Chapter 7

GRAPHS

Objectives

- Introduction
- Graph Terminology
- Graph Representation
- Traversing a Graph
- Path Matrix or Reachability Matrix
- Shortest Path Problem
- Spanning Tree
- Operations on Graph

7.1 Introduction

In this chapter, we investigate an important mathematical structure called graphs. Graph is a non linear data structure. Graphs have been used in wide variety of applications. Some of these applications are: analysis of electrical circuit, finding shortest routes, project planning, to represent highway structures, communication lines, railway lines and so on.

Definition

A Graph G consists of two sets, V and E where V is a finite and non empty set of vertices (or nodes) and E is a set of edges (or arcs). Thus a graph is defined as a collection of vertices and edges and it is represented by G = (V, E).

An *edge* is defined as a pair of vertices, which are adjacent to each other.

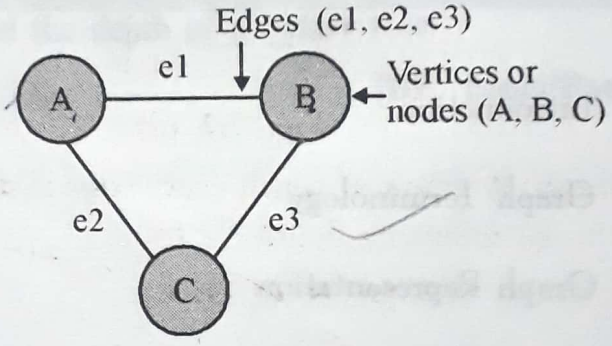


Fig. 7.1 A simple graph

Figure 7.1 shows a simple graph, for which

$$V(G) = \{A, B, C\}$$

and

$$E(G) = \{e1, e2, e3\}$$

Where V(G) is a set of vertices and E (G) is a set of edges. In figure 7.1, there are three vertices and three edges in the graph. You may notice the edge e1 is incident with node A and node B; so we could write as

$$e1 = (A, B) \text{ or } (B, A)$$

and similarly,

$$e2 = (A, C)$$

$$e3 = (B, C)$$

then

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

7.2 Graph Terminology

1. **Directed Edge:** An edge that is directed from one vertex to another is called a directed edge. It is shown in figure 7.2 (i)

Here $V = \{A, B\}$

and $e = \{A, B\}$

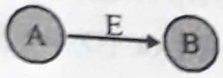


Fig. 7.2 (i)

2. **Undirected Edge:** An edge which has no specific direction is called an undirected edge.

Here $V = \{A, B\}$

and $e = \{A, B\}$ or $\{B, A\}$

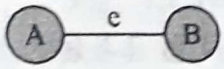


Fig. 7.2 (ii)

3. **Directed Graph:** A graph in which every edge is directed is called a directed graph or digraph. In the following figure 7.2 (iii),

$V = \{A, B, C, D\}$

$E = \{e_1, e_2, e_3, e_4, e_5\}$

Here $e_1 = \{A, B\}$

$e_2 = \{C, A\}$

$e_3 = \{C, D\}$

$e_4 = \{D, B\}$

$e_5 = \{A, D\}$

$e_1 = (A, B)$

$e_1 = (B, A)$

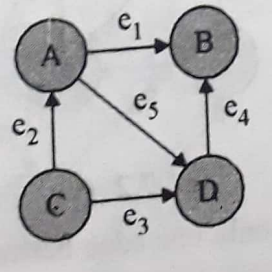


Fig. 7.2(iii)

4. **Undirected Graph:** A graph in which every edge is undirected is called undirected graph. In the following figure 7.2 (iv)

e_1 = {A, B} or {B, A}
e_2 = {C, A} or {A, C}
e_3 = {C, D} or {D, C}
e_4 = {D, B} or {B, D}
e_5 = {A, D} or {D, A}

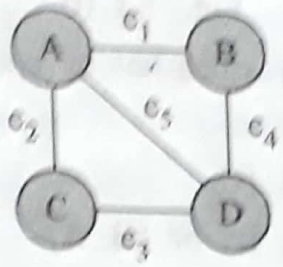


Fig. 7.2 (iv)

5. **Mixed Graph:** If some of the edges are directed and some are undirected in a graph, then the graph is called mixed graph.

6. **Multi Graph:** When two or more edges are used to join two vertices, such edges are called parallel edges and the graph is called multi graph.

Figure 7.2 (v) shows a multi graph. The edges e_1 and e_2 are called multiple edges, if they are connected from the same pair of vertices.

i.e. $e_1 = (B, A)$

and $e_2 = (A, B)$

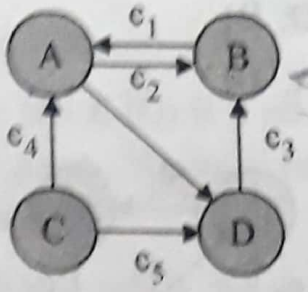


Fig. 7.2 (v)

7. **Simple Graph:** If there is only one edge between a pair of vertices, then such graph is called simple graph.

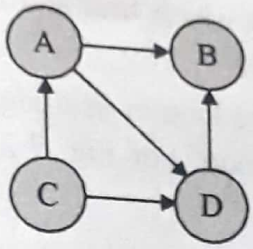


Fig. 7.2 (vi)

8. **Null Graph:** A graph without any edge is called Null graph.

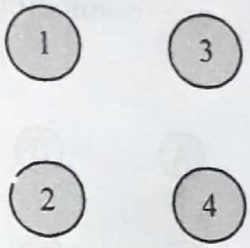


Fig. 7.2 (vii)

9. **Isolated Vertex:** A node which is not adjacent to any other node in the graph is called an isolated vertex.

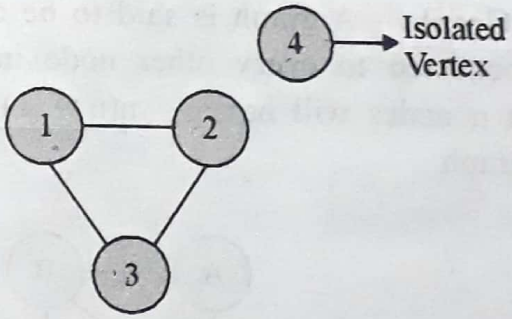


Fig. 7.2 (viii)

10. **Path:** A path is a sequence of consecutive edge from the source node to destination node.

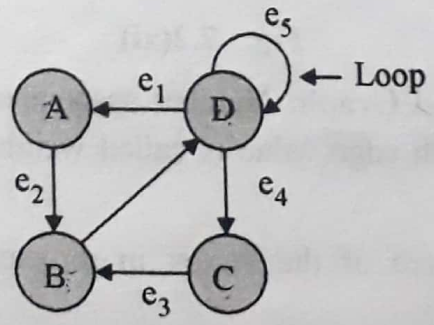


Fig. 7.2 (ix)

Consider the figure 7.2 (ix) above shown, path from node A to D is A, B and D.

11. **Cycle:** A cycle is a path in which first and last vertices are same. In the above Fig. 7.2 (ix) A-B-D-A is a cycle.

12. **Loop:** An edge will be called loop or self edge if it starts and ends on the same node. In Fig. 7.2 (ix), D-D is a loop. For Fig. 7.2 (ix), an edge is called loop, here edge $e_5 = (D, D)$

13. **Cyclic Graph:** If a graph contains loops or cycles it is called cyclic graph.

14. **Connected Graph:** An undirected graph is called connected, if there exist a path from any vertex to any other vertex, or any node is reachable from any node. Graph G_1 and G_3 are connected graphs while graph G_2 is not a connected graph.

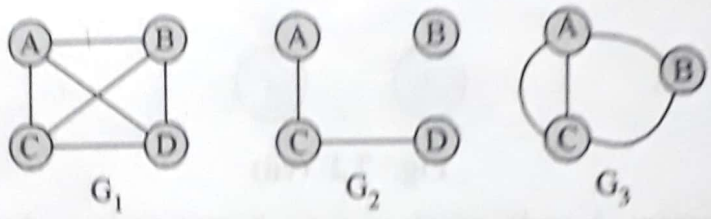


Fig. 7.2(x)

15. **Complete Graph:** – A graph is said to be complete or fully connected if every node is connected to every other node in a graph. An undirected complete graph with n nodes will have $n(n-1)/2$ edges. Figure 7.2(xi) shows a complete graph.

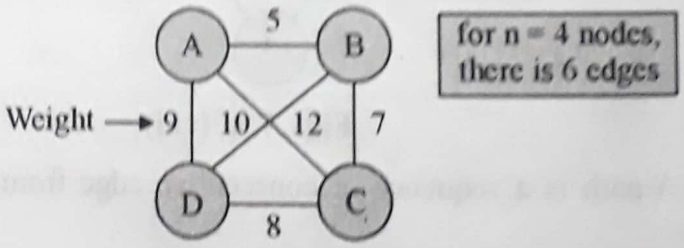


Fig. 7.2(xi)

16. **Network or Weighted Graph:** Number associated with an edge is called its weight or value. A graph with edge value is called weighted graph. Figure 7.2(xi) shows a weighted graph.

17. **Indegree:** The indegree of the vertex in directed graph is the number of edges ending at it.

18. **Outdegree:** The outdegree of the vertex in directed graph is the number of edges starting from it.

19. **Acyclic Graph:** A graph that has no cycles is called acyclic graph
20. **Dag:** A directed acyclic graph is named as dag after its acronym
21. **Maximum Edges in Graph** - In an undirected graph, there can be $n(n-1)/2$ maximum edges and in a directed graph, there can be $n(n-1)$ maximum edges, where n is the total number of nodes in the graph.
22. **Weakly Connected:** A directed graph is called weakly connected if for any pair of vertices u and v there is not a path from u to v or from v to u.
23. **Strongly Connected:** A directed graph is called strongly connected if there is a path from any vertex to any other vertex in a graph. Fig. 7.2 (xi) shows a strongly connected graph.

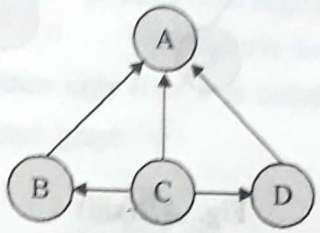


Fig. 7.2 (xii)

For Figure 7.2 (xii), Indegree and Outdegree of graph is given below in table

Node	Indegree	Outdegree
A	3	0
B	1	1
C	0	3
D	1	1

= 3
= 2
= 3
= 2

Note that sum of indegrees of all vertices and sum of outdegrees of all vertices is always equal in any graph.

24. **Total Degree of Vertex:** Total degree of the vertex is the sum of outdegree and indegree of the vertex or the number of edges incident on the vertex determine its degree.

Note:

(i) In case of an undirected graph, total degree of the vertex V is equal to the number of edges incident with V.

(ii) Total degree of the loop is 2.

(iii) For an isolated vertex, total degree is 0.

25. **Tree:** A graph is a tree if it has two properties :

(i) It is connected, and

(ii) There are no cycles in the graph.

The following graphs are tree :

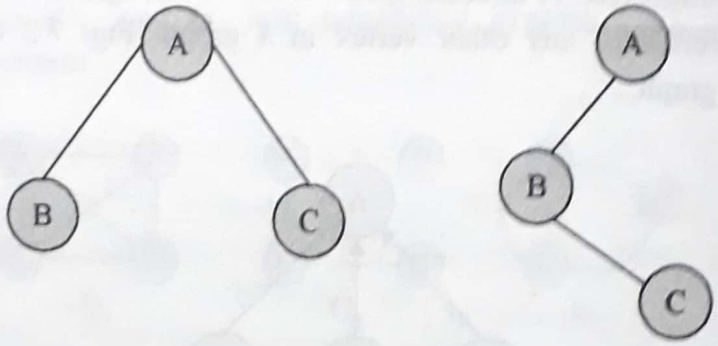


Fig. 7.2(xiii)

26. **Source:** A node, which has no incoming edges but has outgoing edges, is called source. The indegree of source is zero. In graph 7.2 (xii), node C is a source.

27. **Sink:** A node which has no outgoing edges but has incoming edges, is called a sink. The outdegree of sink is zero. In graph 7.2 (xiii) node A is a sink.

28. **Pendant Node:** A node is said to be pendant if its indegree is equal to 1 and out degree is equal to 0.

29. **Diameter of Graph** - It is defined as length of longest path in the graph.

7.3 Graph Representation

We have mainly two components in graph, nodes and edges. Now we have to design the data structure to keep these component in mind. There are two popular ways that are used to represent a graph in computer's memory.

1. Sequential Representation
2. Linked Representation

7.3.1 Sequential Representations

In sequential representation, graph can be represented as matrices (or 2 dimensional arrays). An adjacency matrix is the most common matrix to represent a graph.

Adjacency Matrix

Adjacency matrix is the matrix, which keeps the information of adjacent vertices. Suppose there are 5 vertices in graph, then row 1 represents the vertex 1, row 2 represents the vertex 2 and so on. Similarly column 1 represents vertex 1, column 2 represents vertex 2 and so on. Hence, it is a sequence matrix having one row and one column for each vertex. The element of the adjacency matrix are either 1 or 0. A value of 1 for row i and column j implies that an edge (e_{ij}) exist between V_i and V_j. A value of 0 means there is no edge between vertex V_i and V_j.

The adjacency matrix A for a graph G = (V, E) with n vertices is an n * n matrix such that,

$$A_{ij} = \begin{cases} 1 & \text{if there is an edge between } V_i \text{ and} V_j \\ 0 & \text{if there is no such edge.} \end{cases}$$

Note: A matrix which contains only 0 or 1 is called a bit matrix or a Boolean matrix.

Figure 7.3(i) shows a directed graph

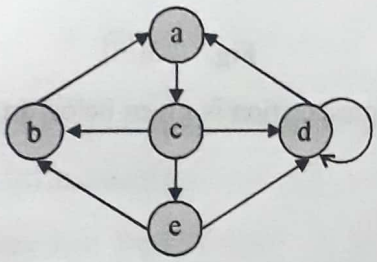


Fig. 7.3(i)

Adjacency matrix representation is given below in figure 7.3(ii) for the directed graph shown in figure 7.3(i):

	a	b	c	d	e
a	0	0	1	0	0
b	1	0	0	0	0
c	0	1	0	1	1
d	1	0	0	1	0
e	0	1	0	1	0

Fig. 7.3(ii) Adjacency Matrix Representation of Directed Graph

Note:

1. From the adjacency matrix, it is clear that number of 1's is equal to the number of edges in the graph.
2. The number in each row shows the outdegree of vertex.
3. If graph is undirected, the adjacency matrix will be symmetric.

Let graph shown in figure 7.3(iii) is undirected.

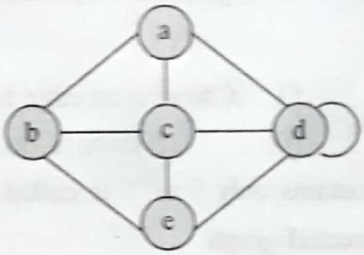


Fig. 7.3(iii)

Adjacency matrix representation is given below in figure 7.3(iv) for the graph shown in figure 7.3(iii):

	a	b	c	d	e
a	0	1	1	1	0
b	1	0	1	0	1
c	1	1	0	1	1
d	1	0	1	1	1
e	0	1	1	1	0

Fig. 7.3(iv) Adjacency Matrix Representation of undirected graph

4. For a NULL graph, adjacency matrix has all of its elements zero.
5. If there are loops at each vertex but no edges in the graph as shown in figure 7.3(v), than the adjacency matrix will be unit matrix. It is shown in figure 7.3(vi).

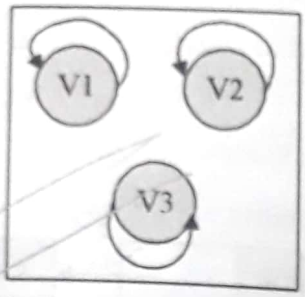


Fig. 7.3(v)

$$A = \begin{array}{c} \\ V1 \\ V2 \\ V3 \end{array} \begin{array}{c} \begin{array}{ccc} V1 & V2 & V3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

Fig. 7.3 (vi)

6. For a weighted graph, the elements of adjacency matrix are the weight on that edge. For figure 7.2 (xi), the adjacency matrix will be as:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 5 & 0 & 0 \\ 0 & 0 & 7 & 0 \\ 12 & 0 & 0 & 8 \\ 9 & 10 & 0 & 0 \end{bmatrix} \end{array}$$

Fig. 7.3(vii)

7.3.2 Linked Representation

It is also called Adjacency List Representation. If the adjacency matrix of the graph is sparse then it is more efficient to represent the graph through adjacency list. For adjacency list representation, we maintain a list for each vertex and then for each vertex, we have a linked list of its adjacent vertices.

Adjacency list for figure 7.3(i) is given below:

NODE	ADJACENCY LIST
a	c
b	a
c	b, d, e,
d	a, d
e	b, d

only outdegree one & loop also

Fig. 7.3(viii) Adjacency list

Adjacency list representation of graph for figure 7.3(i) is shown below:

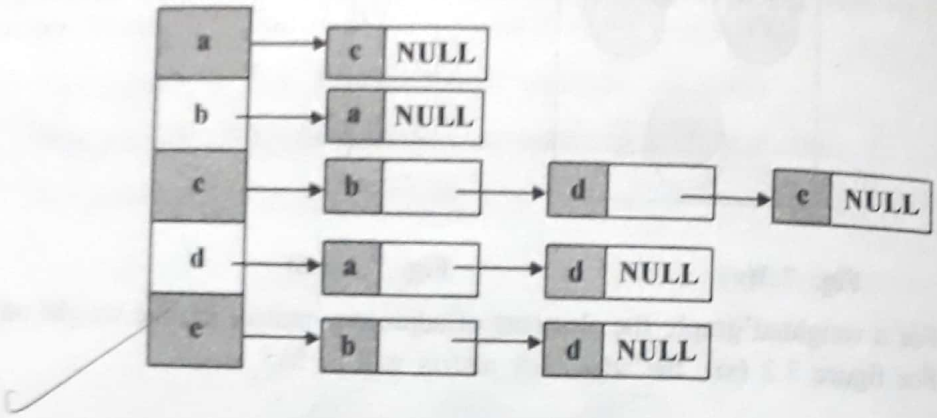


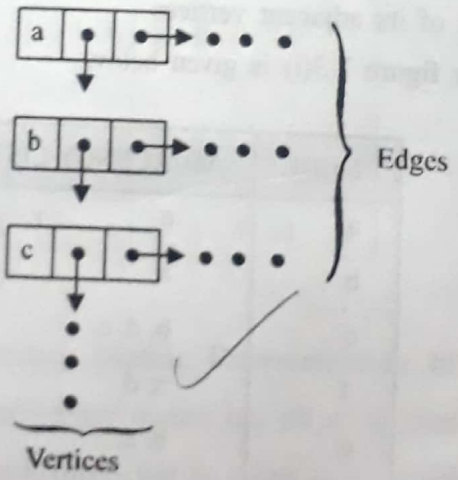
Fig. 7.3(ix) Adjacency List Representation of graph for figure 7.3(i)

Now, in figure 7.3(viii) we have represented graph as linked list in which left most part represents vertices of graph where remaining nodes represents edges.

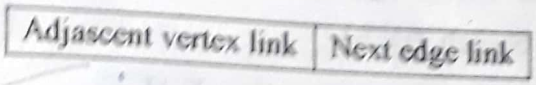
In actual, vertices are represented by node with 3 parts, one data and 2 link parts. The data part represents vertex, one link part represents next vertex and other linkpart represents edges of graph. This node can be represented as :

<Vertex-Name>	Next vertex link	Edge link

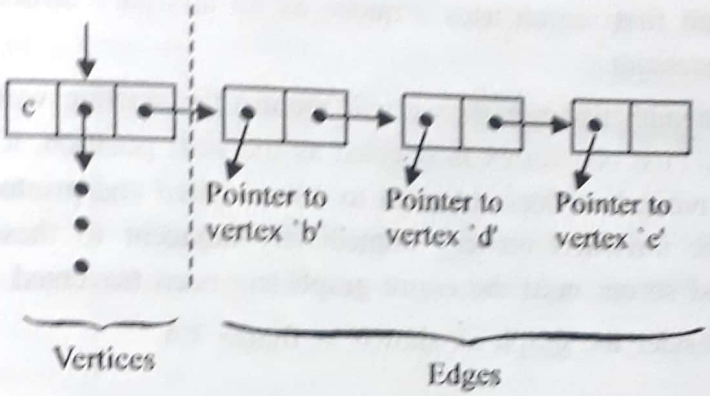
e.g. :



Also, there is one more type of node, which is used to represent edge of graph. This node has 2 linkparts (no data part), one linkpart points to vertex, to which it is connected and other linkpart represent whether more edge is present or not. This is the linked list which does not contain datapart while it has 2 linked parts, and this node can be represented as



e.g.



Now, graph can be completely represented as:

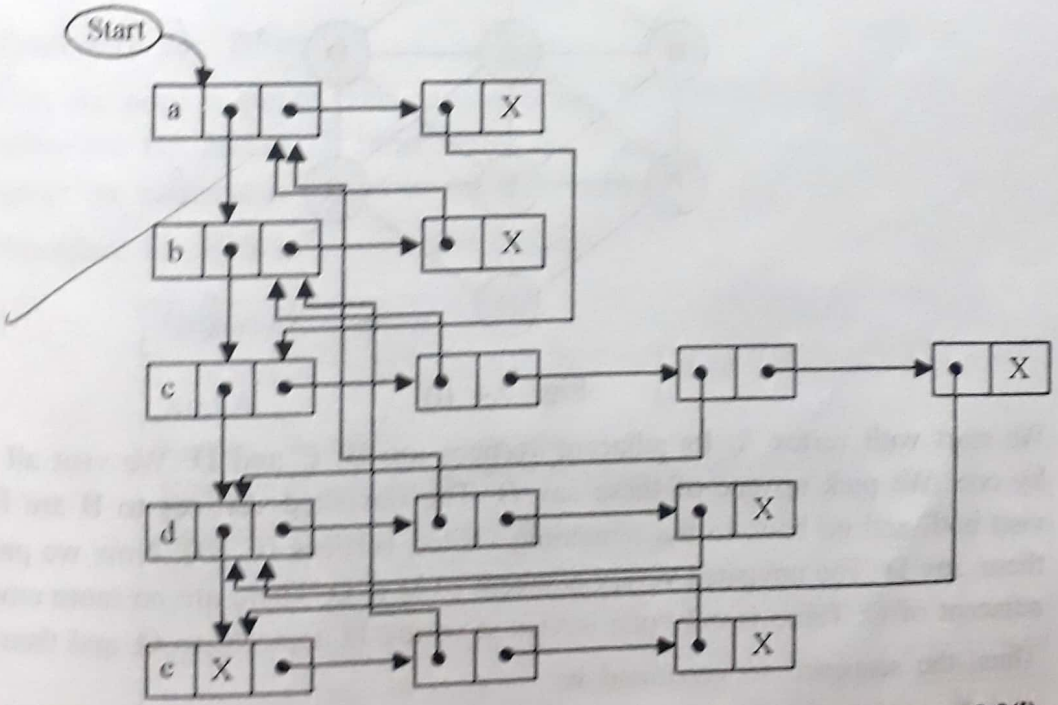


Fig. 7.3 (x) Linked list representation of graph for figure 7.3(i)

7.4 Traversing a Graph

A graph traversal means visiting all the nodes of the graph. There are two standard ways for traversing a graph.

1. Breadth First Search (BFS)
2. Depth First Search (DFS)

7.4.1 Breadth First Search

The breadth first search uses a queue as an auxiliary structure to store the nodes for future processing.

In a breadth first search, we will require the starting vertex from which this search will begin. First one vertex is selected as the start position, it is visited and printed, and then all unvisited vertices, adjacent to it are visited and printed in some sequential order. Finally, the unvisited vertices immediately adjacent to these vertices are visited and printed and so on, until the entire graph has been traversed.

Let us consider the graph as shown in figure 7.4

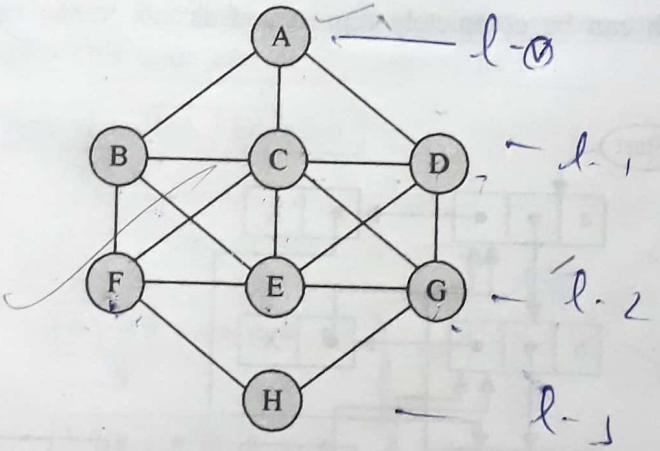


Fig. 7.4 (i)

We start with vertex A. Its adjacent vertices are B, C and D .We visit all vertices one by one .We pick up one of these say B. The unvisited vertices to B are F and E. We visit both and go back to the remaining visited vertices (C, D). Now we pick up one of these, say D .The unvisited vertex adjacent to D is G. There are no more unvisited vertex adjacent of C. There is only one unvisited vertex H adjacent to G, and then it is visited.

Thus, the sequence so generated is:

A B C D F E G H

Traversing through BFS, for figure 7.4(i) it is stepwise shown in figure 7.4 (ii)

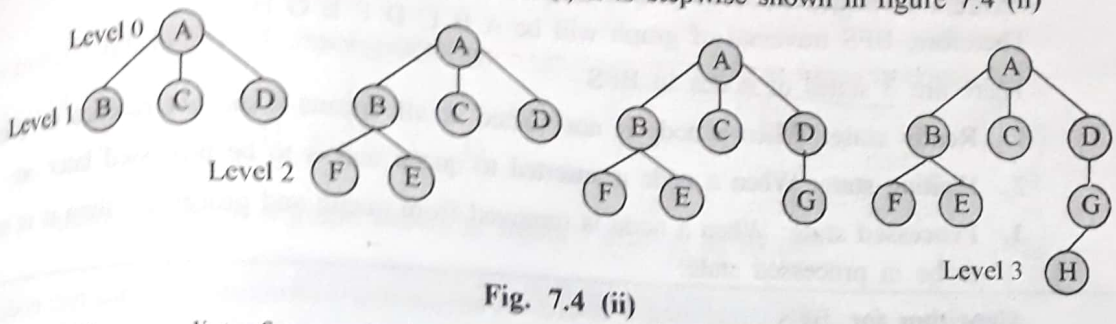


Fig. 7.4 (ii)

Adjacency list of a graph shown in figure 7.4(i) is as follows:

Node	Adjacency List
A	B, C, D
B	A, F, E, C
C	A, B, D, E
D	A, C, E, G
E	B, C, D, F, G
F	B, E, H
G	E, D, H
H	F, G

Procedure for BFS:

Here we need a queue. The vertices which are either processed or already available in queue are not added, all other nodes are added to the queue. We already know that in queue, an element is added at rear and deleted from the front.

Procedure for BFS is given below in table.

Queue Operations	Position of Queue	Node Traversal
Add A	A	
Remove A, add its neighbors at rear	B, C, D	A
Remove B, add its neighbors at rear	C, D, F, E	B
Remove C, add its neighbors at rear	D, F, E	C
Remove D, add its neighbors at rear	F, E, G	D
Remove F, add its neighbors at rear	E, G, H	F
Remove E, add its neighbors at rear	G, H	E
Remove G, add its neighbors at rear	H	G
Remove H, add its neighbors at rear	NULL	H

At last, queue will be empty.

Therefore, BFS traversal of graph will be A B C D F E G H.

There are 3 states of nodes in BFS

1. **Ready state:** When a node is not visited at all, means it did not reaches to queue
2. **Waiting state:** When a node is inserted to queue and is to be processed later on.
3. **Processed state:** When a node is removed from queue and processed, than it is said to be in processed state.

Algorithm for BFS

This Algorithm executes a breath first search on graph G beginning at a starting node A.

Step1. Begin

Step2. Initializing all nodes to the ready state (STATUS = 1).

Step3. Put the starting node A in QUEUE and change its status to the waiting state (STATUS = 2).

Step4. Repeat step 5 and 6 until QUEUE is empty

Step5. Remove the front node N of QUEUE .Process N and change the status of N to the processed state (STATUS = 3).

Step6. Add all the neighbours of N to the rear of QUEUE that are in ready state (STATUS =1), and change their status to the waiting state (STATUS =2).

Step7. Exit.

7.4.2 Depth First Search

We have seen in BFS, the traversal of the graph proceeds level by level. But depth first search traversal follows first a path from a starting node to an ending node. Then another path from the start to the end, and so on until all nodes have been visited.

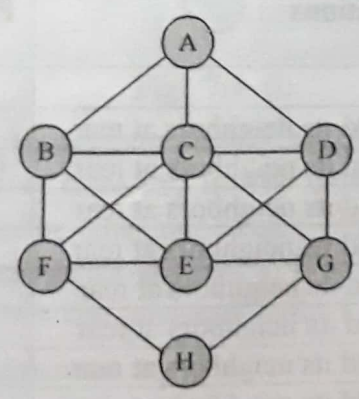


Fig. 7.4(iii)

Depth first traversal of the graph for figure 7.4(iii) is in the following sequence:

A D G H F { E C B
or
A B F H G { E C D

Adjacency list of a graph shown in figure 7.4(iii) is as follows:

Node	Adjacency List
A	B, C, D
B	A, F, E, C
C	A, B, D, E
D	A, C, E, G
E	B, C, D, F, G
F	B, E, H
G	E, D, H
H	F, G

Procedure for DFS

Here we need a stack instead of queue. The nodes which are either traversed or already available in the stack are not pushed. We already know that an element is pushed or popped at the top of the stack.

Procedure for DFS is given below in table.

Stack Operations	Position of Stack	Node Traversal
Push A	A	
Pop A & push its neighbors at top	B, C, D	A
Pop D & push its neighbors at top	B, C, E, G	D
Pop G & push its neighbors at top	B, C, E, H	G
Pop H & push its neighbors at top	B, C, E, F	H
Pop F & push its neighbors at top	B, C, E	F
Pop E & push its neighbors at top	B, C	E
Pop C & push its neighbors at to	B	C
Pop B & push its neighbors at top	NULL	B

At last stack will be empty.

Hence, DFS traversal of the graph will be A D G H F E C B

Note: It is clear from the DFS traversal that depth first search starts from vertex A and visits all vertices reachable from A.

Like BFS, there are also 3 states of node.

(i) **Ready state:** When a node is not pushed to stack

(ii) **Waiting state:** When a node is pushed to stack, but not popped

(iii) **Processed state:** When a node is popped from stack.

Algorithm for DFS

This Algorithm executes a depth first search on graph G beginning at a starting node A.

Step1. Begin

Step2. Initializing all nodes to the ready state (STATUS = 1).

Step3. Push the starting node A in STACK and change its status to the waiting state (STATUS = 2).

Step4. Repeat step 5 and 6 until STACK is empty.

Step5. Remove the top node N of STACK. Process N and change the status of N to the processed state (STATUS = 3).

Step6. Add to the top of STACK all the neighbors of N that are in steady state (STATUS =1), and change their status to the waiting state (STATUS = 2).

[End of step 3 loop]

Step7. Exit.

7.5 Path Matrix or Reachability Matrix

A matrix that tells us whether or not there are paths between the nodes, is called path Matrix. Let us take a graph G with n nodes $v_1, v_2, \ldots\ldots\ldots v_n$. The path matrix or reachable matrix of G can be defined as-

$$P[i][j] = \begin{cases} 1 & \text{if there is a path in between } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

Note: A graph G will be strongly connected if there are no zero entries in path matrix means a path exists between all v_i to v_j and v_j to v_i also

Computing Path matrix from powers of adjacency matrix -

Let us take a graph and compute path matrix for it from its adjacency matrix.

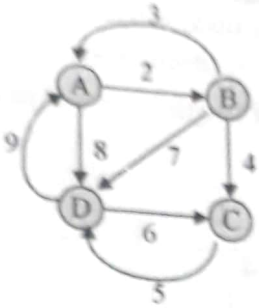


Fig. 7.5

The corresponding weighted adjacency matrix will be as-

Weighted adjacency Matrix

$$W = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 2 & 0 & 8 \\ 3 & 0 & 4 & 7 \\ 0 & 0 & 0 & 5 \\ 9 & 0 & 6 & 0 \end{bmatrix} \end{array}$$

The adjacency matrix will be obtained by replacing non zero values by 1.

Adjacency Matrix $A = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array} = AM_1$

Path length denotes the number of edges in the path. Adjacency matrix is the path matrix of path length 1. Now if we multiply the adjacency matrix with itself then we get the path matrix of length 2.

$$AM_2 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 2 & 0 & 2 & 1 \\ 1 & 1 & 1 & 2 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 2 \end{bmatrix} \end{array}$$

(Where $AM_2 = AM_1 \times AM_1$)

In this matrix, value of AM_2 [i][j] will represent the number of paths of path length 2 from node v_i to v_j. For example, here node A has 2 paths of path length 2 to node C, and node D has 2 paths of path length 2 to itself, node C has 1 path of path length 2 to node A.

To obtain the path matrix of length 3 we will multiply the path matrix of length 2 with adjacency matrix.

$$AM_3 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 1 & 2 & 1 & 4 \\ 3 & 1 & 3 & 3 \\ 0 & 1 & 0 & 2 \\ 3 & 0 & 3 & 1 \end{bmatrix} \end{array} \qquad (\text{Where } AM_3 = AM_2 \times AM_1)$$

Here AM_3 [i][j] will represent the number of paths of path length 3 from node v_i to node v_j. For example, here node A has 4 paths of path length 3 to node D, and node D has no path of path length 3 to node B.

Similarly, we can find out the path matrix for path length 4.

$$AM_4 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 6 & 1 & 6 & 4 \\ 4 & 3 & 4 & 7 \\ 3 & 0 & 3 & 1 \\ 1 & 3 & 1 & 6 \end{bmatrix} \end{array} \qquad (\text{Where } AM_4 = AM_3 \times AM_1)$$

Let us take a matrix X where

$$X = AM_1 + AM_2 + AM_3 + AM_4$$

Here X[i][j] has the value of number of paths, less than or equal to length n, from node v_i to v_j. Here n is the total number of nodes in the graph.

For the above graph the value of X_4 will be as-

$$X = \begin{bmatrix} 9 & 4 & 9 & 10 \\ 9 & 5 & 9 & 13 \\ 4 & 1 & 4 & 4 \\ 5 & 4 & 5 & 9 \end{bmatrix}$$

From definition of path matrix we know that P[i][j] = 1 if there is a path from v_i to v_j and this path can have length n or less than n.

Now in this matrix, if we replace all nonzero entries by 1 then we will get the path matrix or reachability matrix.

$$P = \begin{matrix} A \\ B \\ C \\ D \end{matrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

This graph is strongly connected since all the entries are equal to 1.

7.6 Shortest Path Problem

Graph may be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges may then be assigned weights which may be distance between the two cities connected by the edge or the average time to drive along that section of highway.

To find shortest path, we check,

1. Is there a path from A to B ?
2. If there is more than one path from A to B, which is the shortest path?

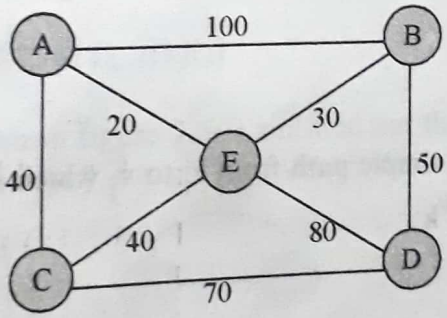


Fig. 7.6

For example, the graph shown in figure 7.6 if we go from A to D through B, we have to travel a distance of 150 KM and if we go from A to D through E or C, we have to travel a distance of 100 KM or 110 KM respectively. Hence A E D is the shortest path.

Thus we can say that a shortest path from source node to destination node is a path for which the sum of weights of the edges on the path should be minimum.

7.6.1 Path Matrix by Warshall's Algorithms

Warshall's Algorithm

As we have seen earlier, we can find the path matrix P of a given graph G with the use of powers of adjacency matrix. But this method is not efficient one. Warshall has given one efficient technique for finding path matrix of a graph which is called Warshall's algorithm.

Let us take a graph G of n vertices $v_1, v_2, v_3 \ldots\ldots\ldots v_n$. First we will take Boolean matrices $P_0, P_1, \ldots\ldots P_n$ where P_k [ij] is defined as-

$$P_k[i][j] = \begin{cases} 1 & \text{If there is a simple path from vertices } v_i \text{ to } v_j \text{ which does not use any other node except possibly } v_1, v_2, v_3 \ldots\ldots v_n \text{ or this path does not use any node numbered higher than k} \\ 0, & \text{Otherwise} \end{cases}$$

Or we can say

$P_0[i][j] = 1$ If there is a simple path from v_i to v_j which does not use any node.

$P_1[i][j] = 1$ If there is a simple path from v_i to v_j which does not use any other nodes except possibly v_1

$P_2[i][j] = 1$ If there is a simple path from v_i to v_j which does not use any other nodes except v_1, v_2.

.....

.....

$P_k[i][j] = 1$ If there is a simple path from v_i to v_j which does not use any other nodes except $v_1, v_2 \ldots\ldots v_k$.

.....

.....

$P_n[i][j] = 1$ If there is a simple path from v_i to v_j which does not use any other nodes except $v_1, v_2 \ldots\ldots v_n$.

Here P_0 represents the adjacency matrix and P_n represents the path matrix.

$P_0[i][j] = 1$, If there is a simple path from v_i to v_j which does not use any other node. The only way to go directly from v_i to v_j without using any node is to go directly from v_i to v_j. Hence $P_0[i][j] = 1$ if there is any edge from v_i to v_j So P_0 will be the adjacency matrix.

$P_n[i][j] = 1$, If there is a simple path from v_i to v_j which does not use any nodes except $v_1, v_2, \ldots v_n$. There are total n nodes means this path can use all n nodes, hence from the definition of path matrix we observe that P_n is the path matrix.

Now we'll see how we can find out the elements of matrix P_k. We know that P_0 is equal to the adjacency matrix. We have to find matrices $P_1, P_2, \ldots P_n$. First, we have a need to know how to find $P_k[i][j]$ from $P_{k-1}[i][j]$ then we can find all these matrices. So first we will find $P_k[i][j]$ from $P_{k-1}[i][j]$.

We can conclude that if $P_{k-1}[i][j] = 0$ then $P_k[i][j]$ can be equal to 1 only if $P_{k-1}[i][k] = 1$ and $P_{k-1}[k][j] = 1$.

So we have two cases where $P_k[i, j] = 1$

1. There is a simple path from v_i to v_j which does not use any other nodes except possibly $v_1, v_2, \ldots v_{k-1}$, hence

   $P_{k-1}[i][j] = 1$ $\qquad v_i \ldots\ldots\ldots> \ldots\ldots\ldots v_j$

2. There is a simple path from v_i to v_k and a simple path from v_k to v_j where each path does not use any other nodes except $v_1, v_2, \ldots v_{k-1}$, hence

   $P_{k-1}[i][k] = 1$ and $P_{k-1}[k][j] = 1$ $\qquad v_i \ldots\ldots> \ldots\ldots v_k \ldots\ldots> \ldots\ldots v_j$

So the element of path matrix P_k can be defined as-

$$P_k[i][j] \begin{cases} P_{k-1}[i][j] \\ \text{or} \\ P_{k-1}[i][k] \text{ and } P_{k-1}[k][j] \end{cases}$$

Let us take a graph shown in figure 7.6(i) and find out the values of P_0, P_1, P_2, P_3, P_4.

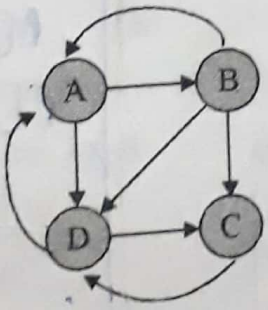


Fig. 7.6(i)

The first matrix P_0 is the adjacency matrix.

$$P_0 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Now we have to find P_1

Now wherever $P_0[i][j] = 1$ $\quad$ $P_1[i][j] = 1$

If $P_0[i][j] = 0$ then see $P_0[i][1]$ and $P_0[1][j]$, if both are 1 then $P_1[i][j] = 1$

$$P_1 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{array} \end{array}$$

Similarly,

$$P_2 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Similarly, $P3 =$

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

$P_0[1][1] = 0$ $P_0[1][2]$

$P_0[2][1] = 1$ $P_0[1][4]$

$P_0[1][2]$ $P_0[4][1] = 1$

$P[1,1] = P[1,1]$ or $(P[1,1]$ & $P[$

$P[1,2] = P[1,2]$ or

$P[1,3] = P[1,3]$ or $(P[1,1]$ & $P[$

$P[1,4] = P[1,4]$

$P[2,1] =$

$P[2,2] = P[2,2]$ or $(P[2,1]$ & $P[$

$P[2,3] =$

$P[2,3] =$

Similarly,

$$P4 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Here P_0 is the adjacency matrix and P_4 is the path matrix of the graph.

Warshall's Algorithm:

A directed graph G with M nodes is maintained in memory by its adjacency matrix A. This algorithm finds the (Boolean) path matrix P of the graph G.

Step1. Begin

Step2. Repeat for I, J = 1, 2,……, M: [Initializes path matrix P.]

If A [I, J] = 0, then:

Set P[I, J]: = 0

Else:

Set P[I, J]: = 1

[End of If]

[End of loop.]

// Evaluate the path matrix

Step3. Repeat step 4 and 5 for K = 1, 2,……., M: [Updates P.]

Step4. Repeat step 5 for I = 1, 2,……., M:

Step5. Repeat for J = 1, 2,……, M:

Set P [I, J]: = P [I, J] or (P [I, K] AND P [K, J])

[End of step 5 for loop]

[End of step 4 for loop]

[End of step 3 for loop]

Step6. Exit

Warshall's Modified Algorithm

We have seen that Warshall's algorithm gives the path matrix of graph. Now by modifying this algorithm, we will find out the shortest path matrix Q. Here Q[i][j] will represent the length of shortest path from v_i to v_j.

There can be many paths from any node v_i to v_j. Our purpose is to find the length of the shortest path in between these nodes.

We have seen that weighted matrix can be defined as-

$$W[i][j] = \begin{cases} \text{weight on edge} & \text{, if there is an edge from node i to node j} \\ 0 & \text{, otherwise} \end{cases}$$

Now we will take weight on each edge as the length of that edge.

So we have two cases where $P_k[\,i\,][\,j\,] = 1$

1. There is a simple path from v_i to v_j which does not use any other nodes except possibly $v_1, v_2 \ldots\ldots v_{k-1}$, hence

   $P_{k-1}[i][j] = 1$ $\qquad v_i \ldots\ldots\ldots\ldots > \ldots\ldots\ldots\ldots v_j$

2. There is a simply path from v_i to v_k and a simple path from v_k to v_j where each path does not use any other nodes except $v_1, v_2 \ldots\ldots v_{k-1}$ hence

   $P_{k-1}[i][k] = 1$ and $P_{k-1}[k][j] = 1$ $\qquad v_i \ldots\ldots\ldots\ldots > \ldots\ldots\ldots\ldots v_j$

   So the element of path matrix P_k can be defined as-

$$P_k[i][j] = \begin{cases} P_{k-1}[i][j] \\ \text{or} \\ P_{k-1}[i][k] \text{ And } P_{k-1}[k][j] \end{cases}$$

Let us take a graph and shown in figure 7.6(ii) and find out the values of P_0, P_1, P_2, P_4

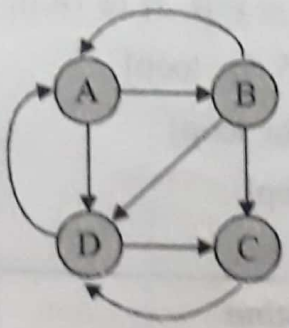


Fig. 7.6(ii)

The first matrix P_0 is the adjancy matrix.

$$P_0 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Now we have to find P_1

Now where $P_0\ [i]\ [j] = 1$ $\qquad P_1\ [i]\ [j] = 1$

If $P_0\ [i]\ [j] = 0$ then see $P_0\ [i]\ [1]$ and $P_0\ [1]\ [j]$, if both are 1 then $P_1\ [i]\ [j] = 1$

$$P_1 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

As in Warshall's algorithm matrices were $P_0, P_1, P_2 \ldots$, here we will take matrices as $Q_0, Q_1, Q_2 \ldots$

Length of shortest path from v_i to v_j using nodes $v_1, v_2, v_3, \ldots v_k$.

$$Q_k\ [i][j] = \begin{cases} \text{length of shortest path from } V_i \text{ to } v_j \text{ using nodes } v_1, v_2, v_3 \ldots\ldots v_k \\ \infty \left(\text{if there is no path from } v_i \text{ to } v_j \text{ using nodes } v_1, v_2, v_3 \ldots v_k\right) \end{cases}$$

Hence we can say that

$Q_0[i]\ [j]$ = length of an edge from v_i to v_j

$Q_1[i]\ [j]$ = length of shortest path from v_i to v_j using v_1.

$Q_2[i]\ [j]$ = length of shortest path from v_i to v_j using v_1, v_2.

$Q_3[i]\ [j]$ = length of shortest path from v_i to v_j using v_1, v_2, v_3

We can find out Q_0 from the weighted adjacency matrix by replacing all zero entry by ∞. If there are n nodes in graph then matrix Q_n will represent the shortest path matrix. So now our purpose is to find out matrices $Q_1, Q_2, Q_3 \ldots\ldots Q_n$. We have already found out matrix Q_0 by weighted adjacency matrix. Now, if we know how to find out the matrix Q_k from matrix Q_{k-1} then we can easily find out matrices $Q_1, Q_2, Q_3 \ldots\ldots Q_n$ also. Now we'll see how to find out matrix Q_k from matrix Q_{k-1}.

We have seen in Warshall's algorithm that $P_k[i][j] = 1$ if any of these two condition is true.

1. $P_{k-1}[i][j] = 1$ *or*

2. $P_{k-1}[i][k] = 1$ *and* $P_{k-1}[k][j] = 1$

This means there is a path from v_i to v_j using $v_1, v_2, v_3 \ldots\ldots v_k$ in two conditions

1. There is path from v_i to v_j using $v_1, v_2, \ldots\ldots v_{k-1}$ (path P_1)

Or

2. There is path from v_i to v_k using $v_1, v_2, v_3 \ldots\ldots v_{k-1}$ and there is a path from v_k to v_j using $v_1, v_2, v_3 \ldots\ldots v_{k-1}$ (path P_2)

Here we are dealing with path lengths so lengths of paths will be as -

Length of first path will be $Q_{k-1}[i][j]$

Length of second path will be $Q_{k-1}[i][k] + Q_{k-1}[k][j]$

Now we'll select the smaller one from these two paths lengths.

So value of $Q_k[i][j]$ = Minimum ($Q_{k-1}[i][j]$, $Q_{k-1}[i][k] + Q_{k-1}[k][j]$)

Now let us take a graph and find out the shortest path matrix for it.

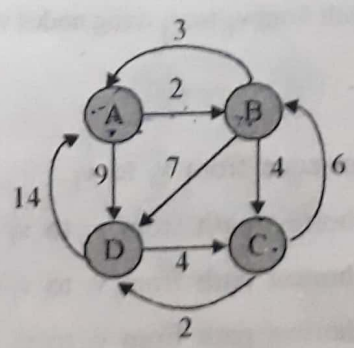


Fig. 7.6(iii)

Weighted adjacency matrix for this graph is

$$W = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 2 & 0 & 9 \\ 3 & 0 & 4 & 7 \\ 0 & 6 & 0 & 2 \\ 14 & 0 & 4 & 0 \end{bmatrix} \end{array}$$

Now, replace all 0's with ∞ in weight adjacency matrix and we get

$$Q_0 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & 2 & \infty & 9 \\ 3 & \infty & 4 & 7 \\ \infty & 6 & \infty & 2 \\ 14 & \infty & 4 & \infty \end{bmatrix} \end{array} \qquad \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} -- & AB & -- & AD \\ BA & -- & BC & BD \\ -- & CB & -- & CD \\ DA & -- & DC & -- \end{bmatrix} \end{array}$$

After including node A(k = 1)

$$Q_1 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & 2 & \infty & 9 \\ 3 & 5 & 4 & 7 \\ \infty & 6 & \infty & 2 \\ 14 & 16 & 4 & 23 \end{bmatrix} \end{array} \qquad \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} -- & AB & -- & AD \\ BA & BAB & BC & BD \\ -- & CB & -- & CD \\ DA & DAB & DC & DAD \end{bmatrix} \end{array}$$

After including node B (k = 2)

$$Q_2 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 5 & 2 & 6 & 9 \\ 3 & 5 & 4 & 7 \\ 9 & 6 & 10 & 2 \\ 14 & 16 & 4 & 23 \end{bmatrix} \end{array} \qquad \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} ABA & AB & ABC & AD \\ BA & BAB & BC & BD \\ CBA & CB & CBC & CD \\ DA & DAB & DC & DAD \end{bmatrix} \end{array}$$

After including node C (k = 3)

$$Q_3 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 5 & 2 & 6 & \mathbf{8} \\ 3 & 5 & 4 & \mathbf{6} \\ 9 & 6 & 10 & 2 \\ \mathbf{13} & \mathbf{10} & 4 & \mathbf{6} \end{bmatrix} \end{array} \qquad \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} ABA & AB & ABC & \mathbf{ABCD} \\ BA & BAB & BC & \mathbf{BCD} \\ CBA & CB & CBC & CD \\ \mathbf{DCBA} & \mathbf{DCB} & DC & \mathbf{DCD} \end{bmatrix} \end{array}$$

After including node D (k = 4)

$$Q_4 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 5 & 2 & 6 & 8 \\ 3 & 5 & 4 & 6 \\ 9 & 6 & \mathbf{6} & 2 \\ 13 & 10 & 4 & 6 \end{bmatrix} \end{array} \qquad \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} ABA & AB & ABC & ABCD \\ BA & BAB & BC & BCD \\ CBA & CB & \mathbf{CDC} & CD \\ DCBA & DCB & DC & DCD \end{bmatrix} \end{array}$$

$Q_0(1,3) = \text{Minimum}\left[Q_0(1,3),\ Q_0(1,1) + Q_0(1,3)\right] = \text{Minimum}(\infty, \infty) = \infty$

$Q_1(2,2) = \text{Minimum}\left[Q_0(2,2),\ Q_0(2,1) + Q_0(2,2)\right] = \text{Minimum}(\infty, 3+2) = 5$

$Q_2(3,1) = \text{Minimum}\left[Q_1(3,1),\ Q_1(3,2) + Q_1(2,1)\right] = \text{Minimum}(\infty, 6+3) = 9$

$Q_3(1,4) = \text{Minimum}\left[Q_2(1,4),\ Q_2(1,3) + Q_2(3,4)\right] = \text{Minimum}(\infty, 6+2) = 8$

$Q_4(3,3) = \text{Minimum}\left[Q_3(3,3),\ Q_3(3,4) + Q_3(4,3)\right] = \text{Minimum}(10, 2+4) = 6$

Warshall's Modified Algorithm(or Shortest Path Algorithm)

A weighted graph G with M nodes is maintained in memory by its weight matrix W. This algorithm finds a matrix Q such that Q [I, J] is the length of the shortest path from node V_i to node V_j. INFINITY is a very large number, and MIN is the minimum value function.

Step1. Begin

Step2. Repeat for I, J = 1, 2,……,M: [Initializes Q.]

If W [I, J] = 0 then:

Set Q [I, J]: = INFINITY (or any larger number say 999)

Else

Set Q [I, J]: = W [I, J]

[End of If]

[End of loop]

```
Step3. Repeat steps 4 and 5 for K = 1, 2,....., M :[Updates Q.]
Step4. Repeat step 5 for I = 1, 2,....., M:
Step5.      Repeat for J = 1, 2,....., M:
               Set Q [I, J] := MIN (Q [I, J], Q [I, K] + Q [K, J])
            [End of step 5 loop]
         [End of step 4 loop]
      [End of step 3 loop]
Step6. Exit.
```

7.7 Spanning Tree

A spanning tree of a graph is an undirected tree consisting of only those edges necessary to connect all the nodes in the original graph. A spanning tree has the properties that for any pair of nodes, there exist only one path between them and the insertion of any edge to a spanning tree form a unique cycle. Spanning tree finds application in obtaining an independent set of circuit equations for an electric network.

A spanning tree of a connected graph G contains all the nodes and has the edges, which connects all the nodes. So number of edges will be 1 less than the number of nodes. Let us take a graph G, shown in figure 7.7(i)

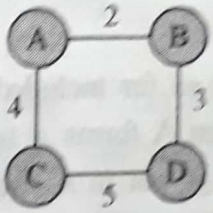


Fig. 7.7(i)

We know that a graph is a tree if there are no cycles in the graph. If we delete any one edge from the graph shown in figure 7.7(i), then we get 4 trees shown in figure 7.7(ii).

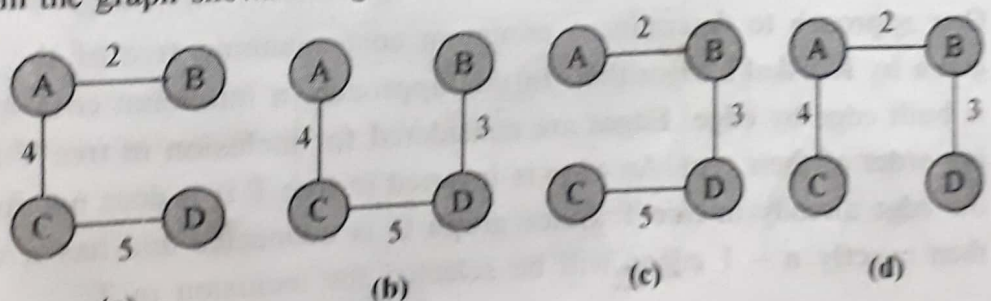


Fig. 7.7(ii)

Here all these trees are spanning trees. The number of edges are 3, which is 1 less than the number of nodes.

When determining the cost of a spanning tree of a weighted graph, the cost is simply the sum of the weights of the tree edges in figure 7.7 (ii) the costs of trees are 11, 12, 10 and 9 for trees a, b, c, and d respectively. A minimum cost spanning tree is formed when the edges are picked to minimize the total cost so, the minimum cost spanning tree among all 4 stated above is (d) with cost 9.

A method to obtain a minimum cost spanning tree builds this tree edge by edge. The next edge to include is choosen according to some optimization criterion. The simplest criterion is to choose an edge that result in a minimum increase in the sum of the cost of the edges so far included. There are two possible ways to interpret this criterion.

1. Kruskal's Algorithm

In this method the edges of the graph are considered in non decreasing order of cost. This interpretation is that the set 't' of edges so far selected for the spanning tree be such that it is possible to complete 't' into a tree. Therefore 't' may not be a tree at all stages in the algorithm. In fact, it will generally only be a forest since the set of edges 't' can be completed into a tree if and only if there are no cycles in 't'. This method is due to Kruskal.

2. Prim's Algorithm

In this method the set of edges so far included or selected form a tree. Therefore is A is the set of selected edges, then A forms a tree. The next edge (u, v) to be included in tree A is a minimum cost edge not in A with the property that (AU{u, v}) is also a tree. The corresponding algorithm is known as prim's algorithm. Let us discuss each method in details.

7.7.1 Kruskal's Algorithm

One approach to determine a minimum cost spanning tree of the graph has been given by Kruskal's algorithm. In this approach, a minimum cost spanning tree (T) is built edge by edge. Edges are considered for inclusion in tree T in node increasing order of their cost. An edge is inserted in tree T if it does not form a cycle with the edge already in tree T. Since graph G is connected and has n vertices (n > 0), then exactly n – 1 edges will be selected for inclusion in T.

In Kruskal's algorithm, initially E is the set of all edges in graph G. The only function, we wish to perform on this set are –

1. Determining an edge with minimum cost
2. Deleting that edge from graph and add to tree

Both of these functions can be performed efficiently if the edges in E are maintained as a sorted sequential list.

We have already discussed that a minimum cost spanning tree is formed when edges are picked to minimize the cost. Minimum spanning tree is the spanning tree in which the sum of weights (cost) on edges is minimum. It is a method in which edges of the graph are considered in increasing order of cost. Figure 7.7(a) shows a graph for which we want to create a minimum cost spanning tree using Kruskal's algorithm.

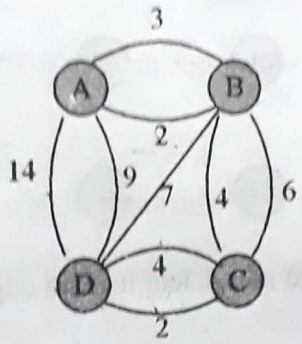


Fig. 7.7(a)

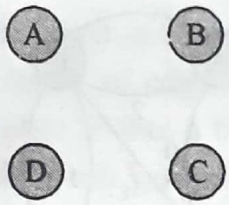
Now, at first we'll create list of edge with weight in graph.

edge	weight
AB	2
AB	3
BC	4
BC	6
BD	7
CD	4
CD	2
DA	9
DA	14

Now we'll sort this list in ascending order of weights:

edge	weight
AB	2
CD	2
AB	3
BC	4
CD	4
BC	6
BD	7
DA	9
DA	14

Now draw all the nodes (without any edge) of graph to get spanning tree:



Now, select first entry from edge list, delete it from edge list and draw it on spanning tree skeleton

edge	*weight*
~~*AB*~~	~~2~~
CD	2
AB	3
BC	4
CD	4
BC	6
BD	7
DA	9
DA	14

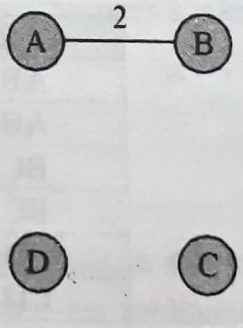


Fig. 7.7 (b)

Here, AB was shortest weighted edge and so it is drawn on skeleton for spanning tree and it has been cut from edge list.

Now, we will select second entry in edge list, which should not construct a cycle in tree. Here, CD is smallest weighted edge with weight = 2 and is not forming a cycle. Therefore it is added to tree, and cut from edge list.

edge	Weight
~~AB~~	~~2~~
~~CD~~	~~2~~
AB	3
BC	4
CD	4
BC	6
BD	7
DA	9
DA	14

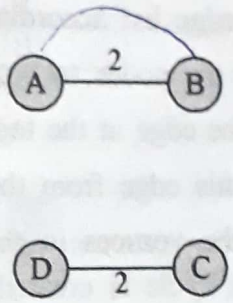


Fig. 7.7 (c)

Now, our aim is to select next smallest weighted edge but which is not forming cycle.

If we select AB, with 3, then it'll form cycle, but BC, with 4, will not from cycle. So, we will select BC as next edge and thus

edge	Weight
~~AB~~	~~2~~
~~CD~~	~~2~~
AB	3
~~BC~~	~~4~~
CD	4
BC	6
BD	7
DA	9
DA	14

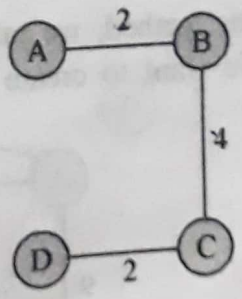


Fig. 7.7 (d)

And, this is our final spanning tree with minimum weight i.e. 2+4+ 2 = 8, and here no. of nodes = 4 so we are selecting only n–1 i.e. 3 edges.

Kruskal's Algorithm

This algorithm will create spanning tree with minimum weight, from a given weighted graph.

Step1. Begin

Step2. Create the edge list of given graph, with their weights.

Step3. Sort the edge list according to their weights in ascending order.

Step4. Draw all the nodes to create skeleton for spanning tree.

Step5. Pick up the edge at the top of the edge list (i.e. edge with minimum weights).

Step6. Remove this edge from the edge list

Step7. Connect the vertices in the skeleton with given edge. If by connecting the vertices, a cycle is created in the skeleton, then discard this edge.

Step8. Repeat steps 5 to 7, until n–1 edges are added or list of edges is over

Step9. Return

7.7.2 Prim's Algorithm

There is one more approach to determine minimum cost spanning tree given by prim's algorithm. In this case, we start with single edge of graph, and we add edges to it and finally we get minimum cost tree. In this case, as well, we have n–1 edges when number of nodes in graph are n.,

In this case, we again and again add edges to tree, and tree is extended to create spening tree, while in case of Kruskal's algorithm, there may be more than one tree, which is finally connected through edge to create spanning tree.

To illustrate this method, we take figure 7.8 (a) as example. Figure 7.8 (a) is the graph for which we want to create spanning tree

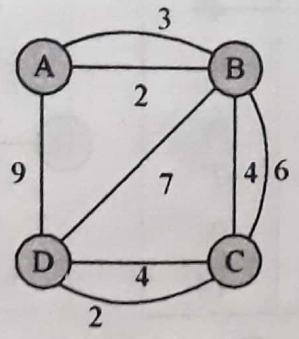


Fig. 7.8 (a)

At first we'll create edge list (with weights) of this graph

edge	weight
AB	2
AB	3
AD	9
BC	4
BC	6
BD	7
CD	4
CD	2

and create skeleton for spanning tree

Fig. 7.8 (b)

Now, select any edge with minimum weight from edge list, and there are 2 edges AB and CD with weight 2, so we'll select any one from them, and delete it from edge list and add it to skeleton.

edge	weight
~~AB~~	~~2~~
AB	3
AD	9
BC	4
BC	6
BD	7
CD	4
CD	2

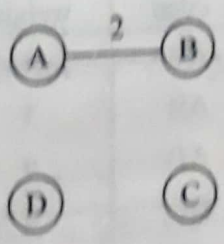


Fig. 7.8 (c)

Now, select edges, whose one end point is either A or B, and such edges are AB(3), AD(9), BC(4), BC(6), BD (7). Out of all these, AB forms cycle so discard it, and among others, BC(4) is edge with minimum weight, and so.

edge	weight
~~AB~~	~~2~~
AB	3
AD	9
~~BC~~	~~4~~
BC	6
BD	7
CD	4
CD	2

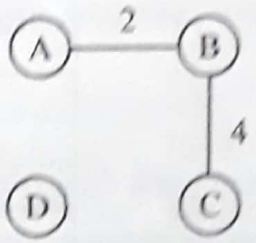


Fig. 7.8 (d)

Now, we need to select edges, whose one end point is either A, B or C and these edges are–

AB (3), AD (9), BC (6), BD (7), CD (4), CD (2)

But AB (3), BC (6), forms cycle so discard them and so, remaining edges are

AD (9), BD (7), CD (4), CD (2)

and out of all these edges, CD is edge with smallest weight i.e. 2 so we'll operate on this edge as:

edge	weight
~~AB~~	~~2~~
AB	3
AD	9
~~BC~~	~~4~~
BC	6
BD	7
CD	4
~~CD~~	~~2~~

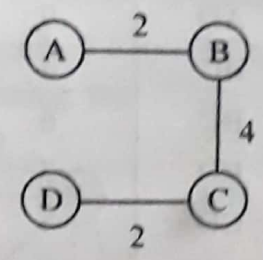


Fig. 7.8 (e)

As we have added n–1 i.e. 3 edges to tree so no more edges are needed.

Prim's Algorithm

This algorithm creates spanning tree with minimum weight from a given weighted graph.

Step1. Begin

Step2. Create edge list of given graph, with their weights.

Step3. Draw all nodes to create skeleton for spanning tree.

Step4. Select an edge with lowest weight and add it to skeleton and delete edge from edge list.

Step5. Add other edges. While adding an edge take care the one end of the edge should always be in the skeleton tree and its cost should be minimum.

Step6. Repeat step 5 until n–1 edges are added.

Step7. Return

7.8 Operations on Graph

As we have seen, a graph can be represented in two ways.

1. Adjacency matrix
2. Adjacency list

The two main operations on graph will be

1. Insertion
2. Deletion

Here insertion and deletion will be also on two things -

1. On node
2. On edge

Now we will describe insertion and deletion operation on adjacency matrix and adjacency list.

7.8.1 Insertion in Adjacency Matrix

1. Node Insertion

Insertion of node requires only addition of one row and one column with zero entries in that row and column. Let us take a graph G shown in figure 7.8(i)

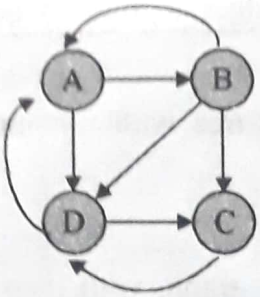


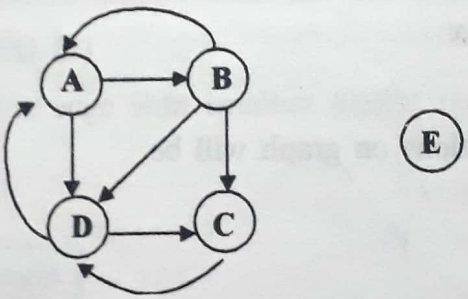
Fig. 7.8(i)

Now the adjacency matrix will be-

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

Suppose we want to add one node E in the graph then we have a need to add one row and one column with all zero entries for node E.

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



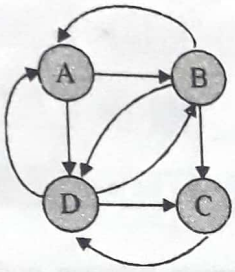
2. Edge insertion

We know that entry of adjacency matrix 1 represents the edge between two nodes, and 0 represents no edge between those two nodes. Therefore, insertion of edge requires changing the value 0 into 1 for those particular nodes.

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

Here we can see that there is no edge between D to B. So adjacency matrix has 0 entry at 4^{th} row 2^{nd} column. Suppose we want to insert an edge between D to B, then we have a need to change the 0 entry into 1 at the position 4^{th} row and 2^{nd} column. Now the adjacency matrix will be-

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



7.8.2 Deletion in Adjacency Matrix

1. Node Deletion

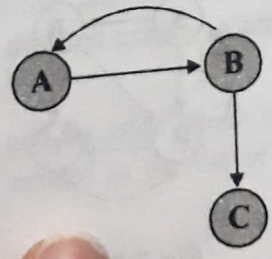
Deletion of node requires deletion of that particular row and column in adjacency matrix for node to be deleted, because node deletion requires deletion of all the edges which are connected to that particular node.

Let us take an adjacency matrix-

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

Suppose we want to delete the node D, then 4^{th} row and 4^{th} column of adjacency matrix will be deleted. Now the adjacency matrix will be-

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



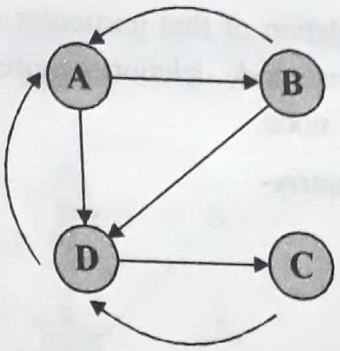
2. Edge Deletion

Deletion of an edge requires changing the value 1 to 0 for those particular nodes. Let us have an adjacency matrix-

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

Here we can see that an edge exists between node B to node C. So adjacency matrix has entry 1 at 2nd row 3rd column. Suppose we want to delete the edge which is in between B and C, then we have a need to change the entry 1 to 0 at the position 2nd row, 3rd column. Now the adjacency matrix will be-

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



7.8.3 Insertion in Adjacency List

1 Node Insertion

Insertion of node in adjacency list requires only insertion of that node in header nodes of adjacency list. Let us take a graph G-

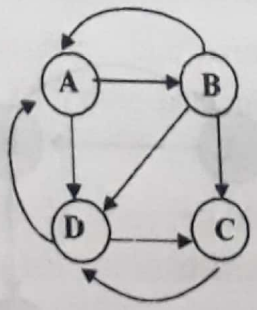


Fig.7.8(ii)

The adjacency list for this graph will be as-

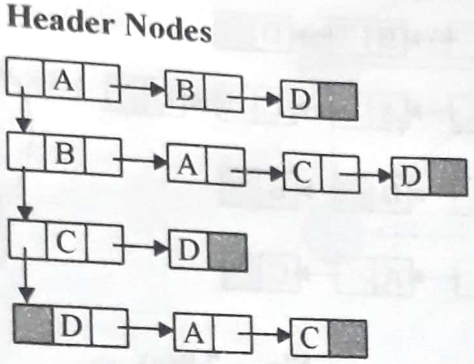


Fig. 7.8(iii)

Suppose we want to insert one node E then it requires addition of node E in header node only. Now the adjacency list will be-

Header Nodes

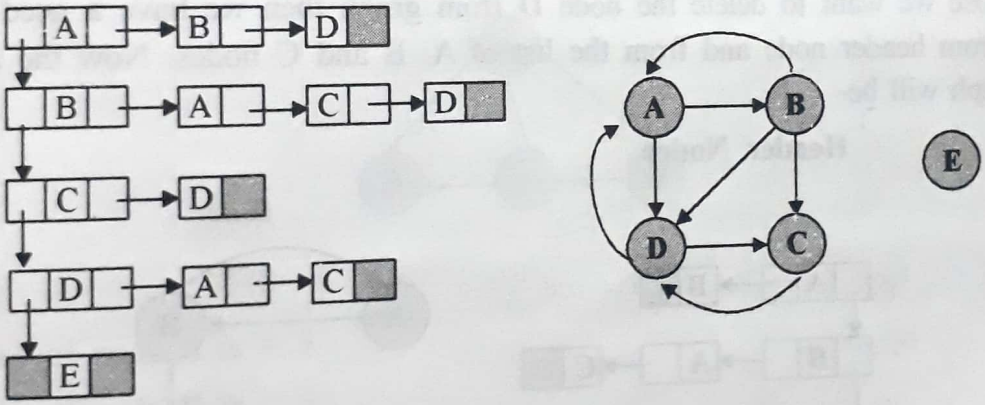


Fig. 7.8(iv)

2. Edge Insertion

Insertion of an edge requires add operation in the list of the starting node of edge. Remember here graph is directed graph. In undirected graph, it will be added in the list of both nodes.

Suppose we want to add the edge which starts from node C and ends edges at node B, then add operation is needed in the list of C node. Now the adjacency list of graph will be-

Header Nodes

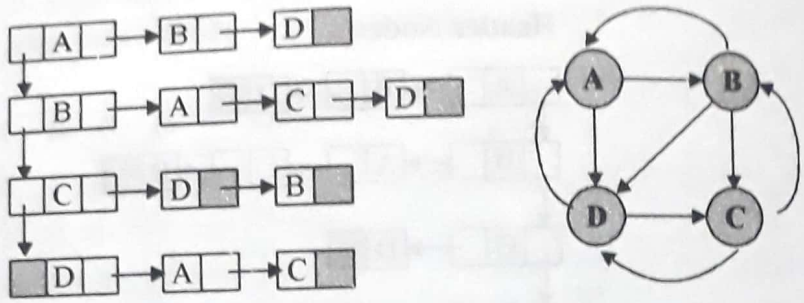


Fig. 7.8(v)

7.8.4 Deletion in Adjacency List

1. Node Deletion

Deletion of node requires deletion of that particular node from header node and from the entire list wherever it is coming. Deletion of node from header will automatically free the list attached to that node.

Suppose we want to delete the node D from graph then we have a need to delete node D from header node and from the list of A, B and C nodes. Now the adjacency list of graph will be-

Header Nodes

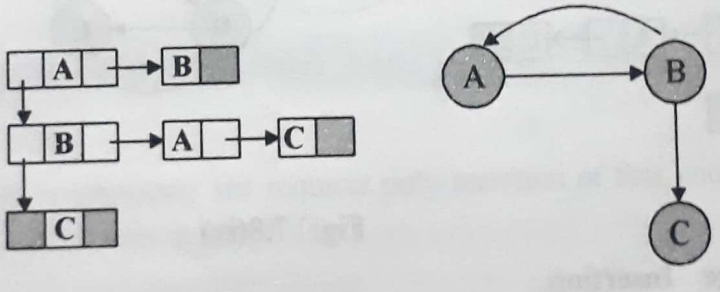


Fig. 7.8(vi)

2. Edge Deletion

Deletion of edge requires deletion in the list of that node where edge starts, and that element of the list will be deleted where the edge ends.

Suppose we want to delete the edge which starts from D and ends at A then we have a need to delete in the list of node D and element in the list deleted will be A. Now the adjacency list of the graph will be-

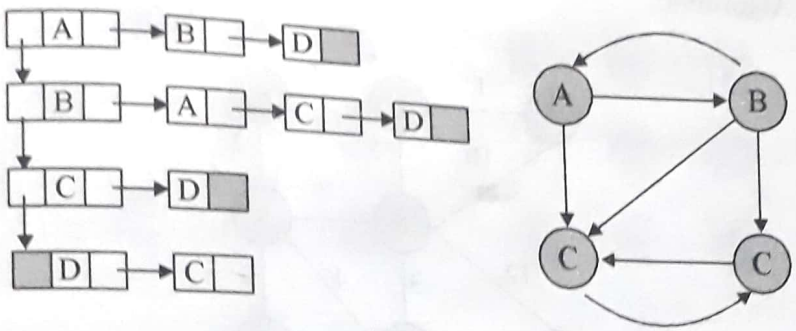
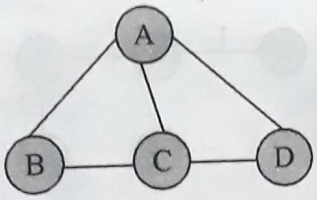


Fig. 7.8(vii)

Solved Examples

Ex.1 What is the relationship between the sum of the degree of the vertices of a graph and the number of edges in the graph?

Sol. Let the graph is-



Degree of node A is = 3

Degree of node B is = 2

Degree of node C is = 3

Degree of node D is = 2

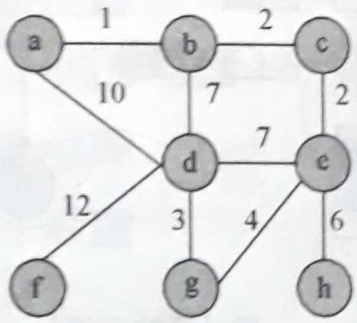
sum of degree = 10

$$\text{Number of edges} = \frac{\textit{Sum of Degree}}{2}$$

$$= \frac{10}{2} = 5$$

Relation: Number of edge is the half of the sum of degree of the vertices in the graph.

Ex.2 Consider the following graph, find the minimum cost spanning tree for the graph using Kruskal's Algorithm.



Sol. Kruskal's Algorithm is used to find the minimum cost spanning tree. According to Kruskal's Algorithm, we can start from the edge of minimum distance. Lets choose vertices 'a' and 'b' which have minimum distance.

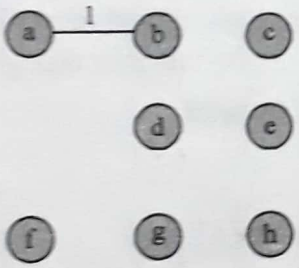


Fig. (1)

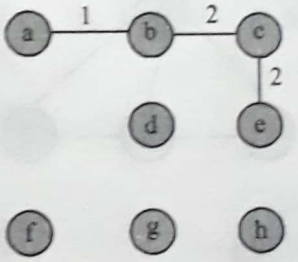


Fig. (2)

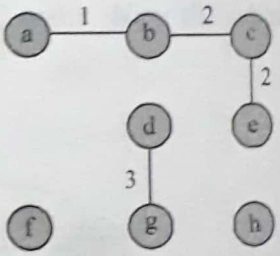


Fig. (3)

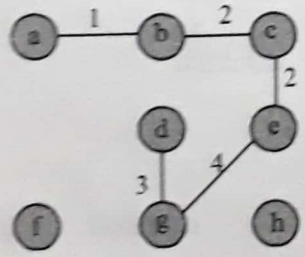


Fig. (4)

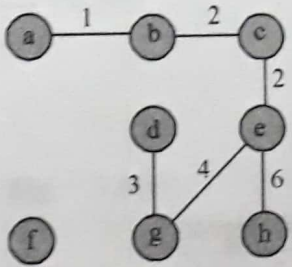


Fig. (5)

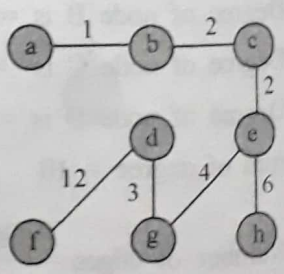


Fig. (6)

Cost = 1 + 2 + 2 + 6 + 4 + 3 + 12 = 30

Ex.3 Solve Q.2. with prim's Algorithm

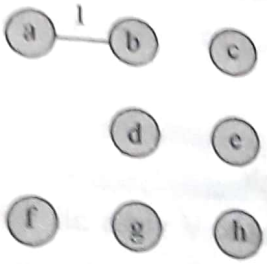


Fig. 1

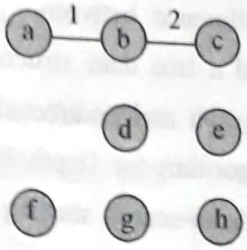


Fig. 2

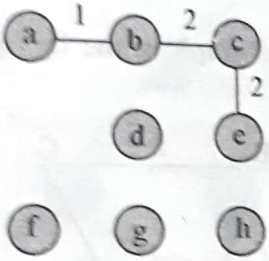


Fig. 3

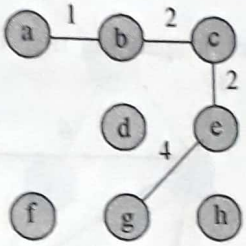


Fig. 4

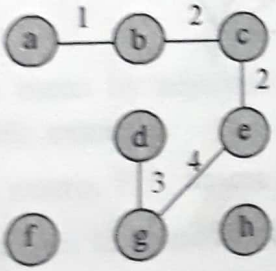


Fig. 5

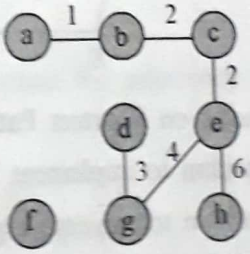


Fig. 6

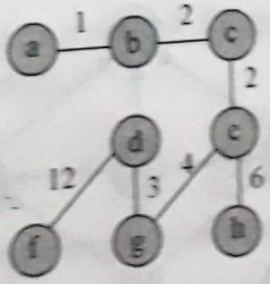


Fig. 7

Cost = 1 + 2 + 2 + 3 + 4 + 6 + 12 = 30