Unit 12 Graphs and Their Applications

Structure:

12.1 Introduction

In computer science, a **graph** is a kind of data structure, specifically an abstract data type (ADT) that consists of a set of nodes and a set of edges that establish relationships (connections) between the nodes. The graph ADT follows directly from the graph concept in mathematics.

A graph *G* is defined as follows: *G= (V, E)*, where *V* is a finite, non-empty set of vertices (singular: vertex) and *E* is a set of edges (links between pairs of vertices). When the edges in a graph have no direction, the graph is called undirected, otherwise called directed. In practice, some information is associated with each node and edge.

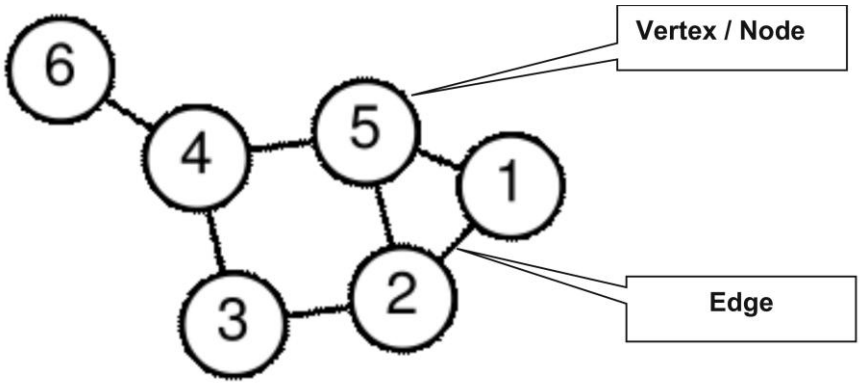


Fig. 12.1: Graph G

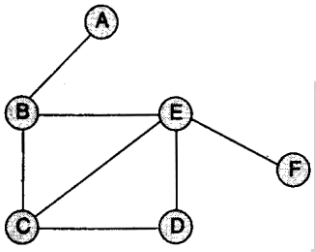
Objectives:

At the end of this unit, you will be able to explain:

- The concept of Graphs.
- The formation of Adjacency lists and Adjacency Matrix.
- Working principles of Depth-First and Breadth-First Traversals.
- Explain Spanning Tree algorithms.
- The applications of graphs

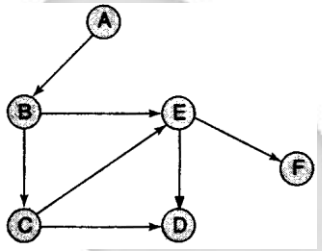
12.2 Graph Terminologies

- **Undirected Graph:** When the edges in a graph have no direction, the graph is called undirected.



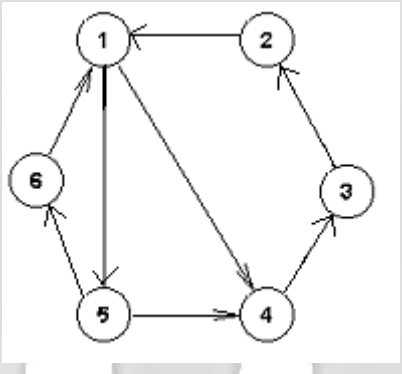
Graph G1

- **Directed Graph (Digraph):** When the edges in a graph have direction, the graph is called undirected**.**

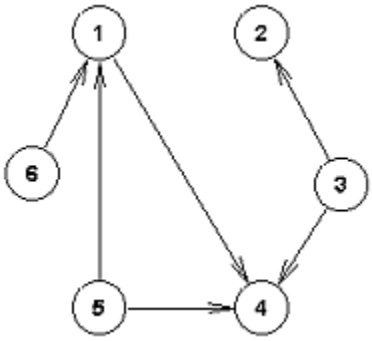


Graph G2

- **Strongly Connected Graph:** Two vertices are said to be connected if there is a path between them. Furthermore, a directed graph is strongly connected if there is a path from each vertex to every other vertex in the digraph.

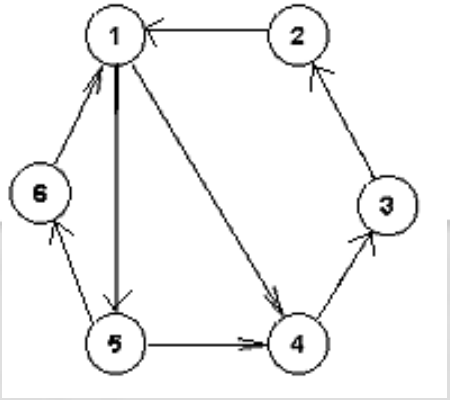


- **Weakly Connected Graph:** A directed graph is weakly connected if at least two vertices are not connected.



Graph G4

- **Degree of a vertex:** The degree of a vertex is the number of lines incident to it.
  1. The **outdegree** of a vertex in a digraph is the number of arcs leaving the vertex.
  2. The **indegree** is the number of arcs entering the vertex.

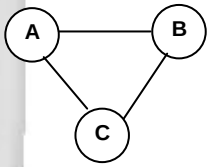


The **indegree** of the vertex 4 is 2 and **outdegree** is 1

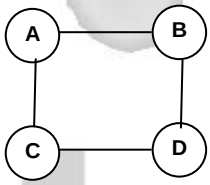
- **Complete Graph:** A graph, in which there is exactly one edge between each pair of distinct vertices, is called a complete graph.

Graph G6

Graph G7

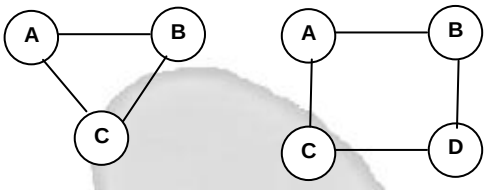


Graph G8

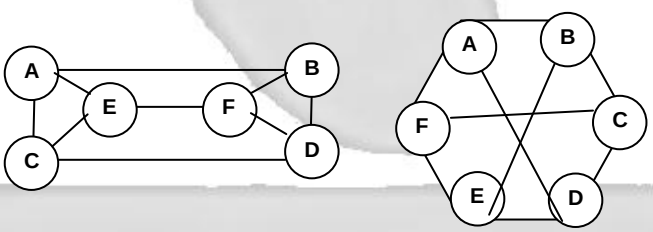


Graph G9

- **Regular Graph:** If every vertex of any graph has the same degree, then the graph is called a regular graph. If every vertex in a regular graph has degree – n, then the graph is called n-regular graph.

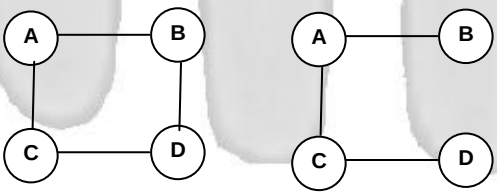


2- Regular Graphs



3- Regular Graphs

- **Sub Graph:** A graph H = (V', E') is called a subgraph of G=(V,G), If V' $\subseteq$ V and E' $\subseteq$ E.
  1. If V' $\subset$ V and E' $\subset$ E, then H is called a proper subgraph of G.
  2. If V' = V then H is called a spanning subgraph of G. A spanning gubgraph of G. A spanning subgraph of G need not contain all its edges.



Graph G Graph G', Subgraph of G

12.3 Graph Representation

Two main data structures for the representation of graphs are used in practice. The first is called an **adjacency list** and is implemented by representing each node as a data structure that contains a list of all adjacent nodes. The second is an **adjacency matrix**, in which the rows and columns

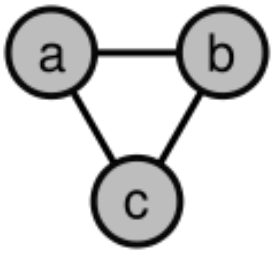
of a two-dimensional array represent source and destination vertices and entries in the graph indicate whether an edge exists between the vertices. Adjacency lists are preferred for sparse graphs; otherwise, an adjacency matrix is a good choice. Finally, for very large graphs with some regularity in the placement of edges, a symbolic graph is a possible choice of representation.

12.3.1 Adjacency Lists

In graph theory, an **adjacency list** is the representation of all edges or arcs in a graph as a list. If the graph is undirected, every entry is a set of two nodes containing the two ends of the corresponding edge; if it is directed, every entry is a tuple of two nodes, one denoting the source node and the other denoting the destination node of the corresponding arc.

Typically, adjacency lists are unordered. In computer science, an adjacency list is a closely related data structure for representing graphs. In an adjacency list representation, we keep, for each vertex in the graph, all other vertices which it has an edge to (that vertex's "adjacency list"). For instance, the representation suggested by Van Rossum, in which a hash table is used to associate each vertex with an array of adjacent vertices, can be seen as an instance of this type of representation, as can the representation in Cormen at all in which an array indexed by vertex numbers points to a singly-linked list of the neighbours of each vertex.

Manipal



The graph pictured above has this adjacency list representation:

A

Adjacent to b,c

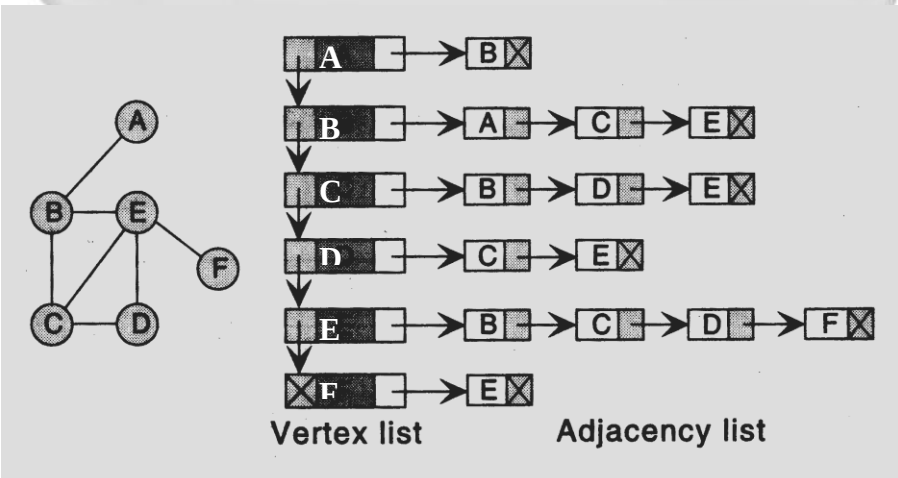
b

Adjacent to a,c

c

Adjacent to a,b

The **adjacency list** uses a two-dimensional ragged array to store the edges. An adjacency list is shown in the following figure

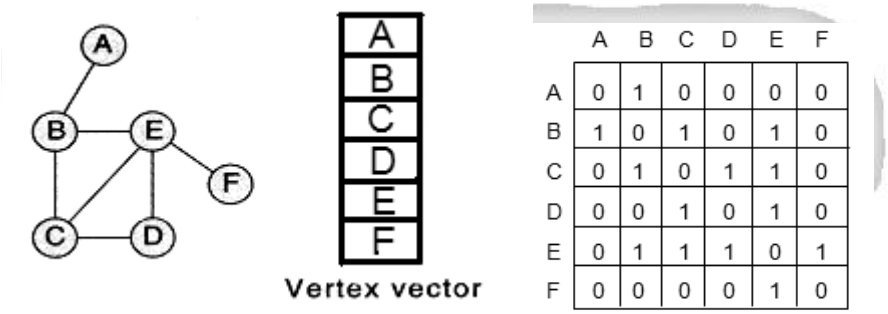


The vertex list is a singly linked list of the vertices in the list. Depending on the application, it could also be implemented using doubly linked lists or circularly linked lists. The pointer at the left of the list links the vertex entries. The pointer at the right in the vertex is a head pointer to a linked list of edges from the vertex. Thus, in the undirected graph on the left in Figure 6.4, there is a path from vertex B to vertices A, C, and E. To find these

edges in the adjacency list, we start at B's vertex list vertex and traverse the linked list to A, then to C, and finally to E.

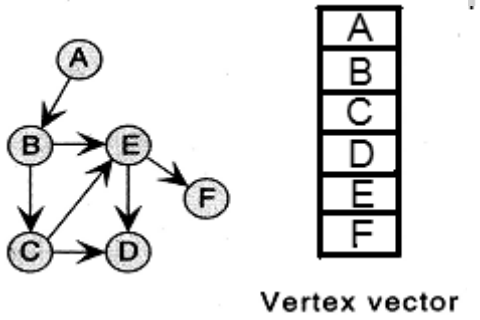
12.3.2 Adjacency Matrix

The **adjacency matrix** uses a vector (one-dimensional array) for the vertices and a matrix (two-dimensional array) to store the edges. If two vertices are adjacent – that is, if there is an edge between them – the matrix intersect has a value of 1; if there is no edge between them, the intersection is set to 0. If the graph is directed, then the intersection in the adjacency matrix indicates the direction.



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

a) Adjacency matrix for non-directed graph



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

b) Adjacency matrix for directed graph

12.4 Graph Traversal

Many graph algorithms require one to systematically examine the nodes and edges of the graph G.

During the execution of the algorithm, each node N of G will be in one of the three status, called the status of N, as follows:

STATUS = 1 (Ready State)

The initial state of the node N

STATUS = 2 (Waiting State)

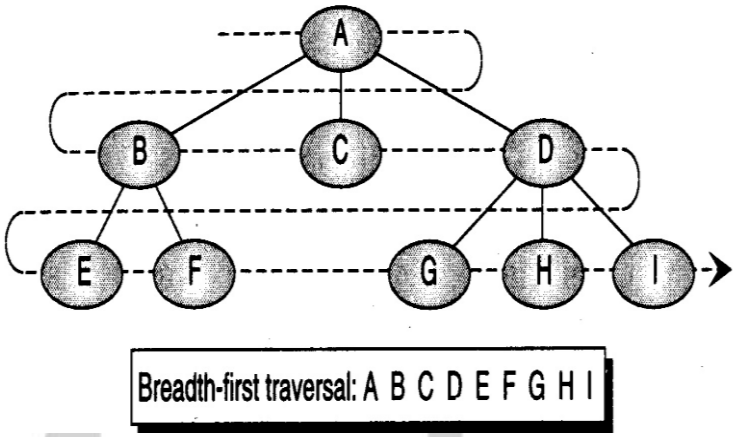
The Node N is on the Queue or Stack. Waiting to be processed

STATUS = 3 (Processed State)

The node N has been processed

12.4.1 Breadth-First Traversal

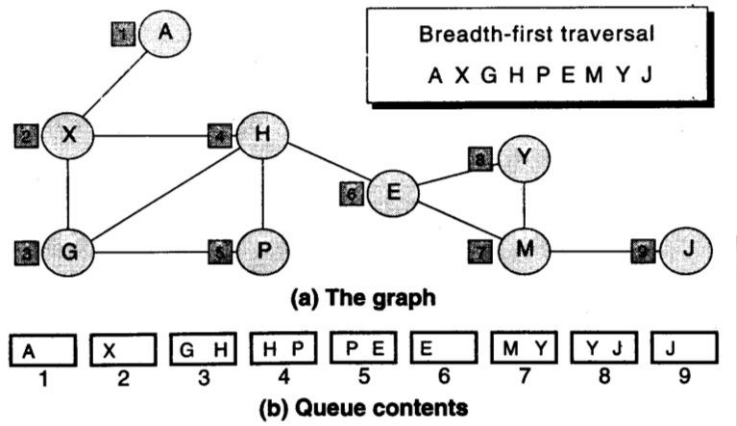
In the breadth-first traversal of a graph, we process all adjacent vertices of a vertex before going to the next level. Looking at the tree, we see that its breadth-first traversal starts at level 1 and then processes all the vertices in level 2 before going on to process the vertices in level 3.



Breadth-first traversal of a tree

The breadth-first traversal of a graph follows the same concept we begin by picking a starting vertex; after processing it, we process all of its adjacent vertices. After we process all of the first vertex adjacent vertices, we pick the first adjacent vertex and process all of its vertices, then the second adjacent

vertex and process all of its vertices and so forth until finished. The breadth-first traversal uses a queue rather than a stack, as we process each vertex; we place all of its adjacent vertices in the queue. Then, to select the next vertex to be processed, we delete a vertex from the queue and process it. Let's trace this logic through the graph in the following figure.



Breadth-first traversal of a graph

Algorithm

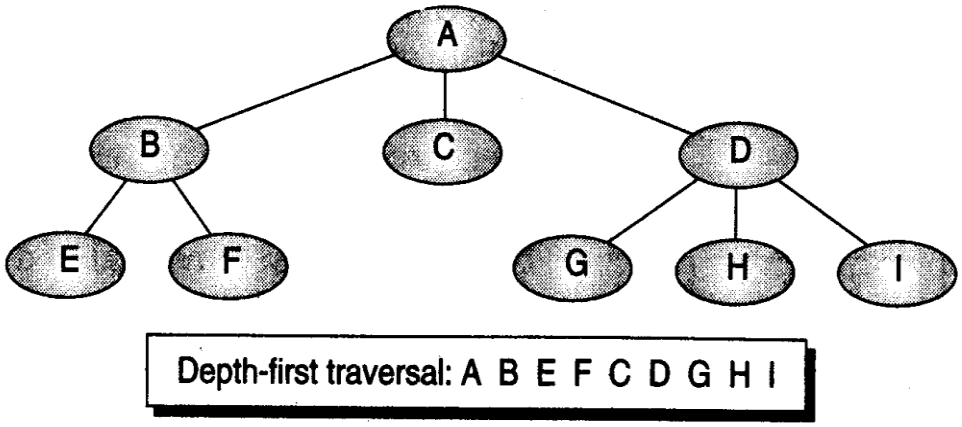
Step I	Initialize all nodes to the ready state (STATUS=1)
Step II	Put the starting node in the Queue and change its status to the waiting state (STATUS=2)
Step III	Repeat Step IV and Step V until Queue is empty
Step IV	Remove the front node N of Queue. Process N and change the status of N to the processed state (STATUS= 3)
Step V	Add to the near of Queue all the neighbour of N that are in the steady state(STATUS=1) and change their status of the waiting state (STATUS=2) [End of Step III Loop]
Step VI	Exit

12.4.2 Depth First Traversal

In the depth-first traversal, we process all of a vertex's descendents before we move to an adjacent vertex. This concept is most easily seen when the graph is a tree. In Figure 7.5, we show the preorder traversal, one of the standard depth-first traversals.

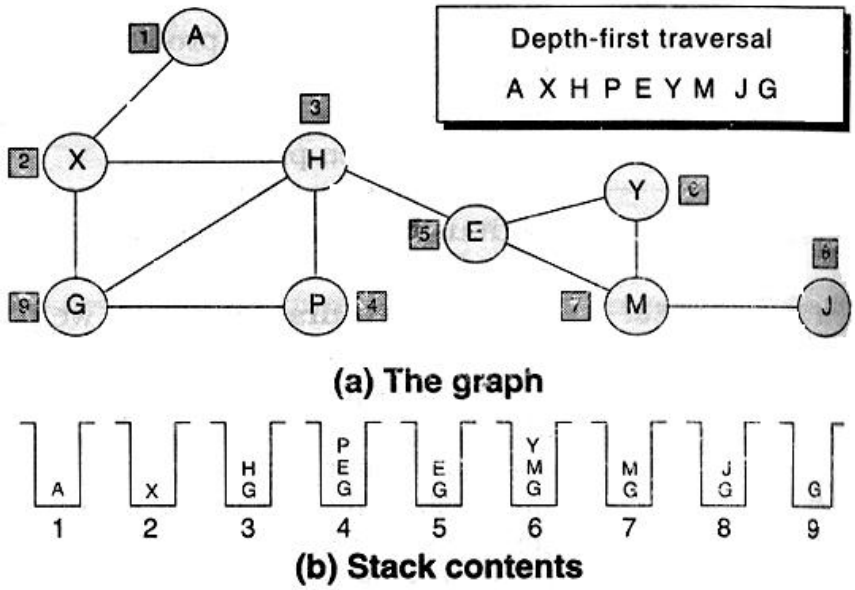
In a similar manner, the depth-first traversal of a graph starts by processing the first vertex of the graph. After processing the first vertex, we select any vertex adjacent to the first vertex and process it. As we process each vertex, we select an adjacent vertex until we reach a vertex with no adjacent entries. This is similar to reaching a leaf in a tree. We then back out of the structure, processing adjacent vertices as we go. It should be obvious that this logic requires a stack (or recursion) to complete the traversal.

The order in which the adjacent vertices are processed depends on how the graph is physically stored.



Depth-first traversal of a tree

Let's trace a depth-first traversal through the graph in following figure. The number in the box next to a vertex indicates the processing order. The stacks below the graph show the stack contents as we way down the graph and then as we back out.



Depth-first traversal of a graph

Algorithm

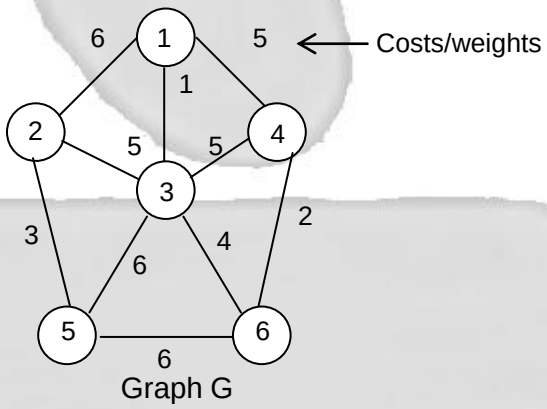
Step – I	Initially all nodes to be ready state (STATUS =1)
Step – II	Put the starting node A in the STACK a... change its status to the waiting state (STATUS=2)
Step –III	Repeat Step IV and V until STACK is empty
Step –IV	Pop the top node N of STACK. Process N and change its status to the processed state (STATUS = 3)
Step –V	Push into STACK all the neighbour of N that are still in ready state (STATUS =1) and change their status to the waiting state (STATUS=2) [End of Step –III Loop]
Step –VI	Exit

12.5 Spanning Trees

A spanning tree of a graph is an undirected tree consisting of only those edges necessary to connect all the nodes in the original graph. For any pair of nodes there exists only one path between them and the insertion of any

edge to a spanning tree forms a unique cycle. Those edges left out of the Spanning tree that were present in the original graph connect paths together in the tree.

A network is a graph that has weights or costs associated with its edges. It is also called weighted graph.



Graph G

Minimum spanning tree: This is a spanning tree that covers all vertices of a network such that the sum of the costs of its edges is minimum. There are two algorithms for finding the minimum spanning tree, given a network.

1) Kruskal's algorithm
2) Prim's algorithm

12.5.1 Kruskal's Algorithm

This algorithm finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. Kruskal's algorithm is an example of a greedy algorithm.

Kruskal's algorithm to find minimum spanning tree

Let G be a graph connected graph with n vertices. In this algorithm we first choose an edge of G which has the smallest weight among the edges of G which are not loops. Next, again avoiding loop, we choose from the

remaining edges the smallest weight possible that does not form a cycle with the edge already chosen. We repeat the process till (n-1) edges have been chosen.

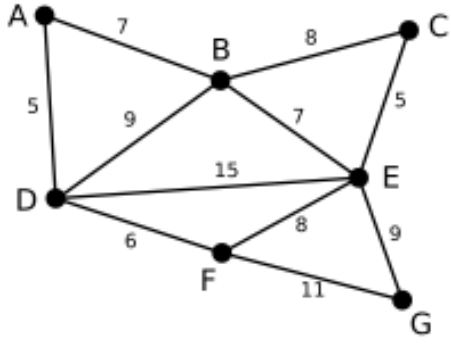
Step I	Choose e_1, an edge of G such that $w(e_1)$ as small as possible and e_1 is not a loop.
Step II	If edges $e_1,e_2,e_3,\ldots,e_i$ have been chosen, then choose an edge e_{i+1}, not already chosen, such that The subgraph induced by $e_1,e_2,e_3,\ldots e_{i+1}$ is a cyclic and $W(e_{i+1})$ is as small as possible, subject to the condition (a)
Step III	If G has n vertices, stop after (n-1) edges have been chosen, otherwise go to step II.

Process

Step 1: Arrange all edges in ascending order of their cost to form an input set.

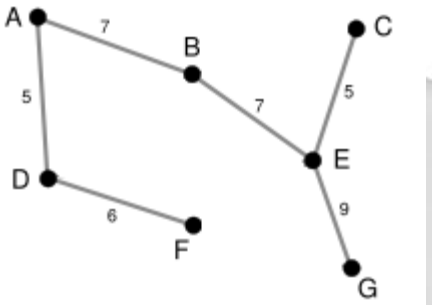
Step 2: From this input set, take each edge and if it does not form a cycle, include it in the output set, which forms the minimum spanning tree. The sum of the costs of the edges in the output set is the least.

Example



Input Graph

Edge	Weight	Remarks
AD	5	√
CE	5	√
DF	6	√



Output Graph

AB
7
√

BE
7
√

BC
8
X

EF
8
X

BD
9
X

EG
9
√

FG
11
X

DE
15
X

12.5.2 Prim's Algorithm

Prim's algorithm to find minimum spanning tree

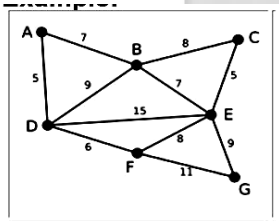
In this algorithm for finding the minimal spanning tree, first choose a vertex v1 of connected graph G (It can be chosen arbitrarily).

We next choose one of the edges of the smallest weight, say e1=(v1,v2) in the Graph G, which is not a loop and which is incident with v1. Then we choose an edge e2=(vi,v3) where i=1or 2, of the smallest weight in G, which is incident with either v1 or v2. We repeat this process of taking edges of

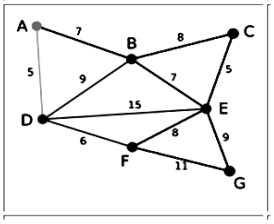
thesmallest weight one of whose ends is a vertex previously chosen until we have chosen (n-1) edges.

Step I	Choose any vertex v1 of G.
Step II	Choose an edge g1=(v_1,v_2) of G, such that $v_1 \neq v_2$ and e_1 has smallest weight among the edges of G incident with v_1.
Step III	If edges e_1,e_2,e_3,…,e_i have been chosen involving endpoint v_1,v_2,v_3,…,v_{i+1} choose an edge ei+1=(v_j,v_k) with V_j {v_1,v_2,…,v_{i+1}} and v_k{v_1,v_2,…,v_{i+1}} Such that, e_{i+1} has the smallest weight among the edge of G with precisely.
Step IV	Stop after (n-1) edges have been chosen, otherwise repeat step III.

Example:

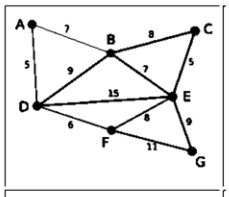


This is our original weighted graph. The numbers near the arcs indicate their weight

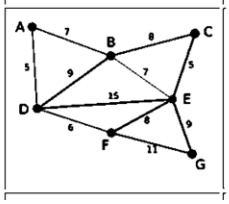


Vertex D has been arbitrarily chosen as a starting point. Vertices A, B, E and F are connected to D through a single edge. A is the vertex nearest to D and will be chosen as the second vertex along with the edge AD.

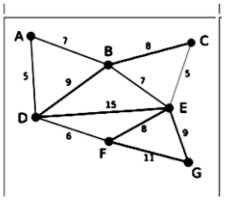
Manipal



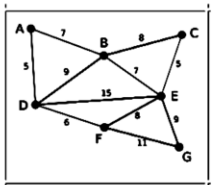
The next vertex chosen is the vertex nearest to *either* **D** or **A**. **B** is 9 away from **D** and 7 away from **A**, **E** is 15, and **F** is 6. **F** is the smallest distance away, so we highlight the vertex **F** and the arc **DF**.



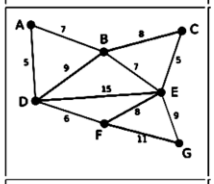
The algorithm carries on as above. Vertex **B**, which is 7 away from **A**, is highlighted.



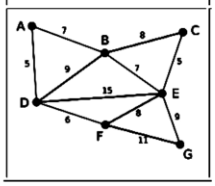
In this case, we can choose between **C**, **E**, and **G**. **C** is 8 away from **B**, **E** is 7 away from **B**, and **G** is 11 away from **F**. **E** is nearest, so we highlight the vertex **E** and the arc **EB**.



Here, the only vertices available are **C** and **G**. **C** is 5 away from **E**, and **G** is 9 away from **E**. **C** is chosen, so it is highlighted along with the arc **EC**.



Vertex **G** is the only remaining vertex. It is 11 away from **F**, and 9 away from **E**. **E** is nearer, so we highlight it and the arc **EG**.



Now all the vertices have been selected and the minimum spanning tree is shown in green. In this case, it has weight **39**.

12.6 Summary

A graph is a kind of data structure, specifically an abstract data type (ADT) that consists of a set of nodes and a set of edges that establish relationships (connections) between the nodes. The graph ADT follows directly from the

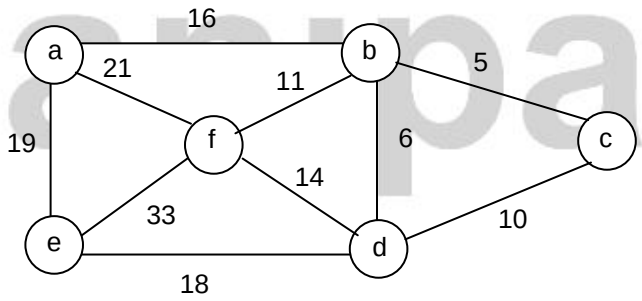
graph concept from mathematics. There are number of different applications of graph that are clearly explained. This chapter covers introduction of graph, traversals, spanning trees.

Self Assessment Questions

1. Graph is a Tree. (True / False)
2. If every vertex of any graph has the same degree, then the graph is called a regular graph. (True / False)
3. A graph, in which there is exactly one edge between each pair of distinct vertices, is called a complete graph. (True / False)
4. An adjacency list is the representation of all edges or arcs in a graph as a list. (True / False)
5. DFS & BFS are similar. (True / False)
6. The result of Kruskal's and Prims's are same. (True / False)

12.7 Terminal Questions

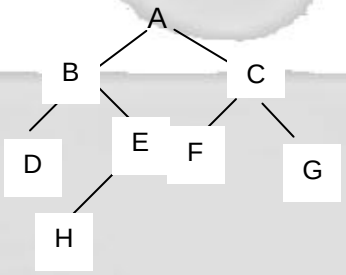
1. Define a graph.
2. Define directed and undirected graph.
3. Explain the Adjacency lists with a suitable example.
4. Explain the Adjacency matrix for directed graph.
5. Explain the depth-first traversal of a graph with a suitable example.
6. Derive the minimum spanning tree for the graph below using both the Prim's and Kruskal's methods.



7. Define Spanning Trees. Explain the *Prim's algorithm to find minimum spanning tree* with a suitable example.

8. A binary tree has 10 nodes. The preorder and inorder traversal of the tree are shown below. Draw the tree.
   Preorder : J C B A D E F I G H
   Inorder : A B C E D F J G I H

9. Give the inorder, postorder and breadth first traversal for the following tree:



10. Show the BST and B tree after the insertion of the following elements according to their respective insertion algorithms – 43, 64, 80, 96, 128, 150 and 250.

11. Given the BST below:

a) Write the algorithm to find the smallest value in the tree.
b) Draw the tree obtained after separately deleting from the original tree i) node 13 ii) node 14 iii) node 23

Manipal

12. Construct the binary tree if the result of traversing it is as below:

    Postorder : I E J F C G K L H D B A

    Inorder : E I C F J B G D K H L A

13. Suppose that keys 1, 2, 3, …, 19, 20 are inserted in that order into a B tree with m = 4. Show the B tree after each insertion. How many internal nodes does the final B tree have?

12.8 Answer to Self Assessment and Terminal Questions

Self Assessment Questions

1. False
2. True
3. True
4. True
5. False
6. True

Terminal Questions

1. Section 12.2
2. Section 12.2
3. Section 12.3.1
4. Section 12.3.2
5. Section 12.4.2
6. Section 12.5.1 – 2
7. Section 12.5.2
8. Section 12.3

Manipal