CHAPTER 3

Graph

Objectives ...

- To study Basic Concepts of Graph Data Structure
- To learn Graph Terminology
- To understand Representation and Operations of Graph
- To study Graph Traversals
- To learn Greedy Strategy, Dynamic Programming Strategy of Graphs

3.0 INTRODUCTION

- Graph is one of the most important non-linear data structure. A graph is a pictorial representation of a set of objects where some pairs of objects (vertices) are connected by links (edges).
- A graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices.

3.1 CONCEPT AND TERMINOLOGY

- In this section we study basic concepts and terminology in graph.

3.1.1 Concept

- A graph G is a set of two tuples G = (V, E) where, V is a finite non-empty set of vertices, and E is the set of pairs of vertices called edges.
- Fig. 3.1 shows an example of graph.

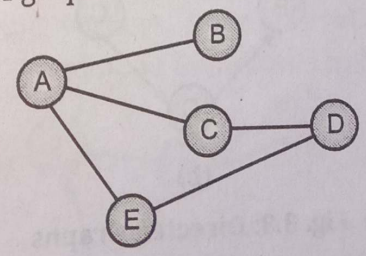


Fig. 3.1: Graph

The set of vertices in the above graph is, V = {A, B, C, D, E}.

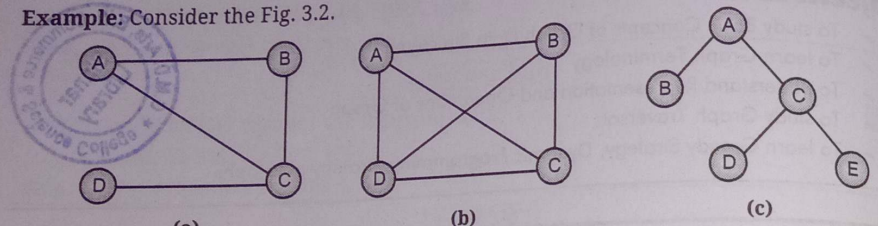
The set of edges, E = {(A, B), (A, C), (C, D), (A, E), (E, D)}.

- There are two types of Graph, Undirected graph and Directed graph.

1. Undirected Graph:

- In an undirected graph, the pair of vertices representing any edge is unordered i.e. the pairs (v_1, v_2) and (v_2, v_1) represent the same edge.
- In other words, the edges have no direction in undirected graph.

Example: Consider the Fig. 3.2.



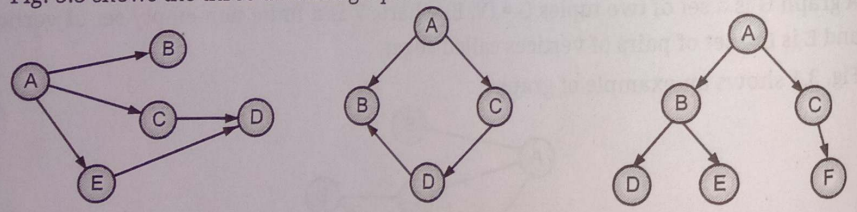
(a) (b) (c)

Fig. 3.2: Undirected Graph

- In Fig. 3.2 (a) G = (V, E) where V = {A, B, C, D} and E = {(A, B), (A, C), (B, C), (C, D)}.
- In Fig. 3.2 (b) G = (V, E) where V = {A, B, C, D} and E = {(A, B), (A, C), (A, D), (B, D), (B, C), (D, C)}.
- In Fig. 3.2 (c) G = (V, E) where V = {A, B, C, D, E} and E= {(A, B), (A, C), (C, D), (C, E)}. This undirected graph is called Tree. Tree is a special case of graph.

2. Directed Graph:

- In a directed graph each edge is represented by a directed pair (v_1, v_2), v_1 is the tail and v_2 is head of the edge i.e. the pairs (v_1, v_2) and (v_2, v_1) are different edges.
- In other words, the edges have direction in directed graph. Directed graph is also called as Digraph.
- Fig. 3.3 shows the three directed graphs.



(a) (b) (c)

Fig. 3.3: Directed Graphs

- In Fig. 3.3 (a) G = (V, E) where V = {A, B, C, D, E} and E = {(A, B), (A, C), (C, D), (A, E), (E, D)}.
- In Fig. 3.3 (b) G = (V, E) where V = {A, B, C, D} and E = {(A, B), (A, C), (C, D), (D, B)}.
- In Fig. 3.3 (c) G = (V, E) where V = {A, B, C, D} and E = {(A, B), (A, C), (B, D), (B, E), (C, F)}.

- Following table compares tree and graph data structures:

Sr. No.	Tree	Graph
1.	A tree is a data structure in which each node is attached to one or more nodes as children.	A graph is a collection of vertices or nodes, which are joined as pairs by lines (links) or edges.
2.	Tree is a non-linear data structure.	Graph is also a non-linear data structure.
3.	All the trees can be graphs.	All the graphs are not trees.
4.	The common tree traversal methods are inorder, preorder, postorder traversals.	The two common graph traversal method are Breadth First Search (BFS) and Depth First Search (DFS).
5.	It is undirected and connected.	It can be directed or undirected; can be connected or not-connected.
6.	It cannot be cyclic.	It can be cyclic or acyclic.
7.	There is a root (first) node in trees.	There is no root node in graph.
8.	Tree data is represented in hierarchical manner so parent to child relation exists between the nodes.	Graph did not represent the data is hierarchical manner so there is no parent child relation between data representation.
9.	A tree cannot have a loop structure.	A graph can have a loop structure.

3.1.2 Terminology

- Basic graph terminology listed below:

1. **Adjacent Vertex:** When there is an edge from one vertex to another then these vertices are called adjacent vertices. Node 1 is called adjacent to node 2 as there exists an edge from node 1 and node 2 as shown in Fig. 3.4.

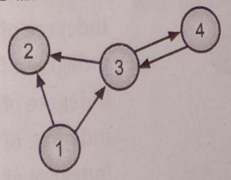


Fig. 3.4

2. **Cycle:** A path from a vertex to itself is called a cycle. Thus, a cycle is a path in which the initial and last vertices are same.

Example: Fig. 3.5 shows the path (A, B, C, A) or (A, C, D, B, A) are cycles of different lengths. If a graph contains a cycle it is cyclic otherwise acyclic.

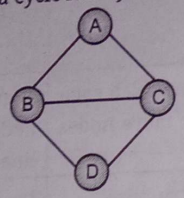


Fig. 3.5: Cycle

3. **Complete Graph:** A graph G is said to be complete if every vertex in a graph is adjacent to every other vertex. In this graph, number of edges = n(n-1)/2, where n = no. of vertices.

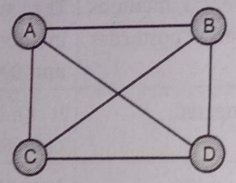


Fig. 3.6: Complete Graph

Example: In Fig. 3.6, Number of vertices n=4, Number of edges=4(4–1)/2=6.

4. **Connected Graph:** An undirected graph G is said to be connected if for every pair of distinct vertices v_i, v_j in V(G) there is a path from v_i to v_j.

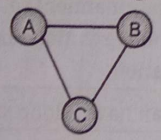


Fig. 3.7: Connected Graph

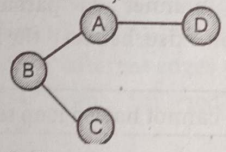


Fig. 3.8 Non-connected Graph

5. **Degree of a Vertex:** It is the number of edges incident to a vertex. It is written as degree(V), where V is a vertex.
6. **In-degree of a Vertex:** In directed graph, in-degree of a vertex 'v' is the number of edges for which 'v' is the head.

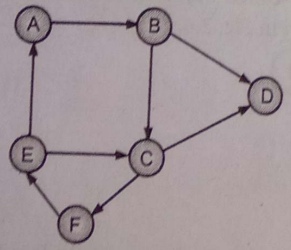


Fig. 3.9: Digraph

Indegree of vertex B = 1
Indegree of vertex C = 2
Indegree of vertex D = 2
Indegree of vertex E = 1
Indegree of vertex F = 1
Indegree of vertex A = 1

7. **Out-degree of a Vertex:** In directed graph, the out-degree of a vertex 'v' is the total number of edges for which 'v' is the tail.

From Fig. 3.9:
Outdegree of A = 1
Outdegree of B = 2
Outdegree of C = 2
Outdegree of D = 0
Outdegree of E = 2
Outdegree of F = 1

8. **Isolated Vertex:** If any vertex does not belong to any edge then it is called isolated vertex.

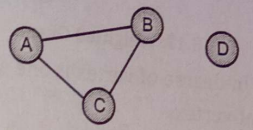


Fig. 3.10: Isolated Node D in the Graph

9. **Source Vertex:** A vertex with in-degree zero is called a source vertex, i.e., vertex has only outgoing edges and no incoming edges. For example, in Fig. 3.11, 'C' is source vertex.
10. **Sink Vertex:** A vertex with out-degree zero is called a sink vertex i.e. vertex has only incoming edge and no outgoing edge. For example, in Fig. 3.11, 'B' is sink node.
11. **Acyclic Graph:** A graph without cycle is called acyclic graph. **[Oct. 18]**

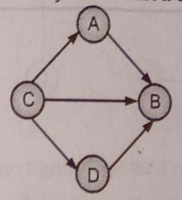
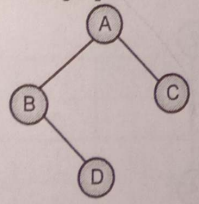
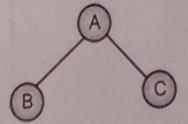


Fig. 3.11: Acyclic Graph

12. **Subgraph:** A subgraph of G is a graph G' such that V(G') ⊆ V(G) and E(G') ⊆ E(G).



(a) Graph G



(b) Subgraph of G

Fig. 3.12

IMP

13. Weighted Graph: A weighted graph is a graph in which every edge is assigned a weight. In Fig. 3.13 weight in a graph denote the distance between the two vertices connected by the corresponding edge. The weight of an edge is also called its cost. In case of a weighted graph, an edge is a 3-tuple (U, V, W) where U and V are vertices and W is weight of an edge (U, V).

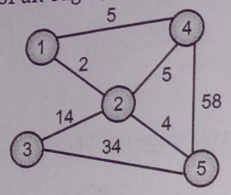


Fig. 3.13: Weighted Graph

14. Pendant Vertex: When in-degree of vertex is one and out-degree is zero then such a vertex is called pendant vertex.

15. Spanning Tree: A spanning tree of a graph G = (V, E) is a subgraph of G having all vertices of G and containing only those edges that are necessary to join all the vertices in the graph. A spanning tree does not contain cycle in it. Fig. 3.14 shows spanning Trees for graph in Fig. 3.6.

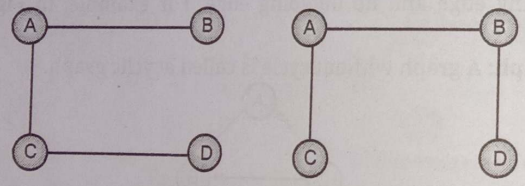


Fig. 3.14: Spanning Trees

16. Sling or Loop: An edge of a graph, which joins a node to itself, is called a sling or loop. Fig. 3.15 shows an example of loop.

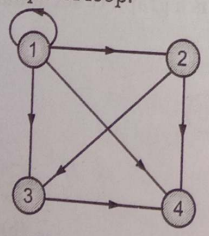


Fig. 3.15: Loop

3.2 REPRESENTATION OF GRAPH

- Representation of graph is a process to store the graph data into the computer memory. Graph is represented by the following three ways:
  1. Sequential representation using Arrays, (by means of Adjacency Matrix).
  2. Linked representation using Linked List, (by means of Adjacency List).
  3. Inverse adjacency list.
  4. Adjacency multi-list representation.

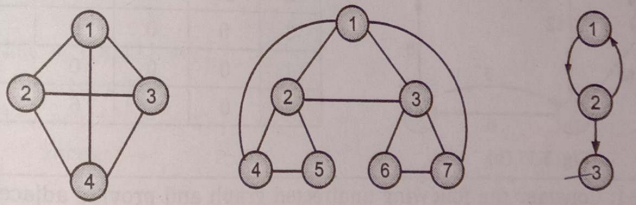
3.2.1 Sequential Representation of Graph [April 16, 17, 18, 19 Oct. 17, 18]

- Graphs can be represented through matrix (array) in computer system's memory. This is sequential in nature. This type of representation is called sequential representation of graphs.
- The graph when represented using sequential representation using matrix, is commonly known as Adjacency Matrix.
- Let G = (V, E) be a graph with n vertices, where n>=1. An adjacency matrix of G is a 2-dimentional n × n array, say A, with the following property:

  A[i][j] = 1 if the edge (v_i, v_j) is in E(G)
  = 0 if there is no such edge in G

- If the graph is undirected then,

  A[i] [j] = A[j] [i] = 1



G_1 G_2 G_3

Fig. 3.16: Undirected Graph G_1, G_2 and Directed Graph G_3

- The graphs G_1, G_2 and G_3 of Fig. 3.16 are represented using adjacency matrix in Fig. 3.17.

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

G1

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

G3

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

G_2

Fig. 3.17 (a): Adjacency Matrix for G_1, G_2 and G_3 of Fig. 3.16

- **Adjacency Matrix Representation of a Weighted Graph:**

For weighted graph, the matrix A is represented as,

A[i] [j] = weight of the edge (i, j)

= 0 otherwise 1

Here, weight is labelled associated with edge.

Example, following is the weighted graph and its associated adjacency matrix.

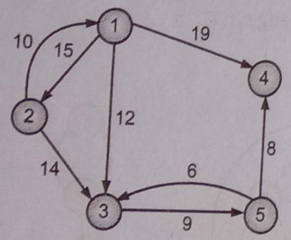
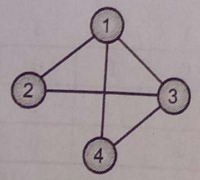


Fig. 3.17 (b)

	1	2	3	4	5
1	0	15	12	19	0
2	10	0	14	0	0
3	0	0	0	0	9
4	0	0	0	0	0
5	0	0	6	8	0

Example 1: Consider the following undirected graph and provide adjacency matrix. The graph has 4 vertices and it is undirected graph. Write 1 to Number of vertices i.e. 1 to 4 as row and column headers to represent it in adjacency matrix. If edge exists between any two nodes (row, column headers indicate nodes) write it as 1 otherwise 0 in the matrix.



(a) Undirected Graph

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

(b) Adjacency Matrix

Fig. 3.18

Example 2: Consider the following directed graph and provide adjacency matrix. The graph has 5 vertices and it is directed graph. Write 1 to Number of vertices i.e. 1 to 5 as row and column headers to represent it in adjacency matrix. If edge exists between any two nodes (row, column headers indicate nodes) write it as 1 at (i, j) and (j, i) position in matrix otherwise 0 in the matrix.

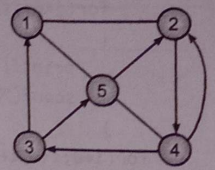


Fig. 3.19

The representation of the above graph using adjacency matrix is given below:

$$\text{Adj-matrix} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

Program 3.1: Program to represent graph as adjacency matrix.

```
#include <stdio.h>
#define MAX 10
void degree(int adj[][MAX],int x,int n)
{
   int i,incount=0, outcount =0;
   for(i=0;i<n;i++)
   {
      if( adj[x][i] ==1)
         outcount++;
      if( adj[i][x] ==1)
         incount++;
   }
   printf("The indegree of the node %d is %d\n",x,incount++);
   printf("The outdegree of the node %d is %d\n",x,outcount++);
}
int main()
{
   int adj[MAX][MAX],n,i,j;
   setbuf(stdout, NULL);
   printf("Enter the total number of nodes in graph");
   scanf("%d",&n);
   for(i=0;i<n;i++)
```

```
        for(j=0;j<n;j++)
        {
           printf("Enter Edge from %d to %d,(1: Edge 0: No edge) \n",i,j);
           scanf("%d",&adj[i][j]);
        }
     for(i=0;i<n;i++)
     {
        degree(adj,i,n);
     }
     return 0;
  }
```

Output:

```
Enter the total number of nodes in graph4
Enter Edge from 0 to 0,(1: Edge 0: No edge)
0
Enter Edge from 0 to 1,(1: Edge 0: No edge)
1
Enter Edge from 0 to 2,(1: Edge 0: No edge)
0
Enter Edge from 0 to 3,(1: Edge 0: No edge)
1
Enter Edge from 1 to 0,(1: Edge 0: No edge)
1
Enter Edge from 1 to 1,(1: Edge 0: No edge)
0
Enter Edge from 1 to 2,(1: Edge 0: No edge)
1
Enter Edge from 1 to 3,(1: Edge 0: No edge)
1
Enter Edge from 2 to 0,(1: Edge 0: No edge)
0
Enter Edge from 2 to 1,(1: Edge 0: No edge)
1
Enter Edge from 2 to 2,(1: Edge 0: No edge)
0
Enter Edge from 2 to 3,(1: Edge 0: No edge)
1
Enter Edge from 3 to 0,(1: Edge 0: No edge)
1
Enter Edge from 3 to 1,(1: Edge 0: No edge)
1
Enter Edge from 3 to 2,(1: Edge 0: No edge)
1
Enter Edge from 3 to 3,(1: Edge 0: No edge)
0
```



```
The indegree of the node 0 is 2
The outdegree of the node 0 is 2
The indegree of the node 1 is 3
The outdegree of the node 1 is 3
The indegree of the node 2 is 2
The outdegree of the node 2 is 2
The indegree of the node 3 is 3
The outdegree of the node 3 is 3
```

Advantages of Array representation of Graph:

1. Simple and easy to understand.
2. Graphs can be constructed at run-time.
3. Efficient for dense (lots of edges) graphs.
4. Simple and easy to program.
5. Adapts easily to different kinds of graphs.

Disadvantages of Array representation of Graph:

1. Adjacency matrix consumes huge amount of memory for storing big or large graphs.
2. Adjacency matrix requires huge efforts for adding/removing a vertex.
3. The matrix representation of graph does not keep track of the information related to the nodes.
4. Requires that graph access be a command rather than a computation.

3.2.2 Linked Representation of Graphs

[April 16, 17, 18, 19 Oct. 17, 18]

- We use the adjacency list for the linked representation of the graph. In adjacency lists representation the n rows of the adjacency matrix are represented as n linked lists.
- The adjacency list representation maintains each node of the graph and a link to the nodes that are adjacent to this node. When we traverse all the adjacent nodes, we set the next pointer to null at the end of the list.
- **Example:** Consider the graph G. The Fig. 3.20 shows the graph G and linked representation of G in memory. The linked representation will contain two lists:
  1. **A node vertex list:** To keep the track of all the N nodes of the graph.
  2. **An edge list:** To keep the information of adjacent vertices of each and every vertex of a graph.
- Head node is used to represent each list, i.e. we can represent G by an array Head, where Head [i] is a pointer to the adjacency list of vertex i.

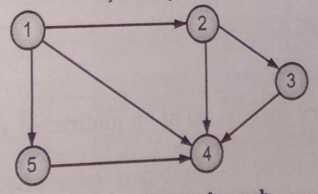


Fig. 3.20: Directed graph

Vertex and its Adjacent Vertices	
Vertex	**Adjacent Vertices**
1	2, 4, 5
2	3, 4
3	4
4	–
5	4

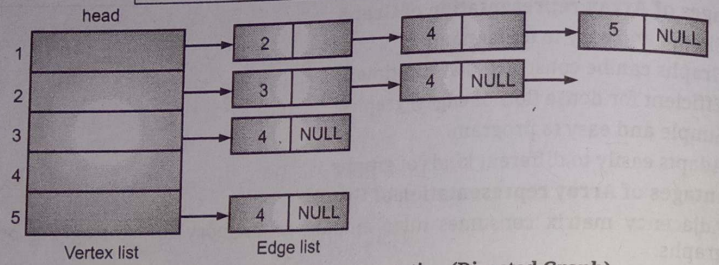
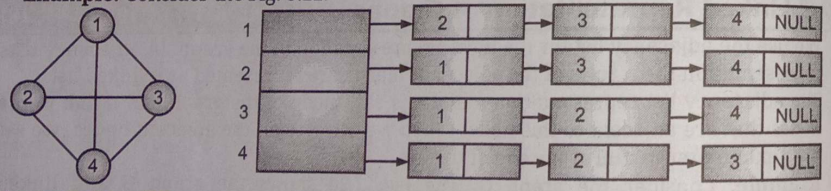


Fig. 3.21: Adjacency List Representation (Directed Graph)

Adjacency List Representation for Undirected Graph:

- In this representation, n rows of the adjacency matrix are represented as n linked lists. The nodes in list i represent the vertices that are adjacent to vertex i. head[i] is used to represent i^{th} vertex adjacency list.
- **Example:** Consider the Fig. 3.22.



Undirected graph G_1

Fig.3.22: Adjacency List Representation of G_1 (Undirected Graph)

- The structure for adjacency list representation can be defined in C as follows:

```
#define MAX_V 20
struct node
{
    int vertex;
    struct node * next;
}
struct node head[MAX_V];
```

- In this representation every edge (v_i, v_j) of an undirected graph is represented twice, once in the list of v_i and in the list of v_j.
- For directed graph time required is O(n) to determine whether, there is an edge from vertex i to vertex j and for undirected graph time is O(n + e).

Example 1: Consider the undirected graph in Fig. 3.23 and provide adjacency list.

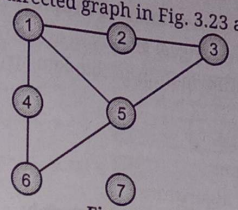


Fig. 3.23

Solution: The adjacency list is given in Fig. 3.24

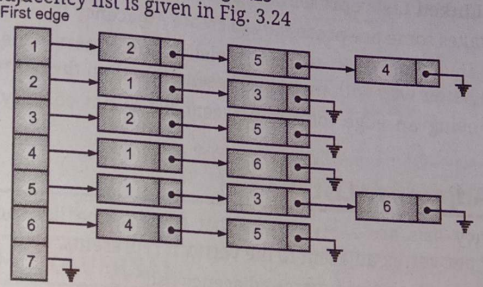


Fig. 3.24

Example 2: Consider the weighted undirected graph in Fig. 3.25 and provide adjacency list.

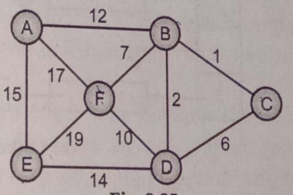


Fig. 3.25

Solution: An adjacency list for this graph is:

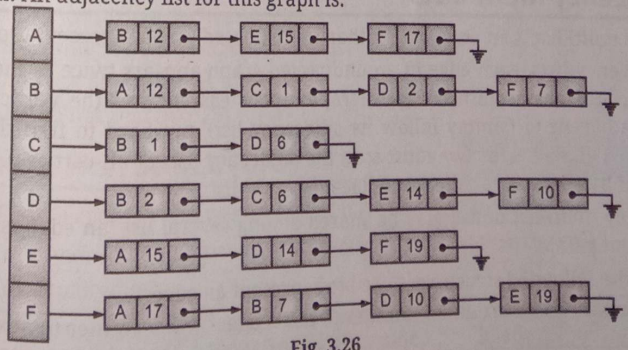


Fig. 3.26

Advantages of Linked List representation n of Graph:

1. Less storage for sparse (few edges) graphs.
2. Easy to store additional information in the data structure like vertex degree, edge weight etc.
3. Better memory space usage.
4. Better graph traversal times.
5. Generally better for most algorithms.

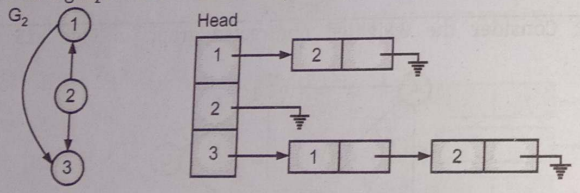
Disadvantages of Linked List representation of Graph:

1. Generally, takes some pre-processing to create Adjacency list.
2. Algorithms involving edge creation, deletion and querying edge between two vertices are better way with matrix representation than the list representation.
3. Adding/removing an edge to/from adjacent list is not so easy as for adjacency matrix.

3.2.3 Inverse Adjacency List

[Oct. 17]

- Inverse adjacency lists are a set of lists that contain one list for vertex. Each list contains a node per vertex adjacent to the vertex it represents.
- Fig. 3.27 shows a graph and its inverse adjacency list.



(a) Graph (b) Linked List

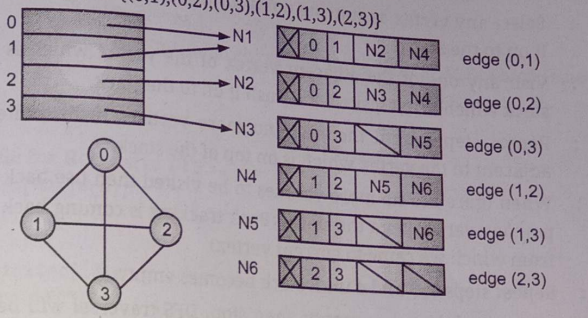
Fig. 3.27

3.2.4 Adjacency Multi-Lists

- Adjacency multi-list is an edge, rather than vertex based graph representation.
- With adjacency lists, each edge in an undirected graph appears twice in the list. Also, there is an obvious asymmetry for digraphs - it is easy to find the vertices a given vertex is adjacent to (simply follow its adjacency list), but hard to find the vertices adjacent to a given vertex (we must scan the adjacency lists of all vertices). These can be rectified by a structure called an adjacency multi-list.
- In adjacency multi-list nodes may be shared among several list (an edge is shared by two different paths).
- Typically, the following structure is used to represent an edge.

Visited	Vertex at tail	Vertex at head	Pointer to next edge containing vertex at tail	Pointer to next edge containing vertex at head

Example: Consider following undirected graph G = (V, E)
where, V={0,1,2,3}, E={(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)}



Lists: Vertex 0: N1→N2→N3, Vertex 1: N1→N4→N5
Vertex 2: N2→N4→N6, Vertex 3: N3→N5→N6

Fig. 3.28: Adjacency Multi-List Representation

3.3 GRAPH TRAVERSAL

- Graph traversal means visiting every vertex and edge exactly once in a well-defined order.
- In many situations, we need to visit all the vertices and edges in a systematic fashion. The graph traversal is used to decide the order of vertices to be visited in the search process.
- A graph traversal finds the edges to be visited without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.
- Traversal means visiting each element of the data structure representation, in graph it refers to visiting each vertex of the graph. Traversing a graph means visiting all the vertices in a graph exactly one.
- The two techniques are used for traversals:
  1. Depth First Search (DFS), and
  2. Breadth First Search (BFS).

3.3.1 Depth First Search (DFS)

[April 16, 17, 18, 19 Oct. 17, 18]

- Depth First Search (DFS) is used to perform traversal of a graph.
- In this method, all the vertices are stored in a Stack and each vertex of the graph is visited or explored once.
- The newest vertex (added last) in the Stack is explored first. To traverse the graph, Adjacent List of a graph is created first.

- We use the following steps to implement DFS traversal:

Step 1 : Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 2 : Visit any one of the adjacent vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.

Step 3 : Repeat step 2 until there are no more vertices to be visited which are adjacent to the vertex which is on top of the stack.

Step 4 : When there are no more vertices to be visited then use back tracking and pop one vertex from the stack. (Back tracking is coming back to the vertex from which we came to current vertex)

Step 5 : Repeat steps 2, 3 and 4 until stack becomes empty.

Step 6 : When stack becomes empty, then stop. DFS traversal will be sequence of vertices in which they are visited using above steps.

- The algorithm for DFS can be outlined as follows:

1. for V = 1 to n do (Recursive)
   visited[V] = 0 {unvisited}
2. i = 1 {start at vertex 1)
3. DFS (i)
   begin
   visited[i] = 1
   display vertex i
   for each vertex j adjacent to i do
   if (visited[j] = 0)
   then DFS(j)
   end.

- Non-recursive DFS can be implemented by using stack for pushing all unvisited vertices adjacent to the one being visited and popping the stack to find the next unvisited vertex.

Algorithm for DFS (Non-recursive) using Stack:

1. Push start vertex onto STACK
2. While (not empty (STACK)) do
   begin
   v = POP (STACK)
   if (not visited (v))

   begin
   visited[v] = 1
   display vertex i
   push anyone adjacent vertex x of v onto STACK which is not visted
   end
   end
3. Stop.

Pseudo 'C' Code for Recursive DFS:

```
int visited[MAX]={0};
DFS (int i)
{    int k;
     visited[i] = 1;
     printf(" %d",i);
     for(k=0;k<n;k++)
        if (A[i,k]==1  && visited[k]==0)
            DFS (k);
}
```

- Let us consider graph 3.29 drawn below:

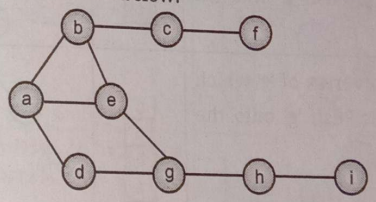


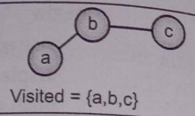
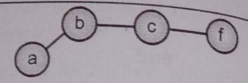
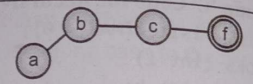
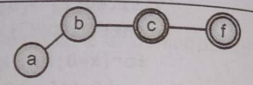
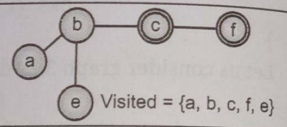
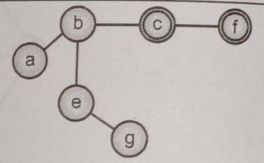
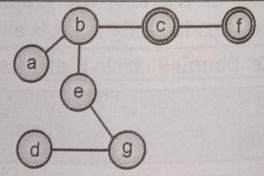
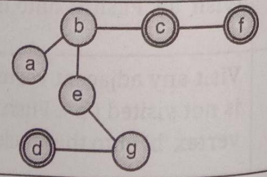
Fig. 3.29: Graph

- Let us traverse the graph using DFS algorithm which uses stack. Let 'a' be a start vertex. Initially stack is empty.

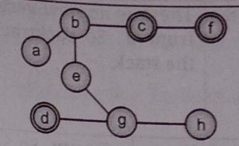
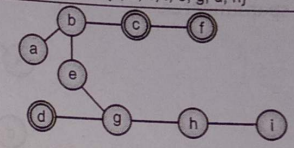
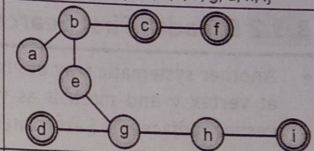
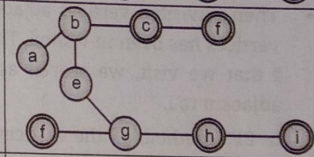
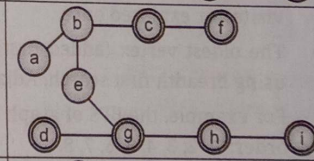
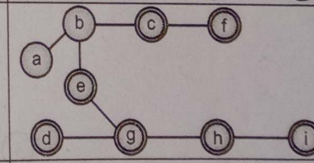
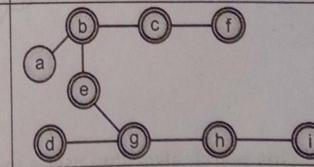
Note: Doubled circle indicates backtracking point.

Sr. No.	Steps	Stack	Spanning Tree
1.	Select vertex 'a' as starting point (visit 'a'). Push 'a' onto the stack	a	a Visited = {a}
2.	Visit any adjacent vertex of 'a' which is not visited ('b'). Push newly visited vertex 'b' onto the stack.	b a	b a Visited = {a,b}

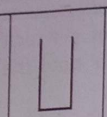
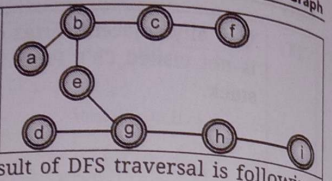
contd. ...

3.	Visit any adjacent vertex of 'b' which is not visited ('c'). Push 'c' onto the stack.	c b a	 Visited = {a,b,c}
4.	Visit any adjacent vertex of 'c' which is not visited ('f'). Push 'f' onto the stack.	f c b a	 Visited ={a,b,c,f }
5.	There is no new vertex to be visited from 'f'. So backtrack. Pop 'f' from the stack.	c b a	
6.	There is no new vertex to be visited form 'c'. So backtrack. Pop 'c' from the stack.	b a	
7.	Visit any adjacent vertex of 'b' which is not visited ("e"). Push 'e' onto the stack.	e b a	 Visited = {a, b, c, f, e}
8.	Visit any adjacent vertex of 'e' which is not visited ('g'). Push 'g' onto the stack.	g e b a	 Visited = {a, b, c, f, e, g}
9.	Visit any adjacent vertex of 'g' which is not visited ('d'). Push 'd' onto the stack.	d g e b a	 Visited = {a, b, c, f, e, g, d}
10.	There is no new vertex to be visited from 'd'. So backtrack. Pop 'd' from the stack.	g e b a	

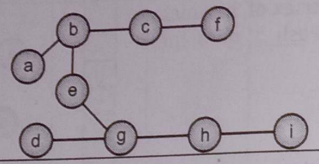
contd. ...

11.	Visit any adjacent vertex of 'g' which is not visited ('h'). Push 'h' onto the stack.	h g e b a	 Visited = {a, b, c, f, e, g, d, h}
12.	Visit any adjacent vertex of 'h' which is not visited ('i'). Push 'i' onto the stack.	i h g e b a	 Visited = {a, b, c, f, e, g, d, h, i}
13.	There is no new vertex to be visited from 'i'. So backtrack pop 'i' from the stack.	h g e b a	
14.	There is no new vertex to be visited from 'h'. So backtrack. Pop 'h' from the stack.	g e b a	
15.	There is no new vertex to be visited from 'g'. So backtrack. Pop 'g' from the stack.	e b a	
16.	There is no new vertex to be visited from 'e'. So backtrack. Pop 'e' from the stack.	b a	
17.	There is no new vertex to be visited from 'b'. So backtrack. Pop 'b' from the stack.	a	

contd. ...

18.	There is no new vertex to be visited from 'a'. So backtrack. Pop 'a' from the stack.		

Stack becomes Empty. So Stop DFS traversal, final result of DFS traversal is following spanning tree.



3.3.2 Breadth First Search (BFS)

[April 16, 17, 18, 19 Oct. 17, 18]

- Another systematic way of visiting the vertices is Breadth-First Search (BFS). Starting at vertex v and mark it as visited, the BFS differs from DFS, in that all unvisited vertices adjacent to i are visited next.
- Then unvisited vertices adjacent to these vertices are visited and so on until the all vertices has been in visited. The approach is called 'breadth first' because from vertex i that we visit, we search as broadly as possible by next visiting all the vertices adjacent to i.
- In BFS method, all the vertices are stored in a Queue and each vertex of the graph is visited or explored once.
- The oldest vertex (added first) in the Queue is explored first. To traverse the graph using breadth first search, Adjacent List of a graph is created first.
- For example, the BFS of graph of Fig. 3.30 results in visiting the nodes in the following order: 1, 2, 3, 4, 5, 6, 7, 8.

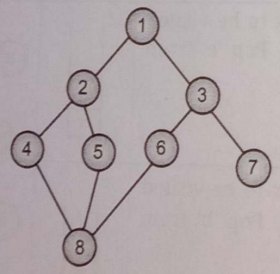


Fig. 3.30

- This search algorithm uses a queue to store the adjacent vertices of each vertex of the graph as and when it is visited.

- These vertices are then taken out from queue in a sequence (FIFO) and their adjacent vertices are visited and so on until all the vertices have been visited. The algorithm terminates when the queue is empty.
- The algorithm for BFS is given below. The algorithm initializes the Boolean array visited[ ] to 0 (false) i.e. marks each vertex as unvisited.

```
for i = 1 to n do
    visited[i] = 0.
```

Algorithm:

```
BFS (i)
begin
    visited[i] = 1
    add (Queue, i)
        while not empty (Queue) do
            begin
                i = delete (Queue)
                for all vertices j adjacent to i do
                    begin
                    if (visited[j] = 0)
                        add (Queue, j)
                        visited[j] = 1
                end
        end
end
```

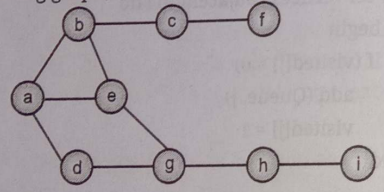
- Here, while loop is executed n time as n is the number of vertices and each vertex is inserted in a queue once. If adjacency list representation is used, then adjacent nodes are computed in for loop.

Pseudo 'C' code for recursive BFS:

```
BFS (int i)
      {
         int k,visited[MAX];
         struct queue Q;
         for(k=0;k<n;k++)
         visited[k] = 0;
         visited[i] = 1;
         printf(" %d",i);
         insert(Q,i);
```

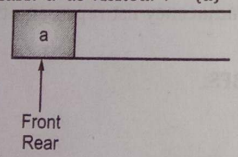
```
while (!Empty (Q))
{
  j = delete(Q);
  for(k=0;k<n;k++)
  {
    if (A[j,k]&&!visited[k])
    {
      insert(Q,k);
      visited[k] = 1;
    printf(" %d",k);
    }
  }
}
}
```

- Let us consider following graph,



- Let us traverse the graph using non-recursive algorithm which uses queue. Let 'a' be a start vertex. Initially queue is empty. Initial set of visited vertices, V = ϕ.

(i) Add 'a' to the queue. Mark 'a' as visited. V = {a}

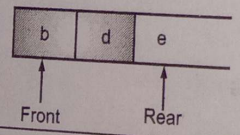


(ii) As queue is not empty,

vertex = delete( ) = 'a'.

Add all unvisited adjacent vertices of 'a' to the queue. Also mark them visited.

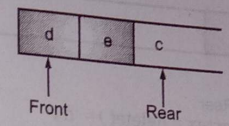
V = {a, b, d, e}



(iii) As queue is not empty,

Vertex = delete( ) = 'b'. Insert all adjacent, unvisited vertices of 'b' to the queue.

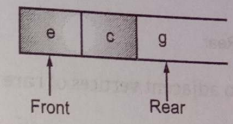
V = {a, b, d, e, c}



(iv) As queue is not empty,

Vertex = delete( ) = 'd'. Now insert all adjacent, unvisited vertices of 'd' to the queue and mark them as visited

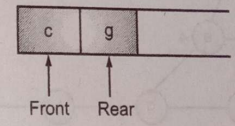
V = {a, b, d, e, c, g}



(v) As queue is not empty,

Vertex = delete( ) = 'e'. There are no unvisited adjacent vertices of 'e'. The queue is as:

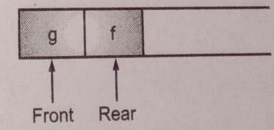
V = {a, b, d, e, c, g}.



(vi) As queue is not empty, vertex = delete( ) = 'c'.

Insert all adjacent, unvisited vertices of 'c' to the queue and mark them visited.

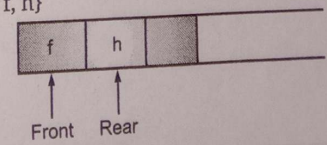
V = {a, b, d, e, c, g, f}



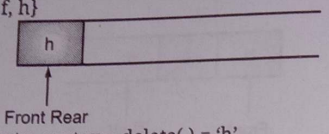
(vii) As queue is not empty, vertex = delete( ) = 'g'.

Insert all unvisited adjacent vertices of 'g' to the queue and mark them visited.

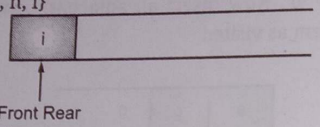
V = {a, b, d, e, c, g, f, h}



(viii) As queue is not empty, vertex = delete( ) = 'f'.
There are no unvisited adjacent vertices of 'f'. The queue is as:
V = {a, b, d, e, c, g, f, h}



(ix) As queue is not empty, vertex = delete( ) = 'h'.
Insert its unvisited adjacent vertices to queue and Mark them.
V = {a, b, d, e, c, g, f, h, i}



(x) As queue is not empty,
Vertex = delete( ) = 'i', No adjacent vertices of i are unvisited.

(xi) As queue is empty, stop.
The sequence in which vertices are visited by BFS is as:
a, b, d, e, c, g, f, h, i.

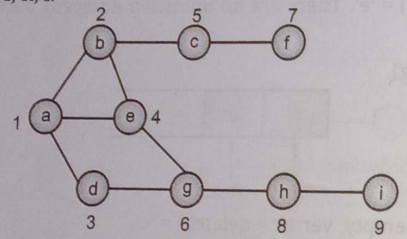


Fig. 3.31

Program 3.2: To create a graph and represent using adjacency matrix and adjacency list and traverse in BFS order.

```
#include<stdio.h>
#include<stdlib.h>
struct q
{
   int data[20];
   int front, rear;
} q1;
struct node
{
   int vertex;
   struct node * next;
} * v[10];
```



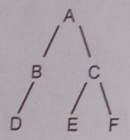
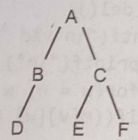
```
void add (int n)
{
   q1.rear++;
   q1.data[q1.rear]=n;
}
int del()
{
   q1.front++;
   return q1.data[q1.front];
}
void initq()
{
   q1.front = q1.rear = - 1;
}
int emptyq()
{
   return (q1.rear == q1.front);
}
void create(int m[10][10], int n)
{
   int i, j;
   char ans;
   for (i=0; i<n; i++)
   for (j=0; j<n; j++)
   {
      m[i][i] = 0;
      if(i != j)
      {
         printf("\n \t Is there an edge between %d and %d:", i+1, j+1);
         scanf("%d", &m[i][j]);
      }
   }
} /* end of create */
void disp(int m[10][10], int n)
{
   int i, j;
   printf("\n \t the adjacency matrix is: \n");
```

```
        for(i=0; i<n; i++)
        {
           for (j=0; j<n; j++)
           printf("%5d", m[i][j]);
           printf("\n");
        }
  }  /* end of display */
  void create1 (int m[10][10], int n)
  {
     int i, j;
     struct node *temp, *newnode;
     for(i=0; i<n; i++)
     {
        v[i] = NULL;
        for(j=0; j<n; j++)
        {
           if(m[i][j] == 1)
           {
                 newnode=(struct node *) malloc(sizeof(struct node));
                 newnode -> next = NULL;
                 newnode -> vertex = j+1;
                 if(v[i]==NULL)
                 v[i] = temp = newnode;
                 else
                 {
                       temp -> next = newnode;
                       temp = newnode;
                 }
           }
        }
     }
  }
void displist(int n)
{
   struct node *temp;
   int i;
   for (i=0; i<n; i++)
   {
      printf("\nv%d | ", i+1);
         temp = v[i];
```



```
         while (temp)
         {
               printf("v%d -> ", temp -> vertex);
               temp = temp -> next;
         }
         printf("null");
   }
}
void bfs(int m[10][10], int n)  //create adjacency list
{
   int i, j, v, w;
   int visited[20];
   initq();
   for(i=0; i<n; i++)
   visited[i] = 0;
   printf("\n \t The BFS traversal is: \n");
   v=0;
   visited[v] = 1;
   add(v);
   while(! emptyq())
   {
      v = del();
      printf("\n v%d ", v + 1);
         printf("\n");
         for(w = 0; w < n; w++)
         if((m[v][w] ==1) && (visited[w] == 0))
         {
               add(w);
               visited[w] = 1;
         }
   }
}
/* main program */
void main()
{
   int m[10][10], n;
   printf("\n \t enter no. of vertices");
      scanf("%d", &n);
      create(m,n);
      disp(m,n);
      create1(m,n);
      displist(n);
      bfs(m,n);
}
```

Differentiation between DFS and BFS:

Sr. No.	DFS	BSF
1.	DFS stands for Depth First Search.	BFS stands for Breadth First Search.
2.	DFS uses Stack implementation i.e. LIFO.	BFS uses Queue implementation i.e. FIFO.
3.	DFS is faster.	Slower than DFS.
4.	DFS requires less memory.	BFS uses a large amount of memory.
5.	DFS is easy to implement in a procedural language.	DFS is complex or hard to implement in a procedural language.
6.	The aim of DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node.	The aim of BFS algorithm is to traverse the graph as close as possible to the root node.
7.	Used for topological sorting.	Used for finding the shortest path between two nodes.
8.	**Example:**  A, B, D, C, E, F	**Example:**  A, B, C, D, E, F

3.4 APPLICATIONS OF GRAPH [April 18, Oct. 18]

- In this we study various operations on graph like topological sorting, minimal spanning tree, and so on.

3.4.1 Topological Sort [April 16, 19]

- A topological sort has important applications for graphs. Topological sorting is possible if and only if the graph is a directed acyclic graph.
- Topological sorting of vertices of a directed acyclic graph is an ordering of the vertices $v_1, v_2, \ldots v_n$ in such a way, that if there is an edge directed towards vertex v_i from vertex v_i, then v_i comes before v_j.
- Topological ordering is not possible if the graph has a cycle. The Fig. 3.32 shows the topological sorting.

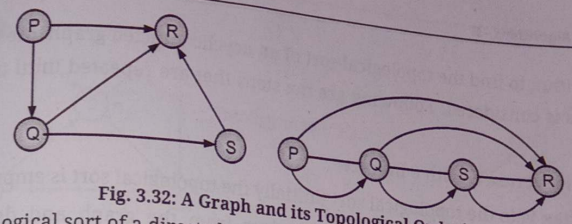


Fig. 3.32: A Graph and its Topological Ordering

- The topological sort of a directed acyclic graph is a linear ordering of the vertices, such that if there exists a path from vertex x to y, then x appears before y in the topological sort.
- Formally, for a directed acyclic graph G = (V, E), where $V = (V_1, V_2, V_3, \ldots, V_n)$, if there exists a path from any V_i to V_j, the V_i appears before V_j in the topological sort.
- An acyclic directed graph and have more than one topological sorts. For example, two different topological sorts for the graph shown in Fig. 3.33 are (1, 4, 2, 3) and (1, 2, 4, 3).

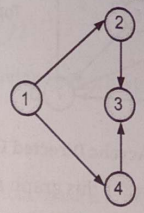


Fig. 3.33: Acyclic Directed Graph

- Clearly, if a directed graph contains a cycle, topological ordering of vertices is not possible. It is because for any two vertices V_i and V_i in the cycle, V_i precedes V_i as well as V_i precedes V_i. To exemplify this, consider a simple cyclic directed graph shown in Fig. 3.34.
- The topological sort for this graph is (1, 2, 3, 4) (assuming the vertex 1 as starting vertex). Now, since there exists a path from the vertex 4 to 1, then according to the definition of topological sort, the vertex 4 must appear before the vertex 1, which contradicts the topological sort generated for this graph. Hence, topological sort can exist only for the acyclic graph.

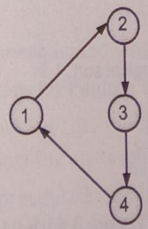


Fig. 3.34: Cyclic Directed Graph

- In an algorithm to find the topological sort of an acyclic directed graph, the indegree of the vertices is considered. Following are the steps that are repeated until the graph is empty.
  1. Select any vertex V_i with 0 indegree.
  2. Add vertex V_i to the topological sort (initially the topological sort is empty).
  3. Remove the vertex V_i along with its edges from the graph and decrease the indegree of each adjacent vertex of V_i by one.
- To illustrate this algorithm, consider an acyclic directed graph shown in Fig. 3.35.

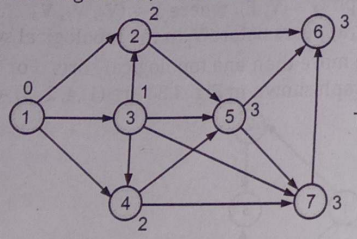
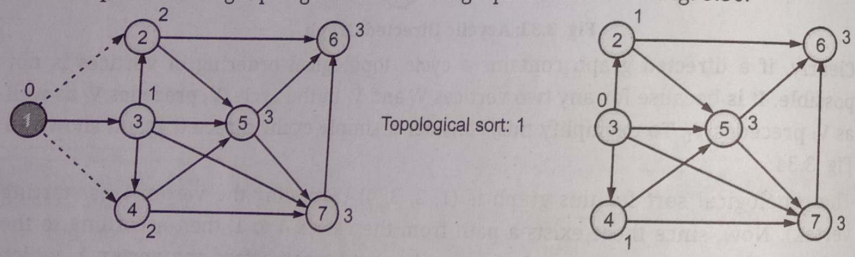
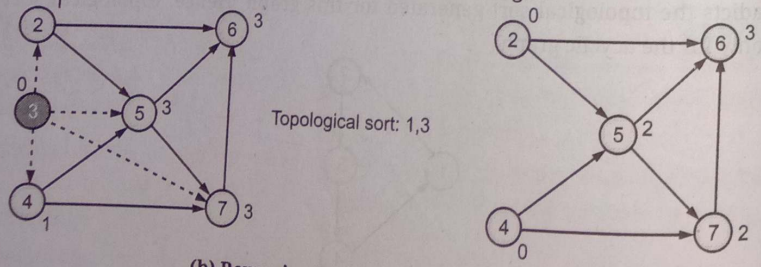


Fig. 3.35: Acyclic Directed Graph

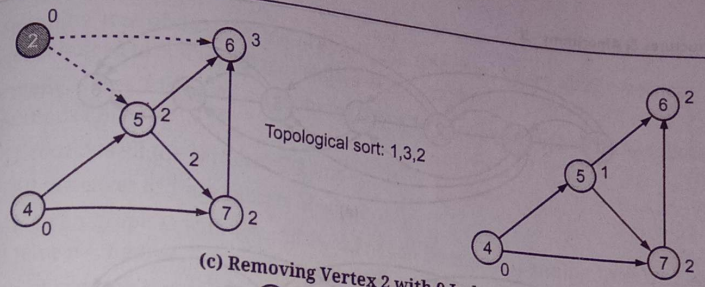
- The steps for finding topological sort for this graph are shown in Fig. 3.36.



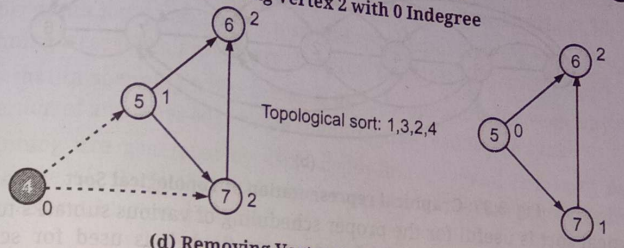
(a) Removing Vertex 1 with 0 Indegree



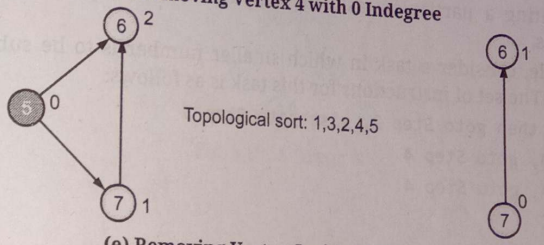
(b) Removing Vertex 3 with 0 Indegree



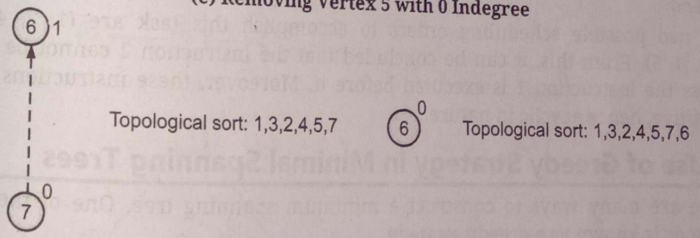
(c) Removing Vertex 2 with 0 Indegree



(d) Removing Vertex 4 with 0 Indegree



(e) Removing Vertex 5 with 0 Indegree



(f) Removing node 7 with 0 Indegree

Fig. 3.36: Steps for Finding Topological Sort

- Another possible topological sort for this graph is (1, 3, 4, 2, 5, 7, 6). Hence, it can be concluded that the topological sort for an acyclic graph is not unique.
- Topological ordering can be represented graphically. In this representation, edges are also included to justify the ordering of vertices (See Fig. 3.37).

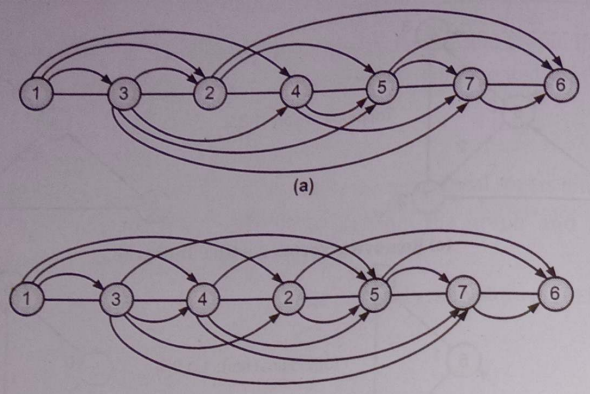


Fig. 3.37: Graphical representation of Topological Sort

- Topological sort is useful for the proper scheduling of various subtasks to be executed for completing a particular task. In computer field, it is used for scheduling the instructions.
- For example, consider a task in which smaller number is to be subtracted from the larger one. The set of instructions for this task is as follows:

```
1. if A>B then goto Step 2, else goto Step 3
2. C = A-B, goto Step 4
3. C = B-A, goto Step 4
4. Print C
5. End
```

- The two possible scheduling orders to accomplish this task are (1, 2, 4, 5) and (1, 3, 4, 5). From this, it can be concluded that the instruction 2 cannot be executed unless the instruction 1 is executed before it. Moreover, these instructions are non-repetitive, hence acyclic in nature.

3.4.2 Use of Greedy Strategy in Minimal Spanning Trees

- There are many ways to construct a minimum spanning tree. One of the simplest methods is known as a greedy strategy.
- Minimum Spanning Tree (MST) algorithms are a classic example of the greedy method.
- Greedy strategy is used to solve many problems, such as finding the minimal spanning tree in a graph using Prim's /Kruskal's algorithm.

Spanning Tree:

- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges.

- A spanning tree of a connected graph G is a tree that covers all the vertices and the edges required to connect those vertices in the graph.
- Formally, a tree T is called a spanning tree of a connected graph G if the following two conditions hold:
  1. T contains all the vertices of G, and
  2. All the edges of T are subsets of edges of G.
- For a given graph G with n vertices, there can be many spanning trees and each tree will have n – 1 edges. For example, consider a graph as shown in Fig. 3.38.
- Since, this graph has 4 vertices, each spanning tree must have 4 – 1 = 3 edges. Some of the spanning trees for this graph are shown in Fig. 3.39.
- Observe that in spanning trees, there exists only one path between any two vertices and insertion of any other edge in the spanning tree results in a cycle.
- The spanning tree generated by using depth-first traversal is known as depth-first spanning tree. Similarly, the spanning tree generated by using breadth-first traversal is known as breadth-first spanning tree.

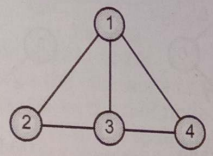


Fig. 3.38: A Graph G

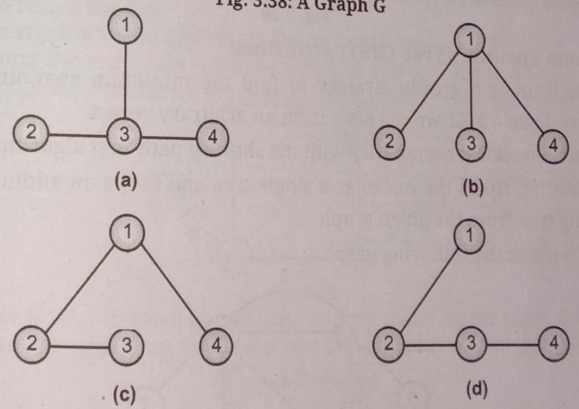
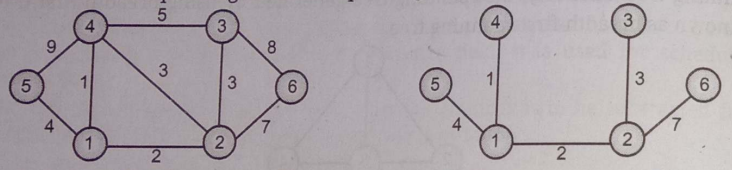


Fig. 3.39: Spanning Trees of Graph G

- For a connected weighted graph G, it is required to construct a spanning tree T such that the sum of weights of the edges in T must be minimum. Such a tree is called a minimum spanning tree.

- There are various approaches for constructing a minimum spanning tree out of which Kruskal's algorithm and Prim's algorithm are commonly used.
- If each edge of the graph is associated with a weight and there exists more than one spanning tree, we need to find the minimum spanning tree of the graph.
- A Minimum Spanning Tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.
- A minimum spanning tree in an undirected connected weighted graph is a spanning tree of minimum weight (among all spanning trees).
- In a minimum spanning tree of a graph, the maximum weight of an edge is the minimum possible from all possible spanning trees of that graph.
- In a minimum spanning tree of a graph, the maximum weight of an edge is the minimum possible from all possible spanning trees of that graph.
- In the left image you can see a weighted undirected graph, and in the right image you can see the corresponding minimum spanning tree.



(a) Weighted Undirected Graph **(b) Minimum Spanning Tree**

Fig. 3.40

Prim's Minimum Spanning Tree (MST) Algorithm:

- Prim's algorithm is a greedy strategy to find the minimum spanning tree. In this algorithm, to form a MST we can start from an arbitrary vertex.
- Prim's algorithm shares a similarity with the shortest path first algorithms.
- Prim's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.
- Example: Consider the following graph.

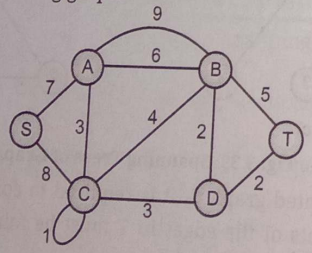


Fig. 3.41

Step 1: Remove all loops and parallel edges:

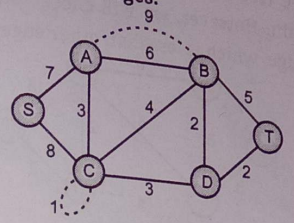


Fig. 3.42

- Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

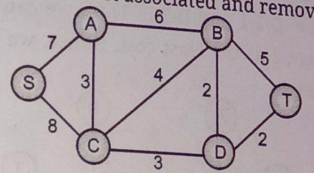


Fig. 3.43

Step 2: Choose any arbitrary node as root node:

- In this case, we choose S node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3: Check outgoing edges and select the one with less cost:

- After choosing the root node S, we see that S, A and S, C are two edges with weight 7 and 8, respectively. We choose the edge S, A as it is lesser than the other.

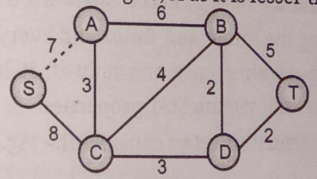


Fig. 3.44

- Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.

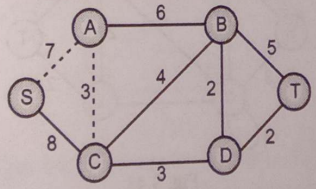


Fig. 3.45

- After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.

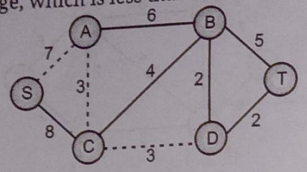


Fig. 3.46

- After adding node D to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.

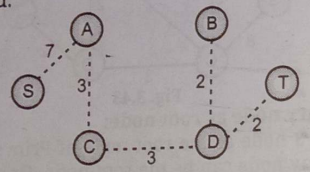


Fig. 3.47

- We may find that the output spanning tree of the same graph using two different algorithms is same.

Kruskal's Minimum Spanning Tree (MST) Algorithm:

- Kruskal's algorithm to find the minimum cost spanning tree uses the greedy strategy.
- Kruskal's algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.
- To understand Kruskal's algorithm let us consider the Fig. 3.48.

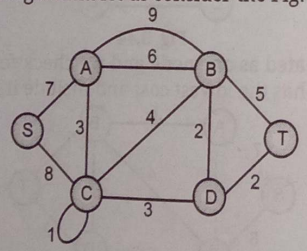


Fig. 3.48

Step 1: Remove all Loops and Parallel Edges:

- Remove all loops and parallel edges from the given graph.

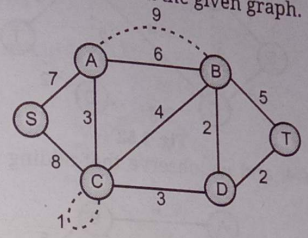


Fig. 3.49

- In case of parallel edges, keep the one which has the least cost associated and remove all others.

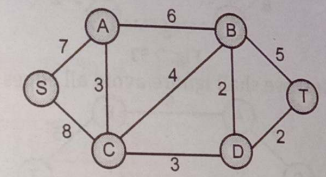


Fig. 3.50

Step 2: Arrange all edges in their increasing order of weight:

- The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

Step 3: Add the edge which has the least weightage:

- Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

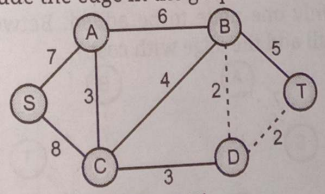


Fig. 3.51

- The least cost is 2 and edges involved are B, D and D, T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

- Next cost is 3, and associated edges are A, C and C, D. We add them again:

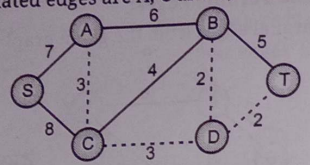


Fig. 3.52

- Next cost in the table is 4, and we observe that adding it will create a circuit in the graph.

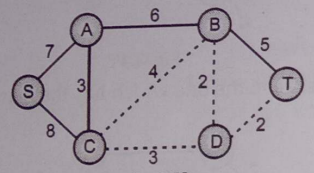


Fig. 3.53

- We ignore it. In the process we shall ignore/avoid all edges that create a circuit.

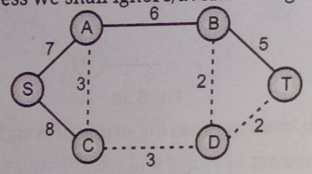


Fig. 3.54

- We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.

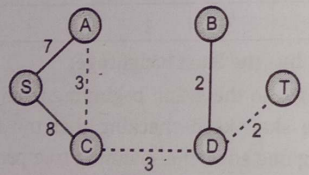


Fig. 3.55

- Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.

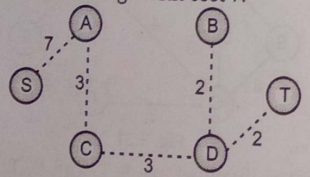


Fig. 3.56

- By adding edge S, A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Example: Consider the graph in Fig. 3.57.

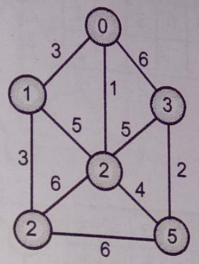
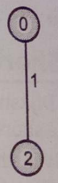


Fig. 3.57

- Procedure for finding Minimum Spanning Tree:

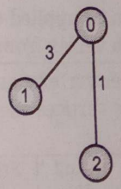
Step 1:

No. of Nodes	0	1	2	3	4	5
Distance	0	3	**1**	6	∞	∞
Distance From		0	0	0		



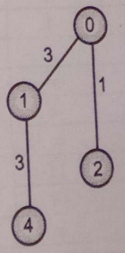
Step 2:

No. of Nodes	0	1	2	3	4	5
Distance	0	**3**	0	5	6	4
Distance From		0		2	2	2



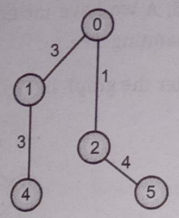
Step 3:

No. of Nodes	0	1	2	3	4	5
Distance	0	0	0	5	**3**	4
Distance From				2	1	2



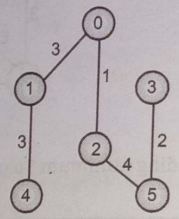
Step 4:

No. of Nodes	0	1	2	3	4	5
Distance	0	0	0	5	0	4
Distance From				2		2



Step 5:

No. of Nodes	0	1	2	3	4	5
Distance	0	0	0	3	0	0
Distance From				2		2



Minimum Cost = 1+2+3+3+4 = 13.

3.4.3 Single Source Shortest Path (Dijkstra's Algorithm)

- Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm)[2] is an algorithm for finding the shortest paths between nodes in a graph.
- Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph G = (V, E), where all the edges are non-negative (i.e., w(u, v) ≥ 0 for each edge (u, v) ∈ E).
- Dijkstra's Algorithm can be applied to either a directed or an undirected graph to find the shortest path to each vertex from a single source.

Example: Let us consider vertex 1 and 9 as the start and destination vertex respectively. Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by 0.

Vertex	Initial	Step1 V_1	Step2 V_3	Step3 V_2	Step4 V_4	Step5 V_5	Step6 V_7	Step7 V_8	Step8 V_6
1	0	0	0	0	0	0	0	0	0
2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2
4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11
8	∞	∞	∞	∞	∞	16	13	13	13
9	∞	∞	∞	∞	∞	∞	∞	∞	20

Hence, the minimum distance of vertex 9 from vertex 1 is 20. And the path is
1 → 3 → 7 → 8 → 6 → 9
This path is determined based on predecessor information.

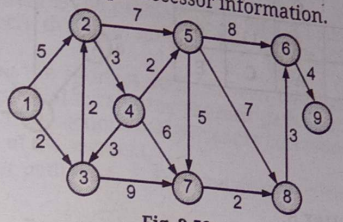


Fig. 3.58

Example: Consider the graph in Fig. 3.59.

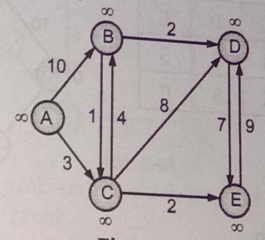
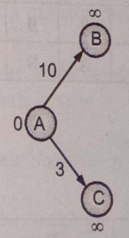


Fig. 3.59

Procedure for Dijkstra's Algorithm:

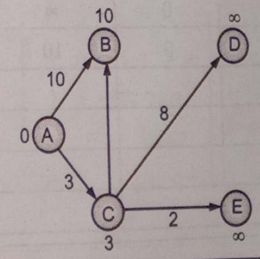
Step 1: Consider A as source Vertex.

No. of Nodes	A	B	C	D	E
Distance	0	10	3	∞	∞
Distance From		A	A		



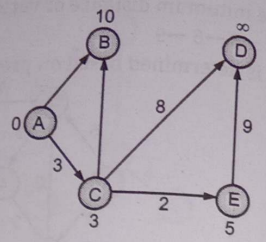
Step 2: Now consider vertex C.

No. of Nodes	A	B	C	D	E
Distance	0	7	3	11	5
Distance From		C	C	C	C



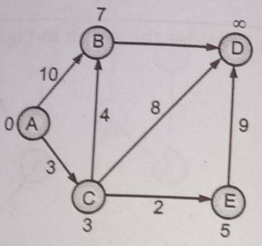
Step 3 : Now consider vertex E.

No. of Nodes	A	B	C	D	E
Distance	0	7	3	11	0
Distance From		C	A	C	E



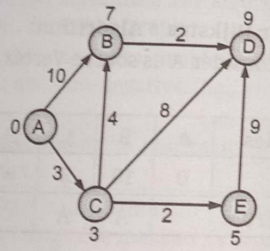
Step 4 : Now consider vertex B.

No. of Nodes	A	B	C	D	E
Distance	0	7	3	9	5
Distance From		C	A	B	C



Step 5 : Now consider vertex D.

No. of Nodes	A	B	C	D	E
Distance	0	7	3	9	11
Distance From	A	C	A	B	C



Therefore,

	A	B	C	D	E
	0	∞	∞	∞	∞
A	0	10	3	∞	∞
C		7	3	11	5
E				14	5
B				9	
D					16

3.4.4 Dynamic Programming Strategy

- The all pairs shortest paths algorithm of Floyd and Warshall uses a dynamic programming strategy.
- The shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.
- The problem of finding the shortest path between two intersections on a road map may be modeled as a special case of the shortest path problem in graphs, where the vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of the segment.
- Fig. 3.60 shows shortest path (A, C, E, D, F) between vertices A and F in the weighted directed graph.

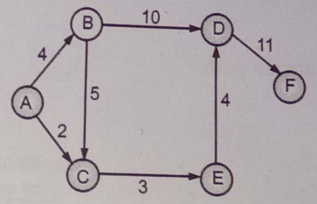
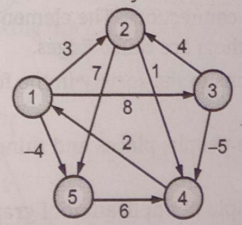


Fig. 3.60

- The all-pairs shortest path problem, in which we have to find shortest paths between every pair of vertices in the graph.
- The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph.
- The Floyd-Warshall algorithm is used for finding the shortest path between every pair of vertices in the graph.
- This algorithm works for both directed as well as undirected graphs. This algorithm is invented by Robert Floyd and Stephen Warshall hence it is often called as Floyed - Warshall algorithm.
- Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph.
- As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.



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

Fig. 3.61

- At first, the output matrix is the same as the given cost matrix of the graph. After that, the output matrix will be updated with all vertices k as the intermediate vertex.

Input and Output:

Input: The cost matrix of the graph.

0 3 6 ∞ ∞ ∞ ∞
3 0 2 1 ∞ ∞ ∞
6 2 0 1 4 2 ∞
∞ 1 1 0 2 ∞ 4
∞ ∞ 4 2 0 2 1
∞ ∞ 2 ∞ 2 0 1
∞ ∞ ∞ 4 1 1 0

Output: Matrix of all pair shortest path.

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

3.4.5 Use of Graphs in Social Networks

- A graph is made up of nodes; just like that a social media is a kind of a social network, where each person or organization represents a node.
- A graph is made up of nodes; just like that a social media is a kind of a social network, where each person or organization represents a node.
- In World Wide Web (WWW), web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of Directed graph.
- Graphs are awesome data structures that you use every day through Google Search, Google Maps, GPS, and social media.
- They are used to represent elements that share connections. The elements in the graph are called Nodes and the connections between them are called Edges.
- When we need to represent any form of relations in the society in the form of links, it can be termed as Social Network.
- Social graphs draw edges between you and the people, places and things you interact with online.
- Facebook's Graph API is perhaps the best example of application of graphs to real life problems. The Graph API is a revolution in large-scale data provision.

- On The Graph API, everything is a vertice or node. This are entities such as Users, Pages, Places, Groups, Comments, Photos, Photo Albums, Stories, Videos, Notes, Events and so forth. Anything that has properties that store data is a vertice.
- The Graph API uses this, collections of vertices and edges (essentially graph data structures) to store its data.
- The Graph API has come into some problems because of it's ability to obtain unusually rich info about user's friends.

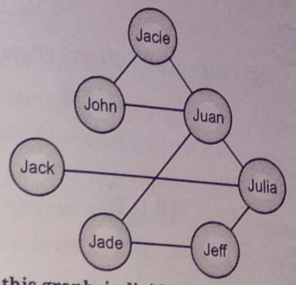
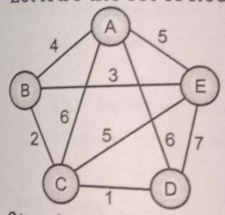


Fig. 3.62: A sample Graph (in this graph, individuals are represented with nodes (circles) and individuals who know each other are connected with edges (lines))

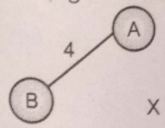
Example: Define spanning tree and minimum spanning tree. Find the minimum spanning tree of the graph shown in Fig. 3.62.

Using Prim's Algorithm:

Let X be the set of nodes explored, initially X = {A}.

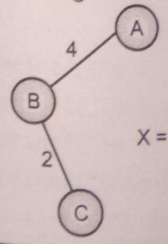


Step 1 : Taking minimum weight edge of all Adjacent edges of X = {A}.



X = {A, B}

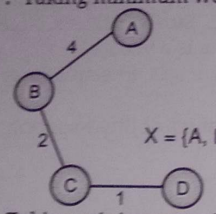
Step 2 : Taking minimum weight edge of all Adjacent edges of X = {A, B}.



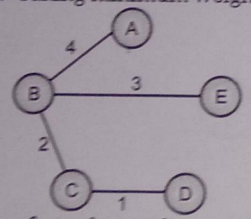
X = {A, B, C}

~~A-B|4~~
A-E|5
A-C|6
A-D|6
~~B-E|3~~
~~B-C|2~~
C-E|6
~~C-D|1~~
D-E|7

Step 3 : Taking minimum weight edge of all Adjacent edges of X = {A, B, C}.



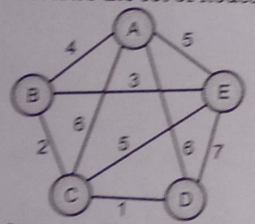
Step 4 : Taking minimum weight edge of all Adjacent edges of X = {A, B, C, D}.



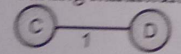
All nodes of graph are there with set X, so we obtained minimum spanning tree of cost: 4 + 2 + 1 + 3 = 10.

Using Kruskal's Algorithm:

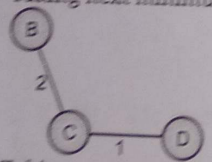
- Let X be the set of nodes explored, initially X = (A).



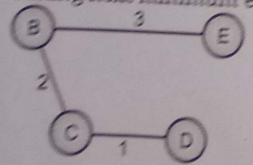
Step 1 : Taking minimum edge (C, D).



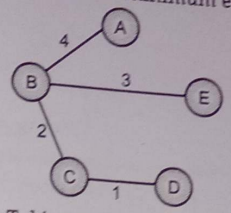
Step 2 : Taking next minimum edge (B, C).



Step 3 : Taking next minimum edge (B, E).



Step 4 : Taking next minimum edge (A, B).



Step 5 : Taking next minimum edge (A, E) it forms cycle so do not consider.

Step 6 : Taking next minimum edge (C, E) it forms cycle so do not consider.

Step 7 : Taking next minimum edge (A, D) it forms cycle so do not consider.

Step 8 : Taking next minimum edge (A, C) it forms cycle so do not consider.

Step 9 : Taking next minimum edge (E, D) it forms cycle so do not consider.

All edges of graph have been visited, so we obtained minimum spanning tree of cost: 4 + 2 + 1 + 3 = 10.

Program 3.3: Program that accepts the vertices and edges of a graph. Create adjacency list and display the adjacency list.

```
#include <stdio.h>
#include <stdlib.h>
#define new_node (struct node*)malloc(sizeof(struct node))
struct node
{
   int vertex;
   struct node *next;
};
void main()
{
   int choice;
   do
   {
      printf("\n A Program to represent a Graph by using an
                                                Adjacency List \n ");
      printf("\n 1. Directed Graph ");
      printf("\n 2. Un-Directed Graph ");
      printf("\n 3. Exit ");
      printf("\n\n Select a proper choice : ");
      scanf("%d", &choice);
```

```
        switch(choice)
        {
           case 1 : dir_graph();
                    break;
           case 2 : undir_graph();
                    break;
           case 3 : exit(0);
        }
    }while(1);
}
int dir_graph()
{
   struct node *adj_list[10], *p;
   int n;
   int in_deg, out_deg, i, j;
   printf("\n How Many Vertices ? : ");
   scanf("%d", &n);
   for( i = 1 ; i <= n ; i++ )
       adj_list[i] = NULL;
  read_graph (adj_list, n);
  printf("\n Vertex \t In_Degree \t Out_Degree \t Total_Degree ");
  for (i = 1; i <= n ; i++ )
  {
     in_deg = out_deg = 0;
     p = adj_list[i];
     while( p != NULL )
     {
        out_deg++;
        p = p -> next;
     }
     for ( j = 1 ; j <= n ; j++ )
     {
        p = adj_list[j];
        while( p != NULL )
        {
           if ( p -> vertex == i )
              in_deg++;
           p = p -> next;
        }
     }
```



```
     printf("\n\n %5d\t\t\t%d\t\t%d\t\t%d\n\n", i, in_deg,
                                                out_deg, in_deg + out_deg);
  }
  return;
}
int undir_graph()
{
   struct node *adj_list[10], *p;
   int deg, i, j, n;
   printf("\n How Many Vertices ? : ");
   scanf("%d", &n);
   for ( i = 1 ; i <= n ; i++ )
       adj_list[i] = NULL;
   read_graph(adj_list, n);
   printf("\n Vertex \t Degree ");
   for ( i = 1 ; i <= n ; i++ )
   {
      deg = 0;
      p = adj_list[i];
      while( p != NULL )
      {
         deg++;
         p = p -> next;
      }
      printf("\n\n %5d \t\t %d\n\n", i, deg);
   }
   return;
}
int read_graph ( struct node *adj_list[10], int n )
{
   int i, j;
   char reply;
   struct node *p, *c;
   for ( i = 1 ; i <= n ; i++ )
   {
      for ( j = 1 ; j <= n ; j++ )
      {
         if ( i == j )
            continue;
         printf("\n Vertices %d & %d are Adjacent ? (Y/N) :", i, j);
         scanf("%c", &reply);
```

```
            if ( reply == 'y' || reply == 'Y' )
            {
               c = new_node;
               c -> vertex = j;
               c -> next = NULL;
               if ( adj_list[i] == NULL )
                  adj_list[i] = c;
               else
               {
                  p = adj_list[i];
                  while ( p -> next != NULL )
                  p = p -> next;
                  p -> next = c;
               }
            }
         }
      }
      return;
   }
```

Output:

```
A Program to represent a Graph by using an Adjacency Matrix method
1. Directed Graph
2. Un-Directed Graph
3. Exit
Select a proper choice :
How Many Vertices ? :
Vertices 1 & 2 are Adjacent ? (Y/N) : N
Vertices 1 & 3 are Adjacent ? (Y/N) : Y
Vertices 1 & 4 are Adjacent ? (Y/N) : Y
Vertices 2 & 1 are Adjacent ? (Y/N) : Y
Vertices 2 & 3 are Adjacent ? (Y/N) : Y
Vertices 2 & 4 are Adjacent ? (Y/N) : N
Vertices 3 & 1 are Adjacent ? (Y/N) : Y
Vertices 3 & 2 are Adjacent ? (Y/N) : Y
Vertices 3 & 4 are Adjacent ? (Y/N) : Y
Vertices 4 & 1 are Adjacent ? (Y/N) : Y
Vertices 4 & 2 are Adjacent ? (Y/N) : N
Vertices 4 & 3 are Adjacent ? (Y/N) : Y
```



```
Vertex      In_Degree      Out_Degree      Total_Degree
  1             2              0               2
  2             1              2               3
  3             0              1               1
  4             1              1               2
```

PRACTICE QUESTIONS

Q. I Multiple Choice Questions:

1. Which is a non-linear data structure?
   (a) Graph (b) Array
   (c) Queue (d) Stack
2. A graph G is represented as G = (V, E) where,
   (a) V is set of vertices (b) E is set of edges
   (c) Both (a) and (b) (d) None of these
3. In which representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices.
   (a) Adjacency List (b) Adjacency Matrix
   (c) Adjacency Queue (d) Adjacency Stack
4. Each node of the graph is called a _____.
   (a) Edge (b) Path
   (c) Vertex (d) Cycle
5. Which representation of graph is based on linked lists?
   (a) Adjacency List (b) Adjacency Matrix
   (c) Both (a) and (b) (d) None of these
6. Which of a graph means visiting each of its nodes exactly once?
   (a) Insert (b) Traversal
   (c) Delete (d) Merge
7. Which is a vertex based technique for finding a shortest path in graph?
   (a) DFS (b) BST
   (c) BFS (d) None of these
8. Which usually implemented using a stack data structure?
   (a) DFS (b) BST
   (c) BFS (d) None of these
9. Which is a graph in which all the edges are uni-directional i.e. the edges point in a single direction?
   (a) Undirected (b) Directed
   (c) Cyclic (d) Acyclic

10. Which sorting involves displaying the specific order in which a sequence of vertices must be followed in a directed graph?
    (a) Cyclic (b) Acyclic
    (c) Topological (d) None of these
11. Which is a subset of an undirected graph that has all the vertices connected by minimum number of edges?
    (a) Spanning tree (b) Minimum spanning tree
    (c) Both (a) and (b) (d) None of these
12. Which is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight?
    (a) Spanning tree (b) Minimum Spanning Tree (MST)
    (c) Both (a) and (b) (d) None of these

Answers

1. (a)	2. (c)	3. (b)	4. (c)	5. (a)	6. (b)	7. (c)
8. (a)	9. (b)	10.(c)	11. (a)	12. (b)		

Q. II Fill in the Blanks:

1. Graph represented as _____.
2. A graph is _____ if the graph comprises a path that starts from a vertex and ends at the same vertex.
3. Graph is a _____ data structure.
4. Individual data element of a graph is called as _____ (also known as node). An edge is a connecting link between two vertices. Edge is also known as _____.
5. Graph traversal is a technique used for a _____ vertex in a graph.
6. We use _____ data structure with maximum size of total number of vertices in the graph to implement DFS traversal.
7. The number of edges connected directly to the node is called as _____ of node.
8. The number edges pointing _____ the node are called in-degree/in-order.
9. A graph which has set of empty edges or is containing only isolated nodes is called a _____ graph or isolated graph.
10. The _____ (or path) between two vertices is called an edge.
11. We use _____ data structure with maximum size of total number of vertices in the graph to implement BFS traversal.
12. The graph traversal is also used to decide the order of vertices is _____ in the search process.

13. The sequence of nodes that we need to follow when we have to travel from one vertex to another in a graph is called the _____.
14. An adjacency _____ is a matrix of size n x n where n is the number of vertices in the graph.
15. Topological ordering of vertices in a graph is possible only when the graph is a _____ acyclic graph.
16. A _____ (or minimum weight spanning tree) for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree.
17. The _____ algorithm is an example of an all-pairs shortest paths algorithm.
18. A graph _____ cycle is called acyclic graph.

Answers

1. G = (V, E)	2. cyclic	3. non-linear	4. Vertex, Arc
5. searching	6. Stack	7. degree	8. toward
9. Null	10. link	11. Queue	12. visited
13. path	14. matri	15. directed	
16. Minimum Spanning Tree (MST)		17. Floyd-Warshall	18. without

Q. III State True or False:

1. An acyclic graph is a graph that has no cycle.
2. A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.
3. A graph in which weights are assigned to every edge is called acyclic graph.
4. The Floyd-Warshall algorithm is for solving the all pairs shortest path problem.
5. The number edges pointing away from the node are called out-degree/out-order.
6. In a graph, if two nodes are connected by an edge then they are called adjacent nodes or neighbors.
7. Dijkstra's Algorithm can be applied to either a directed or an undirected graph to find the shortest path to each vertex from a single source.
8. The graph is a linear data structures.
9. BFS usually implemented using a queue data structure.
10. The spanning tree does not have any cycle (loops).
11. Prim's Algorithm will find the minimum spanning tree from the graph G.
12. Given an undirected, connected and weighted graph, find Minimum Spanning Tree (MST) of the graph using Kruskal's algorithm.

13. The number of edges that are connected to a particular node is called the path of the node.
14. A spanning tree of that graph is a subgraph that is a tree and connects all the vertices together.
15. Directed graph is also called as Digraph.

Answers

1. (T)	2. (T)	3. (F)	4. (T)	5. (T)	6. (T)	7. (T)	8. (T)
9. (F)	10. (T)	11. (T)	12. (T)	13. (F)	14. (T)	15. (T)	

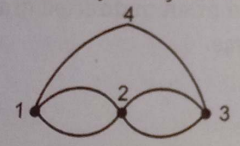
Q. IV Answer the following Questions:

(A) Short Answer Questions:

1. What is graph?
2. List traversals of graph.
3. Define the term cycle and path in graph.
4. Which data structure is used in to represent graph in adjacency list.
5. Define in-degree and out-degree of vertex.
6. What is weighted graph.
7. Define spanning tree.
8. List ways to represent a graph.
9. What are the applications of graph?
10. What is BFS and DFS.
11. Give uses of graphs in social networks.
12. Define MST?

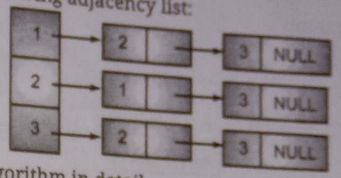
(B) Long Answer Questions:

1. Define graph. How to represent it? Explain with diagram.
2. What is topological sort. Explain with example.
3. What DFS? Describe in detail.
4. Give adjacency list representation of following graph:

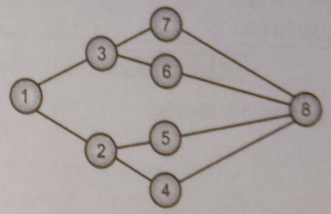


5. Write short note on: Inverse adjacency list of graph.
6. Describe following algorithms with example:
   (i) Dijkstra's algorithm
   (ii) Floyd Warshall.

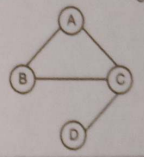
7. With the help of diagram describe adjacency matrix representation of graph.
8. What is adjacency multi-list? How to represent it? Explain with example.
9. With the help of example describe BFS.
10. Describe the term MST with example.
11. Draw graph for following adjacency list:



12. Describe Prim's algorithm in detail.
13. Explain Kruskal's algorithm with example.
14. For the following graph, give result of depth first traversal and breadth first traversal:



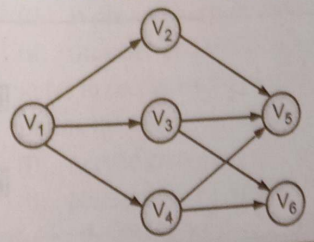
15. Give adjacency list representation of following graph.



UNIVERSITY QUESTIONS AND ANSWERS

April 2016

1. Consider the following graph: **[M]**



(i) Write adjacency matrix

(ii) Draw adjacency list

(iii) DFS and BFS traversals (start vertex v_1).

Ans. Refer to Sections 3.2.1, 3.2.2 and 3.3.

2. Define the following terms: **[2 M]**

(i) Degree of vertex

(ii) Topological sort.

Ans. Refer to Sections 3.1.2, Point (5) and 3.4.1.

April 2017

1. Which data structure is used for BFS. **[1 M]**

Ans. Refer to Section 3.3.2.

2. Define the term complete graph. **[1 M]**

Ans. Refer to Section 3.1.2, Point (3).

October 2017

1. Consider the following adjacency matrix: **[5 M]**

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

(i) Draw the graph

(ii) Draw adjacency list

(iii) Draw inverse adjacency list.

Ans. Refer to Section 3.2.1, 3.2.2 and 3.2.3.

2. Write an algorithm for BFS traversal of a graph. **[3 M]**

Ans. Refer to Section 3.3.2.

April 2018

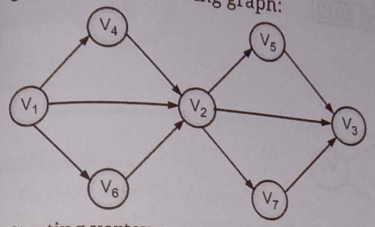
1. List any methods of representing graphs. **[1 M]**

Ans. Refer to Section 3.2.

2. List two applications of graph. **[1 M]**

Ans. Refer to Section 3.4.

3. Consider the following graph: **[5 M]**



Starting vertex v_1

(i) Draw adjacency list

(ii) Give DFS and BFS traversals.

Ans. Refer to Sections 3.2.2 and 3.3.

October 2018

1. State any two applications of graph. **[1 M]**

Ans. Refer to Section 3.4.

2. Consider the following specification of a graph G:

V(G) = {1, 2, 3, 4}

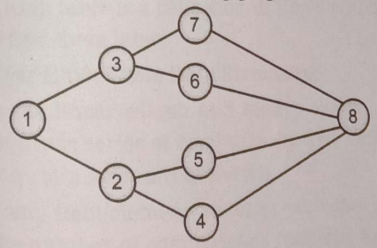
E(G) = {(1, 2),(1, 3),(3, 3),(3, 4),(4, 1)}

(i) Draw a picture of the undirected graph.

(ii) Draw adjacency matrix of lists. **[5 M]**

Ans. Refer to Sections 3.2.1 and 3.2.2.

3. Consider the following graph: **[4 M]**



(i) Write adjacency matrix

(ii) Give DFS and BFS (Source vertex v_1). **[3 M]**

Ans. Refer to Sections 3.2.1 and 3.3.

4. Define the following terms: **[2 M]**

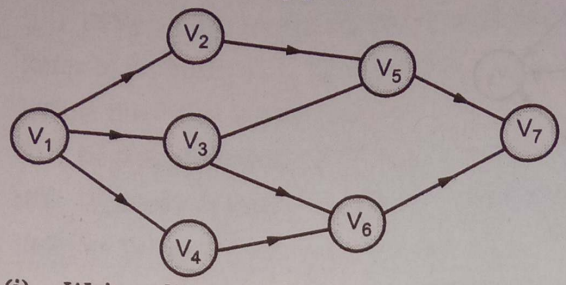
(i) Acrylic graph

(ii) Multigraph.

Ans. Refer to Section 3.1.2.

April 2019

1. Consider the following graph: [1 M]



(i) Write adjacency matrix
(ii) Draw adjacency list
(iii) DFS and BFS traversals (start vertex v_1).

Ans. Refer to Sections 3.2.1, 3.2.2 and 3.3.

2. Define the term topological sort. [1 M]

Ans. Refer to Section 3.4.1.

Syllabus ...

1. Tree **(10 Hrs.)**

1.1 Concept and Terminologies

1.2 Types of Binary Trees - Binary Tree, Skewed Tree, Strictly Binary Tree, Full Binary Tree, Complete Binary Tree, Expression Tree, Binary Search Tree, Heap

1.3 Representation - Static and Dynamic

1.4 Implementation and Operations on Binary Search Tree - Create, Insert, Delete, Search, Tree Traversals - Preorder, Inorder, Postorder (Recursive Implementation), Level-Order Traversal using Queue, Counting Leaf, Non-Leaf and Total Nodes, Copy, Mirror

1.5 Applications of Trees

1.5.1 Heap Sort, Implementation

1.5.2 Introduction to Greedy Strategy, Huffman Encoding (Implementation using Priority Queue)

2. Efficient Search Trees **(8 Hrs.)**

2.1 Terminology: Balanced Trees - AVL Trees, Red Black Tree, Splay Tree, Lexical Search Tree -Trie

2.2 AVL Tree - Concept and Rotations

2.3 Red Black Trees - Concept, Insertion and Deletion

2.4 Multi-Way Search Tree - B and B+ Tree - Insertion, Deletion

3. Graph **(12 Hrs.)**

3.1 Concept and Terminologies

3.2 Graph Representation - Adjacency Matrix, Adjacency List, Inverse Adjacency List, Adjacency Multi-list

3.3 Graph Traversals - Breadth First Search and Depth First Search (With Implementation)

3.4 Applications of Graph

3.4.1 Topological Sorting

3.4.2 Use of Greedy Strategy in Minimal Spanning Trees (Prim's and Kruskal's Algorithm)

3.4.3 Single Source Shortest Path - Dijkstra's Algorithm

3.4.4 Dynamic Programming Strategy, All Pairs Shortest Path - Floyd Warshall Algorithm

3.4.5 Use of Graphs in Social Networks

4. Hash Table **(6 Hrs.)**

4.1 Concept of Hashing

4.2 Terminologies - Hash Table, Hash Function, Bucket, Hash Address, Collision, Synonym, Overflow etc.

4.3 Properties of Good Hash Function

4.4 Hash Functions: Division Function, Mid Square, Folding Methods

4.5 Collision Resolution Techniques

4.5.1 Open Addressing - Linear Probing, Quadratic Probing, Rehashing

4.5.2 Chaining - Coalesced, Separate Chaining

■■■