(b) ***Queue:*** A *queue*, also called a *first-in first-out* (FIFO) system, is a linear list in which deletions can only take place at one end of the list, the "front" of the list, and insertions can only take place at the other end of the list, the "rear" of the list. The structure operates in much the same way as a line of people waiting at a bus stop, as pictured in Fig. 8.3(*b*). That is, the first person in line is the first person to board the bus, and a new person goes to the end of the line.

(c) ***Priority queue:*** Let S be a set of elements where new elements may be periodically inserted, but where the current largest element (element with the "highest priority") is always deleted. Then S is called a *priority queue*. The rules "women and children first" and "age before beauty" are examples of priority queues. Stacks and ordinary queues are special kinds of priority queues. Specifically, the element with the highest priority in a stack is the last element inserted, but the element with the highest priority in a queue is the first element inserted.

8.2 GRAPHS AND MULTIGRAPHS

A *graph* G consists of two things:

(i) A set $V = V(G)$ whose elements are called *vertices*, *points*, or *nodes* of G.
(ii) A set $E = E(G)$ of unordered pairs of distinct vertices called *edges* of G.

We denote such a graph by $G(V, E)$ when we want to emphasize the two parts of G.

Vertices u and v are said to be *adjacent* if there is an edge $e = \{u, v\}$. In such a case, u and v are called the *endpoints* of e, and e is said to *connect* u and v. Also, the edge e is said to be *incident* on each of its endpoints u and v.

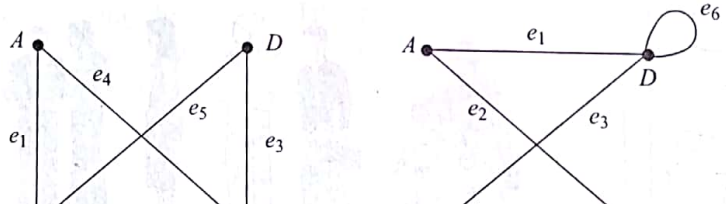
Graphs are pictured by diagrams in the plane in a natural way. Specifically, each vertex v in V is represented by a dot (or small circle), and each edge $e = \{v_1, v_2\}$ is represented by a curve which connects its endpoints v_1 and v_2. For example, Fig. 8.4(a) represents the graph $G(V, E)$ where:

(i) V consists of vertices A, B, C, D.
(ii) E consists of edges $e_1 = \{A, B\}$, $e_2 = \{B, C\}$, $e_3 = \{C, D\}$, $e_4 = \{A, C\}$, $e_5 = \{B, D\}$.

In fact, we will usually denote a graph by drawing its diagram rather than explicitly listing its vertices and edges.

Multigraphs

Consider the diagram in Fig. 8.4(*b*). The edges e_4 and e_5 are called *multiple edges* since they connect the same endpoints, and the edge e_6 is called a *loop* since its endpoints are the same vertex. Such a diagram is called a *multigraph*; the formal definition of a graph permits neither multiple edges nor loops. Thus a graph may be defined to be a multigraph without multiple edges or loops.



(*a*) Graph (*b*) Multigraph

Fig. 8.4

Remark: Some texts use the term *graph* to include multigraphs and use the term *simple graph* to mean a graph without multiple edges and loops.

Degree of a Vertex

The *degree* of a vertex v in a graph G, written deg (v), is equal to the number of edges in G which contain v, that is, which are incident on v. Since each edge is counted twice in counting the degrees of the vertices of G, we have the following simple by important result.

Theorem 8.2: The sum of the degrees of the vertices of a graph G is equal to twice the number of edges in G. This is termed as handshaking lemma.

Consider, for example, the graph in Fig. 8.4(a). We have

$$\deg(A) = 2, \quad \deg(B) = 3, \quad \deg(C) = 3, \quad \deg(D) = 2$$

The sum of the degrees equals 10 which, as expected, is twice the number of edges. A vertex is said to be *even* or *odd* according as its degree is an even or an odd number. Thus A and D are even vertices whereas B and C are odd vertices.

Theorem 8.2 also holds for multigraphs where a loop is counted twice towards the degree of its endpoint. For example, in Fig. 8.4(b) we have deg $(D) = 4$ since the edge e_6 is counted twice; hence D is an even vertex.

A vertex of degree zero is called an *isolated* vertex.

Finite Graphs, Trivial Graph

A multigraph is said to be *finite* if it has a finite number of vertices and a finite number of edges. Observe that a graph with a finite number of vertices must automatically have a finite number of edges and so must be finite. The finite graph with one vertex and no edges, i.e., a single point, is called the *trivial graph*. Unless otherwise specified, the multigraphs in this book shall be finite.

8.3 SUBGRAPHS, ISOMORPHIC AND HOMEOMORPHIC GRAPHS

This section will discuss important relationships between graphs.

Subgraphs

Consider a graph $G = G(V, E)$. A graph $H = H(V', E')$ is called a *subgraph* of G if the vertices and edges of H are contained in the vertices and edges of G, that is, if $V' \subseteq V$ and $E' \subseteq E$. In particular:

(i) A subgraph $H(V', E')$ of $G(V, E)$ is called the subgraph *induced* by its vertices V' if its edge set E' contains all edges in G whose endpoints belong to vertices in H.

(ii) If v is a vertex in G, then $G - v$ is the subgraph of G obtained by deleting v from G and deleting all edges in G which contain v.

(iii) If e is an edge in G, then $G - e$ is the subgraph of G obtained by simply deleting the edge e from G.

Isomorphic Graphs

Graphs $G(V, E)$ and G^* (V^*, E^*) are said to be *isomorphic* if there exists a one-to-one correspondence $f: V \to V^*$ such that $\{u, v\}$ is an edge of G if and only if $\{f(u), f(v)\}$ is an edge of G^*. Normally, we do not distinguish between isomorphic graphs (even though their diagrams may "look different"). Figure 8.5 gives ten graphs pictured as letters. We note that A and R are isomorphic graphs. Also, F and T, K and X, and M, S, V, and Z are isomorphic graphs.

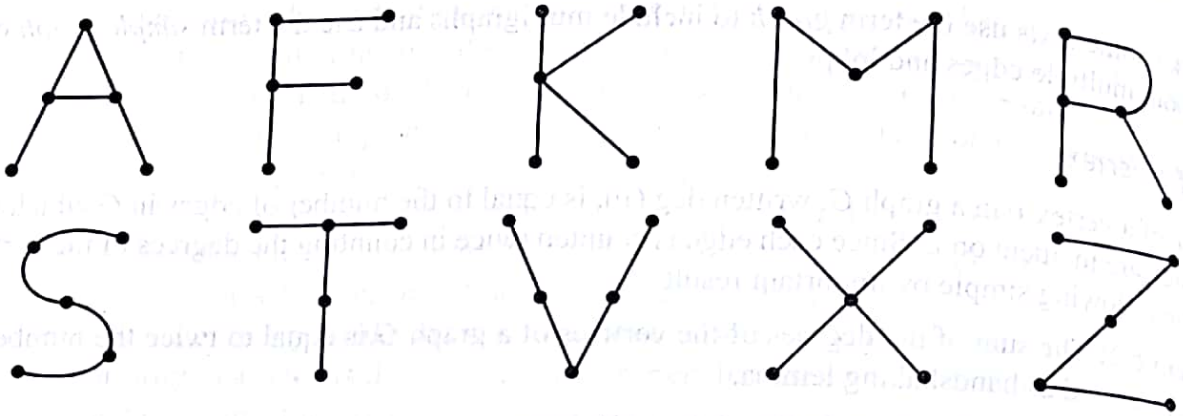


Fig. 8.5

Homeomorphic Graphs

Given any graph G, we can obtain a new graph by dividing an edge of G with additional vertices. Two graphs G and G^* are said to *homeomorphic* if they can be obtained from the same graph or isomorphic graphs by this method. The graphs (a) and (b) in Fig. 8.6 are not isomorphic, but they are homeomorphic since they can be obtained from the graph (c) by adding appropriate vertices.

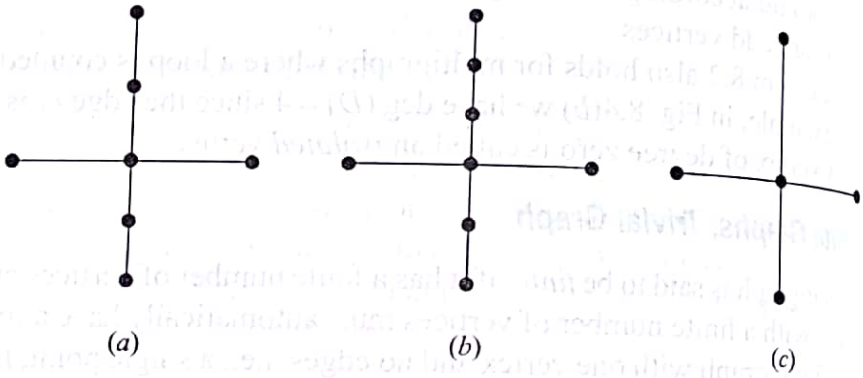


Fig. 8.6

8.4 PATHS, CONNECTIVITY

A *path* in a multigraph G consists of an alternating sequence of vertices and edges of the form

$$v_0, e_1, v_1, e_2, v_2, \ldots, e_{n-1}, v_{n-1}, e_n, v_n$$

where each edge e_i contains the vertices v_{i-1} and v_i (which appear on the sides of e_i in the sequence). The number n of edges is called the *length* of the path. When there is no ambiguity, we denote a path by its sequence of vertices $(v_0, v_1, \ldots, v_n)$. The path is said to be closed if $v_0 = v_n$. Otherwise, we say the path is from v_0 to v_n, or between v_0 and v_n, or connects v_0 to v_n.

A *simple path* is a path in which all vertices are distinct. (A path in which all edges are distinct will be called a *trail*) A *cycle* is a closed path in which all vertices are distinct except $v_0 = v_n$. A cycle of length k is called a *k-cycle*. In a graph, any cycle must have length 3 or more.

The sequence α is a path from P_4 to P_6; but it is not a trail since the edge $\{P_1, P_2\}$ is used twice. The sequence β is not a path since there is no edge $\{P_2, P_6\}$. The sequence γ is a trail since no edge is used twice; but it is not a single path since the vertex P_5 is used twice. The sequence δ is a simple path from P_4 to P_6; but it is not the shortest path (with respect to length) from P_4 to P_6. The shortest path from P_4 to P_6 is the simple path (P_4, P_5, P_6) which as length 2.

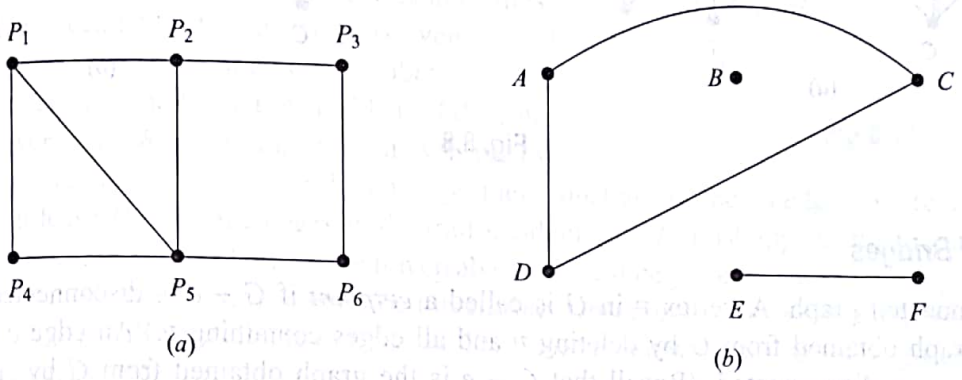


Fig. 8.7

By eliminating unnecessary edges, it is not difficult to see that any path from a vertex u to a vertex v can be replaced by a simple path from u to v. We state this reslt formally.

Theorem 8.3: There is a path from a vertex u to a vertex v if and only if there exists a simple path from u to v.

Connectivity, Connected Components

A graph G is *connected* if there is a path between any two of its vertices. The graph in Fig. 8.7(a) is connected, but the graph in Fig. 8.7(b) is not connected since, for example, there is no path between vertices D and E.

Suppose G is a graph. A connected subgraph H of G is called a *connected component* of G if H is not contained in any larger connected subgraph of G. It is intuitively clear that any graph G can be partitioned into its connected components. For example, the graph G in Fig. 8.7(b) has three connected components, the subgraphs induced by the vertex sets $\{A, C, D\}$, $\{E, F\}$, and $\{B\}$.

The vertex B in Fig. 8.7(b) is called an *isolated vertex* since B does not belong to any edge or, in other words, deg $(B) = 0$. Therefore, as noted, B itself forms a connected component of the graph.

Remark: Formally speaking, assuming any vertex u is connected to itself, the relation "u is connected to v" is an equivalence relation on the vertex set of a graph G and the equivalence classes of the relation form the connected components of G.

Distance and Diameter

Consider a connected graph G. The *distance* between vertices u and v in G, written $d(u, v)$, is the length of the shortest path between u and v. The *diameter* of G, written diam(G), is the maximum distance between any two points in G. For example, in Fig. 8.8(a), $d(A, F) = 2$ and diam$(G) = 3$, whereas in Fig. 8.8(b), $d(A, F) = 3$ and diam$(G) = 4$.

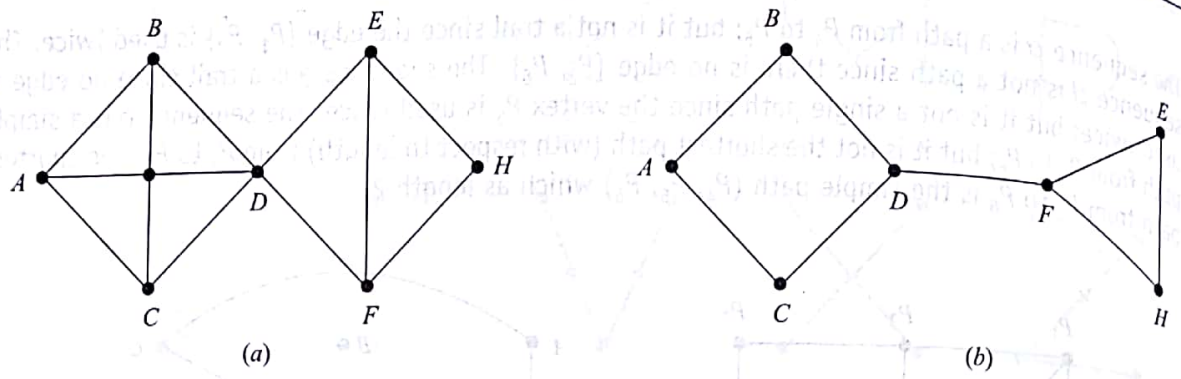


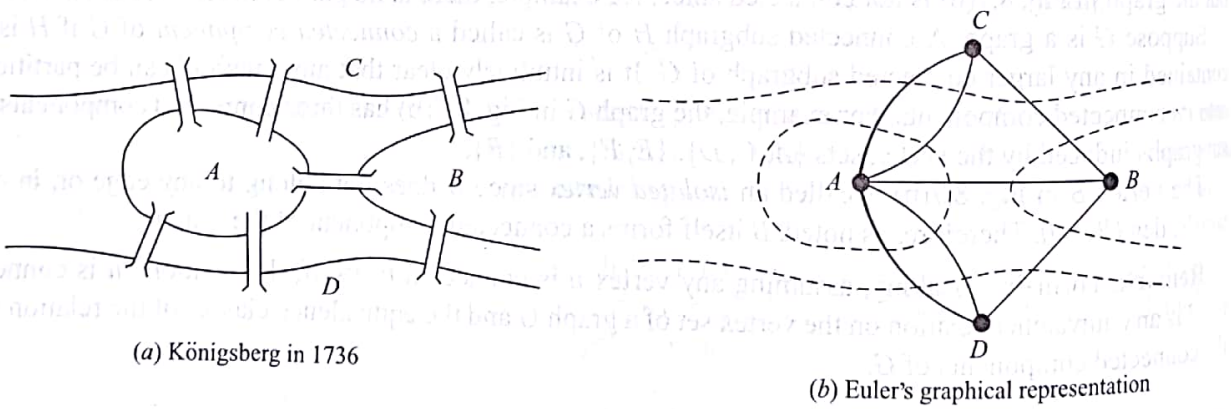
Fig. 8.8

Cutpoints and Bridges

Let G be a connected graph. A vertex v in G is called a *cutpoint* if $G - v$ is disconnected. (Recall that $G - v$ is the graph obtained from G by deleting v and all edges containing v.) An edge e of G is called a *bridge* if $G - e$ is disconnected. (Recall that $G - e$ is the graph obtained from G by simply deleting the edge e.) In Fig. 8.8(a), the vertex D is a cutpoint and there are no bridges. In Fig. 8.8(b), the edge $e = \{D, F\}$ is a bridge. (Its endpoints D and F are necessarily cutpoints.)

8.5 THE BRIDGES OF KÖNIGSBERG, TRAVERSABLE MULTIGRAPHS

The eighteenth-century East Prussian town of Königsberg included two islands and seven bridges as shown in Fig. 8.9(a). Question: Beginning anywhere and ending anywhere, can a person walk through town crossing all seven bridges but not crossing any bridge twice? The people of Königsberg wrote to the celebrated Swiss mathematician L. Euler about this question. Euler proved in 1736 that such a walk is impossible. He replaced the islands and the two sides of the river by points and the bridges by curves, obtaining Fig. 8.9(b).



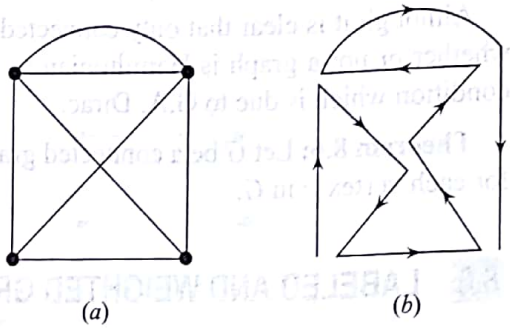
(a) Königsberg in 1736

(b) Euler's graphical representation

Fig. 8.9

Observe that Fig. 8.9(b) is a multigraph. A multigraph is said to be *traversable* if it "can be drawn without any breaks in the curve and without repeating any edges", that is, if there is a path which includes all vertices and uses each edge exactly once. Such a path must be a trail (since no edge is used twice) and will be called a *traversable trail*. Clearly a traversable multigraph must be finite and connected. Figure 8.10(b) shows a

traversable trail of the multigraph in Fig. 8.10(*a*). (To indicate the direction of the trail, the diagram misses touching vertices which are actually traversed.) Now it is not difficult to see that the walk in Königsberg is possible if and only if the multigraph in Fig. 8.9(*b*) is traversable.



(*a*) (*b*)

Fig 8.10

We now show how Euler proved that the multigraph in Fig. 8.9(*b*) is not traversable and hence that the walk in Königsberg is impossible. Recall first that a vertex is even or odd according as its degree is an even or an odd number. Suppose a multigraph is traversable and that a traversable trail does not begin or end at a vertex P. We claim that P is an even vertex. For whenever the traversable trail enters P by an edge, there must always be an edge not previously used by which the trail can leave P. Thus the edges in the trail incident with P must appear in pairs, and so P is an even vertex. Therefore, if a vertex Q is odd, the traversable trail must begin or end at Q. Consequently, a multigraph with more than two odd vertices cannot be traversable. Observe that the multigraph corresponding to the Königsberg bridge problem has four odd vertices. Thus one cannot walk through Königsberg so that each bridge is crossed exactly once.

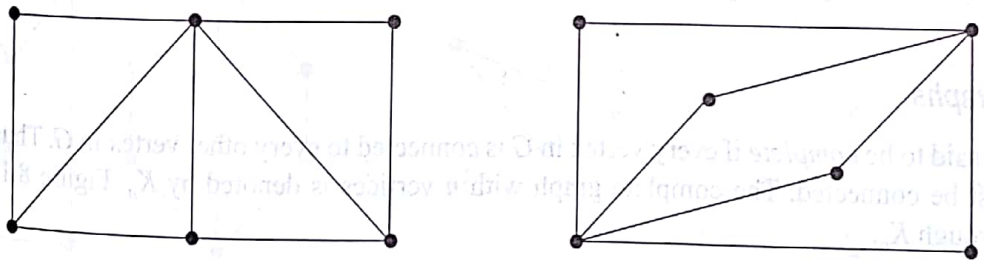
Euler actually proved the converse of the above statement, which is contained in the following theorem and corollary. (The theorem is proved in Solved Problem 8.9.) A graph G is called an *Eulerian* graph if there exists a closed traversable trail, called an *Eulerian* trail.

Theorem 8.4 (Euler): A finite connected graph is Eulerian if and only if each vertex has even degree.

Corollary 8.5: Any finite connected graph with two odd vertices is traversable. A traversable trail may begin at either odd vertex and will end at the other odd vertex.

Hamiltonian Graphs

The above discussion of Eulerian graphs emphasized traveling edges; here we concentrate on visiting vertices. A *Hamiltonian circuit* in a graph G, named after the nineteenth-century Irish mathematician William Hamilton (1805–1865), is a closed path that visits every vertex in G exactly once. (Such a closed path must be a cycle.) If G does admit a Hamiltonian circuit, then G is called a *Hamiltonian graph*. Note that an Eulerian circuit traverses every edge exactly once, but may repeat vertices, while a Hamiltonian circuit visits each vertex exactly once but may repeat edges. Figure 8.11 gives an example of a graph which is Hamiltonian but not Eulerian,and vice versa.



(*a*) Hamiltonian and non-Eulerian (*b*) Eulerian and non-Hamiltonian

Fig. 8.11

Although it is clear that only connected graphs can be Hamiltonian, there is no simple criterion to tell whether or not a graph is Hamiltonian as there is for Eulerian graphs. We do have the following sufficient condition which is due to G.A. Dirac.

Theorem 8.6: Let G be a connected graph with n vertices. Then G is Hamiltonian if $n \geq 3$ and $n \leq \deg(v)$ for each vertex v in G.

8.6 LABELED AND WEIGHTED GRAPHS

A graph G is called a *labeled graph* if its edges and/or vertices are assigned data of one kind or another. In particular, G is called a *weighted graph* if each edge e of G is assigned a nonnegative number $w(e)$ called the *weight* or *length* of e. Figure 8.12 shows a weighted graph where the weight of each edge is given in the obvious way. The *weight* (or *length*) of a path in such a weighted graph G is defined to be the sum of the weights of the edges in the path. One important problem in graph theory is to find a *shortest path*, that is, a path of minimum weight (length), between any two given vertices. The length of a shortest path between P and Q in Fig. 8.12 is 14; one such path is

$$(P, A_1, A_2, A_5, A_3, A_6, Q)$$

The reader can try to find another shortest path.

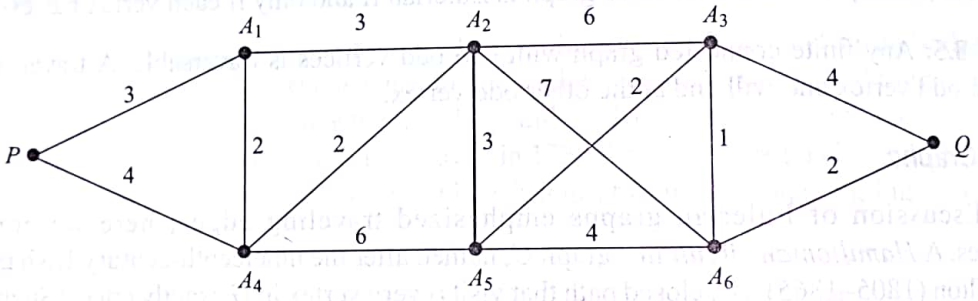


Fig. 8.12

8.7 COMPLETE, REGULAR, AND BIPARTITE GRAPHS

There are many different types of graphs. This section considers three of them, complete, regular, and bipartite graphs.

Complete Graphs

A graph G is said to be *complete* if every vertex in G is connected to every other vertex in G. Thus a complete graph G must be connected. The complete graph with n vertices is denoted by K_n. Figure 8.13 shows the graphs K_1 through K_6.

Regular Graphs

A graph G is *regular of degree k* or *k-regular* if every vertex has degree k. In other words, a graph is regular if every vertex has the same degree.

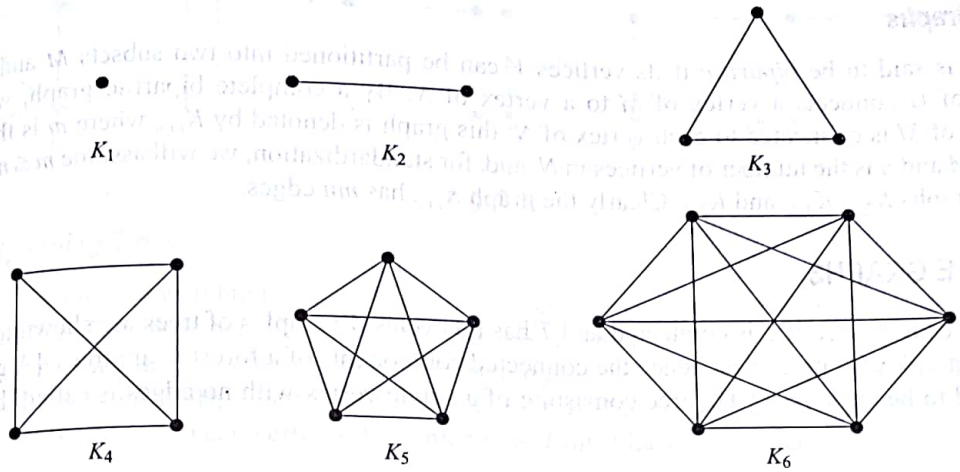


Fig. 8.13

The connected regular graphs of degrees 0, 1, or 2 are easily described. The connected 0-regular graph is the trivial graph with one vertex and no edges. The connected 1-regular graph is the graph with two vertices and one edge connecting them. The connected 2-regular graph with n vertices is the graph which consists of a single n-cycle.

See Fig. 8.14.

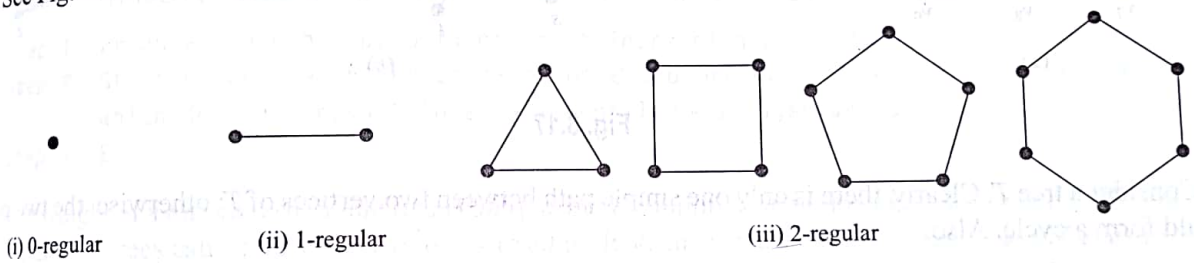
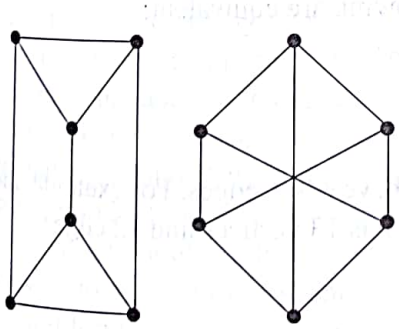


Fig. 8.14 *Regular graph*

The 3-regular graphs must have an even number of vertices since the sum of the degrees of the vertices is an even number (Theorem 8.1). Figure 8.15 shows two connected 3-regular graphs with six vertices. In general, regular graphs can be quite complicated. For example, there are nineteen 3-regular graphs with ten vertices. We note that the complete graph with n vertices K_n is regularto degree $n - 1$.



3-regular

Fig. 8.15

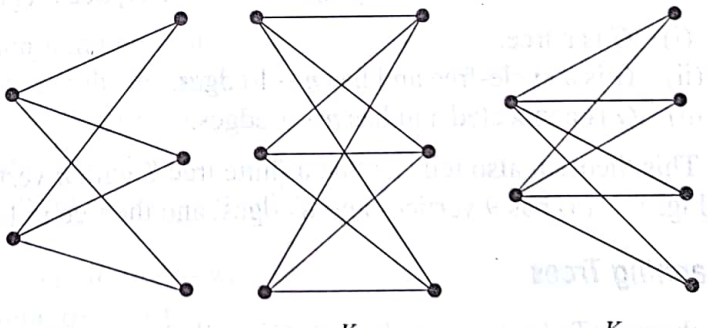


Fig. 8.16

Bipartite Graphs

A graph G is said to be *bipartite* if its vertices V can be partitioned into two subsets M and N such that each edge of G connects a vertex of M to a vertex of N. By a complete bipartite graph, we mean that each vertex of M is connected to each vertex of N; this graph is denoted by $K_{m,n}$ where m is the number of vertices in M and n is the number of vertices in N, and, for standardization, we will assume $m \leq n$. Figure 8.16 shows the graphs $K_{2,3}$, $K_{3,3}$, and $K_{2,4}$. Clearly the graph $K_{m,n}$ has mn edges.

8.8 TREE GRAPHS

A graph T is called a *tree* if T is connected and T has no cycles. Examples of trees are shown in Fig. 8.17. A *forest* G is a graph with no cycles; hence the connected components of a forest G are trees. [A graph without cycles is said to be *cycle-free*.] The tree consisting of a single vertex with no edges is called the *degenerate tree*.

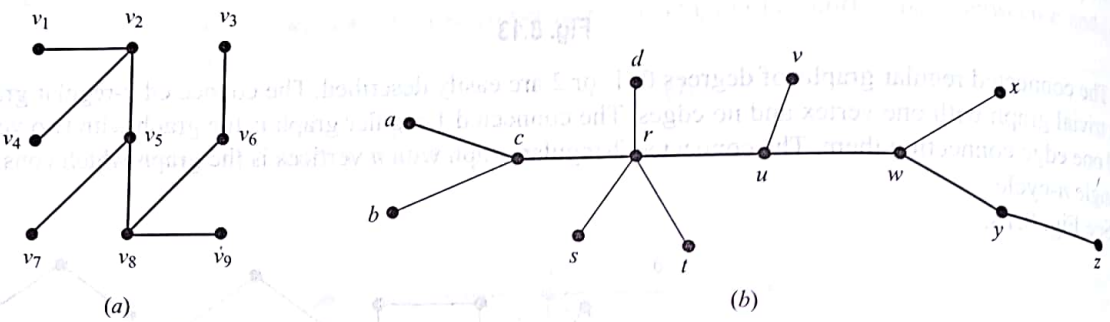


Fig. 8.17

Consider a tree T. Clearly, there is only one simple path between two vertices of T; otherwise, the two paths would form a cycle. Also:

(a) Suppose there is no edge $\{u, v\}$ in T and we add the edge $e = \{u, v\}$ to T. Then the simple path from u to v in T and e will form a cycle; hence T is no longer a tree.

(b) On the other hand, suppose there is an edge $e = \{u, v\}$ in T, and we delete e from T. Then T is no longer connected (since there cannot be a path from u to v); hence T is no longer a tree.

The following theorem (proved in Solved Problem 8.16) applies when our graphs are finite.

Theorem 8.7: Let G be a graph with $n > 1$ vertices. Then the following are equivalent:

(i) G is a tree.
(ii) G is a cycle-free and has $n - 1$ edges.
(iii) G is connected and has $n - 1$ edges.

This theorem also tells us that a finite tree T with n vertices must have $n - 1$ edges. For example, the tree in Fig. 8.17(a) has 9 vertices and 8 edges, and the tree in Fig. 8.17(b) has 13 vertices and 12 edges.

Spanning Trees

A subgraph T of a connected graph G is called a *spanning tree* of G if T is a tree and T includes all the vertices of G. Figure 8.18 shows a connected graph G and spanning trees T_1, T_2 and T_3 of G.

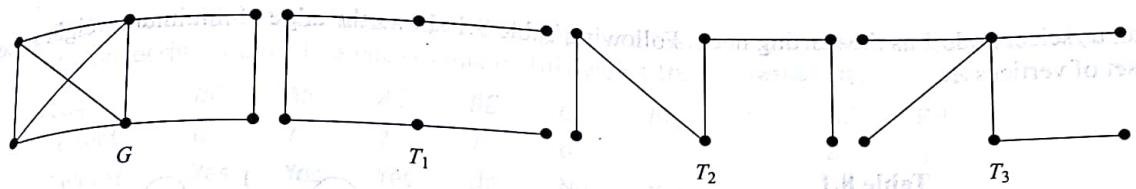


Fig. 8.18

Minimum Spanning Trees

Suppose G is a connected weighted graph. That is, each edge of G is assigned a nonnegative number called the *weight* of the edge. Then any spanning tree T of G is assigned a total weight obtained by adding the weights of the edges in T. A *minimal spanning tree* of G is a spanning tree whose total weight is as small as possible.

Algorithms 8.8A and 8.8B, which follow, enable us to find a minimal spanning tree T of a connected weighted graph G where G has n vertices. (In which case T must have $n - 1$ edges.)

Algorithm 8.8A: The input is a connected weighted graph G with n vertices.

Step 1. Arrange the edges of G in the order of decreasing weights.
Step 2. Proceeding sequentially, delete each edge that does not disconnect the graph until $n - 1$ edges remain.
Step 3. Exit.

Algorithm 8.8B (Kruskal): The input is a connected weighted graph G with n vertices.

Step 1. Arrange the edges of G in order of increasing weights.
Step 2. Starting only with the vertices of G and proceeding sequentially, add each edge which does not result in a cycle until $n - 1$ edges are added.
Step 3. Exit.

The weight of a minimal spanning tree is unique, but the minimal spanning tree itself is not. Different minimal spanning trees can occur when two or more edges have the same weight. In such a case, the arrangement of the edges in Step 1 of Algorithms 8.8A or 8.8B is not unique and hence may result in different minimal spanning trees as illustrated in Example 8.2.

Prim's Algorithm

All vertices of connected graph are included in minimum spanning tree of a graph G. Prim's algorithm starts with one vertex and grows the rest of the tree one vertex at a time, by adding associated edge. This algorithm builds a tree by iteratively adding edges until a minimal sapping tree is obtained, that is, till all nodes are added. At each iteration a minimum weight edge that does not complete a cycle is added to the existing tree.

Let $G = (V, E)$ be an original graph. Let T be a spanning tree. $T = (A, B)$, where initially A and B are empty sets. Let us select an arbitrary vertex i from V and add it to A. Now $A = \{i\}$. At each step prim's algorithm looks for the shortest possible edge $<u, v>$ such that $u \in A$ and $v \in V - A$. It then adds v to $A (A = A \cup \{v\})$ and adds edge $<u, v>$ to B. In this way, the edges in B form at any instant a minimum spanning tree for the vertices in A. We continue as long as $A \neq V$.

To illustrate the algorithm, let us consider the graph in Fig. 8.19.

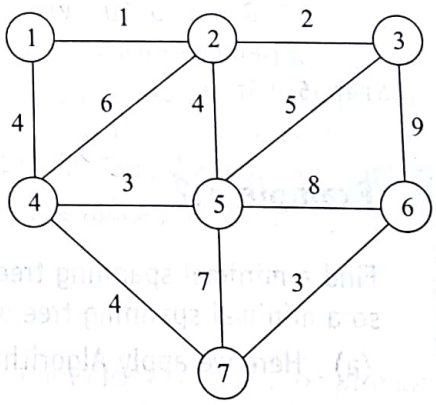


Fig. 8.19 *A weighted graph*

Let us select node 1 as the starting node. Following Table 8.1 shows the edge of minimum weight selected and set of vertices A.

Table 8.1

Step No.	Edge <u, v>	Set A
Initial	–	{1}
1	<1, 2>	{1, 2}
2	<2, 3>	{1, 2, 3}
3	<1, 4>	{1, 2, 3, 4}
4	<4, 5>	{1, 2, 3, 4, 5}
5	<4, 7>	{1, 2, 3, 4, 5, 7}
6	<7, 6>	{1, 2, 3, 4, 5, 7, 6}

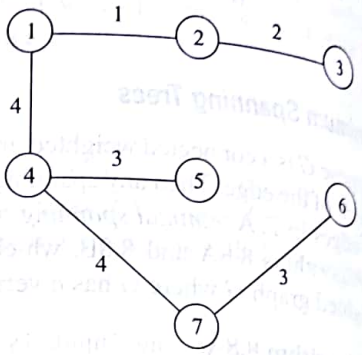


Fig. 8.20 *Minimum spanning tree*

When the algorithm stops, B contains the chosen edges $B = \{<1, 2>, <2, 3>, <1, 4>, <4, 5>, <4, 7>, <7, 6>\}$
The resultant spanning tree is drawn in Fig. 8.20 and is of weight = 17.
Following is an informal statement of algorithm:

Algorithm 8.8C (Prim's method): The input is a connected weighted graph G of n vertices.

{This algorithm generates minimum a spanning tree.}
Here G is graph and T is a spanning tree to be computed.
Step 1. Let $G = (V, E)$ and $T = (A, B)$
$A = \Phi$ and $B = \Phi$
Step 2. Let $i \in V$, i is a start vertex.
Step 3. $A = A \cup \{i\}$
Step 4. While $A \neq V$ do
begin
Find edge $\{u, v\} \in E$ of minimum length such that
$u \in A$ and $v \in V - A$
$A = A \cup \{v\}$ and
$B = B \cup \{u, v\}$
end
Step 5. Stop

Example 3.2

Find a minimal spanning tree of the weighted graph Q in Fig. 8.19(a). Note that Q has six vertices, so a minimal spanning tree will have five edges.

(a) Here we apply Algorithm 8.8A.

First we order the edges by decreasing weights, and then we successively delete edges without disconnecting Q until five edges remain. This yields the following data:

Edges	BC	AF	AC	BE	CE	BF	AE	DF	BD
Weight	8	7	7	7	6	5	4	4	3
Delete?	Yes	Yes	Yes	No	No	Yes			

Thus the minimal spanning tree of Q which is obtained contains the edges

$$BE,\quad CE,\quad AE,\quad DF,\quad BD$$

The spanning tree has weight 24 and it is shown in Fig. 8.21(b).

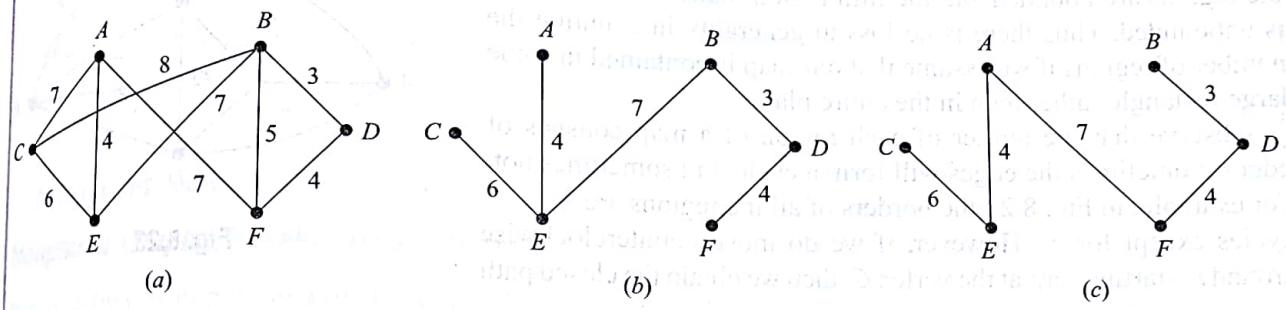


Fig. 8.21

(b) Here we apply Algorithm 8.8B.

First we order the edges by increasing weights, and then we successively add edges without forming any cycles until five edges are included. This yields the following data:

Edges	BD	AE	DF	BF	CE	AC	AF	BE	BC
Weight	3	4	4	5	6	7	7	7	8
Add?	Yes	Yes	Yes	No	Yes	No	Yes		

Thus the minimal spanning tree of Q which is obtained contains the edges

$$BD,\quad AE,\quad DF,\quad CE,\quad AF$$

The spanning tree appears in Fig. 8.19(c). Observe that this spanning tree is not the same as the one obtained using Algorithm 8.8A.

Remark: The above algorithms are easily executed when the graph G is relatively small as in Fig. 8.21(a). Suppose G has dozens of vertices and hundreds of edges which, say, are given by a list of pairs of vertices. Then even deciding whether G is connected is not obvious; it may require some type of depth-first search (DFS) or breadth-first search (BFS) graph algorithm. Later sections and the next chapter will discuss ways of representing graphs G in memory and will discuss various graph algorithms.

8.9 PLANAR GRAPHS

A graph or multigraph which can be drawn in the plane so that its edges do not cross is said to be *planar*. Although the complete graph with four vertices K_4 is usually pictured with crossing edges as in Fig. 8.22(a),

it can also be drawn with noncrossing edges as in Fig. 8.22(*b*); hence K_4 is planar. Tree graphs form an important class of planar graphs. This section introduces our reader to these important graphs.

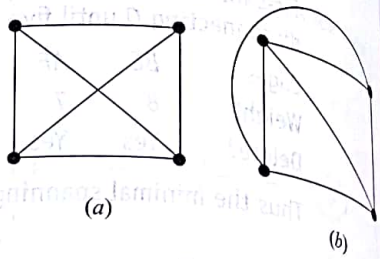


Fig. 8.22

Maps, Regions

A particular planar representation of a finite planar multigraph is called a *map*. We say that the map is *connected* if the underlying multigraph is connected. A given map divides the plane into various regions. For example, the map in Fig. 8.23 with six vertices and nine edges divides the plane into five regions. Observe that four of the regions are bounded, but the fifth region, outside the diagram, is unbounded. Thus there is no loss in generality in counting the number of regions if we assume that our map is contained in some large rectangle rather than in the entire plane.

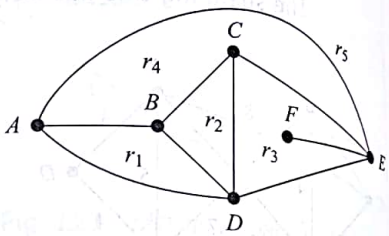


Fig. 8.23

Observe that the border of each region of a map consists of edges. Sometimes the edges will form a cycle, but sometimes not. For example, in Fig. 8.23 the borders of all the regions are cycles except for r_3. However, if we do move counterclockwise around r_3 starting, say, at the vertex C, then we obtain the closed path

$$(C, D, E, F, E, C)$$

where the edge $\{E, F\}$ occurs twice. By the *degree* of a region r, written deg (r), we mean the length of the cycle or closed walk which borders r. We note that each edge either borders two regions or is contained in a region and will occur twice in any walk along the border of the region. Thus we have a theorem for regions which is analogous to Theorem 8.2 for vertices.

Theorem 8.9: The sum of the degrees of the regions of a map is equal to twice the number of edges.

The degrees of the regions of Fig. 8.23 are:

$$\deg(r_1) = 3, \quad \deg(r_2) = 3, \quad \deg(r_3) = 5, \quad \deg(r_4) = 4, \quad \deg(r_5) = 3$$

The sum of the degrees is 18, which, as expected, is twice the number of edges.

For notational convenience we shall picture the vertices of a map with dots or small circles, or we shall assume that any intersections of lines or curves in the plane are vertices.

Euler's Formula

Euler gave a formula which connects the number V of vertices, the number E of edges, and the number R of regions of any connected map. Specifically:

Theorem 8.10 (Euler): $V - E + R = 2$.

(The proof of Theorem 8.8 appears in Problem 8.20.)

Observe that, in Fig. 8.21, $V = 6$, $E = 9$, and $R = 5$; and, as expected by Euler's formula.

$$V - E + R = 6 - 9 + 5 = 2$$

We emphasize that the underlying graph of a map must be connected in order for Euler's formula to hold.

Let G be a connected planar multigraph with three or more vertices, so G is neither K_1 nor K_2. Let M be a planar representation of G. It is not difficult to see that (1) a region of M can have degree 1 only if its border is a loop, and (2) a region of M can have degree 2 only if its border consists of two multiple edges. Accordingly,

if G is a graph, not a multigraph, then every region of M must have degree 3 or more. This comment together with Euler's formula is used to prove the following result on planar graphs.

Theorem 8.11: Let G be a connected planar graph with p vertices and q edges, where $p \geq 3$. Then $q \geq 3p - 6$.

Note that the theorem is not true for K_1 where $p = 1$ and $q = 0$, and is not true for K_2 where $p = 2$ and $q = 1$.

Proof: Let r be the number of regions in a planar representation of G. By Euler's formula,

$$p - q + r = 2$$

Now the sum of the degrees of the regions equals $2q$ by Theorem 8.7. But each region has degree 3 or more; hence

$$2q \geq 3r$$

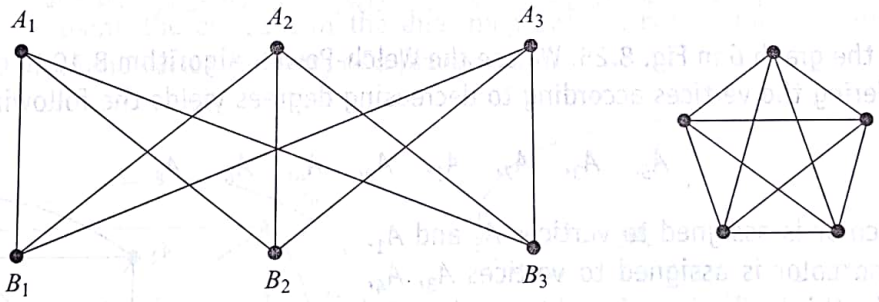
Thus $r \geq 2q/3$. Substituting this in Euler's formula gives

$$2 = p - q + r \leq p - q + \frac{2q}{3} \quad \text{or} \quad 2 \leq p - \frac{q}{3}$$

Multiplying the inequality by 3 gives $6 \leq 3p - q$ which gives us our result.

Nonplanar Graphs, Kuratowski's Theorem

We give two examples of nonplanar graphs. Consider first the *utility graph;* that is, three houses A_1, A_2, A_3 are to be connected to outlets for water, gas and electricity, B_1, B_2, B_3, as in Fig. 8.24(a). Observe that this is the graph $K_{3,3}$ and it has $p = 6$ vertices and $q = 9$ edges. Suppose the graph is planar. By Euler's formula a planar representation has $r = 5$ regions. Observe that no three vertices are connected to each other; hence the degree of each region must be 4 or more and so the sum of the degrees of the regions must be 20 or more. By Theorem 5.9 the graph must have 10 or more edges. This contradicts the fact that the graph has $q = 9$ edges. Thus the utlity graph $K_{3,3}$ is nonplanar.



(a) $K_{3,3}$ (b) K_5

Fig. 8.24

Consider next the *star graph* in Fig. 8.24(b). This is the complete graph K_5 on $p = 5$ vertices and has $q = 10$ edges. If the graph is planar, then by Theorem 8.9.

$$10 = q \leq 3p - 6 = 15 - 6 = 9$$

which is impossible. Thus, K_5 is nonplanar.

For many years mathematicians tried to characterize planar and nonplanar graphs. This problem was finally solved in 1930 by the Polish mathematician K. Kuratowski. The proof of this result, stated below, lies beyond the scope of this text.

Theorem 8.12 (Kuratowski): A graph is nonplanar if and only if it contains a subgraph homeomorphic to $K_{3,3}$ or K_5.

8.10 GRAPH COLORINGS

Consider a graph G. A *vertex coloring*, or simply a *coloring* of G is an assignment of colors to the vertices of G such that adjacent vertices have different colors. We say that G is n-colorable if there exists a coloring of G which uses n colors. (Since the word "color" is used as a noun, we will try to avoid its use as a verb by saying, for example, "paint" G rather than "color" G when we are assigning colors to the vertices of G.) The minimum number of colors needed to paint G is called the *chromatic number* of G and is denoted by $\chi(G)$.

We give an algorithm by Welch and Powell for a coloring of a graph G. We emphasize that this algorithm does not always yield a minimal coloring of G.

Algorithm 8.13 (Welch-Powell): The input is a graph G.

Step 1. Order the vertices of G according to decreasing degrees.

Step 2. Assign the first color C_1 to the first vertex and then, in sequential order, assign C_1 to each vertex which is not adjacent to a previous vertex which was assigned C_1.

Step 3. Repeat Step 2 with a second color C_2 and the subsequence of noncolored vertices.

Step 4. Repeat Step 3 with a third color C_3, then a fourth color C_4, and so on until all vertices are colored.

Step 5. Exit.

Example 8.3

(a) Consider the graph G in Fig. 8.25. We use the Welch-Powell Algorithm 8.10 to obtain a coloring of G. Ordering the vertices according to decreasing degrees yields the following sequence:

$$A_5,\ A_3,\ A_7,\ A_1,\ A_2,\ A_4,\ A_6,\ A_8$$

The first color is assigned to vertices A_5 and A_1. The second color is assigned to vertices A_3, A_4, and A_8. The third color is assigned to vertices A_7, A_2, and A_6. All the vertices have been assigned a color, and so G is 3-colorable. Observe that G is not 2-colorable since vertices A_1, A_2, and A_3, which are connected to each other, must be assigned different colors. Accordingly, $\chi(G) = 3$.

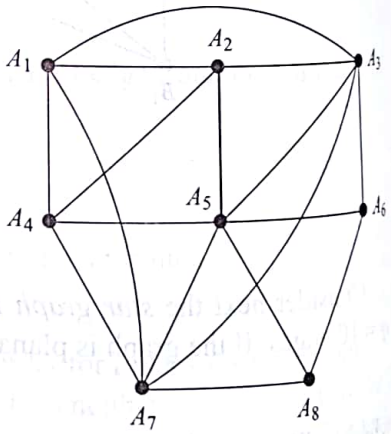


Fig. 8.25

(b) Consider the complete graph K_n with n vertices. Since very vertex is adjacent to every other vertex, K_n requires n colors in any coloring. Thus $\chi(K_n) = n$.

There is no simple way to actually determine whether an arbitrary graph is n-colorable. However, the following theorem (proved in Solved Problem 8.22) gives a simple characterization of 2-colorable graphs.

Theorem 8.14: The following are equivalent for a graph G:

(i) G is 2-colorable.
(ii) G is bipartite.
(iii) Every cycle of G has even length.

There is no limit on the number of colors that may be required for a coloring of an arbitrary graph since, for example, the complete graph K_n requires n colors. However, if we restrict ourselves to planar graphs, regardless of the number of vertices, five colors suffice. Specifically, in Solved Problem 8.24 we prove:

Theorem 8.15: Any planar graph is 5-colorable.

Actually, since the 1850s mathematicians have conjectured that planar graphs are 4-colorable since every known planar graph is 4-colorable. Apple and Haken finally proved this conjecture to be true in 1976. That is:

Four Color Theorem (Appel and Haken): Any planar graph is 4-colorable.

We discuss this theorem in the next subsection.

Dual Maps and the Four Color Theorem

Consider a map M, say the map M in Fig. 8.24(a). In other words, M is a planar representation of a planar multigraph. Two regions of M are said to be *adjacent* if they have an edge in common. Thus the regions r_2 and r_5 in Fig. 8.26(a) are adjacent, but the regions r_3 and r_5 are not. By a *coloring* of M we mean an assignment of a color to each region of M such that adjacent regions have different colors. A map M is n-colorable if there exists a coloring of M which uses n colors. Thus the map M in Fig. 8.26(a) is 3-colorable since the regions can be assigned the following colors:

r_1 red, r_2 white, r_3 red, r_4 white, r_5 red, r_6 blue

Observe the similarity between this discussion on coloring maps and the previous discussion on coloring graphs. In fact, using the concept of the dual map defined below, the coloring of a map can be shown to be equivalent to the vertex coloring of a planar graph.

that each vertex of M^* corresponds to exactly one region of M. Figure 8.26(b) shows the dual of the map of Fig. 8.26(a). One can prove that each region of M^* will contain exactly one vertex of M and that each edge of M^* will intersect exactly one edge of M and vice versa. Thus M will be the dual of the map M^*.

Observe that any coloring of the regions of a map M will correspond to a coloring of the vertices of the dual map M^*. Thus M is n-colorable if and only if the planar graph of the dual map M^* is vertex n-colorable. Thus the above theorem can be restated as follows:

Four Color Theorem (Appel and Haken)

If the regions of any map M are colored so that adjacent regions have different colors, then no more than four colors are required.

The proof of the above theorem uses computers in an essential way. Specifically, Appel and Haken first showed that if the four color theorem was false, then there must be a counterexample among one of approximately 2000 different types of planar graphs. They then showed, using the computer, that none of these types of graphs has such a counterexample. The examination of each different type of graph seems to be beyond the grasp of human beings without the use of a computer. Thus the proof, unlike most proofs in mathematics, is technology dependent; that is, it depended on the development of high-speed computers.

8.11 REPRESENTING GRAPHS IN COMPUTER MEMORY

There are two standard ways of maintaining a graph G in the memory of a computer. One way, called the *sequential representation* of G, is by means of its adjacency matrix A. The other way, called the *linked representation* or *adjacency structure* of G, uses linked lists of neighbors. Matrices are usually used when the graph G is dense, and linked lists are usually used when G is sparse. (A graph G with m vertices and n edges is said to be *dense* when $m = O(n^2)$ and sparse when $m = O(n)$ or even $O(n \log n)$.)

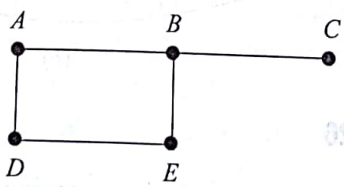
Regardless of the way one maintains a graph G in memory, the graph G is normally input into the computer by its formal definition, that is, as a collection of vertices and a collection of pairs of vertices (edges).

Adjacency Matrix

Suppose G is a graph with m vertices, and suppose the vertices have been ordered, say, $v_1, v_2, \ldots, v_m$. Then the *adjacency matrix* $A = [a_{ij}]$ of the graph G s the $m \times m$ matrix defined by

$$a_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases}$$

Figure 8.27(b) contains the adjacency matrix of the graph G in Fig. 8.27(a) where the vertices are ordered A, B, C, D, E. Observe that each edge $\{v_i, v_j\}$ of G is represented twice, by $a_{ij} = 1$ and $a_{ji} = 1$. Thus, in particular, the adjacency matrix is symmetric.



(a)

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	0	1
C	0	1	0	0	0
D	1	0	0	0	1
E	0	1	0	1	0

(b)

Fig. 8.27

The adjacency matrix A of a graph G does depend on the ordering of the vertices of G, that is, a different ordering of the vertices yields a different adjacency matrix. However, any two such adjacency matrices are closely related in that one can be obtained from the other by simply interchanging rows and columns. On the other hand, the adjacency matrix does not depend on the order in which the edges (pairs of vertices) are input into the computer.

There are variations of the above representation. If G is a multigraph, then we usually let a_{ij} denote the number of edges $\{v_i, v_j\}$. Moreover, if G is a weighted graph, then we may let a_{ij} denote the weight of the edge $\{v_i, v_j\}$.

Linked Representation of a Graph G

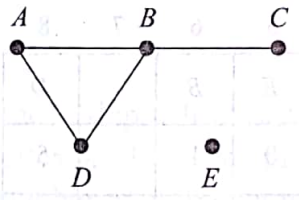
Let G be a graph with m vertices. The representation of G in memory by its adjacency matrix A has a number of major drawbacks. First of all it may be difficult to insert or delete vertices in G. The reason is that the size of A may need to be changed and the vertices may need to be reordered, so there may be many, many changes in the matrix A. Furthermore, suppose the number of edges is $O(m)$ or even $O(m \log m)$, that is, suppose G is sparse. Then the matrix A will contain many zeros; hence a great deal of memory space will be wasted. Accordingly, when G is sparse, G is usually represented in memory by some type of *linked representation*, also called an *adjacency structure*, which is described below by means of an example.

Consider the graph G in Fig. 8.28(a). Observe that G may be equivalently defined by the table in Fig. 8.26(b) which shows each vertex in G followed by its *adjacency list*, i.e. its list of adjacent vertices (*neighbors*). Here the symbol $\varnothing$ denotes an empty list. This table may also be presented in the compact form

$$G = [A{:}B, D;\quad B{:}A, C, D;\quad C{:}B;\quad D{:}A, B;\quad E{:}\varnothing]$$

where a colon ":" separates a vertex from its list of neighbors, and a semicolon ";" separates the different lists.

Remark: Observe that each edge of a graph G is represented twice in an adjacency structure; that is, any edge, say $\{A, B\}$, is represented by B in the adjacency list of A, and also by A in the adjacency list of B. The graph G in Fig. 8.28(a) has four edges, and so there must be 8 vertices in the adjacency lists. On the other hand, each vertex in an adjacency list corresponds to a unique edge in the graph G.



(a)

Vertex	Adjacency list
A	B, D
B	A, C, D
C	B
D	A, B
E	$\varnothing$

(b)

Fig. 8.28

The *linked representation* of a graph G, which maintains G in memory by using its adjacency lists, will normally contain two files (or sets of records), one called the Vertex File and the other called the Edge File, as follows.

(a) ***Vertex File:*** The Vertex File will contain the list of vertices of the graph G usually maintained by an array or by a linked list. Each recordof the Vertex File will have the form

VERTEX	NEXT-V	PTR

Here VERTEX will be the name of the vertex, NEXT-V points to the next vertex in the list of vertices in the Vertex File when the vertices are maintained by a linked list, and PTR will point to the first element in the adjacency list of the vertex appearing in the Edge File. The shaded area indicates that there may be other information in the record corresponding to the vertex.

(b) ***Edge File:*** The Edge File contains the edges of the graph G. Specifically, the Edge File will contain all the adjacency lists of G where each list is maintained in memory by a linked list. Each record of the Edge File will correspond to a vertex in an adjacency list and hence, indirectly, to an edge of G. The record will usually have the form

Here:

1. EDGE will be the name of the edge (if it has one).
2. ADJ points to the location of the vertex in the Vertex File.
3. NEXT points to the location of the next vertex in the adjacency list.

We emphasize that each edge is represented twice in the Edge File, but each record of the file corresponds to a unique edge. The shaded area indicated that there may be other information in the record corresponding to the edge.

Figure 8.29 shows how the graph G in Fig. 8.28(a) may appear in memory. Here the vertices of G are maintained in memory by a linked list using the variable START to point to the first vertex. (Alternatively, one could use a linear array for the list of vertices, and then NEXT-V would not be required.) Note that the field EDGE is not needed here since the edges have no name. Figure 8.29 also shows, with the arrows, the adjacency list $[D, C, A]$ of the vertex B.

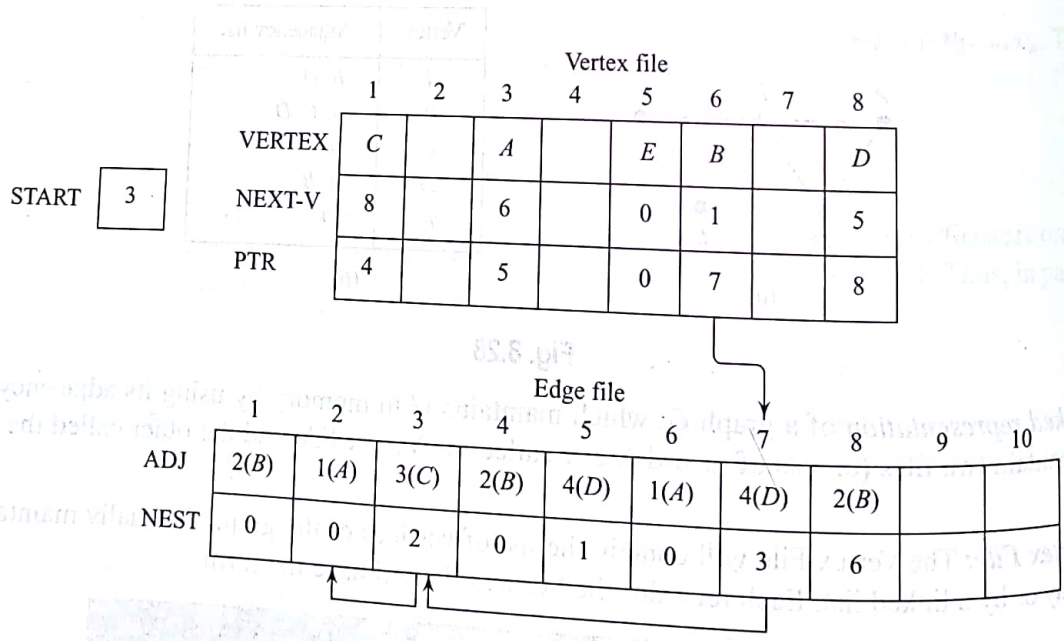
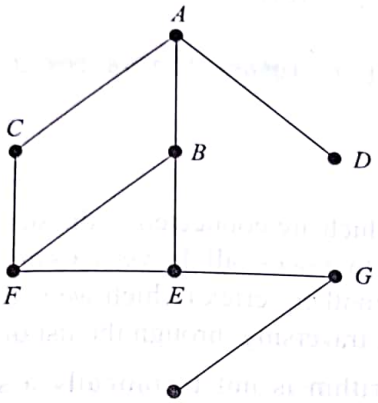


Fig. 8.29

8.12 GRAPH ALGORITHMS

This section discusses two important graph algorithms which systematically examine the vertices and edges of a graph G. One is called a *depth-first search* (DFS) and the other is called a *breadth-first search* (BFS). Other graph algorithms will be discussed in the next chapter in connection with directed graphs. Any particular graph algorithm may depend on the way G is maintained in memory. Here we assume G is maintained in memory by its adjacency structure. Out test graph G with its adjacency structure appears in Fig. 8.28.



(a)

Vertex	Adjacency list
A	B, C, D
B	A, E, F
C	A, F
D	A
E	B, F, G
F	B, C, E
G	E, H
H	G

(b)

Fig. 8.30

During the execution of our algorithms, each vertex (node) N of G will be in one of three states, called the *status* of N, as follows:

STATUS = 1:	(Ready state)	The initial state of the vertex N.
STATUS = 2:	(Waiting state)	The vertex N is on a (waiting) list, waiting to be processed.
STATUS = 3:	(Processed state)	The vertex N has been processed.

The waiting list for the depth-first search will be a (modified) STACK, whereas the waiting list for the breadth-first search will be a QUEUE.

Depth-first Search

The general idea behind a depth-first search beginning at a starting vertex A is as follows. First we process the starting vertex A. Then we process each vertex N along with a path P which begins at A; that is, we process a neighbor of A, then a neighbor of a neighbor of A, and so on. After coming to a "dead end", that is, to a vertex with no unprocessed neighbor, we backtrack on the path P until we can continue along another path P', and so on. The backtracking is accomplished by using a STACK to hold the initial vertices of future possible paths. We also need a field STATUS which tells us the current status of any vertex so that no vertex is processed more than once. The algorithm follows.

Algorithm 8.16A (Depth-first Search): This algorithm executes a depth-first search on a graph G beginning with a starting vertex A.

Step 1. Initialize all vertices to the ready state (STATUS = 1).

Step 2. Push the starting vertex A onto STACK and change the status of A to the waiting state (STATUS = 2).

Example 8.4

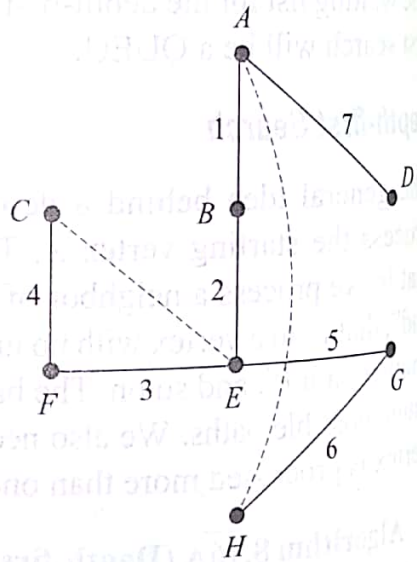
Suppose the DFS Algorithm 8.16A is applied to the graph in Fig. 8.28. The vertices are processed in the following order:

A, B, E, F, C, G, H, D

Specifically, Fig. 8.31(*a*) shows the sequence of waiting lists in STACK and the vertices being processed. We have used the slash/to indicate that a vertex is deleted from the waiting list. Each vertex, excluding *A*, comes from an adjacency list and hence corresponds to an edge of the graph. These edges form a spanning tree of *G* which is pictured in Fig. 8.31(*b*). The numbers indicate the order that the edges are added to the tree, and the dashed lines indicate backtracking.

Vertex	STACK
	A
A	*B, C. D*
B	*E, F, C, D*
E	*F, G, F C, D*
F	*C, G, C, D*
C	*G, D*
G	*H, D*
H	*D*
D	

(*a*)



(*b*)

Fig. 8.31

Breadth-first Search

The general idea behind a breadth-first search beginning at a starting vertex A is as follows. First we process the starting vertex A. Then we process all the neighbors of A. Then we process all the neighbors of neighbors of A. And so on. Naturally we need to keep track of the neighbors of a vertex, and we need to guarantee that no vertex is processed twice. This is accomplished by using a QUEUE to hold vertices that are waiting to be processed, and by a field STATUS which tells us the current status of a vertex. The algorithm follows.

Algorithm 8.16B (Breadth-first Search): This algorithm executes a breadth-first search on a graph G beginning with a starting vertex A.

Step 1. Initialize all vertices to the ready state (STATUS = 1).

Step 2. Put the starting vertex A is QUEUE and change the status of A to the waiting state (STATUS = 2).

Step 3. Repeat Steps 4 and 5 until QUEUE is empty.

Step 4. Remove the front vertex N of QUEUE. Process N, and set STATUS (N) = 3, the processed state.

Step 5. Examine each neighbor J of N.

(a) If STATUS (J) = 1 (ready state), add J to the rear of QUEUE and reset STATUS (J) = 2 (waiting state).

(b) If STATUS (J) = 2 (waiting state) or STATUS (J) = 3 (processed state), ignore the vertex J.

[End of Step 3 loop.]

Step 6. Exit.

Again, the above algorithm will process only those vertices which are connected to the starting vertex A, that is, the connected component including A. Suppose one wants to process all the vertices in the graph G. Then the algorithm must be modified so that it begins again with another vertex (which we call B) that is still in the ready state (STATUS = 1). This vertex B can be obtained by traversing through the list of vertices.

Example 8.5

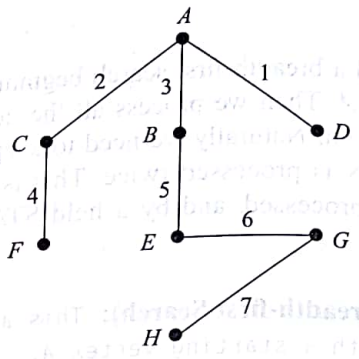
Suppose the BFS Algorithm 8.16B is applied to the graph in Fig. 8.30. The vertices are processed in the following order:

$$A,\quad D,\quad C,\quad B,\quad F,\quad E,\quad G,\quad H$$

Specifically, Fig. 8.32(a) shows the sequence of waiting lists in QUEUE and the vertices being processed. Again, each vertex, excluding A, comes from an adjacency list and hence corresponds to an edge of the graph. These edges form a spanning tree of G which is pictured in Fig. 8.32(b). Again, the numbers indicate the order the edges are added to the tree. Observe that this spanning tree is different from the one in Fig. 8.27(b) which came from a depth-first search.

Vertex	QUEUE
	A
A	B, C. D
D	B,C
C	F, B
B	E, F
F	E
E	G
G	H
H	

(a)



(b)

Fig. 8.32

SOLVED PROBLEMS

GRAPH TERMINOLOGY

8.1 Consider Fig. 8.33(*a*) Describe formally the graph G in the diagram, that is, find the set $V(G)$ of vertices of G and the set $E(G)$ of edges of G. (b) Find the degree of each vertex and verify Theorem 8.2 for this graph.

(a) There are five vertices so $V(G) = \{A, B, C, D, E\}$. There are seven pairs $\{x, y\}$ of vertices where the vertex s is connected with the vertex y; hence

$$E(G) = [\{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{B, E\}, \{C, D\}, \{C, E\}]$$

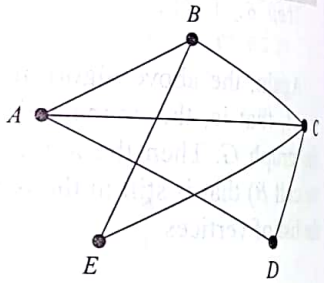


Fig. 8.33

(b) The degree of a vertex is equal to the number of edges to which it belongs; e.g. deg $(A) = 3$ since A belongs to the three edges $\{A, B\}$, $\{A, C\}$, $\{A, D\}$. Similarly,

$$\deg(B) = 3, \deg(C) = 4, \deg(D) = 2, \deg(E) = 2$$

The sum of the degrees of the vertices is

$$3 + 3 + 4 + 2 + 2 = 14$$

which does equal twice the number of edges.

8.2 Consider the graph G in Fig. 8.34. Find (a) all simple paths from A to F; (b) all trails from A to F; (c) $d(A, F)$; the distance from A to F; (d) diam(G), the diameter of G; (e) all cycles which include vertex A; (f) all cycles in G.