CHAPTER 2

The Design of Efficient Algorithms

1. The Complexity of Computer Algorithms

With the advent of the high-speed electronic computer, new branches of applied mathematics have sprouted forth. One area that has enjoyed a most rapid growth in the past decade is the complexity analysis of computer algorithms. At one level, we may wish to compare the relative efficiencies of procedures which solve the same problem. At a second level, we can ask whether one problem is intrinsically harder to solve than another problem. It may even turn out that a task is too hard for a computer to solve within a reasonable amount of time. Measuring the costs to be incurred by implementing various algorithms is a vital necessity in computer science, but it can be a formidable challenge.

Let us reflect for a moment on the differences between *computability* and *computational complexity*. These two topics, along with formal languages, become the pillars of the theory of computation. Computability addresses itself mostly to questions of existence: Is there an algorithm which solves problem Π? An early surprise for many math and computer science students is that one can prove mathematically that computers cannot do everything. A standard example is the unsolvability of the halting problem. Loosely stated, this says that *it is impossible for a professor to write a computer program which will accept as data any student's programming assignment and will return either the answer "yes, this student's program will halt within finite time" or "no, this student's program (has an infinite loop and) will run forever."* Proving that a problem *is* computable usually, but not always, consists of demonstrating an actual algorithm which will terminate with a correct answer for every input. The amount of resources (time and space) used in the calculation, although finite, is unlimited. Thus, computability gives us an understanding of the capabilities and limitations of the machines that mankind can build, but without regard to resource restrictions.

In contrast to this, computational complexity deals precisely with the quantitative aspects of problem solving. It addresses the issue of what can be computed within a *practical* or *reasonable* amount of time and space by measuring the resource requirements exactly or by obtaining upper and lower bounds. Complexity is actually determined on three levels: the problem, the algorithm, and the implementation. Naturally, we want the best algorithm which solves our problem, and we want to choose the best implementation of that algorithm.

A *problem* consists of a question to be answered, a requirement to be fulfilled, or a best possible situation or structure to be found, called a *solution*, usually in response to several input *parameters* or *variables*, which are described but whose values are left unspecified. A *decision problem* is one which requires a simple "yes" or "no" answer. An *instance* of a problem Π is a specification of particular values for its parameters. An *algorithm* for Π is a step-by-step procedure which when applied to any instance of Π produces a solution.

Usually we can rewrite an optimization problem as a decision problem which at first seems to be much easier to solve than the original but turns out to be just about as hard. Consider the following two versions of the graph coloring problem.

GRAPH COLORING (optimization version)
Instance: An undirected graph G.
Question: What is the smallest number of colors needed for a proper coloring of G?

GRAPH COLORING (decision version)
Instance: An undirected graph G and an integer $k > 0$.
Question: Does there exist a proper k coloring of G?

The optimization version can be solved by applying an algorithm for the decision version n times for an n-vertex graph. If the n decision problems are solved sequentially, then the time needed to solve the optimization version is larger than that for the decision version by at most a factor of n. However, if they can be solved simultaneously (in parallel), then the time needed for both versions is essentially the same.

It is customary to express complexity as a function of the size of the input. We say that an algorithm $\mathscr{A}$ for Π runs in time $O(f(m))$ if for some constant

$c > 0$ there exists an implementation of $\mathscr{A}$ which terminates after at most $cf(m)$ (computational) steps for all instances of size m. The complexity of an algorithm $\mathscr{A}$ is the smallest function f such that $\mathscr{A}$ runs in $O(f(m))$. The complexity of a problem Π is the smallest f for which there exists an $O(f(n))$-time algorithm $\mathscr{A}$ for Π, i.e., the minimum complexity over all possible algorithms solving Π. Thus, demonstrating and analyzing the complexity of a particular algorithm for Π provides us with an *upper bound* on the complexity of Π.*

By presenting faster and more efficient algorithms and implementations of algorithms, successive researchers have improved the complexity upper bounds (i.e., lowered them) for many problems in recent years. Consider the example of testing a graph for planarity. A graph is *planar* if it can be drawn on the plane (or on the surface of a sphere) such that no two edges cross one another. Kuratowski's [1930] characterization of planar graphs in terms of forbidden configurations provides an obvious exponential-time planarity algorithm, namely, verify that no subset of vertices induces a subgraph homeomorphic to K_5 or $\overline{2K_3}$. Auslander and Parter [1961] gave a planar embedding procedure, which Goldstein [1963] was able to formulate in such a way that halting was guaranteed. Shirey [1969] implemented this algorithm to run in $O(n^3)$ time for an n-vertex graph. In the meantime, Lempel, Even, and Cederbaum [1967] gave a different planarity algorithm, and, although they did not specify a time bound, an easy $O(n^2)$ implementation exists. Hopcroft and Tarjan [1972, 1974] then improved the Auslander–Parter method first to $O(n \log n)$ and finally to $O(n)$, which is the best possible. Booth and Leuker showed that the Lempel–Even–Cederbaum method could also be implemented to run in $O(n)$ time. Table 2.1 shows the stages of improvement for the planarity problem and for the maximum-network-flow problem. Tarjan [1978] summarizes the progress on a number of other problems.

Determining the complexity of a problem Π requires a two-sided attack:

(1) *The upper bound*—the minimum complexity over all *known* algorithms solving Π.

(2) *The lower bound*—the largest function f for which it has been proved (mathematically) that all *possible* algorithms solving Π are required to have complexity at least as high as f.

Our ultimate goal is to make these bounds coincide. A gap between (1) and (2) tells us how much more research is needed to achieve this goal. For many

* We have just described the worst-case complexity analysis. One may also formulate complexity according to the average case. A good discussion of the pros and cons of average-case analysis can be found in Weide [1977, Section 4].

Table 2.1

Progress on the complexity of two combinatorial problems

Planarity: A graph with n vertices		Maximum network flow: A network with n vertices and e edges	
exp	Kuratowski [1930]	Nonterminating under certain conditions	Ford and Fulkerson [1962]
		ne^2	Edmonds and Karp [1972][a]
n^3	Auslander and Parter [1961] Goldstein [1963] Shirey [1969]		
		n^2e	Dinic [1970][a]
n^2	Lempel, Even, and Cederbaum [1967]	n^3	Karzanov [1974]
$n \log n$	Hopcroft and Tarjan [1972]	$n^2e^{1/2}$	Cherkasky [1977]
n	Hopcroft and Tarjan [1974] Booth and Leuker [1976]	$n^{5/3}e^{2/3}$	Galil [1978]
		$ne \log^2 n$	Galil and Naamad [1979]
		?	

[a]Done independently.

problems this gap is stubbornly large. An example of this is the problem of matrix multiplication.

In Strassen [1969] an algorithm is presented for multiplying a pair of 2×2 matrices using only seven scalar multiplications. It is now known that seven multiplications is the best possible. For arbitrary $n \times n$ matrices Strassen's algorithm may be applied recursively (by first embedding the matrices into the next larger power of 2 in size) to obtain a general algorithm whose complexity is $O(n^{\log_2 7}) \approx O(n^{2.81})$. Until recently, $O(n^{2.81})$ was the best result known. The best algorithm known for the case of 3×3 matrices is given by Laderman [1976]; it uses 23 scalar multiplications. By appropriately composing these two methods with themselves or each other, we can obtain the best algorithms known for $n = 4, 6, 7, 8, 9$, and many other values. For

$n = 5$, G. A. Schachtel has an algorithm using 103 multiplications, an improvement of one given by O. Sýkora, which used 105. Asymptotically, however, in order to improve Strassen's general bound, one would need an algorithm for $n = 3$ using 21 or fewer multiplications (since $\log_3 21 < \log_2 7 < \log_3 22$) or an algorithm for $n = 5$ using 91 or fewer multiplications, etc. Amazingly, Pan [1978, 1979a] has discovered a collection of algorithms which do improve upon Strassen's bound. The best of these is an algorithm for $n = 48$ which uses 47,216 multiplications. Since $\log_{48} 47{,}216 \approx 2.78$, Pan's algorithm has a complexity of $O(n^{2.78})$. In very recent work Pan [1979b] has reduced the complexity down to $O(n^{2.6054})$ for very large n. This is currently the upper bound for the matrix multiplication problem. On the other hand, the tightest lower bound known to date for this problem is only $O(n^2)$ [see Aho, Hopcroft, and Ullman, 1974, p. 438].

The biggest open question involving the gap between upper and lower complexity bounds involves the so called NP-complete problems (discussed below). For each of the problems in this class only exponential-time algorithms are known, yet the best lower bounds proven so far are polynomial functions. Furthermore, if a polynomial-time algorithm exists for one of them, then such an algorithm exists for all of them. Included among the NP-complete problems on graphs are finding a Hamiltonian circuit, a minimum coloring, or a maximum clique. Appendix A contains a small collection of NP-complete problems which will suffice for the purposes of this book. For a more comprehensive list, the reader is referred to Garey and Johnson [1978]. Let us discuss the basics of this theory.

The *state* of an algorithm consists of the current values of all variables and the location of the current instruction to be executed. A *deterministic algorithm* is one for which each state upon execution of the instruction uniquely determines at most one next state. Virtually all computers, as we know them, run deterministically. A problem Π is in the class P if there exists a deterministic polynomial-time algorithm which solves Π.

A *nondeterministic algorithm* is one for which a state may determine many next states and which follows up on each of the next states *simultaneously*. We may regard a nondeterministic algorithm as having the capability of branching off into many copies of itself, one for each next state. Thus, while a deterministic algorithm must explore a set of alternatives one at a time, a nondeterministic algorithm examines all alternatives at the same time.

Following Reingold, Nievergelt, and Deo [1977], three special instructions are used in writing nondeterministic algorithms for decision problems:

$x \leftarrow$ **choice**(S) creates $|S|$ copies of the algorithm, and assigns every member of the set S to the variable x in one of the copies.

failure causes that copy of the algorithm to stop execution.

success causes all copies of the algorithm to stop execution and indicates a "yes" answer to that instance of the problem.

A nondeterministic polynomial-time algorithm for the decision version of the CLIQUE problem is the following: Let $G = (V, E)$ be an undirected graph and let $k \geq 0$.

```
   procedure CLIQUE($G$, $k$):
   begin
1.   $A \leftarrow \emptyset$;
2.   for all $v \in V$ do $A \leftarrow$ choice($\{A + \{v\}, A\}$);;
3.   if $|A| < k$ then failure;
4.   for all $v, w \in A, v \neq w$ do
5.      if $vw \notin E$ then failure;;
6.   success;
   end
```

The loop in line 2 nondeterministically selects a subset of vertices $A \subseteq V$; lines 4–6 decide if A is a complete set. If **success** is reached in one of the copies, then the final value of A in that copy is a clique of size at least k. Using the above procedure we obtain a nondeterministic polynomial-time algorithm for the optimization version of the CLIQUE problem as follows: Let G be an undirected graph with n vertices.

```
procedure MAXCLIQUE($G$):
begin
  for $k \leftarrow n$ to 1 step $-1$ do
     if CLIQUE($G$, $k$) then return $k$;
end
```

A problem Π is in the class NP if there exists a nondeterministic polynomial-time algorithm which solves Π. We have just demonstrated that CLIQUE $\in$ NP by presenting an appropriate algorithm. Clearly, $\mathrm{P} \subseteq \mathrm{NP}$. An important open question in the theory of computation is whether the containment of P in NP is proper; i.e., is $\mathrm{P} \neq \mathrm{NP}$?

One problem Π_1 is *polynomially transformable* to another problem Π_2, denoted $\Pi_1 \preccurlyeq \Pi_2$, if there exists a function f mapping the instances of Π_1 into the instances of Π_2 such that

(i) f is computable deterministically in polynomial time, and
(ii) a solution to the instance $f(I)$ of Π_2 gives a solution to the instance I of Π_1, for all I.

Intuitively this means that Π_1 is no harder to solve than Π_2 up to an added polynomial term, for we could solve Π_1 by combining the transformation f with the best algorithm for solving Π_2. Thus, if $\Pi_1 \preccurlyeq \Pi_2$, then

$$\text{COMPLEXITY}(\Pi_1) \leq \text{COMPLEXITY}(\Pi_2) + \text{POLYNOMIAL}.$$

If Π_2 has a deterministic polynomial-time algorithm, then so does Π_1; if every deterministic algorithm solving Π_1 requires at least an exponential amount of time, then the same is true of Π_2.

A problem Π is *NP-hard* if any one of the following equivalent conditions holds:

(H_1) $\Pi' \preccurlyeq \Pi$ for all $\Pi' \in$ NP;
(H_2) $\Pi \in$ P $\Rightarrow$ P = NP;
(H_3) the existence of a deterministic polynomial-time algorithm for Π would imply the existence of a polynomial-time algorithm for every problem in NP.

A problem Π is *NP-complete* if it is both a member of NP and is NP-hard (see Figure 2.1). The NP-complete problems are the most difficult of those in the "zone of uncertainty."

The topic of NP-completeness was initiated by Cook [1971]. Emphasizing the significance of polynomial-time reducibility, he focused attention on NP decision problems. He proved that the **SATISFIABILITY** problem of

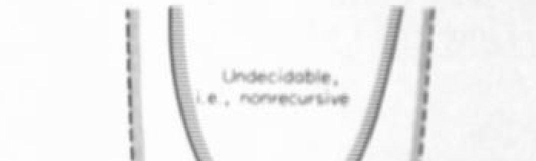


Figure 2.1. The hierarchy of complexities. The big open question is whether or not the "zone of uncertainty," NP–P, is empty.

mathematical logic is NP-complete (Cook's theorem), and he suggested other problems which might be NP-complete. Karp [1972] presented a large collection of NP-complete problems (about two dozen) arising from combinatorics, logic, set theory, and other areas of discrete mathematics. In the next few years, hundreds of problems were shown to be NP-complete. The standard technique employed with NP-completeness is as follows: First, by Cook's theorem, place SATISFIABILITY in the bag of NP-complete problems. Next, repeat the following sequence of instructions a few hundred times:

Find a candidate Π which might be NP-complete. Select an appropriate Π' from the bag of NP-complete problems. Show that $\Pi \in$ NP and $\Pi' \preccurlyeq \Pi$. Add Π to the bag.

An amount of cleverness is needed in selecting Π' and finding a transformation from Π' to Π. By way of illustration we will demonstrate such a reduction in the proof of the next theorem. For a more complete treatment of Cook's theorem and the reductions following from it, see (in increasing level of scope) Reingold, Nievergelt, and Deo [1977], Aho, Hopcroft, and Ullman [1974], and Garey and Johnson [1978].

To illustrate the technique of reduction, we present the following result.

Theorem 2.1 (Poljak [1974]). (i) STABLE SET $\preccurlyeq$ STABLE SET ON TRIANGLE-FREE GRAPHS;

(ii) STABLE SET $\preccurlyeq$ GRAPH COLORING.

Proof. (i) Let G be an undirected graph with n vertices and e edges. The idea of our proof will be to construct from G a certain triangle-free graph H with the property that knowing $\alpha(H)$ will immediately give us $\alpha(G)$. Subdivide each edge of G into a path of length 3; call the resulting graph H. Clearly, H is a triangle-free graph with $n + 2e$ vertices and $3e$ edges. Also, H can be constructed from G in $O(n + e)$ steps. Finally, since $\alpha(H) = \alpha(G) + e$, a deterministic polynomial time algorithm which solves for $\alpha(H)$ yields a solution to $\alpha(G)$.

(ii) Let G be an undirected graph and construct H as in part (i). Next we construct H' from H as follows. The vertices of H' correspond to the edges of H, and we connect two vertices of H' if their corresponding edges in H do not share a common vertex. This construction can be easily carried out in $O(e^2)$ steps. Since H is triangle-free, $\chi(H') = (2e + n) - \alpha(H) = e + n - \alpha(G)$. Thus, $\alpha(G)$ can be determined from $\chi(H')$. ∎

Since it is well known that STABLE SET is NP-complete, we obtain the following lesser known result.

Corollary 2.2. STABLE SET ON TRIANGLE-FREE GRAPHS is NP-complete.

A graph theoretic or other type of problem Π which is normally hard to solve in the general case may have an efficient solution if the input domain is suitably restricted. The HAMILTONIAN CIRCUIT problem, for example, is trivial if the only graphs considered are trees. However, we have seen that restricting the STABLE SET problem to triangle-free graphs is not sufficient to allow fast calculation (until someone proves that P = NP). Research has found interesting families $\mathscr{F}$ of graphs for which certain hard problems Π when restricted to $\mathscr{F}$ are nontrivial and tractable (i.e., in P). In this book we will consider this situation for various families of perfect graphs and some not so perfect graphs. A more perplexing topic currently under investigation by many complexity theorists is that of finding and understanding the cause of the *boundary* between the tractability and intractability of various problems Π.

One final note: Our definition of complexity suppressed one fundamental point. An implementation of an algorithm is always taken relative to some specified type of machine. As an underlying assumption throughout this book we will take the random access machine (RAM), introduced by Cook and Reckhow [1973], as our model of computation. The RAM is an abstraction of a general-purpose digital computer in which each storage cell has a unique address, allowing it to perform in one computational step an access to any cell, an arithmetic or Boolean operation, or a comparison. A computation is performed sequentially by a RAM, one step at a time. The theory of NP-complete problems is usually formulated using the Turing machine model rather than the RAM. This presents no difficulty, however, since any RAM can be simulated on a deterministic Turing machine with only a polynomial increase in running time.

Summary

Besides providing a basis for comparing algorithms which solve the same problem, algorithmic analysis has other practical uses. Most importantly, it affords us the opportunity to know *in advance of the computation* an estimate or a bound on the storage and run time requirements. Such advance knowledge would be essential when designing a computer system for a manned spacecraft in which the ability to calculate trajectories and fire the guidance rockets appropriately within tight constraints had better be guaranteed. Even in less urgent situations, having advanced estimates allows a programmer to set job card limits to abort those runs which exceed the expected bounds and hence probably contain errors, and to avoid aborting correct programs. Also such estimates are needed by the person who must decide whether or not it is worthwhile spending the necessary funds on computer time to carry out a certain (very large) computation.

2. Data Structures

As the name suggests, data structures provide a systematic framework in which the variables being processed (both input and internal) can be organized. Data structures are really mathematical objects, but we will usually refer to their computer implementations by the same names. The most familiar data structure is the *array*, which is used in conjunction with subscripted variables. A *0-dimensional array* is a single variable or storage location. A *d-dimensional array* can be defined recursively as a finite sequence of $(d-1)$-dimensional arrays all of the same size. A *vector* is usually stored as a 1-dimensional array and a *matrix* as a 2-dimensional array. It is generally accepted that the entries of an array must be homogeneous (i.e., all of the same type and all requiring the same amount of space).

The main feature of an array is its indexing capability. The subscripts should uniquely determine the location of each data item. The entries of an array are stored consecutively, and an addressing scheme using *multipliers* allows access to any entry in a constant amount of time, independent of the size of the array, on a random access machine. Thus, a query of the form "Is $A_{5,12} > 0$?" can be executed in essentially one step.

For those unfamiliar with the use of multipliers, the technique will be illustrated for an $m_1 \times m_2$ matrix A. Let us assume that the entries of A are stored sequentially in locations of size s in the order $A_{1,1}, A_{1,2}, \ldots, A_{1,m_2}, A_{2,1}, A_{2,2}, \ldots, A_{2,m_2}, \ldots, A_{m_1,1}, A_{m_1,2}, \ldots, A_{m_1 m_2}$ (row-major ordering). Then the space used by each row of A equals $m_2 s$. Now $A_{i,j}$ could be accessed by starting at $A_{1,1}$, jumping down $i-1$ rows, and then moving over $j-1$ columns. Thus, if $B = \text{ADDRESS}(A_{1,1})$, then we have the formula

$$\text{ADDRESS}(A_{i,j}) = B + (i-1)m_2 s + (j-1)s.$$

An analogous formula can be obtained for column-major ordering. This idea easily extends to d-dimensional arrays (Exercise 14).

A *list* is a data structure which consists of homogeneous *records* which are linked together in a linear fashion. Each record will contain one field or more of data and one field or more of pointers. Figure 2.2 shows two singly linked lists; each record has a single forward pointer. Unlike an array, in which the

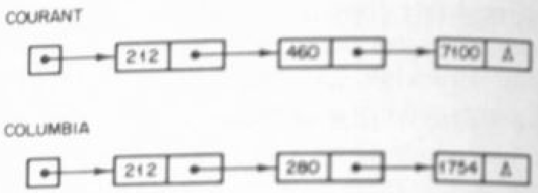


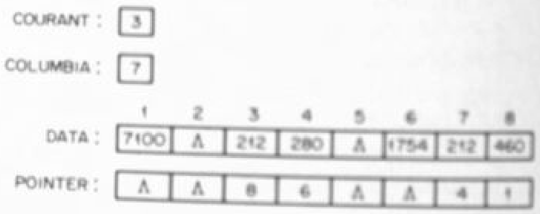
Figure 2.2. Two singly linked lists.

data is stored sequentially in memory, the records of a list can be scattered throughout memory. The pointers maintain law and order. This allows the flexibility of changing the size of the data structure, inserting and deleting items, by simply changing the values of a few pointers rather than shifting large blocks of data. An implementation of our examples is given in Figure 2.3. It uses two arrays and two single variables. The Λ is a special symbol indicating undefined. The list COURANT can be printed out by the following program:

```
begin
  $P \leftarrow$ COURANT;
  while $P \neq \Lambda$
    print DATA($P$);
    $P \leftarrow$ POINTER($P$);;
end
```

This is an example of *scanning* a list. Scanning takes time proportional to the length of the list.

Two special types of lists should be mentioned here because of their usefulness in computer science. A *stack* is a list in which we are only permitted to insert and delete elements at one end, called the *top* of the stack. A *queue* is a list in which we are only permitted to insert at one end, called the *tail* of the queue, and delete from the other end, called the *head* of the queue.



	1	2	3	4	5	6	7	8
DATA :	7100	Λ	212	280	Λ	1754	212	460
POINTER :	Λ	Λ	8	6	Λ	Λ	4	1

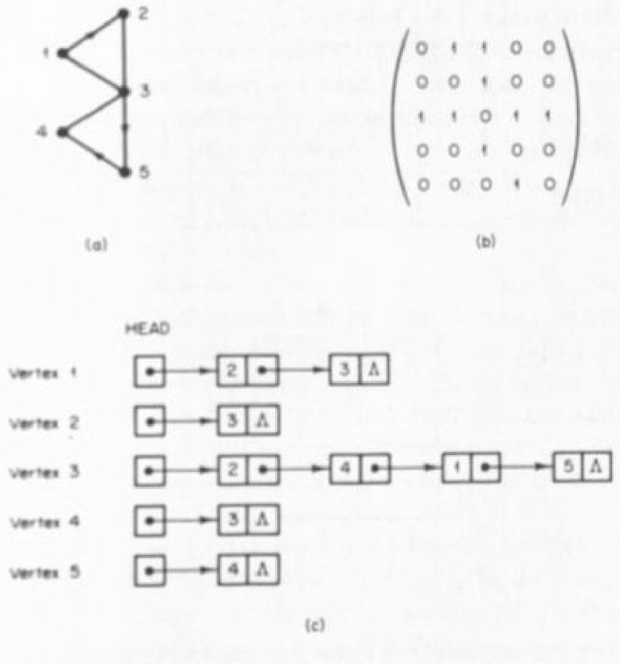
Figure 2.3. An implementation of the lists COURANT and COLUMBIA using arrays. (In what year was Columbia founded?)

The Adjacency Matrix of a Graph

Let $G = (V, E)$ be a graph whose vertices have been (arbitrarily) ordered $v_1, v_2, \ldots, v_n$. The adjacency matrix $\mathbf{M} = (m_{i,j})$ of G is an $n \times n$ matrix with entries

$$m_{i,j} = \begin{cases} 0 & \text{if } v_i v_j \notin E, \\ 1 & \text{if } v_i v_j \in E \end{cases}$$

(see Figure 2.4b). By definition, the main diagonal of $\mathbf{M}$ is all zeros, and M is symmetric about the main diagonal if and only if G is an undirected graph. If $\mathbf{M}$ is stored in a computer as a 2-dimensional array, then only one step (more precisely $O(1)$ time) is required for the statements "Is $v_i v_j \in E$?" or "Erase the edge $v_i v_j$." An instruction such as "mark each vertex which is adjacent to v_j" requires scanning the entire column j and hence takes n steps. Similarly, "mark each edge" takes n^2 steps. The space requirement for the array representation is $O(n^2)$. A graph whose edges are weighted can be represented in the same fashion. In this more general case $m_{i,j}$ will equal the weight of $v_i v_j$; a nonedge will have weight either zero or infinity depending upon the application.



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Figure 2.4. (a) The graph G. (b) The adjacency matrix of G. (c) The adjacency lists of G.

Some of the performance figures above can be improved upon when the density of **M** is low. We use the term *sparse* to indicate that $|E| \ll n^2$, i.e., the number of edges is *much* less than n^2. One of the most talked about classes of sparse graphs are the planar graphs for which Euler proved that $\|E\| \leq 3n - 6$.

The Adjacency Lists of a Graph

For each vertex v_i of G we create a list Adj(v_i) containing those vertices adjacent to v_i. The adjacency lists are not necessarily sorted although one might wish them to be (see Figure 2.4c). The space requirement for the adjacency list representation of a graph with n vertices and e edges is

$$O\left(\sum_i [1 + d_i]\right) = O(n + e),$$

where d_i denotes the degree of v_i (see Figure 2.5). Thus, from storage considerations, it is usually more advantageous to use adjacency lists than the

	1	2	3	4	5
HEAD:	3	2	6	10	1

	1	2	3	4	5	6	7	8	9	10	11
DATA:	4	3	2	1	4	2	Λ	3	5	3	Λ
LINK:	Λ	Λ	8	9	4	5	Λ	Λ	Λ	Λ	Λ

(a)

	1	2	3	4	5
HEAD:	1	3	4	8	9
DEGREE:	2	1	4	1	1

	1	2	3	4	5	6	7	8	9
DATA:	2	3	3	2	4	1	5	3	4

(b)

Figure 2.5. Two implementations of Figure 2.4c. (a) An implementation of the adjacency sets of G as linked lists. (b) An implementation of the adjacency sets of G in sequential storage.

Table 2.2

Some typical graph operations and their complexity with respect to three data structures[a]

	Adjacency matrix stored as an array	Adjacency sets stored as lists	Adjacency sets stored sequentially
Is v_iv_j an edge?	$O(1)$	$O(d_i)^*$	$O(d_i)^*$
Mark each vertex which is adjacent to v_i	$O(n)$	$O(d_i)$	$O(d_i)$
Mark each edge	$O(n^2)$	$O(e)$	$O(e)$
Add an edge v_iv_j	$O(1)$	$O(1)^{**}$	$O(e)$
Erase an edge v_iv_j	$O(1)$	$O(d_i)^*$	$O(e)$

[a] If the adjacency sets are sorted, then the starred entries can be reduced to $O(\log\ d_i)$ using a binary search, but the double starred entry will increase to $O(d_i)$.

adjacency matrix to store a sparse graph. Often, it is also advantageous from *time* considerations to store a sparse graph using adjacency lists. For example, the instruction "mark each vertex which is adjacent to v_j" requires scanning the list Adj(v_j) and hence takes d_j steps. Similarly, "mark each edge" takes $O(e)$ steps using adjacency lists, a substantial saving over the adjacency matrix for a sparse graph. However, erasing an edge is more complex with lists than with the matrix (see Table 2.2). Thus *there is no representation of a graph that is best for all operations and processes.* Since the selection of a particular data structure can noticeably affect the speed and efficiency of an algorithm, decisions about the representation must incorporate a knowledge of the algorithms to be applied. Conversely, the choice of an algorithm may depend on how the data is initially given. For example, an algorithm to set up the adjacency lists of a sparse graph will take longer if we are initially given its adjacency matrix as an $n \times n$ array rather than as a collection of ordered pairs representing the edges.

A graph problem is said to be *linear in the size of the graph*, or simply *linear*, if it has an algorithm which can be implemented to run in $O(n + e)$ steps on a graph with n vertices and e edges. This is usually the best that one could expect for a graph problem. By a careful choice of algorithm and data structure a number of simple problems can be solved in linear time; these include testing for connectivity (Section 2.3), biconnectivity (Exercise 5), and planarity (Table 2.1). We will illustrate this on the problem of converting the adjacency lists of a graph into *sorted adjacency lists.*

It is by now a well-known fact that any algorithm which correctly sorts a set of k numbers using comparisons will require at least $k \log k$ comparisons both in the worst case and in the average case.* Furthermore, many

* All logarithms will be base 2.

$O(k \log k)$-time algorithms for sorting by comparisons are available: HEAP-SORT, BINARY INSERTION, MERGESORT, etc. This might suggest that sorting the adjacency list of v_i requires $d_i \log d_i$ steps, so that sorting *all* the adjacency lists would take $\Sigma_{i=1}^{n} d_i \log d_i$ steps, which is superlinear, i.e., greater than $O(n + e)$. As an alternative to comparison sorting, Adj(v_i) could be put into order using a radix or bucket sort. This method takes $O(n + d_i)$ moves and is executed as follows:

```
1. Initialize bit vector: $\langle b_1, b_2, \ldots, b_n \rangle \leftarrow \langle 0, 0, \ldots, 0 \rangle$
2. Scan Adj($v_i$) assigning: $b_j \leftarrow 1$ for each $v_j \in$ Adj($v_i$)
3. Set SortedAdj($v_i$) $\leftarrow \emptyset$
4. Scan bit vector: for $j \leftarrow 1$ to $n$ do
       if $b_j = 1$ then CONCATENATE $v_j$ to SortedAdj($v_i$)
```

If this were done for all adjacency sets, it would require $O(n^2)$ steps which is superlinear for sparse graphs. Happily, there is yet another method for ordering the adjacency lists, which turns out to be linear. It is conceptually very simple and differs from the above in that the SortedAdj(v_i) are not created separately, but rather, simultaneously.

Algorithm 2.1. Sorting the adjacency lists of a graph.

Input: The unsorted adjacency lists of a graph $G = (V, E)$ whose vertices are numbered $v_1, v_2, \ldots, v_n$.
Output: The sorted adjacency lists of the reversal $G^{-1} = (V, E^{-1})$. (If G is undirected, then $G = G^{-1}$; otherwise run the algorithm a second time on G^{-1}.)
Method: The algorithm is as follows:

```
   begin
1.   for $i \leftarrow 1$ to $n$ do SortedAdj($v_i$) $\leftarrow$ empty list;
2.   for $i \leftarrow 1$ to $n$ do
3.     for each $v_j \in$ Adj($v_i$) do
4.       CONCATENATE $v_i$ to SortedAdj($v_j$);
   end
```

Theorem 2.3. Algorithm 2.1 runs in $O(n + e)$ time.

Proof. Line 1 is a loop which takes $O(n)$ steps. Concatenation is independent of the length of a list provided that a pointing variable is used to remember the address of the end of the list. Thus line 4 takes $O(1)$ steps, and the loop 3–4 takes $O(d_i)$ steps. Therefore, the nested loops 2–4 require a total of $\sum_{i=1}^{n} O(d_i) = O(e)$ steps, which proves the theorem. ∎

The usual implementation of adjacency sets as linked lists is illustrated in Figure 2.5a. There is an alternate way of storing the adjacency sets when no inserting or deleting is anticipated. Under these circumstances sequential storage can be used to eliminate the links that were present in the list representation and thus save space. In both implementations HEAD(i) points to the first member of Adj(v_i), but Adj(v_i) is now stored in *consecutive locations* DATA(HEAD(i)), ..., DATA(HEAD(i) + DEGREE(i) − 1) (see Figure 2.5b) and Exercise 9).

For further reading on data structures and their uses see Knuth [1969], Aho, Hopcroft, and Ullman [1974], Horowitz and Sahni [1976], Lewis and Smith [1976], Wirth [1976], Goodman and Hedetniemi [1977], Reingold, Nievergelt, and Deo [1977], and Gotleib and Gotleib [1978].

3. How to Explore a Graph

In designing algorithms we frequently require a mechanism for exploring the vertices and edges of a graph. Having the adjacency sets at hand allows us to repeatedly pass from a vertex to one of its neighbors and thus "walk" through the graph. Typically, in the midst of such a searching algorithm, some of the vertices will have been visited, the remainder not yet visited. A decision will have to be made as to which vertex x is being visited next. Since, in general, there will be many eligible candidates for x, we may want to establish some sort of priority among them.

Two criteria of priority which prove to be especially useful in exploring a graph are discussed in this section. They are depth-first search (DFS) and breadth-first search (BFS). In both methods each edge is traversed exactly once in the forward and reverse directions and each vertex is visited. By examining a graph in such a structured way, some algorithms become easier to understand and faster to execute. The choice of which method to use will often affect the efficiency of the algorithm. Thus, simply selecting a clever data structure is not sufficient to insure a good implementation. A carefully chosen search technique is also needed.

Depth-First Search

In DFS we select and visit a vertex a, then visit a vertex b adjacent to a, continuing with a vertex c adjacent to b (but different from a), followed by an "unvisited" d adjacent to c, and so forth. As we go deeper and deeper into the graph, we will eventually visit a vertex y with no unvisited neighbors; when this happens, we return to the vertex x immediately preceeding y in the search and revisit x. Note that if G is a connected undirected graph, then

```
   procedure DFSEARCH($v$):
   begin
1.    mark $v$ "visited"; $i \leftarrow i + 1$; DFSNUMBER($v$) $\leftarrow i$;
2.    for each $w \in$ Adj($v$) do
3.    if $w$ is marked "unvisited" then
         begin
4.          add the edge $vw$ to $T$; FATHER($w$) $\leftarrow v$;
5.          DFSEARCH($w$);
         end
   end
```

Figure 2.6. Depth-first search.

each vertex will be visited and every edge will be explored once in both directions. If G is not connected, then such a search is carried out for each connected component of G.

A depth-first search of an undirected graph $G = (V, E)$ partitions the edge set into two classes T and B where T comprises a *spanning forest* of G with one spanning tree for each component of G. The edge xy is placed into T if vertex y was visited for the first time immediately following a visit to x. In this case x is called the *father* of y and y is the *son* of x. The origin of this male-dominated nomenclature appears to be biblical. The edges in T are called *tree edges*. The remaining edges, called *back edges*, are placed into B; they are also called *fronds* by an(n) arborist graph theorist. If G is connected then (V, T) is called a *depth-first spanning tree*. We consider each tree of the depth-first spanning forest to be *rooted* at the vertex at which the DFS of that tree was begun.

An algorithm for depth-first search is given below.

Algorithm 2.2. Depth-first search of a graph.

Input: An undirected graph $G = (V, E)$ represented by adjacency sets Adj(v), for $v \in V$.
Output: A partition of E into a set T of tree edges and a set B of back edges.
Method: All vertices are initially marked "unvisited." The procedure DFSEARCH in Figure 2.6 is used recursively. All edges in E not placed into T are assumed to be in B. In addition, the vertices are numbered from 1 to n according to the order in which they are first visited; DFSNUMBER(v) denotes this number for a vertex v. The algorithm is as follows:

```
   begin
6.    initialize: $T \leftarrow \varnothing$; $i \leftarrow 0$;
7.    for all $v \in V$ do mark $v$ "unvisited";
8.    while there exists $v \in V$ marked "unvisited" do
9.       DFSEARCH($v$);
   end
```

In general a graph may have many depth-first spanning forests. Indeed there is quite a bit of freedom in choosing the vertices in lines 2 and 8. Nonetheless, a depth-first spanning forest T has some important and useful properties, which we now state.

(D1) If v is a proper ancestor of w in T, then DFSNUMBER(v) < DFSNUMBER(w).

(D2) For every edge of G, whether tree or back edge, one of its endpoints is an ancestor of the other endpoint, that is, there are no "cross edges."

We leave the proof of properties (D1) and (D2) as an exercise.

DFSEARCH(v) is an example of a *recursive procedure*, that is, it calls itself. Such a procedure is implemented using a stack. When a call to itself is made, the current values of all variables local to the procedure and the line of the procedure which made the call are stored at the top of the stack. In this way when control is returned the computation can continue where it had left off. Some computer languages, like ALGOL, PL/I, PASCAL, and SETL, allow recursive subroutines and set up the stack automatically for you. Other languages, like FORTRAN, COBOL, or BASIC, do not have this feature, so that the programmer must set up his own stack to simulate the recursion.

Breadth-First Search

In BFS we select a vertex and put it on an initially empty queue of vertices *to be visited*. We repeatedly remove the vertex x at the head of the queue and then place onto the queue all vertices adjacent to x which have never been enqueued. As in the case of depth-first search, BFS is carried out once for each connected component of the graph. However, in BFS each vertex is visited only once (and is thus exhausted, having produced all its offspring in one visit).

A breadth-first search of an undirected graph $G = (V, E)$ also partitions the edge set into two classes: the tree edges in T and the back edges in B. Here an edge xy is placed into T if vertex y is enqueued during the visit to x. The (partial) subgraph (V, T) is called a *breadth-first spanning forest*.

An algorithm for breadth-first search is given below.

Algorithm 2.3. Breadth-first search of a graph.

Input: An undirected graph $G = (V, E)$ represented by adjacency sets Adj(v), for $v \in V$.

Output: A partition of E into a set T of tree edges and a set B of back edges.

Method: All vertices are initially marked "never enqueued." The procedure BFSEARCH in Figure 2.7 is used to visit a vertex. All edges in E not placed

```
procedure BFSEARCH($x$):
begin
1.   $i \leftarrow i + 1$; BFSNUMBER($x$) $\leftarrow i$;
2.   for each $y \in$ Adj($x$) do
3.      if $y$ is marked "never enqueued" then
           begin
4.            add the edge $xy$ to $T$; FATHER($y$) $\leftarrow x$;
5.            add $y$ to $Q$; mark $y$ "enqueued";
           end
end
```

Figure 2.7. Breadth-first search.

into T are assumed to be in B. An array BFSNUMBER records the order in which the vertices are enqueued and visited. The algorithm is as follows:

```
begin
6.   initialize: $T \leftarrow \varnothing$; $Q \leftarrow$ empty queue; $i \leftarrow 0$;
7.   for all $v \in V$ do mark $v$ "never enqueued";
8.   while $Q$ is empty and there exists $v \in V$ marked "never enqueued"
9.      add $v$ to $Q$; mark $v$ "enqueued";
10.     while $Q$ is nonempty
11.        $x \leftarrow$ head of $Q$; $Q \leftarrow Q - x$;
12.        BFSEARCH($x$);
        end
     end
end
```

Let T be a breadth-first spanning forest of an undirected graph $G = (V, E)$. As was the case for DFS, a graph may have many breadth-first spanning forests. The *level* (in T) of a vertex v is defined inductively:

$$\text{LEVEL}(v) = \begin{cases} 0, & \text{if } v \text{ is a root of a tree in } T \\ 1 + \text{LEVEL}(\text{FATHER}(v)), & \text{otherwise.} \end{cases}$$

A breadth-first spanning forest T satisfies the following properties.

(B1) If v is a proper ancestor of w in T, then BFSNUMBER(v) < BFSNUMBER(w).

(B2) Every edge of G, whether tree or back edge, connects two vertices whose level in T differs by at most 1.

(B3) If v is a vertex in the connected component of G whose root in T is r, then the level of v equals the length of the shortest path from r to v in G.

In Section 4.3 we will discuss a variant of the process described here, called lexicographic breadth-first search, in which the vertices of a given level are not searched in the same order as they are enqueued, but rather according to a priority which depends on their ancestors.

Implementation and Complexity

Let $G = (V, E)$ be a graph. Both Algorithms 2.2 and 2.3 can be implemented to run in time and space proportional to $|V| + |E|$. Such an implementation is said to be *linear* in the size of G. This is usually the best that one can expect from a graph algorithm, since it is reasonable to assume each vertex and each edge must be processed. Let us describe in detail a linear implementation of Algorithm 2.2 and leave Algorithm 2.3 as an exercise.

The adjacency sets of G can be stored either as singly linked lists or by using sequential allocation*; thus, the input can be entered in $O(|V| + |E|)$ time and space. A Boolean array VISITED of size $|V|$ can serve to mark each vertex v *unvisited* if VISITED$(v) = 0$ and *visited* if VISITED$(v) = 1$. Thus, line 7 of Algorithm 2.2 can be executed in $O(|V|)$ time, and the tests in lines 3 and 8 can be done in constant time. The set T can be a singly linked list, while FATHER and DFSNUMBER will be arrays of size $|V|$. Hence statements 1, 4–6, and 9 can each be done in constant time. Now comes the crucial part of the complexity analysis. Statement 8 requires a pointing variable which will scan, or run through, all the vertices exactly once. That is, when this pointer finds an unvisited vertex, the pointer's value will be saved, so that the next time statement 8 is required the search for an unvisited vertex can resume at the spot where it had last left off (rather than starting at the beginning of V each time). Therefore, the total number of operations summed over all executions of statement 8 is proportional to $|V|$. Exactly the same technique is used in statement 2 to scan Adj(v) which, together with our previous comments, implies that the entire procedure DFSEARCH(v) takes $O(|\text{Adj}(v)|)$ time. Finally, the procedure is called once for each vertex, so the total time (and space) complexity of our implementation is

$$O(|V|) + \sum_{v \in V} O(|\text{Adj}(v)|),$$

which equals $O(|V| + |E|)$.

As we mentioned in the opening paragraphs of this section, one search technique may be preferable over another, that is, it may give us a more efficient implementation. We list some instances of problems for which DFS and BFS are most effective, respectively.

DFS—planarity testing; certain connectivity related problems (biconnectivity, triconnectivity); topological sorting; testing for cycles in an oriented graph.

* If inserting or deleting of edges were required in the algorithm, then sequential allocation would not be advisable.

```
    procedure DFSEARCH($v$):
    begin
1.    mark $v$ "visited"; $i \leftarrow i + 1$; DFSNUMBER($v$) $\leftarrow i$;
2.    for each $w \in \text{Adj}(v)$ do
3.    if $w$ is marked "unvisited" then
        begin
4.        add the edge $vw$ to $T$; FATHER($w$) $\leftarrow v$;
5.        DFSEARCH($w$);
        end
    end
```

Figure 2.6. Depth-first search.

each vertex will be visited and every edge will be explored once in both directions. If G is not connected, then such a search is carried out for each connected component of G.

A depth-first search of an undirected graph $G = (V, E)$ partitions the edge set into two classes T and B where T comprises a *spanning forest* of G with one spanning tree for each component of G. The edge xy is placed into T if vertex y was visited for the first time immediately following a visit to x. In this case x is called the *father* of y and y is the *son* of x. The origin of this male-dominated nomenclature appears to be biblical. The edges in T are called *tree edges*. The remaining edges, called *back edges*, are placed into B; they are also called *fronds* by an(n) arborist graph theorist. If G is connected then (V, T) is called a *depth-first spanning tree*. We consider each tree of the depth-first spanning forest to be *rooted* at the vertex at which the DFS of that tree was begun.

An algorithm for depth-first search is given below.

Algorithm 2.2. Depth-first search of a graph.

Input: An undirected graph $G = (V, E)$ represented by adjacency sets Adj(v), for $v \in V$.
Output: A partition of E into a set T of tree edges and a set B of back edges.
Method: All vertices are initially marked "unvisited." The procedure DFSEARCH in Figure 2.6 is used recursively. All edges in E not placed into T are assumed to be in B. In addition, the vertices are numbered from 1 to n according to the order in which they are first visited; DFSNUMBER(v) denotes this number for a vertex v. The algorithm is as follows:

```
   begin
6.   initialize: $T \leftarrow \varnothing$; $i \leftarrow 0$;
7.   for all $v \in V$ do mark $v$ "unvisited";
8.   while there exists $v \in V$ marked "unvisited" do
9.     DFSEARCH($v$);
   end
```

In general a graph may have many depth-first spanning forests. Indeed there is quite a bit of freedom in choosing the vertices in lines 2 and 8. Nonetheless, a depth-first spanning forest T has some important and useful properties, which we now state.

(D1) If v is a proper ancestor of w in T, then DFSNUMBER(v) < DFSNUMBER(w).

(D2) For every edge of G, whether tree or back edge, one of its endpoints is an ancestor of the other endpoint, that is, there are no "cross edges."

We leave the proof of properties (D1) and (D2) as an exercise.

DFSEARCH(v) is an example of a *recursive procedure*, that is, it calls itself. Such a procedure is implemented using a stack. When a call to itself is made, the current values of all variables local to the procedure and the line of the procedure which made the call are stored at the top of the stack. In this way when control is returned the computation can continue where it had left off. Some computer languages, like ALGOL, PL/I, PASCAL, and SETL, allow recursive subroutines and set up the stack automatically for you. Other languages, like FORTRAN, COBOL, or BASIC, do not have this feature, so that the programmer must set up his own stack to simulate the recursion.

Breadth-First Search

In BFS we select a vertex and put it on an initially empty queue of vertices *to be visited*. We repeatedly remove the vertex x at the head of the queue and then place onto the queue all vertices adjacent to x which have never been enqueued. As in the case of depth-first search, BFS is carried out once for each connected component of the graph. However, in BFS each vertex is visited only once (and is thus exhausted, having produced all its offspring in one visit).

A breadth-first search of an undirected graph $G = (V, E)$ also partitions the edge set into two classes: the tree edges in T and the back edges in B. Here an edge xy is placed into T if vertex y is enqueued during the visit to x. The (partial) subgraph (V, T) is called a *breadth-first spanning forest*.

An algorithm for breadth-first search is given below.

Algorithm 2.3. Breadth-first search of a graph.

Input: An undirected graph $G = (V, E)$ represented by adjacency sets Adj(v), for $v \in V$.

Output: A partition of E into a set T of tree edges and a set B of back edges.

Method: All vertices are initially marked "never enqueued." The procedure BFSEARCH in Figure 2.7 is used to visit a vertex. All edges in E not placed

```
    procedure BFSEARCH($x$):
    begin
1.  $i \leftarrow i + 1$; BFSNUMBER($x$) $\leftarrow i$;
2.  for each $y \in$ Adj($x$) do
3.      if $y$ is marked "never enqueued" then
          begin
4.          add the edge $xy$ to $T$; FATHER($y$) $\leftarrow x$;
5.          add $y$ to $Q$; mark $y$ "enqueued";
          end
    end
```

Figure 2.7. Breadth-first search.

into T are assumed to be in B. An array BFSNUMBER records the order in which the vertices are enqueued and visited. The algorithm is as follows:

```
    begin
6.  initialize: $T \leftarrow \varnothing$; $Q \leftarrow$ empty queue; $i \leftarrow 0$;
7.  for all $v \in V$ do mark $v$ "never enqueued";
8.  while $Q$ is empty and there exists $v \in V$ marked "never enqueued"
9.      add $v$ to $Q$; mark $v$ "enqueued";
10.     while $Q$ is nonempty
11.         $x \leftarrow$ head of $Q$; $Q \leftarrow Q - x$;
12.         BFSEARCH($x$);
        end
      end
    end
```

Let T be a breadth-first spanning forest of an undirected graph $G = (V, E)$. As was the case for DFS, a graph may have many breadth-first spanning forests. The *level* (in T) of a vertex v is defined inductively:

$$\text{LEVEL}(v) = \begin{cases} 0, & \text{if } v \text{ is a root of a tree in } T \\ 1 + \text{LEVEL}(\text{FATHER}(v)), & \text{otherwise.} \end{cases}$$

A breadth-first spanning forest T satisfies the following properties.

(B1) If v is a proper ancestor of w in T, then BFSNUMBER(v) < BFSNUMBER(w).

(B2) Every edge of G, whether tree or back edge, connects two vertice whose level in T differs by at most 1.

(B3) If v is a vertex in the connected component of G whose root in T is r then the level of v equals the length of the shortest path from r to v in G.

In Section 4.3 we will discuss a variant of the process described here, callec lexicographic breadth-first search, in which the vertices of a given level are no searched in the same order as they are enqueued, but rather according to priority which depends on their ancestors.

Implementation and Complexity

Let $G = (V, E)$ be a graph. Both Algorithms 2.2 and 2.3 can be implemented to run in time and space proportional to $|V| + |E|$. Such an implementation is said to be *linear* in the size of G. This is usually the best that one can expect from a graph algorithm, since it is reasonable to assume each vertex and each edge must be processed. Let us describe in detail a linear implementation of Algorithm 2.2 and leave Algorithm 2.3 as an exercise.

The adjacency sets of G can be stored either as singly linked lists or by using sequential allocation*; thus, the input can be entered in $O(|V| + |E|)$ time and space. A Boolean array VISITED of size $|V|$ can serve to mark each vertex v *unvisited* if $\text{VISITED}(v) = 0$ and *visited* if $\text{VISITED}(v) = 1$. Thus, line 7 of Algorithm 2.2 can be executed in $O(|V|)$ time, and the tests in lines 3 and 8 can be done in constant time. The set T can be a singly linked list, while FATHER and DFSNUMBER will be arrays of size $|V|$. Hence statements 1, 4–6, and 9 can each be done in constant time. Now comes the crucial part of the complexity analysis. Statement 8 requires a pointing variable which will scan, or run through, all the vertices exactly once. That is, when this pointer finds an unvisited vertex, the pointer's value will be saved, so that the next time statement 8 is required the search for an unvisited vertex can resume at the spot where it had last left off (rather than starting at the beginning of V each time). Therefore, the total number of operations summed over all executions of statement 8 is proportional to $|V|$. Exactly the same technique is used in statement 2 to scan $\text{Adj}(v)$ which, together with our previous comments, implies that the entire procedure DFSEARCH(v) takes $O(|\text{Adj}(v)|)$ time. Finally, the procedure is called once for each vertex, so the total time (and space) complexity of our implementation is

$$O(|V|) + \sum_{v \in V} O(|\text{Adj}(v)|),$$

which equals $O(|V| + |E|)$.

As we mentioned in the opening paragraphs of this section, one search technique may be preferable over another, that is, it may give us a more efficient implementation. We list some instances of problems for which DFS and BFS are most effective, respectively.

DFS—planarity testing; certain connectivity related problems (biconnectivity, triconnectivity); topological sorting; testing for cycles in an oriented graph.

* If inserting or deleting of edges were required in the algorithm, then sequential allocation would not be advisable.

BFS—shortest-path problems; testing for chordless cycles (Sections 4.3 and 4.4); network flow problems.

In the next section we will discuss one of these problems—topological sorting. Upon completing that section the reader will have been exposed to all the algorithmic tools needed for the remainder of the book. For additional reading in this area see Aho, Hopcroft, and Ullman [1974], Goodman and Hedetniemi [1977], and Reingold, Nievergelt, and Deo [1977].

4. Transitive Tournaments and Topological Sorting

Let F be an orientation of the complete graph K_n on n vertices. Each edg xy of F may be regarded as the outcome of a contest between the vertices and y, where x was the loser and y the winner. We call F a *transitive tourna-ment* if, for all triples of vertices,

$$xy \in F \quad \text{and} \quad yz \in F \qquad \text{implies} \quad xz \in F. \qquad (1$$

Condition (1) simply says that F has no 3-cycles. A stronger statement ca be made.

Theorem 2.4. Let F be an orientation of the complete graph K_n. Th following statements are equivalent.

(i) F is a transitive tournament.
(ii) F is acyclic.

Moreover, the vertices can be linearly ordered $[v_1, v_2, \ldots, v_n]$ such that
(iii) v_i has in-degree $i - 1$ in F, for all i, and
(iv) $v_iv_j \in F$ if and only if $i < j$.

This linear ordering of the vertices is unique. Figure 2.8 shows a transiti tournament and the linear ordering of its vertices.

Proof. (i) ⇒ (ii) Since F is transitive, it has no 3-cycle. Suppose F has a l-cycle ($l > 3$) where l is smallest possible. But this l-cycle has a chord whic shortcuts it, producing a cycle of shorter length and thus contradicting th minimality of l. Hence, F is acyclic.

(ii) ⇒ (iii) If F is acyclic, then it has a sink (a vertex of out-degree zero Call the sink v_n. Clearly v_n has in-degree $n - 1$. Deleting v_n from the grap we obtain a smaller acyclic oriented graph, and the conclusion follows b induction.

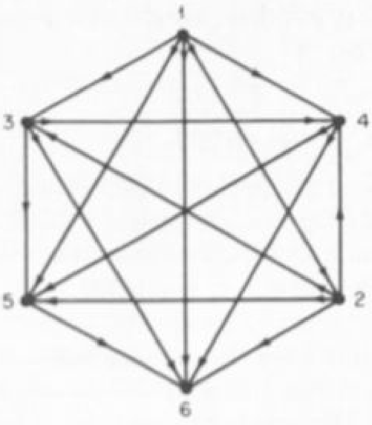


Figure 2.8. A transitive tournament.

(iii) ⇒ (iv) By induction.
(iv) ⇒ (i) Obvious.

This theorem provides us with a linear time algorithm for recognizing transitive tournaments. First, calculate the in-degree of each vertex; then, using a Boolean vector, verify that there are no duplicates among the in-degrees. The technique of recognizing a class of graphs *solely on the basis of the degrees of their vertices* will be seen again when we study threshold graphs (Chapter 10) and split graphs (Chapter 6).

A slightly more general problem than recognizing transitive tournaments is that of topologically sorting an arbitrary acyclic oriented graph $G = (V, F)$. What we seek is a linear ordering of the vertices $[v_1, v_2, \ldots, v_n]$ which is consistent with the edges of G; that is,

$$v_i v_j \in F \Rightarrow i < j \qquad \text{(for all } i, j). \tag{2}$$

An ordering which satisfies (2) is called a *topological sorting* of G. If G had a cycle, then a topological sorting would clearly be impossible. Why? But, if G is acyclic, then it is always possible. One method for finding an ordering satisfying (2) is the following:

```
for j ← |V| to 1 step −1
    Locate a sink v of the remaining graph and call it v_j;          (3)
    Delete v and all edges incident on v from the graph;
next j;
```

The correctness of this method is left as an exercise. In practice, we can implement (3) without actually deleting anything from our data structures.

Rather, we employ a depth-first search and some clever labeling. The algorithm is presented below.

Algorithm 2.4. Topological sorting.

Input: An acyclic oriented graph $G = (V, F)$ stored as adjacency lists.
Output: A DFS numbering of the vertices called DFSNUMBER and a topological sort numbering of the vertices called TSNUMBER. The algorithm also tests to make sure that G is acyclic.
Method: To find the jth vertex of the desired ordering, the depth-first search procedure TOPSORT in Figure 2.9 locates a vertex v all of whose successors in G have already been searched and numbered and are therefore considered as having been deleted. This vertex v is then numbered. The entire algorithm is as follows:

```
begin
  for each $x \in V$ do
    DFSNUMBER($x$) ← 0;
    TSNUMBER($x$) ← 0;;
  $j \leftarrow |V|$;
  $i \leftarrow 0$;
  for each $x \in V$ do
    if DFSNUMBER($x$) = 0 then
      TOPSORT($x$);
end
```

Algorithm 2.4 is illustrated in an example in Appendix C.

```
procedure TOPSORT($v$):
  $i \leftarrow i + 1$;
  DFSNUMBER($v$) ← $i$;
  for all $w \in$ Adj($v$) do
    begin
      if DFSNUMBER($w$) = 0 then
        TOPSORT($w$);
      else if TSNUMBER($w$) = 0 then
        "$G$ is not acyclic";
    end
  comment: We now label $v$ with a value smaller than
      the value assigned to any descendant.
  TSNUMBER($v$) ← $j$;
  $j \leftarrow j - 1$;
return
```

Figure 2.9.

EXERCISES

1. (a) Show that a spanning tree of the complete graph K_4 is either a depth-first spanning tree or a breadth-first spanning tree.

(b) Find a spanning tree of the complete graph K_5 which is neither a depth-first nor a breadth-first spanning tree.

2. Modify the DFS and BFS Algorithms 2.2 and 2.3 to count the number of connected components of an undirected graph G.

3. Prove properties (D1) and (D2) for any depth-first search spanning forest T.

4. A vertex x is an *articulation vertex* of G if deleting x and all edges incident on it increases the number of connected components. Let G be a connected undirected graph, and let T be a DFS spanning tree of G. Prove that a vertex x is an articulation vertex of G if and only if one of the following holds:

(i) x is the root of T, and x has more than one son;

(ii) x is not the root of T, and for some son s of x there is no back edge between any descendent of s (including s itself) and a proper ancestor of x.

Remark. A connected undirected graph G is *biconnected* (there are two vertex-disjoint paths between every pair of vertices) if and only if G has no articulation vertex.

5 (Biconnectivity). Let T be a DFS spanning tree of an undirected graph G. Assume that the vertices are numbered consecutively as they are first visited during depth-first search, and let $\pi(v)$ denote this number. For each vertex x, define

$$\text{LOW}(x) = \text{MIN}\{\pi(x), \pi(w)\},$$

where w runs over all proper ancestors of x accessible from a son of x by going down some tree edges and then up *one* back edge.

(a) Write a depth-first search algorithm which assigns the values $\pi(x)$ and calculates the values LOW(x) for all vertices x.

(b) Prove that your algorithm can run in $O(|V| + |E|)$ time for an arbitrary graph $G = (V, E)$.

(c) Show how your algorithm can detect articulation vertices using the function LOW.

6. Describe an efficient implementation of Algorithm 2.3 and prove that it is linear in the size of the graph.

7. Let S and T be subsets of the integers $1, 2, 3, \ldots, n$, and let A be a one-dimensional array of size n whose values have been initialized $A(1) = A(2) = \cdots = A(n) = 0$. Write subroutines which calculate $S \cup T$ and $S \cap T$ in time proportional to $|S| + |T|$. Assume that S and T are stored as (unordered) singly linked lists. (The answer appears in Appendix B.)

8. Let $H = (V, F)$ be an acyclic oriented graph. A height function h is defined on the vertices inductively:

$$h(v) = \begin{cases} 0 & \text{if } v \text{ is a sink,} \\ 1 + \max\{h(w) \mid w \in \mathrm{Adj}(v)\} & \text{otherwise.} \end{cases}$$

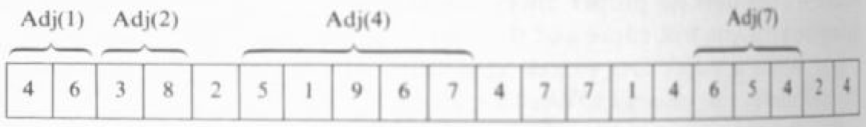
Write a DFS algorithm which assigns a height function h to the vertices. Prove that your algorithm can be implemented to run in $O(|V| + |F|)$ time.

9. Let $V = \{1, 2, \ldots, n\}$ and let E be a collection of m ordered pairs representing the edges of a graph $G = (V, E)$. Write a FORTRAN program which allocates *sequential space* in an array A of size m to store the adjacency sets of G, where Adj(1) is followed by Adj(2), etc. Let b_i denote the location in A of the beginning of Adj(i), and let d_i denote the out-degree of vertex i (i.e., the number of ordered pairs in which it is the first coordinate). You are permitted exactly two scans of E, one to calculate the out-degrees of the vertices and one to fill the array. For example, if

$$\begin{aligned} E = \{&(4, 5), (1, 4), (6, 7), (3, 2), (4, 1), (5, 4), (8, 2), \\ &(7, 6), (2, 3), (2, 8), (9, 4), (1, 6), (4, 9), (6, 1), \\ &(5, 7), (4, 6), (4, 7), (7, 5), (6, 4), (7, 4)\}, \end{aligned}$$

then the array A should look as indicated in Figure 2.10. Note that

$$b_i = 1 + \sum_{j<i} d_j.$$



$$\begin{array}{llllll} d_1 = 2 & d_4 = 5 & d_7 = 3 & b_1 = 1 & b_4 = 6 & b_7 = 16 \\ d_2 = 2 & d_5 = 2 & d_8 = 1 & b_2 = 3 & b_5 = 11 & b_8 = 19 \\ d_3 = 1 & d_6 = 3 & d_9 = 1 & b_3 = 5 & b_6 = 13 & b_9 = 20 \end{array}$$

Figure 2.10.

10. Using the data structure from Exercise 9 implement the algorithm from Exercise 8 and test it on some sample graphs.

11. Using the data structure from Exercise 9 implement the algorithm from Exercise 5 to test some sample undirected graphs for biconnectivity.

12. Let $U(n)$ be the set of all upper triangular, (0, 1)-valued $n \times n$ matrices. That is, an $n \times n$ matrix $\mathbf{M} = [m_{ij}]$ is in $U(n)$ if

$$m_{ij} = \begin{cases} 0 & i > j, \\ 1 & i = j, \\ 0 \text{ or } 1 & i < j. \end{cases}$$

Show that $U(n)$ forms a group under matrix multiplication over the two element field GF(2) and that the identity matrix I is the identity element of this group. (In GF(2): $0 + 0 = 1 + 1 = 0$, $0 + 1 = 1 + 0 = 1$, $0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 0$, and $1 \cdot 1 = 1$.)

13. Let $G = (V, F)$ be an acyclic, topologically sorted, oriented graph; i.e., its vertices have been renamed such that $V = \{1, 2, \ldots, n\}$ and

$$ij \in F \quad \text{implies} \quad i < j \qquad (i, j \in V).$$

Clearly, this numbering implies that the adjacency matrix $\mathbf{M}(G)$ of G is upper triangular and is therefore in $U(n)$ (see Exercise 12). Consider the subset

$$U_G = \{\mathbf{M} \in U(n) \mid m_{ij} = 1, i < j \Rightarrow ij \in F\}$$

consisting of those matrices in $U(x)$ with nonzeros only in the positions determined by the nonzeros of $\mathbf{M}(G)$.

(i) Show that the elements of U_G can be ordered by set inclusion to form a complete, distributive lattice.

(ii) Show that U_G is a subgroup of $U(n)$ if and only if F is transitive (i.e., a strict partial order).

14. Let A be a d-dimensional array of size $m_1 \times m_2 \times \cdots \times m_d$. Discuss how A may be stored in consecutive storage locations of size s in a manner similar to row-major or column-major ordering. Give a formula for obtaining the address of $A_{i_1, i_2, \ldots, i_d}$.

15. In the proof of Theorem 2.1, show that the following claims are valid:

(i) The transformation $G \mapsto H$ is $O(n + e)$.

(ii) $\alpha(H) = \alpha(G) + e$.

(iii) The transformation $H \mapsto H'$ is $O(e^2)$. Can this be improved?

(iv) $\chi(H') = e + n - \alpha(G)$.

16. Prove the following: If G has n vertices, then $\chi(G) \leq r$ if and only if $\alpha(G \times K_r) = r$, where $\times$ denotes the Cartesian product. (The Cartesian product of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the graph $G = (V_1 \times V_2, E)$, where $E = \{((v_1, v_2), (v_1', v_2')) \mid \text{either } v_1 = v_1' \text{ and } (v_2, v_2') \in E_2 \text{ or } v_2 = v_2' \text{ and } (v_1, v_1') \in E_1\}$ (Chvátal [1973, p. 326]).

17. Using Exercise 16, show that GRAPH COLORING $\preccurlyeq$ STABLE SET.

18. Prove that assigning a minimum coloring to a bipartite graph has complexity which is linear in the size of the graph.

19. Prove that STABLE SET restricted to bipartite graphs has complexity which is polynomial in the size of the graph.
20. Prove that HAMILTONIAN CIRCUIT restricted to bipartite graphs is NP-complete.
21. If a positive integer m can be stored in $1 + [\log_2 m]$ space, show that the numbers $1, 2, 3, \ldots, n$ can be stored in a total of $O(n)$ space.
22. Algorithms $\mathscr{A}$ and $\mathscr{B}$ run in n^2 and 2^n steps, respectively, on an input of size n.

(i) If current computers can execute 10^9 steps/sec, what size input can be processed by each algorithm in one minute? In one hour? In one year?

(ii) Suppose that by the time this book reaches your university library the computer industry has a technological breakthrough, which increases the speed of execution by 100-fold. What will be the corresponding increased capability of algorithms $\mathscr{A}$ and $\mathscr{B}$?

Bibliography

Aho, A. V., Hopcroft, J. E., and Ullman, J. D.
[1974] "The Design and Analysis of Computer Algorithms." Addison-Wesley, Reading, Massachusetts.

Auslander, L., and Parter, S.
[1961] On embedding graphs in the sphere, *J. Math. Mech.* **10**, 517–523. MR25 #1548.

Booth, K. S., and Leuker, G. S.
[1976] Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms, *J. Comput. System Sci.* **13**, 335–379. MR55 #6932.

Cherkasky, B. V.
[1977] Algorithm of construction of maximal flow in networks with complexity of $O(V^2\sqrt{E})$ operations, (in Russian), *Math. Methods of Solution of Economical Problems* 7, 117–125.

Christofides, Nicos
[1975] "Graph Theory—An Algorithmic Approach." Academic Press, New York.

Chvátal, Václáv.
[1973] Edmonds polytopes and a hierarchy of combinatorial problems, *Discrete Math.* 4, 305–337.

Cook, Stephen A.
[1971] The complexity of theorem-proving procedures, *Proc. 3rd Ann. ACM Symp. on Theory of Computing Machinery, New York*, pp. 151–158.

Cook, S. A., and Reckhow, R. A.
[1973] Time bounded random access machines, *J. Comput. System Sci.* **7**, 354–375.

Dinic, E. A.
[1970] Algorithm for solution of a problem of maximal flow in a network with proper estimation, *Soviet Math. Dokl.* **11**, 1277–1280. MR44 #5178.

Edmonds, Jack, and Karp, Richard M.
[1972] Theoretical improvements in algorithmic efficiency for network flow problems, *J. ACM* **19**, 248–264.

Ford, L. R., and Fulkerson, D. R.
[1962] "Flows in Networks." Princeton Univ. Press, Princeton, New Jersey

Galil, Zvi
[1978] A new algorithm for the maximum flow problem, *Proc. 19th IEEE Annu. Symp. on Foundations of Computer Science, Ann Arbor, Michigan, 16–18 October*, pp. 231–245.

Galil, Zvi, and Naamad, Amnon
[1979] Network flow and generalized path compression, *Proc. 11th Annu. ACM Symp. on Theory of Computing*.

Garey, Michael R., and Johnson, David S.
[1978] "Computers and Intractability: A Guide to the Theory of NP-completeness." Freeman, San Francisco, California.

Goldstein, A. J.
[1963] An efficient and constructive algorithm for testing whether a graph can be embedded in the plane, *Graph and Combinatorics Conf.*, Office of Naval Research Logistics Proj., Dept. of Math., Princeton Univ., Princeton, New Jersey.

Goodman, S. E., and Hedetniemi, S. T.
[1977] "Introduction to the Design and Analysis of Algorithms." McGraw-Hill, New York.

Gotlieb, Calvin, C., and Gotlieb, Leo R.
[1978] "Data types and structures." Prentice-Hall, Englewood Cliffs, New Jersey.

Hamacher, H.
[1979] Numerical investigations on the maximal flow algorithm of Karzanov, *Computing* **22**, 17–29.

Hopcroft, John E., and Tarjan, Robert Endre
[1972] Planarity testing in $V \log V$ steps: Extended abstract, *in* "Information Processing 71," Vol. 1, "Foundations and Systems," pp. 85–90. North-Holland, Amsterdam.
[1974] Efficient planarity testing, *J. ACM* **21**, 549–568.

Horowitz, E., and Sahni, S.
[1976] "Fundamentals of Data Structures." Computer Science Press, Potomac, Maryland.

Karp, Richard
[1972] Reducibility among combinatorial problems, *in* "Complexity of Computer Computations" (R. E. Miller and J. W. Thatcher, eds.), pp. 85–103. Plenum, New York.

Karzanov, A. V.
[1974] Determining the maximum flow in a network by the method of preflows, *Soviet Math. Dokl.* **15**, 434–437.
See Hamacher [1979] for an implementation of this algorithm.

Knuth, Donald E.
[1969] "The Art of Computer Programming," Vol. 1. Addison-Wesley, Reading, Massachusetts.
[1973] "The Art of Computer Programming," Vol. 3. Addison-Wesley, Reading, Massachusetts.

Kuratowski, C.
[1930] Sur le problème des corbes gauches en topologie, *Fund. Math.* **15**, 271–283.

Laderman, Julian D.
[1976] A noncommutative algorithm for multiplying 3 × 3 matrices using 23 multiplications, *Bull. Amer. Math. Soc.* **82**, 126–128.

Lempel, A., Even, S., and Cederbaum, I.
[1967] An algorithm for planarity testing of graphs, *in* "Theory of Graphs: Int. Symp., Rome, July 1966" (P. Rosentiehl, ed.), pp. 215–232. Gordon and Breach, New York.

Lewis, T. G., and Smith, M. Z.
[1976] "Applying Data Structures." Houghton Mifflin, Boston, Massachusetts.

Malhotra, V. M., Kumar, M. Pramodh, and Maheshwari, S. N.
[1978] An $O(n^3)$ algorithm for finding the maximum flows in networks, *Inf. Processing L*
7, 277–278.

Pan, Viktor Ya.
[1978] Strassen's algorithm is not optimal; trilinear techniques of aggregating, unifying a
cancelling for constructing fast algorithms for matrix operations, *Proc. 19th IE*
Annu. Symp. on Foundations of Computer Science, Ann Arbor, Michigan, 16–
October, pp. 166–176.
[1979a] New fast algorithms for matrix operations, *SIAM J. Comput.*, to be published.
[1979b] Field extension and trilinear aggregating, uniting and canceling for the acceleratio
of matrix multiplications, *Proc. 20th IEEE Annu. Symp. on Foundations of Comput*
Science, San Juan, Puerto Rico (29–31 October), pp. 28–38.

Poljak, S.
[1974] A note on stable sets and colorings of graphs, *Commun. Math. Univ. Carolinae* 1
307–309.

Reingold, Edward M., Nievergelt, Jurg, and Deo, Narsingh
[1977] "Combinatorial Algorithms: Theory and Practice." Prentice-Hall, Englewoo
Cliffs, New Jersey.

Shirey, R. W.
[1969] Implementation and analysis of efficient graph planarity testing algorithms, Ph. D
thesis, Univ. of Wisconsin.

Strassen, V.
[1969] Gaussian elimination is not optimal, *Numer. Math.* **13**, 354–356. MR40 # 2223.

Tarjan, Robert Endre
[1978] Complexity of combinatorial algorithms, *SIAM Rev.* **20**, 457–491.

Weide, Bruce
[1977] A survey of analysis techniques for discrete algorithms, *Comput. Surveys* **9**, 291–313.

Wirth, N.
[1976] "Algorithms + Data Structures = Programs." Prentice-Hall, Englewood Cliffs,
New Jersey.

CHAPTER 3

Perfect Graphs

1. The Star of the Show

In this section we introduce the main character of the book—*the perfect graph.* He was "discovered" by Claude Berge, who has been his agent since the early 1960s. P.G. has appeared in such memorable works as "Färbung von Graphen, deren sämtliche bzw. deren ungerade Kreise starr sind" and "Caractérisation des graphes non orientés dont on peut orienter les arrêtes de manière à obtenir le graphe d'une relation d'ordre." Despite his seemingly assuming name, P.G. has mixed the highbrow glamorous life with an intense dedication to improving the plight of mankind. His feature role in "Perfect graphs and an application to optimizing municipal services" has won him admiration and respect around the globe. Traveling incognito, a further sign of his modesty, he has been spotted by fans disguised as a graph parfait or as a (banana) split graph in a local ice cream parlor. So, ladies and gentlemen, without further ado, the management proudly presents

THE PERFECT GRAPH

Let us recall the following parameters of an undirected graph, which were defined in Section 1.1.

$\omega(G)$, the *clique number* of G: the size of the largest complete subgraph of G.

$\chi(G)$, the *chromatic number* of G: the fewest number of colors needed to properly color the vertices of G, or equivalently, the fewest number of stable sets needed to cover the vertices of G.