

Chapter 6

Graphs

6.1. Introduction to Graphs:

Graph G is a pair (V, E) , where V is a finite set of vertices and E is a finite set of edges. We will often denote $n = |V|$, $e = |E|$.

A graph is generally displayed as figure 6.5.1, in which the vertices are represented by circles and the edges by lines.

An edge with an orientation (i.e., arrow head) is a directed edge, while an edge with no orientation is our undirected edge.

If all the edges in a graph are undirected, then the graph is an undirected graph. The graph in figure 6.5.1(a) is an undirected graph. If all the edges are directed; then the graph is a directed graph. The graph of figure 6.5.1(b) is a directed graph. A directed graph is also called as digraph. A graph G is connected if and only if there is a simple path between any two nodes in G .

A graph G is said to be complete if every node a in G is adjacent to every other node v in G . A complete graph with n nodes will have $n(n-1)/2$ edges. For example, Figure 6.5.1.(a) and figure 6.5.1.(d) are complete graphs.

A directed graph G is said to be connected, or strongly connected, if for each pair (u, v) for nodes in G there is a path from u to v and also a path from v to u . On the other hand, G is said to be unilaterally connected if for any pair (u, v) of nodes in G there is a path from u to v or a path from v to u . For example, the digraph shown in figure 6.5.1 (e) is strongly connected.

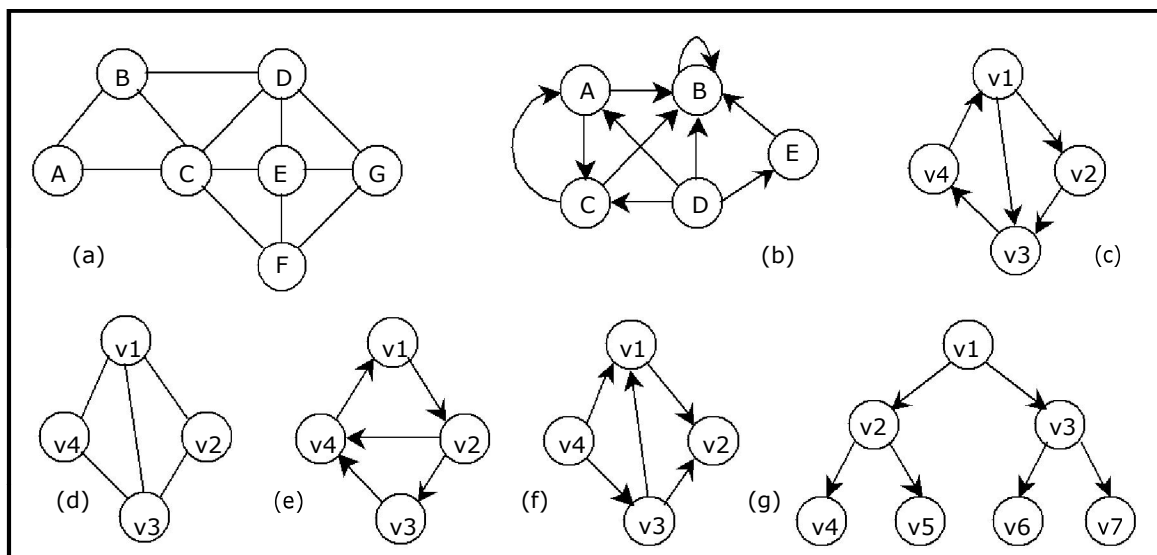


Figure 6.5.1 Various Graphs

We can assign weight function to the edges: $w_G(e)$ is a weight of edge $e \in E$. The graph which has such function assigned is called weighted graph.

The number of incoming edges to a vertex v is called in-degree of the vertex (denote $\text{indeg}(v)$). The number of outgoing edges from a vertex is called out-degree (denote $\text{outdeg}(v)$). For example, let us consider the digraph shown in figure 6.5.1(f),

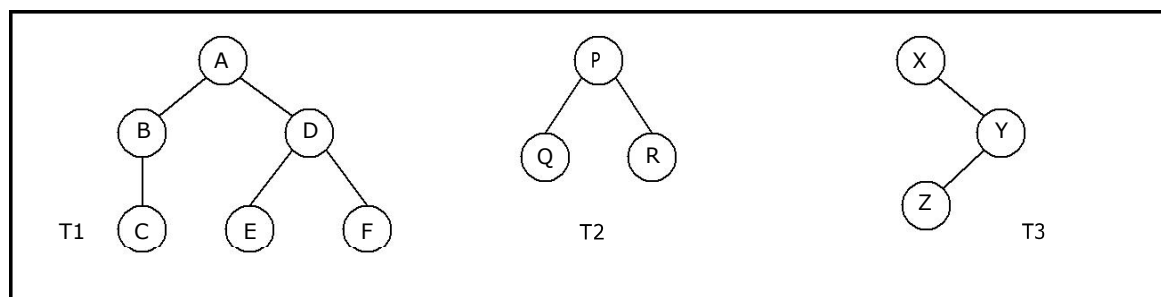
$$\begin{array}{ll} \text{indegree}(v_1) = 2 & \text{outdegree}(v_1) = 1 \\ \text{indegree}(v_2) = 2 & \text{outdegree}(v_2) = 0 \end{array}$$

A path is a sequence of vertices (v_1, v_2, \dots, v_k) , where for all i , $(v_i, v_{i+1}) \in E$. A path is simple if all vertices in the path are distinct. If there is a path containing one or more edges which starts from a vertex V_i and terminates into the same vertex then the path is known as a cycle. For example, there is a cycle in figure 6.5.1(a), figure 6.5.1(c) and figure 6.5.1(d).

If a graph (digraph) does not have any cycle then it is called **acyclic graph**. For example, the graphs of figure 6.5.1 (f) and figure 6.5.1 (g) are acyclic graphs.

A graph $G' = (V', E')$ is a sub-graph of graph $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$.

A **Forest** is a set of disjoint trees. If we remove the root node of a given tree then it becomes forest. The following figure shows a forest F that consists of three trees T_1 , T_2 and T_3 .



A Forest F

A graph that has either self loop or parallel edges or both is called **multi-graph**.

Tree is a connected acyclic graph (there aren't any sequences of edges that go around in a loop). A spanning tree of a graph $G = (V, E)$ is a tree that contains all vertices of V and is a subgraph of G . A single graph can have multiple spanning trees.

Let T be a spanning tree of a graph G . Then

1. *Any two vertices in T are connected by a unique simple path.*
2. *If any edge is removed from T , then T becomes disconnected.*
3. *If we add any edge into T , then the new graph will contain a cycle.*
4. *Number of edges in T is $n-1$.*

6.2. Representation of Graphs:

There are two ways of representing digraphs. They are:

- Adjacency matrix.
- Adjacency List.
- Incidence matrix.

Adjacency matrix:

In this representation, the adjacency matrix of a graph G is a two dimensional $n \times n$ matrix, say $A = (a_{i,j})$, where

$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{Otherwise} \end{cases}$$

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed. This matrix is also called as Boolean matrix or bit matrix.

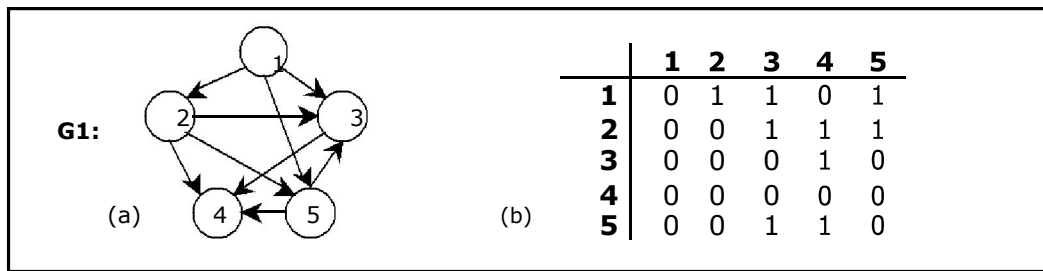


Figure 6.5.2. A graph and its Adjacency matrix

Figure 6.5.2(b) shows the adjacency matrix representation of the graph $G1$ shown in figure 6.5.2(a). The adjacency matrix is also useful to store multigraph as well as weighted graph. In case of multigraph representation, instead of entry 0 or 1, the entry will be between number of edges between two vertices.

In case of weighted graph, the entries are weights of the edges between the vertices. The adjacency matrix for a weighted graph is called as cost adjacency matrix. Figure 6.5.3(b) shows the cost adjacency matrix representation of the graph $G2$ shown in figure 6.5.3(a).

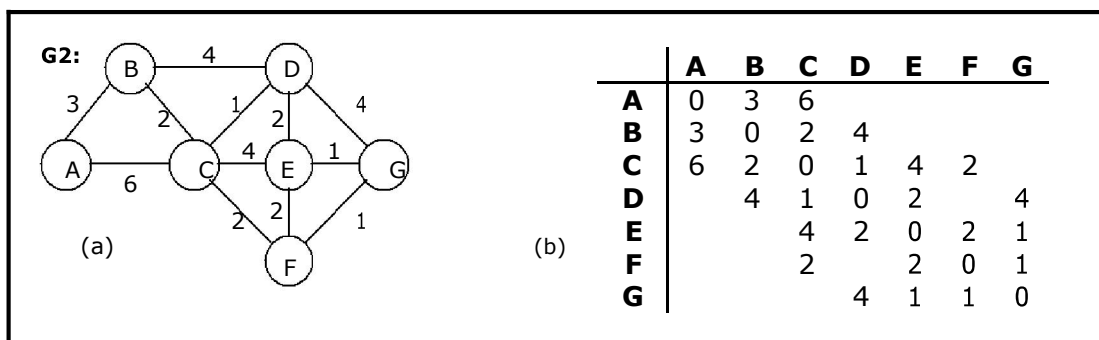


Figure 6.5.3 Weighted graph and its Cost adjacency matrix

Adjacency List:

In this representation, the n rows of the adjacency matrix are represented as n linked lists. An array $Adj[1, 2, \dots, n]$ of pointers where for $1 \leq v \leq n$, $Adj[v]$ points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements. For the graph G in figure 6.5.4(a), the adjacency list is shown in figure 6.5.4 (b).

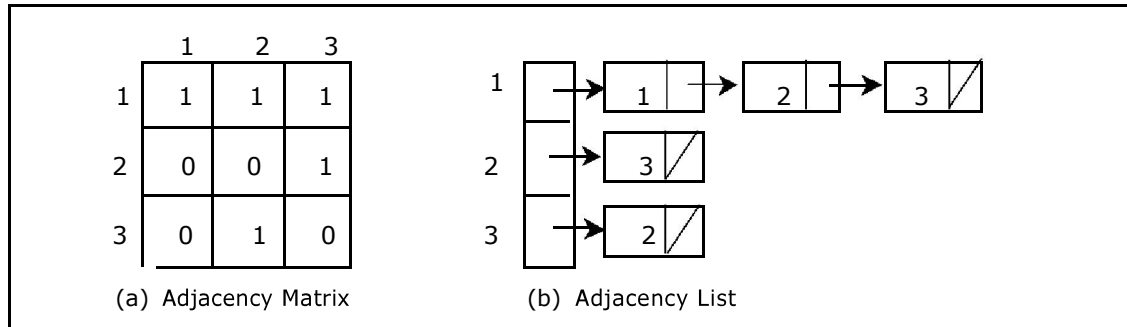


Figure 6.5.4 Adjacency matrix and adjacency list

Incidence Matrix:

In this representation, if G is a graph with n vertices, e edges and no self loops, then incidence matrix A is defined as an n by e matrix, say $A = (a_{i,j})$, where

$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge } j \text{ incident to } v_i \\ 0 & \text{Otherwise} \end{cases}$$

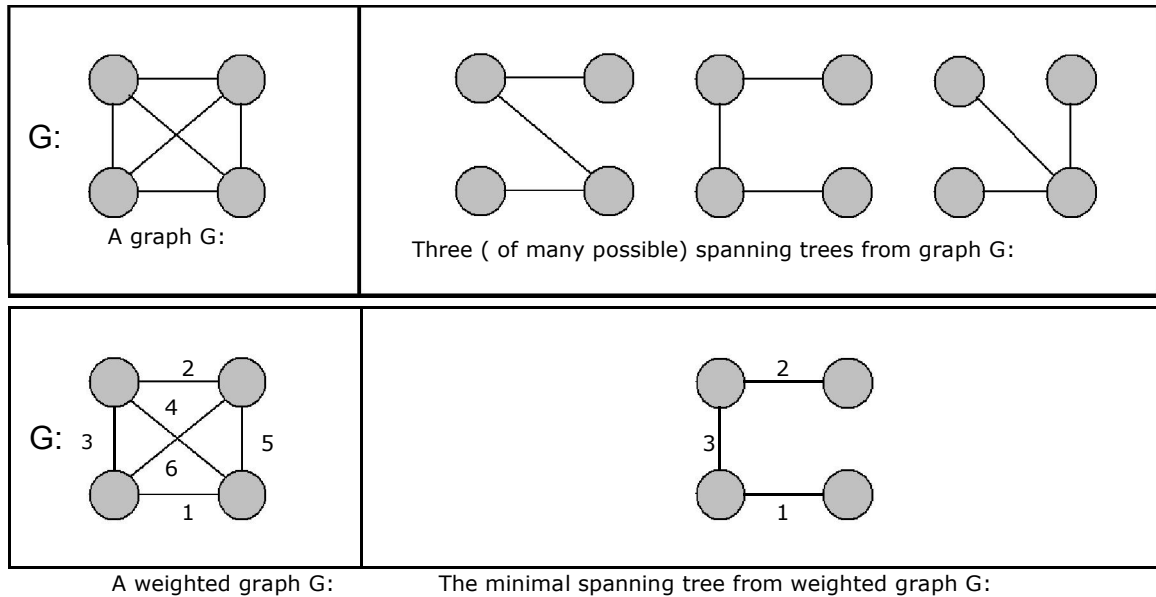
Here, n rows correspond to n vertices and e columns correspond to e edges. Such a matrix is called as vertex-edge incidence matrix or simply incidence matrix.

6.3. Minimum Spanning Tree (MST):

A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

Weight of a spanning tree $w(T)$ is the sum of weights of all edges in T . Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.

Example:



Let's consider a couple of real-world examples on minimum spanning tree:

- One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.
- Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

Minimum spanning tree, can be constructed using any of the following two algorithms:

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

Both algorithms differ in their methodology, but both eventually end up with the MST. *Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections in determining the MST. In Prim's algorithm at any instance of output it represents tree whereas in Kruskal's algorithm at any instance of output it may represent tree or not.*

6.3.1. Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost.

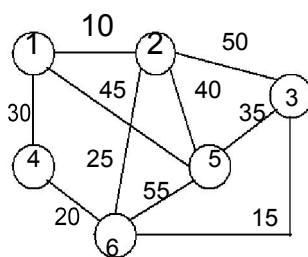
The order in which the edges are chosen, in this case, does not matter. Different MST's may result, but they will all have the same total cost, which will always be the minimum cost.

Kruskal's Algorithm for minimal spanning tree is as follows:

1. Make the tree T empty.
2. Repeat the steps 3, 4 and 5 as long as T contains less than n - 1 edges and E is not empty otherwise, proceed to step 6.
3. Choose an edge (v, w) from E of lowest cost.
4. Delete (v, w) from E.
5. If (v, w) does not create a cycle in T
 then Add (v, w) to T
 else discard (v, w)
6. If T contains fewer than n - 1 edges then print no spanning tree.

Example 1:

Construct the minimal spanning tree for the graph shown below:



Arrange all the edges in the increasing order of their costs:

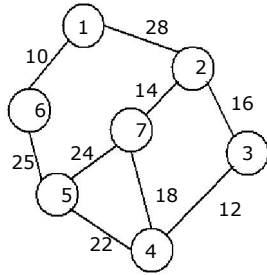
Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	COST	STAGES IN KRUSKAL'S ALGORITHM	REMARKS
(1, 2)	10		The edge between vertices 1 and 2 is the first edge selected. It is included in the spanning tree.
(3, 6)	15		Next, the edge between vertices 3 and 6 is selected and included in the tree.
(4, 6)	20		The edge between vertices 4 and 6 is next included in the tree.
(2, 6)	25		The edge between vertices 2 and 6 is considered next and included in the tree.
(1, 4)	30	Reject	The edge between the vertices 1 and 4 is discarded as its inclusion creates a cycle.
(3, 5)	35		Finally, the edge between vertices 3 and 5 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 105.

Example 2:

Construct the minimal spanning tree for the graph shown below:



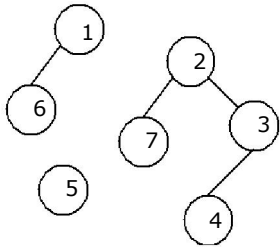
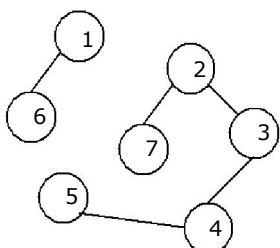
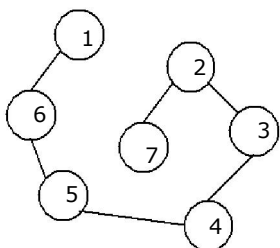
Solution:

Arrange all the edges in the increasing order of their costs:

Cost	10	12	14	16	18	22	24	25	28
Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	COST	STAGES IN KRUSKAL'S ALGORITHM	REMARKS
(1, 6)	10		The edge between vertices 1 and 6 is the first edge selected. It is included in the spanning tree.
(3, 4)	12		Next, the edge between vertices 3 and 4 is selected and included in the tree.
(2, 7)	14		The edge between vertices 2 and 7 is next included in the tree.

(2, 3)	16		The edge between vertices 2 and 3 is next included in the tree.
(4, 7)	18	Reject	The edge between the vertices 4 and 7 is discarded as its inclusion creates a cycle.
(4, 5)	22		The edge between vertices 4 and 7 is considered next and included in the tree.
(5, 7)	24	Reject	The edge between the vertices 5 and 7 is discarded as its inclusion creates a cycle.
(5, 6)	25		Finally, the edge between vertices 5 and 6 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 99.

6.3.2. MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree. Prim's algorithm is an example of a greedy algorithm.

Prim's Algorithm:

E is the set of edges in G . $\text{cost}[1:n, 1:n]$ is the cost adjacency matrix of an n vertex graph such that $\text{cost}[i, j]$ is either a positive real number or if no edge (i, j) exists. A minimum spanning tree is computed and stored as a set of edges in the array $t[1:n-1, 1:2]$. $(t[i, 1], t[i, 2])$ is an edge in the minimum-cost spanning tree. The final cost is returned.

Algorithm Prim (E, cost, n, t)

```
{
  Let  $(k, l)$  be an edge of minimum cost in  $E$ ;
  mincost := cost  $[k, l]$ ;
   $t[1, 1] := k; t[1, 2] := l$ ;
  for  $i := 1$  to  $n$  do // Initialize near
    if  $(\text{cost}[i, l] < \text{cost}[i, k])$  then near  $[i] := l$ ;
    else near  $[i] := k$ ;
  near  $[k] := \text{near}[l] := 0$ ;
  for  $i := 2$  to  $n - 1$  do // Find  $n - 2$  additional edges for  $t$ .
  {
    Let  $j$  be an index such that near  $[j] \neq 0$  and
    cost  $[j, \text{near}[j]]$  is minimum;
     $t[i, 1] := j; t[i, 2] := \text{near}[j]$ ;
    mincost := mincost + cost  $[j, \text{near}[j]]$ ;
    near  $[j] := 0$ 
    for  $k := 1$  to  $n$  do // Update near[.
      if  $((\text{near}[k] \neq 0) \text{ and } (\text{cost}[k, \text{near}[k]] > \text{cost}[k, j]))$ 
      then near  $[k] := j$ ;
  }
  return mincost;
}
```

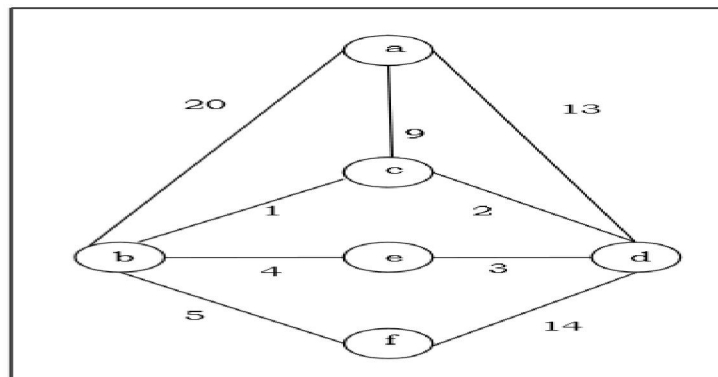
Prim, is a greedy algorithm that finds a minimum spanning tree for a connected weighted graph. It finds a tree of that graph which includes every vertex and the total weight of all the edges in the tree is less than or equal to every possible spanning tree. Prim's algorithm is faster on dense graphs.

Algorithm

- Initialize the minimal spanning tree with a single vertex, randomly chosen from the graph.
- Repeat steps 3 and 4 until all the vertices are included in the tree.
- Select an edge that connects the tree with a vertex not yet in the tree, so that the weight of the edge is minimal and inclusion of the edge does not form a cycle.
- Add the selected edge and the vertex that it connects to the tree.

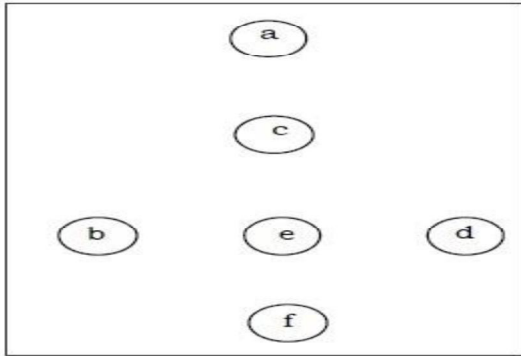
Problem

Suppose we want to find minimum spanning tree for the following graph G using Prim's algorithm.

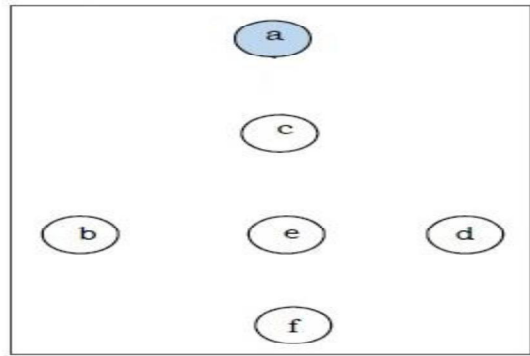


Solution

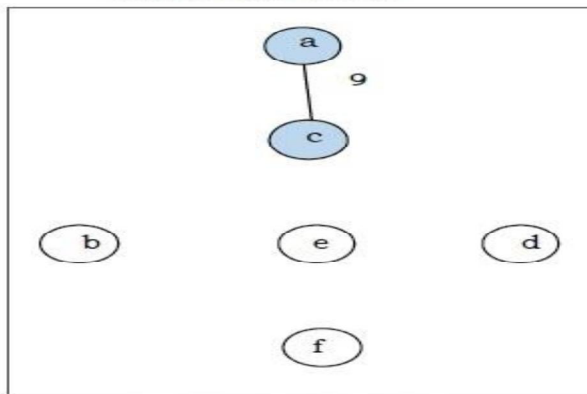
Here we start with the vertex 'a' and proceed.



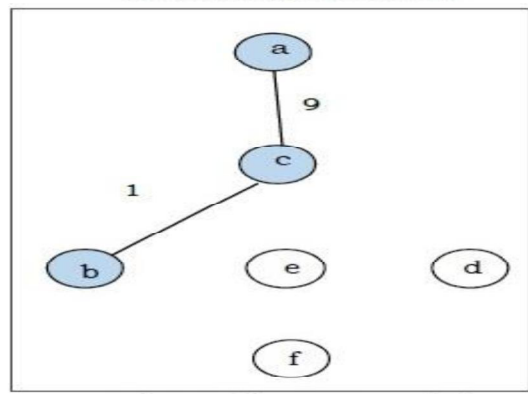
No vertices added



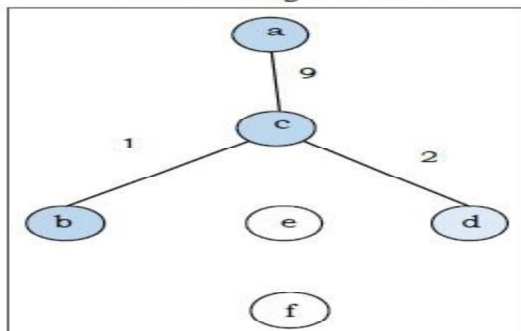
After adding vertex 'a'



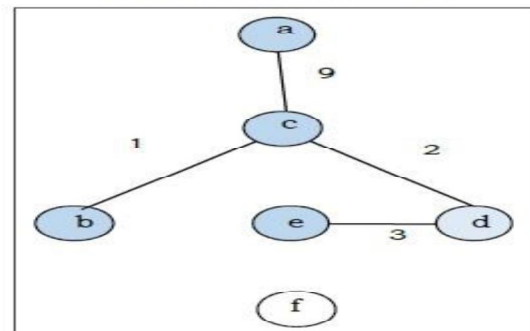
After adding vertex 'c'



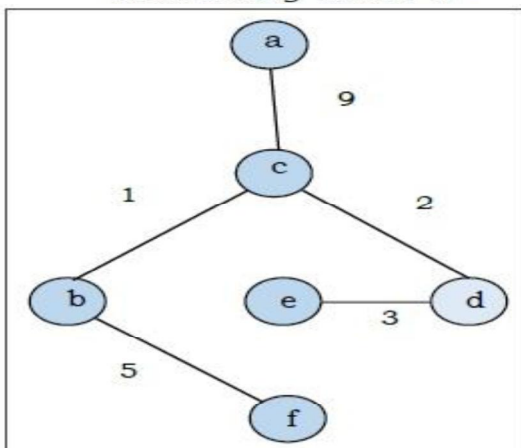
After adding vertex 'b'



After adding vertex 'd'



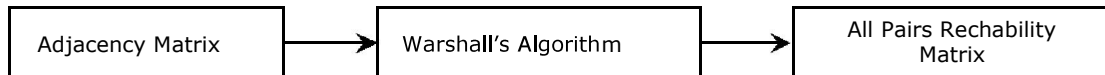
After adding vertex 'e'



After adding vertex 'f'

6.4. Reachability Matrix (Warshall's Algorithm):

Warshall's algorithm requires knowing which edges exist and which does not. It doesn't need to know the lengths of the edges in the given directed graph. This information is conveniently displayed by adjacency matrix for the graph, in which a '1' indicates the existence of an edge and '0' indicates non-existence.



It begins with the adjacency matrix for the given graph, which is called A_0 , and then updates the matrix 'n' times, producing matrices called A_1, A_2, \dots, A_n and then stops.

In warshall's algorithm the matrix A_i contains information about the existence of i -paths. A one entry in the matrix A_i will correspond to the existence of i -paths and zero entry will correspond to non-existence. Thus when the algorithm stops, the final matrix A_n , contains the desired connectivity information.

A one entry indicates a pair of vertices, which are connected and zero entry indicates a pair, which are not. This matrix is called a *reachability matrix or path matrix* for the graph. It is also called the *transitive closure* of the original adjacency matrix.

The update rule for computing A_i from A_{i-1} in warshall's algorithm is:

$$A_i[x, y] = A_{i-1}[x, y] \vee (A_{i-1}[x, i] \wedge A_{i-1}[i, y]) \quad \text{----} \quad (1)$$

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

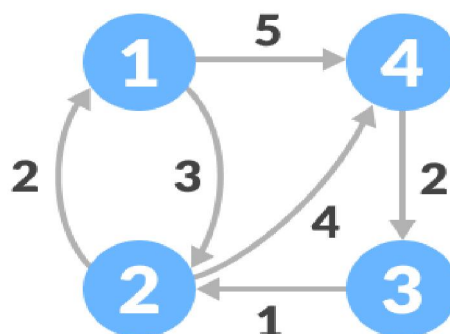
A weighted graph is a graph in which each edge has a numerical value associated with it.

Floyd-Warshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm or WFI algorithm.

This algorithm follows the dynamic programming approach to find the shortest paths.

How Floyd-Warshall Algorithm Works?

Let the given graph be:



Follow the steps below to find the shortest path between all the pairs of vertices.

1. Create a matrix A^1 of dimension $n \times n$ where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

Each cell $A[i][j]$ is filled with the distance from the i^{th} vertex to the j^{th} vertex. If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.

$$A^0 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{matrix}$$

2. Now, create a matrix A^1 using matrix A^0 . The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. $A[i][j]$ is filled with $(A[i][k] + A[k][j])$ if $(A[i][j] > A[i][k] + A[k][j])$.

That is, if the direct distance from the source to the destination is greater than the path through the vertex k , then the cell is filled with $A[i][k] + A[k][j]$.

In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k .

$$A^1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{matrix} \longrightarrow \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{matrix}$$

For example: For $A^1[2, 4]$, the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since $4 < 7$, $A^0[2, 4]$ is filled with 4.

3. In a similar way, A^2 is created using A^1 . The elements in the second column and the second row are left as they are.

In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in **step 2**.

$$A^2 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & & \\ 2 & 0 & 9 & 4 \\ & 1 & 0 & \\ & \infty & & 0 \end{bmatrix} \end{matrix} \longrightarrow \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{matrix}$$

4. Similarly, A^3 and A^4 is also created.

$$A^3 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & & \infty & \\ & 0 & 9 & \\ \infty & 1 & 0 & 8 \\ & & 2 & 0 \end{bmatrix} \end{matrix} \longrightarrow \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{matrix}$$

$$A^4 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & & & 5 \\ & 0 & & 4 \\ & & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{matrix} \longrightarrow \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{matrix}$$

5. A^4 gives the shortest path between each pair of vertices.

Floyd-Warshall Algorithm

n = no of vertices

A = matrix of dimension n*n

for k = 1 to n

 for i = 1 to n

 for j = 1 to n

$A^k[i, j] = \min (A^{k-1}[i, j], A^{k-1}[i, k] + A^{k-1}[k, j])$

return A

6.5. Traversing a Graph

Many graph algorithms require one to systematically examine the nodes and edges of a graph G . There are two standard ways to do this. They are:

- Breadth first traversal (BFT)
- Depth first traversal (DFT)

The BFT will use a queue as an auxiliary structure to hold nodes for future processing and the DFT will use a STACK.

During the execution of these algorithms, each node N of G will be in one of three states, called the *status* of N , as follows:

1. STATUS = 1 (Ready state): The initial state of the node N .
2. STATUS = 2 (Waiting state): The node N is on the QUEUE or STACK, waiting to be processed.
3. STATUS = 3 (Processed state): The node N has been processed.

Both BFS and DFS impose a tree (the BFS/DFS tree) on the structure of graph. So, we can compute a spanning tree in a graph. The computed spanning tree is not a minimum spanning tree. The spanning trees obtained using depth first search are called depth first spanning trees. The spanning trees obtained using breadth first search are called Breadth first spanning trees.

6.5.1. Breadth first search and traversal:

The general idea behind a breadth first traversal beginning at a starting node A is as follows. First we examine the starting node A . Then we examine all the neighbors of A . Then we examine all the neighbors of neighbors of A . And so on. We need to keep track of the neighbors of a node, and we need to guarantee that no node is processed more than once. This is accomplished by using a QUEUE to hold nodes that are waiting to be processed, and by using a field STATUS that tells us the current status of any node. The spanning trees obtained using BFS are called Breadth first spanning trees.

Breadth first traversal algorithm on graph G is as follows:

This algorithm executes a BFT on graph G beginning at a starting node A .

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

1. Put the starting node A in QUEUE and change its status to the waiting state (STATUS = 2).
2. Repeat the following steps until QUEUE is empty:
 - a. Remove the front node N of QUEUE. Process N and change the status of N to the processed state (STATUS = 3).
 - b. Add to the rear of QUEUE all the neighbors of N that are in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).
3. Exit.

6.5.2. Depth first search and traversal:

Depth first search of undirected graph proceeds as follows: First we examine the starting node V. Next an unvisited vertex 'W' adjacent to 'V' is selected and a depth first search from 'W' is initiated. When a vertex 'U' is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited, which has an unvisited vertex 'W' adjacent to it and initiate a depth first search from W. The search terminates when no unvisited vertex can be reached from any of the visited ones.

This algorithm is similar to the inorder traversal of binary tree. DFT algorithm is similar to BFS except now use a STACK instead of the QUEUE. Again field STATUS is used to tell us the current status of a node.

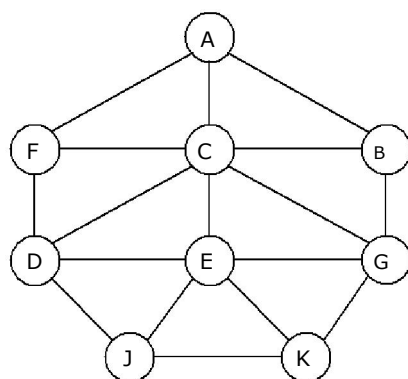
The algorithm for depth first traversal on a graph G is as follows.

This algorithm executes a DFT on graph G beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1).
2. Push the starting node A into STACK and change its status to the waiting state (STATUS = 2).
3. Repeat the following steps until STACK is empty:
 - a. Pop the top node N from STACK. Process N and change the status of N to the processed state (STATUS = 3).
 - b. Push all the neighbors of N that are in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).
4. Exit.

Example 1:

Consider the graph shown below. Traverse the graph shown below in breadth first order and depth first order.



A Graph G

Node	Adjacency List
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

Adjacency list for graph G

Breadth-first search and traversal:

The steps involved in breadth first traversal are as follows:

Current Node	QUEUE	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	F C B	A	3	2	2	1	1	2	1	1	1
F	C B D	A F	3	2	2	2	1	3	1	1	1
C	B D E G	A F C	3	2	3	2	2	3	2	1	1
B	D E G	A F C B	3	3	3	2	2	3	2	1	1
D	E G J	A F C B D	3	3	3	3	2	3	2	2	1
E	G J K	A F C B D E	3	3	3	3	3	3	2	2	2
G	J K	A F C B D E G	3	3	3	3	3	3	3	2	2
J	K	A F C B D E G J	3	3	3	3	3	3	3	3	2
K	EMPTY	A F C B D E G J K	3	3	3	3	3	3	3	3	3

For the above graph the breadth first traversal sequence is: **A F C B D E G J K**.

Depth-first search and traversal:

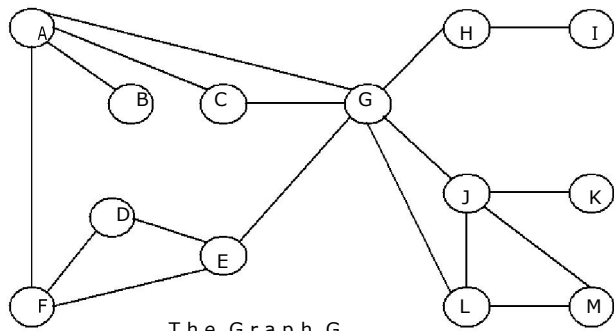
The steps involved in depth first traversal are as follows:

Current Node	Stack	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	B C F	A	3	2	2	1	1	2	1	1	1
F	B C D	A F	3	2	2	2	1	3	1	1	1
D	B C E J	A F D	3	2	2	3	2	3	1	2	1
J	B C E K	A F D J	3	2	2	3	2	3	1	3	2
K	B C E G	A F D J K	3	2	2	3	2	3	2	3	3
G	B C E	A F D J K G	3	2	2	3	2	3	3	3	3
E	B C	A F D J K G E	3	2	2	3	3	3	3	3	3
C	B	A F D J K G E C	3	2	3	3	3	3	3	3	3
B	EMPTY	A F D J K G E C B	3	3	3	3	3	3	3	3	3

For the above graph the depth first traversal sequence is: **A F D J K G E C B**.

Example 2:

Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.

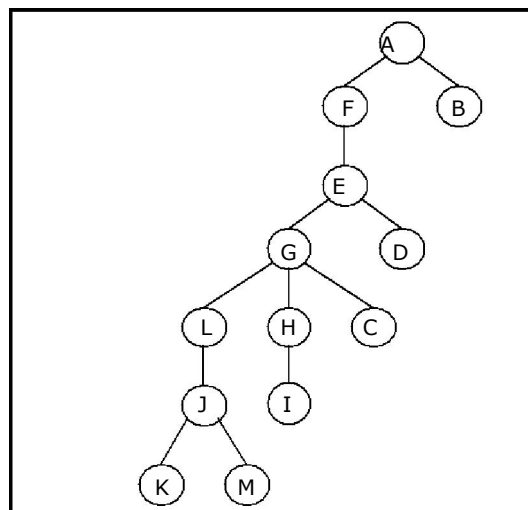


The Graph G

Node	Adjacency List
A	F, B, C, G
B	A
C	A, G
D	E, F
E	G, D, F
F	A, E, D
G	A, L, E, H, J, C
H	G, I
I	H
J	G, L, K, M
K	J
L	G, J, M
M	L, J

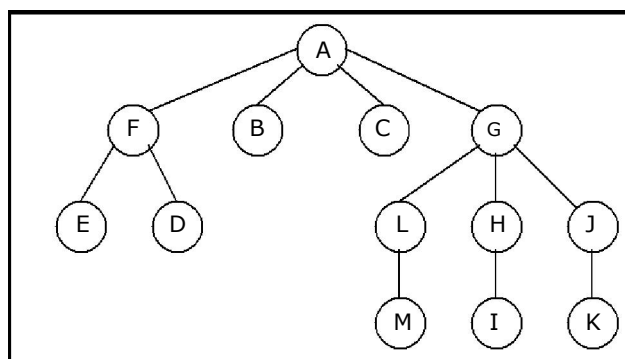
The adjacency list for the graph G

If the depth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: **A F E G L J K M H I C D B**. The depth first spanning tree is shown in the figure given below:



Depth first Traversal

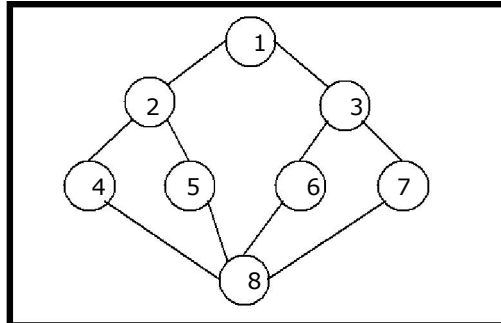
If the breadth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: **A F B C G E D L H J M I K**. The breadth first spanning tree is shown in the figure given below:



Breadth first traversal

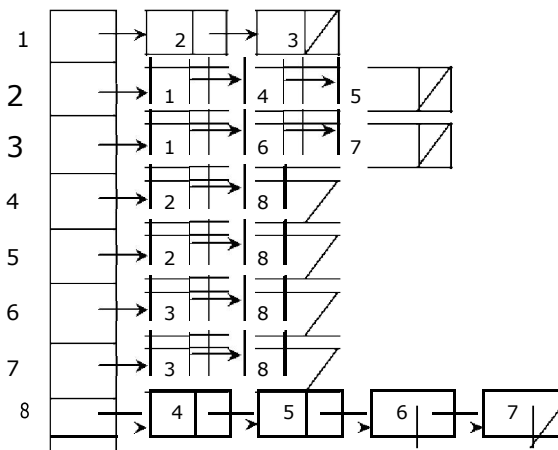
Example 3:

Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.



Graph G

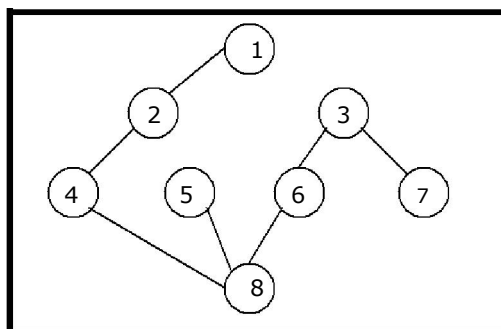
Head Nodes



Adjacency list for graph G

Depth first search and traversal:

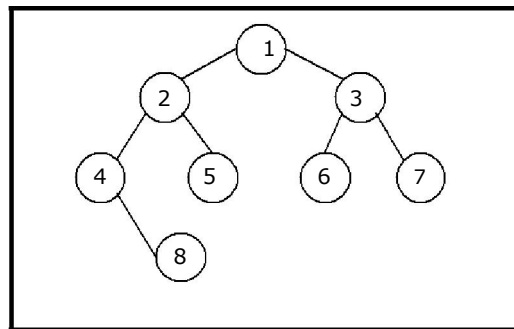
If the depth first is initiated from vertex 1, then the vertices of graph G are visited in the order: 1, 2, 4, 8, 5, 6, 3, 7. The depth first spanning tree is as follows:



Depth First Spanning Tree

Breadth first search and traversal:

If the breadth first search is initiated from vertex 1, then the vertices of G are visited in the order: 1, 2, 3, 4, 5, 6, 7, 8. The breadth first spanning tree is as follows:

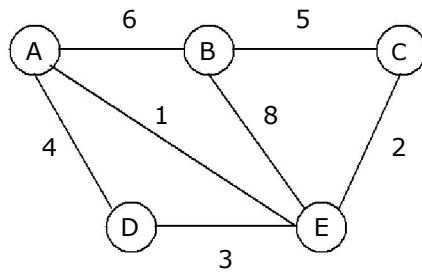


Breadth First Spanning Tree

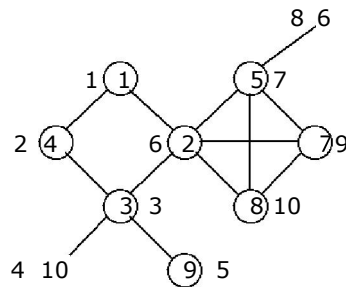
EXCERCISES

1. Show that the sum of degrees of all vertices in an undirected graph is twice the number of edges.
2. Show that the number of vertices of odd degree in a finite graph is even.
3. How many edges are contained in a complete graph of "n" vertices.
4. Show that the number of spanning trees in a complete graph of "n" vertices is $2^{n-1} - 1$.
5. Prove that the edges explored by a breadth first or depth first traversal of a connected graph form a tree.
6. Explain how existence of a cycle in an undirected graph may be detected by traversing the graph in a depth first manner.
7. Write a "C" function to generate the incidence matrix of a graph from its adjacency matrix.
8. Give an example of a connected directed graph so that a depth first traversal of that graph yields a forest and not a spanning tree of the graph.
9. Rewrite the algorithms "BFSearch" and "DFSearch" so that it works on adjacency matrix representation of graphs.
10. Write a "C" function to find out whether there is a path between any two vertices in a graph (i.e. to compute the transitive closure matrix of a graph)
11. Write a "C" function to delete an existing edge from a graph represented by an adjacency list.
12. Construct a weighted graph for which the minimal spanning trees produced by Kruskal's algorithm and Prim's algorithm are different.

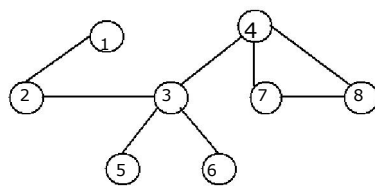
13. Describe the algorithm to find a minimum spanning tree T of a weighted graph G . Find the minimum spanning tree T of the graph shown below.



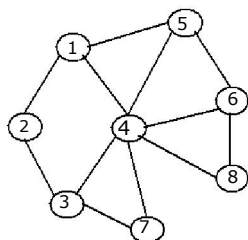
14. For the graph given below find the following:
- Linked representation of the graph.
 - Adjacency list.
 - Depth first spanning tree.
 - Breadth first spanning tree.
 - Minimal spanning tree using Kruskal's and Prim's algorithms.



15. For the graph given below find the following:
- Linked representation of the graph.
 - Adjacency list.
 - Depth first spanning tree.
 - Breadth first spanning tree.
 - Minimal spanning tree using Kruskal's and Prim's algorithms.



16. For the graph given below find the following:
- Linked representation of the graph.
 - Adjacency list.
 - Depth first spanning tree.
 - Breadth first spanning tree.
 - Minimal spanning tree using Kruskal's and Prim's algorithms.



11. A simple graph has no loops. What other property must a simple graph have? []
 A. It must be directed. C. It must have at least one vertex.
 B. It must be undirected. D. It must have no multiple edges.
12. Suppose you have a directed graph representing all the flights that an airline flies. What algorithm might be used to find the best sequence of connections from one city to another? []
 A. Breadth first search. C. A cycle-finding algorithm.
 B. Depth first search. D. A shortest-path algorithm.
13. If G is an directed graph with 20 vertices, how many boolean values will be needed to represent G using an adjacency matrix? []
 A. 20 C. 200
 B. 40 D. 400
14. Which graph representation allows the most efficient determination of the existence of a particular edge in a graph? []
 A. An adjacency matrix. C. Incidence matrix
 B. Edge lists. D. none of the above
15. What graph traversal algorithm uses a queue to keep track of vertices which need to be processed? []
 A. Breadth-first search. C Level order search
 B. Depth-first search. D. none of the above
16. What graph traversal algorithm uses a stack to keep track of vertices which need to be processed? []
 A. Breadth-first search. C Level order search
 B. Depth-first search. D. none of the above
17. What is the expected number of operations needed to loop through all the edges terminating at a particular vertex given an adjacency matrix representation of the graph? (Assume n vertices are in the graph and m edges terminate at the desired node.) []
 A. $O(m)$ C. $O(m^2)$
 B. $O(n)$ D. $O(n^2)$
18. What is the expected number of operations needed to loop through all the edges terminating at a particular vertex given an adjacency list representation of the graph? (Assume n vertices are in the graph and m edges terminate at the desired node.) []
 A. $O(m)$ C. $O(m^2)$
 B. $O(n)$ D. $O(n^2)$
19. []

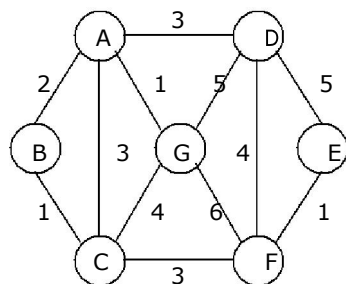


FIGURE 3

For the figure 3, starting at vertex A, which is a correct order for Prim's minimum spanning tree algorithm to add edges to the minimum spanning tree?

- A. (A, G) then (G, C) then (C, B) then (C, F) then (F, E) then (E, D)
- B. (A, G) then (A, B) then (B, C) then (A, D) then (C, F) then (F, E)
- C. (A, G) then (B, C) then (E, F) then (A, B) then (C, F) then (D, E)
- D. (A, G) then (A, B) then (A, C) then (A, D) then (A, D) then (C, F)

20. For the figure 3, which is a correct order for Kruskal's minimum spanning tree algorithm to add edges to the minimum spanning tree? []

- A. (A, G) then (G, C) then (C, B) then (C, F) then (F, E) then (E, D)
- B. (A, G) then (A, B) then (B, C) then (A, D) then (C, F) then (F, E)
- C. (A, G) then (B, C) then (E, F) then (A, B) then (C, F) then (D, E)
- D. (A, G) then (A, B) then (A, C) then (A, D) then (A, D) then (C, F)

21. Which algorithm does not construct an in-tree as part of its processing? []

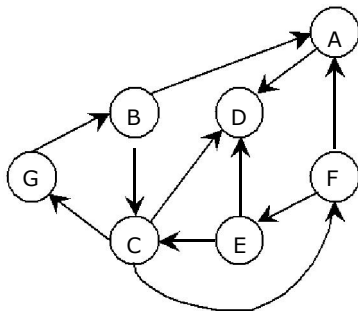
- A. Dijkstra's Shortest Path Algorithm
- B. Prim's Minimum Spanning Tree Algorithm
- C. Kruskal's Minimum Spanning Tree Algorithm
- D. The Depth-First Search Trace Algorithm

22. The worst-case running time of Kruskal's minimum-cost spanning tree algorithm on a graph with n vertices and m edges is: []

- A. C.
- B. D.

23. An adjacency matrix representation of a graph cannot contain information of: []

- A. Nodes
- B. Edges
- C. Direction of edges
- D. Parallel edges



Node	Adjacency List
A	D
B	A C
C	G D F
D	----
E	C D
F	E A
G	B

FIGURE 4 and its adjacency list

24. For the figure 4, which edge does not occur in the depth first spanning tree resulting from depth first search starting at node B: []

- A. F → E
- B. E → C
- C. C → G
- D. C → F

25. The set of all edges generated by DFS tree starting at node B is: []

- A. B A D C G F E
- B. A D
- C. B A C D G F E
- D. Cannot be generated

26. The set of all edges generated by BFS tree starting at node B is: []

- A. B A D C G F E
- B. A D
- C. B A C D G F E
- D. Cannot be generated