2.6 GRAPHS

2.6.1 Introduction

The first recorded evidence of the use of graphs dates back to 1736, when Leonhard Euler used them to solve the now classical Königsberg bridge problem. In the town of Königsberg (now Kaliningrad) the river Pregel (Pregolya) flows around the island Kneiphof and then divides into two. There are, therefore, four land areas that have this river on their borders (see Figure 2.24(a)). These land areas are interconnected by seven bridges, labeled a to g. The land areas themselves are labeled A to D. The Königsberg bridge problem is to determine whether, starting at one land area, it is possible to walk across all the bridges exactly once in returning to the starting land area. One possible walk: Starting from land area B, walk across bridge a to island A, take bridge e to area D, take bridge g to C, take bridge d to A, take bridge b to B, and take bridge f to D.

This walk does not go across all bridges exactly once, nor does it return to the starting land area B. Euler answered the Königsberg bridge problem in the negative: The people of Königsberg cannot walk across each bridge exactly once and return to the starting point. He solved the problem by representing the land areas as vertices and the bridges as edges in a graph (actually a multigraph) as in Figure 2.24(b). His solution is elegant and applies to all graphs. Defining the *degree* of a vertex to be the number of edges incident to it, Euler showed that there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex if and only if the degree of each vertex is even. A walk that does this is called *Eulerian*. There is no Eulerian walk for the Königsberg bridge problem, as all four vertices are of odd degree.

Since this first application, graphs have been used in a wide variety of applications. Some of these applications are the analysis of electric circuits, finding shortest routes, project planning, identification of chemical compounds, statistical mechanics, genetics, cybernetics, linguistics, social sciences, and so on. Indeed, it might well be said that of all mathematical structures, graphs are the most widely used.

2.6.2 Definitions

A graph G consists of two sets V and E. The set V is a finite, nonempty set of *vertices*. The set E is a set of pairs of vertices; these pairs are called *edges*. The notations $V(G)$ and $E(G)$ represent the sets of vertices and edges, respectively, of graph G. We also write $G = (V, E)$ to represent a graph. In an *undirected graph* the pair of vertices representing any edge is unordered. Thus, the pairs (u, v) and (v, u) represent the same edge. In a *directed graph* each edge is represented by a directed pair $\langle u, v \rangle$; u is the *tail* and v the

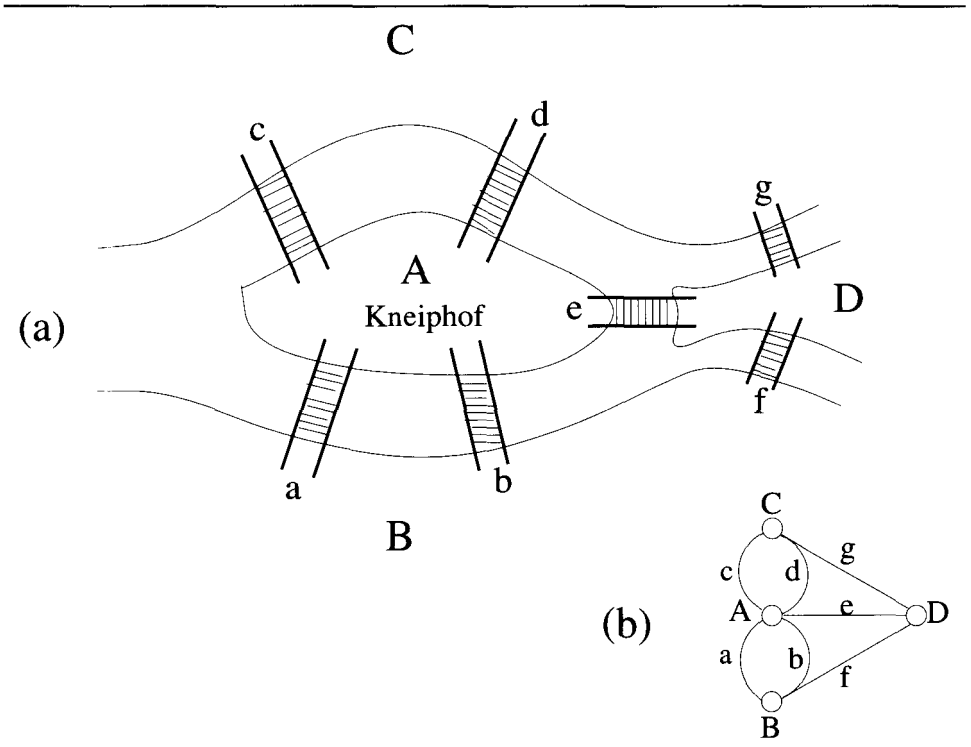
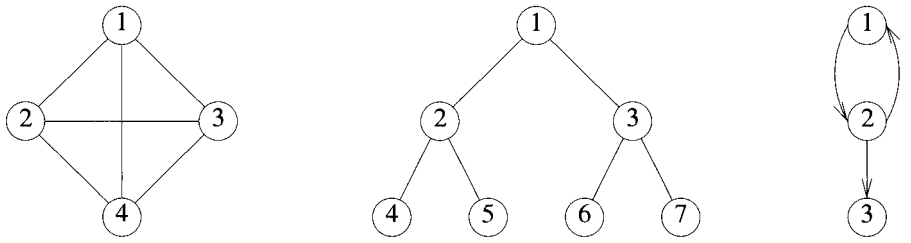


Figure 2.24 Section of the river Pregel in Königsberg and Euler's graph

head of the edge. Therefore, $\langle v, u \rangle$ and $\langle u, v \rangle$ represent two different edges. Figure 2.25 shows three graphs: G_1, G_2, and G_3. The graphs G_1 and G_2 are undirected; G_3 is directed.



(a) G_1 (b) G_2 (c) G_3

Figure 2.25 Three sample graphs

The set representations of these graphs are

$$V(G_1) = \{1,2,3,4\} \qquad E(G_1) = \{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$$
$$V(G_2) = \{1,2,3,4,5,6,7\} \qquad E(G_2) = \{(1,2),(1,3),(2,4),(2,5),(3,6),(3,7)\}$$
$$V(G_3) = \{1,2,3\} \qquad E(G_3) = \{\langle 1,2\rangle, \langle 2,1\rangle, \langle 2,3\rangle\}$$

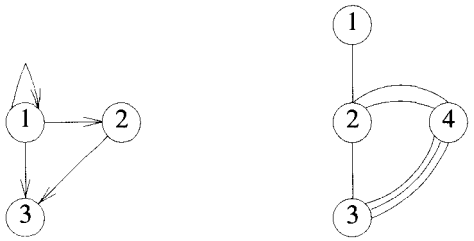
Notice that the edges of a directed graph are drawn with an arrow from the tail to the head. The graph G_2 is a tree; the graphs G_1 and G_3 are not.

Since we define the edges and vertices of a graph as sets, we impose the following restrictions on graphs:

1. A graph may not have an edge from a vertex v back to itself. That is, edges of the form (v, v) and $\langle v, v \rangle$ are not legal. Such edges are known as *self-edges* or *self-loops*. If we permit self-edges, we obtain a data object referred to as a *graph with self-edges*. An example is shown in Figure 2.26(a).

2. A graph may not have multiple occurrences of the same edge. If we remove this restriction, we obtain a data object referred to as a *multigraph* (see Figure 2.26(b)).

The number of distinct unordered pairs (u, v) with $u \neq v$ in a graph with n vertices is $\frac{n(n-1)}{2}$. This is the maximum number of edges in any n-vertex, undirected graph. An n-vertex, undirected graph with exactly $\frac{n(n-1)}{2}$ edges is said to be *complete*. The graph G_1 of Figure 2.25(a) is the complete graph



(a) Graph with a self edge (b) Multigraph

Figure 2.26 Examples of graphlike structures

on four vertices, whereas G_2 and G_3 are not complete graphs. In the case of a directed graph on n vertices, the maximum number of edges is $n(n-1)$.

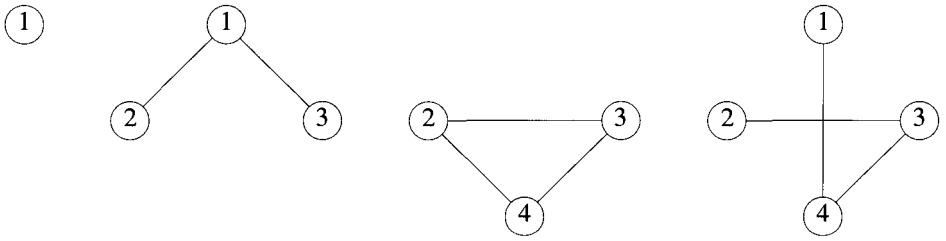
If (u, v) is an edge in $E(G)$, then we say vertices u and v are *adjacent* and edge (u, v) is *incident* on vertices u and v. The vertices adjacent to vertex 2 in G_2 are 4, 5, and 1. The edges incident on vertex 3 in G_2 are (1,3), (3,6), and (3,7). If $\langle u, v\rangle$ is a directed edge, then vertex u is *adjacent to* v, and v is *adjacent from* u. The edge $\langle u, v\rangle$ is incident to u and v. In G_3, the edges incident to vertex 2 are $\langle 1,2\rangle$, $\langle 2,1\rangle$, and $\langle 2,3\rangle$.

A *subgraph* of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. Figure 2.27 shows some of the subgraphs of G_1 and G_3.

A *path* from vertex u to vertex v in graph G is a sequence of vertices $u, i_1, i_2, \ldots, i_k, v$, such that $(u, i_1), (i_1, i_2), \ldots, (i_k, v)$ are edges in $E(G)$. If G' is directed, then the path consists of the edges $\langle u, i_1\rangle, \langle i_1, i_2\rangle, \ldots, \langle i_k, v\rangle$ in $E(G')$. The *length* of a path is the number of edges on it. A *simple path* is a path in which all vertices except possibly the first and last are distinct. A path such as (1,2), (2,4), (4,3), is also written as 1, 2, 4, 3. Paths 1, 2, 4, 3 and 1, 2, 4, 2 of G_1 are both of length 3. The first is a simple path; the second is not. The path 1, 2, 3 is a simple directed path in G_3, but 1, 2, 3, 2 is not a path in G_3, as the edge $\langle 3,2\rangle$ is not in $E(G_3)$.

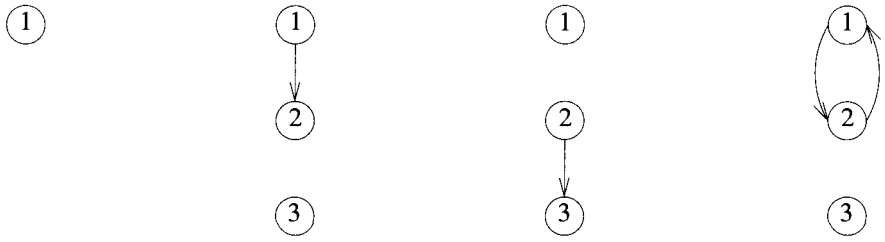
A *cycle* is a simple path in which the first and last vertices are the same. The path 1, 2, 3, 1 is a cycle in G_1 and 1, 2, 1 is a cycle in G_3. For directed graphs we normally add the prefix "directed" to the terms cycle and path.

In an undirected graph G, two vertices u and v are said to be *connected* iff there is a path in G from u to v (since G is undirected, this means there must also be a path from v to u). An undirected graph is said to be connected iff for every pair of distinct vertices u and v in $V(G)$, there is a path from u to v in G. Graphs G_1 and G_2 are connected, whereas G_4 of Figure 2.28 is not.



(i) (ii) (iii) (iv)

(a) Some of the subgraphs of G_1



(i) (ii) (iii) (iv)

(b) Some of the subgraphs of G_3

Figure 2.27 Some subgraphs

A *connected component* (or simply a component) H of an undirected graph is a *maximal* connected subgraph. By "maximal," we mean that G contains no other subgraph that is both connected and properly contains H. G_4 has two components, H_1 and H_2 (see Figure 2.28).

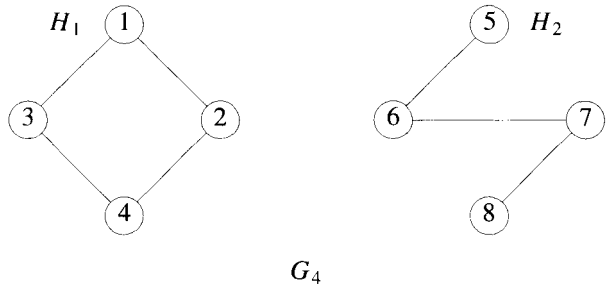


Figure 2.28 A graph with two connected components

A *tree* is a connected acyclic (i.e., has no cycles) graph. A directed graph G is said to be *strongly connected* iff for every pair of distinct vertices u and v in $V(G)$, there is a directed path from u to v and also from v to u. The graph G_3 (repeated in Figure 2.29(a)) is not strongly connected, as there is no path from vertex 3 to 2. A *strongly connected component* is a maximal subgraph that is strongly connected. The graph G_3 has two strongly connected components (see Figure 2.29(b)).

The degree of a vertex is the number of edges incident to that vertex. The degree of vertex 1 in G_1 is 3. If G is a directed graph, we define the *in-degree* of a vertex v to be the number of edges for which v is the head. The *out-degree* is defined to be the number of edges for which v is the tail. Vertex 2 of G_3 has in-degree 1, out-degree 2, and degree 3. If d_i is the degree of vertex i in a graph G with n vertices and e edges, then the number of edges is

$$e = \left(\sum_{i=1}^{n} d_i\right)/2$$

In the remainder of this chapter, we refer to a directed graph as a *digraph*. When we use the term graph, we assume that it is an undirected graph.

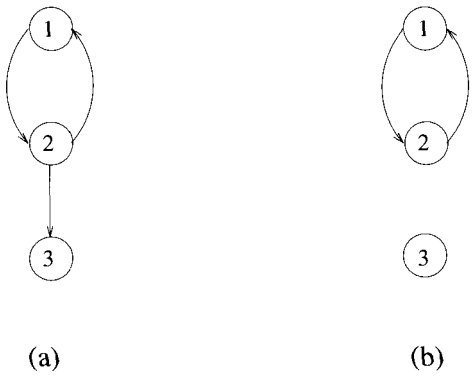


Figure 2.29 A graph and its strongly connected components

2.6.3 Graph Representations

Although several representations for graphs are possible, we study only the three most commonly used: adjacency matrices, adjacency lists, and adjacency multilists. Once again, the choice of a particular representation depends on the application we have in mind and the functions we expect to perform on the graph.

Adjacency Matrix

Let $G = (V, E)$ be a graph with n vertices, $n \geq 1$. The adjacency matrix of G is a two-dimensional $n \times n$ array, say a, with the property that $a[i, j] = 1$ iff the edge (i, j) ($\langle i, j \rangle$ for a directed graph) is in $E(G)$. The element $a[i, j] = 0$ if there is no such edge in G. The adjacency matrices for the graphs G_1, G_3, and G_4 are shown in Figure 2.30. The adjacency matrix for an undirected graph is symmetric, as the edge (i, j) is in $E(G)$ iff the edge (j, i) is also in $E(G)$. The adjacency matrix for a directed graph may not be symmetric (as is the case for G_3). The space needed to represent a graph using its adjacency matrix is n^2 bits. About half this space can be saved in the case of an undirected graph by storing only the upper or lower triangle of the matrix.

From the adjacency matrix, we can readily determine whether there is an edge connecting any two vertices i and j. For an undirected graph the degree of any vertex i is its row sum:

	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

(a) G_1

	1	2	3
1	0	1	0
2	1	0	1
3	0	0	0

(b) G_3

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	0	1	0	0	0	0
3	1	0	0	1	0	0	0	0
4	0	1	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0
6	0	0	0	0	1	0	1	0
7	0	0	0	0	0	1	0	1
8	0	0	0	0	0	0	1	0

(c) G_4

Figure 2.30 Adjacency matrices

$$\sum_{j=1}^{n} a[i,j]$$

For a directed graph the row sum is the out-degree, and the column sum is the in-degree.

Suppose we want to answer a nontrivial question about graphs, such as How many edges are there in G? or Is G connected? Adjacency matrices require at least n^2 time, as $n^2 - n$ entries of the matrix (diagonal entries are zero) have to be examined. When graphs are sparse (i.e., most of the terms in the adjacency matrix are zero), we would expect that the former question could be answered in significantly less time, say $O(e+n)$, where e is the number of edges in G, and $e < \frac{n^2}{2}$. Such a speedup is made possible through the use of a representation in which only the edges that are in G are explicitly stored. This leads to the next representation for graphs, adjacency lists.

Adjacency Lists

In this representation of graphs, the n rows of the adjacency matrix are represented as n linked lists. There is one list for each vertex in G. The nodes in list i represent the vertices that are adjacent from vertex i. Each node has at least two fields: *vertex* and *link*. The *vertex* field contains the indices of the vertices adjacent to vertex i. The adjacency lists for G_1, G_3,

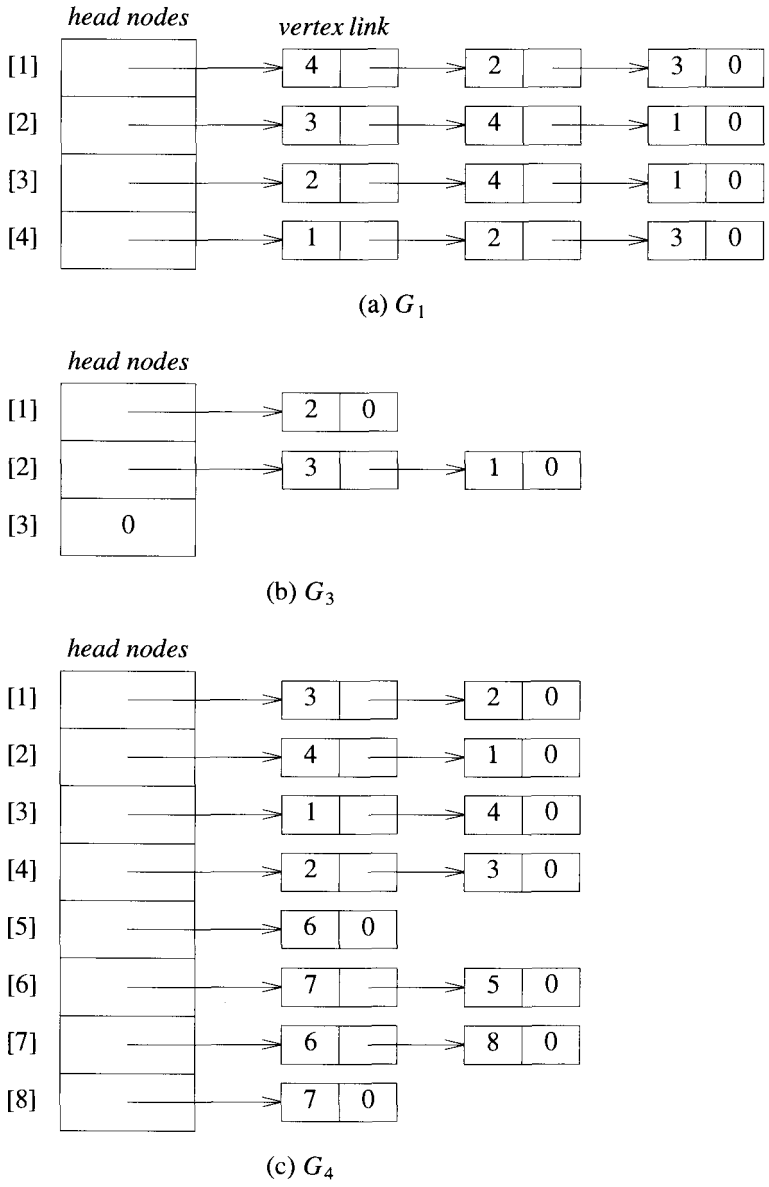


Figure 2.31 Adjacency lists

and G_4 are shown in Figure 2.31. Notice that the vertices in each list are not required to be ordered. Each list has a head node. The head nodes are sequential, and so provide easy random access to the adjacency list for any particular vertex.

For an undirected graph with n vertices and e edges, this representation requires n head nodes and $2e$ list nodes. Each list node has two fields. In terms of the number of bits of storage needed, this count should be multiplied by $\log n$ for the head nodes and $\log n + \log e$ for the list nodes, as it takes $O(\log m)$ bits to represent a number of value m. Often, you can sequentially pack the nodes on the adjacency lists, and thereby eliminate the use of pointers. In this case, an array $node\ [1 : n + 2e + 1]$ can be used. The $node[i]$ gives the starting point of the list for vertex i, $1 \leq i \leq n$, and $node[n + 1]$ is set to $n + 2e + 2$. The vertices adjacent from vertex i are stored in $node[i], \ldots, node[i + 1] - 1$, $1 \leq i \leq n$. Figure 2.32 shows the sequential representation for the graph G_4 of Figure 2.28.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
10	12	14	16	18	19	21	23	24	3	2	4	1	1	4	2	3	6	7	5	6	8	7

Figure 2.32 Sequential representation of graph G_4:
Array $node[1 : n + 2e + 1]$

The degree of any vertex in an undirected graph can be determined by just counting the number of nodes in its adjacency list. So, the number of edges in G can be determined in $O(n + e)$ time.

For a digraph, the number of list nodes is only e. The out-degree of any vertex can be determined by counting the number of nodes on its adjacency list. Hence, the total number of edges in G can be determined in $O(n + e)$ time. Determining the in-degree of a vertex is a little more complex. If there is a need to access repeatedly all vertices adjacent to another vertex, then it may be worth the effort to keep another set of lists in addition to the adjacency lists. This set of lists, called *inverse adjacency lists*, contains one list for each vertex. Each list contains a node for each vertex adjacent to the vertex it represents (see Figure 2.33).

One can also adopt a simpler version of the list structure in which each node has four fields and represents one edge. The node structure is

tail	head	column link for head	row link for tail

Figure 2.34 shows the resulting structure for the graph G_3 of Figure 2.25(c). The head nodes are stored sequentially.

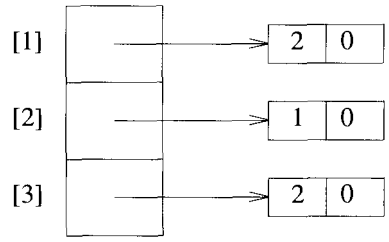


Figure 2.33 Inverse adjacency lists for G_3 of Figure 2.25(c)

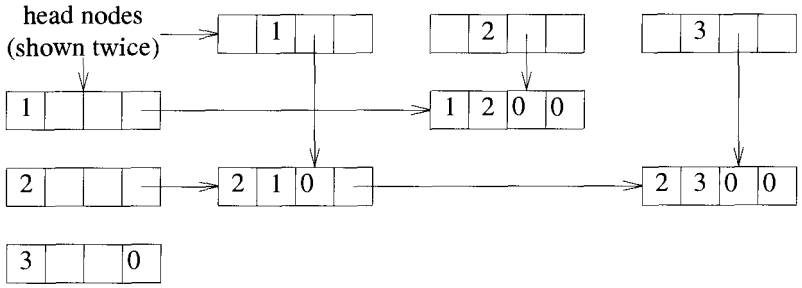


Figure 2.34 Orthogonal list representation for G_3 of Figure 2.25(c)

Adjacency Multilists

In the adjacency-list representation of an undirected graph, each edge (u, v) is represented by two entries, one on the list for u and the other on the list for v. In some applications it is necessary to be able to determine the second entry for a particular edge and mark that edge as having been examined. This can be accomplished easily if the adjacency lists are maintained as multilists (i.e., lists in which nodes can be shared among several lists). For each edge there is exactly one node, but this node is in two lists (i.e., the adjacency lists for each of the two nodes to which it is incident). The new node structure is

m	$vertex1$	$vertex2$	$list1$	$list2$

where m is a one-bit mark field that can be used to indicate whether the edge has been examined. The storage requirements are the same as for normal adjacency lists, except for the addition of the mark bit m. Figure 2.35 shows the adjacency multilists for G_1 of Figure 2.25(a).

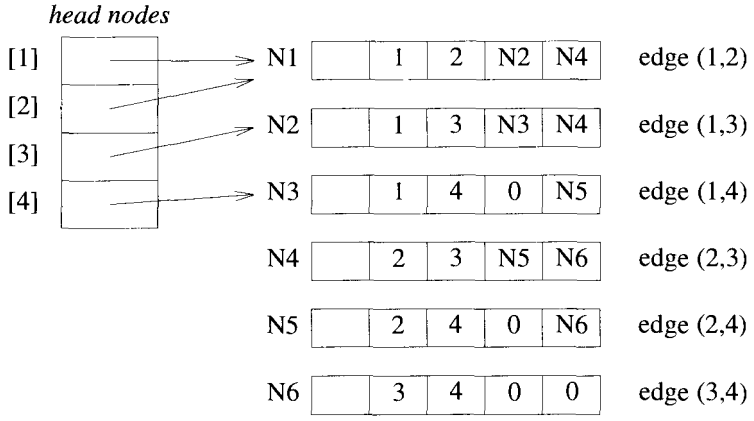


Figure 2.35 Adjacency multilists for G_1 of Figure 2.25(a)

Weighted Edges

In many applications, the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one vertex to an adjacent vertex. In these applications, the adjacency matrix entries $a[i, j]$ keep this information too. When adjacency lists are used, the weight information can be kept in the list nodes by including an additional field, *weight*. A graph with weighted edges is called a *network*.

EXERCISES

1. Does the multigraph of Figure 2.36 have an Eulerian walk? If so, find one.

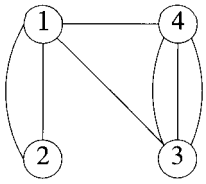


Figure 2.36 A multigraph

2. For the digraph of Figure 2.37 obtain
   (a) the in-degree and out-degree of each vertex
   (b) its adjacency-matrix representation
   (c) its adjacency-list representation
   (d) its adjacency-multilist representation
   (e) its strongly connected components

3. Devise a suitable representation for graphs so that they can be stored on disk. Write an algorithm that reads in such a graph and creates its adjacency matrix. Write another algorithm that creates the adjacency lists from the disk input.

4. Draw the complete undirected graphs on one, two, three, four, and five vertices. Prove that the number of edges in an n-vertex complete graph is $\frac{n(n-1)}{2}$.

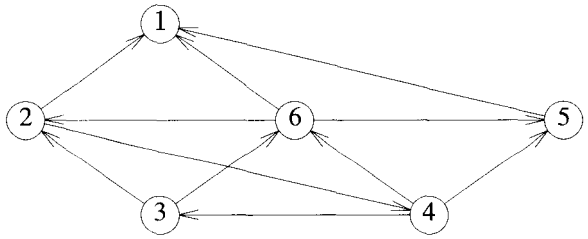


Figure 2.37 A digraph

5. Is the directed graph of Figure 2.38 strongly connected? List all the simple paths.

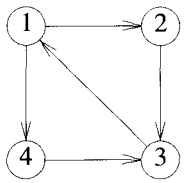


Figure 2.38 A directed graph

6. Obtain the adjacency-matrix, adjacency-list, and adjacency-multilist representations of the graph of Figure 2.38.

7. Show that the sum of the degrees of the vertices of an undirected graph is twice the number of edges.

8. Prove or disprove:

   If $G(V, E)$ is a finite directed graph such that the out-degree of each vertex is at least one, then there is a directed cycle in G.

9. (a) Let G be a connected, undirected graph on n vertices. Show that G must have at least $n - 1$ edges and that all connected, undirected graphs with $n - 1$ edges are trees.

(b) What is the minimum number of edges in a strongly connected digraph with n vertices? What form do such digraphs have?

10. For an undirected graph G with n vertices, prove that the following are equivalent:

    (a) G is a tree.

    (b) G is connected, but if any edge is removed, the resulting graph is not connected.

    (c) For any two distinct vertices $u \in V(G)$ and $v \in V(G)$, there is exactly one simple path from u to v.

    (d) G contains no cycles and has $n - 1$ edges.

11. Write an algorithm to input the number of vertices in an undirected graph and its edges one by one and to set up the linked adjacency-list representation of the graph. You may assume that no edge is input twice. What is the run time of your procedure as a function of the number of vertices and the number of edges?

12. Do the preceding exercise but now set up the multilist representation.

13. Let G be an undirected, connected graph with at least one vertex of odd degree. Show that G contains no Eulerian walk.

2.7 REFERENCES AND READINGS

A wide-ranging examination of data structures and their efficient implementation can be found in the following:

Fundamentals of Data Structures in C++, by E. Horowitz, S. Sahni, and D. Mehta, Computer Science Press, 1995.

Data Structures and Algorithms 1: Sorting and Searching, by K. Mehlhorn, Springer-Verlag, 1984.

Introduction to Algorithms: A Creative Approach, by U. Manber, Addison-Wesley, 1989.

Handbook of Algorithms and Data Structures, second edition, by G. H. Gonnet and R. Baeza-Yates, Addison-Wesley, 1991.

Proof of Lemma 2.4 can be found in "Worst-case analysis of set union algorithms," by R. Tarjan and J. Van Leeuwen, *Journal of the ACM* 31; no. 2 (1984): 245–281.