8 GRAPHS

OBJECTIVES

After completing this chapter, the reader will be able to understand the following:

- Graphs as one of the most important non-linear data structures
- The representation that models various kinds of graphs
- Some useful graph algorithms

In many application areas such as cartography, sociology, chemistry, geography, mathematics, electrical engineering, and computer science, we often need a representation that reflects an arbitrary relationship among the objects. One of the most powerful and natural solutions that models such a relationship is a *graph*. There are many concrete, practical problems such as electrical circuits, Königsberg's bridges, and Instant Insanity that have been simplified and solved using graphs.

Non-linear data structures are used to represent the data containing a network or hierarchical relationship among the elements. Graphs are one of the most important non-linear data structures. In non-linear data structures, every data element may have more than one predecessor as well as successor. Elements do not form any particular linear sequence. We shall study various representations of graphs and important algorithms for processing them in this chapter.

8.1 INTRODUCTION

The seven bridges of Königsberg is an ancient classic problem. It was creatively solved by the great Swiss mathematician Leonhard Euler in 1736, which laid the foundations of graph theory. Another example is Instant Insanity. It is a puzzle consisting of four cubes where each of the four faces of these cubes is painted with one of the four different colours—red, blue, white, or green. The problem is to stack the cubes, one on the top of the other so that whether the cubes are viewed from front, back, left, or right, one sees all the four colours. Since 331,776 different stack combinations are possible, solving it by hand or by the trial-and-error method is impractical. However, the use of graphs makes it possible to discover a solution in a few minutes!

There are many such problems that can be represented and solved using graphs. Finding an abstract mathematical model of the concrete problem can be a difficult task, which

may require both skill and experience. Some real-world applications of graphs include communication networking, analysis of electrical circuits, activity network, linguistics, and so on.

8.2 GRAPH ABSTRACT DATA TYPE

Graphs as non-linear data structures represent the relationship among data elements, having more than one predecessor and/or successor. A graph G is a collection of nodes (*vertices*) and arcs joining a pair of the nodes (*edges*). Edges between two vertices represent the relationship between them. For finite graphs, V and E are finite. We can denote the graph as $G = (V, E)$.

Let us define the graph ADT. We need to specify both sets of vertices and edges. Basic operations include creating a graph, inserting and deleting a vertex, inserting and deleting an edge, traversing a graph, and a few others.
A graph is a set of vertices and edges $\{V, E\}$ and can be declared as follows:

```
graph
     create()→Graph
     insert_vertex(Graph, v)→Graph
     delete_vertex(Graph, v)→Graph
     insert_edge(Graph, u, v)→Graph
     delete_edge(Graph, u, v)→Graph
     is_empty(Graph)→Boolean;
end graph
```

These are the primitive operations that are needed for storing and processing a graph.

Create

The create operation provides the appropriate framework for the processing of graphs. The `create()` function is used to create an empty graph. An empty graph has both V and E as null sets. The empty graph has the total number of vertices and edges as zero. However, while implementing, we should have V as a non-empty set and E as an empty set as the mathematical notation normally requires the set of vertices to be non-empty.

Insert Vertex

The insert vertex operation inserts a new vertex into a graph and returns the modified graph. When the vertex is added, it is isolated as it is not connected to any of the vertices in the graph through an edge. If the added vertex is related with one (or more) vertices in the graph, then the respective edge(s) are to be inserted.

Figure 8.1(a) shows a graph $G(V, E)$, where $V = \{a, b, c\}$ and $E = \{(a, b), (a, c), (b, c)\}$, and the resultant graph after inserting the node d. The resultant graph G is shown in Fig. 8.1(b). It shows the inserted vertex with resultant $V = \{a, b, c, d\}$. We can show the adjacency relation with other vertices by adding the edge. So now, E would be $E = \{(a, b), (a, c), (b, c), (b, d)\}$ as shown in Fig. 8.1(c).

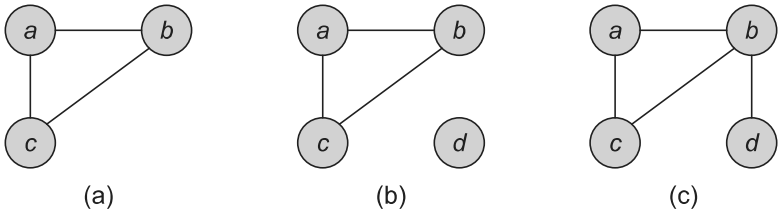


Fig. 8.1 Inserting a vertex in a graph (a) Graph *G* (b) After inserting vertex *d* (c) After adding an edge

Delete Vertex

The delete vertex operation deletes a vertex and all the incident edges on that vertex and returns the modified graph.

Figure 8.2(a) shows a graph $G(V, E)$ where $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c), (b, d)\}$, and the resultant graph after deleting the node c is shown in Fig. 8.2(b) with $V = \{a, b, d\}$ and $E = \{(a, b), (b, d)\}$.

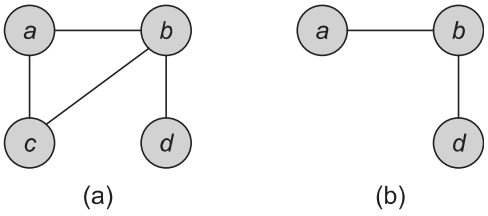


Fig. 8.2 Deleting a vertex from a graph (a) Graph *G* (b) Graph after deleting vertex *c*

Insert Edge

The insert edge operation adds an edge incident between two vertices. In an undirected graph, for adding an edge, the two vertices u and v are to be specified, and for a directed graph along with vertices, the start vertex and the end vertex should be known.

Figure 8.3(a) shows a graph $G(V, E)$ where $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c), (b, d)\}$ and the resultant graph after inserting the edge (c, d) is shown in Fig. 8.3(b) with $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c), (b, d), (c, d)\}$.

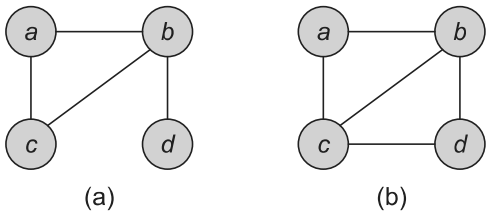


Fig. 8.3 Inserting an edge in a graph (a) Graph *G* (b) After inserting edge (*c*, *d*)

Delete Edge

The delete edge operation removes one edge from the graph. Let the graph G be $G(V, E)$. Now, deleting the edge (u, v) from G deletes the edge incident between vertices u and v and keeps the incident vertices u, v.

Figure 8.4(a) shows a graph $G(V, E)$, where $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c), (b, d)\}$. The resultant graph after deleting the edge (b, d) is shown in Fig. 8.4(b) with $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c)\}$.

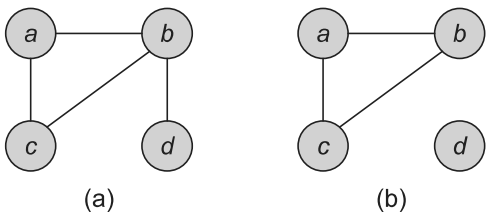


Fig. 8.4 Deleting edge in graph (a) Graph *G* (b) Graph after deleting the edge (*b*, *d*)

Is_empty

The is_empty operation checks whether the graph is empty and returns true if empty else returns false. An empty graph is one where the set V is a null set.

These are the basic operations on graphs, and a few more include getting the set of adjacent nodes of a vertex or an edge and traversing a graph. Checking the adjacency between vertices means verifying the relationship between them, and the relationship is maintained using a suitable data structure.

Graph traversal is also known as searching through a graph. It means systematically passing through the edges and visiting the vertices of the graph. A graph search algorithm can help in listing all vertices, checking connectivity, and discovering the structure of a graph. We shall discuss traversals in Section 8.4.

8.3 REPRESENTATION OF GRAPHS

We need to store two sets V and E to represent a graph. Here V is a set of vertices and E is a set of incident edges. These two sets basically represent the vertices and adjacency relationship among them. There are two standard representations of a graph given as follows:

1. Adjacency matrix (sequential representation) and
2. Adjacency list (linked representation)

Using these two representations, graphs can be realized using the adjacency matrix, adjacency list, or adjacency multilist. Let us study each of them.

8.3.1 Adjacency Matrix

Adjacency matrix is a square, two-dimensional array with one row and one column for each vertex in the graph. An entry in row i and column j is 1 if there is an edge incident between vertex i and vertex j, and is 0 otherwise. If a graph is a weighted graph, then the entry 1 is replaced with the weight. It is one of the most common and simple representations of the edges of a graph; programs can access this information very efficiently.

For a graph $G = (V, E)$, suppose $V = \{1, 2, \ldots, n\}$. The adjacency matrix for G is a two-dimensional $n \times n$ Boolean matrix $\boldsymbol{A}$ and can be represented as

$$\boldsymbol{A}[i][j] = \begin{cases} 1 & \text{if there exists an edge } <i, j> \\ 0 & \text{if edge } <i, j> \text{ does not exist} \end{cases}$$

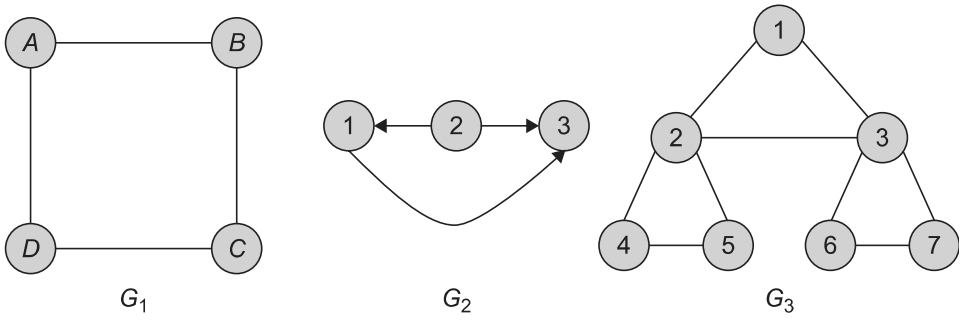
The adjacency matrix $\boldsymbol{A}$ has a natural implementation as in the following:

$\boldsymbol{A}[i][j]$ is 1 (or true) if and only if vertex i is adjacent to vertex j. If the graph is undirected, then

$$\boldsymbol{A}[i][j] = \boldsymbol{A}[j][i] = 1$$

If the graph is directed, we interpret 1 stored at $\boldsymbol{A}[i][j]$, indicating that the edge from i to j exists and not indicating whether or not the edge from j to i exists in the graph.

The graphs G_1, G_2, and G_3 of Fig. 8.5 are represented using the adjacency matrix in Fig. 8.6, among which G_2 is a directed graph.



G_1 G_2 G_3

Fig. 8.5 Graphs G_1, G_2, and G_3

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

G_1

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

G_2

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

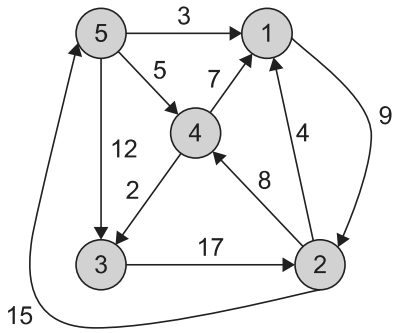
G_3

Fig. 8.6 Adjacency matrix for G_1, G_2, and G_3 of Fig. 8.5

For a weighted graph, the matrix $\boldsymbol{A}$ is represented as

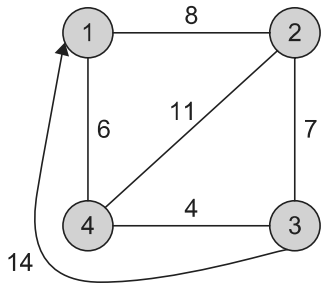
$$\boldsymbol{A}[i][j] = \begin{cases} \text{weight} & \text{if the edge } <i, j> \text{ exists} \\ 0 & \text{if there exists no edge } <i, j> \end{cases}$$

Here, weight is the label associated with the edge of the graph. For example, Figs 8.7(a) and (b) show the weighted graph and its associated adjacency matrix.



	1	2	3	4	5
1	0	9	0	0	0
2	4	0	0	8	15
3	0	17	0	0	0
4	7	0	2	0	0
5	3	0	12	5	0

(a)



	1	2	3	4
1	0	8	14	6
2	8	0	7	11
3	14	7	0	4
4	6	11	4	0

(b)

Fig. 8.7 Adjacency matrix (a) Directed weight graph and its adjacency matrix (b) Undirected weight graph and its adjacency matrix

We can note that the adjacency matrix for an undirected graph is symmetric whereas the adjacency matrix for a directed graph need not be symmetric.

Program Code 8.1 shows the class defined for graph implementation using an adjacency matrix with some basic functions.

PROGRAM CODE 8.1

```
class Graph
{
   private:
      int Adj_Matrix[Max_Vertex];    // Adjacency matrix
      int Vertex;    // Number of vertices
      int Edge;    // Number of edges
   public:
```

```
        Graph();     // Constructor
        bool IsEmpty();
        void Insert_Edge(int u, int v);
        void Insert_Vertex(int u);
        void Delete_Edge(int u, int v);
       void Delete_Vertex(int u);                    .
};
```

Program Code 8.1 depicts a class and its member functions for a graph as an adjacency matrix. In the adjacency matrix representation, the time required to access an element is independent of the size of V and E. The space needed to represent a graph using adjacency matrix is n^2 locations, where $|V| = n$. When the graph is undirected, we need to store only the upper or lower triangular matrix, as the matrix is symmetric and this reduces the space required.

As we represent the edge of a graph using the adjacency matrix, we can place an edge query. For example, to determine whether an edge is incident between the vertices i and j, just examine Adj_Matrix[i][j] in constant time O(1). We may need to get all vertices adjacent to a particular vertex, say i. Finding all the adjacent vertices requires searching the complete i^{th} row in O(n) time.

Most of the algorithms need to process almost all edges and also need to check whether the graph is connected or not. Such queries examine almost all entries in the adjacency matrix. Hence, we need to examine n^2 entries. If we omit diagonal entries, $(n^2 - n)$ entries of the matrix are to be examined (as diagonal entries are 0 in graph without self loops) in O(n^2) of time.

When the graph is sparse, most of the vertices have a few neighbours, that is, a few vertices adjacent to them. Consider the graph in Fig. 8.7. In the adjacency matrix of the graph, very few entries are non-zero. When we need a list of adjacent vertices of a particular vertex, say i, we need to transverse the complete i^{th} row though there are very few non-zero entries. Instead, if we keep one list per vertex and list only the vertices adjacent to it, a rapid retrieval in time O($e + n$) is possible when we need to process almost all edges. Here e is the number of edges in the graph, and the graph is sparse, that is, $e << (n^2/2)$. Such a structure that has a list for each vertex containing all its adjacent vertices is called as *adjacency list*. Let us learn more about adjacency list.

8.3.2 Adjacency List

In this representation, the n rows of the adjacency list are represented as n-linked lists, one list per vertex of the graph. The adjacency list for a vertex i is a list of all vertices adjacent to it. One way of achieving this is to go for an array of pointers, one per vertex. For example, we can represent the graph G by an array `Head`, where `Head[i]` is a pointer to the adjacency list of vertex i. For list, each node of the list has at least two fields: vertex and link. The vertex field contains the vertex id, and the link field stores a pointer to

the next node that stores another vertex adjacent to *i*. Figure 8.8(b) shows an adjacency list representation for a directed graph in Fig. 8.8(a).

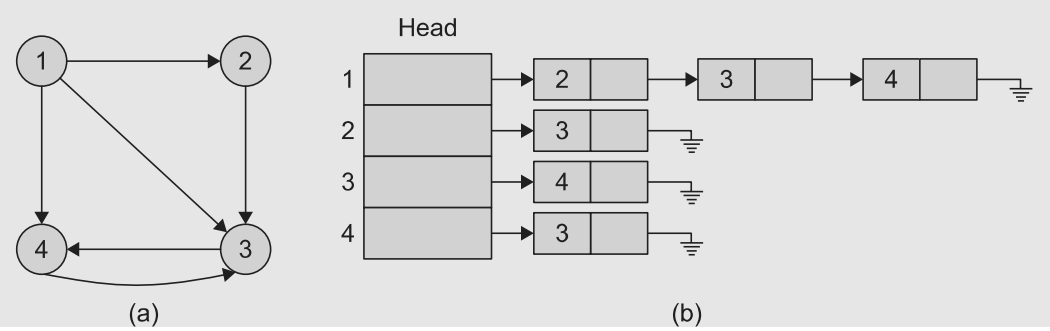


Fig. 8.8 Adjacency list representation (a) Graph G_1 (b) Adjacency list for G_1

Program Code 8.2 lists the class for the node required for adjacency list representation of the graph.

PROGRAM CODE 8.2

```
// Class for the node of the weighted graph
#define max 10
class GraphNode
{
   public:
      int vertex;
      int weight;
      // optional for weight associated with edge
      GraphNode* next;
      GraphNode()
      {
         vertex = 0;
         weight = 0;
         // optional for weight associated with edge
         next = null;
      }
};
class Graph   // class for storing graph as adjacency list
{
   GraphNode* headnodes[max];
   // headnodes list for connected vertices.
   int n;
   int visited[max];
```

```
    public:
        Graph();
        // Constructor to initialize all headnodes to null.
};
Graph :: Graph()
{
    for(int i = 0;i<max;i++)
        headnodes[i] = null;
}
```

The graph in Fig. 8.8(a) is a directed graph. If the graph is a weighted graph, a weight field can be added in the node structure of the list. Figures 8.9(a) and (b) show the adjacency list representation of a weighted directed graph.

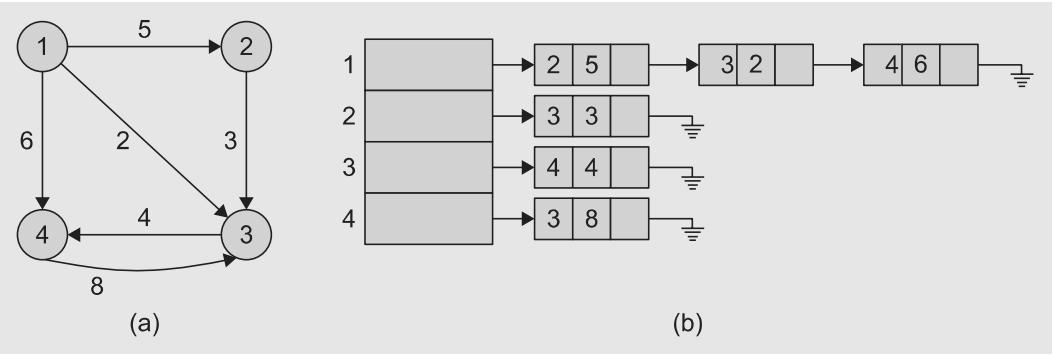


Fig. 8.9 Adjacency list of weighted graph (a) Weighted graph G_2 (b) Adjacency list of G_2

Here, each node has three fields—the first one showing an adjacent node, second showing the weight associated with an edge, and the third showing the link to the next node.

The adjacency list representation of a directed graph requires the storage proportional to the sum of the number of vertices plus the number of edges. It is often used when the number of edges is much lesser, that is, $e << n^2/2$. In case of an undirected graph, with n vertices and e edges, this representation requires $2e$ list nodes. Both directed and undirected graphs require n head nodes per node.

As we represent the edge between the vertices using the adjacency list, we can place an edge query. For example, to determine whether an edge is incident between the vertices i and j, verify by searching the complete list of m nodes adjacent to vertex i in O(m) time and if $m < n$. In worst case, the search time is O(n) when the vertex i has all the remaining $n - 1$ vertices adjacent to it, whereas in adjacency matrix representation, the search time is O(1).

Finding the degree of any vertex, that is, counting the total number of vertices adjacent to it, in an undirected graph may be determined by counting the number of nodes in its

adjacency list in O(n) time. In addition, when all the edges are to be processed, the total edges of G may be processed in time O($n + e$).

In case of a directed graph, the outgoing degree of any vertex i may be determined by counting the number of nodes on its adjacency list. For computing an incoming degree of vertex i, we have to traverse the adjacency lists of each of the other vertices to confirm whether it is incident on i. In other words, we will have to search for the vertex i in the adjacency lists of all other vertices. This is a tedious task; hence, it is better to keep another set of lists in addition to the adjacency list called *inverse adjacency lists*. The inverse adjacency list for a vertex i is a list of all vertices j to which i is adjacent to. Inverse adjacency list can be used to compute the incoming degree of a vertex. We shall learn about inverse adjacency list in Section 8.3.4.

Program Code 8.3 depicts the implementation of storing a graph as an adjacency list.

PROGRAM CODE 8.3

```
// Class for the node of the graph class GraphNode
{
   public:
      int vertex;     // The adjacent node
      GraphNode* next;
      GraphNode()
      {
         vertex = 0;
         next = null;
      }
};

// class for storing graph as adjacency list
class Graph
{
   // List of headnodes containing list of connected
   // vertices
   GraphNode* headnodes[max];
   int n;
   int visited[max];
   public:
      Graph();     // Constructor to initialize all
      headnodes to null
      void create();     // To create graph
      // To initialize the visited array to false
      void initialize_visited();
      void BFS(int v);     // Breadth-first search
```

```
      void DFS(int v);     // Depth-First Search
      int examine_n() const {return n;}
      // Return value of n.
};
Graph :: Graph()
{
   for(int i = 0; i < max; i++)
      headnodes[i] = null;
}
// Function to create a graph

void Graph :: create()
{
// Method to create a Graph represented by adjacency
list
   GraphNode *curr,*prev;
   int n1, i, j, vertex, done = false;
   cout << endl << "Enter the no. of vertices :- ";
   cin >> n;
   for(i = 0; i < n; i++)
   {
      if(!(headnodes[i] = new GraphNode))     // Allocate
      memory for new node
      {
         cout << endl << "Insufficient memory";
         exit(0);
      }
      headnodes[i]->vertex = i + 1;
      cout << endl << "Enter the no. of vertices
      connected to" << (i+1) << ":";
      cin >> n1;
      prev = headnodes[i];
      for(j = 0; j < n1; j++)
      {
         if(!(curr = new GraphNode))
         {
            cout << endl << "Insufficient memory.";
            exit(0);
         }
         done = false;
         do
         {
```

```
            cout << endl << "Enter vertex no. of
            connected vertex :";
            cin >> vertex;
            if(vertex > n && vertex < 1)
            {
               cout << endl << "Vertex out of range";
               cout << endl << "Valid range :- 1 - " << n;
            }
            else
            {
               curr->vertex = vertex;
               prev->next = curr;
               prev = curr;                // Next node
               done = true;
            }
         }
         while(!done);
      }
      if(n1 == 0)
         prev->next = null;
   }
   return;
}
```

8.3.3 Adjacency Multilist

In the adjacency list representation of an undirected graph, each edge (v_i, v_j) is represented by two entries, one on the list of v_i and the other on the list of v_j. For the graph G_1 in Fig. 8.9, the edge connecting the vertices 1 and 2 is represented twice, in the lists of vertices 1 and 2. In applications such as minimum spanning tree computation, if we process any edge once, then it has to be marked as a processed one. To avoid processing of that edge again, we need to find the other entries for that particular edge and mark it as processed. This adds to time complexity, which should be avoided. This can be achieved if the adjacency list is maintained as multilists such that the nodes are shared among several lists. For each edge, there will be exactly one node, but this node will be in two lists, that is, the adjacency lists for each of the two nodes it is incident on. The node structure of such a list can be represented as follows:

Visited tag	V_1	V_2	Link1 for V_1	Link2 for V_2

Here, the visited tag is a one bit mark field that indicates whether or not the edge has been examined. This tag would be set accordingly when the edge is processed. We can note that the storage requirements for this are the same as that of the normal adjacency lists except the tag field. Figure 8.10 shows the adjacency multilists for the graph G_1.

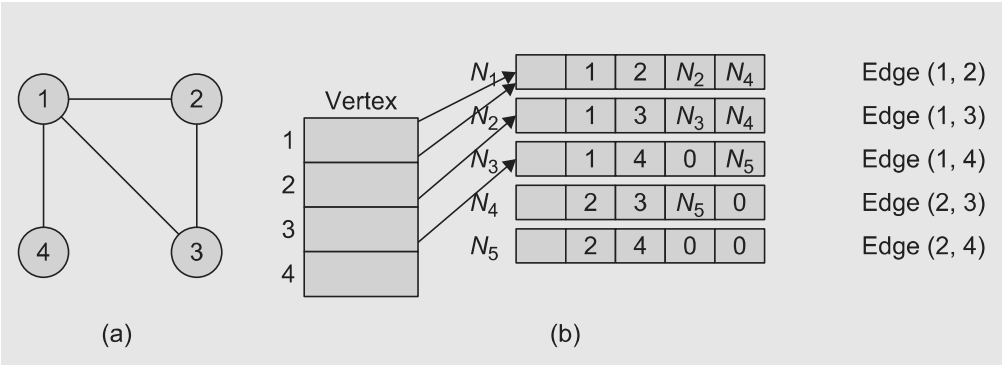


Fig. 8.10 Adjacency multilist (a) Graph G_1 (b) Adjacency multilist for G_1

For Fig. 8.10, the lists are as follows:

Vertex 1: $N_1 \rightarrow N_2 \rightarrow N_3$
Vertex 2: $N_1 \rightarrow N_4 \rightarrow N_5$
Vertex 3: $N_2 \rightarrow N_5$
Vertex 4: $N_3 \rightarrow N_5$

Sometimes, the edges of a graph have weights assigned when the graph is a weighted graph. This weight information can be represented using an adjacency matrix or can also be shown by including an additional field in the node.

8.3.4 Inverse Adjacency List

An *inverse adjacency list* is a set of lists that contains one list for each vertex. Each list contains a node per vertex adjacent to the vertex it represents. Figure 8.11(b) represents the inverse adjacency list for the graph G_2 in Fig. 8.11(a).

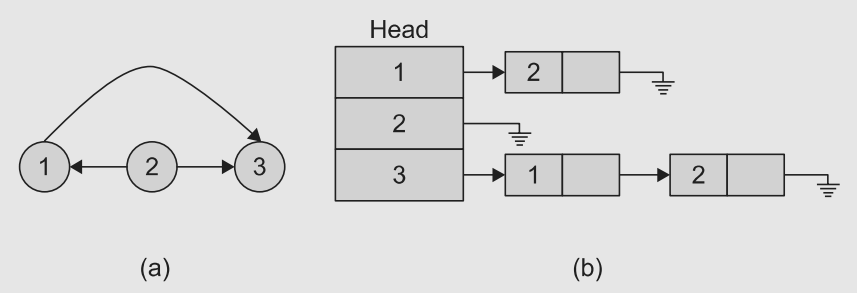


Fig. 8.11 Inverse adjacency list (a) Graph G_2
(b) Inverse adjacency list of G_2

8.3.5 Comparison of Sequential and Linked Representations

Adjacency matrix representation always requires an $n \times n$ matrix with n vertices, regardless of the number of edges. It needs more memory asymptotically. If the graph is sparse, many of the entries are null. However, since it provides direct access, it is suitable for many applications.

Linked representation (adjacency list) of a graph has an advantage of space complexity when a graph is sparse but does not provide direct access. The probable disadvantage of adjacency list is that it does not allow direct access, and hence, we cannot quickly determine whether an edge between any two vertices is incident or not.

When a graph is sparse, the number of edges $|E|$ is much lesser than V_2. The adjacency list representation is usually preferred as it provides a compact way to represent them. For dense graphs, adjacency matrix representation may be preferred since $|E|$ is closer to V_2 and when we also want fast access to information such as whether the edge between any two vertices is incident or not, the weight associated to each edge, and so on.

Though the list representation is asymptotically as efficient as a matrix representation, the simplicity of the matrix is preferred when the graph is small. In addition, for a weighted graph, an additional field is needed in the graph node, whereas for matrix representation, the same matrix can be used. Considering all these aspects, the matrix representation of a graph is more powerful than all the other forms.

8.4 GRAPH TRAVERSAL

To solve many problems modelled with graphs, we need to visit all the vertices and edges in a systematic fashion called *graph traversal*. We shall study two types—*depth-first traversal* and *breadth-first traversal*. Traversal of a graph is commonly used to search a vertex or an edge through the graph; hence, it is also called a *search technique*. Consequently, depth-first and breadth-first traversals are popularly known as *depth-first search* (DFS) and *breadth-first search* (BFS), respectively.

8.4.1 Depth-first Search

In DFS, as the name indicates, from the currently visited vertex in the graph, we keep searching deeper whenever possible. All the vertices are visited by processing a vertex and its descendents before processing its adjacent vertices. This procedure can be written either recursively or non-recursively. For recursive code, the internal stack would be used, and for non-recursive code, we would use a stack.

Depth-first search works by selecting one vertex, say `v` of G as a start vertex; `v` is marked as visited. Then, each unvisited vertex adjacent to `v` is searched using the DFS recursively. Once all the vertices that can be reached from `v` have been visited, the search for `v` is complete. If some vertices remain unvisited, we select an unvisited vertex as a new start vertex and then repeat the process until all the vertices of G are marked as visited.

For non-recursive implementation, whenever we reach a node, we shall push it (vertex or node address) onto the stack. We would then pop the vertex, process it, and push all its adjacent vertices onto the stack. Suppose we have a directed graph *G* where all the vertices are initially marked as unvisited. In a graph, we can reach any vertex more than once through different paths. Hence, to assure that each vertex is visited once, we mark each as visited whenever it is processed. Let us use an array say `visited` for the same. Initially, all vertices are marked unvisited. Marking `visited[i]` to 0 indicates that the vertex `i` is unvisited. Whenever we push the vertex say `j` onto the stack, we mark it visited by setting its `visited[j]` to 1.

The recursive algorithm for DFS can be outlined as in Algorithm 8.1.

Algorithm 8.1 shows the recursive working of DFS of a graph.

ALGORITHM 8.1

```
1. for v = 1 to n do
     visited[v] = 0            {unvisited}
2. i = 1            {Let us start at vertex 1)
3. DepthFirstSearch(i)
   begin
      visited[i] = 1
      for each vertex j adjacent to i do
         if(visited[j] = 0)  then
             DepthFirstSearch(j)
         end
4. stop
```

When we need to show its equivalent non-recursive code, we need to use a stack. Non-recursive DFS can be implemented by using a stack for pushing all unvisited vertices adjacent to the one being visited and popping the stack to find the next unvisited vertex.

Consider the graph in Fig. 8.12(a) and its adjacency list in Fig. 8.12(b).

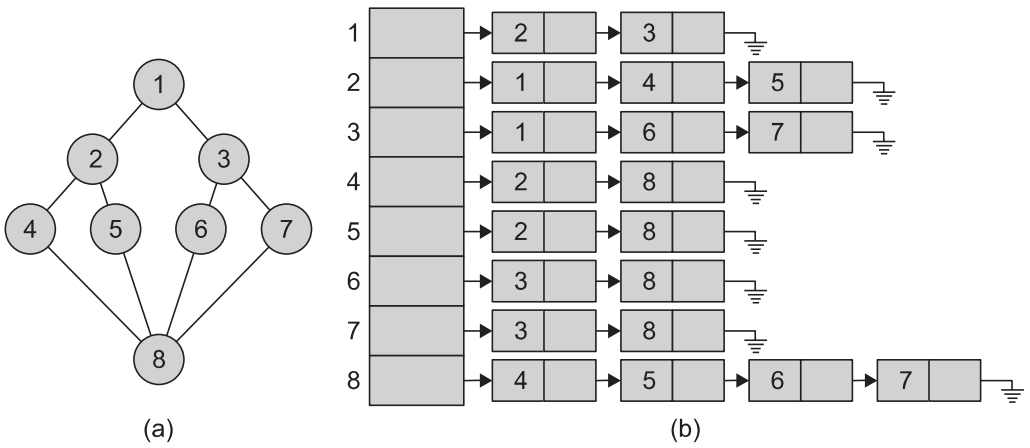


Fig. 8.12 Sample graph for traversal (a) Graph *G* (b) Adjacency list representation of *G*

Let us initiate a traversal from the vertex 1. The order of traversal will be 1, 2, 4, 8, 5, 6, 3, 7. Another possible traversal could be 1, 3, 7, 8, 6, 5, 2, 4. O($n + e$) time is required by the DFS for adjacency list representation and O(n^2) for adjacency matrix representation. Program Code 8.4 is the implementation of the DFS traversal in C++ where the graph is stored as an adjacency matrix.

PROGRAM CODE 8.4

```
// Depth-first search using adjacency matrix
void Graph :: DepthFirstSearch(int i)
{
   int k;
   for(k = 0; k < Vertex; k++)
   visited[k] = 0;
   visited[i] = 1;
   for(k = 0; k < Vertex; k++)
   {
      if(Adj_Matrix[i, k] && !visited[k])
      {
         cout << i + 1;
         void DepthFirstSearch(i);
      }
   }
}

// Function for Depth-first search using adjacency list
void Graph :: DFS(int v)
{
   GraphNode *curr;
   int w;
   curr = headnodes[v];
   cout << "\t" << curr->vertex;
   visited[v] = true;
   curr = curr ->next;
   while(curr != null)
   // For each vertex adjacent to v
   {
      if(!visited[w = (curr->vertex - 1)])
         DFS(w);
      curr = curr->next;
   }
   return;
}
```

Depth-first search for an undirected graph works in a similar way as for a directed graph as shown in Algorithm 8.2. The start vertex `i` is marked visited. Next, an unvisited vertex `j` adjacent to `i` is selected and a DFS from `j` is initiated. When a vertex `k` is reached such that all its adjacent vertices have been visited, the search returns to the last vertex visited which has an unvisited vertex `j` adjacent to it and then initializes the DFS from `j`. The search terminates when no unvisited vertex can be reached from any of the visited vertices. If the graph *G* is represented by its adjacency lists, the adjacent vertices `j` from `i` can be easily searched by following the chain of links through the list of vertex `i`.

ALGORITHM 8.2

```
1. Let us start search at vertex j
2. Push j onto stack
3. Mark all vertices as unvisited
   for i = 1 to n do
      visited[i] = 0
4. while(not empty (stack)) do
   begin
      v = pop(stack)
      if(not visited(v))
      begin
        visited[v] = 1
        push all adjacent vertices of v onto stack
      end
   end
5. stop
```

Let us now consider the graph in Fig. 8.13.

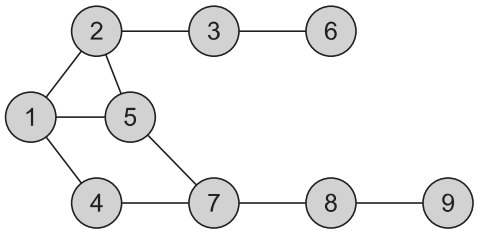
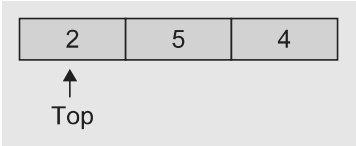


Fig. 8.13 Sample graph

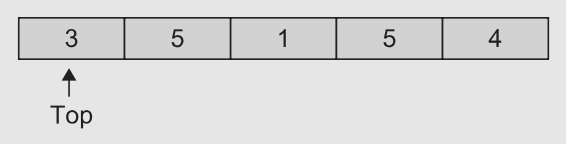
Let us traverse the graph using a non-recursive algorithm that uses stack. Let 1 be the start vertex. Note that the stack is empty initially.

1. Initially, V = set of visited vertices = ϕ. Push 1 onto the stack.
2. As the stack is not empty, `vertex = pop();` we get 1. As 1 is not visited, mark it as visited.

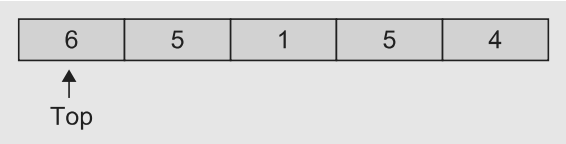
Now $V = \{1\}$. Push all the adjacent vertices of 1 onto the stack.
Since the stack is not empty, `vertex = pop();` we get 2.



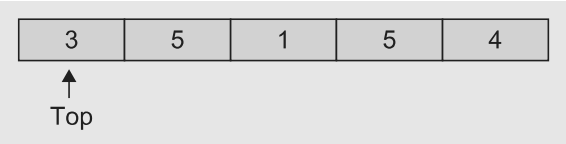
3. As 2 is not visited, mark it as visited, and now $V = \{1, 2\}$. Then, push all the adjacent vertices of 2.



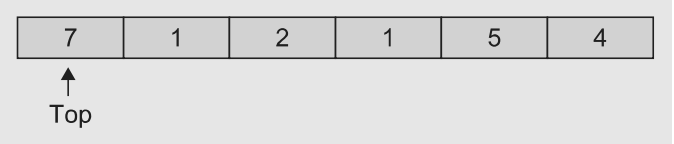
4. Since the stack is not empty, `vertex = pop();` we get 3. As 3 is not visited, mark it as visited. Now $V = \{1, 2, 3\}$. We then push all the adjacent vertices of 3 onto the stack.



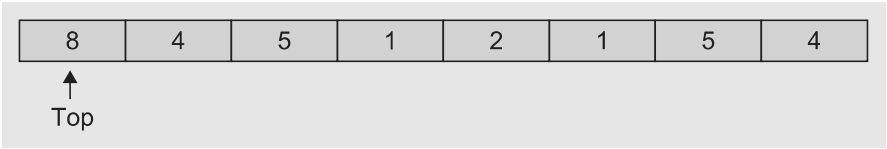
5. Since the stack is not empty, `vertex = pop();` we get 6. As 6 is not visited, mark it as visited. Now $V = \{1, 2, 3, 6\}$. We then push all the adjacent vertices of 6 onto the stack.



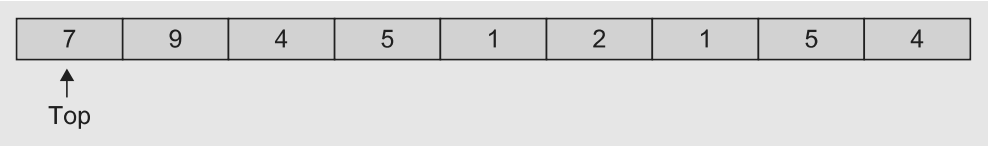
6. Since the stack is not empty, `vertex = pop();` we get 3. As 3 is visited, pop again `vertex = pop();` we then get 5. As 5 is not visited, mark it as visited. Now $V = \{1, 2, 3, 6, 5\}$. Push all the adjacent vertices onto the stack.



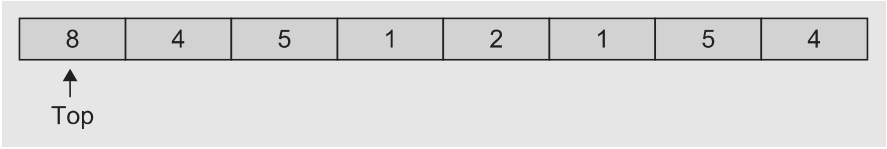
7. As the stack is not empty, `vertex = pop();` we get 7, which is not visited. Hence, $V = \{1, 2, 3, 6, 5, 7\}$; we now push all the adjacent vertices of 7 onto the stack.



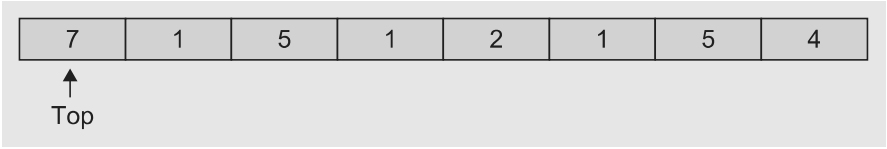
8. As the stack is not empty, `vertex = pop();` we get 8, which is not visited. Hence, $V = \{1, 2, 3, 6, 5, 7, 8\}$. Push all the adjacent vertices of 8 onto the stack.



9. As the stack is not empty, `vertex = pop()` = 7, which is visited; `vertex = pop()` = 9, which is not visited. Hence, $V = \{1, 2, 3, 6, 5, 7, 8, 9\}$. Push all the adjacent vertices of 9 onto the stack.



10. As the stack is not empty, `vertex = pop()` = 8, which is visited; so again `vertex = pop()` = 4, which is not visited. Hence, $V = \{1, 2, 3, 6, 5, 7, 8, 9, 4\}$. Push all the adjacent vertices of 4 onto the stack.



11. The stack is not empty. So the following operations yield:

    `vertex = pop()` we get 7, visited
    `vertex = pop()` we get 1, visited
    `vertex = pop()` we get 5, visited
    `vertex = pop()` we get 1, visited
    `vertex = pop()` we get 2, visited
    `vertex = pop()` we get 1, visited
    `vertex = pop()` we get 5, visited
    `vertex = pop()` we get 4, visited

12. The stack is now empty, Hence, we stop.

The set $V = \{1, 2, 3, 6, 5, 7, 8, 9, 4\}$ represents the order in which they are visited. Hence, the DFS of the graph (Fig. 8.13) gives the sequence as 1, 2, 3, 6, 5, 7, 8, 9, and 4. This is shown in Fig. 8.14.

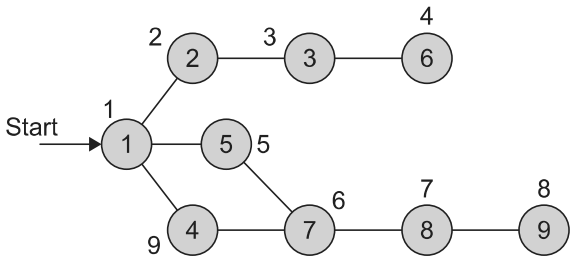


Fig. 8.14 Depth-first traversal for graph in Fig. 8.13

The label at each of the vertices in Fig. 8.14 is the sequence of visit of the traversal. The DFS of the graph is roughly analogous to the preorder traversal of an ordered tree. To find the vertices adjacent to the current vertex, we use a data structure that stores the graph to be traversed. This could be one of the suitable data structures used for graphs, such as adjacency matrix or adjacency list. The sequence in which they are pushed onto the stack and then popped depends on the graph's storage. Hence, the same graph with two different adjacency lists may generate two sequences for DFS, specially, when the graph is an undirected one. A sample graph is given in Fig. 8.15.

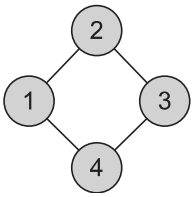


Fig. 8.15 Sample graph

If the adjacency list is as in Fig. 8.16, then the DFS gives the sequences as the following: 1, 2, 3, 4, where the start vertex is 1.

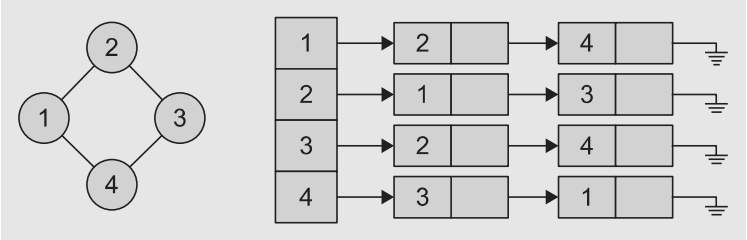


Fig. 8.16 Sample graph *G* and its adjacency list representation

If the adjacency list for the same graph is as in Fig. 8.17, then the DFS sequence will be 1, 4, 3, 2 where the start vertex is 1 and 4, 1, 2, 3 where the start vertex is 4.

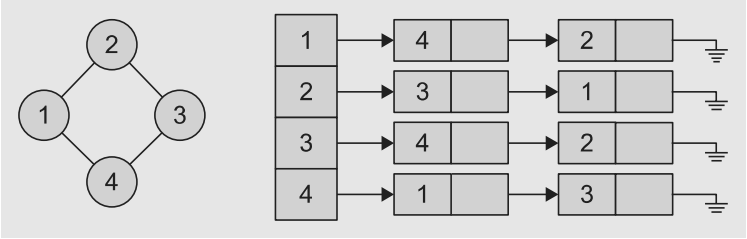


Fig. 8.17 Alternate adjacency list representation of sample graph *G*

8.4.2 Breadth-first Search

Another systematic way of visiting the vertices is the breadth-first search (BFS). The BFS differs from DFS in a way that all the unvisited vertices adjacent to `i` are visited after visiting the start vertex `i` and marking it visited. Next, the unvisited vertices adjacent to these vertices are visited and so on until the entire graph has been traversed. The approach is called 'breadth-first' because from the vertex `i` that we visit, we search as broadly as possible by next visiting all the vertices adjacent to `i`. For example, the BFS of the graph of Fig. 8.13 results in visiting the nodes in the following order: 1, 2, 3, 4, 5, 6, 7, and 8.

This search algorithm uses a queue to store the vertices of each level of the graph as and when they are visited. These vertices are then taken out from the queue in sequence, that is, first in first out (FIFO), and their adjacent vertices are visited until all the vertices have been visited. The algorithm terminates when the queue is empty. The working of the BFS is given in Algorithm 8.3. The algorithm initializes the Boolean array `visited[]` to 0 (false), that is, marks each vertex as unvisited.

ALGORITHM 8.3

```
Breadth-first search (vertex j)
1. Let us start search at vertex j
2. Mark all vertices as unvisited
      for i = 1 to n do
         visited[i] = 0
3. Mark j as visited
      visited[j] = 1
4. Add j in queue
5. while not queue empty do
   begin
      i = delete from queue
      for all vertices j adjacent to i do
      begin
         if(not visited[j] = 1)
            Add j in queue
            visited[j] = 1
         end
      end
6. stop
```

In the step 5 of Algorithm 8.3, the `while` loop is executed n times. Here, n is the number of vertices, and each vertex is inserted in the queue once. If the adjacency list representation is used, then the adjacent nodes are computed in the `for` loop. The `for` loop is executed e number of times. Hence, BFS needs $O(n + e)$ time for adjacency list and $O(n^2)$ for adjacency matrix representation.

In Program Code 8.5, we use queue `Q` as a data structure for traversal.

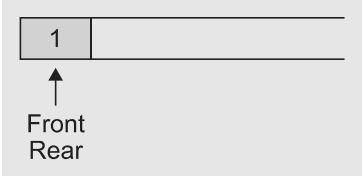
PROGRAM CODE 8.5

```
// Breadth-first traversal using adjacency matrix
void Graph :: BreadthFirstSearch(int i)
{
   int k, visited[max];
   queue Q;
   for(k = 1; k <= n; k++)
     visited[k] = 0;
   visited[i] = 1;
   Q.Add(i);
   while(!Q.IsEmpty())
   {
     j = Q.Delete();
     for(k = 1; k <= n; k++)
     {
        if(Adj_Matrix [j,k] && !visited[k])
        {
           Q.Add(k);
           visited[k] = 1;
        }
     }
   }
}
// Function for breadth-first search
void Graph :: BFS(int v)
{
   Queue q;
   GraphNode* curr;
   visited[v] = true;
   cout << "\t" << headnodes[v]->vertex;
   q.addq(headnodes[v]);
   while(!q.emptyq())
   {
     curr = q.deleteq();
     curr = curr->next;
     while(curr != null)
     {
        if(!visited[curr->vertex - 1])
        {
           q.addq(headnodes[curr->vertex - 1]);
           cout << "\t" << curr->vertex;
```

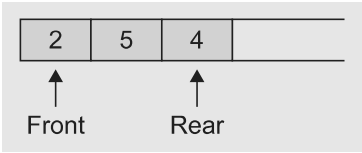
```
            visited[curr->vertex - 1] = true;
        }
        curr = curr->next;
      }
    }
    return;
}
```

Here, `add()` and `delete()` are the member functions for adding and deleting the elements from the queue, respectively. Let us consider Fig. 8.13, the graph, again for BFS. Let us traverse the graph using a non-recursive algorithm that uses a queue. Let 1 be the start vertex. Initially, the queue is empty, and the initial set of visited vertices, $V = \phi$.

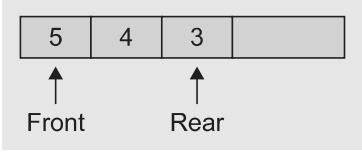
1. Add 1 to the queue. Mark 1 as visited. $V = \{1\}$.



2. As the queue is not empty, `vertex = delete()` from queue, and we get 1.
   Add all the un-visited adjacent vertices of 1 to the queue. In addition, mark them as visited.
   Now, $V = \{1, 2, 5, 4\}$.

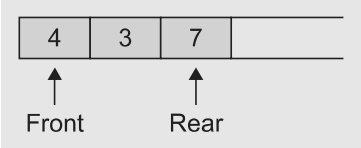


3. As the queue is not empty, `vertex = delete()` and we get 2.
   Add all the adjacent, un-visited vertices of 2 to the queue and mark them as visited.
   Now $V = \{1, 2, 5, 4, 3\}$.

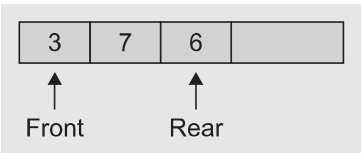


4. As the queue is not empty, `vertex = delete()` from queue, and we get 5.
   Now, add all the adjacent, un-visited vertices adjacent to 5 to the queue and mark them as visited.

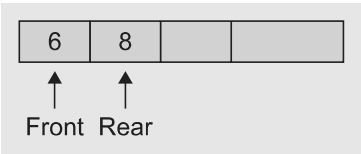
Now, $V = \{1, 2, 5, 4, 3, 7\}$.



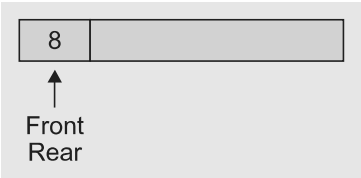
5. As the queue is not empty, `vertex = delete()` from queue, and we get 4.
   Now, add all the adjacent, not visited vertices adjacent to 4 to the queue. The vertices 1 and 7 are adjacent to 4 and hence are already visited. Now the next element we get from the queue is 3.
   Now, we add all the un-visited vertices adjacent to 3 to the queue, making $V = \{1, 2, 5, 4, 3, 7, 6\}$.



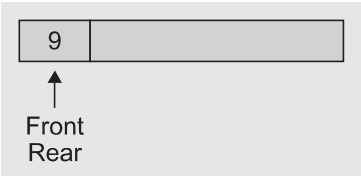
6. As the queue is not empty, `vertex = delete()` and we get 7.
   Add all the adjacent, un- visited vertices of 7 to the queue and mark them as visited.
   Now, $V = \{1, 2, 5, 4, 3, 7, 6, 8\}$.



7. As the queue is not empty, `vertex = delete()`, and we get 6.
   Then, add all the un-visited adjacent vertices of 6 to the queue and mark them as visited.
   Now $V = \{1, 2, 5, 4, 3, 7, 6, 8\}$.



8. As queue is not empty, `vertex = delete()` and we get 8.
   Add its adjacent un-visited vertices to the queue and mark them as visited.
   $V = \{1, 2, 5, 4, 3, 7, 6, 8, 9\}$.



9. As the queue is not empty, `vertex = delete()` = 9.
   Here, note that no adjacent vertices of 9 are un-visited.

10. As the queue is empty, we stop.
    The sequence in which the vertices are visited by the BFS is 1, 2, 5, 4, 3, 7, 6, 8, 9
    This is represented in Fig. 8.18.

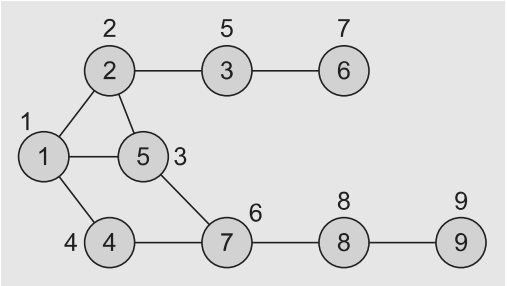


Fig. 8.18 Breadth-first search sequence for the graph in Fig. 8.13

8.5 SPANNING TREE

A tree is a connected graph with no cycles. A spanning tree is a sub-graph of *G* that has all vertices of *G* and is a tree. A minimum spanning tree of a weighted graph *G* is the spanning tree of *G* whose edges sum to minimum weight.

There can be more than one minimum spanning tree for a graph. Figure 8.19 shows a graph, one of its spanning trees, and a minimum spanning tree.

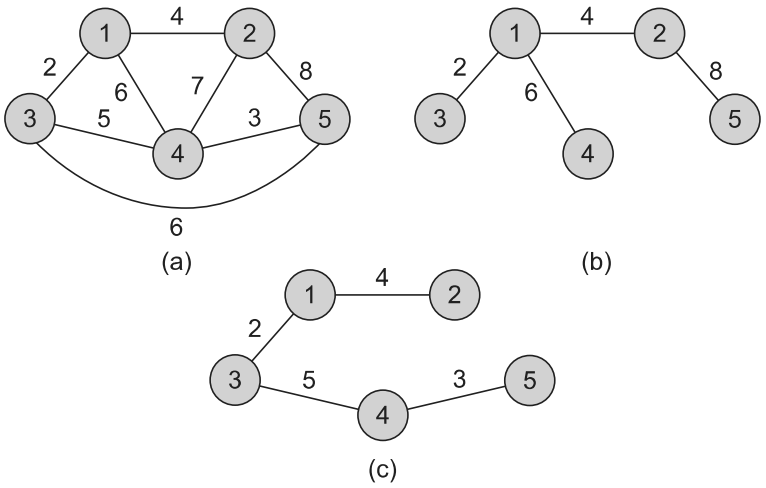


Fig. 8.19 Spanning trees (a) Graph (b) Spanning tree (c) Minimum spanning tree

Minimum spanning trees are useful in many applications such as finding the least amount of wire needed to connect a group of computers, houses, or cities. A minimum spanning tree minimizes the total length over all possible spanning trees.

We want to compute a minimum spanning tree efficiently. In theory, we could enumerate all the spanning trees of a weighted graph and simply choose the tree of least weight. However, if the graph is a complicated one, this is not an easy and efficient way to get it. In this section, we shall study the two most efficient ways discovered in the 1950s by J.B. Kruskal and R.C. Prim. Both the algorithms are greedy algorithms which produce a minimum spanning tree by adding an edge at each stage making the best choice of the next edge. These two popular methods used to compute the minimum spanning tree of a graph are

1. Prim's algorithm
2. Kruskal's algorithm

Before discussing these algorithms, let us learn about connected components.

8.5.1 Connected Components

An undirected graph is *connected* if there is at least one path between every pair of vertices in the graph. A *connected component* of a graph is a *maximal connected* sub-graph, that is, every vertex in a connected component is *reachable* from the vertices in the component.

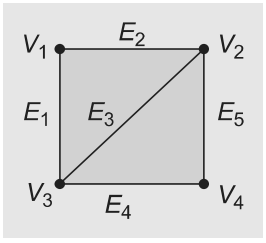


Fig. 8.20 Sample graph G_1 with one connected component

Consider the graph G_1 in Fig. 8.20.
In this undirected graph, there is only one connected component, the graph G_1 itself.

If we delete the edges e_4 and e_5 from the graph G_1, we get a graph G_2 with two connected components: $(\{V_1, V_2, V_3\}, \{E_1, E_2, E_3\})$ and $(\{V_4\}, \emptyset)$. This is represented in Fig. 8.21.

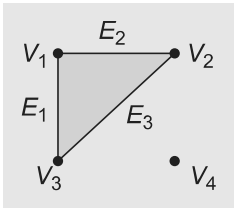


Fig. 8.21 Graph G_2 with two connected components

8.5.2 Prim's Algorithm

All vertices of any connected graph are included in a minimum cost spanning tree of a graph G. Prim's algorithm starts from one vertex and grows the rest of the tree by adding one vertex at a time, by adding the associated edges. This algorithm builds a tree by iteratively adding edges until a minimal spanning tree is obtained, that is, when all nodes

are added. At each iteration, a next minimum weight edge is added that adds a new vertex to the tree, if adding that edge does not form a cycle.

Let G = (V, E) be the original graph. Let T be a spanning tree. T = (A, B), where A and B are empty sets initially. Let us select an arbitrary vertex i from V and add it to set A. Now A = {i}. At each step, Prim's algorithm looks for the shortest possible edge <u, v> such that u ∈ A and v ∈ V – A. It then adds v to A making A = A ∪ {v} and adds the edge <u, v> to B. In this way, the edges in B at any instant form a minimum spanning tree for the vertices in A. We continue thus as long as A ≠ V. To illustrate the algorithm, let us consider the graph in Fig. 8.22.

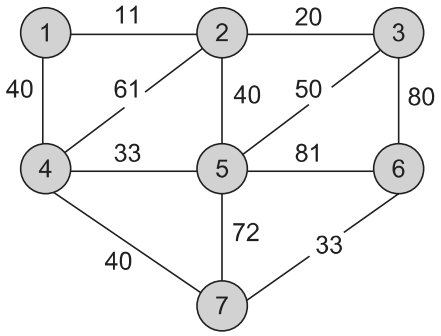


Fig. 8.22 A weighted graph

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

Table 8.1 Construction of spanning tree for graph in Fig. 8.22

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}

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. 8.23, which is of weight 177.

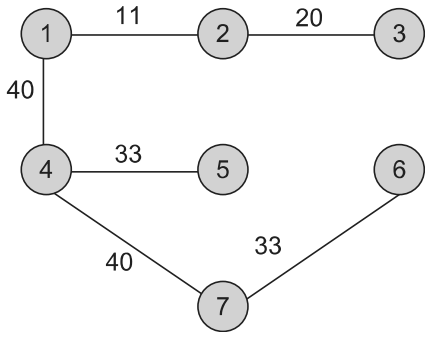


Fig. 8.23 Minimum spanning tree for graph in Fig. 8.22

Algorithm 8.4 is an informal statement of the algorithm. Here, G is a graph and T is a spanning tree to be computed.

ALGORITHM 8.4

```
1. Let G = {V, E} and T = {A, B}
         A = ϕ and B = ϕ
2. Let i ∈ V, i is a start vertex
3. A = A ∪ {i}
4. while A ≠ V do
   begin
   find edge <u,v> ∈ E of minimum length
       such that u ∈ A and v ∈ V - A
       A = A ∪ {v} and
       B = B ∪ {<u,v>}
   end
5. stop
```

To obtain a simple implementation in any programming language say C++, suppose that the vertices of G are numbered from 1 to n so that V = {1, 2, …, n}. Let the matrix M give the length of each edge and L[i][j] = ∞ if the edge <i,j> ∉ E, that is, edge <i,j> does not exist. Let us use two arrays—Nearest[] and Min_Dist[]. Let T = {A, B} be the minimum spanning tree where initially A and B are empty. For each vertex i ∈ V − A, the array Nearest[i] gives the vertex in A that is nearest to i. Similarly, for each vertex i ∈ V − A, the array Min_Dist[i] gives the distance from i to this nearest vertex. For a vertex i ∈ A, we set Min_Dist[i] = −1. In this way, we can find out whether a vertex is in A or not. The set A arbitrarily initializes to {1}.

Consider the graph in Fig. 8.24.

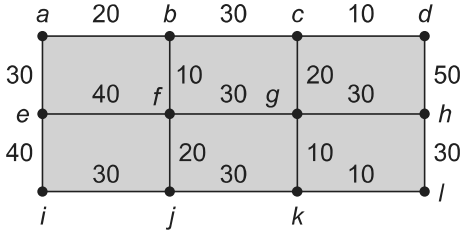
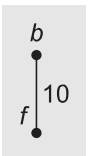


Fig. 8.24 Sample graph

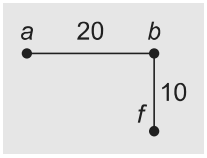
Using Prim's algorithm, we get a spanning tree for this graph in the following steps:

1. Let f be the start vertex.
   Among vertices e, b, g, and j, the vertex b is the nearest one with edge <f, b> and weight 10.



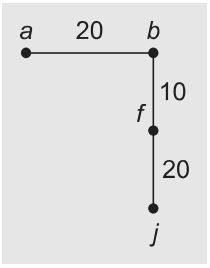
$w(f, e) = 40$
$w(f, b) = 10 \leftarrow \min$
$w(f, g) = 30$
$w(f, j) = 20$

2. Among the vertices adjacent to b and f, the vertex a is the nearest one with edge $<b, a>$ and weight 20.



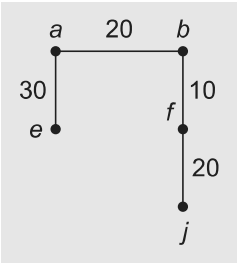
$w(b, a) = 20 \quad \leftarrow$ min
$w(b, c) = 30$
$w(f, e) = 40$
$w(f, g) = 30$
$w(f, j) = 20$

3. Similarly, the nearest vertex adjacent to one of a, b, and f is j with the edge $<f, j>$ and weight 20.



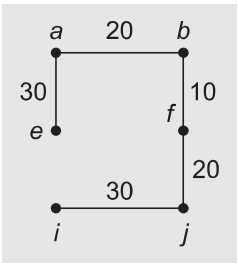
$w(a, e) = 30$
$w(b, c) = 30$
$w(f, e) = 40$
$w(f, g) = 30$
$w(f, j) = 20 \quad \leftarrow$ min

4. Similarly, the next edge added is $<a, e>$ with weight 30.



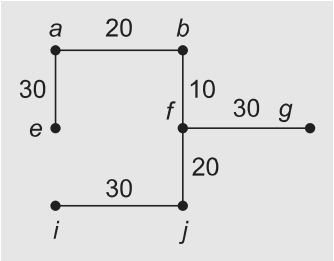
$w(a, e) = 30 \quad \leftarrow$ min
$w(b, c) = 30$
$w(f, e) = 40$
$w(f, g) = 30$
$w(j, i) = 30$
$w(j, k) = 30$

5. Edge selected = $<j, i>$ with weight 30.



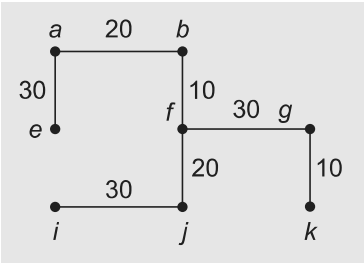
$w(b, c) = 30$
$w(f, e) = 40$
$w(f, g) = 30$
$w(j, i) = 30 \quad \leftarrow$ min
$w(j, k) = 30$
$w(e, i) = 40$
$w(e, f) = 40$

6. Edge selected = $<f, g>$ with weight 30.



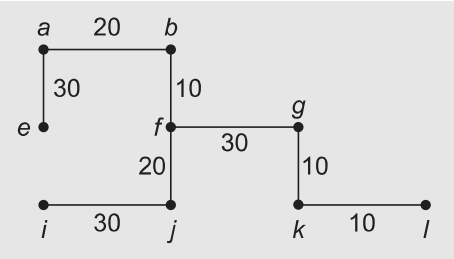
$w(b, c) = 30$
$w(f, e) = 40$
$w(f, g) = 30 \quad \leftarrow$ min
$w(j, k) = 30$
$w(e, i) = 40$
$w(e, f) = 40$

7. Edge selected = <*g*, *k*> with weight 10.



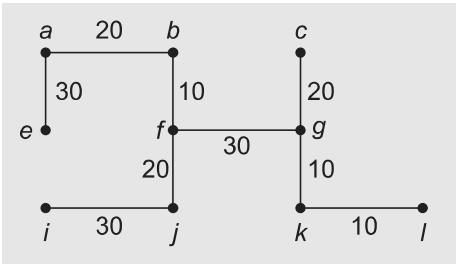
$w(b, c) = 30$
$w(f, e) = 40$
$w(j, k) = 30$
$w(e, i) = 40$
$w(e, f) = 40$
$w(g, c) = 20$
$w(g, k) = 10 \quad \leftarrow$ min
$w(g, h) = 30$

8. Edge selected = <*k*, *l*> with weight 10.



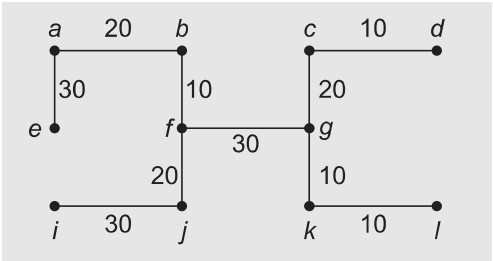
$w(b, c) = 30$
$w(f, e) = 40$
$w(j, k) = 30$
$w(e, i) = 40$
$w(e, f) = 40$
$w(g, c) = 20$
$w(g, h) = 30$
$w(k, l) = 10 \quad \leftarrow$ min

9. Edge selected = <*g*, *c*> with weight 20.



$w(b, c) = 30$
$w(f, e) = 40$
$w(j, k) = 30$
$w(e, i) = 40$
$w(e, f) = 40$
$w(g, c) = 20 \quad \leftarrow$ min
$w(g, h) = 30$
$w(l, h) = 30$

10. Edge selected = <*c*, *d*> with weight 10.



$w(b, c) = 30$
$w(f, e) = 40$
$w(j, k) = 30$
$w(e, i) = 40$
$w(e, f) = 40$
$w(g, h) = 30$
$w(l, h) = 30$
$w(c, d) = 10 \quad \leftarrow$ min

11. Finally the edge selected = <*g*, *k*> with weight 30.

As all the vertices are added, the algorithm ends. The resultant spanning tree is shown in Fig. 8.25 with a total weight of 220.

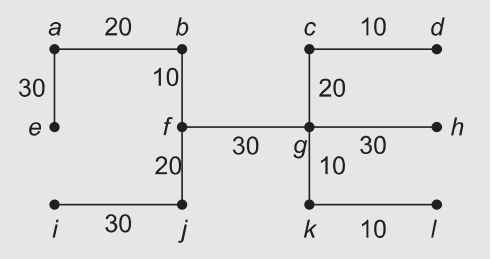


Fig. 8.25 Minimum cost spanning tree for the graph in Fig. 8.24

8.5.3 Kruskal's Algorithm

We studied Prim's algorithm to find the minimum spanning tree. Another way to construct a minimum spanning tree for a graph `G` is to start with a graph `T = (V', E' = ∅)` consisting of the `n` vertices of `G` and having no edges. Each vertex is therefore a connected component in itself. In Prim's algorithm, we start with one connected component, add a vertex to have one connected component and no cycles, and end up with one connected component. Here, we start with `n` connected components; at each step, the number of connected components would reduce by one and end up with one connected component. Here, `n` indicates the total number of vertices in a graph.

We start with all vertices; each vertex is therefore a connected component in itself. As the algorithm progresses, we add an edge to `T = (V', E' = ∅)` by examining the edges from `E`. If the edge connects two vertices in two different connected components, then we add the edge to `T`. In other words, if the edge does not form a cycle in `T`, only then an edge is added. If an edge joins two vertices of two different connected components, we add it to `T`. Consequently, the two connected components now form only one component, and the total number of connected components would be decremented by one. If it forms a cycle, that is, if the edge connects two vertices in the same component, then we discard the edge. At the end of the algorithm, only one connected component remains, so `T` is then a minimum spanning tree for all the vertices of `G`. To build a bigger component, we examine the edges of `G` in the increasing order of their associated weights.

To illustrate the method, consider the graph in Fig. 8.26.

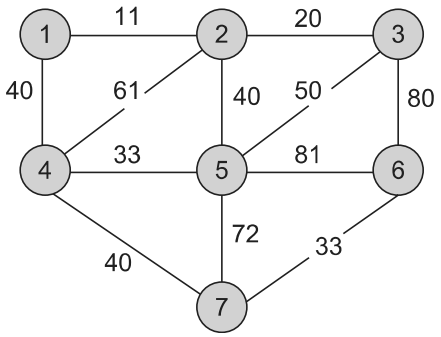


Fig. 8.26 Sample graph

Let us arrange the edges in an increasing order of their weights: <1, 2>, <2, 3>, <4, 5>, <6, 7>, <1, 4>, <2, 5>, <4, 7>, <3, 5>, <2, 4>, <3, 6>, <5, 7>, and <5, 6> with weights 11, 20, 33, 33, 40, 40, 40, 50, 61, 80, 72, and 81, respectively. Selection and addition of edges in a step-by-step manner is shown in Table 8.2.

Table 8.2 Construction of spanning tree for graph in Fig. 8.26

Step no.	Edge considered	Action	Connected component
Initial	–	–	{1} {2} {3} {4} {5} {6} {7}
1	<1, 2>	Add	{1, 2} {3} {4} {5} {6} {7}
2	<2, 3>	Add	{1,2,3} {4} {5} {6} {7}
3	<4, 5>	Add	{1,2,3} {4,5} {6} {7}
4	<6, 7>	Add	{1,2,3} {4,5} {6,7}
5	<1, 4>	Add	{1,2,3,4,5} {6,7}
6	<2, 5>	Rejected	{1,2,3,4,5} {6,7}
7	<4, 7>	Add	{1,2,3,4,5,6,7}

When the algorithm stops, `T` contains the chosen edges <1, 2>, <2, 3>, <4, 5>, <6, 7>, <1, 4>, and <4, 7>. This minimum spanning tree has the weight as 177 and is drawn in Fig. 8.22. Algorithm 8.5 states these steps in brief.

ALGORITHM 8.5

```
1. Let G = {V, E} and T = {A, B}
2. A = V and B = ϕ, |A| = n and |B| = 0
3. while(|B| < n - 1) do
   begin
     find edge <u,v> of minimum length and add to B
     only if addition of edge <u,v> does not complete a cycle in T
   end
4. stop
```

The graph `T` initially consists of the vertices of `G` but no edges. At each iteration, we add an edge `<u, v>` to `T` having minimum weight that does not complete a cycle in `T`. When `T` gets (`n` – 1) edges, the algorithm stops. To implement the algorithm, we have to handle a certain number of sets that include vertices of each connected component. Two operations have to be carried out:

1. `Member(x)` tells us which connected component the vertex `x` is a member of.
2. `Merge(u, v)` is to merge two connected components `u` and `v`.

Let us rewrite Algorithm 8.5 by elaborating these steps in Algorithm 8.6.

ALGORITHM 8.6

```
1. Sort E in increasing order of weights
2. Let G = (V, E) and T = (A, B), A = V and E = Null set
   And let n = length (V)
```

```
3. Initialize n sets, each containing a different element of v
4. while(|B| < n - 1) do
   begin
       e = <u,v> the shortest edge not yet considered
       U = Member(u)
       V = Member(v)
       if(U ≠ V)
       {
          Merge(U,V)
          Union(B,u,v)
       }
   end
5. T is the minimum spanning tree
6. stop
```

In step 4 of Algorithm 8.6, when the edge <u,v> with minimum weight is to be added in an existing tree, the function Member() checks for u and v for the connected component they belong to. If they are members of two different connected components, the edge is added as it would not form a cycle. If they belong to the same connected component, then adding the edge forms a cycle.

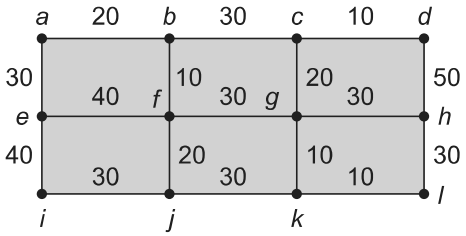


Fig. 8.27 Sample graph for Prim's spanning tree computation

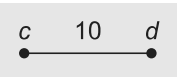
Consider the graph as in Fig. 8.27

Let us use Kruskal's algorithm.

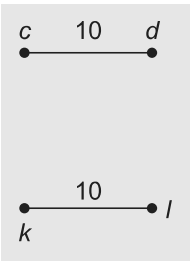
Step 1: The edge with minimum weight is selected edge = <*c, d*>.

Weight of the selected edge = 10.

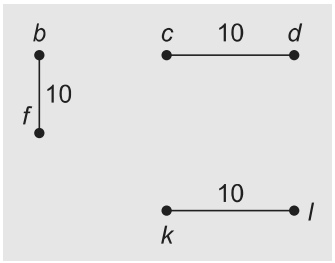
As the addition of edge to the existing tree does not form a cycle, an edge is added.



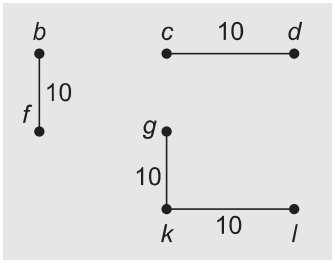
Step 2: Selected edge <*k, l*> with weight 10.



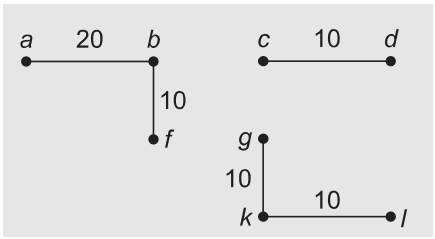
Step 3: Selected edge <*b, f*> with weight 10.



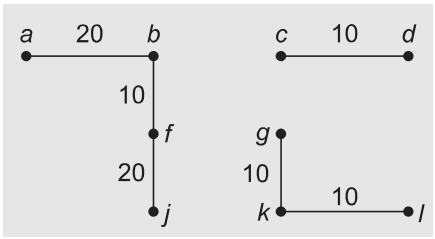
Step 4: Selected edge <*g, k*> with weight 20.



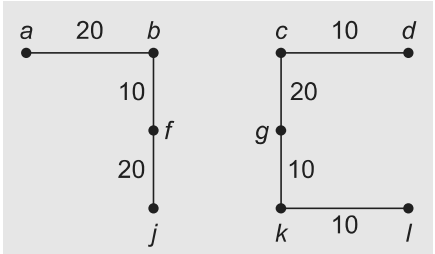
Step 5: Selected edge <*a, b*> weight 20.



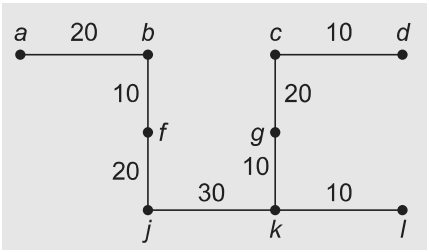
Step 6: Selected edge <*f, j*> with weight 20.



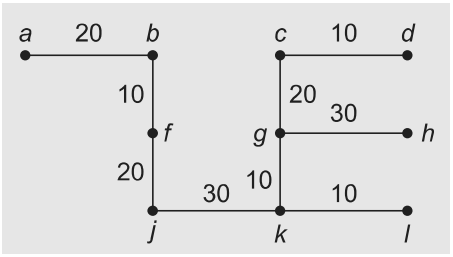
Step 7: Selected edge <*c, g*> with weight 30.



Step 8: Selected edge <*j, k*> with weight 30.



Step 9: Selected edge <*g, h*> with weight 30.



Step 10: Selected edge <*i, j*> with weight 30.

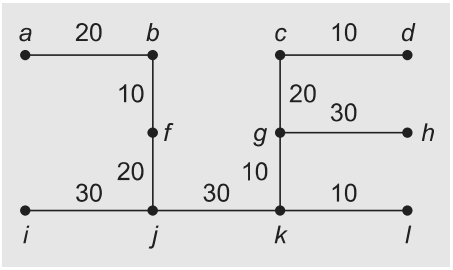


Figure 8.28 is a spanning tree with weight 220.

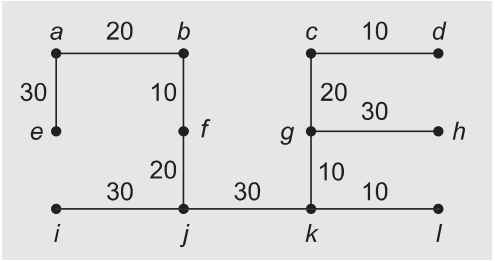


Fig. 8.28 Spanning tree for graph in Fig. 8.27

8.5.4 Biconnected Components

Depth-first search traversal of a graph, one of the most important techniques used for solving a variety of problems is described in Section 8.4.1. DFS can be used to find the connected components of an undirected graph. There are a few non-trivial graph algorithmic problems to be considered.

Consider a graph modelling a communication network problem. We expect the network to be robust under failures of any of the nodes. Even if a node fails, the remaining network should still remain connected. A graph is said to be biconnected if this condition is satisfied.

Often, we need to test whether a given undirected graph is biconnected or not. A biconnected component is a maximal biconnected sub-graph of the graph $G = (V, E)$. Edges and non-separation vertices belong to exactly one component, whereas separation vertices belong to at least two. Biconnected components contain no separation vertices or edges. A separation vertex or edge is one whose removal disconnects G. Between any two vertices, there exists at least two disjoint paths, and G has a simple cycle containing them. Any connected graph can be decomposed into a tree of biconnected components called the *block tree* of the graph. The blocks are attached to each other at shared vertices called *cut vertices* or *articulation points*. Specifically, a cut vertex is any vertex, which, when removed increases the number of connected components. In Fig. 8.29, the separation edge e_1 is between A and B, and the separation vertex is E.

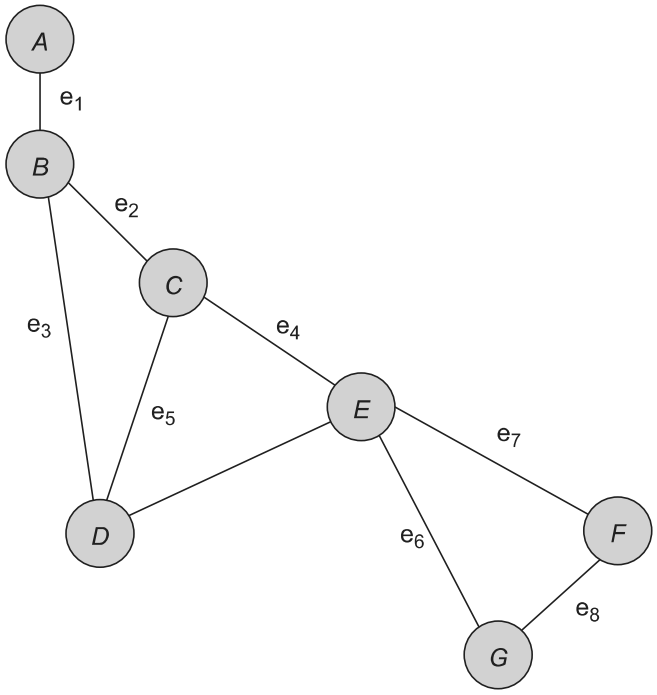


Fig. 8.29 Sample graph with biconnected components

8.5.5 Disjoint Set Operations

In minimum spanning tree computation algorithms, we have used two important set operations. Similar to those, there are many algorithms in which a disjoint-set data structure is used. This data structure keeps track of a set of elements partitioned into a number of disjoint subsets. A union–find algorithm is one that performs two useful operations (i.e., find and union) on such a data structure.

Find This is a membership check of the element. It determines the set in which a particular element is located and is also useful for determining whether two elements are in the same set or not.

Union This operation combines or merges two sets into a single set.

These two operations are supported by a disjoint-set data structure. Hence, it is also called as a *union–find data structure* or *merge–find set*.

8.6 SHORTEST PATH ALGORITHM

A *weighted graph* is a graph where the values are assigned to the edges and the length of a path is the sum of the weight of the edges in the path. We let `w(i, j)` denote the weight of edge `(i, j)`. In a weighted graph, we often need to find the shortest path. The shortest path between two given vertices is the path having minimum length. This problem can be solved by one of the greedy algorithms, by Edger W. Dijkstra, often called as *Dijkstra's algorithm*.

Consider a directed graph `G = {A, B}`. Each edge has a non-negative length. One of the nodes is the source vertex. Suppose we are to determine the shortest path from `a` to the destination vertex `z`. Let us use two sets of vertices, visited and unvisited. Let `V` denote the set of visited vertices that contains the vertices that have already been chosen and the minimal distance from the source is already known for every vertex in `V`. The set `U` contains all other vertices whose minimal distance from the source is not yet known.

Let an array `Dist` hold the length of the shortest distance and the array `Path` hold the shortest path between the source and each of the vertices. At each step, `Dist[i]` shows the shortest distance between `a` and `i`, and `Path[i]` shows the shortest path between `a` and `i`. The basic idea of the algorithm is to determine the minimum cost from `i` to one vertex at each of the iterations and call it `j`, mark `j` as visited, and recalculate the cost from `i` to each of the unvisited vertices going through `j`.

Initially, `a` is the only vertex in `V`. At each step we add to `V`, another vertex, for which the shortest path from `a` has been determined. The array `Dist[]` is initialized by setting `Dist[i]` to the weight of the edge from `a` to `i` if it exists and to ∞ if it does not. To determine which vertex to add to `V` at each step, we apply the criteria of choosing the vertex `j` with the smallest distance recorded in `Dist` such that `j` is not the visited

one. When we add `j` to `V` (set of visited vertices), we must update the entries of `Dist` by checking, for each vertex `k` that is not in `V`, whether a path through `j` and then directly to `k` is shorter than the previously recorded distance of `k`. That is, we replace `Dist[k]` by `Dist[j]` + weight of the edge from `j` to `k` if the value of the latter quantity is lesser. Here, `j` is the currently selected vertex. Let `k` be a vertex whose distance is updated. If the distance is updated, then the path is also updated. Then, `path[k]` becomes the path of `j` followed by `k`.

In brief,

if `Dist[k] > (Dist[j] + weight<j,k>)` then

```
Dist[k] = Dist[j] + w<j,k>
```

and

```
Path[k] = Path[j] U{k}
```

Algorithm 8.7 is for computing the shortest path from the source vertex to the destination vertex.

ALGORITHM 8.7

```
1. Let G = (A, B) where A = set of vertices
2. Initially, let V = {a} and U = V - {a}
3. Let U be the unvisited and V be the visited vertices
4. Let Dist[t] = w[(a, t)] for every vertex a ∈ A
5. Select the vertex in U that has the smallest value Dist[x]. Let
   x denote this vertex.
6. If x is the vertex we wish to reach from a, goto 9. If not, let
   V = V - {x} and U = U - {x}
7. For every vertex t in A, compute Dist[t] with respect to V as,
        Dist[t] = min{Dist[t], Dist[t] + w (x,t)}
8. Repeat steps 5, 6, and 7
9. Stop
```

Let us consider the graph in Fig. 8.30, and let us compute the shortest path between a and all other vertices using this algorithm.

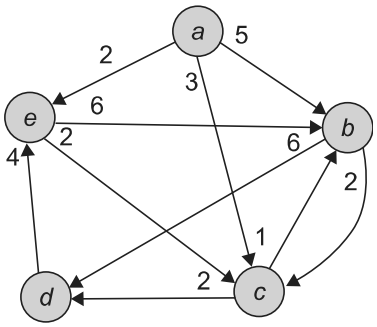


Fig. 8.30 Directed weighted graph

1. Initial step
   The set $V = \{a\}$, where a is the source vertex
   and $U = \{b, c, d, e\}$ is the set of unvisited vertices.
   Dist[ ] = {–, 5, 3, ∞, 2}. This array can also be written as

	b	c	d	e
Distance	5	3	∞	2

This Dist[ ] array represents the current shortest distance between a and other vertices.
   Path = {Ø, *ab, ac,* Ø, *ac*}
2. Now, the distance to vertex e is the shortest, so e is added to set V.
   We get, $V = \{a, e\}$; let us update Dist array now.

	b	c	d	e
Distance	5	3	6	2

The weight of the edge between the current selected vertex e and d is 4 and the distance from a to e is 2; hence the distance between a and d becomes 6 as it is less than ∞. Hence, the path is also updated for vertex d by the path of current selected vertex, that is, the path of e.

Path = {Ø, *ab, ac, aed, ae*}

3. Now the distance to vertex c among the unvisited vertices is the shortest. Hence, c is current selected vertex which gets to V.
   Therefore $V = \{a, e, c\}$. Let us update Dist array now.

	b	c	d	e
Distance	4	3	5	2

Here, the shortest distance between the source a to b and d are updated as,

$$\begin{aligned} \text{Dist}[b] &= \min\{5, \text{Dist}[c] + w(c, b)\} \\ &= \min\{5, 3 + 1\} \\ &= 4 \end{aligned}$$

and

$$\begin{aligned}\text{Dist}[d] &= \min\{6, \text{Dist}[c] + \text{w}(c, d)\} \\ &= \min\{6, 3 + 2\} \\ &= 5\end{aligned}$$

As the shortest distance of *b* and *d* are updated, their respective paths are also updated as in the following expression:

$$\text{Path} = \{\emptyset, acb, ac, acd, ae\}$$

The path vector can also be shown as follows:

Path	*b*	*c*	*d*	*e*
	acb	*ac*	*acd*	*ae*

4. Now *b* is the vertex that has the shortest distance and is unvisited.

$$\text{Hence, } V = \{a, e, c, b\}$$

Distance	*b*	*c*	*d*	*e*
	4	3	5	2

Here, none of the shortest distances is updated. Hence, the path also remains unchanged.

Path	*b*	*c*	*d*	*e*
	acb	*ac*	*acd*	*ae*

5. Now *d* is the next selected vertex, and the final distance and path vectors are the same as stated. Hence, the shortest distances between *a* and {*b, c, d, e*} are {4, 3, 5, 2}, respectively. In addition, the shortest path between *a* and {*b, c, d, e*} are {*acb, ac, acd, ae*}, respectively.

In the final two steps, adding the vertices *b* and *d* to *V* yield the paths and distances as shown in Fig. 8.31.

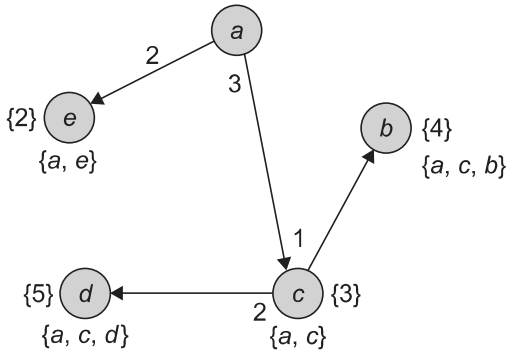


Fig. 8.31 Shortest paths and distances

To implement this algorithm in C++, let us use an adjacency matrix implementation as it facilitates random access to all the vertices of a graph. Moreover, by storing the weights in the matrix, we can use the matrix to give weights as well as adjacencies. We shall place a special large value 9999 (to represent ∞) in positions for which the corresponding edge does not exist (Program Code 8.6).

PROGRAM CODE 8.6

```
// Shortest distance using Dijkstra's algorithm

#include<iostream.h>
#include<conio.h>
#define infinite 999
class graph
{
   int Graph[20][20];
   // Adjacency Matrix int No_of_Vertices;
   public:
      void Accept();
      void Display();
      int Calc_Shortest_Dist();
};

void graph :: Accept()
{
   int i,j;
   cout << "Enter no of vertex";
   cin >> No_of_Vertices;
   for(i = 1; i<= No_of_Vertices; i++)
  {
      for(j = 1; j<= No_of_Vertices; j++)
     {
```

```
            Graph[i][j] = infinite;
         }
      }
      for(i = 1; i<= No_of_Vertices; i++)
      {
         for(j = i + 1; j<= No_of_Vertices; j++)
         {
            cout << "\n Please enter weight from
            "<<i<<"to"<<j<<":";
            cin>> Graph[i][j];
            Graph[j][i] = Graph[i][j];
         }
      }
}

void graph :: Display()
{
     int i,j;
     cout << "Graphs Adjacency Matrix is\n";
     for(i = 1; i<= No_of_Vertices; i++)
     {
        for(j = 1; j<= No_of_Vertices; j++)
        {
           cout << "\t"<< Graph [i][j];
        }
        cout << "\n";
      }
}

int graph :: Calc_Shortest_Dist()
{
   int cost, curr, src, cost1 = 0, desti, start, new1,
   i, k = 1, temp;
   int  visited[20], dist[20];
   cout << "\nEnter the source";
   cin >> src;
   cout << "\nEnter the destination";
   cin >> desti;
   for(i = 0; i<= No_of_Vertices; i++)
   {
      visited[i] = 0;
      dist[i] = infinite;
   }
```

```
    visited[src] = 1;
    dist[src] = 0;
    curr = src;
    cout << "\nPath is"<<src;
    while(curr != desti)
    {
       cost = infinite;
       start = dist[curr];
       for(i = 1; i<= No_of_Vertices; i++)
       {
          if(visited[i] == 0)
          {
             new1 = start + Graph[curr][i];
             if(new1 < dist[i])
                 dist[i] = new1;
                 if(dist[i]<cost)
                 {
                     cost = dist[i];
                     temp = i;
                 }
             }
          }
          curr = temp;
          visited[curr] = 1;
          cout << "\nCurr node is"<<curr;
          // cost1 = cost1 + cost;
    }
return cost1;
}

void main()
{
    clrscr();
    graph G;
    int Shortest_Distance;
    G.Accept();
    G.Display();
    Shortest_Distance = G. Calc_Shortest_Dist();
    cout << "\ndistance is"<< Shortest_Distance;
    getch();
}
```

RECAPITULATION

- Graphs are one of the most important non-linear data structures. A graph is a representation of relation. Vertices represent elements and edges represent relationships. In other words, a graph is a collection of nodes (vertices) and arcs joining pairs of the nodes (edges). The edges between two vertices represent the relationship between them.
- Graphs are classified as directed and undirected graphs. In an undirected graph, an edge is a set of two vertices where order does not make any relevance, whereas in a directed graph, an edge is an ordered pair.
- Graphs are implemented using an array or a linked list representation. An adjacency list is a data structure for representing a graph by keeping a list of the neighbour vertices for each vertex. An adjacency matrix is a data structure for representing a graph as a Boolean matrix where 0 means no edge and 1 corresponds to an edge.
- There are two standard graph traversals—depth-first and breadth-first.
- A minimum spanning tree is a tree, containing all the vertices of a graph, where the total weight of the edges is minimum. The two popularly used algorithms to compute minimum spanning tree are Prim's and Kruskal's algorithms.
- A biconnected component is a maximal sub-graph. A component of biconnected graph is useful in modelling a robust communication network.
- A disjoint set is a type of data structure that keeps track of a set of elements partitioned into a number of disjoint subsets. Operations such as union and find are performed on it for respectively merging two sets into one and determining the location of a given set.
- Dijkstra's algorithm is another common algorithm for graphs to find the shortest path between two vertices of a graph.

KEY TERMS

Adjacency list In an adjacency list, the n rows of the adjacency list are represented as n-linked lists, one list per vertex of the graph. We can represent G by an array Head, where Head[i] is a pointer to the adjacency list of vertex i. Each node of the list has at least two fields: vertex and link. The vertex field contains the vertex id, and link field stores the pointer to the next node storing another vertex adjacent to i.

Adjacency matrix The graphs represented using a sequential representation using matrices is called an adjacency matrix.

Adjacency multilist Multilists are lists where nodes may be shared among several other lists. For each edge, instead of two, there will be exactly one node, but this node will be in two lists, that is, the adjacency lists for each of the two nodes it is incident on.

Biconnected component A biconnected component is a maximal biconnected sub-graph of graph $G = (V, E)$ containing no separation vertices or edges.

Breadth-first search (BFS) In BFS, all the unvisited vertices adjacent to i are visited after visiting the start vertex i and marking it visited. Next, the unvisited vertices adjacent to these vertices are visited and so on until the entire graph has been traversed.

Connected component An undirected graph is connected if there is at least one path between

every pair of vertices in the graph. A connected component of a graph is a maximal connected sub-graph, that is, every vertex in a connected component is reachable from the vertices in the component.

Depth-first search (DFS) DFS differs from BFS. It starts at the vertex v of G as a start vertex and v is marked as visited. Then, each unvisited vertex adjacent to v is searched using the DFS recursively. Once all the vertices that can be reached from v have been visited, the search of v is complete. If some vertices remain unvisited, we select an unvisited vertex as a new start vertex and then repeat the process until all the vertices of G are marked visited.

Disjoint set This is a type of data structure that keeps track of a set of elements partitioned into a number of disjoint subsets.

Graph traversal Visiting all the vertices and edges in a systematic fashion is called as a graph traversal. The two most common traversals are depth-first traversal and breadth-first traversal.

Graph A graph G is a discrete structure consisting of nodes (vertices) and the lines joining the nodes (edges). For finite graphs, V and E are finite. We can write a graph as $G = (V, E)$.

Inverse adjacency list Inverse adjacency lists is a set of lists that contain one list for vertex. Each list contains a node per vertex adjacent to the vertex it represents.

Spanning tree A tree is a connected graph with no cycles. A spanning tree is a sub-graph of G that has all vertices of G and is a tree. A minimum spanning tree of a weighted graph G is the spanning tree of G whose edges sum to minimum weight.

EXERCISES

Multiple choice questions

1. Consider an undirected unweighted graph G. Let a breadth-first traversal be done starting from a node r. Let the distance $d(r, u)$ and $d(r, v)$ be the lengths of the shortest paths from r to u and v, respectively, in G. If u is visited before v during the breadth-first traversal, which of the following statements is correct?
   (a) $d(r, u) < d(r, v)$
   (b) $d(r, u) > d(r, v)$
   (c) $d(r, u) \leq d(r, v)$
   (d) None of these
2. Kruskal's algorithm for finding a minimum spanning tree of a weighted graph G with n vertices and m edges has the time complexity of
   (a) $O(n^2)$
   (b) $O(m, n)$
   (c) $O(m + n)$
   (d) $(m \log n)$
   (e) $O(m^2)$
3. Consider a simple connected graph G with n vertices and n edges ($n > 2$). Then, which of the following statements is true?
   (a) G has no cycles
   (b) The graph obtained by removing any edge from G is not connected
   (c) G has at least one cycle
   (d) The graph obtained by removing any two edges from G is not connected
   (e) None of the above
4. Which of the following statements is false?
   (a) Optimal binary search tree construction can be performed efficiently using dynamic programming.
   (b) BFS cannot be used to find the component of a graph.
   (c) The prefix and postfix walks over a binary tree cannot be uniquely constructed.
   (d) DFS can be used to find the connected components of a graph.