Graphs

12

THIS CHAPTER introduces important mathematical structures called graphs that have applications in subjects as diverse as sociology, chemistry, geography, and electrical engineering. We shall study methods to represent graphs with the data structures available to us and shall construct several important algorithms for processing graphs. Finally, we look at the possibility of using graphs themselves as data structures.

12.1 Mathematical Background 555
- 12.1.1 Definitions and Examples 555
- 12.1.2 Undirected Graphs 556
- 12.1.3 Directed Graphs 556

12.2 Computer Representation 557
- 12.2.1 The Set Representation 557
- 12.2.2 Adjacency Lists 559
- 12.2.3 Information Fields 560

12.3 Graph Traversal 560
- 12.3.1 Methods 560
- 12.3.2 Depth-First Algorithm 561
- 12.3.3 Breadth-First Algorithm 563

12.4 Topological Sorting 564
- 12.4.1 The Problem 564
- 12.4.2 Depth-First Algorithm 566
- 12.4.3 Breadth-First Algorithm 567

12.5 A Greedy Algorithm: Shortest Paths 568
- 12.5.1 The Problem 568
- 12.5.2 Method 569
- 12.5.3 Example 570
- 12.5.4 Implementation 570

12.6 Minimal Spanning Trees 573
- 12.6.1 The Problem 573
- 12.6.2 Method 574
- 12.6.3 Implementation 575
- 12.6.4 Verification of Prim's Algorithm 578

12.7 Graphs as Data Structures 579

Pointers and Pitfalls 582
Review Questions 582
References for Further Study 583

Definition

A ***digraph*** G consists of a set V, called the ***vertices*** of G, and, for all $v \in V$, a subset A_v of V, called the set of vertices ***adjacent*** to v.

From the subsets A_v we can reconstruct the edges as ordered pairs by the rule: The pair (v, w) is an edge if and only if $w \in A_v$. It is easier, however, to work with sets of vertices than with pairs. This new definition, moreover, works for both directed and undirected graphs. The graph is undirected means that it satisfies the following symmetry property: $w \in A_v$ implies $v \in A_w$ for all v, $w \in V$. This property can be restated in less formal terms: It means that an undirected edge between v and w can be regarded as made up of two directed edges, one from v to w and the other from w to v.

1. Implementation of Sets

sets as Boolean arrays

There are two general ways for us to implement sets of vertices in data structures and algorithms. One way is to represent the set as a *list* of its elements; this method we shall study presently. The other implementation, often called a ***bit string***, keeps a Boolean value for each potential element of the set to indicate whether or not it is in the set. For simplicity, we shall consider that the potential elements of a set are indexed with the integers from 0 to max_set − 1, where max_set denotes the maximum number of elements that we shall allow. This latter strategy is easily implemented either with the standard template library class std::bitset<max_set> or with our own class template that uses a template parameter to give the maximal number of potential members of a set.

```
template <int max_set>
struct Set {
  bool is_element[max_set];
};
```

We can now fully specify a first representation of a graph:

first implementation: sets

```
template <int max_size>
class Digraph {
   int count;                   //  Number of vertices, at most max_size.
   Set<max_size> neighbors[max_size];
};
```

In this implementation, the vertices are identified with the integers from 0 to count − 1. If v is such an integer, the array entry neighbors[v] is the set of all vertices adjacent to the vertex v.

2. Adjacency Tables

sets as arrays

In the foregoing implementation, the structure Set is essentially implemented as an array of **bool** entries. Each entry indicates whether or not the corresponding vertex is a member of the set. If we substitute this array for a set of neighbors, we find that the array neighbors in the definition of **class** Graph can be changed to an array of arrays, that is, to a two-dimensional array, as follows:

2. Linked Implementation

Greatest flexibility is obtained by using linked objects for both the vertices and the adjacency lists. This implementation is illustrated in part (a) of Figure 12.5 and results in a definition such as the following:

fourth implementation: linked vertices and edges

```
class Edge;                       //  forward declaration
class Vertex {
   Edge *first_edge;              //  start of the adjacency list
   Vertex *next_vertex;           //  next vertex on the linked list
};
class Edge {
   Vertex *end_point;             //  vertex to which the edge points
   Edge *next_edge;               //  next edge on the adjacency list
};
class Digraph {
   Vertex *first_vertex;          //  header for the list of vertices
};
```

12.2.3 Information Fields

Many applications of graphs require not only the adjacency information specified in the various representations but also further information specific to each vertex or each edge. In the linked representations, this information can be included as additional members within appropriate records, and, in the contiguous representations, it can be included by making array entries into records. An especially important case is that of a ***network***, which is defined as a graph in which a numerical ***weight*** is attached to each edge. For many algorithms on networks, the best representation is an adjacency table, where the entries are the weights rather than Boolean values. We shall return to this topic later in the chapter.

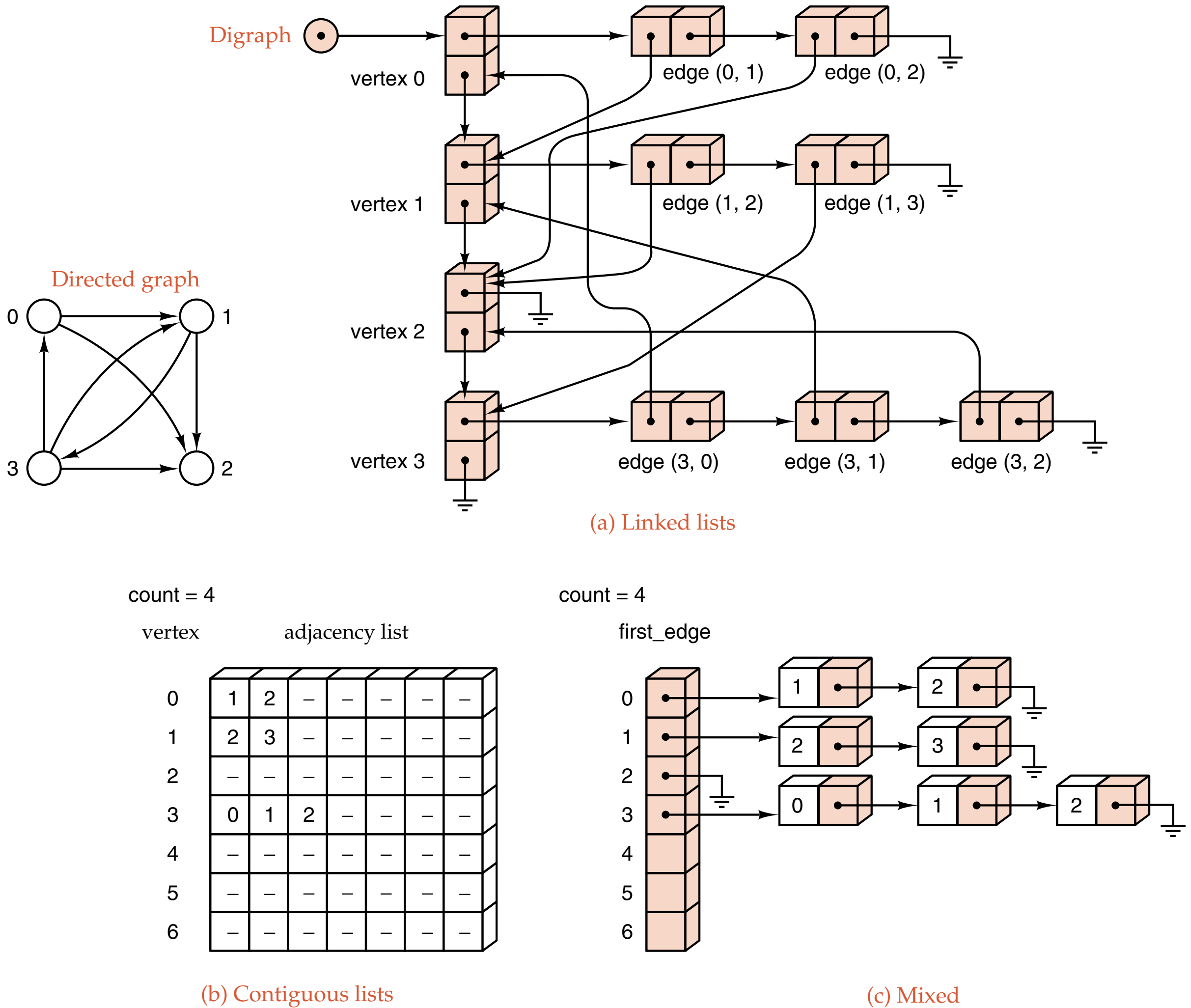
networks, weights

12.3 GRAPH TRAVERSAL

12.3.1 Methods

In many problems, we wish to investigate all the vertices in a graph in some systematic order, just as with binary trees, where we developed several systematic traversal methods. In tree traversal, we had a root vertex with which we generally started; in graphs, we often do not have any one vertex singled out as special, and therefore the traversal may start at an arbitrary vertex. Although there are many possible orders for visiting the vertices of the graph, two methods are of particular importance. ***Depth-first traversal*** of a graph is roughly analogous to preorder traversal of an ordered tree. Suppose that the traversal has just visited a vertex v, and let $w_1, w_2, \ldots, w_k$ be the vertices adjacent to v. Then we shall next visit w_1 and keep $w_2, \ldots, w_k$ waiting. After visiting w_1, we traverse all the vertices to which

depth-first



vertex							
0	1	2	–	–	–	–	–
1	2	3	–	–	–	–	–
2	–	–	–	–	–	–	–
3	0	1	2	–	–	–	–
4	–	–	–	–	–	–	–
5	–	–	–	–	–	–	–
6	–	–	–	–	–	–	–

Figure 12.5. Implementations of a graph with lists

breadth-first

it is adjacent before returning to traverse $w_2, \ldots, w_k$. ***Breadth-first traversal*** of a graph is roughly analogous to level-by-level traversal of an ordered tree. If the traversal has just visited a vertex v, then it next visits *all* the vertices adjacent to v, putting the vertices adjacent to these in a waiting list to be traversed after all vertices adjacent to v have been visited. Figure 12.6 shows the order of visiting the vertices of one graph under both depth-first and breadth-first traversals.

12.3.2 Depth-First Algorithm

Depth-first traversal is naturally formulated as a recursive algorithm. Its action,

```
            while (q.serve(w) != fail)
               if (visited[w] == false) {
                  visited[w] = true;
                  (*visit)(w);
                  for (all x adjacent to w)
                     q.append(x);
               }
         }
      }
```

12.4 TOPOLOGICAL SORTING

12.4.1 The Problem

topological order

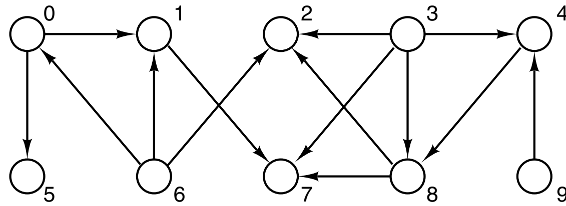
If G is a directed graph with no directed cycles, then a ***topological order*** for G is a sequential listing of all the vertices in G such that, for all vertices $v, w \in G$, if there is an edge from v to w, then v precedes w in the sequential listing. Throughout this section, we shall consider only directed graphs that have no directed cycles. The term ***acyclic*** is often used to mean that a graph has no cycles.

applications

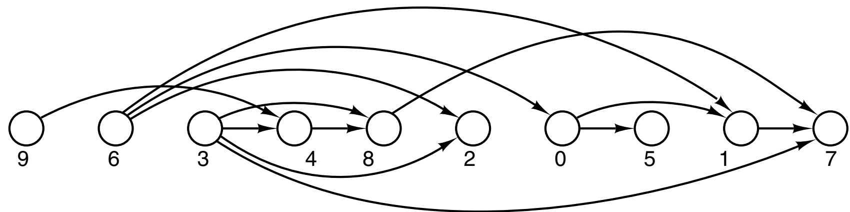
Such graphs arise in many problems. As a first application of topological order, consider the courses available at a university as the vertices of a directed graph, where there is an edge from one course to another if the first is a prerequisite for the second. A topological order is then a listing of all the courses such that all prerequisites for a course appear before it does. A second example is a glossary of technical terms that is ordered so that no term is used in a definition before it is itself defined. Similarly, the author of a textbook uses a topological order for the topics in the book. Two different topological orders of a directed graph are shown in Figure 12.7. As an example of algorithms for graph traversal, we shall develop functions that produce a topological ordering of the vertices of a directed graph that has no cycles. We shall develop two methods: first, using depth-first traversal, and, then, using breadth-first traversal. Both methods apply to an object of a **class** Digraph that uses the list based implementation. Thus we shall assume the following class specification.

graph representation

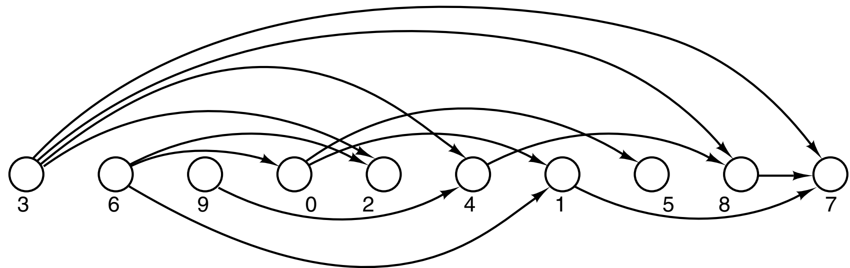
```
typedef int Vertex;

template <int graph_size>
class Digraph {
public:
   Digraph();
   void read();
   void write();
//    methods to do a topological sort
   void depth_sort(List<Vertex> &topological_order);
   void breadth_sort(List<Vertex> &topological_order);
```



Directed graph with no directed cycles



Depth-first ordering



Breadth-first ordering

Figure 12.7. Topological orderings of a directed graph

```
private:
   int count;
   List <Vertex> neighbors[graph_size];
   void recursive_depth_sort(Vertex v, bool visited[ ],
                                    List<Vertex> &topological_order);
};
```

The auxiliary member function recursive_depth_sort will be used by the method depth_sort. Both sorting methods should create a list giving a topological order of the vertices.

performance To estimate the running time of this function, we note that the main loop is executed $n - 1$ times, where n is the number of vertices, and within the main loop are two other loops, each executed $n - 1$ times, so these loops contribute a multiple of $(n - 1)^2$ operations. Statements done outside the loops contribute only $O(n)$, so the running time of the algorithm is $O(n^2)$.

12.6 MINIMAL SPANNING TREES

12.6.1 The Problem

The shortest-path algorithm of the last section applies without change to networks and graphs as well as to directed networks and digraphs. For example, in Figure 12.11 we illustrate the result of its application to find shortest paths (shown in color) from a source vertex, labelled 0, to the other vertices of a network.

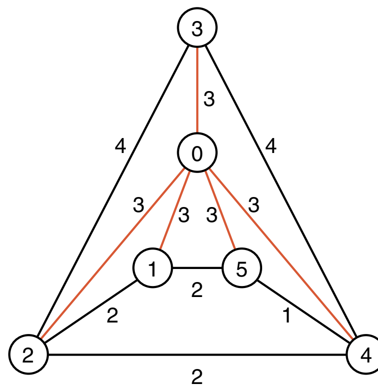


Figure 12.11. Finding shortest paths in a network.

If the original network is based on a connected graph G, then the shortest paths from a particular source vertex link that source to all other vertices in G. Therefore, as we can see in Figure 12.11, if we combine the computed shortest paths together, we obtain a tree that links up all the vertices of G. In other words we obtain a connected tree that is build up out of all the vertices and some of the edges of G. We shall refer to any such tree as a ***spanning tree*** of G. As in the previous section, we can think of a network on a graph G as a map of airline routes, with each vertex representing a city and the weight on each edge the cost of flying from one city to the second. A spanning tree of G represents a minimal set of routes that will allow passengers to complete any conceivable trip between cities. Of course, passengers will frequently have to use several flights to complete journeys. However, this inconvenience for the passengers is offset by lower costs for the airline and cheaper tickets. In fact, spanning trees have been commonly adopted by airlines as hub-spoke route systems. If we imagine the network of Figure 12.11 as representing a hub-spoke system, then the source vertex corresponds to the hub airport, and

the paths emerging from this vertex are the spoke routes. It is important for an airline running a hub-spoke system to minimize its expenses by choosing a system of routes whose costs have a minimal sum. For example, in Figure 12.12, where a pair of spanning trees of a network are illustrated with colored edges, an airline would prefer the second spanning tree, because the sum of its labels is smaller. To model an optimal hub-spoke system, we make the following definition:

Definition *minimal spanning tree*

A ***minimal spanning tree*** of a connected network is a spanning tree such that the sum of the weights of its edges is as small as possible.

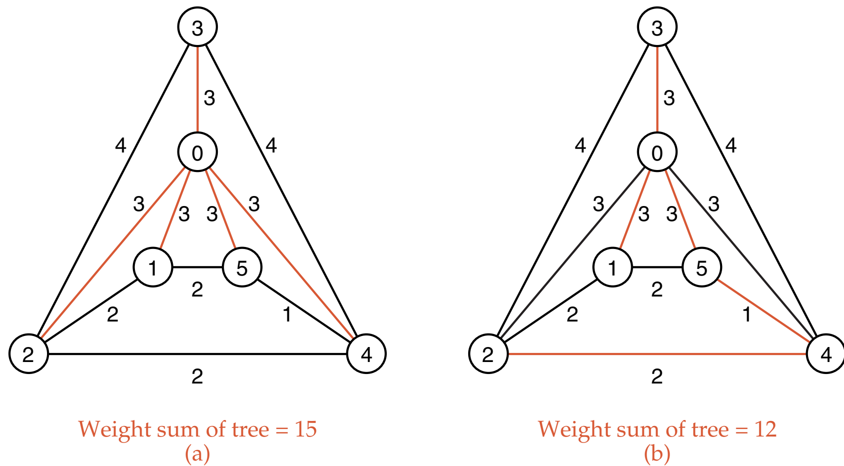


Figure 12.12. Two spanning trees in a network.

Although it is not difficult to compare the two spanning trees of Figure 12.12, it is much harder to see whether there are any other, cheaper, spanning trees. Our problem is to devise a method that determines a minimal spanning tree of a connected network.

12.6.2 Method

We already know an algorithm for finding a spanning tree of a connected graph, since the shortest path algorithm will do this. It turns out that we can make a small change to our shortest path algorithm to obtain a method, first implemented in 1957 by R. C. Prim, that finds a minimal spanning tree.

We start out by choosing a starting vertex, that we call source, and, as we proceed through the method, we keep a set X of those vertices whose paths to the source in the minimal spanning tree that we are building have been found. We also need to keep track of the set Y of edges that link the vertices in X in the tree under construction. Thus, over the course of time, we can visualize the vertices in X and edges in Y as making up a small tree that grows to become our final spanning tree. Initially, source is the only vertex in X, and the edge set Y is empty. At each step,

we add an additional vertex to X: this vertex is chosen so that an edge back to X has as small as possible a weight. This minimal edge back to X is added to Y.

It is quite tricky to prove that Prim's algorithm does give a minimal spanning tree, and we shall postpone this verification until the end of this section. However, we can understand the selection of the new vertex that we add to X and the new edge that we add to Y by noting that eventually we must incorporate an edge linking X to the other vertices of our network into the spanning tree that we are building. The edge chosen by Prim's criterion provides the cheapest way to accomplish this linking, and so according to the greedy criterion, we should use it. When we come to implement Prim's algorithm, we shall maintain a list of vertices that belong to X as the entries of a Boolean array component. It is convenient for us to store the edges in Y as the edges of a graph that will grow to give the output tree from our program.

neighbor table
distance table

We shall maintain an auxiliary table neighbor that gives, for each vertex v, the vertex of X whose edge to v has minimal cost. It is convenient to maintain a second table distance that records these minimal costs. If a vertex v is not joined by an edge to X we shall record its distance as the value infinity. The table neighbor is initialized by setting neighbor[v] to source for all vertices v, and distance is initialized by setting distance[v] to the weight of the edge from source to v if it exists and to infinity if not.

maintain the invariant

To determine what vertex to add to X at each step, we choose the vertex v with the smallest value recorded in the table distance, such that v is not already in X. After this we must update our tables to reflect the change that we have made to X. We do this by checking, for each vertex w not in X, whether there is an edge linking v and w, and if so, whether this edge has a weight less than distance[w]. In case there is an edge (v, w) with this property, we reset neighbor[w] to v and distance[w] to the weight of the edge.

For example, let us work through the network shown in part (a) of Figure 12.13. The initial situation is shown in part (b): The set X (colored vertices) consists of source alone, and for each vertex w the vertex neighbor[w] is visualized by following any arrow emerging from w in the diagram. (The value of distance[w] is the weight of the corresponding edge.) The distance to vertex 1 is among the shortest, so 1 is added to X in part (c), and the entries in tables distance and neighbor are updated for vertices 2 and 5. The other entries in these tables remain unchanged. Among the next closest vertices to X is vertex 2, and it is added in part (d), which also shows the effect of updating the distance and neighbor tables. The final three steps, are shown in parts (e), (f), and (g).

12.6.3 Implementation

To implement Prim's algorithm, we must begin by choosing a C++ **class** to represent a network. The similarity of the algorithm to the shortest path algorithm of the last section suggests that we should base a **class** Network on our earlier **class** Digraph.

REFERENCES FOR FURTHER STUDY

The study of graphs and algorithms for their processing is a large subject and one that involves both mathematics and computing science. Three books, each of which contains many interesting algorithms, are

R. E. TARJAN, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, 1983, 131 pages.

SHIMON EVEN, *Graph Algorithms*, Computer Science Press, Rockville, Md., 1979, 249 pages.

E. M. REINGOLD, J. NIEVERGELT, N. DEO, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, N. J., 1977, 433 pages.

The original reference for the greedy algorithm determining the shortest paths in a graph is

E. W. DIJKSTRA, "A note on two problems in connexion with graphs," *Numerische Mathematik* 1 (1959), 269–271.

Prim's algorithm for minimal spanning trees is reported in

R. C. PRIM, "Shortest Connection Networks and Some Generalizations," *Bell System Technical Journal* 36 (1957), 1389–1401.

Kruskal's algorithm is described in

J. B. KRUSKAL, "On the Shortest Spanning Tree of a Graph and the Traveling Salesman Problem." *Proceedings of the American Mathematical Society* 7 (1956), 48–50.

The original reference for Dijkstra's algorithm for minimal spanning trees is

E. W. DIJKSTRA, "Some Theorems on Spanning Subtrees of a Graph," *Indagationes Mathematicae* 28 (1960), 196–199.