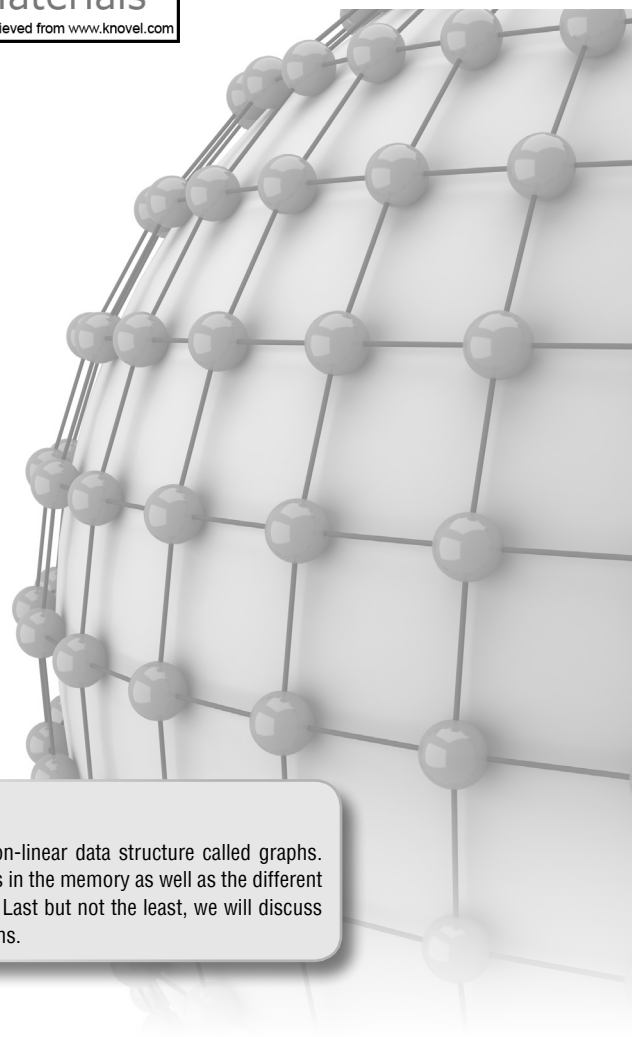
CHAPTER 13

Graphs



LEARNING OBJECTIVE

In this chapter, we will discuss another non-linear data structure called graphs. We will discuss the representation of graphs in the memory as well as the different operations that can be performed on them. Last but not the least, we will discuss some of the real-world applications of graphs.

13.1 INTRODUCTION

A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.

Why are Graphs Useful?

Graphs are widely used to model any situation where entities or things are related to each other in pairs. For example, the following information can be represented by graphs:

- *Family trees* in which the member nodes have an edge from parent to each of their children.
- *Transportation networks* in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.

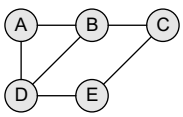


Figure 13.1 Undirected graph

Definition

A graph `G` is defined as an ordered set `(V, E)`, where `V(G)` represents the set of vertices and `E(G)` represents the edges that connect these vertices.

Figure 13.1 shows a graph with `V(G) = {A, B, C, D and E}` and `E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)}`. Note that there are five vertices or nodes and six edges in the graph.

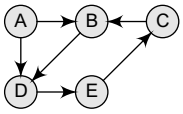


Figure 13.2 Directed graph

A graph can be directed or undirected. In an undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A. Figure 13.1 shows an undirected graph because it does not give any information about the direction of the edges.

Look at Fig. 13.2 which shows a directed graph. In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).

13.2 Graph Terminology

Adjacent nodes or neighbours For every edge, e = (u, v) that connects nodes u and v, the nodes u and v are the end-points and are said to be the adjacent nodes or neighbours.

Degree of a node Degree of a node u, deg(u), is the total number of edges containing the node u. If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an isolated node.

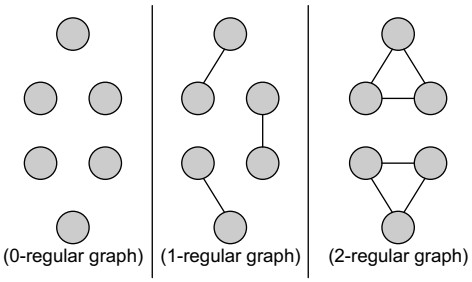


Figure 13.3 Regular graphs

Regular graph It is a graph where each vertex has the same number of neighbours. That is, every node has the same degree. A regular graph with vertices of degree k is called a k-regular graph or a regular graph of degree k. Figure 13.3 shows regular graphs.

Path A path P written as P = {v_0, v_1, v_2, ..., v_n), of length n from a node u to v is defined as a sequence of (n+1) nodes. Here, u = v_0, v = v_n and v_{i-1} is adjacent to v_i for i = 1, 2, 3, ..., n.

Closed path A path P is known as a closed path if the edge has the same end-points. That is, if $v_0 = v_n$.

Simple path A path P is known as a simple path if all the nodes in the path are distinct with an exception that v_0 may be equal to v_n. If $v_0 = v_n$, then the path is called a closed simple path.

Cycle A path in which the first and the last vertices are same. A *simple cycle* has no repeated edges or vertices (except the first and last vertices).

Connected graph A graph is said to be connected if for any two vertices (u, v) in V there is a path from u to v. That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree. Therefore, a tree is treated as a special graph (Refer Fig. 13.4(b)).

Complete graph A graph G is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has n(n–1)/2 edges, where n is the number of nodes in G.

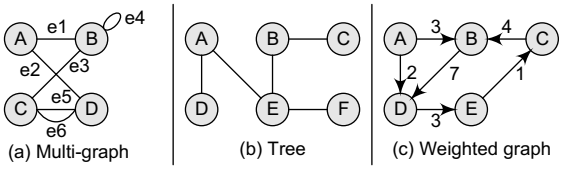


Figure 13.4 Multi-graph, tree, and weighted graph

Clique In an undirected graph G = (V, E), clique is a subset of the vertex set C ⊆ V, such that for every two vertices in C, there is an edge that connects two vertices.

Labelled graph or weighted graph A graph is said to be labelled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. The weight of an edge denoted by `w(e)` is a positive value which indicates the cost of traversing the edge. Figure 13.4(c) shows a weighted graph.

Multiple edges Distinct edges which connect the same end-points are called multiple edges. That is, `e = (u, v)` and `e' = (u, v)` are known as multiple edges of `G`.

Loop An edge that has identical end-points is called a loop. That is, `e = (u, u)`.

Multi-graph A graph with multiple edges and/or loops is called a multi-graph. Figure 13.4(a) shows a multi-graph.

Size of a graph The size of a graph is the total number of edges in it.

13.3 DIRECTED GRAPHS

A directed graph `G`, also known as a *digraph*, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair `(u, v)` of nodes in `G`. For an edge `(u, v)`,

- The edge begins at `u` and terminates at `v`.
- `u` is known as the origin or initial point of `e`. Correspondingly, `v` is known as the destination or terminal point of `e`.
- `u` is the predecessor of `v`. Correspondingly, `v` is the successor of `u`.
- Nodes `u` and `v` are adjacent to each other.

13.3.1 Terminology of a Directed Graph

Out-degree of a node The out-degree of a node `u`, written as `outdeg(u)`, is the number of edges that originate at `u`.

In-degree of a node The in-degree of a node `u`, written as `indeg(u)`, is the number of edges that terminate at `u`.

Degree of a node The degree of a node, written as `deg(u)`, is equal to the sum of in-degree and out-degree of that node. Therefore, `deg(u) = indeg(u) + outdeg(u)`.

Isolated vertex A vertex with degree zero. Such a vertex is not an end-point of any edge.

Pendant vertex (also known as leaf vertex) A vertex with degree one.

Cut vertex A vertex which when deleted would disconnect the remaining graph.

Source A node `u` is known as a source if it has a positive out-degree but a zero in-degree.

Sink A node `u` is known as a sink if it has a positive in-degree but a zero out-degree.

Reachability A node `v` is said to be reachable from node `u`, if and only if there exists a (directed) path from node `u` to node `v`. For example, if you consider the directed graph given in Fig. 13.5(a), you will observe that node `D` is reachable from node `A`.

Strongly connected directed graph A digraph is said to be strongly connected if and only if there exists a path between every pair of nodes in `G`. That is, if there is a path from node `u` to `v`, then there must be a path from node `v` to `u`.

Unilaterally connected graph A digraph is said to be unilaterally connected if there exists a path between any pair of nodes `u`, `v` in `G` such that there is a path from `u` to `v` or a path from `v` to `u`, but not both.

Weakly connected digraph A directed graph is said to be weakly connected if it is connected by ignoring the direction of edges. That is, in such a graph, it is possible to reach any node from any other node by traversing edges in any direction (may not be in the direction they point). The nodes in a weakly connected directed graph must have either out-degree or in-degree of at least 1.

Parallel/Multiple edges Distinct edges which connect the same end-points are called multiple edges. That is, `e = (u, v)` and `e' = (u, v)` are known as multiple edges of `G`. In Fig. 13.5(a), `e3` and `e5` are multiple edges connecting nodes `C` and `D`.

Simple directed graph A directed graph `G` is said to be a simple directed graph if and only if it has no parallel edges. However, a simple directed graph may contain cycles with an exception that it cannot have more than one loop at a given node.

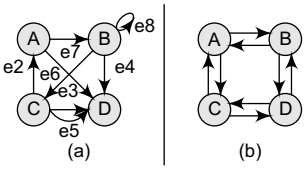


Figure 13.5 (a) Directed acyclic graph and (b) strongly connected directed acyclic graph

The graph `G` given in Fig. 13.5(a) is a directed graph in which there are four nodes and eight edges. Note that edges `e3` and `e5` are parallel since they begin at `C` and end at `D`. The edge `e8` is a loop since it originates and terminates at the same node. The sequence of nodes, `A`, `B`, `D`, and `C`, does not form a path because `(D, C)` is not an edge. Although there is a path from node `C` to `D`, there is no way from `D` to `C`.

In the graph, we see that there is no path from node `D` to any other node in `G`, so the graph is not strongly connected. However, `G` is said to be unilaterally connected. We also observe that node `D` is a sink since it has a positive in-degree but a zero out-degree.

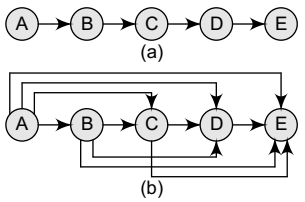


Figure 13.6 (a) A graph G and its (b) transitive closure G*

13.3.2 Transitive Closure of a Directed Graph

A transitive closure of a graph is constructed to answer reachability questions. That is, is there a path from a node `A` to node `E` in one or more hops? A binary relation indicates only whether the node `A` is connected to node `B`, whether node `B` is connected to node `C`, etc. But once the transitive closure is constructed as shown in Fig. 13.6, we can easily determine in `O(1)` time whether node `E` is reachable from node `A` or not. Like the adjacency list, discussed in Section 13.5.2, transitive closure is also stored as a matrix `T`, so if `T[1][5] = 1`, then node 5 can be reached from node 1 in one or more hops.

Definition

For a directed graph `G = (V,E)`, where `V` is the set of vertices and `E` is the set of edges, the transitive closure of `G` is a graph `G* = (V,E*)`. In `G*`, for every vertex pair `v, w` in `V` there is an edge `(v, w)` in `E*` if and only if there is a valid path from `v` to `w` in `G`.

Where and Why is it Needed?

Finding the transitive closure of a directed graph is an important problem in the following computational tasks:

- Transitive closure is used to find the reachability analysis of transition networks representing distributed and parallel systems.
- It is used in the construction of parsing automata in compiler construction.
- Recently, transitive closure computation is being used to evaluate recursive database queries (because almost all practical recursive queries are transitive in nature).

Algorithm

The algorithm to find the transitive closure of a graph G is given in Fig. 13.8. In order to determine the transitive closure of a graph, we define a matrix t where t_{ij}^{k} = 1, for i, j, k = 1, 2, 3, ... n if there exists a path in G from the vertex i to vertex j with intermediate vertices in the set (1, 2, 3, ..., k) and 0 otherwise. That is, G* is constructed by adding an edge (i, j) into E* if and only if t_{ij}^{k} = 1. Look at Fig. 13.7 which shows the relation between k and T_{ij}^{k}.

When k = 0	$T_{ij}^{0} = \begin{cases} 0 \text{ if } (i, j) \text{ is not in } E \\ 1 \text{ if } (i, j) \text{ is in } E \end{cases}$
When k ≥ 1	$T_{ij}^{k} = T_{ij}^{k-1} \vee (T_{ik}^{k-1} \wedge T_{kj}^{k-1})$

Figure 13.7 Relation between k and T_{ij}^{k}

```
Transitive_Closure(A, t, n)

Step 1: SET i=1, j=1, k=1
Step 2: Repeat Steps 3 and 4 while i<=n
Step 3:     Repeat Step 4 while j<=n
Step 4:          IF (A[i][j] = 1)
                   SET t[i][j] = 1
                 ELSE
                   SET t[i][j] = 0
            INCREMENT j
            [END OF LOOP]
        INCREMENT i
        [END OF LOOP]
Step 5: Repeat Steps 6 to 11 while k<=n
Step 6:    Repeat Steps 7 to 10 while i<=n
Step 7:        Repeat Steps 8 and 9 while j<=n
Step 8:            SET t[i,j] = t[i][j] V (t[i][k] Λ t[k][j])
Step 9:            INCREMENT j
               [END OF LOOP]
Step 10:    INCREMENT i
          [END OF LOOP]
Step 11:  INCREMENT k
         [END OF LOOP]
Step 12: END
```

Figure 13.8 Algorithm to find the transitive enclosure of a graph G

13.4 BI-CONNECTED COMPONENTS

A vertex v of G is called an articulation point, if removing v along with the edges incident on v, results in a graph that has at least two connected components.

A bi-connected graph (shown in Fig. 13.10) is defined as a connected graph that has no articulation vertices. That is, a bi-connected graph is connected and non-separable in the sense that even if we remove any vertex from the graph, the resultant graph is still connected. By definition,

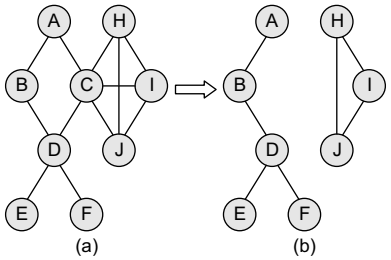


Figure 13.9 Non bi-connected graph

- A bi-connected undirected graph is a connected graph that cannot be broken into disconnected pieces by deleting any single vertex.
- In a bi-connected directed graph, for any two vertices v and w, there are two directed paths from v to w which have no vertices in common other than v and w.

Note that the graph shown in Fig. 13.9(a) is not a bi-connected graph, as deleting vertex C from the graph results in two disconnected components of the original graph (Fig. 13.9(b)).

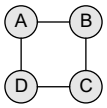


Figure 13.10 Bi-connected graph

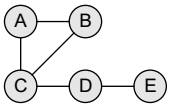


Figure 13.11 Graph with bridges

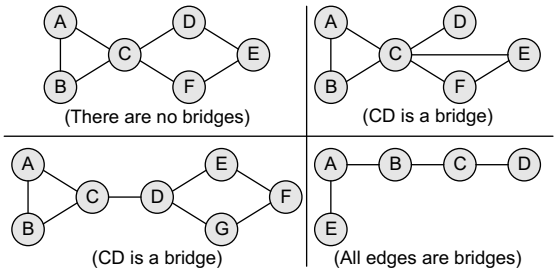


Figure 13.12 Graph with bridges

As for vertices, there is a related concept for edges. An edge in a graph is called a *bridge* if removing that edge results in a disconnected graph. Also, an edge in a graph that does not lie on a cycle is a bridge. This means that a bridge has at least one articulation point at its end, although it is not necessary that the articulation point is linked to a bridge. Look at the graph shown in Fig. 13.11.

In the graph, CD and DE are bridges. Consider some more examples shown in Fig. 13.12.

13.5 REPRESENTATION OF GRAPHS

There are three common ways of storing graphs in the computer's memory. They are:

- *Sequential representation* by using an adjacency matrix.
- *Linked representation* by using an adjacency list that stores the neighbours of a node using a linked list.
- *Adjacency multi-list* which is an extension of linked representation.

In this section, we will discuss both these schemes in detail.

13.5.1 Adjacency Matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them.

In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of $n \times n$.

In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry a_{ij} in the adjacency matrix will contain 1, if vertices v_i and v_j are adjacent to each other. However, if the nodes are not adjacent, a_{ij} will be set to zero. It is summarized in Fig. 13.13.

$$a_{ij} = \begin{cases} 1 & \text{[if } v_i \text{ is adjacent to } v_j\text{, that is there is an edge } (v_i, v_j)\text{]A} \\ 0 & \text{[otherwise]} \end{cases}$$

Figure 13.13 Adjacency matrix entry

Since an adjacency matrix contains only 0s and 1s, it is called a *bit matrix* or a *Boolean matrix*. The entries in the matrix depend on the ordering of the nodes in G. Therefore, a change in the order of nodes will result in a different adjacency matrix. Figure 13.14 shows some graphs and their corresponding adjacency matrices.

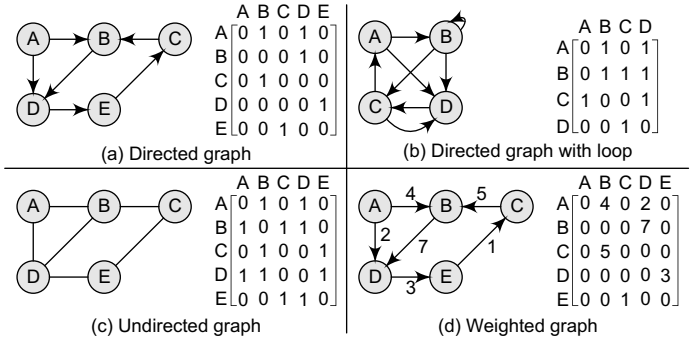


Figure 13.14 Graphs and their corresponding adjacency matrices

From the above examples, we can draw the following conclusions:

- For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.
- The adjacency matrix of an undirected graph is symmetric.
- The memory use of an adjacency matrix is $O(n^2)$, where n is the number of nodes in the graph.
- Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

Now let us discuss the powers of an adjacency matrix. From adjacency matrix A^1, we can conclude that an entry 1 in the ith row and jth column means that there exists a path of length 1 from v_i to v_j. Now consider, A^2, A^3, and A^4.

$$(a_{ij})^2 = \sum a_{ik}\, a_{kj}$$

Any entry $a_{ij} = 1$ if $a_{ik} = a_{kj} = 1$. That is, if there is an edge (v_i, v_k) and (v_k, v_j), then there is a path from v_i to v_j of length 2.

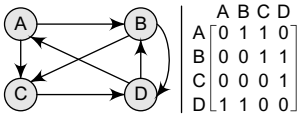


Figure 13.15 Directed graph with its adjacency matrix

Similarly, every entry in the ith row and jth column of A^3 gives the number of paths of length 3 from node v_i to v_j.

In general terms, we can conclude that every entry in the ith row and jth column of A^n (where n is the number of nodes in the graph) gives the number of paths of length n from node v_i to v_j. Consider a directed graph given in Fig. 13.15. Given its adjacency matrix A, let us calculate A^2, A^3, and A^4.

$$A^2 = A^1 \times A^1$$

$$A^2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 4 & 2 \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 3 \\ 3 & 4 & 3 & 4 \end{bmatrix}$$

Now, based on the above calculations, we define matrix B as:

$$B^r = A^1 + A^2 + A^3 + \ldots + A^r$$

An entry in the ith row and jth column of matrix B^r gives the number of paths of length r or less than r from vertex v_i to v_j. The main goal to define matrix B is to obtain the path matrix P. The path matrix P can be calculated from B by setting an entry $P_{ij} = 1$, if B_{ij} is non-zero and $P_{ij} = 0$,

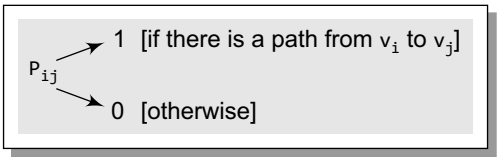


Figure 13.16 Path matrix entry

if otherwise. The path matrix is used to show whether there exists a simple path from node v_i to v_j or not. This is shown in Fig. 13.16.

Let us now calculate matrix B and matrix P using the above discussion.

$$B = \begin{bmatrix}0&1&1&0\\0&0&1&1\\0&0&0&1\\1&1&0&0\end{bmatrix} + \begin{bmatrix}0&0&1&2\\1&1&0&1\\1&1&0&0\\1&1&2&1\end{bmatrix} + \begin{bmatrix}2&2&0&1\\1&2&2&1\\0&1&2&1\\1&2&2&3\end{bmatrix} + \begin{bmatrix}1&3&4&2\\1&2&3&4\\1&1&1&3\\3&4&3&4\end{bmatrix} = \begin{bmatrix}3&6&6&5\\3&5&6&7\\2&3&3&5\\6&8&7&8\end{bmatrix}$$

Now the path matrix P can be given as:

$$P = \begin{bmatrix}1&1&1&1\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{bmatrix}$$

13.5.2 Adjacency List Representation

An adjacency list is another way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

The key advantages of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

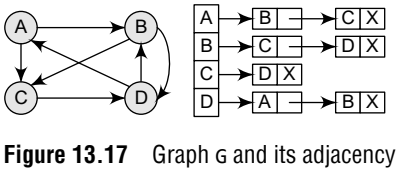


Figure 13.17 Graph G and its adjacency list

Consider the graph given in Fig. 13.17 and see how its adjacency list is stored in the memory.

For a directed graph, the sum of the lengths of all adjacency lists is equal to the number of edges in G. However, for an undirected graph, the sum of the lengths of all adjacency lists is equal to twice the number of edges in G because an edge (u, v) means an edge from node u to v as well as an edge from v to u. Adjacency lists can also be modified to store weighted graphs. Let us now see an adjacency list for an undirected graph as well as a weighted graph. This is shown in Fig. 13.18.

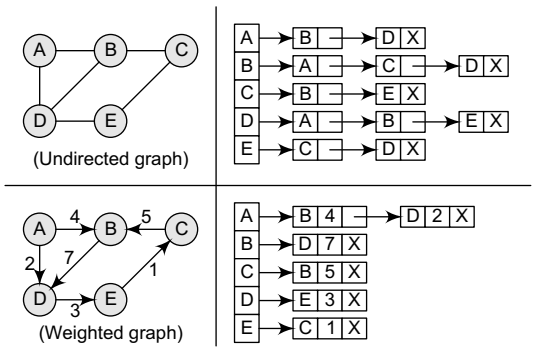


Figure 13.18 Adjacency list for an undirected graph and a weighted graph

13.5.3 Adjacency Multi-list Representation

Graphs can also be represented using multi-lists which can be said to be modified version of adjacency lists. Adjacency multi-list is an edge-based rather than a vertex-based representation of graphs. A multi-list representation basically consists of two parts—a directory of nodes' information and a set of linked lists storing information about edges. While there is a single entry for each node in the node directory, every node, on the other hand, appears in two adjacency lists (one for the node at each end of the edge). For example, the directory entry for node *i* points to the adjacency list for node *i*. This means that the nodes are shared among several lists.

In a multi-list representation, the information about an edge (v_i, v_j) of an undirected graph can be stored using the following attributes:

M: A single bit field to indicate whether the edge has been examined or not.

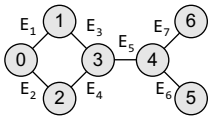


Figure 13.19 Undirected graph

v_i: A vertex in the graph that is connected to vertex v_j by an edge.

v_j: A vertex in the graph that is connected to vertex v_i by an edge.

Link i for v_i: A link that points to another node that has an edge incident on v_i.

Link j for v_i: A link that points to another node that has an edge incident on v_j.

Consider the undirected graph given in Fig. 13.19.

The adjacency multi-list for the graph can be given as:

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

Using the adjacency multi-list given above, the adjacency list for vertices can be constructed as shown below:

VERTEX	LIST OF EDGES
0	Edge 1, Edge 2
1	Edge 1, Edge 3
2	Edge 2, Edge 4
3	Edge 3, Edge 4, Edge 5
4	Edge 5, Edge 6, Edge 7
5	Edge 6
6	Edge 7

Programming Example

1. Write a program to create a graph of *n* vertices using an adjacency list. Also write the code to read and print its information and finally to delete the graph.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
struct node
{
        char vertex;
        struct node *next;
};
struct node *gnode;
void displayGraph(struct node *adj[], int no_of_nodes);
void deleteGraph(struct node *adj[], int no_of_nodes);
void createGraph(struct node *adj[], int no_of_nodes);
int main()
{
        struct node *Adj[10];
        int i, no_of_nodes;
        clrscr();
        printf("\n Enter the number of nodes in G: ");
        scanf("%d", &no_of_nodes);
        for(i = 0; i < no_of_nodes; i++)
                Adj[i] = NULL;
        createGraph(Adj, no_of_nodes);
        printf("\n The graph is: ");
        displayGraph(Adj, no_of_nodes);
        deleteGraph(Adj, no_of_nodes);
        getch();
        return 0;
}
void createGraph(struct node *Adj[], int no_of_nodes)
{
        struct node *new_node, *last;
        int i, j, n, val;
        for(i = 0; i < no_of_nodes; i++)
        {
                last = NULL;
                printf("\n Enter the number of neighbours of %d: ", i);
                scanf("%d", &n);
                for(j = 1; j <= n; j++)
                {
                        printf("\n Enter the neighbour %d of %d: ", j, i);
                        scanf("%d", &val);
                        new_node = (struct node *) malloc(sizeof(struct node));
                        new_node -> vertex = val;
                        new_node -> next = NULL;
                        if (Adj[i] == NULL)
                                Adj[i] = new_node;
                        else
                                last -> next = new_node;
                        last = new_node
                }
        }
}
void displayGraph (struct node *Adj[], int no_of_nodes)
```

```
{
        struct node *ptr;
        int i;
        for(i = 0; i < no_of_nodes; i++)
        {
                ptr = Adj[i];
                printf("\n The neighbours of node %d are:", i);
                while(ptr != NULL)
                {
                        printf("\t%d", ptr -> vertex);
                        ptr = ptr -> next;
                }
        }
}
void deleteGraph (struct node *Adj[], int no_of_nodes)
{
        int i;
        struct node *temp, *ptr;
        for(i = 0; i <= no_of_nodes; i++)
        {
                ptr = Adj[i];
                while(ptr ! = NULL)
                {
                        temp = ptr;
                        ptr = ptr -> next;
                        free(temp);
                }
                Adj[i] = NULL;
        }
}
```

Output

```
Enter the number of nodes in G: 3
Enter the number of neighbours of 0: 1
Enter the neighbour 1 of 0: 2
Enter the number of neighbours of 1: 2
Enter the neighbour 1 of 1: 0
Enter the neighbour 2 of 1: 2
Enter the number of neighbours of 2: 1
Enter the neighbour 1 of 2: 1
The neighbours of node 0 are: 1
The neighbours of node 1 are: 0 2
The neighbours of node 2 are: 0
```

Note If the graph in the above program had been a weighted graph, then the structure of the node would have been:

```
typedef struct node
{
      int vertex;
      int weight;
      struct node *next;
};
```

13.6 GRAPH TRAVERSAL ALGORITHMS

In this section, we will discuss how to traverse graphs. By traversing a graph, we mean the method of examining the nodes and edges of the graph. There are two standard methods of graph traversal which we will discuss in this section. These two methods are:

1. Breadth-first search
2. Depth-first search

While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a stack. But both these algorithms make use of a variable STATUS. During the execution of the algorithm, every node in the graph will have the variable STATUS set to 1 or 2, depending on its current state. Table 13.1 shows the value of STATUS and its significance.

Table 13.1 Value of status and its significance

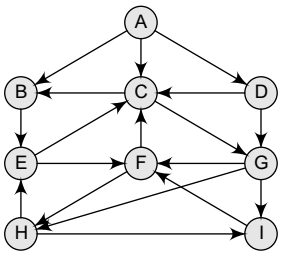
Status	State of the node	Description
1	Ready	The initial state of the node N
2	Waiting	Node N is placed on the queue or stack and waiting to be processed
3	Processed	Node N has been completely processed

13.6.1 Breadth-First Search Algorithm

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm (Fig. 13.20) explores their unexplored neighbour nodes, and so on, until it finds the goal.

```
Step 1: SET STATUS = 1 (ready state)
        for each node in G
Step 2: Enqueue the starting node A
        and set its STATUS = 2
        (waiting state)
Step 3: Repeat Steps 4 and 5 until
        QUEUE is empty
Step 4: Dequeue a node N. Process it
        and set its STATUS = 3
        (processed state).
Step 5: Enqueue all the neighbours of
        N that are in the ready state
        (whose STATUS = 1) and set
        their STATUS = 2
        (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

Figure 13.20 Algorithm for breadth-first search



Adjacency lists
A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Figure 13.21 Graph G and its adjacency list

That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth. This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

Example 13.1 Consider the graph G given in Fig. 13.21. The adjacency list of G is also given. Assume that G represents the daily flights between different cities and we want to fly from city A to I with minimum stops. That is, find the minimum path P from A to I given that every edge has a length of 1.

Solution

The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when I is encountered. During the execution of the algorithm, we use two arrays:
QUEUE and ORIG. While QUEUE is used to hold the nodes that have to be processed, ORIG is used to keep track of the origin of each edge. Initially, FRONT = REAR = –1. The algorithm for this is as follows:
(a) Add A to QUEUE and add NULL to ORIG.

FRONT = 0	QUEUE = A
REAR = 0	ORIG = \0

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

FRONT = 1	QUEUE = A	B	C	D
REAR = 3	ORIG = \0	A	A	A

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

FRONT = 2	QUEUE = A	B	C	D	E
REAR = 4	ORIG = \0	A	A	A	B

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

FRONT = 3	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

FRONT = 4	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

FRONT = 5	QUEUE = A	B	C	D	E	G	F
REAR = 6	ORIG = \0	A	A	A	B	C	E

(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

FRONT = 6	QUEUE = A	B	C	D	E	G	F	H	I
REAR = 9	ORIG = \0	A	A	A	B	C	E	G	G

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have P as A –> C –> G –> I.

Features of Breadth-First Search Algorithm

Space complexity In the breadth-first search algorithm, all the nodes at a particular level must be saved until their child nodes in the next level have been generated. The space complexity is therefore proportional to the number of nodes at the deepest level of the graph. Given a graph with branching factor b (number of children at each node) and depth d, the asymptotic space complexity is the number of nodes at the deepest level $O(b^d)$.

If the number of vertices and edges in the graph are known ahead of time, the space complexity can also be expressed as O (| E | + | V |), where | E | is the total number of edges in G and | V | is the number of nodes or vertices.

Time complexity In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches $O(b^d)$. However, the time complexity can also be expressed as O(| E | + | V |), since every vertex and every edge will be explored in the worst case.

Completeness Breadth-first search is said to be a complete algorithm because if there is a solution, breadth-first search will find it regardless of the kind of graph. But in case of an infinite graph where there is no possible solution, it will diverge.

Optimality Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node. But generally, in real-world applications, we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.

Applications of Breadth-First Search Algorithm

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

PROGRAMMING EXAMPLE

2. Write a program to implement the breadth-first search algorithm.

```
#include <stdio.h>
#define MAX 10
void breadth_first_search(int adj[][MAX],int visited[],int start)
{
        int queue[MAX],rear = -1,front =- 1, i;
        queue[++rear] = start;
        visited[start] = 1;
        while(rear != front)
        {
                start = queue[++front];
                if(start == 4)
                        printf("5\t");
                else
                        printf("%c \t",start + 65);
                for(i = 0; i < MAX; i++)
                {
                        if(adj[start][i] == 1 && visited[i] == 0)
                        {
                                queue[++rear] = i;
                                visited[i] = 1;
                        }
                }
        }
}
int main()
```

```
{
        int visited[MAX] = {0};
        int adj[MAX][MAX], i, j;
        printf("\n Enter the adjacency matrix: ");
        for(i = 0; i < MAX; i++)
                for(j = 0; j < MAX; j++)
                        scanf("%d", &adj[i][j]);
        breadth_first_search(adj,visited,0);
        return 0;
}
```

Output

```
Enter the adjacency matrix:
0 1 0 1 0
1 0 1 1 0
0 1 0 0 1
1 1 0 0 1
0 0 1 1 0
A B D C E
```

13.6.2 Depth-first Search Algorithm

The depth-first search algorithm (Fig. 13.22) progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.

In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.

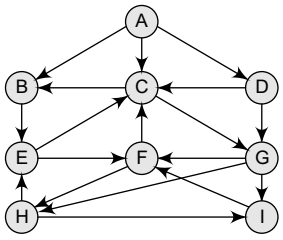
```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set
        its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4:   Pop the top node N. Process it and set its
          STATUS = 3 (processed state)
Step 5:   Push on the stack all the neighbours of N that
          are in the ready state (whose STATUS = 1) and
          set their STATUS = 2 (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

Figure 13.22 Algorithm for depth-first search

The algorithm proceeds like this until we reach a dead-end (end of path P). On reaching the dead-end, we backtrack to find another path P′. The algorithm terminates when backtracking leads back to the starting node A. In this algorithm, edges that lead to a new vertex are called *discovery edges* and edges that lead to an already visited vertex are called *back edges*.

Observe that this algorithm is similar to the in-order traversal of a binary tree. Its implementation is similar to that of the breadth-first search algorithm but here we use a stack instead of a queue. Again, we use a variable STATUS to represent the current state of the node.

Example 13.2 Consider the graph G given in Fig. 13.23. The adjacency list of G is also given. Suppose we want to print all the nodes that can be reached from the node H (including H itself). One alternative is to use a depth-first search of G starting at node H. The procedure can be explained here.



Adjacency lists
A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Figure 13.23 Graph G and its adjacency list

Solution

(a) Push H onto the stack.

STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H STACK: E, I

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I STACK: E, F

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F STACK: E, C

(e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C STACK: E, B, G

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G STACK: E, B

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B STACK: E

(h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

```
H, I, F, C, G, B, E
```

These are the nodes which are reachable from the node H.

Features of Depth-First Search Algorithm

Space complexity The space complexity of a depth-first search is lower than that of a breadth-first search.

Time complexity The time complexity of a depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as (O(|V| + |E|)).

Completeness Depth-first search is said to be a complete algorithm. If there is a solution, depth-first search will find it regardless of the kind of graph. But in case of an infinite graph, where there is no possible solution, it will diverge.

Applications of Depth-First Search Algorithm

Depth-first search is useful for:

- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

Programming Example

3. Write a program to implement the depth-first search algorithm.

```
#include <stdio.h>
#define MAX 5
void depth_first_search(int adj[][MAX],int visited[],int start)
{
        int stack[MAX];
        int top = -1, i;
        printf("%c-",start + 65);
        visited[start] = 1;
        stack[++top] = start;
        while(top ! = -1)
        {
                start = stack[top];
                for(i = 0; i < MAX; i++)
                {
                        if(adj[start][i] && visited[i] == 0)
                        {
                                stack[++top] = i;
                                printf("%c-", i + 65);
                                visited[i] = 1;
                                break;
                        }
                }
                if(i == MAX)
                top--;
        }
}
int main()
{
        int adj[MAX][MAX];
        int visited[MAX] = {0}, i, j;
```

```
            printf("\n Enter the adjacency matrix: ");
            for(i = 0; i < MAX; i++)
                    for(j = 0; j < MAX; j++)
                            scanf("%d", &adj[i][j]);
            printf("DFS Traversal: ");
            depth_first_search(adj,visited,0);
            printf("\n");
            return 0;
    }
```

Output

```
Enter the adjacency matrix:
0 1 0 1 0
1 0 1 1 0
0 1 0 0 1
1 1 0 0 1
0 0 1 1 0
DFS Traversal: A -> C -> E ->
```

13.7 TOPOLOGICAL SORTING

Topological sort of a directed acyclic graph (DAG) G is defined as a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more number of topological sorts.

A topological sort of a DAG G is an ordering of the vertices of G such that if G contains an edge (u, v), then u appears before v in the ordering. Note that topological sort is possible only on directed acyclic graphs that do not have any cycles. For a DAG that contains cycles, no linear ordering of its vertices is possible.

In simple words, a topological ordering of a DAG G is an ordering of its vertices such that any directed path in G traverses the vertices in increasing order.

Topological sorting is widely used in scheduling applications, jobs, or tasks. The jobs that have to be completed are represented by nodes, and there is an edge from node u to v if job u must be completed before job v can be started. A topological sort of such a graph gives an order in which the given jobs must be performed.

Example 13.3 Consider three DAGs shown in Fig. 13.24 and their possible topological sorts.

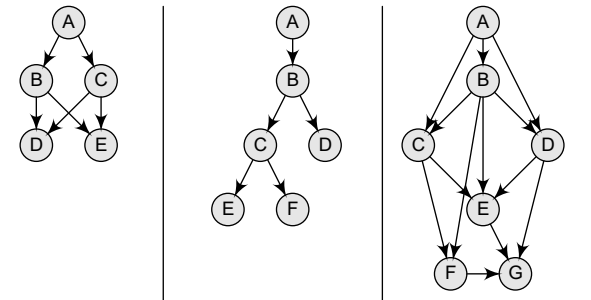


Figure 13.24 Topological sort

One main property of a DAG is that more the number of edges in a DAG, fewer the number of topological orders it has. This is because each edge (u, v) forces node u to occur before v, which restricts the number of valid permutations of the nodes.

Algorithm

The algorithm for the topological sort of a graph (Fig. 13.25) that has no cycles focuses on selecting a node N with zero in-degree, that is, a node that has no predecessor. The two main steps involved in the topological sort algorithm include:

- Selecting a node with zero in-degree
- Deleting N from the graph along with its edges

```
Step 1: Find the in-degree INDEG(N) of every node
        in the graph
Step 2: Enqueue all the nodes with a zero in-degree
Step 3: Repeat Steps 4 and 5 until the QUEUE is empty
Step 4:   Remove the front node N of the QUEUE by setting
          FRONT = FRONT + 1
Step 5:   Repeat for each neighbour M of node N:
           a) Delete the edge from N to M by setting
              INDEG(M) = INDEG(M) - 1
           b) IF INDEG(M) = 0, then Enqueue M, that is,
              add M to the rear of the queue
           [END OF INNER LOOP]
     [END OF LOOP]
Step 6: Exit
```

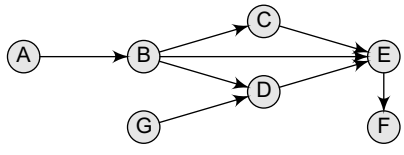
Figure 13.25 Algorithm for topological sort

We will use a QUEUE to hold the nodes with zero in-degree. The order in which the nodes will be deleted from the graph will depend on the sequence in which the nodes are inserted in the QUEUE. Then, we will use a variable INDEG, where INDEG(N) will represent the in-degree of node N.

Notes

1. The in-degree can be calculated in two ways—either by counting the incoming edges from the graph or traversing through the adjacency list.
2. The running time of the algorithm for topological sorting can be given linearly as the number of nodes plus the number of edges O(|V|+|E|).

Example 13.4 Consider a directed acyclic graph G given in Fig. 13.26. We use the algorithm given above to find a topological sort T of G. The steps are given as below:



Adjacency lists
A: B
B: C, D, E
C: E
D: E
E: F
G: D

Figure 13.26 Graph G

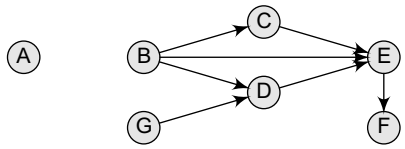
Step 1: Find the in-degree INDEG(N) of every node in the graph
INDEG(A) = 0 INDEG(B) = 1 INDEG(C) = 1 INDEG(D) = 2
INDEG(E) = 3 INDEG(F) = 1 INDEG(G) = 0

Step 2: Enqueue all the nodes with a zero in-degree
FRONT = 1 REAR = 2 QUEUE = A, G

Step 3: Remove the front element A from the queue by setting FRONT = FRONT + 1, so
FRONT = 2 REAR = 2 QUEUE = A, G

Step 4: Set INDEG(B) = INDEG(B) – 1, since B is the neighbour of A. Note that INDEG(B) is 0, so add it on the queue. The queue now becomes
FRONT = 2 REAR = 3 QUEUE = A, G, B

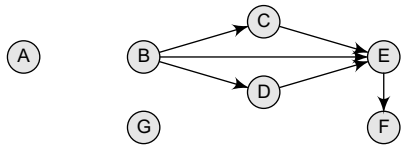
Delete the edge from A to B. The graph now becomes as shown in the figure below



Step 5: Remove the front element B from the queue by setting FRONT = FRONT + 1, so
FRONT = 2 REAR = 3 QUEUE = A, G, B

Step 6: Set INDEG(D) = INDEG(D) – 1, since D is the neighbour of G. Now,
INDEG(C) = 1 INDEG(D) = 1 INDEG(E) = 3 INDEG(F) = 1

Delete the edge from G to D. The graph now becomes as shown in the figure below

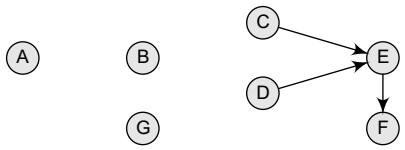


Step 7: Remove the front element B from the queue by setting FRONT = FRONT + 1, so
FRONT = 4 REAR = 3 QUEUE = A, G, B

Step 8: Set INDEG(C) = INDEG(C) – 1, INDEG(D) = INDEG(D) – 1, INDEG(E) = INDEG(E) – 1, since C, D, and E are the neighbours of B. Now,
INDEG(C) = 0, INDEG(D) = 1 and INDEG(E) = 2

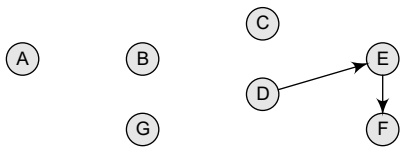
Step 9: Since the in-degree of node C and D is zero, add C and D at the rear of the queue. The queue can be given as below:
FRONT = 4 REAR = 5 QUEUE = A, G, B, C, D

The graph now becomes as shown in the figure below



Step 10: Remove the front element C from the queue by setting FRONT = FRONT + 1, so
FRONT = 5 REAR = 5 QUEUE = A, G, B, C, D

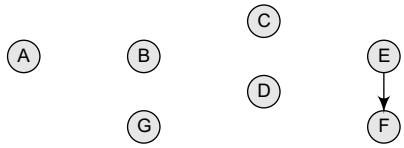
Step 11: Set INDEG(E) = INDEG(E) – 1, since E is the neighbour of C. Now, INDEG(E) = 1
The graph now becomes as shown in the figure below



Step 12: Remove the front element D from the queue by setting FRONT = FRONT + 1, so
FRONT = 6 REAR = 5 QUEUE = A, B, G, C, D

Step 13: Set INDEG(E) = INDEG(E) – 1, since E is the neighbour of D. Now, INDEG(E) = 0, so add E to the queue. The queue now becomes.
FRONT = 6 REAR = 6 QUEUE = A, G, B, C, D, E

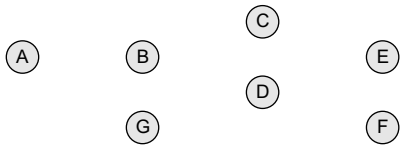
Step 14: Delete the edge between D an E. The graph now becomes as shown in the figure below



Step 15: Remove the front element D from the queue by setting FRONT = FRONT + 1, so
FRONT = 7 REAR = 6 QUEUE = A, G, B, C, D, E

Step 16: Set INDEG(F) = INDEG(F) – 1, since F is the neighbour of E. Now INDEG(F) = 0, so add F to the queue. The queue now becomes,
FRONT = 7 REAR = 7 QUEUE = A, G, B, C, D, E, F

Step 17: Delete the edge between E and F. The graph now becomes as shown in the figure below



There are no more edges in the graph and all the nodes have been added to the queue, so the topological sort T of G can be given as: A, G, B, C, D, E, F. When we arrange these nodes in a sequence, we find that if there is an edge from u to v, then u appears before v.

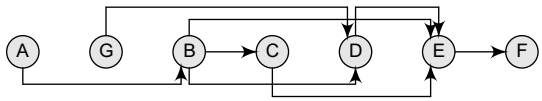


Figure 13.27 Topological sort of G

Programming Example

4. Write a program to implement topological sorting.

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int n,adj[MAX][MAX];
int front = -1,rear = -1,queue[MAX];
void create_graph(void);
void display();
void insert_queue(int);
int delete_queue(void);
int find_indegree(int);
void main()
{
        int node,j = 0,del_node, I;
        int topsort[MAX],indeg[MAX];
        create_graph();
        printf("\n The adjacency matrix is:");
        display();
/*Find the in-degree of each node*/
for(node = 1; node <= n; node++)
{
        indeg[node] = find_indegree(node);
        if( indeg[node] == 0 )
```

```
                insert_queue(node);
        }
        while(front <= rear) /*Continue loop until queue is empty */
        {
                del_node = delete_queue();
                topsort[j] = del_node; /*Add the deleted node to topsort*/
                j++;
        /*Delete the del_node edges */
        for(node = 1; node <= n; node++)
        {
                if(adj[del_node][node] == 1 )
                {
                        adj[del_node][node] = 0;
                        indeg[node] = indeg[node] - 1;
                        if(indeg[node] == 0)
                        insert_queue(node);
                }
           }
        }
        printf("The topological sorting can be given as :\n");
        for(node=0;i<j;node++)
                printf("%d ",topsort[node]);
        }
        void create_graph()
        {
                int i,max_edges,org,dest;
                printf("\n Enter the number of vertices: ");
                scanf("%d",&n);
                max_edges = n*(n - 1);
                for(i = 1; i <= max_edges; i++)
                {
                        printf("\n Enter edge %d(0 to quit): ",i);
                        scanf("%d %d",&org,&dest);
                        if((org == 0) && (dest == 0))
                        break;
                        if( org > n || dest > n || org <= 0 || dest <= 0)
                        {
                                printf("\n Invalid edge");
                                i--;
                        }
                        else
                                adj[org][dest] = 1;
                }
        }
        void display()
        {
                int i,j;
                for(i=1;i<=n;i++)
                {
                        printf("\n");
                        for(j=1;j<=n;j++)
                                printf("%3d",adj[i][j]);
                }
        }
        void insert_queue(int node)
        {
                if (rear==MAX-1)
                        printf("\n OVERFLOW ");
                else
                {
                        if (front == -1) /*If queue is initially empty */
```

```
                        front=0;
                queue[++rear] = node ;
        }
}
int delete_queue()
{
        int del_node;
        if (front == -1 || front > rear)
        {
                printf("\n UNDERFLOW ");
                return ;
        }
        else
        {
                del_node = queue[front++];
                return del_node;
        }
}
int find_indegree(int node)
{
                int i,in_deg = 0;
                for(i = 1; i <= n; i++)
                {
                        if( adj[i][node] == 1 )
                                in_deg++;
                }
                return in_deg;
}
```

Output

```
Enter number of vertices: 7
Enter edge 1(0 to quit): 1 2
Enter edge 2(0 to quit): 2 3
Enter edge 3(0 to quit): 2 5
Enter edge 4(0 to quit): 2 4
Enter edge 5(0 to quit): 3 5
Enter edge 6(0 to quit): 4 5
Enter edge 7(0 to quit): 5 6
Enter edge 8(0 to quit): 7 4
The topological sorting can be given as:
1 7 2 3 4 5 6
```

13.8 SHORTEST PATH ALGORITHMS

In this section, we will discuss three different algorithms to calculate the shortest path between the vertices of a graph G. These algorithms include:

- Minimum spanning tree
- Dijkstra's algorithm
- Warshall's algorithm

While the first two use an adjacency list to find the shortest path, Warshall's algorithm uses an adjacency matrix to do the same.

13.8.1 Minimum Spanning Trees

A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together. A graph G can have many different spanning trees. We can assign *weights* to each edge (which is a number that represents how unfavourable the edge is), and use it to assign a weight to a spanning tree by calculating the sum of the weights of the edges in that spanning

tree. A *minimum spanning tree* (MST) is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree. In other words, a minimum spanning tree is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

An Analogy

Take an analogy of a cable TV company laying cable in a new neighbourhood. If it is restricted to bury the cable only along particular paths, then we can make a graph that represents the points that are connected by those paths. Some paths may be more expensive (due to their length or the depth at which the cable should be buried) than the others. We can represent these paths by edges with larger weights.

Therefore, a spanning tree for such a graph would be a subset of those paths that has no cycles but still connects to every house. Many distinct spanning trees can be obtained from this graph, but a minimum spanning tree would be the one with the lowest total cost.

Properties

Possible multiplicity There can be multiple minimum spanning trees of the same weight. Particularly, if all the weights are the same, then every spanning tree will be minimum.

Uniqueness When each edge in the graph is assigned a different weight, then there will be only one unique minimum spanning tree.

Minimum-cost subgraph If the edges of a graph are assigned *non-negative* weights, then a minimum spanning tree is in fact the minimum-cost subgraph or a tree that connects all vertices.

Cycle property If there exists a cycle `c` in the graph `G` that has a weight larger than that of other edges of `c`, then this edge cannot belong to an MST.

Usefulness Minimum spanning trees can be computed quickly and easily to provide optimal solutions. These trees create a sparse subgraph that reflects a lot about the original graph.

Simplicity The minimum spanning tree of a weighted graph is nothing but a spanning tree of the graph which comprises of `n-1` edges of minimum total weight. Note that for an unweighted graph, any spanning tree is a minimum spanning tree.

Example 13.5 Consider an unweighted graph `G` given below (Fig. 13.28). From `G`, we can draw many distinct spanning trees. Eight of them are given here. For an unweighted graph, every spanning tree is a minimum spanning tree.

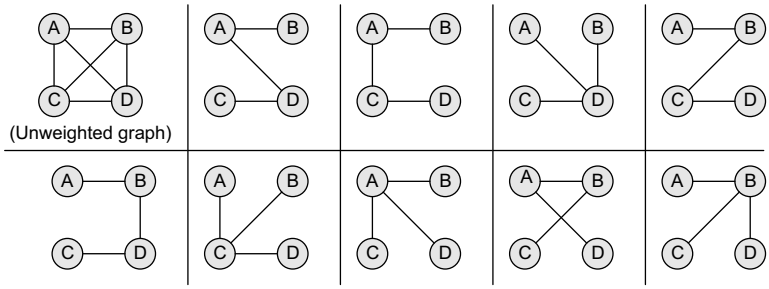


Figure 13.28 Unweighted graph and its spanning trees

Example 13.6 Consider a weighted graph `G` shown in Fig. 13.29. From `G`, we can draw three distinct spanning trees. But only a single minimum spanning tree can be obtained, that is, the one that has the minimum weight (cost) associated with it.

Of all the spanning trees given in Fig. 13.29, the one that is highlighted is called the minimum spanning tree, as it has the lowest cost associated with it.

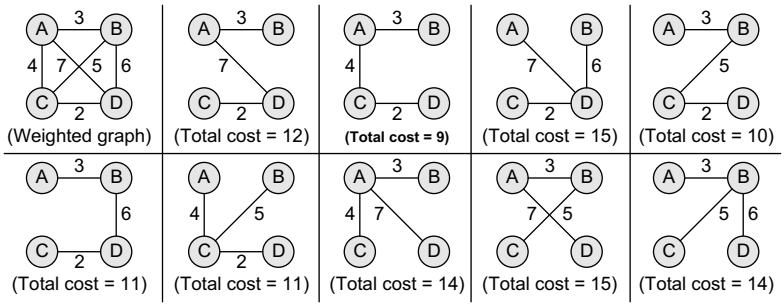


Figure 13.29 Weighted graph and its spanning trees

Applications of Minimum Spanning Trees

1. MSTs are widely used for designing networks. For instance, people separated by varying distances wish to be connected together through a telephone network. A minimum spanning tree is used to determine the least costly paths with no cycles in this network, thereby providing a connection that has the minimum cost involved.
2. MSTs are used to find airline routes. While the vertices in the graph denote cities, edges represent the routes between these cities. No doubt, more the distance between the cities, higher will be the amount charged. Therefore, MSTs are used to optimize airline routes by finding the least costly path with no cycles.
3. MSTs are also used to find the cheapest way to connect terminals, such as cities, electronic components or computers via roads, airlines, railways, wires or telephone lines.
4. MSTs are applied in routing algorithms for finding the most efficient path.

13.8.2 Prim's Algorithm

Prim's algorithm is a greedy algorithm that is used to form a minimum spanning tree for a connected weighted undirected graph. In other words, the algorithm builds a tree that includes every vertex and a subset of the edges in such a way that the total weight of all the edges in the tree is minimized. For this, the algorithm maintains three sets of vertices which can be given as below:

- ***Tree vertices*** Vertices that are a part of the minimum spanning tree T.
- ***Fringe vertices*** Vertices that are currently not a part of T, but are adjacent to some tree vertex.
- ***Unseen vertices*** Vertices that are neither tree vertices nor fringe vertices fall under this category.

The steps involved in the Prim's algorithm are shown in Fig. 13.30.

```
Step 1: Select a starting vertex
Step 2: Repeat Steps 3 and 4 until there are fringe vertices
Step 3:     Select an edge e connecting the tree vertex and
            fringe vertex that has minimum weight
Step 4:     Add the selected edge and the vertex to the
            minimum spanning tree T
        [END OF LOOP]
Step 5: EXIT
```

Figure 13.30 Prim's algorithm

- Choose a starting vertex.
- Branch out from the starting vertex and during each iteration, select a new vertex and an edge. Basically, during each iteration of the algorithm, we have to select a vertex from the fringe vertices in such a way that the edge connecting the tree vertex and the new vertex has the minimum weight assigned to it.

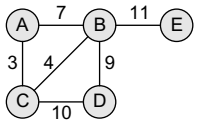


Figure 13.31 Graph G

The running time of Prim's algorithm can be given as O(E log V) where E is the number of edges and V is the number of vertices in the graph.

Example 13.7 Construct a minimum spanning tree of the graph given in Fig. 13.31.

Step 1: Choose a starting vertex A.

Step 2: Add the fringe vertices (that are adjacent to A). The edges connecting the vertex and fringe vertices are shown with dotted lines.

Step 3: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting A and C has less weight, add C to the tree. Now C is not a fringe vertex but a tree vertex.

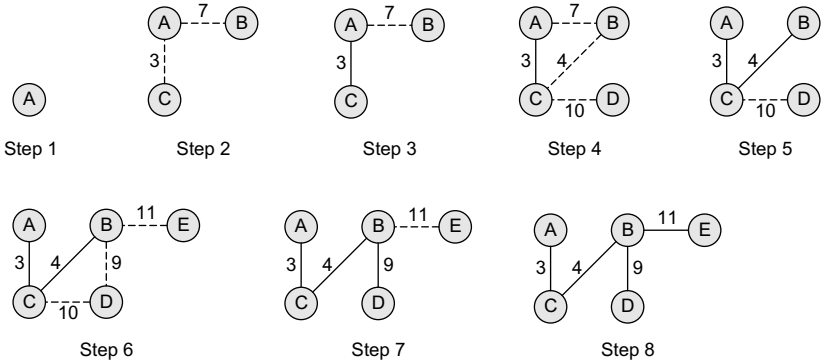
Step 4: Add the fringe vertices (that are adjacent to C).

Step 5: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting C and B has less weight, add B to the tree. Now B is not a fringe vertex but a tree vertex.

Step 6: Add the fringe vertices (that are adjacent to B).

Step 7: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting B and D has less weight, add D to the tree. Now D is not a fringe vertex but a tree vertex.

Step 8: Note, now node E is not connected, so we will add it in the tree because a minimum spanning tree is one in which all the n nodes are connected with n-1 edges that have minimum weight. So, the minimum spanning tree can now be given as,



Example 13.8 Construct a minimum spanning tree of the graph given in Fig. 13.32. Start the Prim's algorithm from vertex D.

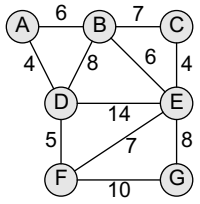
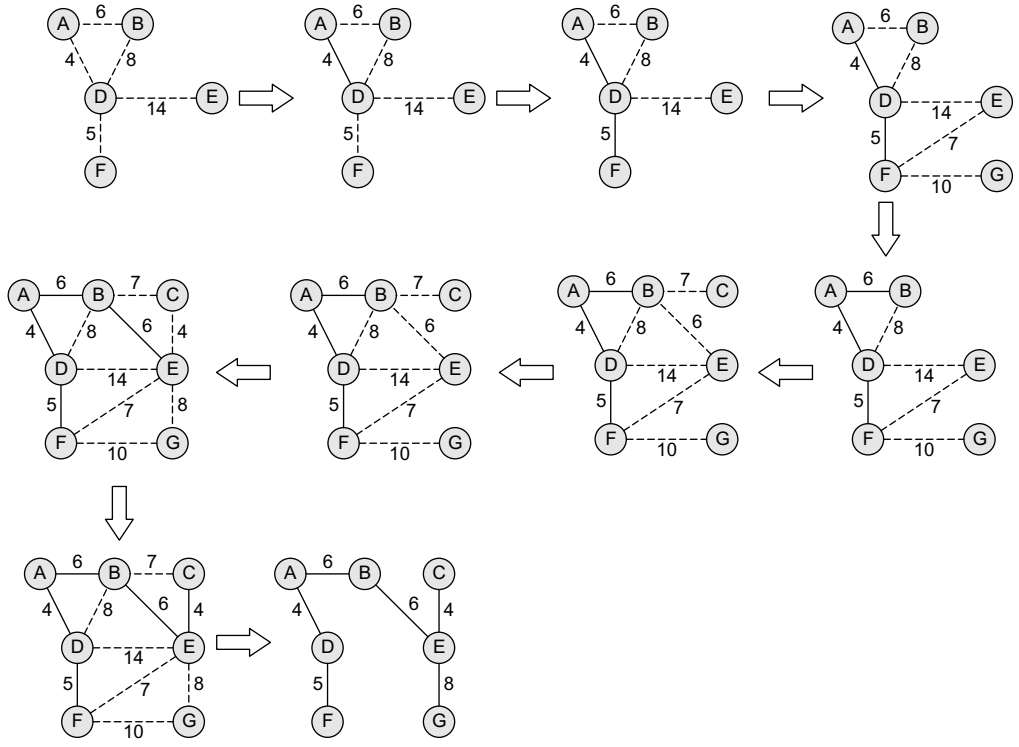


Figure 13.32 Graph G



13.8.3 Kruskal's Algorithm

Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph. The algorithm aims to find a subset of the edges that forms a tree that includes every vertex. The total weight of all the edges in the tree is minimized. However, if the graph is not connected, then it finds a *minimum spanning forest*. Note that a forest is a collection of trees. Similarly, a minimum spanning forest is a collection of minimum spanning trees.

Kruskal's algorithm is an example of a greedy algorithm, as it makes the locally optimal choice at each stage with the hope of finding the global optimum. The algorithm is shown in Fig. 13.33.

```
Step 1: Create a forest in such a way that each graph is a separate
        tree.
Step 2: Create a priority queue Q that contains all the edges of the
        graph.
Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY
Step 4:      Remove an edge from Q
Step 5: IF the edge obtained in Step 4 connects two different trees,
        then Add it to the forest (for combining two trees into one
        tree).
        ELSE
             Discard the edge
Step 6: END
```

Figure 13.33 Kruskal's algorithm

In the algorithm, we use a priority queue `Q` in which edges that have minimum weight takes a priority over any other edge in the graph. When the Kruskal's algorithm terminates, the forest has only one component and forms a minimum spanning tree of the graph. The running time of Kruskal's algorithm can be given as `O(E log V)`, where `E` is the number of edges and `V` is the number of vertices in the graph.

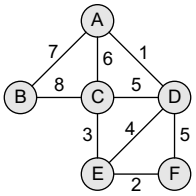


Figure 13.34

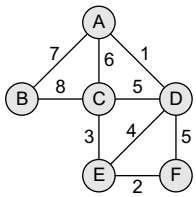
Example 13.9 Apply Kruskal's algorithm on the graph given in Fig. 13.34.

Initially, we have F = {{A}, {B}, {C}, {D}, {E}, {F}}

MST = {}

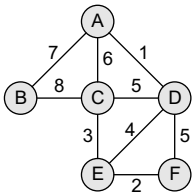
Q = {(A, D), (E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 1: Remove the edge `(A, D)` from `Q` and make the following changes:



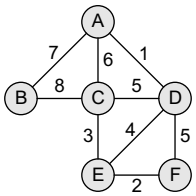
F = {{A, D}, {B}, {C}, {E}, {F}}
MST = {A, D}
Q = {(E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 2: Remove the edge `(E, F)` from `Q` and make the following changes:



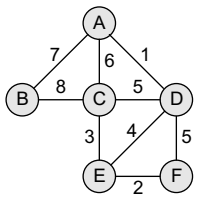
F = {{A, D}, {B}, {C}, {E, F}}
MST = {(A, D), (E, F)}
Q = {(C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 3: Remove the edge `(C, E)` from `Q` and make the following changes:



F = {{A, D}, {B}, {C, E, F}}
MST = {(A, D), (C, E), (E, F)}
Q = {(E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 4: Remove the edge `(E, D)` from `Q` and make the following changes:



F = {{A, C, D, E, F}, {B}}
MST = {(A, D), (C, E), (E, F), (E, D)}
Q = {(C, D), (D, F), (A, C), (A, B), (B, C)}

Step 5: Remove the edge `(C, D)` from `Q`. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting `(A, D, C, E, F)` to `B` will be added to the MST. Therefore,

$$F = \{\{A, C, D, E, F\}, \{B\}\}$$
$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$
$$Q = \{(D, F), (A, C), (A, B), (B, C)\}$$

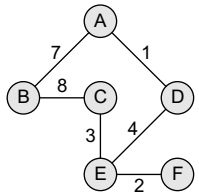
Step 6: Remove the edge `(D, F)` from `Q`. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting `(A, D, C, E, F)` to `B` will be added to the MST.

$$F = \{\{A, C, D, E, F\}, \{B\}\}$$
$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$
$$Q = \{(A, C), (A, B), (B, C)\}$$

Step 7: Remove the edge `(A, C)` from `Q`. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting `(A, D, C, E, F)` to `B` will be added to the MST.

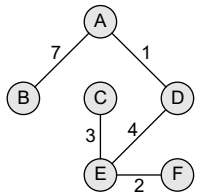
$$F = \{\{A, C, D, E, F\}, \{B\}\}$$
$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$
$$Q = \{(A, B), (B, C)\}$$

Step 8: Remove the edge `(A, B)` from `Q` and make the following changes:



F = {A, B, C, D, E, F}
MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}
Q = {(B, C)}

Step 9: The algorithm continues until `Q` is empty. Since the entire forest has become one tree, all the remaining edges will simply be discarded. The resultant MS can be given as shown below.



F = {A, B, C, D, E, F}
MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}
Q = {}

Programming Example

5. Write a program which finds the cost of a minimum spanning tree.

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
int adj[MAX][MAX], tree[MAX][MAX], n;
void readmatrix()
{
        int i, j;
        printf("\n Enter the number of nodes in the Graph : ");
        scanf("%d", &n);
        printf("\n Enter the adjacency matrix of the Graph");
        for (i = 1; i <= n; i++)
                for (j = 1; j <= n; j++)
                        scanf("%d", &adj[i][j]);
}
int spanningtree(int src)
```

```
{
        int visited[MAX], d[MAX], parent[MAX];
        int i, j, k, min, u, v, cost;
        for (i = 1; i <= n; i++)
        {
                d[i] = adj[src][i];
                visited[i] = 0;
                parent[i] = src;
        }
        visited[src] = 1;
        cost = 0;
        k = 1;
        for (i = 1; i < n; i++)
        {
                min = 9999;
                for (j = 1; j <= n; j++)
                {
                        if (visited[j]==0 && d[j] < min)
                        {
                                min = d[j];
                                u = j;
                                cost += d[u];
                        }
                }
                visited[u] = 1;
                //cost = cost + d[u];
                tree[k][1] = parent[u];
                tree[k++][2] = u;
                for (v = 1; v <= n; v++)
                        if (visited[v]==0 && (adj[u][v] < d[v]))
                        {
                                d[v] = adj[u][v];
                                parent[v] = u;
                        }
        }
        return cost;
}
void display(int cost)
{
        int i;
        printf("\n The Edges of the Mininum Spanning Tree are");
        for (i = 1; i < n; i++)
                printf(" %d %d \n", tree[i][1], tree[i][2]);
        printf("\n The Total cost of the Minimum Spanning Tree is : %d", cost);
}
main()
{
        int source, treecost;
        readmatrix();
        printf("\n Enter the Source : ");
        scanf("%d", &source);
        treecost = spanningtree(source);
        display(treecost);
        return 0;
}
```

Output

```
Enter the number of nodes in the Graph : 4
Enter the adjacency matrix : 0    1       1       0
0       0       0       1
0       1       0       0
```

```
1       0       1       0
Enter the source : 1
The edges of the Minimum Spanning Tree are    1    4
4       2
2       3
The total cost of the Minimum Spanning Tree is : 1
```

13.8.4 Dijkstra's Algorithm

Dijkstra's algorithm, given by a Dutch scientist Edsger Dijkstra in 1959, is used to find the shortest path tree. This algorithm is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First).

Given a graph G and a source node A, the algorithm is used to find the shortest path (one having the lowest cost) between A (source node) and every other node. Moreover, Dijkstra's algorithm is also used for finding the costs of the shortest paths from a source node to a destination node.

For example, if we draw a graph in which nodes represent the cities and weighted edges represent the driving distances between pairs of cities connected by a direct road, then Dijkstra's algorithm when applied gives the shortest route between one city and all other cities.

Algorithm

Dijkstra's algorithm is used to find the length of an *optimal* path between two nodes in a graph. The term *optimal* can mean anything, shortest, cheapest, or fastest. If we start the algorithm with an initial node, then the distance of a node Y can be given as the distance from the initial node to that node. Figure 13.35 explains the Dijkstra's algorithm.

```
1. Select the source node also called the initial node
2. Define an empty set N that will be used to hold nodes to which a shortest
   path has been found.
3. Label the initial node with 0, and insert it into N.
4. Repeat Steps 5 to 7 until the destination node is in N or there are no more
   labelled nodes in N.
5. Consider each node that is not in N and is connected by an edge from
   the newly inserted node.
6. (a) If the node that is not in N has no label then SET the label of the
       node = the label of the newly inserted node + the length of the edge.
   (b) Else if the node that is not in N was already labelled, then SET its new
       label = minimum (label of newly inserted vertex + length of edge, old
       label)
7. Pick a node not in N that has the smallest label assigned to it and add it
   to N.
```

Figure 13.35 Dijkstra's algorithm

Dijkstra's algorithm labels every node in the graph where the labels represent the distance (cost) from the source node to that node. There are two kinds of labels: *temporary* and *permanent*. Temporary labels are assigned to nodes that have not been reached, while permanent labels are given to nodes that have been reached and their distance (cost) to the source node is known. A node must be a permanent label or a temporary label, but not both.

The execution of this algorithm will produce either of the following two results:

1. If the destination node is labelled, then the label will in turn represent the distance from the source node to the destination node.
2. If the destination node is not labelled, then there is no path from the source to the destination node.

Example 13.10 Consider the graph G given in Fig. 13.36. Taking D as the initial node, execute the Dijkstra's algorithm on it.

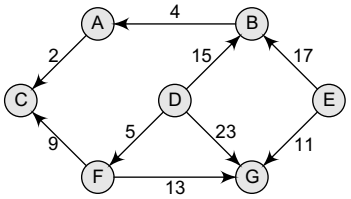


Figure 13.36 Graph G

Step 1: Set the label of D = 0 and N = {D}.
Step 2: Label of D = 0, B = 15, G = 23, and F = 5. Therefore, N = {D, F}.
Step 3: Label of D = 0, B = 15, G has been re-labelled 18 because minimum (5 + 13, 23) = 18, C has been re-labelled 14 (5 + 9). Therefore, N = {D, F, C}.
Step 4: Label of D = 0, B = 15, G = 18. Therefore, N = {D, F, C, B}.
Step 5: Label of D = 0, B = 15, G = 18 and A = 19 (15 + 4). Therefore, N = {D, F, C, B, G}.
Step 6: Label of D = 0 and A = 19. Therefore, N = {D, F, C, B, G, A}

Note that we have no labels for node E; this means that E is not reachable from D. Only the nodes that are in N are reachable from B.

The running time of Dijkstra's algorithm can be given as $O(|V|^2+|E|)=O(|V|^2)$ where V is the set of vertices and E in the graph.

Difference between Dijkstra's Algorithm and Minimum Spanning Tree

Minimum spanning tree algorithm is used to traverse a graph in the most efficient manner, but Dijkstra's algorithm calculates the distance from a given vertex to every other vertex in the graph.

Dijkstra's algorithm is very similar to Prim's algorithm. Both the algorithms begin at a specific node and extend outward within the graph, until all other nodes in the graph have been reached. The point where these algorithms differ is that while Prim's algorithm stores a minimum cost edge, Dijkstra's algorithm stores the total cost from a source node to the current node. Moreover, Dijkstra's algorithm is used to store the summation of minimum cost edges, while Prim's algorithm stores at most one minimum cost edge.

13.8.5 Warshall's Algorithm

If a graph G is given as G=(V, E), where V is the set of vertices and E is the set of edges, the path matrix of G can be found as, $P = A + A^2 + A^3 + \ldots + A^n$. This is a lengthy process, so Warshall has given a very efficient algorithm to calculate the path matrix. Warshall's algorithm defines matrices $P_0, P_1, P_2, \ldots, P_n$ as given in Fig. 13.37.

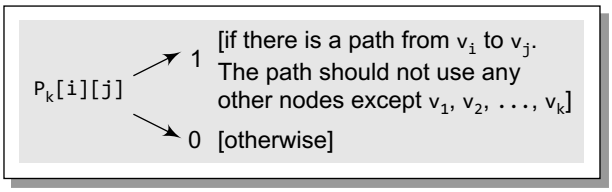


Figure 13.37 Path matrix entry

This means that if $P_0[i][j] = 1$, then there exists an edge from node v_i to v_j.

If $P_1[i][j] = 1$, then there exists an edge from v_i to v_j that does not use any other vertex except v_1.

If $P_2[i][j] = 1$, then there exists an edge from v_i to v_j that does not use any other vertex except v_1 and v_2.

Note that P_0 is equal to the adjacency matrix of G. If G contains n nodes, then $P_n = P$ which is the path matrix of the graph G.

From the above discussion, we can conclude that $P_k[i][j]$ is equal to 1 only when either of the two following cases occur:

- There is a path from v_i to v_j that does not use any other node except $v_1, v_2, \ldots, v_{k-1}$. Therefore, $P_{k-1}[i][j] = 1$.

- There is a path from v_i to v_k and a path from v_k to v_j where all the nodes use $v_1, v_2, \ldots, v_{k-1}$. Therefore,

$$P_{k-1}[i][k] = 1 \text{ AND } P_{k-1}[k][j] = 1$$

Hence, the path matrix P_n can be calculated with the formula given as:

$$P_k[i][j] = P_{k-1}[i][j] \vee (P_{k-1}[i][k] \wedge P_{k-1}[k][j])$$

where V indicates logical OR operation and Λ indicates logical AND operation.

Figure 13.38 shows the Warshall's algorithm to find the path matrix P using the adjacency matrix A.

```
Step 1: [INITIALIZE the Path Matrix] Repeat Step 2 for I = 0 to n-1,
        where n is the number of nodes in the graph
Step 2:     Repeat Step 3 for J = 0 to n-1
Step 3:           IF A[I][J] = 0, then SET P[I][J] = 0
                  ELSE P[I][J] = 1
              [END OF LOOP]
      [END OF LOOP]
Step 4: [Calculate the path matrix P] Repeat Step 5 for K = 0 to n-1
Step 5:     Repeat Step 6 for I = 0 to n-1
Step 6:           Repeat Step 7 for J=0 to n-1
Step 7:                     SET P_K[I][J] = P_K-1[I][J] V (P_K-1[I][K]
                            Λ P_K-1[K][J])
Step 8: EXIT
```

Figure 13.38 Warshall's algorithm

Example 13.11 Consider the graph in Fig. 13.39 and its adjacency matrix A. We can straightaway calculate the path matrix P using the Warshall's algorithm.

The path matrix P can be given in a single step as:

$$P = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

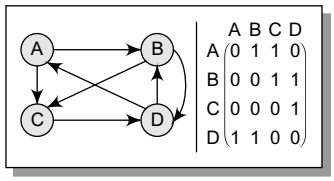


Figure 13.39 Graph G and its path matrix P

Thus, we see that calculating A, A2, A3, A4, ..., A5 to calculate P is a very slow and inefficient technique as compared to the Warshall's technique.

Programming Example

6. Write a program to implement Warshall's algorithm to find the path matrix.

```
#include <stdio.h>
#include <conio.h>
void read (int mat[5][5], int n);
void display (int mat[5][5], int n);
void mul(int mat[5][5], int n);
int main()
{
        int adj[5][5], P[5][5], n, i, j, k;
        clrscr();
```

```
        printf("\n Enter the number of nodes in the graph : ");
        scanf("%d", &n);
        printf("\n Enter the adjacency matrix : ");
        read(adj, n);
        clrscr();
        printf("\n The adjacency matrix is : ");
        display(adj, n);
        for(i=0;i<n;i++)
        {
                for(j=0;j<n;j++)
                {
                        if(adj[i][j] == 0)
                                P[i][j] = 0;
                        else
                                P[i][j] = 1;
                }
         }
        for(k=0; k<n;k++)
        {
                for(i=0;i<n;i++)
                {
                        for(j=0;j<n;j++)
                        P[i][j] = P[i][j] | ( P[i][k] & P[k][j]);
                }
        }
        printf("\n The Path Matrix is :");
        display (P, n);
        getch();
        return 0;
}
void read(int mat[5][5], int n)
{
        int i, j;
        for(i=0;i<n;i++)
        {
                for(j=0;j<n;j++)
                {
                        printf("\n mat[%d][%d] = ", i, j);
                        scanf("%d", &mat[i][j]);
                }
        }
}
void display(int mat[5][5], int n)
{
        int i, j;
        for(i=0;i<n;i++)
        printf("\n");
                for(j=0;j<n;j++)
                        printf("%d\t", mat[i][j]);
        }
}
```

Output

```
The adjacency matrix is
0       1       1       0
0       0       1       1
0       0       0       1
1       1       0       0
```

```
The Path Matrix is
1       1       1       1
1       1       1       1
1       1       1       1
1       1       1       1
```

13.8.6 Modified Warshall's Algorithm

Warshall's algorithm can be modified to obtain a matrix that gives the shortest paths between the nodes in a graph G. As an input to the algorithm, we take the adjacency matrix A of G and replace all the values of A which are zero by infinity (∞). Infinity (∞) denotes a very large number and indicates that there is no path between the vertices. In Warshall's modified algorithm, we obtain a set of matrices Q_0, Q_1, Q_2, ..., Q_m using the formula given below.

$Q_k[i][j] = Minimum(M_{k-1}[i][j], M_{k-1}[i][k] + M_{k-1}[k][j])$

Q_0 is exactly the same as A with a little difference that every element having a zero value in A is replaced by (∞) in Q_0. Using the given formula, the matrix Q_n will give the path matrix that has the shortest path between the vertices of the graph. Warshall's modified algorithm is shown in Fig. 13.40.

```
Step 1: [Initialize the Shortest Path Matrix, Q] Repeat Step 2 for I = 0
        to n-1, where n is the number of nodes in the graph
Step 2:     Repeat Step 3 for J = 0 to n-1
Step 3:           IF A[I][J] = 0, then SET Q[I][J] = Infinity (or 9999)
                  ELSE Q[I][J] = A[I][j]
            [END OF LOOP]
      [END OF LOOP]
Step 4: [Calculate the shortest path matrix Q] Repeat Step 5 for K = 0
        to n-1
Step 5:     Repeat Step 6 for I = 0 to n-1
Step 6:           Repeat Step 7 for J=0 to n-1
Step 7:                     IF Q[I][J] <= Q[I][K] + Q[K][J]
                            SET Q[I][J] = Q[I][J]
                        ELSE SET Q[I][J] = Q[I][K] + Q[K][J]
                        [END OF IF]
            [END OF LOOP]
      [END OF LOOP]
      [END OF LOOP]
Step 8: EXIT
```

Figure 13.40 Modified Warshall's algorithm

Example 13.12 Consider the unweighted graph G given in Fig. 13.41 and apply Warshall's algorithm to it.

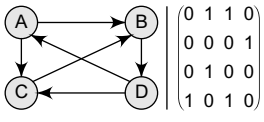


Figure 13.41 Graph G

$$\mathbf{Q0} = \begin{bmatrix} 9999 & 1 & 1 & 9999 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 1 & 9999 & 9999 \\ 1 & 9999 & 1 & 9999 \end{bmatrix} \quad \mathbf{Q1} = \begin{bmatrix} 9999 & 1 & 1 & 9999 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 1 & 9999 & 9999 \\ 1 & 2 & 1 & 9999 \end{bmatrix}$$

$$\mathbf{Q2} = \begin{bmatrix} 9999 & 1 & 1 & 2 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 1 & 9999 & 2 \\ 1 & 2 & 9999 & 3 \end{bmatrix} \quad \mathbf{Q3} = \begin{bmatrix} 9999 & 1 & 1 & 2 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 1 & 9999 & 2 \\ 1 & 2 & 9999 & 3 \end{bmatrix} \quad \mathbf{Q4} = \mathbf{Q} \begin{bmatrix} 3 & 1 & 1 & 2 \\ 2 & 3 & 2 & 1 \\ 3 & 1 & 3 & 2 \\ 1 & 2 & 1 & 3 \end{bmatrix}$$

Example 13.13 Consider a weighted graph G given in Fig. 13.42 and apply Warshall's shortest path algorithm to it.

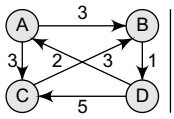


Figure 13.42 Graph G

$$\mathbf{Q0} = \begin{bmatrix} 9999 & 3 & 3 & 9999 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 9999 \\ 2 & 9999 & 5 & 9999 \end{bmatrix}$$

$$\mathbf{Q1} = \begin{bmatrix} 9999 & 3 & 3 & 9999 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 9999 \\ 2 & 5 & 5 & 9999 \end{bmatrix}$$

$$\mathbf{Q2} = \begin{bmatrix} 9999 & 3 & 3 & 4 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 4 \\ 2 & 5 & 5 & 6 \end{bmatrix}$$

$$\mathbf{Q3} = \begin{bmatrix} 9999 & 3 & 3 & 4 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 6 \\ 2 & 5 & 5 & 6 \end{bmatrix}$$

$$\mathbf{Q4} = \mathbf{Q}\begin{bmatrix} 6 & 3 & 3 & 4 \\ 3 & 6 & 6 & 1 \\ 6 & 3 & 9 & 4 \\ 2 & 5 & 5 & 6 \end{bmatrix}$$

Programming Example

7. Write a program to implement Warshall's modified algorithm to find the shortest path.

```
#include <stdio.h>
#include <conio.h>
#define INFINITY 9999
void read (int mat[5][5], int n);
void display(int mat[5][5], int n);
int main()
{
        int adj[5][5], Q[5][5], n, i, j, k;
        clrscr();
        printf("\n Enter the number of nodes in the graph : ");
        scanf("%d", &n);
        printf("\n Enter the adjacency matrix : ");
        read(adj, n);
        clrscr();
        printf("\n The adjacency matrix is : ");
        display(adj, n);
        for(i=0;i<n;i++)
        {
                for(j=0;j<n;j++)
                {
                        if(adj[i][j] == 0)
                                Q[i][j] = INFINITY;
                        else
                                Q[i][j] = adj[i][j];
                }
        }
        for(k=0; k<n;k++)
        {
                for(i=0;i<n;i++)
                {
```

```
                        for(j=0;j<n;j++)
                        {
                                if(Q[i][j] <= Q[i][k] + Q[k][j])
                                        Q[i][j] = Q[i][j];
                                else
                                        Q[i][j] = Q[i][k] + Q[k][j];
                        }
                }
                printf("\n\n");
                display(Q, n);
        }
        getch();
        return 0;
}
void read(int mat[5][5], int n)
{
        int i, j;
        for(i=0;i<n;i++)
        {
                for(j=0;j<n;j++)
                {
                        printf("\n mat[%d][%d] = ", i, j);
                        scanf("%d", &mat[i][j]);
                }
        }
}
void display(int mat[5][5], int n)
{
        int i, j;
        for(i=0;i<n;i++)
        {printf("\n");
                for(j=0;j<n;j++)
                        printf("%d\t", mat[i][j]);
        }
}
```

Output

```
6       3       3       4
3       6       6       1
6       3       9       4
1       5       5       6
```

13.9 APPLICATIONS OF GRAPHS

Graphs are constructed for various types of applications such as:

- In circuit networks where points of connection are drawn as vertices and component wires become the edges of the graph.
- In transport networks where stations are drawn as vertices and routes become the edges of the graph.
- In maps that draw cities/states/regions as vertices and adjacency relations as edges.
- In program flow analysis where procedures or modules are treated as vertices and calls to these procedures are drawn as edges of the graph.
- Once we have a graph of a particular concept, they can be easily used for finding shortest paths, project planning, etc.
- In flowcharts or control-flow graphs, the statements and conditions in a program are represented as nodes and the flow of control is represented by the edges.

- In state transition diagrams, the nodes are used to represent states and the edges represent legal moves from one state to the other.
- Graphs are also used to draw activity network diagrams. These diagrams are extensively used as a project management tool to represent the interdependent relationships between groups, steps, and tasks that have a significant impact on the project.

An Activity Network Diagram (AND) also known as an Arrow Diagram or a PERT (Program Evaluation Review Technique) is used to identify time sequences of events which are pivotal to objectives. It is also helpful when a project has multiple activities which need simultaneous management. ANDs help the project development team to create a realistic project schedule by drawing graphs that exhibit:

- the total amount of time needed to complete the project
- the sequence in which activities must be performed
- the activities that can be performed simultaneously
- the critical activities that must be monitored on a regular basis.

A sample AND is shown in Fig. 13.43.

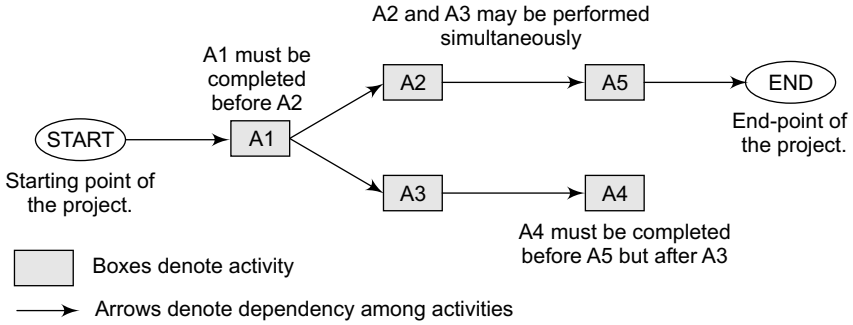


Figure 13.43 Activity network diagram

Points to Remember

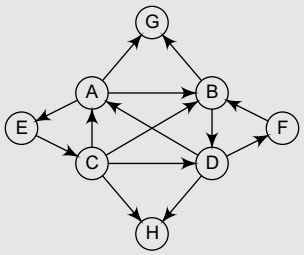
- A graph is basically a collection of vertices (also called nodes) and edges that connect these vertices.
- Degree of a node `u` is the total number of edges containing the node `u`. When the degree of a node is zero, it is also called an isolated node. A path `P` is known as a closed path if the edge has the same end-points. A closed simple path with length 3 or more is known as a cycle.
- A graph in which there exists a path between any two of its nodes is called a connected graph. An edge that has identical end-points is called a loop. The size of a graph is the total number of edges in it.
- The out-degree of a node is the number of edges that originate at `u`.
- The in-degree of a node is the number of edges that terminate at `u`. A node `u` is known as a sink if it has a positive in-degree but a zero out-degree.
- A transitive closure of a graph is constructed to answer reachability questions.
- Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The memory use of an adjacency matrix is $O(n^2)$, where `n` is the number of nodes in the graph.
- Topological sort of a directed acyclic graph `G` is defined as a linear ordering of its nodes in which each node comes before all the nodes to which it has outbound edges. Every DAG has one or more number of topological sorts.
- A vertex `v` of `G` is called an articulation point if removing `v` along with the edges incident to `v` results in a graph that has at least two connected components.
- A biconnected graph is defined as a connected graph that has no articulation vertices.

- Breadth-first search is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.
- The depth-first search algorithm progresses by expanding the starting node of G and thus going deeper and deeper until a goal node is found, or until a node that has no children is encountered.
- A spanning tree of a connected, undirected graph `G` is a sub-graph of `G` which is a tree that connects all the vertices together.
- Kruskal's algorithm is an example of a greedy algorithm, as it makes the locally optimal choice at each stage with the hope of finding the global optimum.
- Dijkstra's algorithm is used to find the length of an optimal path between two nodes in a graph.

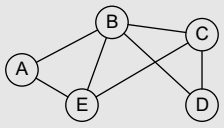
EXERCISES

Review Questions

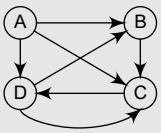
1. Explain the relationship between a linked list structure and a digraph.
2. What is a graph? Explain its key terms.
3. How are graphs represented inside a computer's memory? Which method do you prefer and why?
4. Consider the graph given below.
   (a) Write the adjacency matrix of G.
   (b) Write the path matrix of G.
   (c) Is the graph biconnected?
   (d) Is the graph complete?
   (e) Find the shortest path matrix using Warshall's algorithm.



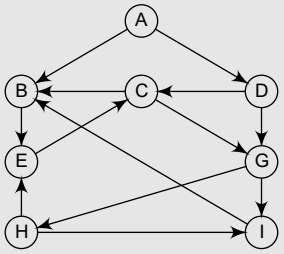
5. Explain the graph traversal algorithms in detail with example.
6. Draw a complete undirected graph having five nodes.
7. Consider the graph given below and find out the degree of each node.



8. Consider the graph given below. State all the simple paths from A to D, B to D, and C to D. Also, find out the in-degree and out-degree of each node. Is there any source or sink in the graph?



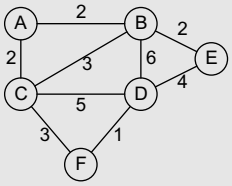
9. Consider the graph given below. Find out its depth-first and breadth-first traversal scheme.



10. Differentiate between depth-first search and breadth-first search traversal of a graph.
11. Explain the topological sorting of a graph `G`.
12. Define spanning tree.
13. When is a spanning tree called a minimum spanning tree? Take a weighted graph of your choice and find out its minimum spanning tree.
14. Explain Prim's algorithm.
15. Write a brief note on Kruskal's algorithm.
16. Write a short note on Dijkstra's algorithm.

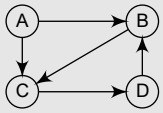
17. Differentiate between Dijkstra's algorithm and minimum spanning tree algorithm.

18. Consider the graph given below. Find the minimum spanning tree of this graph using (a) Prim's algorithm, (b) Kruskal's algorithm, and (c) Dijkstra's algorithm.



19. Briefly discuss Warshall's algorithm. Also, discuss its modified version.

20. Show the working of Floyd-Warshall's algorithm to find the shortest paths between all pairs of nodes in the following graph.



21. Write a short note on transitive closure of a graph.

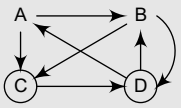
22. Given the adjacency matrix of a graph, write a program to calculate the degree of a node `N` in the graph.

23. Given the adjacency matrix of a graph, write a program to calculate the in-degree and the out-degree of a node `N` in the graph.

24. Given the adjacency matrix of a graph, write a function `isFullConnectedGraph` which returns 1 if the graph is fully connected and 0 otherwise.

25. In which kind of graph do we use topological sorting?

26. Consider the graph given below and show its adjacency list in the memory.



27. Consider the graph given in Question 26 and show the changes in the graph as well as its adjacency list when node `E` and edges `(A, E)` and `(C, E)` are added to it. Also, delete edge `(B, D)` from the graph.

28. Given the following adjacency matrix, draw the weighted graph.

$$\begin{pmatrix} 0 & 4 & 0 & 2 & 0 \\ 0 & 0 & 0 & 7 & 0 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

29. Consider five cities: (1) New Delhi, (2) Mumbai, (3) Chennai, (4) Bangalore, and (5) Kolkata, and a list of flights that connect these cities as shown in the following table. Use the given information to construct a graph.

Flight No.	Origin	Destination
101	2	3
102	3	2
103	5	3
104	3	4
105	2	5
106	5	2
107	5	1
108	1	4
109	5	4
110	4	5

Programming Exercises

1. Write a program to create and print a graph.
2. Write a program to determine whether there is at least one path from the source to the destination.

Multiple-choice Questions

1. An edge that has identical end-points is called a
(a) Multi-path (b) Loop
(c) Cycle (d) Multi-edge

2. The total number of edges containing the node u is called
(a) In-degree (b) Out-degree
(c) Degree (d) None of these

3. A graph in which there exists a path between any two of its nodes is called
(a) Complete graph (b) Connected graph
(c) Digraph (d) In-directed graph

4. The number of edges that originate at u are called
(a) In-degree (b) Out-degree
(c) Degree (d) source

5. The memory use of an adjacency matrix is
(a) $O(n)$ (b) $O(n^2)$
(c) $O(n^3)$ (d) $O(\log n)$

6. The term optimal can mean
(a) Shortest (b) Cheapest
(c) Fastest (d) All of these

7. How many articulation vertices does a biconnected graph contain?
(a) 0 (b) 1
(c) 2 (d) 3

True or False

1. Graph is a linear data structure.

2. In-degree of a node is the number of edges leaving that node.

3. The size of a graph is the total number of vertices in it.

4. A sink has a zero in-degree but a positive out-degree.

5. The space complexity of depth-first search is lower than that of breadth-first search.

6. A node is known as a sink if it has a positive out-degree but the in-degree = 0.

7. A directed graph that has no cycles is called a directed acyclic graph.

8. A graph `G` can have many different spanning trees.

9. Fringe vertices are not a part of `T`, but are adjacent to some tree vertex.

10. Kruskal's algorithm is an example of a greedy algorithm.

Fill in the Blanks

1. _____ has a zero degree.

2. In-degree of a node is the number of edges that _____ at `u`.

3. Adjacency matrix is also known as a _____.

4. A path `P` is known as a _____ path if the edge has the same end-points.

5. A graph with multiple edges and/or a loop is called a _____.

6. Vertices that are a part of the minimum spanning tree `T` are called _____.

7. A _____ of a graph is constructed to answer reachability questions.

8. An _____ is a vertex `v` of `G` if removing `v` along with the edges incident to `v` results in a graph that has at least two connected components.

9. A _____ graph is a connected graph that is not broken into disconnected pieces by deleting any single vertex.

10. An edge is called a _____ if removing that edge results in a disconnected graph.