem was first attempted by Babbage, whose analytical engine had many ingredients of a modern electronic computer (although it was based on mechanical devices). Problem 3 is an important problem in information theory and arises in the following setting. Physical theories are aimed at creating simple laws to explain large volumes of experimental data. A famous example is Kepler's laws, which explained Tycho Brahe's huge and meticulous observational data. Problem 3 asks if this compression process can be automated. When we allow the inference rules to be sufficiently strong, this problem becomes **undecidable**. We will not discuss this problem further in this section but will refer the reader to some related formal systems discussed in Li and Vitányi [1993]. The tiling problem is not merely an interesting puzzle. It is an art form of great interest to architects and painters. Tiling has recently found applications in crystallography. Linear programming is a problem of central importance in economics, game theory, and operations research.

In the remainder of the section, we will present some basic algorithm design techniques and sketch how these techniques can be used to solve some of the problems listed (or their special cases). The main purpose of this discussion is to present techniques for showing the decidability (or partial decidability) of these problems. The reader can learn more advanced techniques of algorithm design in some later sections of this chapter as well as in many later chapters of this volume.

6.2.1.1 Table Lookup

The basic idea is to create a table for a function f, which needs to be computed by tabulating in one column an input x and the corresponding $f(x)$ in a second column. Then the table itself can be used as an algorithm. This method cannot be used directly because the set of all inputs is infinite. Therefore, it is not very useful, although it can be made to work in conjunction with the technique described subsequently.

6.2.1.2 Bounding the Search Domain

The difficulty of establishing the decidability of a problem is usually caused by the fact that the object we are searching for may have no known upper limit. Thus, if we can place such an upper bound (based on the structure of the problem), then we can reduce the search to a finite domain. Then table lookup can be used to complete the search (although there may be better methods in practice). For example, consider the following special case of the tiling problem: Let k be a fixed integer, say 1000. Given a set of tiles, we want to determine whether all rectangular rooms of shape $k \times n$ can be tiled for all n. (Note the difference between this special case and the general problem. The general one allows k and n both to have unbounded value. But here we allow only n to be unbounded.) It can be shown (see Section 6.5 for details) that there are two bounds n_0 and n_1 (they depend on k) such that if there is at least one tile of size $k \times t$ that can be tiled for some $n_0 \leq t \leq n_1$ then every tile of size $k \times n$ can be tiled. If no $k \times t$ tile can be tiled for any t between n_0 and n_1, then obviously the answer is no. Thus, we have reduced an infinite search domain to a finite one.

As another example, consider the linear programming problem. The set of possible solutions to this problem is infinite, and thus a table search cannot be used. But it is possible to reduce the search domain to a finite set using the geometric properties of the set of solutions of the linear programming problem. The fact that the set of solutions is convex makes the search especially easy.

6.2.1.3 Use of Subroutines

This is more of a program design tool than a tool for algorithm design. A central concept of programming is repetitive (or iterative) computation. We already observed how GOTO statements can be used to perform a sequence of steps repetitively. The idea of a subroutine is another central concept of programming. The idea is to make use of a program P itself as a single step in another program Q. Building programs from simpler programs is a natural way to deal with the complexity of programming tasks. We will illustrate the idea with a simple example. Consider the problem of multiplying two positive integers i and j. The input to the problem will be the form $11\ldots1011\ldots1$ (i 1s followed by a 0, followed by j 1s) and the output will be $i * j$ 1s (with possibly some 0s on either end). We will use the notation 1^i01^j to denote the starting configuration of the tape. This just means that the tape contains i 1s followed by a 0 followed by j 1s.

TABLE 6.1 Coding the GOTO Instructions

Instruction	Code
PRINT i	0001^{i+1}
GO LEFT	001
GO RIGHT	010
GO TO j IF i IS SCANNED	$0111^{j}01^{i+1}$
STOP	100

The basic idea behind a GOTO program for this problem is simple; add j 1s on the right end of tape exactly $i-1$ times and then erase the original sequence of i 1s on the left. A little thought reveals that the subroutine we need here is to duplicate a string of 1s so that if we start with $x02^k1^j$ a call to the subroutine will produce $x02^{k+j}1^j$. Here x is just any sequence of symbols. Note the role played by the symbol 2. As new 1s are created on the right, the old 1s change to 2s. This will ensure that there are exactly j 1s on the right end of the tape all of the time. This duplication subroutine is very similar to the doubling program, and the reader should have very little difficulty writing this program. Finally, the multiplication program can be done using the copy subroutine $(i-1)$ times.

6.2.2 A Universal Algorithm

We will now present in some detail a (partial) solution to problem 2 by arguing that there is a program U written in the GOTO language, which takes as input a program P (also written using the GOTO language) and an input x and produces as output $P(x)$, the output of P on input x. For convenience, we will assume that all programs written in the GOTO language use a fixed alphabet containing just 0, 1, and B. Because we have assumed this for all programs in the GOTO language, we should first address the issue of how an input to program U will look. We cannot directly place a program P on the tape because the alphabet used to write the program P uses letters G, O, T, O, etc. This minor problem can be easily circumvented by coding. The idea is to represent each instruction using only 0 and 1. One such coding scheme is shown in Table 6.1.

To encode an entire program, we simply write down in order (without the line numbers) the code for each instruction as given in the table. For example, here is the code for the doubling program shown in Figure 6.1:

000100101111011000110100111111101100011010011101 1100

Note that the encoded string contains all of the information about the program so that the encoding is completely reversible. From now on, if P is a program in the GOTO language, then code(P) will denote its binary code as just described. When there is no confusion, we will identify P and code(P). Before proceeding further, the reader may want to test his/her understanding of the encoding/decoding process by decoding the following string: 010011101100.

The basic idea behind the construction of a universal algorithm is simple, although the details involved in actually constructing one are enormous. We will present the central ideas and leave out the actual construction. Such a construction was carried out in complete detail by Turing himself and was simplified by others.* U has as its input code(P) followed by the string x. U simulates the computational steps of P on input x. It divides the input tape into three segments, one containing the program P, the second one essentially containing the contents of the tape of P as it changes with successive moves, and the third one containing the line number in program P of the instruction being currently simulated (similar to a *program counter* in an actual computer).

*A particularly simple exposition can be found in Robinson [1991].

We now describe a *cycle* of computation by U, which is similar to a central processing unit (CPU) cycle in a real computer. A single instruction of P is implemented by U in one cycle. First, U should know which location on the tape that P is currently reading. A simple artifact can handle this as follows: U uses in its tape alphabet two special symbols $0'$ and $1'$. U stores the tape of P in the tape segment alluded to in the previous paragraph exactly as it would appear when the program P is run on the input x with one minor modification. The symbol currently being read by program P is stored as the *primed version* ($0'$ is the primed version of 0, etc.). As an example, suppose after completing 12 instructions, P is reading the fourth symbol (from left) on its tape containing 01001001. Then the tape region of U after 12 cycles looks like $0100'1001$. At the beginning of a new cycle, U uses a subroutine to move to the region of the tape that contains the ith instruction of program P where i is the value of the program counter. It then decodes the ith instruction. Based on what type it is, U proceeds as follows: If it is a PRINT i instruction, then U scans the tape until the unique primed symbol in the *tape region* is reached and rewrites it as instructed. If it is a GO LEFT or GO RIGHT symbol, U locates the primed symbol, unprimes it, and primes its left or right neighbor, as instructed. In both cases, U returns to the program counter and increments it. If the instruction is GO TO i IF j IS SCANNED, U reads the primed symbol, and if it is j', U changes the program counter to i. This completes a cycle. Note that the three regions may grow and contract while U executes the cycles of computation just described. This may result in one of them running into another. U must then shift one of them to the left or right and make room as needed.

It is not too difficult to see that all of the steps described can be done using the instructions of the GOTO language. The main point to remember is that these actions will have to be coded as a single program, which has nothing whatsoever to do with program P. In fact, the program U is totally independent of P. If we replace P with some other program Q, it should simulate Q as well. The preceding argument shows that problem 2 is partially decidable. But it does not show that this problem is decidable. Why? It is because U may not halt on all inputs; specifically, consider an input consisting of a program P and a string x such that P does not halt on x. Then U will also keep executing cycle after cycle the moves of P and will never halt. In fact, in Section 6.3, we will show that problem 2 is not decidable.

6.3 Undecidability

Recall the definition of an undecidable problem. In this section, we will establish the undecidability of Problem 2, Section 6.2. The simplest way to establish the existence of undecidable problems is as follows: There are more problems than there are programs, the former set being uncountable, whereas the latter is countably infinite.* But this argument is purely existential and does not identify any specific problem as undecidable. In what follows, we will show that Problem 2 introduced in Section 6.2 is one such problem.

6.3.1 Diagonalization and Self-Reference

Undecidability is inextricably tied to the concept of self-reference, and so we begin by looking at this rather perplexing and sometimes paradoxical concept. The idea of self-reference seems to be many centuries old and may have originated with a barber in ancient Greece who had a sign board that read: "I shave all those who do not shave themselves." When the statement is applied to the barber himself, we get a self-contradictory statement. Does he shave himself? If the answer is yes, then he is one of those who shaves himself, and so the barber should not shave him. The contrary answer no is equally untenable. So neither yes nor no seems to be the correct answer to the question; this is the essence of the paradox. The barber's

*The reader who does not know what countable and uncountable infinities are can safely ignore this statement; the rest of the section does not depend on it.

paradox has made entry into modern mathematics in various forms. We will present some of them in the next few paragraphs.*

The first version, called Berry's paradox, concerns English descriptions of natural numbers. For example, the number 7 can be described by many different phrases: seven, six plus one, the fourth smallest prime, etc. We are interested in the *shortest* of such descriptions, namely, the one with the fewest letters in it. Clearly there are (infinitely) many positive integers whose shortest descriptions exceed 100 letters. (A simple counting argument can be used to show this. The set of positive integers is infinite, but the set of positive integers with English descriptions in fewer than or equal to 100 letters is finite.) Let D denote the set of positive integers that do not have English descriptions with fewer than 100 letters. Thus, D is not empty. It is a well-known fact in set theory that any nonempty subset of positive integers has a smallest integer. Let x be the smallest integer in D. Does x have an English description with fewer than or equal to 100 letters? By the definition of the set D and x, we have: x is "the smallest positive integer that cannot be described in English in fewer than 100 letters." This is clearly absurd because part of the last sentence in quotes is a description of x and it contains fewer than 100 letters in it. A similar paradox was found by the British mathematician Bertrand Russell when he considered the set of all sets that do not include themselves as elements, that is, $S = \{x \mid x \notin x\}$. The question "Is $S \in S$?" leads to a similar paradox.

As a last example, we will consider a charming self-referential paradox due to mathematician William Zwicker. Consider the collection of all two-person games (such as chess, tic-tac-toe, etc.) in which players make alternate moves until one of them loses. Call such a game *normal* if it has to end in a finite number of moves, no matter what strategies the two players use. For example, tic-tac-toe must end in at most nine moves and so it is normal. Chess is also normal because the 50-move rule ensures that the game cannot go forever. Now here is *hypergame*. In the first move of the hypergame, the first player calls out a normal game, and then the two players go on to play the game, with the second player making the first move. The question is: "Is hypergame normal?" Suppose it is normal. Imagine two players playing hypergame. The first player can call out hypergame (since it is a normal game). This makes the second player call out the name of a normal game, hypergame can be called out again and they can keep saying hypergame without end, and this contradicts the definition of a normal game. On the other hand, suppose it is not a normal game. But now in the first move, player 1 cannot call out hypergame and would call a normal game instead, and so the infinite move sequence just given is not possible, and so hypergame is normal after all!

In the rest of the section, we will show how these paradoxes can be modified to give nonparadoxical but surprising conclusions about the decidability of certain problems. Recall the encoding we presented in Section 6.2 that encodes any program written in the GOTO language as a binary string. Clearly this encoding is reversible in the sense that if we start with a program and encode it, it is possible to decode it back to the program. However, not every binary string corresponds to a program because there are many strings that cannot be decoded in a meaningful way, for example, 11010011000110. For the purposes of this section, however, it would be convenient if we can treat *every* binary string as a program. Thus, we will simply stipulate that any undecodable string be decoded to the program containing the single statement

```
1. REJECT
```

In the following discussion, we will identify a string x with a GOTO program to which it decodes. Now define a function f_D as follows: $f_D(x) = 1$ if x, decoded into a GOTO program, does not halt when started with x itself as the input. Note the self-reference in this definition. Although the definition of f_D seems artificial, its importance will become clear in the next section when we use it to show the undecidability of Problem 2. First we will prove that f_D is not computable. Actually, we will prove a stronger statement, namely, that f_D is not even partially decidable. [Recall that a function is partially decidable if there is a GOTO

*The most enchanting discussions of self-reference are due to the great puzzlist and mathematician R. Smullyan who brings out the breadth and depth of this concept in such delightful books as *What is the name of this book?* published by Prentice–Hall in 1978 and *Satan, Cantor, and Infinity* published by Alfred A. Knopf in 1992. We heartily recommend them to anyone who wants to be amused, entertained, and, more importantly, educated on the intricacies of mathematical logic and computability.

program (not necessarily halting) that computes it. An important distinction between computable and semicomputable functions is that a GOTO program for the latter need not halt on inputs with output = 0.]

Theorem 6.1 *Function f_D is not partially decidable.*

The proof is by contradiction. Suppose a GOTO program P' computes the function f_D. We will modify P' into another program P in the GOTO language such that P computes the same function as P' but has the additional property that it will never terminate its computation by ending up in a REJECT statement.* Thus, P is a program with the property that it computes f_D and halts on an input y if and only if $f_D(y) = 1$. We will complete the proof by showing that there is at least one input in which the program produces a wrong output, that is, there is an x such that $f_D(x) \neq P(x)$.

Let x be the encoding of program P. Now consider the question: Does P halt when given x as input? Suppose the answer is yes. Then, by the way we constructed P, here $P(x) = 1$. On the other hand, the definition of f_D implies that $f_D(x) = 0$. (This is the punch line in this proof. We urge the reader to take a few moments and read the definition of f_D a few times and make sure that he or she is convinced about this fact!) Similarly, if we start with the assumption that $P(x) = 0$, we are led to the conclusion that $f_D(x) = 1$. *In both cases,* $f_D(x) \neq P(x)$ and thus P is not the correct program for f_D. Therefore, P' is not the correct program for f_D either because P and P' compute the same function. This contradicts the hypothesis that such a program exists, and the proof is complete.

Note the crucial difference between the paradoxes we presented earlier and the proof of this theorem. Here we do not have a paradox because our conclusion is of the form $f_D(x) = 0$ if and only if $P(x) = 1$ and not $f_D(x) = 1$ if and only if $f_D(x) = 0$. But in some sense, the function f_D was motivated by Russell's paradox. We can similarly create another function f_Z (based on Zwicker's paradox of hypergame). Let f be any function that maps binary strings to $\{0, 1\}$. We will describe a method to generate successive functions f_1, f_2, etc., as follows: Suppose $f(x) = 0$ for all x. Then we cannot create any more functions, and the sequence stops with f. On the other hand, if $f(x) = 1$ for some x, then choose one such x and decode it as a GOTO program. This defines another function; call it f_1 and repeat the same process with f_1 in the place of f. We call f a normal function if no matter how x is selected at each step, the process terminates after a finite number of steps. A simple example of a nonnormal function is as follows: Suppose $P(Q) = 1$ for some program P and input Q and at the same time $Q(P) = 1$ (note that we are using a program and its code interchangeably), then it is easy to see that the functions defined by both P and Q are not normal. Finally, define $f_Z(X) = 1$ if X is a normal program, 0 if it is not. We leave it as an instructive exercise to the reader to show that f_Z is not semicomputable. A perceptive reader will note the connection between Berry's paradox and problem 3 in our list (string compression problem) just as f_Z is related to Zwicker's paradox. Such a reader should be able to show the undecidability of problem 3 by imitating Berry's paradox.

6.3.2 Reductions and More Undecidable Problems

Theory of computation deals not only with the behavior of individual problems but also with relations among them. A **reduction** is a simple way to relate two problems so that we can deduce the (un)decidability of one from the (un)decidability of the other. Reduction is similar to using a subroutine. Consider two problems A and B. We say that problem A can be reduced to problem B if there is an algorithm for B provided that A has one. To define the reduction (also called a *Turing reduction*) precisely, it is convenient to augment the instruction set of the GOTO programming language to include a new instruction CALL X, i, j where X is a (different) GOTO program, and i and j are line numbers. In detail, the execution of such augmented programs is carried out as follows: When the computer reaches the instruction CALL X,

*The modification needed to produce P from P' is straightforward. If P' did not have any REJECT statements at all, then no modification would be needed. If it had, then we would have to replace each one by a looping statement, which keeps repeating the same instruction forever.

i, j, the program will simply start executing the instructions of the program from line 1, treating whatever is on the tape currently as the input to the program X. When (if at all) X finishes the computation by reaching the ACCEPT statement, the execution of the original program continues at line number i and, if it finishes with REJECT, the original program continues from line number j.

We can now give a more precise definition of a reduction between two problems. Let A and B be two computational problems. We say that A is reducible to B if there is a halting program Y in the GOTO language for problem A in which calls can be made to a halting program X for problem B. The algorithm for problem A described in the preceding reduction does not assume the availability of program X and cannot use the details behind the design of this algorithm. The right way to think about a reduction is as follows: Algorithm Y, from time to time, needs to know the solutions to different instances of problem B. It can query an algorithm for problem B (as a black box) and use the answer to the query for making further decisions. An important point to be noted is that the program Y actually can be implemented even if program X was never built as long as someone can correctly answer some questions asked by program Y about the output of problem B for certain inputs. Programs with such calls are sometimes called *oracle programs*. Reduction is rather difficult to assimilate at the first attempt, and so we will try to explain it using a puzzle. How do you play two chess games, one each with Kasparov and Anand (perhaps currently the world's two best players) and ensure that you get at least one point? (You earn one point for a win, 0 for a loss, and 1/2 for a draw.) Because you are a novice and are pitted against two Goliaths, you are allowed a concession. You can choose to play white or black on either board. The well-known answer is the following: Take white against one player, say, Anand, and black against the other, namely, Kasparov. Watch the first move of Kasparov (as he plays white) and make the same move against Anand, get his reply and play it back to Kasparov and keep playing back and forth like this. It takes only a moment's thought that you are guaranteed to win (exactly) 1 point. The point is that your game involves taking the position of one game, applying the algorithm of one player, getting the result and applying it to the other board, etc., and you do not even have to know the rules of chess to do this. This is exactly how algorithm Y is required to use algorithm X.

We will use reductions to show the undecidability as follows: Suppose A can be reduced to B as in the preceding definition. If there is an algorithm for problem B, it can be used to design a program for A by essentially imitating the execution of the augmented program for A (with calls to the oracle for B) as just described. But we will turn it into a negative argument as follows: If A is undecidable, then so is B. Thus, a reduction from a problem known to be undecidable to problem B will prove B's undecidability.

First we define a new problem, Problem 2′, which is a special case of Problem 2. Recall that in Problem 2 the input is (the code of) a program P in GOTO language and a string x. The output required is $P(x)$. In Problem 2′, the input is (only) the code of a program P and the output required is $P(P)$, that is, instead of requiring P to run on a given input, this problem requires that it be run on its own code. This is clearly a special case of problem 2. The reader may readily see the self-reference in Problem 2′ and suspect that it may be undecidable; therefore, the more general Problem 2 may be undecidable as well. We will establish these claims more rigorously as follows.

We first observe a general statement about the decidability of a function f (or problem) and its *complement*. The complement function is defined to take value 1 on all inputs for which the original function value is 0 and vice versa. The statement is that a function f is decidable if and only if the complement $\bar{f}$ is decidable. This can be easily proved as follows. Consider a program P that computes f. Change P into $\bar{P}$ by interchanging all of the ACCEPT and REJECT statements. It is easy to see that $\bar{P}$ actually computes $\bar{f}$. The converse also is easily seen to hold. It readily follows that the function defined by problem 2′ is undecidable because it is, in fact, the complement of f_D.

Finally, we will show that problem 2 is uncomputable. The idea is to use a reduction from problem 2′ to problem 2. (Note the direction of reduction. This always confuses a beginner.) Suppose there is an algorithm for problem 2. Let X be the GOTO language program that implements this algorithm. X takes as input code(P) (for any program P) followed by x, produces the result $P(x)$, and halts. We want to design a program Y that takes as input code(P) and produce the output $P(P)$ using calls to program X. It is clear what needs to be done. We just create the input in proper form code(P) followed by code(P) and call X. This requires first duplicating the input, but this is a simple programming task similar to the

one we demonstrated in our first program in Section 6.2. Then a call to X completes the task. This shows that Problem 2′ reduces to Problem 2, and thus the latter is undecidable as well.

By a more elaborate reduction (from f_D), it can be shown that tiling is not partially decidable. We will not do it here and refer the interested reader to Harel [1992]. But we would like to point out how the undecidability result can be used to infer a result about tiling. This deduction is of interest because the result is an important one and is hard to derive directly. We need the following definition before we can state the result. A different way to pose the tiling problem is whether a given set of tiles can tile *an entire plane* in such a way that all of the adjacent tiles have the same color on the meeting quarter. (Note that this question is different from the way we originally posed it: Can a given set of tiles tile any *finite* rectangular region? Interestingly, the two problems are identical in the sense that the answer to one version is yes if and only if it is yes for the other version.) Call a tiling of the plane periodic if one can identify a $k \times k$ square such that the entire tiling is made by repeating this $k \times k$ square tile. Otherwise, call it *aperiodic*. Consider the question: Is there a (finite) set of unit tiles that can tile the plane, but only aperiodically? The answer is yes and it can be shown from the total undecidability of the tiling problem. Suppose the answer is no. Then, for any given set of tiles, the entire plane can be tiled if and only if the plane can be tiled periodically. But a periodic tiling can be found, if one exists, by trying to tile a $k \times k$ region for successively increasing values of k. This process will eventually succeed (in a finite number of steps) if the tiling exists. This will make the tiling problem partially decidable, which contradicts the total undecidability of the problem. This means that the assumption that the entire plane can be tiled if and only if some $k \times k$ region can be tiled is wrong. Thus, there exists a (finite) set of tiles that can tile the entire plane, but only aperiodically.

6.4 Formal Languages and Grammars

The universe of strings is probably the most general medium for the representation of information. This section is concerned with sets of strings called *languages* and certain systems generating these languages such as *grammars*. Every programming language including Pascal, C, or Fortran can be precisely described by a grammar. Moreover, the grammar allows us to write a computer program (called the lexical analyzer in a compiler) to determine if a piece of code is syntactically correct in the programming language. Would not it be nice to also have such a grammar for English and a corresponding computer program which can tell us what English sentences are grammatically correct?* The focus of this brief exposition is the formalism and mathematical properties of various languages and grammars. Many of the concepts have applications in domains including natural language and computer language processing, string matching, etc. We begin with some standard definitions about languages.

Definition 6.1 An *alphabet* is a finite nonempty set of *symbols*, which are assumed to be *indivisible*.

For example, the alphabet for English consists of 26 uppercase letters $A, B, \ldots, Z$ and 26 lowercase letters $a, b, \ldots, z$. We usually use the symbol Σ to denote an alphabet.

Definition 6.2 A *string* over an alphabet Σ is a finite sequence of symbols of Σ.

The number of symbols in a string x is called its *length*, denoted $|x|$. It is convenient to introduce an empty string, denoted ϵ, which contains no symbols at all. The length of ϵ is 0.

Definition 6.3 Let $x = a_1 a_2 \cdots a_n$ and $y = b_1 b_2 \cdots b_m$ be two strings. The *concatenation* of x and y, denoted xy, is the string $a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m$.

*Actually, English and the other natural languages have grammars; but these grammars are not precise enough to tell apart the correct and incorrect sentences with 100% accuracy. The main problem is that *there is no universal agreement* on what are grammatically correct English sentences.

Thus, for any string x, $\epsilon x = x\epsilon = x$. For any string x and integer $n \geq 0$, we use x^n to denote the string formed by sequentially concatenating n copies of x.

Definition 6.4 The set of all strings over an alphabet Σ is denoted Σ^* and the set of all nonempty strings over Σ is denoted Σ^+. The empty set of strings is denoted $\emptyset$.

Definition 6.5 For any alphabet Σ, a *language* over Σ is a set of strings over Σ. The members of a language are also called the *words* of the language.

Example 6.1

The sets $L_1 = \{01, 11, 0110\}$ and $L_2 = \{0^n 1^n \mid n \geq 0\}$ are two languages over the binary alphabet $\{0, 1\}$. The string 01 is in both languages, whereas 11 is in L_1 but not in L_2.

Because languages are just sets, standard set operations such as union, intersection, and complementation apply to languages. It is useful to introduce two more operations for languages: *concatenation* and *Kleene closure*.

Definition 6.6 Let L_1 and L_2 be two languages over Σ. The concatenation of L_1 and L_2, denoted $L_1 L_2$, is the language $\{xy \mid x \in L_1, y \in L_2\}$.

Definition 6.7 Let L be a language over Σ. Define $L^0 = \{\epsilon\}$ and $L^i = L L^{i-1}$ for $i \geq 1$. The Kleene closure of L, denoted L^*, is the language

$$L^* = \bigcup_{i \geq 0} L^i$$

and the *positive closure* of L, denoted L^+, is the language

$$L^+ = \bigcup_{i \geq 1} L^i$$

In other words, the Kleene closure of language L consists of all strings that can be formed by concatenating some words from L. For example, if $L = \{0, 01\}$, then $LL = \{00, 001, 010, 0101\}$ and L^* includes all binary strings in which every 1 is preceded by a 0. L^+ is the same as L^* except it excludes ϵ in this case. Note that, for any language L, L^* always contains ϵ and L^+ contains ϵ if and only if L does. Also note that Σ^* is in fact the Kleene closure of the alphabet Σ when viewed as a language of words of length 1, and Σ^+ is just the positive closure of Σ.

6.4.1 Representation of Languages

In general, a language over an alphabet Σ is a subset of Σ^*. How can we describe a language rigorously so that we know if a given string belongs to the language or not? As shown in the preceding paragraphs, a finite language such as L_1 in Example 6.1 can be explicitly defined by enumerating its elements, and a simple infinite language such as L_2 in the same example can be described using a rule characterizing all members of L_2. It is possible to define some more systematic methods to represent a wide class of languages. In the following, we will introduce three such methods: regular expressions, pattern systems, and grammars. The languages that can be described by this kind of system are often referred to as *formal languages*.

Definition 6.8 Let Σ be an alphabet. The *regular expressions* over Σ and the languages they represent are defined inductively as follows.

1. The symbol $\emptyset$ is a regular expression, denoting the empty set.
2. The symbol ϵ is a regular expression, denoting the set $\{\epsilon\}$.

3. For each $a \in \Sigma$, a is a regular expression, denoting the set $\{a\}$.
4. If r and s are regular expressions denoting the languages R and S, then $(r + s)$, (rs), and (r^*) are regular expressions that denote the sets $R \cup S$, RS, and R^*, respectively.

For example, $((0(0 + 1)^*) + ((0 + 1)^*0))$ is a regular expression over $\{0, 1\}$, and it represents the language consisting of all binary strings that begin or end with a 0. Because the set operations union and concatenation are both associative, many parentheses can be omitted from regular expressions if we assume that Kleene closure has higher precedence than concatenation and concatenation has higher precedence than union. For example, the preceding regular expression can be abbreviated as $0(0 + 1)^* + (0 + 1)^*0$. We will also abbreviate the expression rr^* as r^+. Let us look at a few more examples of regular expressions and the languages they represent.

Example 6.2

The expression $0(0 + 1)^*1$ represents the set of all strings that begin with a 0 and end with a 1.

Example 6.3

The expression $0 + 1 + 0(0 + 1)^*0 + 1(0 + 1)^*1$ represents the set of all nonempty binary strings that begin and end with the same bit.

Example 6.4

The expressions 0^*, 0^*10^*, and $0^*10^*10^*$ represent the languages consisting of strings that contain no 1, exactly one 1, and exactly two 1s, respectively.

Example 6.5

The expressions $(0 + 1)^*1(0 + 1)^*1(0 + 1)^*$, $(0 + 1)^*10^*1(0 + 1)^*$, $0^*10^*1(0 + 1)^*$, and $(0 + 1)^*10^*10^*$ all represent the same set of strings that contain at least two 1s.

For any regular expression r, the language represented by r is denoted as $L(r)$. Two regular expressions representing the same language are called *equivalent*. It is possible to introduce some identities to algebraically manipulate regular expressions to construct equivalent expressions, by tailoring the set identities for the operations union, concatenation, and Kleene closure to regular expressions. For more details, see Salomaa [1966]. For example, it is easy to prove that the expressions $r(s + t)$ and $rs + rt$ are equivalent and $(r^*)^*$ is equivalent to r^*.

Example 6.6

Let us construct a regular expression for the set of all strings that contain no consecutive 0s. A string in this set may begin and end with a sequence of 1s. Because there are no consecutive 0s, every 0 that is not the last symbol of the string must be followed by at least a 1. This gives us the expression $1^*(01^+)^*1^*(\epsilon + 0)$. It is not hard to see that the second 1^* is redundant, and thus the expression can in fact be simplified to $1^*(01^+)^*(\epsilon + 0)$.

Regular expressions were first introduced in Kleene [1956] for studying the properties of neural nets. The preceding examples illustrate that regular expressions often give very clear and concise representations of languages. Unfortunately, not every language can be represented by regular expressions. For example, it will become clear that there is no regular expression for the language $\{0^n1^n \mid n \geq 1\}$. The languages represented by regular expressions are called the **regular languages**. Later, we will see that regular languages are exactly the class of languages generated by the so-called **right-linear grammars**. This connection allows one to prove some interesting mathematical properties about regular languages as well as to design an efficient algorithm to determine whether a given string belongs to the language represented by a given **regular expression**.

Another way of representing languages is to use *pattern systems* [Angluin 1980, Jiang et al. 1995].

Definition 6.9 A *pattern system* is a triple (Σ, V, p), where Σ is the alphabet, V is the set of *variables* with $\Sigma \cap V = \emptyset$, and p is a string over $\Sigma \cup V$ called the *pattern*.

An example pattern system is $(\{0,1\}, \{v_1, v_2\}, v_1 v_1 0 v_2)$.

Definition 6.10 The language generated by a pattern system (Σ, V, p) consists of all strings over Σ that can be obtained from p by replacing each variable in p with a string over Σ.

For example, the language generated by $(\{0,1\}, \{v_1, v_2\}, v_1 v_1 0 v_2)$ contains words $0, 00, 01, 000, 001$, $010, 011, 110$, etc., but does not contain strings, $1, 10, 11, 100, 101$, etc. The pattern system $(\{0,1\}, \{v_1\}$, $v_1 v_1)$ generates the set of all strings, which is the concatenation of two equal substrings, that is, the set $\{xx \mid x \in \{0,1\}^*\}$. The languages generated by pattern systems are called the *pattern languages*.

Regular languages and pattern languages are really different. One can prove that the pattern language $\{xx \mid x \in \{0,1\}^*\}$ is not a regular language and the set represented by the regular expression 0^*1^* is not a pattern language. Although it is easy to write an algorithm to decide if a string is in the language generated by a given pattern system, such an algorithm most likely would have to be very inefficient [Angluin 1980].

Perhaps the most useful and general system for representing languages is based on grammars, which are extensions of the pattern systems.

Definition 6.11 A grammar is a quadruple (Σ, N, S, P), where:

1. Σ is a finite nonempty set called the alphabet. The elements of Σ are called the *terminals*.
2. N is a finite nonempty set disjoint from Σ. The elements of N are called the *nonterminals* or *variables*.
3. $S \in N$ is a distinguished nonterminal called the *start symbol*.
4. P is a finite set of *productions* (or *rules*) of the form

$$\alpha \to \beta$$

where $\alpha \in (\Sigma \cup N)^* N (\Sigma \cup N)^*$ and $\beta \in (\Sigma \cup N)^*$, that is, α is a string of terminals and nonterminals containing at least one nonterminal and β is a string of terminals and nonterminals.

Example 6.7

Let $G_1 = (\{0,1\}, \{S, T, O, I\}, S, P)$, where P contains the following productions:

$$\begin{aligned} S &\to OT \\ S &\to OI \\ T &\to SI \\ O &\to 0 \\ I &\to 1 \end{aligned}$$

As we shall see, the grammar G_1 can be used to describe the set $\{0^n 1^n \mid n \geq 1\}$.

Example 6.8

Let $G_2 = (\{0,1,2\}, \{S, A\}, S, P)$, where P contains the following productions.

$$\begin{aligned} S &\to 0SA2 \\ S &\to \epsilon \\ 2A &\to A2 \\ 0A &\to 01 \\ 1A &\to 11 \end{aligned}$$

This grammar G_2 can be used to describe the set $\{0^n 1^n 2^n \geq n \geq 0\}$.

Example 6.9

To construct a grammar G_3 to describe English sentences, the alphabet Σ contains all words in English. N would contain nonterminals, which correspond to the structural components in an English sentence, for example, ⟨sentence⟩, ⟨subject⟩, ⟨predicate⟩, ⟨noun⟩, ⟨verb⟩, ⟨article⟩, etc. The start symbol would be ⟨sentence⟩. Some typical productions are

$$\begin{aligned}
\langle \text{sentence} \rangle &\rightarrow \langle \text{subject} \rangle \langle \text{predicate} \rangle \\
\langle \text{subject} \rangle &\rightarrow \langle \text{noun} \rangle \\
\langle \text{predicate} \rangle &\rightarrow \langle \text{verb} \rangle \langle \text{article} \rangle \langle \text{noun} \rangle \\
\langle \text{noun} \rangle &\rightarrow \text{mary} \\
\langle \text{noun} \rangle &\rightarrow \text{algorithm} \\
\langle \text{verb} \rangle &\rightarrow \text{wrote} \\
\langle \text{article} \rangle &\rightarrow \text{an}
\end{aligned}$$

The rule ⟨sentence⟩ → ⟨subject⟩⟨predicate⟩ follows from the fact that a sentence consists of a subject phrase and a predicate phrase. The rules ⟨noun⟩ → mary and ⟨noun⟩ → algorithm mean that both mary and algorithms are possible nouns.

To explain how a grammar represents a language, we need the following concepts.

Definition 6.12 Let (Σ, N, S, P) be a grammar. A *sentential form* of G is any string of terminals and nonterminals, that is, a string over $\Sigma \cup N$.

Definition 6.13 Let (Σ, N, S, P) be a grammar and γ_1 and γ_2 two sentential forms of G. We say that γ_1 *directly derives* γ_2, denoted $\gamma_1 \Rightarrow \gamma_2$, if $\gamma_1 = \sigma\alpha\tau$, $\gamma_2 = \sigma\beta\tau$, and $\alpha \rightarrow \beta$ is a production in P.

For example, the sentential form $00S11$ directly derives the sentential form $00OT11$ in grammar G_1, and $A2A2$ directly derives $AA22$ in grammar G_2.

Definition 6.14 Let γ_1 and γ_2 be two sentential forms of a grammar G. We say that γ_1 *derives* γ_2, denoted $\gamma_1 \Rightarrow^* \gamma_2$, if there exists a sequence of (zero or more) sentential forms $\sigma_1, \ldots, \sigma_n$ such that

$$\gamma_1 \Rightarrow \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \gamma_2$$

The sequence $\gamma_1 \Rightarrow \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \gamma_2$ is called a derivation from γ_1 to γ_2.

For example, in grammar G_1, $S \Rightarrow^* 0011$ because

$$S \Rightarrow \underline{O}T \Rightarrow 0\underline{T} \Rightarrow 0S\underline{I} \Rightarrow 0\underline{S}1 \Rightarrow 0\underline{O}I1 \Rightarrow 00\underline{I}1 \Rightarrow 0011$$

and in grammar G_2, $S \Rightarrow^* 001122$ because

$$S \Rightarrow 0\underline{S}A2 \Rightarrow 00\underline{S}A2A2 \Rightarrow 00\underline{A}2A2 \Rightarrow 0012\underline{A}2 \Rightarrow 0011\underline{A}22 \Rightarrow 001122$$

Here the left-hand side of the relevant production in each derivation step is underlined for clarity.

Definition 6.15 Let (Σ, N, S, P) be a grammar. The language generated by G, denoted $L(G)$, is defined as

$$L(G) = \{x \mid x \in \Sigma^*, S \Rightarrow^* x\}$$

The words in $L(G)$ are also called the *sentences* of $L(G)$.

Clearly, $L(G_1)$ contains all strings of the form 0^n1^n, $n \geq 1$, and $L(G_2)$ contains all strings of the form $0^n1^n2^n$, $n \geq 0$. Although only a partial definition of G_3 is given, we know that $L(G_3)$ contains sentences such as "mary wrote an algorithm" and "algorithm wrote an algorithm" but does not contain sentences such as "an wrote algorithm."

The introduction of formal grammars dates back to the 1940s [Post 1943], although the study of rigorous description of languages by grammars did not begin until the 1950s [Chomsky 1956]. In the next subsection, we consider various restrictions on the form of productions in a grammar and see how these restrictions can affect the power of a grammar in representing languages. In particular, we will know that regular languages and pattern languages can all be generated by grammars under different restrictions.

6.4.2 Hierarchy of Grammars

Grammars can be divided into four classes by gradually increasing the restrictions on the form of the productions. Such a classification is due to Chomsky [1956, 1963] and is called the *Chomsky hierarchy*.

Definition 6.16 Let $G = (\Sigma, N, S, P)$ be a grammar.

1. G is also called a *type*-0 *grammar* or an *unrestricted grammar*.
2. G is *type*-1 or **context sensitive** if each production $\alpha \to \beta$ in P either has the form $S \to \epsilon$ or satisfies $|\alpha| \leq |\beta|$.
3. G is *type*-2 or **context free** if each production $\alpha \to \beta$ in P satisfies $|\alpha| = 1$, that is, α is a nonterminal.
4. G is *type*-3 or right linear or regular if each production has one of the following three forms:

$$A \to aB, \qquad A \to a, \qquad A \to \epsilon$$

where A and B are nonterminals and a is a terminal.

The language generated by a type-i is called a type-i language, $i = 0, 1, 2, 3$. A type-1 language is also called a **context-sensitive language** and a type-2 language is also called a **context-free language**. It turns out that every type-3 language is in fact a regular language, that is, it is represented by some regular expression, and vice versa. See the next section for the proof of the equivalence of type-3 (right-linear) grammars and regular expressions.

The grammars G_1 and G_3 given in the last subsection are context free and the grammar G_2 is context sensitive. Now we give some examples of unrestricted and right-linear grammars.

Example 6.10

Let $G_4 = (\{0, 1\}, \{S, A, O, I, T\}, S, P)$, where P contains

$$\begin{array}{ll} S \to AT & \\ A \to 0AO & A \to 1AI \\ O0 \to 0O & O1 \to 1O \\ I0 \to 0I & I1 \to 1I \\ OT \to 0T & IT \to 1T \\ A \to \epsilon & T \to \epsilon \end{array}$$

Then G_4 generates the set $\{xx \mid x \in \{0, 1\}^*\}$. For example, we can derive the word 0101 from S as follows:

$$S \Rightarrow \underline{A}T \Rightarrow 0\underline{A}OT \Rightarrow 01\underline{A}IOT \Rightarrow 01I\underline{OT} \Rightarrow 01\underline{I0}T \Rightarrow 010\underline{IT} \Rightarrow 0101\underline{T} \Rightarrow 0101$$

Example 6.11

We give a right-linear grammar G_5 to generate the language represented by the regular expression in Example 6.3, that is, the set of all nonempty binary strings beginning and ending with the same bit. Let $G_5 = (\{0, 1\}, \{S, O, I\}, S, P)$, where P contains

$$\begin{array}{ll} S \to 0O & S \to 1I \\ S \to 0 & S \to 1 \\ O \to 0O & O \to 1O \\ I \to 0I & I \to 1I \\ O \to 0 & I \to 1 \end{array}$$

The following theorem is due to Chomsky [1956, 1963].

Theorem 6.2 *For each $i = 0, 1, 2$, the class of type-i languages properly contains the class of type-$(i + 1)$ languages.*

For example, one can prove by using a technique called *pumping* that the set $\{0^n 1^n \mid n \geq 1\}$ is context free but not regular, and the sets $\{0^n 1^n 2^n \mid n \geq 0\}$ and $\{xx \mid x \in \{0, 1\}^*\}$ are context sensitive but not context free [Hopcroft and Ullman 1979]. It is, however, a bit involved to construct a language that is of type-0 but not context sensitive. See, for example, Hopcroft and Ullman [1979] for such a language.

The four classes of languages in the Chomsky hierarchy also have been completely characterized in terms of Turing machines and their restricted versions. We have already defined a Turing machine in Section 6.2. Many restricted versions of it will be defined in the next section. It is known that type-0 languages are exactly those recognized by Turing machines, context-sensitive languages are those recognized by Turing machines running in linear space, context-free languages are those recognized by Turing machines whose worktapes operate as pushdown stacks [called **pushdown automata** (PDA)], and regular languages are those recognized by Turing machines without any worktapes (called **finite-state machine** or **finite automata**) [Hopcroft and Ullman 1979].

Remark 6.1 Recall our definition of a Turing machine and the function it computes from Section 6.2. In the preceding paragraph, we refer to *a language recognized* by a Turing machine. These are two seemingly different ideas, but they are essentially the same. The reason is that the function f, which maps the set of strings over a finite alphabet to $\{0, 1\}$, corresponds in a natural way to the language L_f over Σ defined as: $L_f = \{x \mid f(x) = 1\}$. Instead of saying that a Turing machine computes the function f, we say equivalently that it recognizes L_f.

Because $\{xx \mid x \in \{0, 1\}^*\}$ is a pattern language, the preceding discussion implies that the class of pattern languages is not contained in the class of context-free languages. The next theorem shows that the class of pattern languages is contained in the class of context-sensitive languages.

Theorem 6.3 *Every pattern language is context sensitive.*

The theorem follows from the fact that every pattern language is recognized by a Turing machine in linear space [Angluin 1980] and linear space-bounded Turing machines recognize exactly context-sensitive languages. To show the basic idea involved, let us construct a context-sensitive grammar for the pattern language $\{xx \mid x \in \{0, 1\}^*\}$. The grammar G_4 given in Example 6.10 for this language is almost context-sensitive. We just have to get rid of the two ϵ-productions: $A \to \epsilon$ and $T \to \epsilon$. A careful modification of G_4 results in the following grammar $G_6 = (\{0, 1\}, \{S, A_0, A_1, O, I, T_0, T_1\}, S, P)$,

where P contains

$$
\begin{array}{ll}
S \to \epsilon & \\
S \to A_0 T_0 & S \to A_1 T_1 \\
A_0 \to 0 A_0 O & A_0 \to 1 A_0 I \\
A_1 \to 0 A_1 O & A_1 \to 1 A_1 I \\
A_0 \to 0 & A_1 \to 1 \\
O0 \to 0O & O1 \to 1O \\
I0 \to 0I & I1 \to 1I \\
O T_0 \to 0 T_0 & I T_0 \to 1 T_0 \\
O T_1 \to 0 T_1 & I T_1 \to 1 T_1 \\
T_0 \to O & T_1 \to 1,
\end{array}
$$

which is context sensitive and generates $\{xx \mid x \in \{0,1\}^*\}$. For example, we can derive 011011 as

$$
\begin{aligned}
&\Rightarrow \underline{A_1} T_1 \Rightarrow 0\underline{A_1} O T_1 \Rightarrow 01\underline{A_1} I O T_1 \\
&\Rightarrow 011 I \underline{O T_1} \Rightarrow 011\underline{I0} T_1 \Rightarrow 0110\underline{I T_1} \Rightarrow 01101\underline{T_1} \Rightarrow 011011
\end{aligned}
$$

For a class of languages, we are often interested in the so-called *closure properties* of the class.

Definition 6.17 A class of languages (e.g., regular languages) is said to be *closed* under a particular operation (e.g., union, intersection, complementation, concatenation, Kleene closure) if each application of the operation on language(s) of the class results in a language of the class.

These properties are often useful in constructing new languages from existing languages as well as proving many theoretical properties of languages and grammars. The closure properties of the four types of languages in the Chomsky hierarchy are now summarized [Harrison 1978, Hopcroft and Ullman 1979, Gurari 1989].

Theorem 6.4

1. *The class of type-*0 *languages is closed under union, intersection, concatenation, and Kleene closure but not under complementation.*
2. *The class of context-free languages is closed under union, concatenation, and Kleene closure but not under intersection or complementation.*
3. *The classes of context-sensitive and regular languages are closed under all five of the operations.*

For example, let $L_1 = \{0^m 1^n 2^p \mid m = n \text{ or } n = p\}$, $L_2 = \{0^m 1^n 2^p \mid m = n\}$, and $L_3 = \{0^m 1^n 2^p \mid n = p\}$. It is easy to see that all three are context-free languages. (In fact, $L_1 = L_2 \cup L_3$.) However, intersecting L_2 with L_3 gives the set $\{0^m 1^n 2^p \mid m = n = p\}$, which is not context free.

We will look at context-free grammars more closely in the next subsection and introduce the concept of **parsing** and ambiguity.

6.4.3 Context-Free Grammars and Parsing

From a practical point of view, for each grammar $G = (\Sigma, N, S, P)$ representing some language, the following two problems are important:

1. (Membership) Given a string over Σ, does it belong to $L(G)$?
2. (Parsing) Given a string in $L(G)$, how can it be derived from S?

The importance of the membership problem is quite obvious: given an English sentence or computer program we wish to know if it is grammatically correct or has the right format. Parsing is important because a derivation usually allows us to interpret the meaning of the string. For example, in the case of a Pascal program, a derivation of the program in Pascal grammar tells the compiler how the program should be executed. The following theorem illustrates the decidability of the membership problem for the four classes of grammars in the Chomsky hierarchy. The proofs can be found in Chomsky [1963], Harrison [1978], and Hopcroft and Ullman [1979].

Theorem 6.5 *The membership problem for type-0 grammars is undecidable in general and is decidable for any context-sensitive grammar* (*and thus for any context-free or right-linear grammars*).

Because context-free grammars play a very important role in describing computer programming languages, we discuss the membership and parsing problems for context-free grammars in more detail. First, let us look at another example of context-free grammar. For convenience, let us abbreviate a set of productions with the same left-hand side nonterminal

$$A \to \alpha_1, \ldots, A \to \alpha_n$$

as

$$A \to \alpha_1 \mid \cdots \mid \alpha_n$$

Example 6.12

We construct a context-free grammar for the set of all valid Pascal real values. In general, a real constant in Pascal has one of the following forms:

$$m.n, \qquad m\mathbf{e}q, \qquad m.n\mathbf{e}q,$$

where m and q are signed or unsigned integers and n is an unsigned integer. Let $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \mathbf{e}, +, -, .\}$, $N = \{S, M, N, D\}$, and the set P of the productions contain

$$\begin{aligned} S &\to M.N|M\mathbf{e}M|M.N\mathbf{e}M \\ M &\to N| + N| - N \\ N &\to DN|D \\ D &\to 0|1|2|3|4|5|7|8|9 \end{aligned}$$

Then the grammar generates all valid Pascal real values (including some absurd ones like 001.200**e**000). The value 12.3**e** − 4 can be derived as

$$S \Rightarrow \underline{M}.N\mathbf{e}M \Rightarrow \underline{N}.N\mathbf{e}M \Rightarrow \underline{D}N.N\mathbf{e}M \Rightarrow 1\underline{N}.N\mathbf{e}M \Rightarrow 1\underline{D}.N\mathbf{e}M$$

$$\Rightarrow 12.\underline{N}\mathbf{e}M \Rightarrow 12.\underline{D}\mathbf{e}M \Rightarrow 12.3\mathbf{e}\underline{M} \Rightarrow 12.3\mathbf{e} - \underline{N} \Rightarrow 12.3\mathbf{e} - \underline{D} \Rightarrow 12.3\mathbf{e} - 4$$

Perhaps the most natural representation of derivations for a context-free grammar is *a derivation tree* or *a parse tree*. Each *internal node* of such a tree corresponds to a nonterminal and each *leaf* corresponds to a terminal. If A is an internal node with children $B_1, \ldots, B_n$ ordered from left to right, then $A \to B_1 \cdots B_n$ must be a production. The concatenation of all leaves from left to right yields the string being derived. For example, the derivation tree corresponding to the preceding derivation of 12.3**e** − 4 is given in Figure 6.3. Such a tree also makes possible the extraction of the parts 12, 3, and −4, which are useful in the storage of the real value in a computer memory.

Definition 6.18 A context-free grammar G is **ambiguous** if there is a string $x \in L(G)$, which has two distinct derivation trees. Otherwise G is *unambiguous*.

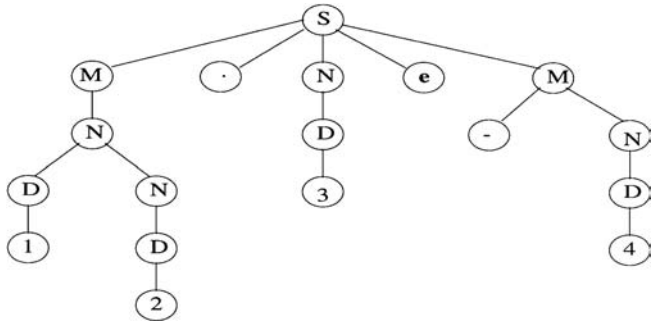


FIGURE 6.3 The derivation tree for $12.3\mathbf{e} - 4$.

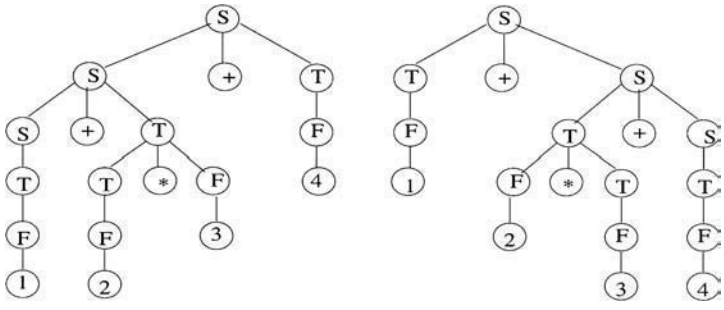


FIGURE 6.4 Different derivation trees for the expression $1 + 2 * 3 + 4$.

Unambiguity is a very desirable property to have as it allows a unique interpretation of each sentence in the language. It is not hard to see that the preceding grammar for Pascal real values and the grammar G_1 defined in Example 6.7 are all unambiguous. The following example shows an ambiguous grammar.

Example 6.13

Consider a grammar G_7 for all valid arithmetic expressions that are composed of unsigned positive integers and symbols $+, *, (,)$. For convenience, let us use the symbol n to denote any unsigned positive integer. This grammar has the productions

$$S \rightarrow T + S \mid S + T \mid T$$
$$T \rightarrow F * T \mid T * F \mid F$$
$$F \rightarrow n \mid (S)$$

Two possible different derivation trees for the expression $1 + 2 * 3 + 4$ are shown in Figure 6.4. Thus, G_7 is ambiguous. The left tree means that the first addition should be done before the second addition and the right tree says the opposite.

Although in the preceding example different derivations/interpretations of any expression always result in the same value because the operations addition and multiplication are associative, there are situations where the difference in the derivation can affect the final outcome. Actually, the grammar G_7 can be made unambiguous by removing some (redundant) productions, for example, $S \rightarrow T + S$ and $T \rightarrow F * T$. This corresponds to the convention that a sequence of consecutive additions (or multiplications) is always evaluated from left to right and will not change the language generated by G_7. It is worth noting that

there are context-free languages which cannot be generated by any unambiguous context-free grammar [Hopcroft and Ullman 1979]. Such languages are said to be *inherently ambiguous*. An example of inherently ambiguous languages is the set

$$\{0^m 1^m 2^n 3^n \mid m, n > 0\} \cup \{0^m 1^n 2^m 3^n \mid m, n > 0\}$$

We end this section by presenting an efficient algorithm for the membership problem for context-free grammars. The algorithm is due to Cocke, Younger, and Kasami [Hopcroft and Ullman 1979] and is often called the CYK algorithm. Let $G = (\Sigma, N, S, P)$ be a context-free grammar. For simplicity, let us assume that G does not generate the empty string ϵ and that G is in the so-called **Chomsky normal form** [Chomsky 1963], that is, every production of G is either in the form $A \to BC$ where B and C are nonterminals, or in the form $A \to a$ where a is a terminal. An example of such a grammar is G_1 given in Example 6.7. This is not a restrictive assumption because there is a simple algorithm which can convert every context-free grammar that does not generate ϵ into one in the Chomsky normal form.

Suppose that $x = a_1 \cdots a_n$ is a string of n terminals. The basic idea of the CYK algorithm, which decides if $x \in L(G)$, is *dynamic programming*. For each pair i, j, where $1 \le i \le j \le n$, define a set $X_{i,j} \subseteq N$ as

$$X_{i,j} = \{A \mid A \Rightarrow^* a_i \cdots a_j\}$$

Thus, $x \in L(G)$ if and only if $S \in X_{1,n}$. The sets $X_{i,j}$ can be computed inductively in the ascending order of $j - i$. It is easy to figure out $X_{i,i}$ for each i because $X_{i,i} = \{A \mid A \to a_i \in P\}$. Suppose that we have computed all $X_{i,j}$ where $j - i < d$ for some $d > 0$. To compute a set $X_{i,j}$, where $j - i = d$, we just have to find all of the nonterminals A such that there exist some nonterminals B and C satisfying $A \to BC \in P$ and for some k, $i \le k < j$, $B \in X_{i,k}$, and $C \in X_{k+1,j}$. A rigorous description of the algorithm in a Pascal style pseudocode is given as follows.

Algorithm CYK($x = a_1 \cdots a_n$):

1. for $i \leftarrow 1$ to n do
2. $\quad X_{i,i} \leftarrow \{A \mid A \to a_i \in P\}$
3. for $d \leftarrow 1$ to $n - 1$ do
4. $\quad$ for $i \leftarrow 1$ to $n - d$ do
5. $\quad X_{i,i+d} \leftarrow \emptyset$
6. $\quad$ for $t \leftarrow 0$ to $d - 1$ do
7. $\quad\quad X_{i,i+d} \leftarrow X_{i,i+d} \cup \{A \mid A \to BC \in P$ for some $B \in X_{i,i+t}$ and $C \in X_{i+t+1,i+d}\}$

Table 6.2 shows the sets $X_{i,j}$ for the grammar G_1 and the string $x = 000111$. It just so happens that every $X_{i,j}$ is either empty or a singleton. The computation proceeds from the main diagonal toward the upper-right corner.

TABLE 6.2 An Example Execution of the CYK Algorithm

		0	0	0	1	1	1
		$j \rightarrow$					
		1	2	3	4	5	6
$i \downarrow$	1	O					S
	2		O			S	T
	3			O	S	T	
	4				I		
	5					I	
	6						I

6.5 Computational Models

In this section, we will present many restricted versions of Turing machines and address the question of what kinds of problems they can solve. Such a classification is a central goal of computation theory. We have already classified problems broadly into (totally) decidable, partially decidable, and totally undecidable. Because the decidable problems are the ones of most practical interest, we can consider further classification of decidable problems by placing two types of restrictions on a Turing machine. The first one is to restrict its structure. This way we obtain many machines of which a finite automaton and a pushdown automaton are the most important. The other way to restrict a Turing machine is to bound the amount of resources it uses, such as the number of time steps or the number of tape cells it can use. The resulting machines form the basis for *complexity theory*.

6.5.1 Finite Automata

The finite automaton (in its deterministic version) was first introduced by McCulloch and Pitts [1943] as a logical model for the behavior of neural systems. Rabin and Scott [1959] introduced the nondeterministic version of the finite automaton and showed the equivalence of the nondeterministic and deterministic versions. Chomsky and Miller [1958] proved that the set of languages that can be recognized by a finite automaton is precisely the regular languages introduced in Section 6.4. Kleene [1956] showed that the languages accepted by finite automata are characterized by regular expressions as defined in Section 6.4.

In addition to their original role in the study of neural nets, finite automata have enjoyed great success in many fields such as sequential circuit analysis in circuit design [Kohavi 1978], asynchronous circuits [Brzozowski and Seger 1994], lexical analysis in text processing [Lesk 1975], and compiler design. They also led to the design of more efficient algorithms. One excellent example is the development of linear-time string-matching algorithms, as described in Knuth et al. [1977]. Other applications of finite automata can be found in computational biology [Searls 1993], natural language processing, and distributed computing.

A finite automaton, as in Figure 6.5, consists of an input tape which contains a (finite) sequence of input symbols such as $aabab\cdots$, as shown in the figure, and a finite-state control. The tape is read by the one-way *read-only* input head from left to right, one symbol at a time. Each time the input head reads an input symbol, the finite control changes its state according to the symbol and the current state of the machine. When the input head reaches the right end of the input tape, if the machine is in a final state, we say that the input is accepted; if the machine is not in a final state, we say that the input is rejected. The following is the formal definition.

Definition 6.19 A *nondeterministic finite automaton* (NFA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of *states*.
- Σ is a finite set of *input symbols*.
- δ, the *state transition function*, is a mapping from $Q \times \Sigma$ to subsets of Q.
- $q_0 \in Q$ is the *initial state* of the NFA.
- $F \subseteq Q$ is the set of *final states*.

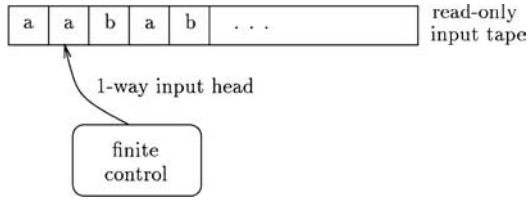


FIGURE 6.5 A finite automaton.

If δ maps $|Q| \times \Sigma$ to singleton subsets of Q, then we call such a machine a *deterministic finite automaton* (DFA).

When an automaton, M, is nondeterministic, then from the current state and input symbol, it may go to one of several different states. One may imagine that the device goes to all such states in parallel. The DFA is just a special case of the NFA; it always follows a single deterministic path. The device *M accepts* an input string x if, starting with q_0 and the read head at the first symbol of x, one of these parallel paths reaches an accepting state when the read head reaches the end of x. Otherwise, we say *M rejects* x. A language, L, is accepted by M if M accepts all of the strings in L and nothing else, and we write $L = L(M)$. We will also allow the machine to make ϵ-*transitions*, that is, changing state without advancing the read head. This allows transition functions such as $\delta(s, \epsilon) = \{s'\}$. It is easy to show that such a generalization does not add more power.

Remark 6.2 The concept of a nondeterministic automaton is rather confusing for a beginner. But there is a simple way to relate it to a concept which must be familiar to all of the readers. It is that of a solitaire game. Imagine a game like *Klondike*. The game starts with a certain arrangement of cards (the input) and there is a well-defined final position that results in success; there are also dead ends where a further move is not possible; you lose if you reach any of them. At each step, the precise rules of the game dictate how a new arrangement of cards can be reached from the current one. But the most important point is that there are many possible moves at each step. (Otherwise, the game would be no fun!) Now consider the following question: What starting positions are *winnable*? These are the starting positions for which *there is a winning move sequence*; of course, in a typical play a player may not achieve it. But that is beside the point in the definition of what starting positions are winnable. The connection between such games and a nondeterministic automaton should be clear. The multiple choices at each step are what make it *nondeterministic*. Our definition of winnable positions is similar to the concept of acceptance of a string by a nondeterministic automaton. Thus, an NFA may be viewed as a formal model to define solitaire games.

Example 6.14

We design a DFA to accept the language represented by the regular expression $0(0+1)^*1$ as in Example 6.2, that is, the set of all strings in $\{0, 1\}$ which begin with a 0 and end with a 1. It is usually convenient to draw our solution as in Figure 6.6. As a convention, each circle represents a state; the state a, pointed at by the initial arrow, is the initial state. The darker circle represents the final states (state c). The transition function δ is represented by the labeled edges. For example, $\delta(a, 0) = \{b\}$. When a transition is missing, for example on input 1 from a and on inputs 0 and 1 from c, it is assumed that all of these lead to an implicit nonaccepting trap state, which has transitions to itself on all inputs.

The machine in Figure 6.6 is nondeterministic because from b on input 1 the machine has two choices: stay at b or go to c.

Figure 6.7 gives an equivalent DFA, accepting the same language.

Example 6.15

The DFA in Figure 6.8 accepts the set of all strings in $\{0, 1\}^*$ with an even number of 1s. The corresponding regular expression is $(0^*10^*1)^*0^*$.

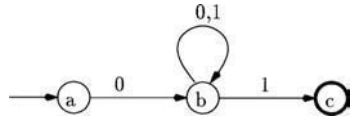


FIGURE 6.6 An NFA accepting $0(0+1)^*1$.

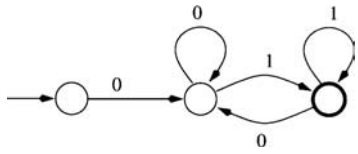


FIGURE 6.7 A DFA accepting $0(0+1)^*1$.

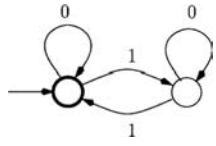


FIGURE 6.8 A DFA accepting $(0^*10^*1)^*0^*$.

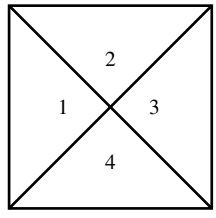


FIGURE 6.9 Numbering the quarters of a tile.

Example 6.16

As a final example, consider the special case of the tiling problem that we discussed in Section 6.2. This version of the problem is as follows: Let k be a fixed positive integer. Given a set of unit tiles, we want to know if they can tile any $k \times n$ area for all n. We show how to deal with the case $k = 1$ and leave it as an exercise to generalize our method for larger values of k. Number the quarters of each tile as in Figure 6.9. The given set of tiles will tile the area if we can find a sequence of the given tiles $T_1, T_2, \ldots, T_m$ such that (1) the third quarter of T_1 has the same color as the first quarter of T_2, and the third quarter of T_2 has the same color as the first quarter of T_3, etc., and (2) the third quarter of T_m has the same color as T_1. These conditions can be easily understood as follows. The first condition states that the tiles T_1, T_2, etc., can be placed adjacent to each other along a row in that order. The second condition implies that the whole sequence $T_1\ T_2 \cdots T_m$ can be replicated any number of times. And a little thought reveals that this is all we need to answer yes on the input. But if we cannot find such a sequence, then the answer must be no. Also note that in the sequence no tile needs to be repeated and so the value of m is bounded by the number of tiles in the input. Thus, we have reduced the problem to searching a finite number of possibilities and we are done.

How is the preceding discussion related to finite automata? To see the connection, define an alphabet consisting of the unit tiles and define a language $L = \{T_1 T_2 \cdots T_m \mid T_1 T_2 \cdots T_m$ is a valid tiling, $m \geq 0\}$. We will now construct an NFA for the language L. It consists of states corresponding to *distinct* colors contained in the tiles plus two states, one of them the start state and another state called the dead state. The NFA makes transitions as follows: From the start state there is an ϵ-transition to each color state, and all states except the dead state are accepting states. When in the state corresponding to color i, suppose it receives input tile T. If the first quarter of this tile has color i, then it moves to the color of the third quarter of T; otherwise, it enters the dead state. The basic idea is to remember the only relevant piece

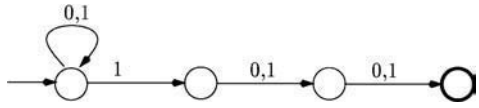


FIGURE 6.10 An NFA accepting L_3.

of information after processing some input. In this case, it is the third quarter color of the last tile seen. Having constructed this NFA, the question we are asking is if the language accepted by this NFA is infinite. There is a simple algorithm for this problem [Hopcroft and Ullman 1979].

The next three theorems show a satisfying result that all the following language classes are identical:

- The class of languages accepted by DFAs
- The class of languages accepted by NFAs
- The class of languages generated by regular expressions, as in Definition 6.8
- The class of languages generated by the right-linear, or type-3, grammars, as in Definition 6.16

Recall that this class of languages is called the *regular languages* (see Section 6.4).

Theorem 6.6 *For each NFA, there is an equivalent DFA.*

Proof An NFA might look more powerful because it can carry out its computation in parallel with its nondeterministic branches. But because we are working with a *finite number* of states, we can simulate an NFA $M = (Q, \Sigma, \delta, q_0, F)$ by a DFA $M' = (Q', \Sigma, \delta', q_0', F')$, where

- $Q' = \{[S] : S \subseteq Q\}$.
- $q_0' = [\{q_0\}]$.
- $\delta'([S], a) = [S'] = [\cup_{q_l \in S} \delta(q_l, a)]$.
- F' is the set of all subsets of Q containing a state in F.

It can now be verified that $L(M) = L(M')$. □

Example 6.17

Example 6.1 contains an NFA and an equivalent DFA accepting the same language. In fact, the proof provides an effective procedure for converting an NFA to a DFA. Although each NFA can be converted to an equivalent DFA, the resulting DFA might be exponentially large in terms of the number of states, as we can see from the previous procedure. This turns out to be the best thing one can do in the worst case. Consider the language: $L_k = \{x : x \in \{0, 1\}^*$ and the kth letter from the right of x is a 1$\}$. An NFA of $k + 1$ states (for $k = 3$) accepting L_k is given in Figure 6.10. A counting argument shows that any DFA accepting L_k must have at least 2^k states.

Theorem 6.7 *L is generated by a right-linear grammar if it is accepted by an* NFA.

Proof Let L be accepted by a right-linear grammar $G = (\Sigma, N, S, P)$. We design an NFA $M = (Q, \Sigma, \delta, q_0, F)$ where $Q = N \cup \{f\}$, $q_0 = S$, $F = \{f\}$. To define the δ function, we have $C \in \delta(A, b)$ if $A \to bC$. For rules $A \to b$, $\delta(A, b) = \{f\}$. Obviously, $L(M) = L(G)$.

Conversely, if L is accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$, we define an equivalent right-linear grammar $G = (\Sigma, N, S, P)$, where $N = Q$, $S = q_0$, $q_i \to aq_j \in N$ if $q_j \in \delta(q_i, a)$, and $q_j \to \epsilon$ if $q_j \in F$. Again it is easily seen that $L(M) = L(G)$. □

Theorem 6.8 *L is generated by a regular expression if it is accepted by an* NFA.

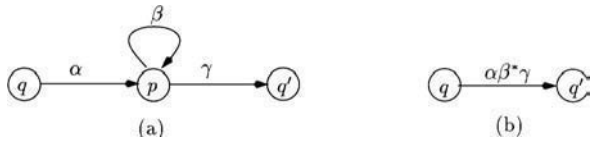


FIGURE 6.11 Converting an NFA to a regular expression.

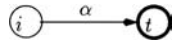


FIGURE 6.12 The reduced NFA.

Proof (Idea) **Part 1.** We inductively convert a regular expression to an NFA which accepts the language generated by the regular expression as follows.

- Regular expression ϵ converts to $(\{q\}, \Sigma, \emptyset, q, \{q\})$.
- Regular expression $\emptyset$ converts to $(\{q\}, \Sigma, \emptyset, q, \emptyset)$.
- Regular expression a, for each $a \in \Sigma$ converts to $(\{q, f\}, \Sigma, \delta(q, a) = \{f\}, q, \{f\})$.
- If α and β are regular expressions, converting to NFAs M_α and M_β, respectively, then the regular expression $\alpha \cup \beta$ converts to an NFA M, which connects M_α and M_β in parallel: M has an initial state q_0 and all of the states and transitions of M_α and M_β; by ϵ-transitions, M goes from q_0 to the initial states of M_α and M_β.
- If α and β are regular expressions, converting to NFAs M_α and M_β, respectively, then the regular expression $\alpha\beta$ converts to NFA M, which connects M_α and M_β sequentially: M has all of the states and transitions of M_α and M_β, with M_α's initial state as M's initial state, ϵ-transition from the final states of M_α to the initial state of M_β, and M_β's final states as M's final states.
- If α is a regular expression, converting to NFA M_α, then connecting all of the final states of M_α to its initial state with ϵ-transitions gives α^+. Union of this with the NFA for ϵ gives the NFA for α^*.

Part 2. We now show how to convert an NFA to an equivalent regular expression. The idea used here is based on Brzozowski and McCluskey [1963]; see also Brzozowski and Seger [1994] and Wood [1987].

Given an NFA M, expand it to M' by adding two extra states i, the initial state of M', and t, the only final state of M', with ϵ transitions from i to the initial state of M and from all final states of M to t. Clearly, $L(M) = L(M')$. In M', remove states other than i and t one by one as follows. To remove state p, for each triple of states q, p, q' as shown in Figure 6.11a, add the transition as shown in Figure 6.11(b). □

If p does not have a transition leading back to itself, then $\beta = \epsilon$. After we have considered all such triples, delete state p and transitions related to p. Finally, we obtain Figure 6.12 and $L(\alpha) = L(M)$.

Apparently, DFAs cannot serve as our model for a modern computer. Many extremely simple languages cannot be accepted by DFAs. For example, $L = \{xx : x \in \{0, 1\}^*\}$ cannot be accepted by a DFA. One can prove this by counting, or using the so-called pumping lemmas; one can also prove this by arguing that x contains more information than a *finite* state machine can *remember*. We refer the interested readers to textbooks such as Hopcroft and Ullmann [1979], Gurari [1989], Wood [1987], and Floyd and Beigel [1994] for traditional approaches and to Li and Vitányi [1993] for a nontraditional approach. One can try to generalize the DFA to allow the input head to be *two way* but still read only. But such machines are not more powerful, they can be simulated by normal DFAs. The next step is apparently to add *storage* space such that our machines can *write* information in.

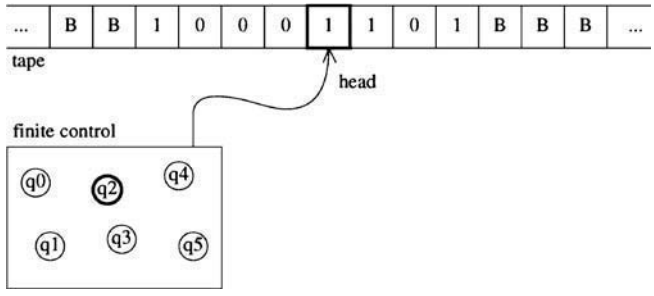


FIGURE 6.13 A Turing machine.

6.5.2 Turing Machines

In this section we will provide an alternative definition of a Turing machine to make it compatible with our definitions of a DFA, PDA, etc. This also makes it easier to define a nondeterministic Turing machine. But this formulation (at least the deterministic version) is essentially the same as the one presented in Section 6.2.

A Turing machine (TM), as in Figure 6.13, consists of a *finite control*, an infinite *tape* divided into cells, and a read/write *head* on the tape. We refer to the two directions on the tape as *left* and *right*. The finite control can be in any one of a finite set Q of states, and each tape cell can contain a 0, a 1, or a *blank* B. Time is discrete and the time instants are ordered $0, 1, 2, \ldots$ with 0 the time at which the machine starts its computation. At any time, the head is positioned over a particular cell, which it is said to *scan*. At time 0 the head is situated on a distinguished cell on the tape called the *start cell*, and the finite control is in the initial state q_0. At time 0 all cells contain Bs, except a contiguous finite sequence of cells, extending from the start cell to the right, which contain 0s and 1s. This binary sequence is called the *input*.

The device can perform the following basic operations:

1. It can write an element from the tape alphabet $\Sigma = \{0, 1, B\}$ in the cell it scans.
2. It can shift the head one cell left or right.

Also, the device executes these operations at the rate of one operation per time unit (a *step*). At the conclusion of each step, the finite control takes on a state in Q. The device operates according to a finite set P of *rules*.

The rules have format (p, s, a, q) with the meaning that if the device is in state p and s is the symbol under scan then write a if $a \in \{0, 1, B\}$ or move the head according to a if $a \in \{L, R\}$ and the finite control changes to state q. At some point, if the device gets into a special *final* state q_f, the device stops and accepts the input.

If every pair of distinct quadruples differs in the first two elements, then the device is *deterministic*. Otherwise, the device is *nondeterministic*. Not every possible combination of the first two elements has to be in the set; in this way we permit the device to perform *no* operation. In this case, we say the device *halts*. In this case, if the machine is not in a final state, we say that the machine *rejects* the input.

Definition 6.20 A Turing machine is a quintuple $M = (Q, \Sigma, P, q_0, q_f)$ where each of the components has been described previously.

Given an input, a deterministic Turing machine carries out a uniquely determined succession of operations, which may or may not terminate in a finite number of steps. If it terminates, then the nonblank symbols left on the tape are the output. Given an input, a **nondeterministic Turing machine** behaves much like an NFA. One may imagine that it carries out its computation in parallel. Such a computation may be viewed as a (possibly infinite) tree. The root of the tree is the starting configuration of the machine.

The children of each node are all possible configurations one step away from this node. If any of the branches terminates in the final state q_f, we say the machine accepts the input. The reader may want to test understanding this new formulation of a Turing machine by redoing the doubling program on a Turing machine with states and transitions (rather than a GOTO program).

A Turing machine *accepts* a language L if $L = \{w : M \text{ accepts } w\}$. Furthermore, if M halts on all inputs, then we say that L is *Turing decidable*, or *recursive*. The connection between a recursive language and a decidable problem (function) should be clear. It is that function f is decidable if and only if L_f is recursive. (Readers who may have forgotten the connection between function f and the associated language L_f should review Remark 6.1.)

Theorem 6.9 *All of the following generalizations of Turing machines can be simulated by a one-tape deterministic Turing machine defined in Definition 6.20.*

- *Larger tape alphabet* Σ
- *More work tapes*
- *More access points, read/write heads, on each tape*
- *Two- or more dimensional tapes*
- *Nondeterminism*

Although these generalizations do not make a Turing machine compute more, they do make a Turing machine more efficient and easier to program. Many more variants of Turing machines are studied and used in the literature. Of all simulations in Theorem 6.9, the last one needs some comments. A nondeterministic computation branches like a tree. When simulating such a computation for n steps, the obvious thing for a deterministic Turing machine to do is to try all possibilities; thus, this requires up to c^n steps, where c is the maximum number of nondeterministic choices at each step.

Example 6.18

A DFA is an extremely simple Turing machine. It just reads the input symbols from left to right. Turing machines naturally accept more languages than DFAs can. For example, a Turing machine can accept $L = \{xx : x \in \{0, 1\}^*\}$ as follows:

- Find the middle point first: it is trivial by using two heads; with one head, one can mark one symbol at the left and then mark another on the right, and go back and forth to eventually find the middle point.
- Match the two parts: with two heads, this is again trivial; with one head, one can again use the marking method matching a pair of symbols each round; if the two parts match, accept the input by entering q_f.

There are types of storage media other than a tape:

- A *pushdown store* is a semi-infinite work tape with one head such that each time the head moves to the left, it erases the symbol scanned previously; this is a last-in first-out storage.
- A *queue* is a semi-infinite work tape with two heads that move only to the right, the leading head is write-only and the trailing head is read-only; this is a first-in first-out storage.
- A *counter* is a pushdown store with a single-letter alphabet (except its one end, which holds a special marker symbol). Thus, a counter can store a nonnegative integer and can perform three operations.

A queue machine can simulate a normal Turing machine, but the other two types of machines are not powerful enough to simulate a Turing machine.

Example 6.19

When the Turing machine tape is replaced by a pushdown store, the machine is called a *pushdown automaton*. Pushdown automata have been thoroughly studied because they accept the class of context-free

languages defined in Section 6.4. More precisely, it can be shown that if L is a context-free language, then it is accepted by a PDA, and if L is accepted by a PDA, then there is a CFG generating L. Various types of PDAs have fundamental applications in compiler design.

The PDA is more restricted than a Turing machine. For example, $L = \{xx : x \in \{0,1\}^*\}$ cannot be accepted by a PDA, but it can be accepted by a Turing machine as in Example 6.18. But a PDA is more powerful than a DFA. For example, a PDA can accept the language $L' = \{0^k 1^k : k \geq 0\}$ easily. It can read the 0s and push them into the pushdown store; then, after it finishes the 0s, each time the PDA reads a 1, it removes a 0 from the pushdown store; at the end, it accepts if the pushdown store is empty (the number of 0s matches that of 1s). But a DFA cannot accept L', because after it has read all of the 0s, it cannot remember k when k has higher information content than the DFA's finite control.

Two pushdown stores can be used to simulate a tape easily. For comparisons of powers of pushdown stores, queues, counters, and tapes, see van Emde Boas [1990] and Li and Vitányi [1993].

The idea of the universal algorithm was introduced in Section 6.2. Formally, a *universal Turing machine*, U, takes an encoding of a pair of parameters (M, x) as input and simulates M on input x. U accepts (M, x) iff M accepts x. The universal Turing machines have many applications. For example, the definition of Kolmogorov complexity [Li and Vitányi 1993] fundamentally relies on them.

Example 6.20

Let $L_u = \{\langle M, w\rangle : M \text{ accepts } w\}$. Then L_u can be accepted by a Turing machine, but it is not Turing decidable. The proof is omitted.

If a language is Turing acceptable but not Turing decidable, we call such a language *recursively enumerable* (r.e.). Thus, L_u is r.e. but not recursive. It is easily seen that if both a language and its complement are r.e., then both of them are recursive. Thus, $\bar{L}_u$ is not r.e.

6.5.2.1 Time and Space Complexity

With Turing machines, we can now formally define what we mean by **time and space complexities**. Such a formal investigation by Hartmanis and Stearns [1965] marked the beginning of the field of *computational complexity*. We refer the readers to Hartmanis' Turing Award lecture [Hartmanis 1994] for an interesting account of the history and the future of this field.

To define the space complexity properly (in the sublinear case), we need to slightly modify the Turing machine of Figure 6.13. We will replace the tape containing the input by a read-only input tape and give the Turing machine some extra work tapes.

Definition 6.21 Let M be a Turing machine. If for each n, for each input of length n, and for each sequence of choices of moves when M is nondeterministic, M makes at most $T(n)$ moves we say that M is of *time complexity* $T(n)$; similarly, if M uses at most $S(n)$ tape cells of the work tape, we say that M is of space complexity $S(n)$.

Theorem 6.10 *Any Turing machine using $s(n)$ space can be simulated by a Turing machine, with just one work tape, using $s(n)$ space. If a language is accepted by a k-tape Turing machine running in time $t(n)$ [space $s(n)$], then it also can be accepted by another k-tape Turing machine running in time $ct(n)$ [space $cs(n)$], for any constant $c > 0$.*

To avoid writing the constant c everywhere, we use the standard big-O notation: we say $f(n)$ is $O(g(n))$ if there is a constant c such that $f(n) \leq cg(n)$ for all but finitely many n. The preceding theorem is called the linear speedup theorem; it can be proved easily by using a larger tape alphabet to encode several cells into one and hence compress several steps into one. It leads to the following definitions.

Definition 6.22

DTIME$[t(n)]$ is the set of languages accepted by multitape deterministic TMs in time $O(t(n))$.
NTIME$[t(n)]$ is the set of languages accepted by multitape nondeterministic TMs in time $O(t(n))$.

DSPACE$[s(n)]$ is the set of languages accepted by multitape deterministic TMs in space $O(s(n))$.
NSPACE$[s(n)]$ is the set of languages accepted by multitape nondeterministic TMs in space $O(s(n))$.
P is the complexity class $\bigcup_{c\in\mathcal{N}}$ DTIME$[n^c]$.
NP is the complexity class $\bigcup_{c\in\mathcal{N}}$ NTIME$[n^c]$.
PSPACE is the complexity class $\bigcup_{c\in\mathcal{N}}$ DSPACE$[n^c]$.

Example 6.21

We mentioned in Example 6.18 that $L = \{xx : x \in \{0,1\}^*\}$ can be accepted by a Turing machine. The procedure we have presented in Example 6.18 for a one-head one-tape Turing machine takes $O(n^2)$ time because the single head must go back and forth marking and matching. With two heads, or two tapes, L can be easily accepted in $O(n)$ time.

It should be clear that any language that can be accepted by a DFA, an NFA, or a PDA can be accepted by a Turing machine in $O(n)$ time. The type-1 grammar in Definition 6.16 can be accepted by a Turing machine in $O(n)$ space. Languages in P, that is, languages acceptable by Turing machines in *polynomial* time, are considered as *feasibly* computable. It is important to point out that all generalizations of the Turing machine, except the nondeterministic version, can all be simulated by the basic one-tape deterministic Turing machine with at most polynomial slowdown. The class NP represents the class of languages accepted in polynomial time by a nondeterministic Turing machine. The nondeterministic version of PSPACE turns out to be identical to PSPACE [Savitch 1970]. The following relationships are true:

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE}$$

Whether or not either of the inclusions is proper is one of the most fundamental open questions in computer science and mathematics. Research in computational complexity theory centers around these questions. To solve these problems, one can identify the hardest problems in NP or PSPACE. These topics will be discussed in Chapter 8. We refer the interested reader to Gurari [1989], Hopcroft and Ullman [1979], Wood [1987], and Floyd and Beigel [1994].

6.5.2.2 Other Computing Models

Over the years, many alternative computing models have been proposed. With reasonable complexity measures, they can all be simulated by Turing machines with at most a polynomial slowdown. The reference van Emde Boas [1990] provides a nice survey of various computing models other than Turing machines. Because of limited space, we will discuss a few such alternatives very briefly and refer our readers to van Emde Boas [1990] for details and references.

Random Access Machines. The *random access machine* (RAM) [Cook and Reckhow 1973] consists of a finite control where a program is stored, with several arithmetic registers and an infinite collection of memory registers $R[1], R[2], \ldots$. All registers have an unbounded word length. The basic instructions for the program are LOAD, ADD, MULT, STORE, GOTO, ACCEPT, REJECT, etc. Indirect addressing is also used. Apparently, compared to Turing machines, this is a closer but more complicated approximation of modern computers. There are two standard ways for measuring time complexity of the model:

- The *unit-cost RAM:* in this case, each instruction takes one unit of time, no matter how big the operands are. This measure is convenient for analyzing some algorithms such as sorting. But it is unrealistic or even meaningless for analyzing some other algorithms, such as integer multiplication.
- The *log-cost RAM:* each instruction is charged for the sum of the lengths of all data manipulated implicitly or explicitly by the instruction. This is a more realistic model but sometimes less convenient to use.

Log-cost RAMs and Turing machines can simulate each other with polynomial overheads. The unit-cost RAM might be exponentially (but unrealistically) faster when, for example, it uses its power of multiplying two large numbers in one step.

Pointer Machines. The pointer machines were introduced by Kolmogorov and Uspenskii [1958] (also known as the Kolmogorov–Uspenskii machine) and by Schönhage in 1980 (also known as the storage modification machine, see Schönhage [1980]). We informally describe the pointer machine here. A pointer machine is similar to a RAM but differs in its memory structure. A pointer machine operates on a storage structure called a Δ structure, where Δ is a finite alphabet of size greater than one. A Δ-structure S is a finite directed graph (the Kolmogorov–Uspenskii version is an undirected graph) in which each node has $k = |\Delta|$ outgoing edges, which are labeled by the k symbols in Δ. S has a distinguished node called the *center*, which acts as a starting point for addressing, with words over Δ, other nodes in the structure. The pointer machine has various instructions to redirect the pointers or edges and thus modify the storage structure. It should be clear that Turing machines and pointer machines can simulate each other with at most polynomial delay if we use the log-cost model as with the RAMs. There are many interesting studies on the efficiency of the preceding simulations. We refer the reader to van Emde Boas [1990] for more pointers on the pointer machines.

Circuits and Nonuniform Models. A *Boolean circuit* is a finite, labeled, directed acyclic graph. Input nodes are nodes without ancestors; they are labeled with input variables $x_1, \ldots, x_n$. The internal nodes are labeled with functions from a finite set of Boolean operations, for example, {and, or, not} or $\{\oplus\}$. The number of ancestors of an internal node is precisely the number of arguments of the Boolean function that the node is labeled with. A node without successors is an output node. The circuit is naturally evaluated from input to output: at each node the function labeling the node is evaluated using the results of its ancestors as arguments. Two cost measures for the circuit model are:

- *Depth:* the length of a longest path from an input node to an output node
- *Size:* the number of nodes in the circuit

These measures are applied to a family of circuits $\{C_n : n \geq 1\}$ for a particular problem, where C_n solves the problem of size n. If C_n can be computed from n (in polynomial time), then this is a *uniform measure*. Such circuit families are equivalent to Turing machines. If C_n cannot be computed from n, then such measures are *nonuniform* measures, and such classes of circuits are more powerful than Turing machines because they simply can compute any function by encoding the solutions of all inputs for each n. See van Emde Boas [1990] for more details and pointers to the literature.

Acknowledgment

We would like to thank John Tromp and the reviewers for reading the initial drafts and helping us to improve the presentation.

Defining Terms

Algorithm A finite sequence of instructions that is supposed to solve a particular problem.

Ambiguous context-free grammar For some string of terminals the grammar has two distinct derivation trees.

Chomsky normal form: Every rule of the context-free grammar has the form $A \to BC$ or $A \to a$, where A, B, and C are nonterminals and a is a terminal.

Computable or decidable function/problem: A function/problem that can be solved by an algorithm (or equivalently, a Turing machine).

Context-free grammar: A grammar whose rules have the form $A \to \beta$, where A is a nonterminal and β is a string of nonterminals and terminals.

Context-free language: A language that can be described by some context-free grammar.

Context-sensitive grammar: A grammar whose rules have the form $\alpha \to \beta$, where α and β are strings of nonterminals and terminals and $|\alpha| \leq |\beta|$.

Context-sensitive language: A language that can be described by some context-sensitive grammar.

Derivation or parsing: An illustration of how a string of terminals is obtained from the start symbol by successively applying the rules of the grammar.

Finite automaton or finite-state machine: A restricted Turing machine where the head is read only and shifts only from left to right.

(Formal) grammar: A description of some language typically consisting of a set of terminals, a set of nonterminals with a distinguished one called the start symbol, and a set of rules (or productions) of the form $\alpha \rightarrow \beta$, depicting what string α of terminals and nonterminals can be rewritten as another string β of terminals and nonterminals.

(Formal) language: A set of strings over some fixed alphabet.

Halting problem: The problem of deciding if a given program (or Turing machine) halts on a given input.

Nondeterministic Turing machine: A Turing machine that can make any one of a prescribed set of moves on a given state and symbol read on the tape.

Partially decidable decision problem: There exists a program that always halts and outputs 1 for every input expecting a positive answer and either halts and outputs 0 or loops forever for every input expecting a negative answer.

Program: A sequence of instructions that is not required to terminate on every input.

Pushdown automaton: A restricted Turing machine where the tape acts as a pushdown store (or a stack).

Reduction: A computable transformation of one problem into another.

Regular expression: A description of some language using operators union, concatenation, and Kleene closure.

Regular language: A language that can be described by some right-linear/regular grammar (or equivalently by some regular expression).

Right-linear or regular grammar: A grammar whose rules have the form $A \rightarrow aB$ or $A \rightarrow a$, where A, B are nonterminals and a is either a terminal or the null string.

Time/space complexity: A function describing the maximum time/space required by the machine on any input of length n.

Turing machine: A simplest formal model of computation consisting of a finite-state control and a semi-infinite sequential tape with a read–write head. Depending on the current state and symbol read on the tape, the machine can change its state and move the head to the left or right.

Uncomputable or undecidable function/problem: A function/problem that cannot be solved by any algorithm (or equivalently, any Turing machine).

Universal algorithm: An algorithm that is capable of simulating any other algorithms if properly encoded.

References

Angluin, D. 1980. Finding patterns common to a set of strings. *J. Comput. Syst. Sci.* 21:46–62.

Brzozowski, J. and McCluskey, E., Jr. 1963. Signal flow graph techniques for sequential circuit state diagram. *IEEE Trans. Electron. Comput.* EC-12(2):67–76.

Brzozowski, J. A. and Seger, C.-J. H. 1994. *Asynchronous Circuits*. Springer–Verlag, New York.

Chomsky, N. 1956. Three models for the description of language. *IRE Trans. Inf. Theory* 2(2):113–124.

Chomsky, N. 1963. Formal properties of grammars. In *Handbook of Mathematical Psychology,* Vol. 2, pp. 323–418. John Wiley and Sons, New York.

Chomsky, N. and Miller, G. 1958. Finite-state languages. *Information and Control* 1:91–112.

Cook, S. and Reckhow, R. 1973. Time bounded random access machines. *J. Comput. Syst. Sci.* 7:354–375.

Davis, M. 1980. What is computation? In *Mathematics Today–Twelve Informal Essays*. L. Steen, ed., pp. 241–259. Vintage Books, New York.

Floyd, R. W. and Beigel, R. 1994. *The Language of Machines: An Introduction to Computability and Formal Languages*. Computer Science Press, New York.

Gurari, E. 1989. *An Introduction to the Theory of Computation*. Computer Science Press, Rockville, MD.

Harel, D. 1992. *Algorithmics: The Spirit of Computing*. Addison–Wesley, Reading, MA.

Harrison, M. 1978. *Introduction to Formal Language Theory*. Addison–Wesley, Reading, MA.

Hartmanis, J. 1994. On computational complexity and the nature of computer science. *Commun. ACM* 37(10):37–43.

Hartmanis, J. and Stearns, R. 1965. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.* 117:285–306.
Hopcroft, J. and Ullman, J. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison–Wesley, Reading, MA.
Jiang, T., Salomaa, A., Salomaa, K., and Yu, S. 1995. Decision problems for patterns. *J. Comput. Syst. Sci.* 50(1):53–63.
Kleene, S. 1956. Representation of events in nerve nets and finite automata. In *Automata Studies*, pp. 3–41. Princeton University Press, Princeton, NJ.
Knuth, D., Morris, J., and Pratt, V. 1977. Fast pattern matching in strings. *SIAM J. Comput.* 6:323–350.
Kohavi, Z. 1978. *Switching and Finite Automata Theory*. McGraw–Hill, New York.
Kolmogorov, A. and Uspenskii, V. 1958. On the definition of an algorithm. *Usp. Mat. Nauk*. 13:3–28.
Lesk, M. 1975. LEX–a lexical analyzer generator. *Tech. Rep. 39. Bell Labs.* Murray Hill, NJ.
Li, M. and Vitányi, P. 1993. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer–Verlag, Berlin.
McCulloch, W. and Pitts, W. 1943. A logical calculus of ideas immanent in nervous activity. *Bull. Math. Biophys.* 5:115–133.
Post, E. 1943. Formal reductions of the general combinatorial decision problems. *Am. J. Math.* 65:197–215.
Rabin, M. and Scott, D. 1959. Finite automata and their decision problems. *IBM J. Res. Dev*. 3:114–125.
Robinson, R. 1991. Minsky's small universal Turing machine. *Int. J. Math.* 2(5):551–562.
Salomaa, A. 1966. Two complete axiom systems for the algebra of regular events. *J. ACM* 13(1):158–169.
Savitch, J. 1970. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci*. 4(2)177–192.
Schönhage, A. 1980. Storage modification machines. *SIAM J. Comput*. 9:490–508.
Searls, D. 1993. The computational linguistics of biological sequences. In *Artificial Intelligence and Molecular Biology*. L. Hunter, ed., pp. 47–120. MIT Press, Cambridge, MA.
Turing, A. 1936. On computable numbers with an application to the Entscheidungs problem. *Proc. London Math. Soc., Ser. 2* 42:230–265.
van Emde Boas, P. 1990. Machine models and simulations. In *Handbook of Theoretical Computer Science*. J. van Leeuwen, ed., pp. 1–66. Elsevier/MIT Press.
Wood, D. 1987. *Theory of Computation*. Harper and Row.

Further Information

The fundamentals of the theory of computation, automata theory, and formal languages can be found in many text books including Floyd and Beigel [1994], Gurari [1989], Harel [1992], Harrison [1978], Hopcroft and Ullman [1979], and Wood [1987]. The central focus of research in this area is to understand the relationships between the different resource complexity classes. This work is motivated in part by some major open questions about the relationships between resources (such as time and space) and the role of control mechanisms (nondeterminism/randomness). At the same time, new computational models are being introduced and studied. One such recent model that has led to the resolution of a number of interesting problems is the interactive proof systems. They exploit the power of randomness and interaction. Among their applications are new ways to encrypt information as well as some unexpected results about the difficulty of solving some difficult problems even approximately. Another new model is the quantum computational model that incorporates quantum-mechanical effects into the basic move of a Turing machine. There are also attempts to use molecular or cell-level interactions as the basic operations of a computer. Yet another research direction motivated in part by the advances in hardware technology is the study of neural networks, which model (albeit in a simplistic manner) the brain structure of mammals. The following chapters of this volume will present state-of-the-art information about many of these developments. The following annual conferences present the leading research work in computation theory: Association of Computer Machinery (ACM) Annual Symposium on Theory of Computing; Institute of Electrical and Electronics Engineers (IEEE) Symposium on the Foundations of Computer Science; IEEE Conference on Structure in Complexity Theory; International Colloquium on Automata,

Languages and Programming; Symposium on Theoretical Aspects of Computer Science; Mathematical Foundations of Computer Science; and Fundamentals of Computation Theory. There are many related conferences such as Computational Learning Theory, ACM Symposium on Principles of Distributed Computing, etc., where specialized computational models are studied for a specific application area. Concrete algorithms is another closely related area in which the focus is to develop algorithms for specific problems. A number of annual conferences are devoted to this field. We conclude with a list of major journals whose primary focus is in theory of computation: *The Journal of the Association of Computer Machinery, SIAM Journal on Computing, Journal of Computer and System Sciences, Information and Computation, Mathematical Systems Theory, Theoretical Computer Science, Computational Complexity, Journal of Complexity, Information Processing Letters, International Journal of Foundations of Computer Science,* and *ACTA Informatica.*

7

Graph and Network Algorithms

Samir Khuller
University of Maryland

Balaji Raghavachari
University of Texas at Dallas

7.1 Introduction

Graphs are useful in modeling many problems from different scientific disciplines because they capture the basic concept of objects (vertices) and relationships between objects (edges). Indeed, many optimization problems can be formulated in graph theoretic terms. Hence, algorithms on graphs have been widely studied. In this chapter, a few fundamental graph algorithms are described. For a more detailed treatment of graph algorithms, the reader is referred to textbooks on graph algorithms [Cormen et al. 2001, Even 1979, Gibbons 1985, Tarjan 1983].

An undirected *graph* $G = (V, E)$ is defined as a set V of *vertices* and a set E of *edges*. An edge $e = (u, v)$ is an unordered pair of vertices. A *directed graph* is defined similarly, except that its edges are ordered pairs of vertices; that is, for a directed graph, $E \subseteq V \times V$. The terms *nodes* and vertices are used interchangeably. In this chapter, it is assumed that the graph has neither self-loops, edges of the form (v, v), nor multiple edges connecting two given vertices. A graph is a **sparse graph** if $|E| \ll |V|^2$.

Bipartite graphs form a subclass of graphs and are defined as follows. A graph $G = (V, E)$ is bipartite if the vertex set V can be partitioned into two sets X and Y such that $E \subseteq X \times Y$. In other words, each edge of G connects a vertex in X with a vertex in Y. Such a graph is denoted by $G = (X, Y, E)$. Because bipartite graphs occur commonly in practice, algorithms are often specially designed for them.

A vertex w is *adjacent* to another vertex v if $(v, w) \in E$. An edge (v, w) is said to be *incident* on vertices v and w. The *neighbors* of a vertex v are all vertices $w \in V$ such that $(v, w) \in E$. The number of edges incident to a vertex v is called the **degree** of vertex v. For a directed graph, if (v, w) is an edge, then we say that the edge goes from v to w. The *out-degree* of a vertex v is the number of edges from v to other vertices. The *in-degree* of v is the number of edges from other vertices to v.

A **path** $p = [v_0, v_1, \ldots, v_k]$ from v_0 to v_k is a sequence of vertices such that (v_i, v_{i+1}) is an edge in the graph for $0 \leq i < k$. Any edge may be used only once in a path. A **cycle** is a path whose end vertices are the same, that is, $v_0 = v_k$. A path is *simple* if all its internal vertices are distinct. A cycle is *simple* if every node has exactly two edges incident to it in the cycle. A **walk** $w = [v_0, v_1, \ldots, v_k]$ from v_0 to v_k is a sequence of vertices such that (v_i, v_{i+1}) is an edge in the graph for $0 \leq i < k$, in which edges and vertices may be repeated. A walk is *closed* if $v_0 = v_k$. A graph is **connected** if there is a path between every pair of vertices. A directed graph is **strongly connected** if there is a path between every pair of vertices in each direction. An acyclic, undirected graph is a **forest**, and a **tree** is a connected forest. A directed graph without cycles is known as a **directed acyclic graph** (DAG). Consider a binary relation C between the vertices of an undirected graph G such that for any two vertices u and v, uCv if and only if there is a path in G between u and v. It can be shown that C is an equivalence relation, partitioning the vertices of G into equivalence classes, known as the connected components of G.

There are two convenient ways of representing graphs on computers. We first discuss the *adjacency list* representation. Each vertex has a linked list: there is one entry in the list for each of its adjacent vertices. The graph is thus represented as an array of linked lists, one list for each vertex. This representation uses $O(|V| + |E|)$ storage, which is good for sparse graphs. Such a storage scheme allows one to scan all vertices adjacent to a given vertex in time proportional to its degree. The second representation, the *adjacency matrix*, is as follows. In this scheme, an $n \times n$ array is used to represent the graph. The $[i, j]$ entry of this array is 1 if the graph has an edge between vertices i and j, and 0 otherwise. This representation permits one to test if there is an edge between any pair of vertices in constant time. Both these representation schemes can be used in a natural way to represent directed graphs. For all algorithms in this chapter, it is assumed that the given graph is represented by an adjacency list.

Section 7.2 discusses various types of tree traversal algorithms. Sections 7.3 and 7.4 discuss depth-first and breadth-first search techniques. Section 7.5 discusses the single source shortest path problem. Section 7.6 discusses minimum spanning trees. Section 7.7 discusses the bipartite matching problem and the single commodity maximum flow problem. Section 7.8 discusses some traversal problems in graphs, and the Further Information section concludes with some pointers to current research on graph algorithms.

7.2 Tree Traversals

A tree is *rooted* if one of its vertices is designated as the root vertex and all edges of the tree are oriented (directed) to point away from the root. In a rooted tree, there is a directed path from the root to any vertex in the tree. For any directed edge (u, v) in a rooted tree, u is v's *parent* and v is u's *child*. The *descendants* of a vertex w are all vertices in the tree (including w) that are reachable by directed paths starting at w. The *ancestors* of a vertex w are those vertices for which w is a descendant. Vertices that have no children are called **leaves**. A *binary tree* is a special case of a rooted tree in which each node has at most two children, namely, the left child and the right child. The trees rooted at the two children of a node are called the *left subtree* and *right subtree*.

In this section we study techniques for processing the vertices of a given binary tree in various orders. We assume that each vertex of the binary tree is represented by a record that contains fields to hold attributes of that vertex and two special fields *left* and *right* that point to its left and right subtree, respectively.

The three major tree traversal techniques are *preorder*, *inorder*, and *postorder*. These techniques are used as procedures in many tree algorithms where the vertices of the tree have to be processed in a specific order. In a preorder traversal, the root of any subtree has to be processed *before* any of its descendants. In a postorder traversal, the root of any subtree has to be processed *after* all of its descendants. In an inorder traversal, the root of a subtree is processed after all vertices in its left subtree have been processed, but

before any of the vertices in its right subtree are processed. Preorder and postorder traversals generalize to arbitrary rooted trees. In the example to follow, we show how postorder can be used to count the number of descendants of each node and store the value in that node. The algorithm runs in linear time in the size of the tree:

Postorder Algorithm. *PostOrder* (T):

```
1 if T ≠ nil then
2    lc ← PostOrder (left[T]).
3    rc ← PostOrder (right[T]).
4    desc[T] ← lc + rc + 1.
5    return desc[T].
6 else
7    return 0.
8 end-if
end-proc
```

7.3 Depth-First Search

Depth-first search (DFS) is a fundamental graph searching technique [Tarjan 1972, Hopcroft and Tarjan 1973]. Similar graph searching techniques were given earlier by Tremaux (see Fraenkel [1970] and Lucas [1882]). The structure of DFS enables efficient algorithms for many other graph problems such as biconnectivity, triconnectivity, and planarity [Even 1979].

The algorithm first initializes all vertices of the graph as being unvisited. Processing of the graph starts from an arbitrary vertex, known as the root vertex. Each vertex is processed when it is first discovered (also referred to as *visiting* a vertex). It is first marked as visited, and its adjacency list is then scanned for unvisited vertices. Each time an unvisited vertex is discovered, it is processed recursively by DFS. After a node's entire adjacency list has been explored, that invocation of the DFS procedure returns. This procedure eventually visits all vertices that are in the same connected component of the root vertex. Once DFS terminates, if there are still any unvisited vertices left in the graph, one of them is chosen as the root and the same procedure is repeated.

The set of edges such that each one led to the discovery of a new vertex form a maximal forest of the graph, known as the **DFS forest**; a *maximal forest* of a graph G is an acyclic subgraph of G such that the addition of any other edge of G to the subgraph introduces a cycle. The algorithm keeps track of this forest using parent pointers. In each connected component, only the root vertex has a *nil* parent in the DFS tree.

7.3.1 The Depth-First Search Algorithm

DFS is illustrated using an algorithm that labels vertices with numbers $1, 2, \ldots$ in such a way that vertices in the same component receive the same label. This labeling scheme is a useful preprocessing step in many problems. Each time the algorithm processes a new component, it numbers its vertices with a new label.

Depth-First Search Algorithm. *DFS-Connected-Component* (G):

```
1  c ← 0.
2  for all vertices v in G do
3     visited[v] ← false.
4     finished[v] ← false.
5     parent[v] ← nil.
6  end-for
7  for all vertices v in G do
8     if not visited [v] then
```

```
 9          c ← c + 1.
10          DFS (v, c).
11       end-if
12    end-for
end-proc
```

```
DFS (v, c):
1  visited[v] ← true.
2  component[v] ← c.
3  for all vertices w in adj[v] do
4     if not visited[w] then
5        parent[w] ← v.
6        DFS (w, c).
7     end-if
8  end-for
9  finished[v] ← true.
end-proc
```

7.3.2 Sample Execution

Figure 7.1 shows a graph having two connected components. DFS was started at vertex a, and the DFS forest is shown on the right. DFS visits the vertices b, d, c, e, and f, in that order. DFS then continues with vertices g, h, and i. In each case, the recursive call returns when the vertex has no more unvisited neighbors. Edges (d, a), (c, a), (f, d), and (i, g) are called *back edges* (these do not belong to the DFS forest).

7.3.3 Analysis

A vertex v is processed as soon as it is encountered, and therefore at the start of DFS (v), *visited*$[v]$ is *false*. Since *visited*$[v]$ is set to true as soon as DFS starts execution, each vertex is visited exactly once. Depth-first search processes each edge of the graph exactly twice, once from each of its incident vertices. Since the algorithm spends constant time processing each edge of G, it runs in $O(|V| + |E|)$ time.

Remark 7.1 In the following discussion, there is no loss of generality in assuming that the input graph is connected. For a rooted DFS tree, vertices u and v are said to be *related*, if either u is an ancestor of v, or vice versa.

DFS is useful due to the special way in which the edges of the graph may be classified with respect to a DFS tree. Notice that the DFS tree is not unique, and which edges are added to the tree depends on the

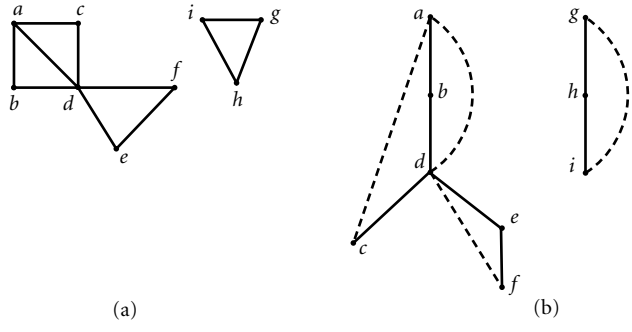


FIGURE 7.1 Sample execution of DFS on a graph having two connected components: (a) graph, (b) DFS forest.

order in which edges are explored while executing DFS. Edges of the DFS tree are known as *tree* edges. All other edges of the graph are known as *back* edges, and it can be shown that for any edge (u, v), u and v must be related. The graph does not have any *cross* edges, edges that connect two vertices that are unrelated. This property is utilized by a DFS-based algorithm that classifies the edges of a graph into **biconnected** components, maximal subgraphs that cannot be disconnected by the removal of any single vertex [Even 1979].

7.3.4 Directed Depth-First Search

The DFS algorithm extends naturally to directed graphs. Each vertex stores an adjacency list of its outgoing edges. During the processing of a vertex, first mark it as visited, and then scan its adjacency list for unvisited neighbors. Each time an unvisited vertex is discovered, it is processed recursively. Apart from tree edges and back edges (from vertices to their ancestors in the tree), directed graphs may also have *forward* edges (from vertices to their descendants) and *cross* edges (between unrelated vertices). There may be a cross edge (u, v) in the graph only if u is visited after the procedure call DFS (v) has completed execution.

7.3.5 Sample Execution

A sample execution of the directed DFS algorithm is shown in Figure 7.2. DFS was started at vertex a, and the DFS forest is shown on the right. DFS visits vertices b, d, f, and c in that order. DFS then returns and continues with e, and then g. From g, vertices h and i are visited in that order. Observe that (d, a) and (i, g) are back edges. Edges (c, d), (e, d), and (e, f) are cross edges. There is a single forward edge (g, i).

7.3.6 Applications of Depth-First Search

Directed DFS can be used to design a linear-time algorithm that classifies the edges of a given directed graph into *strongly connected* components: maximal subgraphs that have directed paths connecting any pair of vertices in them, in each direction. The algorithm itself involves running DFS twice, once on the original graph, and then a second time on G^R, which is the graph obtained by reversing the direction of all edges in G. During the second DFS, we are able to obtain all of the strongly connected components. The proof of this algorithm is somewhat subtle, and the reader is referred to Cormen et al. [2001] for details.

Checking if a graph has a cycle can be done in linear time using DFS. A graph has a cycle if and only if there exists a back edge relative to any of its depth-first search trees. A directed graph that does not have any cycles is known as a directed acyclic graph. DAGs are useful in modeling precedence constraints in scheduling problems, where nodes denote jobs/tasks, and a directed edge from u to v denotes the constraint that job u must be completed before job v can begin execution. Many problems on DAGs can be solved efficiently using dynamic programming.

A useful concept in DAGs is that of a **topological order**: a linear ordering of the vertices that is consistent with the partial order defined by the edges of the DAG. In other words, the vertices can be labeled with

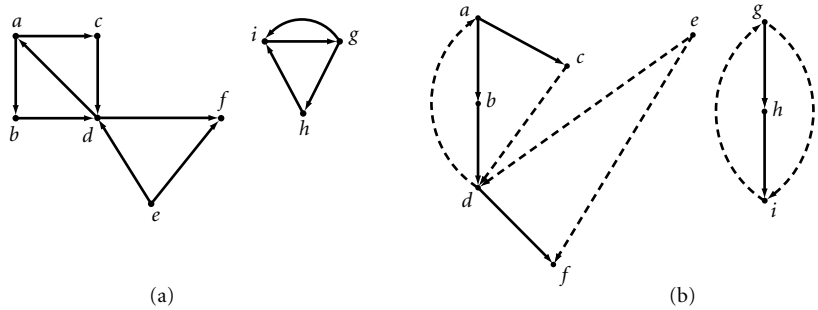


FIGURE 7.2 Sample execution of DFS on a directed graph: (a) graph, (b) DFS forest.

distinct integers in the range $[1 \ldots |V|]$ such that if there is a directed edge from a vertex labeled i to a vertex labeled j, then $i < j$. The vertices of a given DAG can be ordered topologically in linear time by a suitable modification of the DFS algorithm. We keep a counter whose initial value is $|V|$. As each vertex is marked finished, we assign the counter value as its topological number and decrement the counter. Observe that there will be no back edges; and that for all edges (u, v), v will be marked finished before u. Thus, the topological number of v will be higher than that of u. Topological sort has applications in diverse areas such as project management, scheduling, and circuit evaluation.

7.4 Breadth-First Search

Breadth-first search (BFS) is another natural way of searching a graph. The search starts at a root vertex r. Vertices are added to a queue as they are discovered, and processed in (first-in–first-out) (FIFO) order.

Initially, all vertices are marked as unvisited, and the queue consists of only the root vertex. The algorithm repeatedly removes the vertex at the front of the queue, and scans its neighbors in the graph. Any neighbor not visited is added to the end of the queue. This process is repeated until the queue is empty. All vertices in the same connected component as the root are scanned and the algorithm outputs a spanning tree of this component. This tree, known as a **breadth-first tree**, is made up of the edges that led to the discovery of new vertices. The algorithm labels each vertex v by $d[v]$, the distance (length of a shortest path) of v from the root vertex, and stores the BFS tree in the array p, using parent pointers. Vertices can be partitioned into levels based on their distance from the root. Observe that edges not in the BFS tree always go either between vertices in the same level, or between vertices in adjacent levels. This property is often useful.

Breadth-First Search Algorithm. *BFS-Distance* (G, r):

```
 1 MakeEmptyQueue (Q).
 2 for all vertices v in G do
 3    visited[v] ← false.
 4    d[v] ← ∞.
 5    p[v] ← nil.
 6 end-for
 7 visited[r] ← true.
 8 d[r] ← 0.
 9 Enqueue (Q, r).
10 while not Empty (Q) do
11    v ← Dequeue (Q).
12    for all vertices w in adj[v] do
13       if not visited [w] then
14          visited[w] ← true.
15          p[w] ← v.
16          d[w] ← d[v] + 1.
17          Enqueue (Q, w).
18       end-if
19    end-for
20 end-while
end-proc
```

7.4.1 Sample Execution

Figure 7.3 shows a connected graph on which BFS was run with vertex a as the root. When a is processed, vertices b, d, and c are added to the queue. When b is processed, nothing is done since all its neighbors have been visited. When d is processed, e and f are added to the queue. Finally c, e, and f are processed.

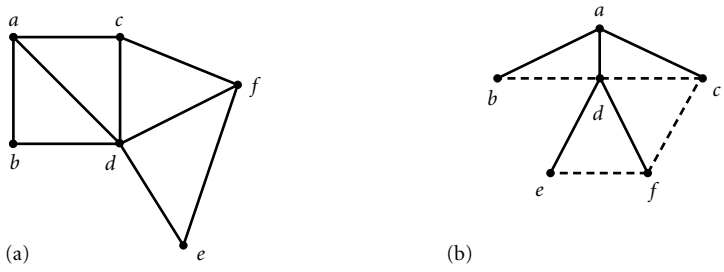


FIGURE 7.3 Sample execution of BFS on a graph: (a) graph, (b) BFS tree.

7.4.2 Analysis

There is no loss of generality in assuming that the graph G is connected, since the algorithm can be repeated in each connected component, similar to the DFS algorithm. The algorithm processes each vertex exactly once, and each edge exactly twice. It spends a constant amount of time in processing each edge. Hence, the algorithm runs in $O(|V| + |E|)$ time.

7.5 Single-Source Shortest Paths

A natural problem that often arises in practice is to compute the shortest paths from a specified node to all other nodes in an undirected graph. BFS solves this problem if all edges in the graph have the same length. Consider the more general case when each edge is given an arbitrary, non-negative length, and one needs to calculate a shortest length path from the root vertex to all other nodes of the graph, where the length of a path is defined to be the sum of the lengths of its edges. The distance between two nodes is the length of a shortest path between them.

7.5.1 Dijkstra's Algorithm

Dijkstra's algorithm [Dijkstra 1959] provides an efficient solution to this problem. For each vertex v, the algorithm maintains an upper bound to the distance from the root to vertex v in $d[v]$; initially $d[v]$ is set to infinity for all vertices except the root. The algorithm maintains a set S of vertices with the property that for each vertex $v \in S$, $d[v]$ is the length of a shortest path from the root to v. For each vertex u in $V - S$, the algorithm maintains $d[u]$, the shortest known distance from the root to u that goes entirely within S, except for the last edge. It selects a vertex u in $V - S$ of minimum $d[u]$, adds it to S, and updates the distance estimates to the other vertices in $V - S$. In this update step, it checks to see if there is a shorter path to any vertex in $V - S$ from the root that goes through u. Only the distance estimates of vertices that are adjacent to u are updated in this step. Because the primary operation is the selection of a vertex with minimum distance estimate, a priority queue is used to maintain the d-values of vertices. The priority queue should be able to handle a DecreaseKey operation to update the d-value in each iteration. The next algorithm implements Dijkstra's algorithm.

Dijkstra's Algorithm. *Dijkstra-Shortest Paths* (G, r):

```
1 for all vertices v in G do
2    visited[v] ← false.
3    d[v] ← ∞.
4    p[v] ← nil.
5 end-for
6 d[r] ← 0.
7 BuildPQ (H, d).
8 while not Empty (H) do
```

```
 9    u ← DeleteMin (H).
10    visited[u] ← true.
11    for all vertices v in adj[u] do
12      Relax (u, v).
13    end-for
14  end-while
end-proc

Relax (u, v)
1  if not visited[v] and d[v] > d[u] + w(u, v) then
2    d[v] ← d[u] + w(u, v).
3    p[v] ← u.
4    DecreaseKey (H, v, d[v]).
5  end-if
end-proc
```

7.5.1.1 Sample Execution

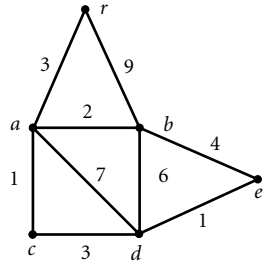
Figure 7.4 shows a sample execution of the algorithm. The column titled Iter specifies the number of iterations that the algorithm has executed through the **while** loop in step 8. In iteration 0, the initial values of the distance estimates are ∞. In each subsequent line of the table, the column marked u shows the vertex that was chosen in step 9 of the algorithm, and the change to the distance estimates at the end of that iteration of the **while** loop. In the first iteration, vertex r was chosen, after that a was chosen because it had the minimum distance label among the unvisited vertices, and so on. The distance labels of the unvisited neighbors of the visited vertex are updated in each iteration.

7.5.1.2 Analysis

The running time of the algorithm depends on the data structure that is used to implement the priority queue H. The algorithm performs $|V|$ DELETEMIN operations and, at most, $|E|$ DECREASEKEY operations. If a binary heap is used to update the records of any given vertex, each of these operations runs in $O(\log |V|)$ time. There is no loss of generality in assuming that the graph is connected. Hence, the algorithm runs in $O(|E| \log |V|)$. If a Fibonacci heap is used to implement the priority queue, the running time of the algorithm is $O(|E| + |V| \log |V|)$. Although the Fibonacci heap gives the best asymptotic running time, the binary heap implementation is likely to give better running times for most practical instances.

7.5.2 Bellman–Ford Algorithm

The shortest path algorithm described earlier directly generalizes to directed graphs, but it does not work correctly if the graph has edges of negative length. For graphs that have edges of negative length, but no



Iter	u	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$
0	—	∞	∞	∞	∞	∞
1	r	3	9	∞	∞	∞
2	a	3	5	4	10	∞
3	c	3	5	4	7	∞
4	b	3	5	4	7	9
5	d	3	5	4	7	8
6	e	3	5	4	7	8

FIGURE 7.4 Dijkstra's shortest path algorithm.

cycles of negative length, there is a different algorithm due to Bellman [1958] and Ford and Fulkerson [1962] that solves the single source shortest paths problem in $O(|V||E|)$ time.

The key to understanding this algorithm is the RELAX operation applied to an edge. In a single scan of the edges, we execute the RELAX operation on each edge. We then repeat the step $|V| - 1$ times. No special data structures are required to implement this algorithm, and the proof relies on the fact that a shortest path is simple and contains at most $|V| - 1$ edges (see Cormen et al. [2001] for a proof).

This problem also finds applications in finding a feasible solution to a system of linear equations, where each equation specifies a bound on the difference of two variables. Each constraint is modeled by an edge in a suitably defined directed graph. Such systems of equations arise in real-time applications.

7.6 Minimum Spanning Trees

The following fundamental problem arises in network design. A set of sites needs to be connected by a network. This problem has a natural formulation in graph-theoretic terms. Each site is represented by a vertex. Edges between vertices represent a potential link connecting the corresponding nodes. Each edge is given a nonnegative cost corresponding to the cost of constructing that link. A tree is a minimal network that connects a set of nodes. The cost of a tree is the sum of the costs of its edges. A minimum-cost tree connecting the nodes of a given graph is called a minimum-cost spanning tree, or simply a **minimum spanning tree**.

The problem of computing a minimum spanning tree (MST) arises in many areas, and as a subproblem in combinatorial and geometric problems. MSTs can be computed efficiently using algorithms that are greedy in nature, and there are several different algorithms for finding an MST. One of the first algorithms was due to Boruvka [1926]. The two algorithms that are popularly known as Prim's algorithm and Kruskal's algorithm are described here. (Prim's algorithm was first discovered by Jarnik [1930].)

7.6.1 Prim's Algorithm

Prim's [1957] algorithm for finding an MST of a given graph is one of the oldest algorithms to solve the problem. The basic idea is to start from a single vertex and gradually grow a tree, which eventually spans the entire graph. At each step, the algorithm has a tree that covers a set S of vertices, and looks for a *good* edge that may be used to extend the tree to include a vertex that is currently not in the tree. All edges that go from a vertex in S to a vertex in $V - S$ are candidate edges. The algorithm selects a minimum-cost edge from these candidate edges and adds it to the current tree, thereby adding another vertex to S.

As in the case of Dijkstra's algorithm, each vertex $u \in V - S$ can attach itself to only one vertex in the tree (so that cycles are not generated in the solution). Because the algorithm always chooses a minimum-cost edge, it needs to maintain a minimum-cost edge that connects u to some vertex in S as the candidate edge for including u in the tree. A priority queue of vertices is used to select a vertex in $V - S$ that is incident to a minimum-cost candidate edge.

Prim's Algorithm. *Prim-MST* (G, r):

```
 1 for all vertices v in G do
 2    visited[v] ← false.
 3    d[v] ← ∞.
 4    p[v] ← nil.
 5 end-for
 6 d[r] ← 0.
 7 BuildPQ (H, d).
 8 while not Empty (H) do
 9    u ← DeleteMin (H).
10    visited[u] ← true.
11    for all vertices v in adj[u] do
12       if not visited[v] and d[v] > w(u,v) then
```

```
13          d[v] ← w(u,v).
14          p[v] ← u.
15          DecreaseKey (H, v, d[v]).
16        end-if
17      end-for
18  end-while
end-proc
```

7.6.1.1 Analysis

First observe the similarity between Prim's and Dijkstra's algorithms. Both algorithms start building the tree from a single vertex and grow it by adding one vertex at a time. The only difference is the rule for deciding when the current label is updated for vertices outside the tree. Both algorithms have the same structure and therefore have similar running times. Prim's algorithm runs in $O(|E|\log|V|)$ time if the priority queue is implemented using binary heaps, and it runs in $O(|E|+|V|\log|V|)$ if the priority queue is implemented using Fibonacci heaps.

7.6.2 Kruskal's Algorithm

Kruskal's [1956] algorithm for finding an MST of a given graph is another classical algorithm for the problem, and is also greedy in nature. Unlike Prim's algorithm, which grows a single tree, Kruskal's algorithm grows a forest. First, the edges of the graph are sorted in nondecreasing order of their costs. The algorithm starts with the empty spanning forest (no edges). The edges of the graph are scanned in sorted order, and if the addition of the current edge does not generate a cycle in the current forest, it is added to the forest. The main test at each step is: does the current edge connect two vertices in the same connected component? Eventually, the algorithm adds $|V|-1$ edges to make a spanning tree in the graph.

The main data structure needed to implement the algorithm is for the maintenance of connected components, to ensure that the algorithm does not add an edge between two nodes in the same connected component. An abstract version of this problem is known as the Union-Find problem for a collection of disjoint sets. Efficient algorithms are known for this problem, where an arbitrary sequence of UNION and FIND operations can be implemented to run in almost linear time [Cormen et al. 2001, Tarjan 1983].

Kruskal's Algorithm. *Kruskal-MST* (G):

```
1 T ← ϕ.
2 for all vertices v in G do
3   Makeset(v).
4 Sort the edges of G by nondecreasing order of costs.
5 for all edges e = (u,v) in G in sorted order do
6   if Find (u) ≠ Find (v) then
7     T ← T ∪ (u,v).
8     Union (u, v).
9 end-proc
```

7.6.2.1 Analysis

The running time of the algorithm is dominated by step 4 of the algorithm in which the edges of the graph are sorted by nondecreasing order of their costs. This takes $O(|E|\log|E|)$ [which is also $O(|E|\log|V|)$] time using an efficient sorting algorithm such as Heap-sort. Kruskal's algorithm runs faster in the following special cases: if the edges are presorted, if the edge costs are within a small range, or if the number of different edge costs is bounded by a constant. In all of these cases, the edges can be sorted in linear time, and the algorithm runs in near-linear time, $O(|E|\,\alpha(|E|,|V|))$, where $\alpha(m,n)$ is the inverse Ackermann function [Tarjan 1983].

Remark 7.2 The MST problem can be generalized to directed graphs. The equivalent of trees in directed graphs are called **arborescences** or **branchings**; and because edges have directions, they are rooted spanning trees. An incoming branching has the property that every vertex has a unique path to the root. An outgoing branching has the property that there is a unique path from the root to each vertex in the graph. The input is a directed graph with arbitrary costs on the edges and a root vertex r. The output is a minimum-cost branching rooted at r. The algorithms discussed in this section for finding minimum spanning trees do not directly extend to the problem of finding optimal branchings. There are efficient algorithms that run in $O(|E|+|V|\log|V|)$ time using Fibonacci heaps for finding minimum-cost branchings [Gibbons 1985, Gabow et al. 1986]. These algorithms are based on techniques for weighted matroid intersection [Lawler 1976]. Almost linear-time deterministic algorithms for the MST problem in undirected graphs are also known [Fredman and Tarjan 1987].

7.7 Matchings and Network Flows

Networks are important both for electronic communication and for transporting goods. The problem of efficiently moving entities (such as bits, people, or products) from one place to another in an underlying network is modeled by the **network flow** problem. The problem plays a central role in the fields of operations research and computer science, and much emphasis has been placed on the design of efficient algorithms for solving it. Many of the basic algorithms studied earlier in this chapter play an important role in developing various implementations for network flow algorithms.

First the **matching** problem, which is a special case of the flow problem, is introduced. Then the **assignment problem**, which is a generalization of the matching problem to the weighted case, is studied. Finally, the network flow problem is introduced and algorithms for solving it are outlined.

The maximum matching problem is studied here in detail only for bipartite graphs. Although this restricts the class of graphs, the same principles are used to design polynomial time algorithms for graphs that are not necessarily bipartite. The algorithms for general graphs are complex due to the presence of structures called *blossoms*, and the reader is referred to Papadimitriou and Steiglitz [1982, Chapter 10], or Tarjan [1983, Chapter 9] for a detailed treatment of how blossoms are handled. Edmonds (see Even [1979]) gave the first algorithm to solve the matching problem in polynomial time. Micali and Vazirani [1980] obtained an $O(\sqrt{|V|}|E|)$ algorithm for nonbipartite matching by extending the algorithm by Hopcroft and Karp [1973] for the bipartite case.

7.7.1 Matching Problem Definitions

Given a graph $G = (V, E)$, a matching M is a subset of the edges such that no two edges in M share a common vertex. In other words, the problem is that of finding a set of independent edges that have no incident vertices in common. The cardinality of M is usually referred to as its *size*.

The following terms are defined with respect to a matching M. The edges in M are called *matched edges* and edges not in M are called *free edges*. Likewise, a vertex is a *matched vertex* if it is incident to a matched edge. A *free vertex* is one that is not matched. The *mate* of a matched vertex v is its neighbor w that is at the other end of the matched edge incident to v. A matching is called *perfect* if all vertices of the graph are matched in it. The objective of the maximum matching problem is to maximize $|M|$, the size of the matching. If the edges of the graph have weights, then the *weight* of a matching is defined to be the sum of the weights of the edges in the matching. A path $p = [v_1, v_2, \ldots, v_k]$ is called an *alternating path* if the edges (v_{2j-1}, v_{2j}), $j = 1, 2, \ldots$, are free and the edges (v_{2j}, v_{2j+1}), $j = 1, 2, \ldots$, are matched. An **augmenting path** $p = [v_1, v_2, \ldots, v_k]$ is an alternating path in which both v_1 and v_k are free vertices. Observe that an augmenting path is defined with respect to a specific matching. The symmetric difference of a matching M and an augmenting path P, $M \oplus P$, is defined to be $(M - P) \cup (P - M)$. The operation can be generalized to the case when P is any subset of the edges.

7.7.2 Applications of Matching

Matchings are the underlying basis for many optimization problems. Problems of assigning workers to jobs can be naturally modeled as a bipartite matching problem. Other applications include assigning a collection of jobs with precedence constraints to two processors, such that the total execution time is minimized [Lawler 1976]. Other applications arise in chemistry, in determining structure of chemical bonds, matching moving objects based on a sequence of photographs, and localization of objects in space after obtaining information from multiple sensors [Ahuja et al. 1993].

7.7.3 Matchings and Augmenting Paths

The following theorem gives necessary and sufficient conditions for the existence of a perfect matching in a bipartite graph.

Theorem 7.1 (Hall's Theorem.) *A bipartite graph $G = (X, Y, E)$ with $|X| = |Y|$ has a perfect matching if and only if $\forall S \subseteq X, |N(S)| \geq |S|$, where $N(S) \subseteq Y$ is the set of vertices that are neighbors of some vertex in S.*

Although Theorem 7.1 captures exactly the conditions under which a given bipartite graph has a perfect matching, it does not lead directly to an algorithm for finding maximum matchings. The following lemma shows how an augmenting path with respect to a given matching can be used to increase the size of a matching. An efficient algorithm that uses augmenting paths to construct a maximum matching incrementally is described later.

Lemma 7.1 *Let P be the edges on an augmenting path $p = [v_1, \ldots, v_k]$ with respect to a matching M. Then $M' = M \oplus P$ is a matching of cardinality $|M| + 1$.*

***Proof* 7.1** Since P is an augmenting path, both v_1 and v_k are free vertices in M. The number of free edges in P is one more than the number of matched edges. The symmetric difference operator replaces the matched edges of M in P by the free edges in P. Hence, the size of the resulting matching, $|M'|$, is one more than $|M|$. □

The following theorem provides a necessary and sufficient condition for a given matching M to be a maximum matching.

Theorem 7.2 *A matching M in a graph G is a maximum matching if and only if there is no augmenting path in G with respect to M.*

***Proof* 7.2** If there is an augmenting path with respect to M, then M cannot be a maximum matching, since by Lemma 7.1 there is a matching whose size is larger than that of M. To prove the converse we show that if there is no augmenting path with respect to M, then M is a maximum matching. Suppose that there is a matching M' such that $|M'| > |M|$. Consider the set of edges $M \oplus M'$. These edges form a subgraph in G. Each vertex in this subgraph has degree at most two, since each node has at most one edge from each matching incident to it. Hence, each connected component of this subgraph is either a path or a simple cycle. For each cycle, the number of edges of M is the same as the number of edges of M'. Since $|M'| > |M|$, one of the paths must have more edges from M' than from M. This path is an augmenting path in G with respect to the matching M, contradicting the assumption that there were no augmenting paths with respect to M. □

7.7.4 Bipartite Matching Algorithm

7.7.4.1 High-Level Description

The algorithm starts with the empty matching $M = \emptyset$, and augments the matching in phases. In each phase, an augmenting path with respect to the current matching M is found, and it is used to increase the size of the matching. An augmenting path, if one exists, can be found in $O(|E|)$ time, using a procedure similar to breadth-first search described in Section 7.4.

The search for an augmenting path proceeds from the free vertices. At each step when a vertex in X is processed, all its unvisited neighbors are also searched. When a matched vertex in Y is considered, only its matched neighbor is searched. This search proceeds along a subgraph referred to as the *Hungarian tree*.

Initially, all free vertices in X are placed in a queue that holds vertices that are yet to be processed. The vertices are removed one by one from the queue and processed as follows. In turn, when vertex v is removed from the queue, the edges incident to it are scanned. If it has a neighbor in the vertex set Y that is free, then the search for an augmenting path is successful; procedure AUGMENT is called to update the matching, and the algorithm proceeds to its next phase. Otherwise, add the mates of all of the matched neighbors of v to the queue if they have never been added to the queue, and continue the search for an augmenting path. If the algorithm empties the queue without finding an augmenting path, its current matching is a maximum matching and it terminates.

The main data structure that the algorithm uses consists of the arrays *mate* and *free*. The array *mate* is used to represent the current matching. For a matched vertex $v \in G$, $mate[v]$ denotes the matched neighbor of vertex v. For $v \in X$, $free[v]$ is a vertex in Y that is adjacent to v and is free. If no such vertex exists, then $free[v] = 0$.

Bipartite Matching Algorithm. *Bipartite Matching* $(G = (X, Y, E))$:

```
 1 for all vertices v in G do
 2    mate[v] ← 0.
 3 end-for
 4 found ← false.
 5 while not found do
 6      Initialize.
 7      MakeEmptyQueue (Q).
 8      for all vertices x ∈ X do
 9         if mate[x] = 0 then
10            Enqueue (Q,x).
11            label[x] ← 0.
12         endif
13      end-for
14      done ← false.
15      while not done and not Empty (Q) do
16         x ← Dequeue (Q).
17         if free[x] ≠ 0 then
18            Augment(x).
19            done ← true.
20         else
21            for all edges (x,x') ∈ A do
22               if label[x'] = 0 then
23                  label[x'] ← x.
24                  Enqueue (Q,x').
25               end-if
26            end-for
```

```
27          end-if
28          if Empty (Q) then
29             found ← true.
30          end-if
31       end-while
32    end-while
end-proc
```

```
Initialize :
1  for all vertices x ∈ X do
2     free[x] ← 0.
3  end-for
4  A ← ∅.
5  for all edges (x,y) ∈ E do
6     if mate[y] = 0 then free[x] ← y
7     else if mate[y] ≠ x then A ← A ∪ (x, mate[y]).
8     end-if
9  end-for
end-proc
```

```
Augment(x):
1  if label[x] = 0 then
2     mate[x] ← free[x].
3     mate[free[x]] ← x
4  else
5     free[label[x]] ← mate[x]
6     mate[x] ← free[x]
7     mate[free[x]] ← x
8     Augment (label[x])
9  end-if
end-proc
```

7.7.4.2 Sample Execution

Figure 7.5 shows a sample execution of the matching algorithm. We start with a partial matching and show the structure of the resulting Hungarian tree. An augmenting path from vertex b to vertex u is found by the algorithm.

7.7.4.3 Analysis

If there are augmenting paths with respect to the current matching, the algorithm will find at least one of them. Hence, when the algorithm terminates, the graph has no augmenting paths with respect to the current matching and the current matching is optimal. Each iteration of the main **while** loop of the algorithm runs in $O(|E|)$ time. The construction of the auxiliary graph A and computation of the array *free* also take $O(|E|)$ time. In each iteration, the size of the matching increases by one and thus there are, at most, $\min(|X|, |Y|)$ iterations of the **while** loop. Therefore, the algorithm solves the matching problem for bipartite graphs in time $O(\min(|X|, |Y|)|E|)$. Hopcroft and Karp [1973] showed how to improve the running time by finding a maximal set of shortest disjoint augmenting paths in a single phase in $O(|E|)$ time. They also proved that the algorithm runs in only $O(\sqrt{|V|})$ phases.

7.7.5 Assignment Problem

We now introduce the assignment problem, which is that of finding a maximum-weight matching in a given bipartite graph in which edges are given nonnegative weights. There is no loss of generality in

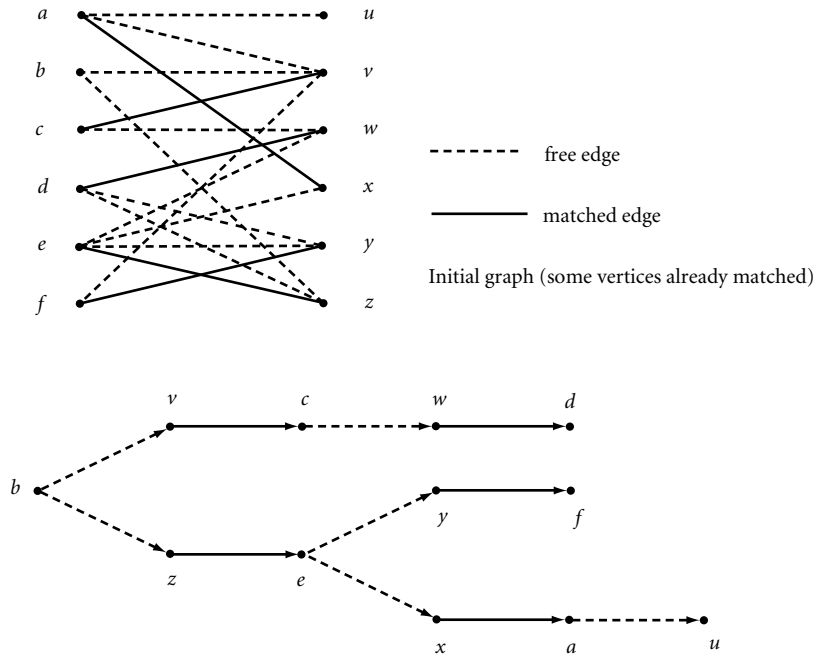


FIGURE 7.5 Sample execution of matching algorithm.

assuming that the graph is complete, since zero-weight edges may be added between pairs of vertices that are nonadjacent in the original graph without affecting the weight of a maximum-weight matching. The minimum-weight perfect matching can be reduced to the maximum-weight matching problem as follows: choose a constant M that is larger than the weight of any edge. Assign each edge a new weight of $w'(e) = M - w(e)$. Observe that maximum-weight matchings with the new weight function are minimum-weight perfect matchings with the original weights. We restrict our attention to the study of the maximum-weight matching problem for bipartite graphs. Similar techniques have been used to solve the maximum-weight matching problem in arbitrary graphs (see Lawler [1976] and Papadimitriou and Steiglitz [1982]).

The input is a complete bipartite graph $G = (X, Y, X \times Y)$ and each edge e has a nonnegative weight of $w(e)$. The following algorithm, known as the Hungarian method, was first given by Kuhn [1955]. The method can be viewed as a primal-dual algorithm in the linear programming framework [Papadimitriou and Steiglitz 1982]. No knowledge of linear programming is assumed here.

A *feasible vertex-labeling* ℓ is defined to be a mapping from the set of vertices in G to the real numbers such that for each edge (x_i, y_j) the following condition holds:

$$\ell(x_i) + \ell(y_j) \geq w(x_i, y_j)$$

The following can be verified to be a feasible vertex labeling. For each vertex $y_j \in Y$, set $\ell(y_j)$ to be 0; and for each vertex $x_i \in X$, set $\ell(x_i)$ to be the maximum weight of an edge incident to x_i,

$$\ell(y_j) = 0,$$
$$\ell(x_i) = \max_j w(x_i, y_j)$$

The *equality subgraph*, G_ℓ, is defined to be the subgraph of G, which includes all vertices of G but only those edges (x_i, y_j) that have weights such that

$$\ell(x_i) + \ell(y_j) = w(x_i, y_j)$$

The connection between equality subgraphs and maximum-weighted matchings is established by the following theorem.

Theorem 7.3 *If the equality subgraph, G_ℓ, has a perfect matching, M^*, then M^* is a maximum-weight matching in G.*

***Proof* 7.3** Let M^* be a perfect matching in G_ℓ. By definition,

$$w(M^*) = \sum_{e \in M^*} w(e) = \sum_{v \in X \cup Y} \ell(v)$$

Let M be any perfect matching in G. Then,

$$w(M) = \sum_{e \in M} w(e) \leq \sum_{v \in X \cup Y} \ell(v) = w(M^*)$$

Hence, M^* is a maximum-weight perfect matching. □

7.7.5.1 High-Level Description

Theorem 7.3 is the basis of the algorithm for finding a maximum-weight matching in a complete bipartite graph. The algorithm starts with a feasible labeling, then computes the equality subgraph and a maximum cardinality matching in this subgraph. If the matching found is perfect, by Theorem 7.3 the matching must be a maximum-weight matching and the algorithm returns it as its output. Otherwise, more edges need to be added to the equality subgraph by *revising* the vertex labels. The revision keeps edges from the current matching in the equality subgraph. After more edges are added to the equality subgraph, the algorithm grows the Hungarian trees further. Either the size of the matching increases because an augmenting path is found, or a new vertex is added to the Hungarian tree. In the former case, the current phase terminates and the algorithm starts a new phase, because the matching size has increased. In the latter case, new nodes are added to the Hungarian tree. In n phases, the tree includes all of the nodes, and therefore there are at most n phases before the size of the matching increases.

It is now described in more detail how the labels are updated and which edges are added to the equality subgraph G_ℓ. Suppose M is a maximum matching in G_ℓ found by the algorithm. Hungarian trees are grown from all the free vertices in X. Vertices of X (including the free vertices) that are encountered in the search are added to a set S, and vertices of Y that are encountered in the search are added to a set T. Let $\overline{S} = X - S$ and $\overline{T} = Y - T$. Figure 7.6 illustrates the structure of the sets S and T. Matched edges are shown in bold; the other edges are the edges in G_ℓ. Observe that there are no edges in the equality subgraph from S to $\overline{T}$, although there may be edges from T to $\overline{S}$. Let us choose δ to be the smallest value such that some edge of $G - G_\ell$ enters the equality subgraph. The algorithm now revises the labels as follows. Decrease all of the labels of vertices in S by δ and increase the labels of the vertices in T by δ. This ensures that edges in the matching continue to stay in the equality subgraph. Edges in G (not in G_ℓ) that go from vertices in S to vertices in $\overline{T}$ are candidate edges to enter the equality subgraph, since one label is decreasing and the other is unchanged. Suppose this edge goes from $x \in S$ to $y \in \overline{T}$. If y is free, then an augmenting path has been found. On the other hand, if y is matched, the Hungarian tree is grown by moving y to T and its matched neighbor to S, and the process of revising labels continues.

7.7.6 B-Matching Problem

The B-Matching problem is a generalization of the matching problem. In its simplest form, given an integer $b \geq 1$, the problem is to find a subgraph H of a given graph G such that the degree of each vertex is exactly equal to b in H (such a subgraph is called a *b-regular subgraph*). The problem can also be formulated as an optimization problem by seeking a subgraph H with most edges, with the degree of each vertex to

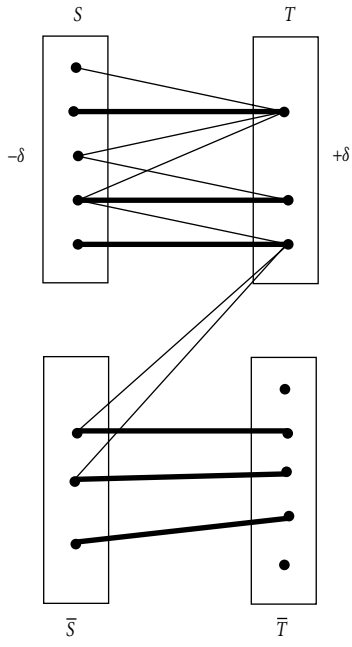


FIGURE 7.6 Sets S and T as maintained by the algorithm. Only edges in G_ℓ are shown.

be at most b in H. Several generalizations are possible, including different degree bounds at each vertex, degrees of some vertices unspecified, and edges with weights. All variations of the B-Matching problem can be solved using the techniques for solving the Matching problem.

In this section, we show how the problem can be solved for the unweighted B-Matching problem in which each vertex v is given a degree bound of $b[v]$, and the objective is to find a subgraph H in which the degree of each vertex v is exactly equal to $b[v]$. From the given graph G, construct a new graph G_b as follows. For each vertex $v \in G$, introduce $b[v]$ vertices in G_b, namely $v_1, v_2, \ldots, v_{b[v]}$. For each edge $e = (u, v)$ in G, add two new vertices e_u and e_v to G_b, along with the edge (e_u, e_v). In addition, add edges between v_i and e_v, for $1 \leq i \leq b[v]$ (and between u_j and e_u, for $1 \leq j \leq b[u]$). We now show that there is a natural one-to-one correspondence between B-Matchings in G and perfect matchings in G_b.

Given a B-Matching H in G, we show how to construct a perfect matching in G_b. For each edge $(u, v) \in H$, match e_u to the next available u_j, and e_v to the next available v_i. Since u is incident to exactly $b[u]$ edges in H, there are exactly enough nodes $u_1, u_2 \ldots u_{b[v]}$ in the previous step. For all edges $e = (u, v) \in G - H$, we match e_u and e_v. It can be verified that this yields a perfect matching in G_b.

We now show how to construct a B-Matching in G, given a perfect matching in G_b. Let M be a perfect matching in G_b. For each edge $e = (u, v) \in G$, if $(e_u, e_b) \in M$, then do not include the edge e in the B-Matching. Otherwise, e_u is matched to some u_j and e_v is matched to some v_i in M. In this case, we include e in our B-Matching. Since there are exactly $b[u]$ vertices $u_1, u_2, \ldots u_{b[u]}$, each such vertex introduces an edge into the B-Matching, and therefore the degree of u is exactly $b[u]$. Therefore, we get a B-Matching in G.

7.7.7 Network Flows

A number of polynomial time flow algorithms have been developed over the past two decades. The reader is referred to Ahuja et al. [1993] for a detailed account of the historical development of the various flow methods. Cormen et al. [2001] review the preflow push method in detail; and to complement their coverage, an implementation of the blocking flow technique of Malhotra et al. [1978] is discussed here.

7.7.8 Network Flow Problem Definitions

First the network flow problem and its basic terminology are defined.

Flow network: A flow network $G = (V, E)$ is a directed graph, with two specially marked nodes, namely, the source s and the sink t. There is a *capacity* function $c : E \mapsto R^+$ that maps edges to positive real numbers.

Max-flow problem: A flow function $f : E \mapsto R$ maps edges to real numbers. For an edge $e = (u, v)$, $f(e)$ refers to the flow on edge e, which is also called the net flow from vertex u to vertex v. This notation is extended to sets of vertices as follows: If X and Y are sets of vertices then $f(X, Y)$ is defined to be $\sum_{x \in X} \sum_{y \in Y} f(x, y)$. A flow function is required to satisfy the following constraints:

- *Capacity constraint.* For all edges e, $f(e) \leq c(e)$.
- *Skew symmetry constraint.* For an edge $e = (u, v)$, $f(u, v) = -f(v, u)$.
- *Flow conservation.* For all vertices $u \in V - \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$.

The capacity constraint says that the total flow on an edge does not exceed its capacity. The skew symmetry condition says that the flow on an edge is the negative of the flow in the reverse direction. The flow conservation constraint says that the total net flow out of any vertex other than the source and sink is zero.

The *value* of the flow is defined as

$$|f| = \sum_{v \in V} f(s, v)$$

In other words, it is the net flow out of the source. In the *maximum-flow problem*, the objective is to find a flow function that satisfies the three constraints, and also maximizes the total flow value $|f|$.

Remark 7.3 This formulation of the network flow problem is powerful enough to capture generalizations where there are many sources and sinks (single commodity flow), and where both vertices and edges have capacity constraints, etc.

First, the notion of cuts is defined, and the max-flow min-cut theorem is introduced. Then, residual networks, layered networks, and the concept of blocking flows are introduced. Finally, an efficient algorithm for finding a blocking flow is described.

An ***s*–*t* cut** of the graph is a partitioning of the vertex set V into two sets S and $T = V - S$ such that $s \in S$ and $t \in T$. If f is a flow, then the net flow across the cut is defined as $f(S, T)$. The capacity of the cut is similarly defined as $c(S, T) = \sum_{x \in X} \sum_{y \in Y} c(x, y)$. The net flow across a cut may include negative net flows between vertices, but the capacity of the cut includes only nonnegative values, that is, only the capacities of edges from S to T.

Using the flow conservation principle, it can be shown that the net flow across an s–t cut is exactly the flow value $|f|$. By the capacity constraint, the flow across the cut cannot exceed the capacity of the cut. Thus, the value of the maximum flow is no greater than the capacity of a minimum s–t cut. The well-known *max-flow min-cut theorem* [Elias et al. 1956, Ford and Fulkerson 1962] proves that the two numbers are actually equal. In other words, if f^* is a maximum flow, then there is some cut $(X, \overline{X})$ such that $|f^*| = c(X, \overline{X})$. The reader is referred to Cormen et al. [2001] and Tarjan [1983] for further details.

The *residual capacity* of a flow f is defined to be a function on vertex pairs given by $c'(v, w) = c(v, w) - f(v, w)$. The residual capacity of an edge (v, w), $c'(v, w)$, is the number of additional units of flow that can be pushed from v to w without violating the capacity constraints. An edge e is *saturated* if $c(e) = f(e)$, that is, if its residual capacity, $c'(e)$, is zero. The residual graph $G_R(f)$ for a flow f is the graph with vertex set V, source and sink s and t, respectively, and those edges (v, w) for which $c'(v, w) > 0$.

An augmenting path for f is a path P from s to t in $G_R(f)$. The residual capacity of P, denoted by $c'(P)$, is the minimum value of $c'(v, w)$ over all edges (v, w) in the path P. The flow can be increased by $c'(P)$, by increasing the flow on each edge of P by this amount. Whenever $f(v, w)$ is changed, $f(w, v)$ is also correspondingly changed to maintain skew symmetry.

Most flow algorithms are based on the concept of augmenting paths pioneered by Ford and Fulkerson [1956]. They start with an initial zero flow and augment the flow in stages. In each stage, a residual graph $G_R(f)$ with respect to the current flow function f is constructed and an augmenting path in $G_R(f)$ is found to increase the value of the flow. Flow is increased along this path until an edge in this path is saturated. The algorithms iteratively keep increasing the flow until there are no more augmenting paths in $G_R(f)$, and return the final flow f as their output.

The following lemma is fundamental in understanding the basic strategy behind these algorithms.

Lemma 7.2 *Let f be any flow and f^* a maximum flow in G, and let $G_R(f)$ be the residual graph for f. The value of a maximum flow in $G_R(f)$ is $|f^*| - |f|$.*

***Proof* 7.4** Let f' be any flow in $G_R(f)$. Define $f + f'$ to be the flow defined by the flow function $f(v,w) + f'(v,w)$ for each edge (v,w). Observe that $f + f'$ is a feasible flow in G of value $|f| + |f'|$. Since f^* is the maximum flow possible in G, $|f'| \leq |f^*| - |f|$. Similarly define $f^* - f$ to be a flow in $G_R(f)$ defined by $f^*(v,w) - f(v,w)$ in each edge (v,w), and this is a feasible flow in $G_R(f)$ of value $|f^*| - |f|$, and it is a maximum flow in $G_R(f)$. □

Blocking flow: A flow f is a **blocking flow** if every path in G from s to t contains a saturated edge. It is important to note that a blocking flow is not necessarily a maximum flow. There may be augmenting paths that increase the flow on some edges and decrease the flow on other edges (by increasing the flow in the reverse direction).

Layered networks: Let $G_R(f)$ be the residual graph with respect to a flow f. The level of a vertex v is the length of a shortest path (using the least number of edges) from s to v in $G_R(f)$. The level graph L for f is the subgraph of $G_R(f)$ containing vertices reachable from s and only the edges (v,w) such that $\text{dist}(s,w) = 1 + \text{dist}(s,v)$. L contains all shortest-length augmenting paths and can be constructed in $O(|E|)$ time.

The Maximum Flow algorithm proposed by Dinitz [1970] starts with the zero flow, and iteratively increases the flow by augmenting it with a blocking flow in $G_R(f)$ until t is not reachable from s in $G_R(f)$. At each step the current flow is replaced by the sum of the current flow and the blocking flow. Since in each iteration the shortest distance from s to t in the residual graph increases, and the shortest path from s to t is at most $|V| - 1$, this gives an upper bound on the number of iterations of the algorithm.

An algorithm to find a blocking flow that runs in $O(|V|^2)$ time is described here, and this yields an $O(|V|^3)$ max-flow algorithm. There are a number of $O(|V|^2)$ blocking flow algorithms available [Karzanov 1974, Malhotra et al. 1978, Tarjan 1983], some of which are described in detail in Tarjan [1983].

7.7.9 Blocking Flows

Dinitz's algorithm to find a blocking flow runs in $O(|V||E|)$ time [Dinitz 1970]. The main step is to find paths from the source to the sink and saturate them by pushing as much flow as possible on these paths. Every time the flow is increased by pushing more flow along an augmenting path, one of the edges on this path becomes saturated. It takes $O(|V|)$ time to compute the amount of flow that can be pushed on the path. Since there are $|E|$ edges, this yields an upper bound of $O(|V||E|)$ steps on the running time of the algorithm.

Malhotra–Kumar–Maheshwari Blocking Flow Algorithm. The algorithm has a current flow function f and its corresponding residual graph $G_R(f)$. Define for each node $v \in G_R(f)$, a quantity $tp[v]$ that specifies its maximum throughput, that is, either the sum of the capacities of the incoming arcs or the sum of the capacities of the outgoing arcs, whichever is smaller. $tp[v]$ represents the maximum flow that could pass through v in any feasible blocking flow in the residual graph. Vertices for which the throughput is zero are deleted from $G_R(f)$.

The algorithm selects a vertex u for which its throughput is a minimum among all vertices with nonzero throughput. It then greedily pushes a flow of $tp[u]$ from u toward t, level by level in the layered residual

graph. This can be done by creating a queue, which initially contains u and which is assigned the task of pushing $tp[u]$ out of it. In each step, the vertex v at the front of the queue is removed, and the arcs going out of v are scanned one at a time, and as much flow as possible is pushed out of them until v's allocated flow has been pushed out. For each arc (v, w) that the algorithm pushed flow through, it updates the residual capacity of the arc (v, w) and places w on a queue (if it is not already there) and increments the net incoming flow into w. Also, $tp[v]$ is reduced by the amount of flow that was sent through it now. The flow finally reaches t, and the algorithm never comes across a vertex that has incoming flow that exceeds its outgoing capacity since u was chosen as a vertex with the smallest throughput. The preceding idea is again repeated to pull a flow of $tp[u]$ from the source s to u. Combining the two steps yields a flow of $tp[u]$ from s to t in the residual network that goes through u. The flow f is augmented by this amount. Vertex u is deleted from the residual graph, along with any other vertices that have zero throughput.

This procedure is repeated until all vertices are deleted from the residual graph. The algorithm has a blocking flow at this stage since at least one vertex is saturated in every path from s to t. In the algorithm, whenever an edge is saturated, it may be deleted from the residual graph. Since the algorithm uses a greedy strategy to send flows, at most $O(|E|)$ time is spent when an edge is saturated. When finding flow paths to push $tp[u]$, there are at most n times, one each per vertex, when the algorithm pushes a flow that does not saturate the corresponding edge. After this step, u is deleted from the residual graph. Hence, in $O(|E| + |V|^2) = O(|V|^2)$ steps, the algorithm to compute blocking flows terminates.

Goldberg and Tarjan [1988] proposed a preflow push method that runs in $O(|V||E| \log |V|^2/|E|)$ time without explicitly finding a blocking flow at each step.

7.7.10 Applications of Network Flow

There are numerous applications of the Maximum Flow algorithm in scheduling problems of various kinds. See Ahuja et al. [1993] for further details.

7.8 Tour and Traversal Problems

There are many applications for finding certain kinds of paths and tours in graphs. We briefly discuss some of the basic problems.

The **traveling salesman problem** (**TSP**) is that of finding a shortest tour that visits all of the vertices in a given graph with weights on the edges. It has received considerable attention in the literature [Lawler et al. 1985]. The problem is known to be computationally intractable (NP-hard). Several heuristics are known to solve practical instances. Considerable progress has also been made for finding optimal solutions for graphs with a few thousand vertices.

One of the first graph-theoretic problems to be studied, the **Euler tour problem** asks for the existence of a closed walk in a given connected graph that traverses each edge exactly once. Euler proved that such a closed walk exists if and only if each vertex has even degree [Gibbons 1985]. Such a graph is known as an **Eulerian graph**. Given an Eulerian graph, a Euler tour in it can be computed using DFS in linear time.

Given an edge-weighted graph, the **Chinese postman problem** is that of finding a shortest closed walk that traverses each edge at least once. Although the problem sounds very similar to the TSP problem, it can be solved optimally in polynomial time by reducing it to the matching problem [Ahuja et al. 1993].

Acknowledgments

Samir Khuller's research is supported by National Science Foundation (NSF) Awards CCR-9820965 and CCR-0113192.

Balaji Raghavachari's research is supported by the National Science Foundation under Grant CCR-9820902.

Defining Terms

Assignment problem: That of finding a perfect matching of maximum (or minimum) total weight.
Augmenting path: An alternating path that can be used to augment (increase) the size of a matching.
Biconnected graph: A graph that cannot be disconnected by the removal of any single vertex.
Bipartite graph: A graph in which the vertex set can be partitioned into two sets X and Y, such that each edge connects a node in X with a node in Y.
Blocking flow: A flow function in which any directed path from s to t contains a saturated edge.
Branching: A spanning tree in a rooted graph, such that the root has a path to each vertex.
Chinese postman problem: Asks for a minimum length tour that traverses each edge at least once.
Connected: A graph in which there is a path between each pair of vertices.
Cycle: A path in which the start and end vertices of the path are identical.
Degree: The number of edges incident to a vertex in a graph.
DFS forest: A rooted forest formed by depth-first search.
Directed acyclic graph: A directed graph with no cycles.
Eulerian graph: A graph that has an Euler tour.
Euler tour problem: Asks for a traversal of the edges that visits each edge exactly once.
Forest: An acyclic graph.
Leaves: Vertices of degree one in a tree.
Matching: A subset of edges that do not share a common vertex.
Minimum spanning tree: A spanning tree of minimum total weight.
Network flow: An assignment of flow values to the edges of a graph that satisfies flow conservation, skew symmetry, and capacity constraints.
Path: An ordered list of edges such that any two consecutive edges are incident to a common vertex.
Perfect matching: A matching in which every node is matched by an edge to another node.
Sparse graph: A graph in which $|E| \ll |V|^2$.
***s*–*t* cut:** A partitioning of the vertex set into S and T such that $s \in S$ and $t \in T$.
Strongly connected: A directed graph in which there is a directed path in each direction between each pair of vertices.
Topological order: A linear ordering of the edges of a DAG such that every edge in the graph goes from left to right.
Traveling salesman problem: Asks for a minimum length tour of a graph that visits all of the vertices exactly once.
Tree: An acyclic graph with $|V| - 1$ edges.
Walk: An ordered sequence of edges (in which edges could repeat) such that any two consecutive edges are incident to a common vertex.

References

Ahuja, R.K., Magnanti, T., and Orlin, J. 1993. *Network Flows*. Prentice Hall, Upper Saddle River, NJ.

Bellman, R. 1958. On a routing problem. *Q. App. Math.*, 16(1):87–90.

Boruvka, O. 1926. O jistem problemu minimalnim. *Praca Moravske Prirodovedecke Spolecnosti*, 3:37–58 (in Czech).

Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. 2001. *Introduction to Algorithms, second edition*. The MIT Press.

DiBattista, G., Eades, P., Tamassia, R., and Tollis, I. 1994. Annotated bibliography on graph drawing algorithms. *Comput. Geom.: Theory Applic.*, 4:235–282.

Dijkstra, E.W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

Dinitz, E.A. 1970. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Dokl.*, 11:1277–1280.

Elias, P., Feinstein, A., and Shannon, C.E. 1956. Note on maximum flow through a network. *IRE Trans. Inf. Theory*, IT-2:117–119.
Even, S. 1979. *Graph Algorithms*. Computer Science Press, Potomac, MD.
Ford, L.R., Jr. and Fulkerson, D.R. 1956. Maximal flow through a network. *Can. J. Math*., 8:399–404.
Ford, L.R., Jr. and Fulkerson, D.R. 1962. *Flows in Networks*. Princeton University Press.
Fraenkel, A.S. 1970. Economic traversal of labyrinths. *Math. Mag.*, 43:125–130.
Fredman, M. and Tarjan, R.E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615.
Gabow, H.N., Galil, Z., Spencer, T., and Tarjan, R.E. 1986. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122.
Gibbons, A.M. 1985. *Algorithmic Graph Theory*. Cambridge University Press, New York.
Goldberg, A.V. and Tarjan, R.E. 1988. A new approach to the maximum-flow problem. *J. ACM*, 35:921–940.
Hochbaum, D.S., Ed. 1996. *Approximation Algorithms for NP-Hand Problems.* PWS Publishing.
Hopcroft, J.E. and Karp, R.M. 1973. An $n^{2.5}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231.
Hopcroft, J.E. and Tarjan, R.E. 1973. Efficient algorithms for graph manipulation. *Commun. ACM*, 16:372–378.
Jarnik, V. 1930. O jistem problemu minimalnim. *Praca Moravske Prirodovedecke Spolecnosti*, 6:57–63 (in Czech).
Karzanov, A.V. 1974. Determining the maximal flow in a network by the method of preflows. *Soviet Math. Dokl.*, 15:434–437.
Kruskal, J.B., 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Am. Math. Soc.*, 7:48–50.
Kuhn, H.W. 1955. The Hungarian method for the assignment problem. *Nav. Res. Logistics Q.*, 2:83–98.
Lawler, E.L. 1976. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston.
Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D.B. 1985. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, New York.
Lucas, E. 1882. *Recreations Mathematiques*. Paris.
Malhotra, V.M., Kumar, M.P., and Maheshwari, S.N. 1978. An $O(|V|^3)$ algorithm for finding maximum flows in networks. *Inf. Process. Lett.*, 7:277–278.
Micali, S. and Vazirani, V.V. 1980. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs, pp. 17–27. In *Proc. 21st Annu. Symp. Found. Comput. Sci.*
Papadimitriou, C.H. and Steiglitz, K. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Upper Saddle River, NJ.
Prim, R.C. 1957. Shortest connection networks and some generalizations. *Bell Sys. Tech. J.*, 36:1389–1401.
Tarjan, R.E. 1972. Depth first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160.
Tarjan, R.E. 1983. *Data Structures and Network Algorithms*. *SIAM*.

Further Information

The area of graph algorithms continues to be a very active field of research. There are several journals and conferences that discuss advances in the field. Here we name a partial list of some of the important meetings: ACM Symposium on Theory of Computing, IEEE Conference on Foundations of Computer Science, ACM–SIAM Symposium on Discrete Algorithms, the International Colloquium on Automata, Languages and Programming, and the European Symposium on Algorithms. There are many other regional algorithms/theory conferences that carry research papers on graph algorithms. The journals that carry articles on current research in graph algorithms are *Journal of the ACM, SIAM Journal on Computing, SIAM Journal on Discrete Mathematics, Journal of Algorithms, Algorithmica, Journal of Computer and System Sciences, Information and Computation, Information Processing Letters,* and *Theoretical Computer Science*.

To find more details about some of the graph algorithms described in this chapter we refer the reader to the books by Cormen et al. [2001], Even [1979], and Tarjan [1983]. For network flows and matching, a more detailed survey regarding various approaches can be found in Tarjan [1983]. Papadimitriou and Steiglitz [1982] discuss the solution of many combinatorial optimization problems using a primal–dual framework.

Current research on graph algorithms focuses on approximation algorithms [Hochbaum 1996], dynamic algorithms, and in the area of graph layout and drawing [DiBattista et al. 1994].

8

Algebraic Algorithms

Angel Diaz
IBM Research

Erich Kaltófen
North Carolina State University

Victor Y. Pan
Lehman College, CUNY

8.1 Introduction

The title's subject is the algorithmic approach to algebra: arithmetic with numbers, polynomials, matrices, differential polynomials, such as $y'' + (1/2 + x^4/4)y$, truncated series, and algebraic sets, i.e., quantified expressions such as $\exists x \in \mathbb{R} : x^4 + p \cdot x + q = 0$, which describes a subset of the two-dimensional space with coordinates p and q for which the given quartic equation has a real root. Algorithms that manipulate such objects are the backbone of modern symbolic mathematics software such as the Maple and Mathematica systems, to name but two among many useful systems. This chapter restricts itself to algorithms in four areas: linear matrix algebra, root finding of univariate polynomials, solution of systems of nonlinear algebraic equations, and polynomial factorization.

8.2 Matrix Computations and Approximation of Polynomial Zeros

This section covers several major algebraic and numerical problems of scientific and engineering computing that are usually solved numerically, with rounding off or chopping the input and computed values to a fixed number of bits that fit the computer precision (Sections 8.2 and 8.3 are devoted to some fundamental

infinite precision symbolic computations, and within Section 8.2 we comment on the infinite precision techniques for some matrix computations). We also study approximation of polynomial zeros, which is an important, fundamental, as well as very popular subject. In our presentation, we will very briefly list the major subtopics of our huge subject and will give some pointers to the references. We will include brief coverage of the topics of the algorithm design and analysis, regarding the complexity of matrix computation and of approximating polynomial zeros. The reader may find further material on these subjects in the survey articles by Pan [1984a, 1991, 1992a, 1995b] and in the books by Bini and Pan [1994, 1996].

8.2.1 Products of Vectors and Matrices, Convolution of Vectors

An $m \times n$ matrix $A = (a_{i,j},\ i = 0, 1, \ldots, m-1;\ j = 0, 1, \ldots, n-1)$ is a two-dimensional array, whose (i, j) entry is $(A)_{i,j} = a_{i,j}$. A is a column vector of dimension m if $n = 1$ and is a row vector of dimension n if $m = 1$. Transposition, hereafter, indicated by the superscript T, transforms a row vector $\boldsymbol{v}^T = [v_0, \ldots, v_{n-1}]$ into a column vector $\boldsymbol{v} = [v_0, \ldots, v_{n-1}]^T$.

For two vectors, $\boldsymbol{u}^T = (u_0, \ldots, u_{m-1})$ and $\boldsymbol{v}^T = (v_0, \ldots, v_{n-1})^T$, their *outer product* is an $m \times n$ matrix,

$$W = \boldsymbol{u}\boldsymbol{v}^T = [w_{i,j},\ i = 0, \ldots, m-1;\ j = 0, \ldots, n-1]$$

where $w_{i,j} = u_i v_j$, for all i and j, and their *convolution* vector is said to equal

$$\boldsymbol{w} = \boldsymbol{u} \circ \boldsymbol{v} = (w_0, \ldots, w_{m+n-2})^T, \qquad w_k = \sum_{i=0}^{k} u_i v_{k-i}$$

where $u_i = v_j = 0$, for $i \geq m$, $j \geq n$; in fact, $\boldsymbol{w}$ is the coefficient vector of the product of two polynomials,

$$u(x) = \sum_{i=0}^{m-1} u_i x^i \qquad \text{and} \qquad v(x) = \sum_{i=0}^{n-1} v_i x^i$$

having coefficient vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, respectively.

If $m = n$, the scalar value

$$\boldsymbol{v}^T\boldsymbol{u} = \boldsymbol{u}^T\boldsymbol{v} = u_0v_0 + u_1v_1 + \cdots + u_{n-1}v_{n-1} = \sum_{i=0}^{n-1} u_i v_i$$

is called the *inner* (*dot*, or *scalar*) *product* of $\boldsymbol{u}$ and $\boldsymbol{v}$.

The straightforward algorithms compute the inner and outer products of $\boldsymbol{u}$ and $\boldsymbol{v}$ and their convolution vector by using $2n-1$, mn, and $mn + (m-1)(n-1) = 2mn - m - n + 1$ arithmetic operations (hereafter, referred to as **ops**), respectively.

These upper bounds on the numbers of ops for computing the inner and outer products are sharp, that is, cannot be decreased, for the general pair of the input vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, whereas (see, e.g., Bini and Pan [1994]) one may apply the *fast fourier transform* (FFT) in order to compute the convolution vector $\boldsymbol{u} \circ \boldsymbol{v}$ much faster, for larger m and n; namely, it suffices to use $4.5K \log K + 2K$ ops, for $K = 2^k$, $k = \lceil \log(m + n + 1) \rceil$. (Here and hereafter, all logarithms are binary unless specified otherwise.)

If $A = (a_{i,j})$ and $B = (b_{j,k})$ are $m \times n$ and $n \times p$ matrices, respectively, and $\boldsymbol{v} = (v_k)$ is a p-dimensional vector, then the straightforward algorithms compute the vector

$$\boldsymbol{w} = B\boldsymbol{v} = (w_0, \ldots, w_{n-1})^T, \qquad w_i = \sum_{j=0}^{p-1} b_{i,j} v_j, \quad i = 0, \ldots, n-1$$

by using $(2p-1)n$ ops (sharp bound), and compute the *matrix product*

$$AB = (w_{i,k},\ i = 0, \ldots, m-1;\ k = 0, \ldots, p-1)$$

by using $2mnp - mp$ ops, which is $2n^3 - n^2$ if $m = n = p$. The latter upper bound is not sharp: the subroutines for $n \times n$ matrix multiplication on some modern computers, such as CRAY and Connection

Machines, rely on algorithms using $O(n^{2.81})$ ops, and some nonpractical algorithms involve $O(n^{2.376})$ ops [Bini and Pan 1994, Golub and Van Loan 1989].

In the special case, where all of the input entries and components are bounded integers having short binary representation, each of the preceding operations with vectors and matrices can be reduced to a single multiplication of 2 longer integers, by means of the techniques of *binary segmentation* (cf. Pan [1984b, Section 40], Pan [1991], Pan [1992b], or Bini and Pan [1994, Examples 36.1–36.3]).

For an $n \times n$ matrix B and an n-dimensional vector $\boldsymbol{v}$, one may compute the vectors $B^i\boldsymbol{v}$, $i = 1, 2, \ldots, k-1$, which define *Krylov sequence* or *Krylov matrix*

$$[B^i\boldsymbol{v},\ i = 0, 1, \ldots, k-1]$$

used as a basis of several computations. The straightforward algorithm takes on $(2n-1)nk$ ops, which is order n^3 if k is of order n. An alternative algorithm first computes the matrix powers

$$B^2, B^4, B^8, \ldots, B^{2^s}, \quad s = \lceil \log k \rceil - 1$$

and then the products of $n \times n$ matrices B^{2^i} by $n \times 2^i$ matrices, for $i = 0, 1, \ldots, s$,

$$\begin{aligned} &B && v \\ &B^2 && (v,\ Bv) = (B^2v,\ B^3v) \\ &B^4 && (v,\ Bv,\ B^2v,\ B^3v) = (B^4v,\ B^5v,\ B^6v,\ B^7v) \\ &\vdots \end{aligned}$$

The last step completes the evaluation of the Krylov sequence, which amounts to $2s$ matrix multiplications, for $k = n$, and, therefore, can be performed (in theory) in $O(n^{2.376} \log k)$ ops.

8.2.2 Some Computations Related to Matrix Multiplication

Several fundamental matrix computations can be ultimately reduced to relatively few [that is, to a constant number, or, say, to $O(\log n)$] $n \times n$ matrix multiplications. These computations include the evaluation of $\det A$, the **determinant** of an $n \times n$ matrix A; of its *inverse* A^{-1} (where A is nonsingular, that is, where $\det A \neq 0$); of the coefficients of its **characteristic polynomial**, $c_A(x) = \det(xI - A)$, x denoting a scalar variable and I being the $n \times n$ identity matrix, which has ones on its diagonal and zeros elsewhere; of its *minimal polynomial*, $m_A(x)$; of its *rank*, rank A; of the solution vector $\boldsymbol{x} = A^{-1}\boldsymbol{v}$ to a nonsingular *linear system of equations*, $A\boldsymbol{x} = \boldsymbol{v}$; of various *orthogonal* and *triangular factorizations* of A; and of a submatrix of A having the maximal rank, as well as some fundamental computations with singular matrices. Consequently, all of these operations can be performed by using (theoretically) $O(n^{2.376})$ ops (cf. Bini and Pan [1994, Chap. 2]). The idea is to represent the input matrix A as a block matrix and, operating with its blocks (rather than with its entries), to apply fast matrix multiplication algorithms. In practice, due to various other considerations (accounting, in particular, for the overhead constants hidden in the O notation, for the memory space requirements, and particularly, for numerical stability problems), these computations are based either on the straightforward algorithm for matrix multiplication or on other methods allowing order n^3 arithmetic operations (cf. Golub and Van Loan [1989]). Many block matrix algorithms supporting the (nonpractical) estimate $O(n^{2.376})$, however, become practically important for parallel computations (see Section 8.2.7).

In the next six sections, we will more closely consider the solution of a linear system of equations, $A\boldsymbol{v} = \boldsymbol{b}$, which is the most frequent operation in practice of scientific and engineering computing and is highly important theoretically. We will partition the known solution methods depending on whether the coefficient matrix A is *dense and unstructured*, **sparse**, or *dense and* **structured**.

8.2.3 Gaussian Elimination Algorithm

The solution of a nonsingular linear system $A\boldsymbol{x} = \boldsymbol{v}$ uses only about n^2 ops if the system is lower (or upper) triangular, that is, if all subdiagonal (or superdiagonal) entries of A vanish. For example (cf. Pan [1992b]), let $n = 3$,

$$\begin{aligned} x_1 + 2x_2 - x_3 &= 3 \\ -2x_2 - 2x_3 &= -10 \\ -6x_3 &= -18 \end{aligned}$$

Compute $x_3 = 3$ from the last equation, substitute into the previous ones, and arrive at a triangular system of $n - 1 = 2$ equations. In $n - 1$ (in our case, 2) such recursive substitution steps, we compute the solution.

The triangular case is itself important; furthermore, every nonsingular linear system is reduced to two triangular ones by means of *forward elimination* of the variables, which essentially amounts to computing the PLU factorization of the input matrix A, that is, to computing two lower triangular matrices L and U^T (where L has unit values on its diagonal) and a permutation matrix P such that $A = PLU$. [A permutation matrix P is filled with zeros and ones and has exactly one nonzero entry in each row and in each column; in particular, this implies that $P^T = P^{-1}$. $P\boldsymbol{u}$ has the same components as $\boldsymbol{u}$ but written in a distinct (fixed) order, for any vector $\boldsymbol{u}$]. As soon as the latter factorization is available, we may compute $\boldsymbol{x} = A^{-1}\boldsymbol{v}$ by solving two triangular systems, that is, at first, $L\boldsymbol{y} = P^T\boldsymbol{v}$, in $\boldsymbol{y}$, and then $U\boldsymbol{x} = \boldsymbol{y}$, in $\boldsymbol{x}$. Computing the factorization (elimination stage) is more costly than the subsequent *back substitution stage*, the latter involving about $2n^2$ ops. The Gaussian classical algorithm for elimination requires about $2n^3/3$ ops, not counting some comparisons, generally required in order to ensure appropriate *pivoting*, also called *elimination ordering*. Pivoting enables us to avoid divisions by small values, which could have caused numerical stability problems. Theoretically, one may employ fast matrix multiplication and compute the matrices P, L, and U in $O(n^{2.376})$ ops [Aho et al. 1974] [and then compute the vectors $\boldsymbol{y}$ and $\boldsymbol{x}$ in $O(n^2)$ ops]. Pivoting can be dropped for some important classes of linear systems, notably, for *positive definite* and for *diagonally dominant* systems [Golub and Van Loan 1989, Pan 1991, 1992b, Bini and Pan 1994].

We refer the reader to Golub and Van Loan [1989, pp. 82–83], or Pan [1992b, p. 794], on sensitivity of the solution to the input and roundoff errors in numerical computing. The output errors grow with the **condition number** of A, represented by $\|A\|\|A^{-1}\|$ for an appropriate matrix norm or by the ratio of maximum and minimum singular values of A. Except for ill-conditioned linear systems $A\boldsymbol{x} = \boldsymbol{v}$, for which the condition number of A is very large, a rough initial approximation to the solution can be rapidly refined (cf. Golub and Van Loan [1989]) via the *iterative improvement algorithm*, as soon as we know P and rough approximations to the matrices L and U of the PLU factorization of A. Then b correct bits of each output value can be computed in $(b + n)n^2$ ops as $b \to \infty$.

8.2.4 Singular Linear Systems of Equations

If the matrix A is **singular** (in particular, if A is rectangular), then the linear system $A\boldsymbol{x} = \boldsymbol{v}$ is either overdetermined, that is, has no solution, or underdetermined, that is, has infinitely many solution vectors. All of them can be represented as $\{\boldsymbol{x}_0 + \boldsymbol{y}\}$, where $\boldsymbol{x}_0$ is a fixed solution vector and $\boldsymbol{y}$ is a vector from the *null space* of A, $\{\boldsymbol{y} : A\boldsymbol{y} = \boldsymbol{0}\}$, that is, $\boldsymbol{y}$ is a solution of the homogeneous linear system $A\boldsymbol{y} = \boldsymbol{0}$. (The null space of an $n \times n$ matrix A is a linear space of the dimension n–rank A.) A vector $\boldsymbol{x}_0$ and a basis for the null-space of A can be computed by using $O(n^{2.376})$ ops if A is an $n \times n$ matrix or by using $O(mn^{1.736})$ ops if A is an $m \times n$ or $n \times m$ matrix and if $m \geq n$ (cf. Bini and Pan [1994]).

For an overdetermined linear system $A\boldsymbol{x} = \boldsymbol{v}$, having no solution, one may compute a vector $\boldsymbol{x}$ minimizing the norm of the residual vector, $\|\boldsymbol{v} - A\boldsymbol{x}\|$. It is most customary to minimize the Euclidean norm,

$$\|\boldsymbol{u}\| = \left(\sum_i |u_i|^2\right)^{1/2}, \quad \boldsymbol{u} = \boldsymbol{v} - A\boldsymbol{x} = (u_i)$$

This defines a least-squares solution, which is relatively easy to compute both practically and theoretically ($O(n^{2.376})$ ops suffice in theory) (cf. Bini and Pan [1994] and Golub and Van Loan [1989]).

8.2.5 Sparse Linear Systems (Including Banded Systems), Direct and Iterative Solution Algorithms

A matrix is sparse if it is filled mostly with zeros, say, if its all nonzero entries lie on 3 or 5 of its diagonals. In many important applications, in particular, solving partial and ordinary differential equations (PDEs and ODEs), one has to solve linear systems whose matrix is sparse and where, moreover, the disposition of its nonzero entries has a certain structure. Then, memory space and computation time can be dramatically decreased (say, from order n^2 to order $n \log n$ words of memory and from n^3 to $n^{3/2}$ or $n \log n$ ops) by using some special data structures and special solution methods. The methods are either direct, that is, are modifications of Gaussian elimination with some special policies of elimination ordering that preserve sparsity during the computation (notably, *Markowitz rule* and *nested dissection* [George and Liu 1981, Gilbert and Tarjan 1987, Lipton et al. 1979, Pan 1993]), or various iterative algorithms. The latter algorithms rely either on computing Krylov sequences [Saad 1995] or on multilevel or multigrid techniques [McCormick 1987, Pan and Reif 1992], specialized for solving linear systems that arise from discretization of PDEs. An important particular class of sparse linear systems is formed by *banded linear systems* with $n \times n$ coefficient matrices $A = (a_{i,j})$ where $a_{i,j} = 0$ if $i - j > g$ or $j - i > h$, for $g + h$ being much less than n. For banded linear systems, the nested dissection methods are known under the name of *block cyclic reduction* methods and are highly effective, but Pan et al. [1995] give some alternative algorithms, too. Some special techniques for computation of Krylov sequences for sparse and other special matrices A can be found in Pan [1995a]; according to these techniques, Krylov sequence is recovered from the solution of the associated linear system $(I - A)\,\boldsymbol{x} = \boldsymbol{v}$, which is solved fast in the case of a special matrix A.

8.2.6 Dense and Structured Matrices and Linear Systems

Many dense $n \times n$ matrices are defined by $O(n)$, say, by less than $2n$, parameters and can be multiplied by a vector by using $O(n \log n)$ or $O(n \log^2 n)$ ops. Such matrices arise in numerous applications (to signal and image processing, coding, algebraic computation, PDEs, integral equations, particle simulation, Markov chains, and many others). An important example is given by $n \times n$ *Toeplitz matrices* $T = (t_{i,j})$, $t_{i,j} = t_{i+1,j+1}$ for $i, j = 0, 1, \ldots, n-1$. Such a matrix can be represented by $2n - 1$ entries of its first row and first column or by $2n - 1$ entries of its first and last columns. The product $T\boldsymbol{v}$ is defined by vector convolution, and its computation uses $O(n \log n)$ ops. Other major examples are given by *Hankel matrices* (obtained by reflecting the row or column sets of Toeplitz matrices), *circulant* (which are a subclass of Toeplitz matrices), and *Bezout*, *Sylvester*, *Vandermonde*, and *Cauchy* matrices. The known solution algorithms for linear systems with such dense structured coefficient matrices use from order $n \log n$ to order $n \log^2 n$ ops. These properties and algorithms are extended via associating some linear operators of displacement and scaling to some more general classes of matrices and linear systems. We refer the reader to Bini and Pan [1994] for many details and further bibliography.

8.2.7 Parallel Matrix Computations

Algorithms for matrix multiplication are particularly suitable for parallel implementation; one may exploit natural association of processors to rows and/or columns of matrices or to their blocks, particularly, in the implementation of matrix multiplication on loosely coupled multiprocessors (cf. Golub and Van Loan [1989] and Quinn [1994]). This motivated particular attention to and rapid progress in devising effective parallel algorithms for block matrix computations. The complexity of parallel computations is usually represented by the computational and communication time and the number of processors involved; decreasing all of these parameters, we face a tradeoff; the product of time and processor bounds (called potential work of parallel algorithms) cannot usually be made substantially smaller than the sequential time bound for the solution. This follows because, according to a variant of *Brent's scheduling principle*, a

single processor can simulate the work of s processors in time $O(s)$. The usual goal of designing a parallel algorithm is in decreasing its parallel time bound (ideally, to a constant, logarithmic or polylogarithmic level, relative to n) and keeping its work bound at the level of the record sequential time bound for the same computational problem (within constant, logarithmic, or at worst polylog factors). This goal has been easily achieved for matrix and vector multiplications, but turned out to be nontrivial for linear system solving, inversion, and some other related computational problems. The recent solution for general matrices [Kaltofen and Pan 1991, 1992] relies on computation of a Krylov sequence and the coefficients of the minimum polynomial of a matrix, by using randomization and auxiliary computations with structured matrices (see the details in Bini and Pan [1994]).

8.2.8 Rational Matrix Computations, Computations in Finite Fields and Semirings

Rational algebraic computations with matrices are performed for a rational input given with no errors, and the computations are also performed with no errors. The precision of computing can be bounded by reducing the computations modulo one or several fixed primes or prime powers. At the end, the exact output values $z = p/q$ are recovered from $z \bmod M$ (if M is sufficiently large relative to p and q) by using the continued fraction approximation algorithm, which is the Euclidean algorithm applied to integers (cf. Pan [1991, 1992a], and Bini and Pan [1994, Section 3 of Chap. 3]). If the output z is known to be an integer lying between $-m$ and m and if $M > 2m$, then z is recovered from $z \bmod M$ as follows:

$$z = \begin{cases} z \bmod M & \text{if } z \bmod M < m \\ -M + z \bmod M & \text{otherwise} \end{cases}$$

The reduction modulo a prime p may turn a nonsingular matrix A and a nonsingular linear system $A\boldsymbol{x} = \boldsymbol{v}$ into singular ones, but this is proved to occur only with a low probability for a random choice of the prime p in a fixed sufficiently large interval (see Bini and Pan [1994, Section 3 of Chap. 4]). To compute the output values z modulo M for a large M, one may first compute them modulo several relatively prime integers $m_1, m_2, \ldots, m_k$ having no common divisors and such that $m_1, m_2, \ldots, m_k > M$ and then easily recover $z \bmod M$ by means of the Chinese remainder algorithm. For matrix and polynomial computations, there is an effective alternative technique of *p-adic* (*Newton–Hensel*) *lifting* (cf. Bini and Pan [1994, Section 3 of Chap. 3]), which is particularly powerful for computations with dense structured matrices, since it preserves the structure of a matrix. We refer the reader to Bareiss [1968] and Geddes et al. [1992] for some special techniques, which enable one to control the growth of all intermediate values computed in the process of performing rational Gaussian elimination, with no roundoff and no reduction modulo an integer.

Gondran and Minoux [1984] and Pan [1993] describe some applications of matrix computations on semirings (with no divisions and subtractions allowed) to graph and combinatorial computations.

8.2.9 Matrix Eigenvalues and Singular Values Problems

The matrix eigenvalue problem is one of the major problems of matrix computation: given an $n \times n$ matrix A, one seeks a $k \times k$ diagonal matrix Λ and an $n \times k$ matrix V of full rank k such that

$$AV = \Lambda V \tag{8.1}$$

The diagonal entries of Λ are called the *eigenvalues* of A; the entry (i, i) of Λ is associated with the ith column of V, called an *eigenvector* of A. The eigenvalues of an $n \times n$ matrix A coincide with the zeros of the characteristic polynomial

$$c_A(x) = \det(xI - A)$$

If this polynomial has n distinct zeros, then $k = n$, and V of Equation 8.1 is a nonsingular $n \times n$ matrix. The matrix $A = I + Z$, where $Z = (z_{i,j})$, $z_{i,j} = 0$ unless $j = i + 1$, $z_{i,i+1} = 1$, is an example of a matrix for which $k = 1$, so that the matrix V degenerates to a vector.

In principle, one may compute the coefficients of $c_A(x)$, the characteristic polynomial of A, and then approximate its zeros (see Section 8.3) in order to approximate the eigenvalues of A. Given the eigenvalues, the corresponding eigenvectors can be recovered by means of the inverse power iteration [Golub and Van Loan 1989, Wilkinson 1965]. Practically, the computation of the eigenvalues via the computation of the coefficients of $c_A(x)$ is not recommended, due to arising numerical stability problems [Wilkinson 1965], and most frequently, the eigenvalues and eigenvectors of a general (unsymmetric) matrix are approximated by means of the QR *algorithm* [Wilkinson 1965, Watkins 1982, Golub and Van Loan 1989]. Before application of this algorithm, the matrix A is simplified by transforming it into the more special (*Hessenberg*) *form* H, by a *similarity transformation*,

$$H = UAU^H \tag{8.2}$$

where $U = (u_{i,j})$ is a unitary matrix, where $U^H U = I$, where $U^H = (\overline{u}_{j,i})$ is the Hermitian transpose of U, with $\overline{z}$ denoting the complex conjugate of z; $U^H = U^T$ if U is a real matrix [Golub and Van Loan 1989]. Similarity transformation into Hessenberg form is one of examples of *rational transformations* of a matrix into special *canonical forms*, of which transformations into *Smith* and *Hermite forms* are two other most important representatives [Kaltofen et al. 1990, Geddes et al. 1992, Giesbrecht 1995].

In practice, the eigenvalue problem is very frequently symmetric, that is, arises for a real symmetric matrix A, for which

$$A^T = (a_{j,i}) = A = (a_{i,j})$$

or for complex Hermitian matrices A, for which

$$A^H = (\overline{a}_{j,i}) = A = (a_{i,j})$$

For real symmetric or Hermitian matrices A, the eigenvalue problem (called symmetric) is treated much more easily than in the unsymmetric case. In particular, in the symmetric case, we have $k = n$, that is, the matrix V of Equation 8.1 is a nonsingular $n \times n$ matrix, and moreover, all of the eigenvalues of A are real and little sensitive to small input perturbations of A (according to the Courant–Fisher minimization criterion [Parlett 1980, Golub and Van Loan 1989]).

Furthermore, similarity transformation of A to the Hessenberg form gives much stronger results in the symmetric case: the original problem is reduced to one for a symmetric tridiagonal matrix H of Equation 8.2 (this can be achieved via the Lanczos algorithm, cf. Golub and Van Loan [1989] or Bini and Pan [1994, Section 3 of Chap. 2]). For such a matrix H, application of the QR algorithm is dramatically simplified; moreover, two competitive algorithms are also widely used, that is, the *bisection* [Parlett 1980] (a slightly slower but very robust algorithm) and the *divide-and-conquer* method [Cuppen 1981, Golub and Van Loan 1989]. The latter method has a modification [Bini and Pan 1991] that only uses $O(n \log^2 n(\log n + \log^2 b))$ arithmetic operations in order to compute all of the eigenvalues of an $n \times n$ symmetric tridiagonal matrix A within the output error bound $2^{-b}\|A\|$, where $\|A\| \le n \max |a_{i,j}|$.

The eigenvalue problem has a generalization, where generalized eigenvalues and eigenvectors for a pair A, B of matrices are sought, such that

$$AV = B\Lambda V$$

(the solution algorithm should proceed without computing the matrix $B^{-1}A$, so as to avoid numerical stability problems).

In another highly important extension of the symmetric eigenvalue problem, one seeks a singular value decomposition (SVD) of a (generally unsymmetric and, possibly, rectangular) matrix A: $A = U\Sigma V^T$, where U and V are unitary matrices, $U^H U = V^H V = I$, and Σ is a diagonal (generally rectangular)

matrix, filled with zeros, except for its diagonal, filled with (positive) singular values of A and possibly, with zeros. The SVD is widely used in the study of numerical stability of matrix computations and in numerical treatment of singular and ill-conditioned (close to singular) matrices. An alternative tool is orthogonal (QR) factorization of a matrix, which is not as refined as SVD but is a little easier to compute [Golub and Van Loan 1989]. The squares of the singular values of A equal the eigenvalues of the Hermitian (or real symmetric) matrix $A^H A$, and the SVD of A can be also easily recovered from the eigenvalue decomposition of the Hermitian matrix

$$\begin{bmatrix} 0 & A^H \\ A & 0 \end{bmatrix}$$

but more popular are some effective direct methods for the computation of the SVD [Golub and Van Loan 1989].

8.2.10 Approximating Polynomial Zeros

Solution of an nth degree polynomial equation,

$$p(x) = \sum_{i=0}^{n} p_i\, x^i = 0, \quad p_n \neq 0$$

(where one may assume that $p_{n-1} = 0$; this can be ensured via shifting the variable x) is a classical problem that has greatly influenced the development of mathematics throughout the centuries [Pan 1995b]. The problem remains highly important for the theory and practice of present day computing, and dozens of new algorithms for its approximate solution appear every year. Among the existent implementations of such algorithms, the practical heuristic champions in efficiency (in terms of computer time and memory space used, according to the results of many experiments) are various modifications of *Newton's iteration*, $z(i+1) = z(i) - a(i)p(z(i))/p'(z(i))$, $a(i)$ being the step-size parameter [Madsen 1973], *Laguerre's method* [Hansen et al. 1977, Foster 1981], and the randomized *Jenkins–Traub algorithm* [1970] [all three for approximating a single zero z of $p(x)$], which can be extended to approximating other zeros by means of deflation of the input polynomial via its numerical division by $x - z$. For simultaneous approximation of all of the zeros of $p(x)$ one may apply the Durand–Kerner algorithm, which is defined by the following recurrence:

$$z_j(i+1) = \frac{z_j(i) - p((z_j(i))}{z_j(i) - z_k(i)}, \quad j = 1, \ldots, n, \quad i = 1, 2, \ldots \tag{8.3}$$

Here, the customary choice for the n initial approximations $z_j(0)$ to the n zeros of

$$p(x) = p_n \prod_{j=1}^{n} (x - z_j)$$

is given by $z_j(0) = Z \exp(2\pi\sqrt{-1}/n)$, $j = 1, \ldots, n$, with Z exceeding (by some fixed factor $t > 1$) $\max_j |z_j|$; for instance, one may set

$$Z = 2t \max_{i<n}(p_i/p_n) \tag{8.4}$$

For a fixed i and for all j, the computation according to Equation 8.3 is simple, only involving order n^2 ops, and according to the results of many experiments, the iteration Equation 8.3 rapidly converges to the solution, though no theory confirms or explains these results. Similar is the situation with various

modifications of this algorithm, which are now even more popular than the original algorithms and many of which are listed in Pan [1992a, 1992b] (also cf. Bini and Pan [1996] and McNamee [1993]).

On the other hand, there are two groups of algorithms that, when implemented, promise to be competitive or even substantially superior to Newton's and Laguerre's iteration, the algorithm by Jenkins and Traub, and all of the algorithms of the Durand–Kerner type. One such group is given by the modern modifications and improvements (due to Pan [1987, 1994a, 1994b] and Renegar [1989]) of *Weyl's quadtree construction* of 1924. In this approach, an initial square S, containing all the zeros of $p(x)$ [say, $S = \{x, |Im\ x| < Z, |Re\ x| < Z\}$ for Z of Eq. (8.4)], is recursively partitioned into four congruent subsquares. In the center of each of them, a proximity test is applied that estimates the distance from this center to the closest zero of $p(x)$. If such a distance exceeds one-half of the diagonal length, then the subsquare contains no zeros of $p(x)$ and is discarded. When this process ensures a strong isolation from each other for the components formed by the remaining squares, then certain extensions of Newton's iteration [Renegar 1989, Pan 1994a, 1994b], or some iterative techniques based on numerical integration [Pan 1987] are applied and very rapidly converge to the desired approximations to the zeros of $p(x)$, within the error bound $2^{-b}Z$ for Z of Equation 8.4. As a result, the algorithms of Pan [1987, 1994a, 1994b] solve the entire problem of approximating (within $2^{-b}Z$) all of the zeros of $p(x)$ at the overall cost of performing $O((n^2 \log n) \log(bn))$ ops (cf. Bini and Pan [1996]), versus order n^2 operations at each iteration of Durand–Kerner type.

The second group is given by the divide-and-conquer algorithms. They first compute a sufficiently wide annulus A, which is free of the zeros of $p(x)$ and contains comparable numbers of such zeros (that is, the same numbers up to a fixed constant factor) in its exterior and its interior. Then the two factors of $p(x)$ are numerically computed, that is, $F(x)$ having all its zeros in the interior of the annulus, and $G(x) = p(x)/F(x)$ having no zeros there. The same process is recursively repeated for $F(x)$ and $G(x)$ until factorization of $p(x)$ into the product of linear factors is computed numerically. From this factorization, approximations to all of the zeros of $p(x)$ are obtained. The algorithms of Pan [1995a, 1996] based on this approach only require $O(n \log(bn)\ (\log n)^2\ )$ ops in order to approximate all of the n zeros of $p(x)$ within $2^{-b}Z$ for Z of Eq. (8.4). (Note that this is a quite sharp bound: at least n ops are necessary in order to output n distinct values.)

The computations for the polynomial zero problem are ill conditioned, that is, they generally require a high precision for the worst-case input polynomials in order to ensure a required output precision, no matter which algorithm is applied for the solution. Consider, for instance, the polynomial $(x - \frac{6}{7})^n$ and perturb its x-free coefficient by 2^{-bn}. Observe the resulting jumps of the zero $x = 6/7$ by 2^{-b}, and observe similar jumps if the coefficients p_i are perturbed by $2^{(i-n)b}$ for $i = 1, 2, \ldots, n-1$. Therefore, to ensure the output precision of b bits, we need an input precision of at least $(n - i)b$ bits for each coefficient $p_i, i = 0, 1, \ldots, n-1$. Consequently, for the worst-case input polynomial $p(x)$, any solution algorithm needs at least about a factor n increase of the precision of the input and of computing versus the output precision.

Numerically unstable algorithms may require even a higher input and computation precision, but inspection shows that this is not the case for the algorithms of Pan [1987, 1994a, 1994b, 1995a, 1996] and Renegar [1989] (cf. Bini and Pan [1996]).

8.2.11 Fast Fourier Transform and Fast Polynomial Arithmetic

To yield the record complexity bounds for approximating polynomial zeros, one should exploit fast algorithms for basic operations with polynomials (their multiplication, division, and transformation under the shift of the variable), as well as FFT, both directly and for supporting the fast polynomial arithmetic. The FFT and fast basic polynomial algorithms (including those for multipoint polynomial evaluation and interpolation) are the basis for many other fast polynomial computations, performed both numerically and symbolically (compare the next sections). These basic algorithms, their impact on the field of algebraic computation, and their complexity estimates have been extensively studied in Aho et al. [1974], Borodin and Munro [1975], and Bini and Pan [1994].

8.3 Systems of Nonlinear Equations and Other Applications

Given a system $\{p_1(x_1,\ldots,x_n), p_2(x_1,\ldots,x_n),\ldots, p_r(x_1,\ldots,x_n)\}$ of nonlinear polynomials with rational coefficients [each $p_i(x_1,\ldots,x_n)$ is said to be an element of $\mathbb{Q}[x_1,\ldots,x_n]$, the ring of polynomials in $x_1,\ldots,x_n$ over the field $\mathbb{Q}$ of rational numbers], the n-tuple of complex numbers $(a_1,\ldots,a_n)$ is a common solution of the system, if $f_i(a_1,\ldots,a_n) = 0$ for each i with $1 \leq i \leq r$. In this section, we explore the problem of exactly solving a system of nonlinear equations over the field $\mathbb{Q}$. We provide an overview and cite references to different symbolic techniques used for solving systems of algebraic (polynomial) equations. In particular, we describe methods involving *resultant* and *Gröbner basis* computations.

The *Sylvester resultant method* is the technique most frequently utilized for determining a common zero of two polynomial equations in one variable [Knuth 1981]. However, using the Sylvester method successively to solve a system of multivariate polynomials proves to be inefficient. Successive resultant techniques, in general, lack efficiency as a result of their sensitivity to the ordering of the variables [Kapur and Lakshman 1992]. It is more efficient to eliminate all variables together from a set of polynomials, thus leading to the notion of the *multivariate resultant*. The three most commonly used multivariate resultant formulations are the *Dixon* [Dixon 1908, Kapur and Saxena 1995], *Macaulay* [Macaulay 1916, Canny 1990, Kaltofen and Lakshman 1988], and *sparse resultant formulations* [Canny and Emiris 1993a, Sturmfels 1991].

The theory of Gröbner bases provides powerful tools for performing computations in multivariate polynomial rings. Formulating the problem of solving systems of polynomial equations in terms of polynomial ideals, we will see that a Gröbner basis can be computed from the input polynomial set, thus allowing for a form of back substitution (cf. Section 8.2) in order to compute the common roots.

Although not discussed, it should be noted that the *characteristic set algorithm* can be utilized for polynomial system solving. Ritt [1950] introduced the concept of a characteristic set as a tool for studying solutions of algebraic differential equations. Wu [1984, 1986], in search of an effective method for automatic theorem proving, converted Ritt's method to ordinary polynomial rings. Given the before mentioned system P, the characteristic set algorithm transforms P into a triangular form, such that the set of common zeros of P is equivalent to the set of roots of the triangular system [Kapur and Lakshman 1992].

Throughout this exposition we will also see that these techniques used to solve nonlinear equations can be applied to other problems as well, such as computer-aided design and automatic geometric theorem proving.

8.3.1 Resultant Methods

The question of whether two polynomials $f(x), g(x) \in \mathbb{Q}[x]$,

$$f(x) = f_n x^n + f_{n-1}x^{n-1} + \cdots + f_1 x + f_0$$
$$g(x) = g_m x^m + g_{m-1}x^{m-1} + \cdots + g_1 x + g_0$$

have a common root leads to a condition that has to be satisfied by the coefficients of both f and g. Using a derivation of this condition due to Euler, the *Sylvester matrix* of f and g (which is of order $m + n$) can be formulated. The vanishing of the determinant of the Sylvester matrix, known as the *Sylvester resultant*, is a necessary and sufficient condition for f and g to have common roots [Knuth 1981].

As a running example let us consider the following system in two variables provided by Lazard [1981]:

$$f = x^2 + xy + 2x + y - 1 = 0$$
$$g = x^2 + 3x - y^2 + 2y - 1 = 0$$

The Sylvester resultant can be used as a tool for eliminating several variables from a set of equations [Kapur and Lakshman 1992]. Without loss of generality, the roots of the Sylvester resultant of f and g treated as polynomials in y, whose coefficients are polynomials in x, are the x-coordinates of the common zeros of

f and g. More specifically, the Sylvester resultant of the Lazard system with respect to y is given by the following determinant:

$$\det\left(\begin{bmatrix} x+1 & x^2+2x-1 & 0 \\ 0 & x+1 & x^2+2x-1 \\ -1 & 2 & x^2+3x-1 \end{bmatrix}\right) = -x^3 - 2x^2 + 3x$$

The roots of the Sylvester resultant of f and g are $\{-3, 0, 1\}$. For each x value, one can substitute the x value back into the original polynomials yielding the solutions $(-3, 1), (0, 1), (1, -1)$.

The method just outlined can be extended recursively, using *polynomial GCD computations*, to a larger set of multivariate polynomials in $\mathbb{Q}[x_1, \ldots, x_n]$. This technique, however, is impractical for eliminating many variables, due to an explosive growth of the degrees of the polynomials generated in each elimination step.

The Sylvester formulations have led to a *subresultant theory*, developed simultaneously by G. E. Collins and W. S. Brown and J. Traub. The subresultant theory produced an efficient algorithm for computing polynomial GCDs and their resultants, while controlling intermediate expression swell [Brown 1971, Brown and Traub 1971, Collins 1967, 1971, Knuth 1981].

It should be noted that by adopting an implicit representation for symbolic objects, the intermediate expression swell introduced in many symbolic computations can be palliated. Recently, polynomial GCD algorithms have been developed that use implicit representations and thus avoid the computationally costly content and primitive part computations needed in those GCD algorithms for polynomials in explicit representation [Díaz and Kaltofen 1995, Kaltofen 1988, Kaltofen and Trager 1990].

The solvability of a set of nonlinear multivariate polynomials over the field $\mathbb{Q}$ can be determined by the vanishing of a generalization of the Sylvester resultant of two polynomials in a single variable.

Due to the special structure of the Sylvester matrix, Bézout developed a method for computing the resultant as a determinant of order $\max(m, n)$ during the 18th century. Cayley [1865] reformulated Bézout's method leading to Dixon's [1908] extension to the bivariate case. Dixon's method can be generalized to a set

$$\{p_1(x_1, \ldots, x_n), p_2(x_1, \ldots, x_n), \ldots, p_{n+1}(x_1, \ldots, x_n)\}$$

of $n + 1$ generic n-degree polynomials in n variables [Kapur et al. 1994]. The vanishing of the Dixon resultant is a necessary and sufficient condition for the polynomials to have a nontrivial projective common zero, and also a necessary condition for the existence of an affine common zero. The Dixon formulation gives the resultant up to a multiple, and hence in the affine case it may happen that the vanishing of the Dixon resultant does not necessarily indicate that the equations in question have a common root. A nontrivial multiple, known as the *projection operator*, can be extracted via a method based on so-called *rank subdeterminant computation* (RSC) [Kapur et al. 1994]. It should be noted that the RSC method can also be applied to the Macaulay and sparse resultant formulations as is detailed here.

In 1916, Macaulay constructed a resultant for n homogeneous polynomials in n variables, which simultaneously generalizes the Sylvester resultant and the determinant of a system of linear equations [Canny et al. 1989, Kapur and Lakshman 1992]. Like the Dixon formulation, the Macaulay resultant is a multiple of the resultant (except in the case of generic homogeneous polynomials, where it produces the exact resultant). For the Macaulay formulation, Canny [1990] has invented a general method that perturbs any polynomial system and extracts a nontrivial projection operator.

Using recent results pertaining to sparse polynomial systems [Gelfand et al. 1994, Sturmfels 1991, Sturmfels and Zelevinsky 1992], the mixed sparse resultant of a system of $n + 1$ sparse polynomials in n variables in its matrix form was given by Canny and Emiris [1993a] and consequently improved in Canny and Emiris [1993b, 1994]. Here, sparsity denotes that only certain monomials in each of the $n + 1$ polynomials have nonzero coefficients. The determinant of the sparse resultant matrix, such as the Macaulay and Dixon matrices, only yields a projection operation, not the exact resultant.

Suppose we are asked to find the common zeros of a set of n polynomials in n variables $\{p_1(x_1, \ldots, x_n), p_2(x_1, \ldots, x_n), \ldots, p_n(x_1, \ldots, x_n)\}$. By augmenting the polynomial set by a generic linear form [Canny 1990, Canny and Manocha 1991, Kapur and Lakshman 1992], one can construct the *u-resultant* of a given system of polynomials. The u-resultant factors into linear factors over the complex numbers, providing

the common zeros of the given polynomials equations. The u-resultant method takes advantage of the properties of the multivariate resultant, and hence can be constructed using either Dixon's, Macaulay's, or sparse formulations.

Consider the previous example augmented by a generic linear form

$$f_1 = x^2 + xy + 2x + y - 1 = 0$$
$$f_2 = x^2 + 3x - y^2 + 2y - 1 = 0$$
$$f_1 = ux + vy + w = 0$$

As described in Canny et al. [1989], the following matrix M corresponds to the Macaulay u-resultant of the preceding system of polynomials, with z being the homogenizing variable:

$$M = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & u & 0 & 0 & 0 \\ 2 & 0 & 1 & 3 & 0 & 1 & 0 & u & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & v & 0 & 0 & 0 \\ 1 & 2 & 1 & 2 & 3 & 0 & w & v & u & 0 \\ -1 & 0 & 2 & -1 & 0 & 3 & 0 & w & 0 & u \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 & -1 & 0 & 0 & v & 0 \\ 0 & -1 & 1 & 0 & -1 & 2 & 0 & 0 & w & v \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & w \end{bmatrix}$$

It should be noted that

$$\det(M) = (u - v + w)(-3u + v + w)(v + w)(u - v)$$

corresponds to the affine solutions $(1, -1)$, $(-3, 1)$, $(0, 1)$, and one solution at infinity. An empirical comparison of the detailed resultant formulations can be found in Kapur and Saxena [1995]. Recently, the multivariate resultant formulations are being used for other applications such as *algebraic and geometric reasoning* [Kapur et al. 1994], *computer-aided design* [Stederberg and Goldman 1986], and for *implicitization and finding base points* [Chionh 1990].

8.3.2 Gröbner Bases

Solving systems of nonlinear equations can be formulated in terms of polynomial ideals [Becker and Weispfenning 1993, Geddes et al. 1992, Winkler 1996]. Let us first establish some terminology.

The ideal generated by a system of polynomial equations $p_1, \ldots, p_r$ over $\mathbb{Q}[x_1, \ldots, x_n]$ is the set of all linear combinations

$$(p_1, \ldots, p_r) = \{h_1 p_1 + \cdots + h_r p_r \mid h_1, \ldots, h_r \in \mathbb{Q}[x_1, \ldots, x_n]\}$$

The algebraic variety of $p_1, \ldots, p_r \in \mathbb{Q}[x_1, \ldots, x_n]$ is the set of their common zeros,

$$V(p_1, \ldots, p_r) = \{(a_1, \ldots, a_n) \in \mathbb{C}^n \mid f_1(a_1, \ldots, a_n) = \cdots = f_r(a_1, \ldots, a_n) = 0\}$$

A version of the *Hilbert Nullstellensatz* states that

$$V(p_1, \ldots, p_r) = \text{the empty set } \emptyset \quad \Longleftrightarrow \quad 1 \in (p_1, \ldots, p_r) \text{ over } \mathbb{Q}[x_1, \ldots, x_n]$$

which relates the solvability of polynomial systems to the ideal membership problem.

A term $t = x_1^{e_1} x_2^{e_2} \ldots x_n^{e_n}$ of a polynomial is a product of powers with $\deg(t) = e_1 + e_2 + \cdots + e_n$. In order to add needed structure to the polynomial ring we will require that the terms in a polynomial be ordered in an admissible fashion [Geddes et al. 1992, Kapur and Lakshman 1992]. Two of the most common admissible orderings are the **lexicographic order** ($\prec_l$), where terms are ordered as in a dictionary, and the **degree order**

($\prec_d$), where terms are first compared by their degrees with equal degree terms compared lexicographically. A variation to the lexicographic order is the *reverse lexicographic order*, where the lexicographic order is reversed [Davenport et al. 1988, p. 96].

It is this previously mentioned structure that permits a type of simplification known as polynomial reduction. Much like a polynomial remainder process, the process of polynomial reduction involves subtracting a multiple of one polynomial from another to obtain a smaller degree result [Becker and Weispfenning 1993, Geddes et al. 1992, Kapur and Lakshman 1992, Winkler 1996].

A polynomial g is said to be reducible with respect to a set $P = \{p_1, \ldots, p_r\}$ of polynomials if it can be reduced by one or more polynomials in P. When g is no longer reducible by the polynomials in P, we say that g is *reduced* or is *a normal form* with respect to P.

For an arbitrary set of basis polynomials, it is possible that different reduction sequences applied to a given polynomial g could reduce to different normal forms. A basis $G \subseteq \mathbb{Q}[x_1, \ldots, x_n]$ is a *Gröbner basis* if and only if every polynomial in $\mathbb{Q}[x_1, \ldots, x_n]$ has a unique normal form with respect to G. Buchberger [1965, 1976, 1983, 1985] showed that every basis for an ideal $(p_1, \ldots, p_r)$ in $\mathbb{Q}[x_1, \ldots, x_n]$ can be converted into a Gröbner basis $\{p_1^*, \ldots, p_s^*\} = GB(p_1, \ldots, p_r)$, concomitantly designing an algorithm that transforms an arbitrary ideal basis into a Gröbner basis. Another characteristic of Gröbner bases is that by using the previously mentioned reduction process we have

$$g \in (p_1, \ldots, p_r) \quad \Longleftrightarrow \quad (g \bmod p_1^*, \ldots, p_s^*) = 0$$

Further, by using the Nullstellensatz it can be shown that $p_1, \ldots, p_r$ viewed as a system of algebraic equations is solvable if and only if $1 \notin GB(p_1, \ldots, p_r)$.

Depending on which admissible term ordering is used in the Gröbner bases construction, an ideal can have different Gröbner bases. However, an ideal cannot have different (reduced) Gröbner bases for the same term ordering.

Any system of polynomial equations can be solved using a lexicographic Gröbner basis for the ideal generated by the given polynomials. It has been observed, however, that Gröbner bases, more specifically lexicographic Gröbner bases, are hard to compute [Becker and Weispfenning 1993, Geddes et al. 1992, Lakshman 1990, Winkler 1996]. In the case of zero-dimensional ideals, those whose varieties have only isolated points, Faugère, et al. [1993] outlined a change of basis algorithm which can be utilized for solving zero-dimensional systems of equations. In the zero-dimensional case, one computes a Gröbner basis for the ideal generated by a system of polynomials under a degree ordering. The so-called *change of basis algorithm* can then be applied to the degree ordered Gröbner basis to obtain a Gröbner basis under a lexicographic ordering.

Turning to Lazard's example in the form of a polynomial basis,

$$f_1 = x^2 + xy + 2x + y - 1$$
$$f_2 = x^2 + 3x - y^2 + 2y - 1$$

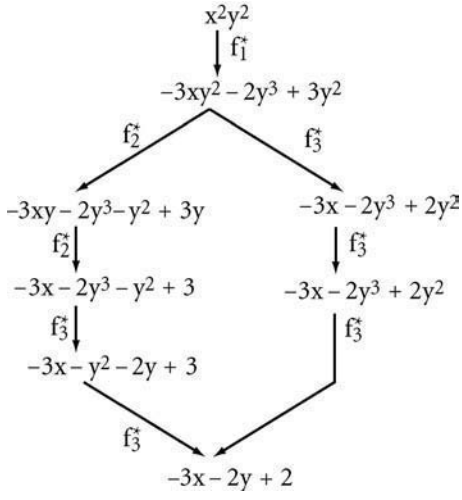
one obtains (under lexicographical ordering with $x \prec_l y$) a Gröbner basis in which the variables are triangularized such that the finitely many solutions can be computed via back substitution:

$$f_1^* = x^2 + 3x + 2y - 2$$
$$f_2^* = xy - x - y + 1$$
$$f_3^* = y^2 - 1$$

It should be noted that the final univariate polynomial is of minimal degree and the polynomials used in the back substitution will have degree no larger than the number of roots.

As an example of the process of polynomial reduction with respect to a Gröbner basis, the following demonstrates two possible reduction sequences to the same normal form. The polynomial x^2y^2 is reduced

with respect to the previously computed Gröbner basis $\{f_1^*, f_2^*, f_3^*\} = GB(f_1, f_2)$ along the following two distinct reduction paths, both yielding $-3x - 2y + 2$ as the normal form.



There is a strong connection between lexicographic Gröbner bases and the previously mentioned resultant techniques. For some types of input polynomials, the computation of a reduced system via resultants might be much faster than the computation of a lexicographic Gröbner basis. A good comparison between the Gröbner computations and the different resultant formulations can be found in Kapur and Saxena [1995].

In a survey article, Buchberger [1985] detailed how Gröbner bases can be used as a tool for many polynomial ideal theoretic operations. Other applications of Gröbner basis computations include automatic geometric theorem proving [Kapur 1986, Wu 1984, 1986], multivariate polynomial factorization and GCD computations [Gianni and Trager 1985], and polynomial interpolation [Lakshman and Saunders 1994, 1995].

8.4 Polynomial Factorization

The problem of factoring polynomials is a fundamental task in symbolic algebra. An example in one's early mathematical education is the factorization $x^2 - y^2 = (x + y) \cdot (x - y)$, which in algebraic terms is a factorization of a polynomial in two variables with integer coefficients. Technology has advanced to a state where most polynomial factorization problems are doable on a computer, in particular, with any of the popular mathematical software, such as the Mathematica or Maple systems. For instance, the factorization of the determinant of a 6×6 symmetric Toeplitz matrix over the integers is computed in Maple as

```
> readlib(showtime) :
> showtime() :
O1 := T := linalg[toeplitz]([a, b, c, d, e, f]);
```

$$T := \begin{bmatrix} a & b & c & d & e & f \\ b & a & b & c & d & e \\ c & b & a & b & c & d \\ d & c & b & a & b & c \\ e & d & c & b & a & b \\ f & e & d & c & b & a \end{bmatrix}$$

```
time 0.03 words7701
O2 := factor(linalg[det](T));
```

$$
\begin{aligned}
&-(2dca - 2bce + 2c^2a - a^3 - da^2 + 2d^2c + d^2a + b^3 + 2abc - 2c^2b \\
&\quad + d^3 + 2ab^2 - 2dcb - 2cb^2 - 2ec^2 + 2eb^2 + 2fcb + 2bae \\
&\quad + b^2f + c^2f + be^2 - ba^2 - fdb - fda - fa^2 - fba + e^2a - 2db^2 \\
&\quad + dc^2 - 2deb - 2dec - dba)(2dca - 2bce - 2c^2a + a^3 \\
&\quad - da^2 - 2d^2c - d^2a + b^3 + 2abc - 2c^2b + d^3 - 2ab^2 + 2dcb \\
&\quad + 2cb^2 + 2ec^2 - 2eb^2 - 2fcb + 2bae + b^2f + c^2f + be^2 - ba^2 \\
&\quad - fdb + fda - fa^2 + fba - e^2a - 2db^2 + dc^2 + 2deb - 2dec \\
&\quad + dba)
\end{aligned}
$$

```
time 27.30 words 857700
```

Clearly, the Toeplitz determinant factorization requires more than tricks from high school algebra. Indeed, the development of modern algorithms for the polynomial factorization problem is one of the great successes of the discipline of symbolic mathematical computation. Kaltofen [1982, 1990, 1992] has surveyed the algorithms until 1992, mostly from a computer science perspective. In this chapter we shall focus on the applications of the known fast methods to problems in science and engineering. For a more extensive set of references, please refer to Kaltofen's survey articles.

8.4.1 Polynomials in a Single Variable over a Finite Field

At first glance, the problem of factoring an integer polynomial modulo a prime number appears to be very similar to the problem of factoring an integer represented in a prime radix. That is simply not so. The factorization of the polynomial $x^{511} - 1$ can be done modulo 2 on a computer in a matter of milliseconds, whereas the factorization of the integer $2^{511} - 1$ into its integer factors is a computational challenge. For those interested: the largest prime factors of $2^{511} - 1$ have 57 and 67 decimals digits, respectively, which makes a tough but not undoable 123 digit product for the number field sieve factorizer [Leyland 1995]. Irreducible factors of polynomials modulo 2 are needed to construct finite fields. For example, the factor $x^9 + x^4 + 1$ of $x^{511} - 1$ leads to a model of the finite field with 2^9 elements, GF(2^9), by simply computing with the polynomial remainders modulo $x^9 + x^4 + 1$ as the elements. Such irreducible polynomials are used for setting up error-correcting codes, such as the BCH codes [MacWilliams and Sloan 1977]. Berlekamp's [1967, 1970] pioneering work on factoring polynomials over a finite field by linear algebra is done with this motivation. The linear algebra tools that Berlekamp used seem to have been introduced to the subject as early as in 1937 by Petr (cf. Št. Schwarz [1956]).

Today, factoring algorithms for univariate polynomials over finite fields form the innermost subalgorithm to lifting-based algorithms for factoring polynomials in one [Zassenhaus 1969] and many [Musser 1975] variables over the integers. When Maple computed the factorization of the previous Toeplitz determinant, it began with factoring a univariate polynomial modulo a prime integer. The case when the prime integer is very large has led to a significant development in computer science itself. As it turns out, by selecting random residues the expected performance of the algorithms can be speeded up exponentially [Berlekamp 1970, Rabin 1980]. Randomization is now an important tool for designing efficient algorithms and has proliferated to many fields of computer science. Paradoxically, the random elements are produced by a congruential random number generator, and the actual computer implementations are quite deterministic, which leads some computer scientists to believe that random bits can be eliminated in general at no exponential slow down. Nonetheless, for the polynomial factoring problem modulo a large prime, no fast methods are known to date that would work without this *probabilistic* approach.

One can measure the computing time of selected algorithms in terms of n, the degree of the input polynomial, and p, the cardinality of the field. When counting arithmetic operations modulo p (including reciprocals), the best known algorithms are quite recent. Berlekamp's 1970 method performs

$O(n^{\omega} + n^{1+o(1)} \log p)$ residue operations. Here and subsequently, ω denotes the exponent implied by the used linear system solver, i.e., $\omega = 3$ when classical methods are used, and $\omega = 2.376$ when asymptotically fast (though impractical) matrix multiplication is assumed. The correction term $o(1)$ accounts for the $\log n$ factors derived from the FFT-based fast polynomial multiplication and remaindering algorithms. An approach in the spirit of Berlekamp's but possibly more practical for $p = 2$ has recently been discovered by Niederreiter [1994]. A very different technique by Cantor and Zassenhaus [1981] first separates factors of different degrees and then splits the resulting polynomials of equal degree factors. It has $O(n^{2+o(1)} \log p)$ complexity and is the basis for the following two methods. Algorithms by von zur Gathen and Shoup [1992] have running time $O(n^{2+o(1)} + n^{1+o(1)} \log p)$ and those by Kaltofen and Shoup [1995] have running time $O(n^{1.815} \log p)$, the latter with fast matrix multiplication.

For n and p simultaneously large, a variant of the method by Kaltofen and Shoup [1995] that uses classical linear algebra and runs in $O(n^{2.5} + n^{1+o(1)} \log p)$ residue operations is the current champion among the practical algorithms. With it Shoup [1996], using his own fast polynomial arithmetic package, has factored a randomlike polynomial of degree 2048 modulo a 2048-bit prime number in about 12 days on a Sparc-10 computer using 68 megabyte of main memory. For even larger n, but smaller p, parallelization helps, and Kaltofen and Lobo [1994] could factor a polynomial of degree $n = 15\,001$ modulo $p = 127$ in about 6 days on 8 computers that are rated at 86.1 MIPS. At the time of this writing, the largest polynomial factored modulo 2 is $X^{216\,091} + X + 1$; this was accomplished by Peter Montgomery in 1991 by using Cantor's fast polynomial multiplication algorithm based on additive transforms [Cantor 1989].

8.4.2 Polynomials in a Single Variable over Fields of Characteristic Zero

As mentioned before, generally usable methods for factoring univariate polynomials over the rational numbers begin with the Hensel lifting techniques introduced by Zassenhaus [1969]. The input polynomial is first factored modulo a suitable prime integer p, and then the factorization is lifted to one modulo p^k for an exponent k of sufficient size to accommodate all possible integer coefficients that any factors of the polynomial might have. The lifting approach is fast in practice, but there are hard-to-factor polynomials on which it runs an exponential time in the degree of the input. This slowdown is due to so-called parasitic modular factors. The polynomial $x^4 + 1$, for example, factors modulo all prime integers but is irreducible over the integers: it is the cyclotomic equation for eighth roots of unity. The products of all subsets of modular factors are candidates for integer factors, and irreducible integer polynomials with exponentially many such subsets exist [Kaltofen et al. 1983].

The elimination of the exponential bottleneck by giving a polynomial-time solution to the integer polynomial factoring problem, due to Lenstra et al. [1982] is considered a major result in computer science algorithm design. The key ingredient to their solution is the construction of integer relations to real or complex numbers. For the simple demonstration of this idea, consider the polynomial

$$x^4 + 2x^3 - 6x^2 - 4x + 8$$

A root of this polynomial is $\alpha \approx 1.236067977$, and $\alpha^2 \approx 1.527864045$. We note that $2\alpha + \alpha^2 \approx 4.000000000$, hence $x^2 + 2x - 4$ is a factor. The main difficulty is to efficiently compute the integer linear relation with relatively small coefficients for the high-precision big-float approximations of the powers of a root. Lenstra et al. [1982] solve this diophantine optimization problem by means of their now famous lattice reduction procedure, which is somewhat reminiscent of the ellipsoid method for linear programming.

The determination of linear integer relations among a set of real or complex numbers is a useful task in science in general. Very recently, some stunning identities could be produced by this method, including the following formula for π [Finch 1995]:

$$\pi = \sum_{n=0}^{\infty} \frac{1}{16^n} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right)$$