UNIT 4 GRAPHS

Structure

4.0 INTRODUCTION

So far you have learnt about arrays, lists, stacks and queues in this block. In this unit we introduce you to an important mathematical structure called Graph. Graphs have found applications in subjects as diverse as Sociology, Chemistry, Geography and Engineering Sciences. They are also widely used in solving games and puzzles. In computer science, graphs are used in many areas one of which is computer design. In day-to-day applications, graphs find their importance as representations of many kinds of physical structure.

Let us consider a very simple case and see how theory of graphs can help us to solve set of problems. A problem which occurs frequently in modern life is that of time tabling a events in such a way as to avoid clashes. Suppose we wish to schedule six one-day exhibitions A, B, C, D, E and F with the provision of holding more than one exhibition a day. Among the potential visitors there are people who wish to visit both

A and B

A and D

C and E

B and F

E and F

And A and F

We can represent the situation by a graph. The vertices correspond to six exhibitions and edges signify the potential clashes.

Just by looking at the graph, you may arrive at a schedule that achieves the objective of clashes given as follows:

Day 1 Day 2 Day 3 Day 4

A&C B&D E F

How many days are necessary in order that the exhibitions may be held without clash?

Historically, graph theory originated in Konigsberg bridge problem. The Prussian city of Konigsberg was built along the Preger River and occupied both banks and two islands. The problem was to seek a route that would enable one to cross all seven bridges in the city exactly once and return to their starting point. Leonhard Euler, a mathematician developed some concepts in solving this problem and these concepts formed the basis of the graph theory. He modeled the problem by treating each land mass as a vertex and each bridge as an edge, deriving the graph as shown in Figure below:

Frequently, we use graphs as models of practical situations involving routes: the vertices represent the cities and edges represent the roads or some other links, specially in transportation management, Assignment problems and many more optimization problems. Electric circuits are another obvious example where interconnections between objects play a central role. Circuits elements like transistors, resistors, and capacitors are intricately wired together. Such circuits can be represented and processed within a computer in order to answer simple questions like "Is everything connected together?" as well as complicated questions like "If this circuit is built, will it work?"

A graph is a mathematical object that accurately models such situations. In this unit, we will examine some basic properties of graphs and we will study a variety of algorithms for answering questions of the type posed above.

4.1 OBJECTIVES

At the end of the Unit, you shall be able to:

- define the terms related to graphs e.g. directed graph, vertex, edge, adjacency list etc.
- represent graphs using adjacency matrix and adjancency list structures
- differentiate between a depth-first and breadth-first search traversals
- find the shortest path between two vertices of a given graph
- build a minimal spanning tree for a given graph.

4.2 DEFINING GRAPH

A Graph G consists of a set V of vertice (nodes) and a set E of edges (arcs). We write G=(V,E). V is a finite and non empty set of vertices. E is a set of pairs of vertices; these pairs are called edges. Therefore

V(G), read as V of G, is set of vertices,

and E(G), read as E of G, is set of edges.

An edge e = (v,w), is a pair of vertices v and w, and is said to be incident with v and w.

A graph may be pictorically represented as given in Figure 1.

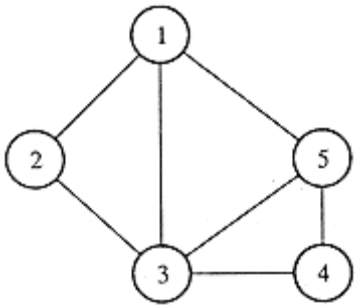


Figure 1 : A Graph

We have numbered the nodes as 1,2,3,4 and 5. Therefore

V(G) = (1,2,3,4,5)

and E(G) = {(1,2), (2,3), (3,4), (4,5), (1,5), (1,3), (3,5) }

You may notice that the edge incident with node 1 and node 5 is written as (1,5); we could also have written (5,1) instead of (1,5). The same applies to all the other edges. Therefore, we may say that ordering of vertices is not significant here. This is true for an undirected graph.

In an undirected graph, pair of vertices representing any edge is unordered. Thus (v,w) and (w,v) represent the same edge. In a directed graph each edge is an ordered pair of vertices, i.e. each edge is represented by a directed pair. If e = (v,w), then v is tail or initial vertex and w is head or final vertex. Subsequently (v,w) and (w,v) represent two different edges.

A directed graph may be pictorically represented as given in Figure 2.

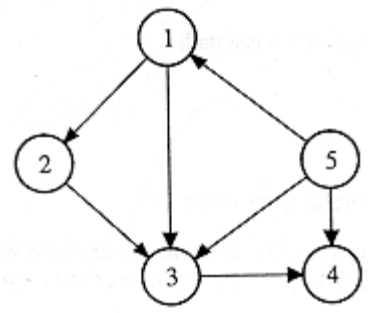


Figure 2 : A Directed Graph

The direction is indicated by an arrow. The set of vertices for this graph remains the same as that of the graph in the earlier example, i.e.

V(G) = (1,2,3,4,5}

However the set of edges would be

E(G) = {(1,2), (2,3), (3,4), (5,4), (5,1), (1,3), (5,3)}

Do you notice the difference? Also note that arrow is always from tail vertex to head vertex. In our further discussion on graphs, we would refer to directed graph as digraph and undirected graph as graph .

Check Your Progress 1

Question 1: Three buildings X, Y and Z have to connect to the Water, Electricity and Telephone supplied: W, E and T. Construct a pictorial representation of it. Can you find a picture, in which the edges do not cross?

Question 2: Draw a graph with six vertices and four edges?

4.3 BASIC TERMINOLOGY

A good deal of nomenclature is associated with graphs. Most of the terms have straight foward definitions, and it is convenient to put them in one place even though we would not be using some of them until later.

Let us define few more terms.

Adjacent vertices Vertex v 1 is said to be adjacent to a vertex v 2 if there is an edge

(v1 , v2) or (v2 , v1)

Let us consider the graph in figure 3.

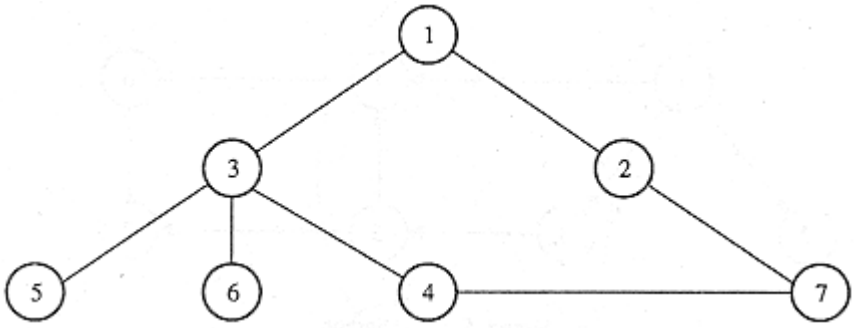


Figure 3

Vertices adjacent to node 3 are 1,5,6 and 4 and that to node 2 are 1 and 7.

Find out the vertices adjacent to remaining nodes of the graph.

Path: A path from vertex v to vertex w is a sequence of vertices, each adjacent to the next.

Consider the above example again. 1,3,4 is a path

1,3,6 is a path
1,2,7 is a path ,
Is 1,4,7 a path?

How many paths are there from vertex 1 to vertex 7?

You may notice that there is a path existing in the above example which starts at vertex 1 and finishes at vertex 1, i.e. path 1,3,4,7,2,1. Such a path is called a cycle.

A cycle is a path in which first and last vertices are the same.

DO we have a path from any vertex to any other vertex in the above example? If you see it carefully, you may find the answer to the above question as YES. Such a graph is said to be connected graph. A graph is called connected if there exists a path from any vertex to any other vertex. There are graphs which are unconnected. Consider the graph in figure 4.

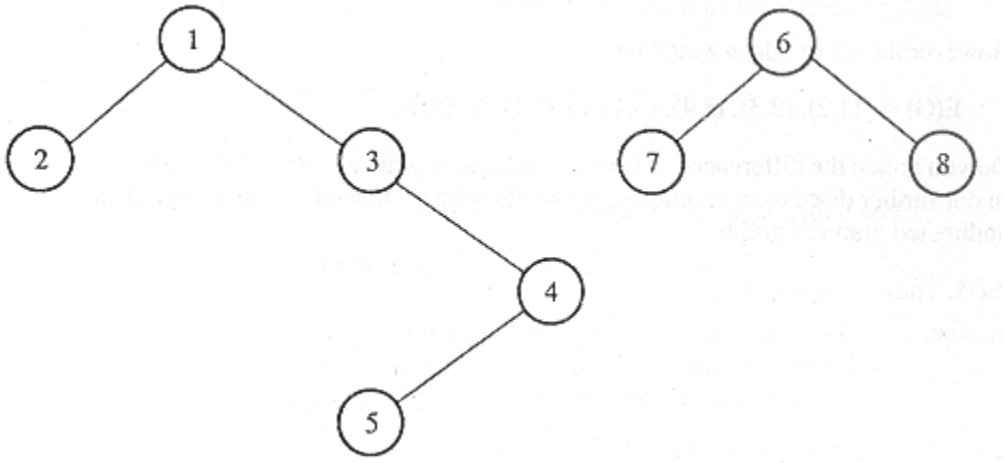


Figure 4 : An Unconnected Graph

It is an unconnected graph. You may say that these are two graphs and not one. Look at the figure in its totality and apply the definition of graph. Does it satisfy the definition of a graph? It does. Therefore, it is one graph having two unconnected components. Since there are unconnected components, it is an unconnected graph.

So far we have talked of paths, cycles and connectivity of undirected graph. In a Digraph the path is called a directed path and a cycle as directed cycle.

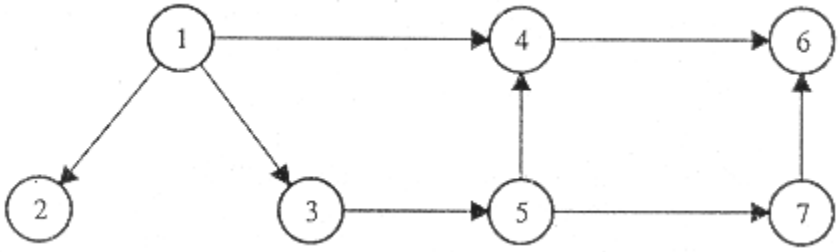


Figure 5 : A Digraph

In Figure 5 1,2 is a directed path; 1,3,5,7,6 is a directed path 1,4,5 is not a directed path.

There is no directed cycle in the above graph. You may verify the above statement. A digraph is called strongly connected if there is a directed path from any vertex to any other vertex.

Consider the digraph given in Figure 6.

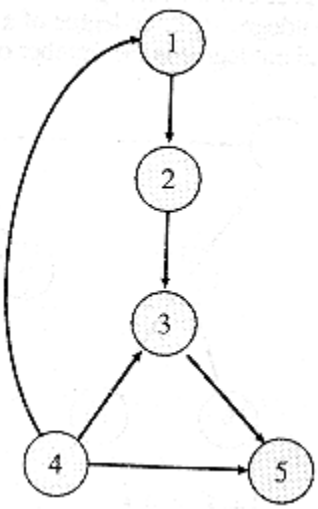


Fig. 6 : A Weakly Connected Graph

There does not exist a directed path from vertex 1 to vertex 4; also from vertex 5 to other vertices; and so on. Therefore, it is a weakly connected graph. Let us make is strongly connected.

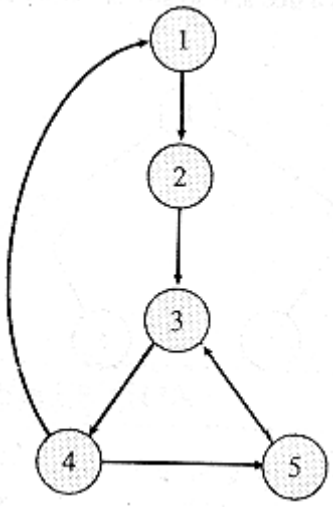


Fig. 7 : A Strongly Connected Graph

The, graph in Figure 7 a strongly connected graph. You may notice that we have added just one arc from vertex 5 to vertex 3.

An alternative could be as given in Figure 8.

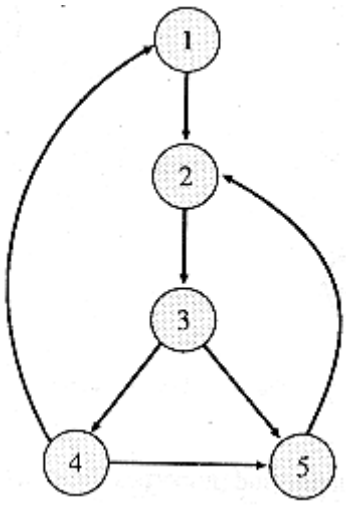


Fig. 8 : A Strongly Connected Graph

There may exist more alternate structures. Make at least one more alternate structure for the same diagraph.

You must have observed that there is no limitation of number of edges incident on one vertex. It could be none, one, or more. The number of edges incident on a vertex determines its degree. Thus in Figure 3 the degree of vertex 3 is 4.

In a digraph we attach an indegree and an outdegree to each of the vertice. In Figure 8, the indegree of vertex 5 is 2 and outdegree is 1. Indegree of a vertex v is the number of edges for which vertex v is a head and outdegree is the number of edges for which vertex is a tail.

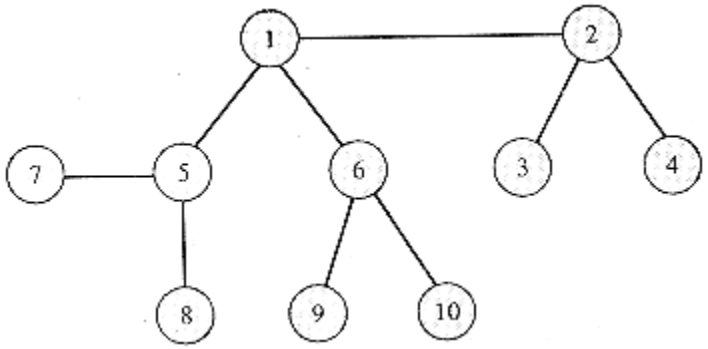


Fig. 9 : A Tree

Let us define a special type of graph called Tree. A graph is a tree if it has two properties:

It is connected, and
there are no cycles in the graph.

Graph depicted in Figure 9 is a tree and so are the ones depicted in Figure 10(a) to 10(e).

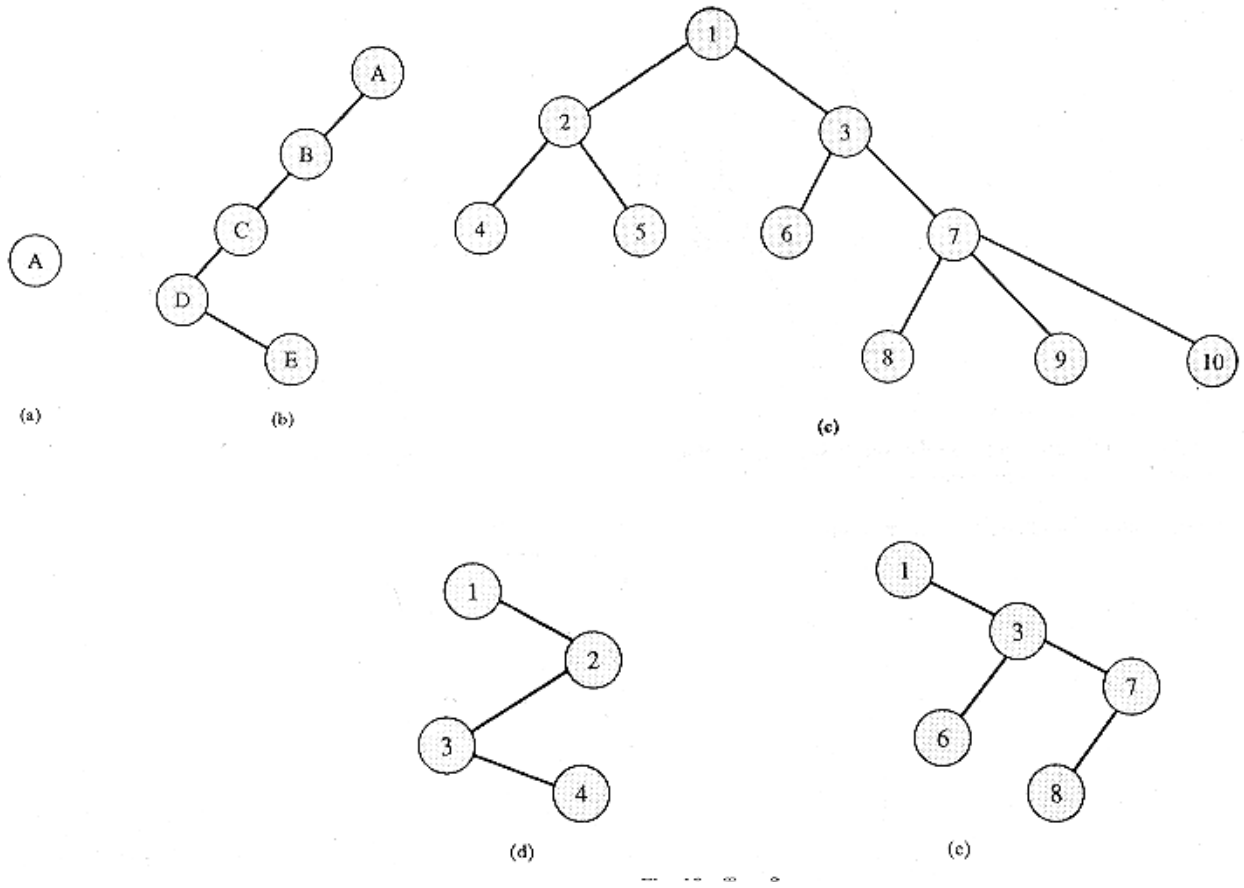
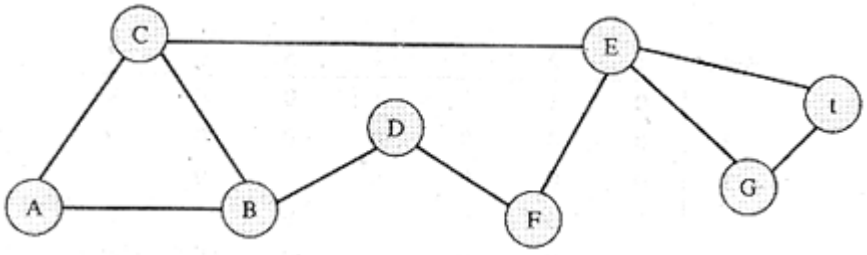


Fig. 10 : Tree Structure

Because of their special structure and properties, trees occur in many different applications in computer science.

Check Your Progress 2

Question 1: Consider the graph given in figure below.

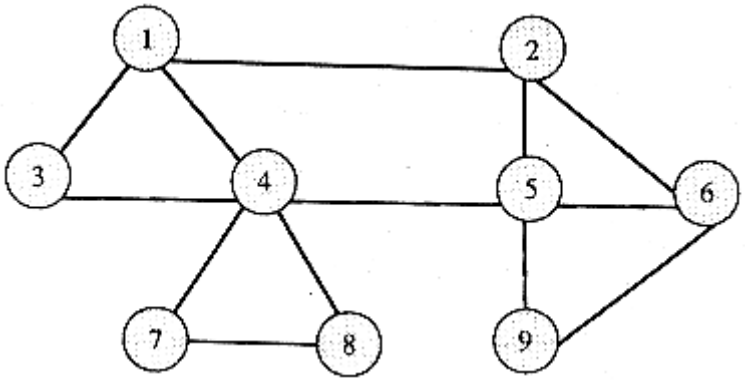


1. Is there a path from A to Z that passes through each vertex exactly once?

2. What is the maximum number of vertices one can visit once in going from A to Z?

Question 2:

1. Find the degree of each vertex of the graph given below:



Is it possible to trace the above graph, beginning and ending at the same point, without lifting your pencil from the paper and without retracing lines? Justify your answer.

Question3: Give an example of a connected graph such that removal of any edge results in a graph that is not connected.

Question 4: Draw a graph with five vertices each of degree 3.

4.4 GRAPH REPRESENTATION

Graph is a mathematical structure and finds its application in many areas of interest in which problems need to be solved using computers. Thus, this mathematical structure must be represented as some kind of data structures. Two such representations are, commonly used. These are Adjacent Matrix and Adjacency List representation. The choice of representation depends on the application and function to be performed on the graph. Let us learn more about these representations.

Adjacency Matrix The adjacency matrix A for a graph G = (V,E) with n vertices, is an nxn matrix of bits, such that A

$_{ij}$ = 1, iff there is an edge from vi to vj and
A_{ij} = 0, if there is no such edge.

Figure 11 (a) shows the adjacency matrix for the graph given in Figure 1.

Vertice	1	2	3	4	5
1	0	1	1	0	1
2	1	0	1	0	0
3	1	1	0	1	1
4	0	0	1	0	1
5	1	0	1	1	0

Figure 11: Adjacency Matrix for the Graph In Figure 1

You may observe that the adjacency matrix for an undirected graph is symmetric, as the lower and upper triangles are same. Also all the diagonal elements are zero., since we consider graphs without any self loops. Let us find adjacency matrix for a digraph given in figure 5.

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

Figure 12. Adjacency Matrix for Digraph in Figure 5

The total number of 1's account for the number of edges in the digraph. The number of 1's in each row tells the outdegree of the corresponding vertex.

We can see that in this representation, we need n2 bits to represent a graph with n nodes. Further adjacency matrix can be implemented as a 2-dimensional array of type boolean. We may have a declaration like

```
Type
vertex\      =    1.. max;
Adjmat       =    array [vertex, vertex] of boolean;
Graph        =    record
                  n: 0.. max;
                  A: adjrnat;
                  end;
```

where, n is the number of vertices in the graph and

```
A[ v,w]      =    True    iff v is adjacent to w
             =    False    if v is not adjacent to w
```

The adjacency matrix is a simple way to represent a graph, but it has two disadvantages.

1. It takes 0(n2) space to represent a graph with n vertices; even for sparse graphs, and

2. It takes .... (n2) time to solve most of the graph problems.

Let us see how the above two disadvantages are dealt with in adjacency list representation.

Adjacency List Representation

In this representation, we store a graph as a linked structure. We store all the vertices in a list and then for each vertex, we have a linked list of its adjacent vertices. Let us see it through an example. Consider the graph given in Figure 13.

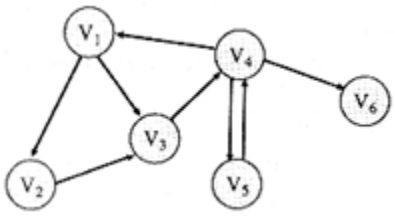
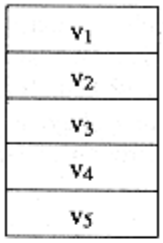


Figure 13

The adjacency list representation needs a list of all of its nodes, i.e.



and for each node a linked list of its adjacent nodes.

Therefore we shall have

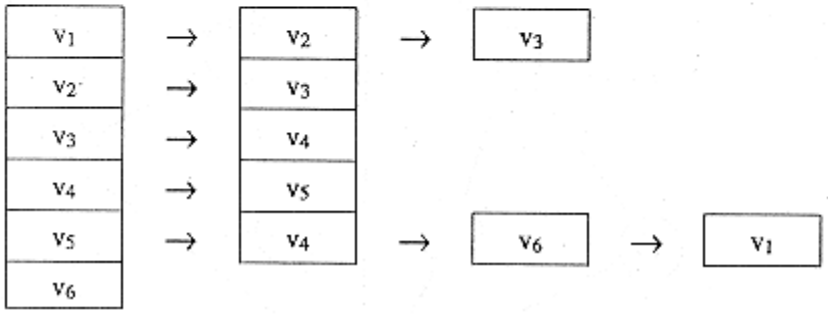


Figure 14: Adjacency List Structure for Graph in Figure 13.

Note that adjacent vertices may appear in the adjacency list in arbitrary order. Also an arrow from v2 to v3 in the list linked to v1 does not mean that V2 and V3 are adjacent.

The adjacency list representation is better for sparse graphs because the space required is O(V + E), as contrasted with the O(V2) required by the adjacency matrix representation.

Check Your Progress 3

Question 1: What must a graph look like if a row of its adjacency matrix consists only of zeros?

Question 2: Write the adjacency matrix of the following graphs

4.5 GRAPH TRAVERSAL

A graph traversal means visiting all the nodes of the graph. Graph traversal may be needed in many of application areas and there may be many methods for visiting the vertices of the graph. Two graph traversal

methods, which are being discussed in this section are the commonly used methods and are also found to be efficient graph traversal methods. These are

Depth First Search or DFS; and

Breadth First Search or BFS

4.5.1 Depth First Search (DFS)

In graphs, we do not have any start vertex or any special vertex singled out to start traversal from. Therefore the traversal may start from any arbitrary vertex.

We start with say, vertex v. An adjacent vertex is selected and a Depth First Search is initiated from it, i.e. let V_1, V_2 .. V_k are adjacent vertices to vertex v. We may select any vertex from this list. Say, we select v_1 . Now all the adjacent vertices to v1 are identified and all of those are visited; next V2 is selected and all its adjacent vertices visited and so on. This process continues till all the vertices are visited. It is very much possible that we reach a traversed vertex second time. Therefore we have to set a flag somewhere to check if the vertex is already visited. Let us see it through an example. Consider the following graph.

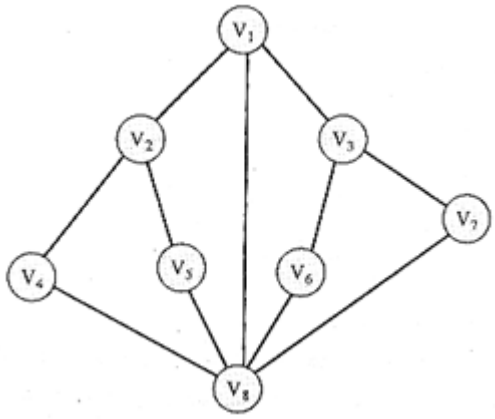


Fig 15 : Example Graph for DFS

Let us start with v_1.

Its adjacent vertices are v_2, v_8 and V_3. Let us pick on V_2.

Its adjacent vertices are v_1 ,v_4, v_5. v_1 is already visited. Let us pick on V_4.

Its adjacent vertices are V_2, V_8.

v_2 is already visited. Let us visit v_8. Its adjacent vertices are V_4, V_5, V_1, V_6, V_7.

V_4 and v_1, are already visited. Let us traverse V_5.

Its adjacent vertices are v_2, v_8. Both are already visited Therefore, we back track.

We had V_6 and V_7 unvisited in the list of v_8. We may visit any. We visit v_6.

Its adjacent are v_8 and v_3. Obviously the choice is v_3.

Its adjacent vertices are v_1, v_7 . We visit v_7.

All the adjacent vertices of v_7 are already visited, we back track and find that we have visited all the vertices.

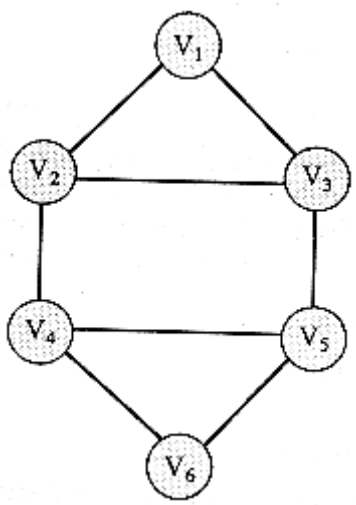


Fig. 16 : Example Graphs for DFS

Therefore the sequence of traversal is

$v_1, v_2, v_4, v_8, v_5, v_6, v_3, v_7$.

This is not a unique or the only sequence possible using this traversal method.

Let us consider another graph as given in Figure 16.

Is $v_1, v_2, v_3, v_5, v_4, v_6$ a traversed sequence using DFS method?

We may implement the Depth First Search method by using a stack. pushing all unvisited vertices adjacent to the one just visited and popping the stack to find the next vertex to visit.

Implementation

We use an array **val[V]** to record the order in which the vertices are visted. Each entry in the array is initialized to the value unseen to indicate that no vertex has yet been visted. The goal is to systematically visit all the vertices of the graph, setting the val entry for the ith vertex visited to i, for i = 1, 2..... ,V. The following program uses a procedure visit that visits all the vertices in the same connected component as the vertex given in the argument.

```
void search( )
        {
                int k;
                for (k = 1; k V; k++) val[k] = unseen;
                for (k = 1; k V; k++)
                   if (val[k] == unseen) visit(k);
        }
```

The first for loop initializes the val array. Then, visit is called for the first vertex, which results in the val values for all the vertices connected to that vertex being set to values different from **unseen**. Then **search**

scans through the val array to find a vertex that hasn't been seen yet and calls visit for that vertex, continuing in this way until all vertices have been visited. Note that this method does not depend on how the graph is represented or how **visit** is implemented.

First we consider a recursive implementation of visit for the adjacency list representation: to visit a vertex, we check all its edges to see if they lead to vertices that have not yet been seen; if so, we visit them.

```
void visit (int k) // DFS, adjacency lists

{

        struct node *t;
        val[k] = ++i;
        for (t = adj [k]; t ! = z; t = t-next)
           if (val[t-vl == unseen) visit (t-v);
}
```

We move to a stack-based implementation:

```
Stack stack(maxV);
void visit(int k) // non-recursive DFS, adjacency lists
{
struct node *t
stack.push(k);
while (!stack.empty ( ))
{
        k = stack.pop(); val[k] = ++id;
        for (t = adj[k]; t != z; t = t-next)
        if (val[t-v} == unseen)
        {stack.push(t-v); val[t-v] = 1;}
```

Vertices that have been touched but not yet visited are kept on a stack. To visit a vertex, we traverse its edges and push onto the stack any vertex that has, not yet been visited and that is not already on the stack. In the recursive implementation, the bookkeeping for the "partially visited" vertices is hidden in the local variable t in the recursive procedure. We could implement this directly by maintaining pointers (corresponding to t) into the adjacency lists, and so on.

Depth-first search immediately solves some basic graph- processing problems. For example, the procedure is based on finding the connected components in turn; the number of connected components is the number of times visit is called in the last line of the program. Testing if a graph has a cycle is also a trivial modification of the above program. A graph has a cycle if and only if a node that is not unseen is discovered in visit. That is, if we encounter an edge pointing to a vertex that we have already visited, then we have a cycle.

4.5.2 Breadth First Search (BFS)

In DFS we pick on one of the adjacent vertices; visit all of its adjacent vertices and back track to visit the unvisited adjacent vertices. In BFS, we first visit all the adjacent vertices of the start vertex and then visit all -the unvisited vertices adjacent to these and so on. Let us consider the same example, given in Figure l5. We start say, with v_1. Its adjacent vertices are V_2, V_8, V_3. We visit all one by one. We pick on one of these, say v_2. The unvisited adjacent vertices to v_2 are v_4, v_5. We visit both. We go back to the remaining visited vertices of v_1 and pick on one of those, say v_3. The unvisited adjacent vertices to v_3 are v_6, v_7. There are no more unvisited adjacent vertices of v_8, v_4, v_5, v_6 and v_7.

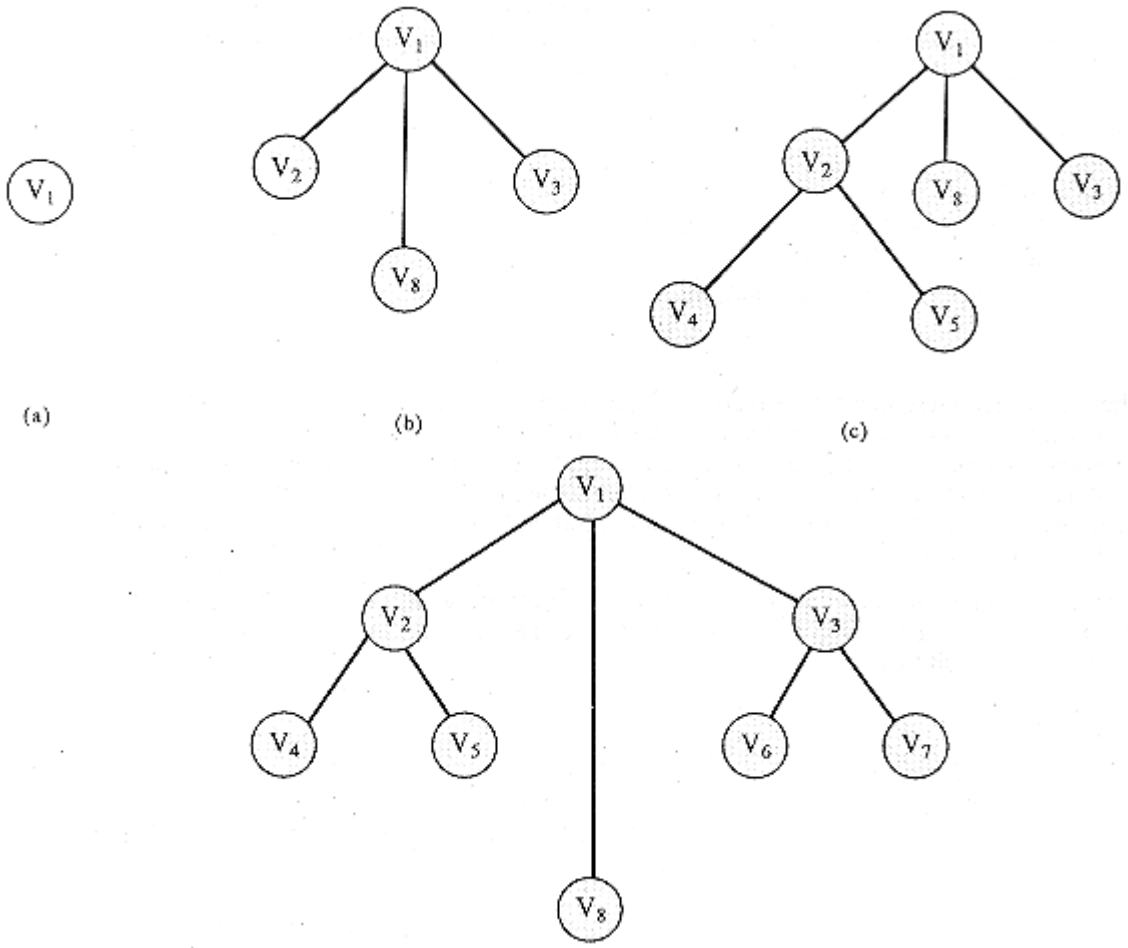


Figure 17:

Thus, the sequence so generated is v_1, v_2, v_8, v_3, v_4, v_5, v_6, v_7. Here we need a queue instead of a stack to implement it. We add unvisited vertices adjacent to the one just visited at the rear and read at from to find the next vertex to visit.

To implement breadth-first search, we change stack operations to queue operations in the stack-based search program above:

```
Queue queue(maxV);
void visit (int k) // BFS, adjacency lists
        {
                struct node *t;
                queue.put (k);
                while (!queue.empty)))
                        {
                         k = queue.geto; val[k] = ++id;
                         for (t = adj[k]; t ! = z; t = t-ncxt)
                         if (val[t-vl == unseen)
                           ( queue.put(t-v); val[t-vl = 1;}
                        }
        }
```

The contrast between depth-first and breadth-first search is quite evident when we consider a larger graph.

In both cases, the search starts at the node at the bottom left. Depth-first search wends it way through the graph, storing on the stack the points where other paths branch off; breadth-first search "sweeps through" the graph, using a queue to remember the frontier of visited places. Depth-first search "explores" the graph by looking for new vertices far away from the start point, taking closer vertices only when dead ends are encountered; breadth-first search completely covers the area close to the starting point, moving farther away only when everything close has been looked at. Again, the order in which the nodes are visited depends largely upon the order in which the edges appear in the input and upon the effects of this ordering on the order in which vertices appears on the adjacency lists.

Depth-first search was first stated formally hundreds of years ago as a method for traversing maxes. Depth-first search is appropriate for one person looking for something in a maze because the "next place to look" is always close by; breadth-first search is more like a group of people looking for something by fanning out in all directions.

4.6 SHORTEST PATH PROBLEM

We have seen in the graph traversals that we can travel through edges of the graph. It is very much likely in applications that these edges have some weights attached to it. This weight may reflect distance, time or some other quantity that corresponds to the cost we incur when we travel through that edge. For example, in the graph in Figure 18, we can go from Delhi to Andaman Nicobar through Madras at a cost of 7 or through Calcutta at a cost of 5. (These numbers may reflect the airfare in thousands.) In these and many other applications, we are often required to find a shortest path, i.e a path having the minimum weight between two vertices. In this section, we shall discuss this problem of finding shortest path for directed graph in which every edge has a non-negative weight attached.

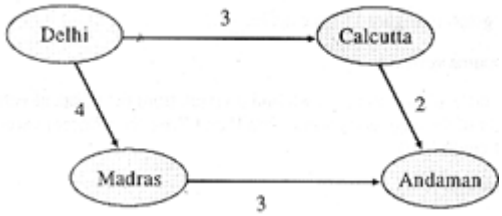


Figure 18: A graph connecting four cities

Let us at this stage recall how do we define a path. A path in a graph is sequence of vertices such that there is an edge that we can follow between each consecutive pair of vertices. Length of the path is the sum of weights of the edges on that path. The starting vertex of the path is called the source vertex and the last vertex of the path is called the destination vertex. Shortest path from vertex v to vertex w is a path for which the sum of the weights of the arcs or edges on the path is minimum.

Here you must note that the path that may look longer if we see the number of edges and vertices visited, at times may be actually shorter costwise.

Also we may have two kinds of problems in finding shortest path. One could be that we have a single source vertex and we seek a shortest path from this source vertex v to every other vertex of the graph. It is called single source shortest path problem.

Consider the weighted graph in Figure 19 with 8 nodes A,B,C,D,E,F,G and H.

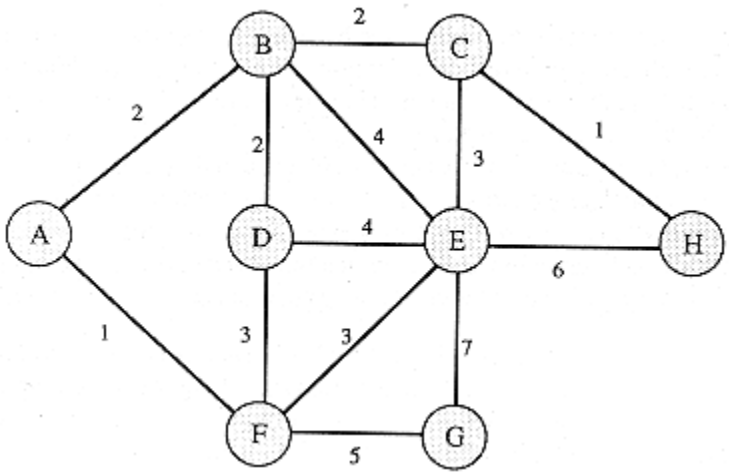


Fig. 19 : A Weighted Graph

There are many paths from A to H.

Length of path AFDEH = 1 + 3 + 4 + 6 = 14

Another path from A to H is ABCEH. Its length is 2+2+3+6 = 13.

We may further look for a path with length shorter than 13, if exists. For graphs with a small number of vertices and edges, one may exploit all the permutations combinations to find shortest path. Further we shall have to exercise this methodology for finding shortest path from A to all the remaining vertices. Thus, this method is obviously not cost effective for even a small sized graph.

There exists an algorithm for solving this problem. It works like as explained below:

Let us consider the graph in Figure 18 once again.

1. We start with source vertex A.

2. We locate the vertex closest to it, i.e. we find a vertex from the adjacent vertices of A for which the length of the edge is minimum. ere B and F are the adjacent vertices of A and Length (AB) Length (AF)

   Therefore we choose F.

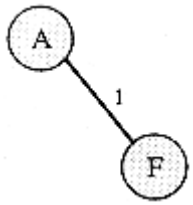


Fig. 20 (a)

3. Now we look for all the adjacent vertices excluding the just earlier vertex of newly added vertex and the remaining adjacent vertices of earlier vertices, i.e. we have D,E and G (as adjacent vertices of F) and B (as remaining adjacent vertex of A).

   Now we again compare the length of the paths from source vertex to these unattached vertices, i.e. compare length (AB), length (AFD), length (AFG) and length (AFE). We find the length (AB) the minimum. There we choose vertex B.

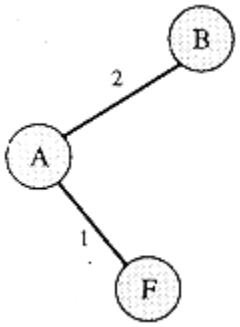


Fig. 20 (b)

4. We go back to step 3 and continue till we exhaust all the vertices.

Let us see how it works for the above example.

Vertices that may be attached	Path from A	Length
	ABD	4
D		
	AFD	4
G	AFG	6
C	ABC	4
	AFE	4
E	ABE	6

We may choose D. C or E.
We choose say D through B.

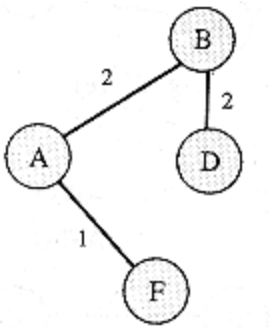


Fig. 20 (g)

□G	AFG	6
C	ABC	4
E	AFE	4
	ABE	6
	BDE	8

We may choose C or E

We choose say C

G	AFG	6
E	AFE	4
H	ABE	6
	ABDE	8
	ABCE	7

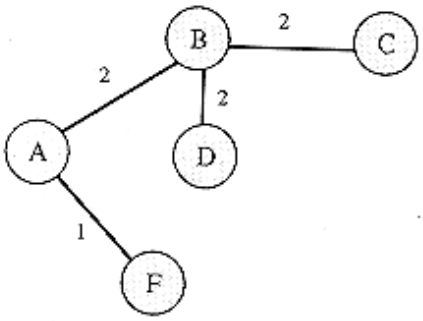


Fig. 20 (d)

	ABCM	5

We choose E via AFE

G	AFG	6
H	AFE	11
	ABCH	5
	AFEH	10

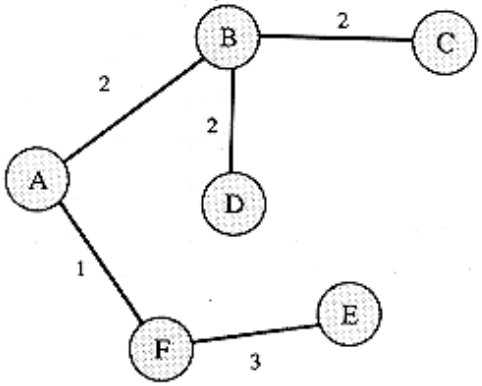
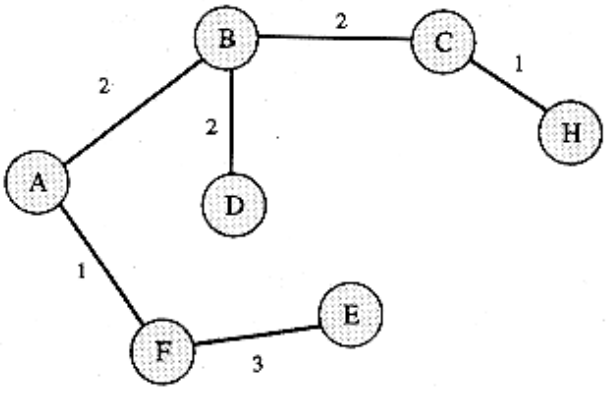


Fig. 20 (e)

We choose H via ABCH

G	AFG	6
	AFEG	11



We choose path AFG

Therefore the shortest paths from source vertex A to all the other vertices are

AB
ABC
DB
AFE
F
AFG
ABCH

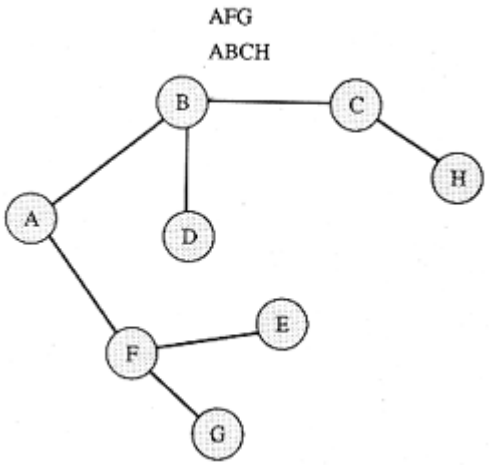
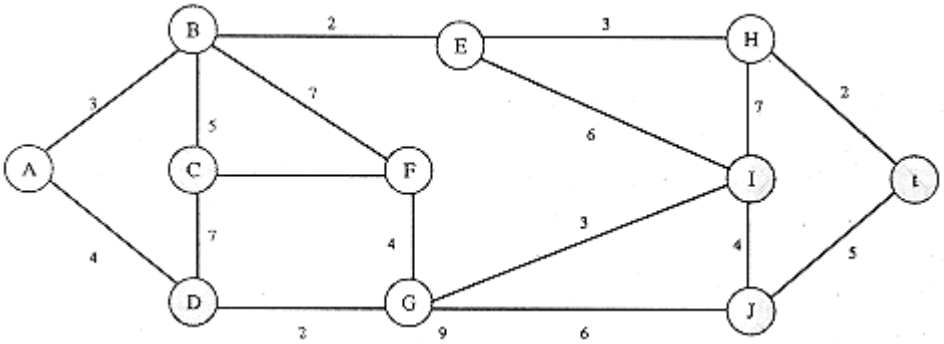


Fig. 20 (f)

Check Your Progress 4

Question 1: Given a graph



Find the length of a shortest path and a shortest path between

(i) A and F

(ii) A and Z

(iii) D and H

Exercises:

Question 1: Write a program that accepts adjacency matrix as an input and outputs a list of edges given as pairs of positive integers.

Question 2: Write a program that accepts an input the edges of a graph and determines whether or not the graph contains an Euler circuit.

4.7 MINIMAL SPANNING TREE

You have already learnt in Section 4.3 that a tree is. a connected acyclic graph. If we are given a graph G = (V,E), we may have more than one V tree structures. Let us see what do we mean by this statement. Consider the graph given in Figure 21 some of the tree structures for this graph are given in Figure 22(a), 22(b) and 22(c).

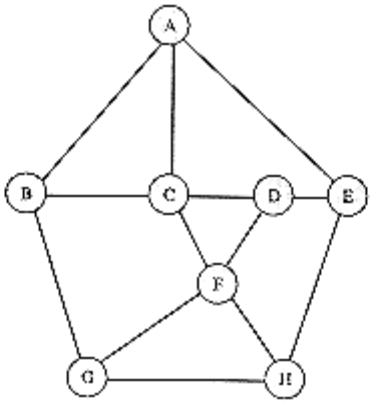


Fig. 21

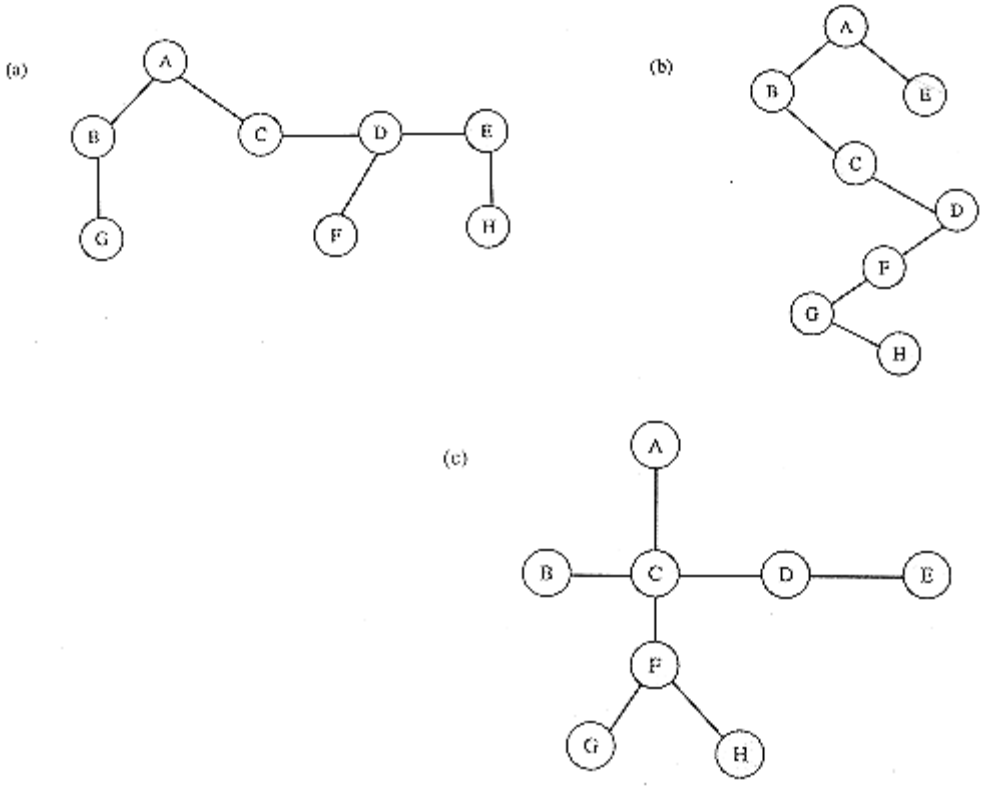


Fig.22

You may notice that they differ from each other significantly, however, for each structure

(i) the vertex set is same as that of Graph G

(ii) the edge set is a subset of G(E); and

(iii) there is no cycle.

Such a structure is called spanning tree of graph. Let us formally define a spanning tree. A tree T is a spanning tree of a connected graph G(V,E) such that

1) every vertex of G belongs to an edge in T and

2) the edges in T form a tree.

Let us see how we construct a spanning tree for a given graph. Take any vertex v as an initial partial tree and add edges one by one so that each edge joins a new vertex to the partial tree. In general if there are n vertices in the graph we shall construct a spanning tree in (n- 1) steps i.e. (n- 1) edges are needed to added.

Frequently, we encounter weighted graphs and we need to built a subgraph that must be connected and must include every vertex in the graph. To construct such a subgraph with least weight or least cost, we must not have cycles in it. Therefore, actually we need to construct a spanning tree with minimum cost or a **Minimal Spanning Tree**.

You must notice the difference between the shortest path problem and the Minimal Spanning tree problem.

Let us see this difference through an example. Consider the graph given in Figure 23.

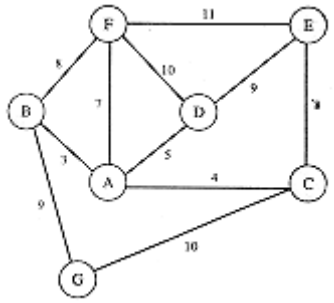


Figure 23

This could, for instance, represent the feasible communication lines between 7 cities and the cost of an edge could be interpreted as the actual cost of building that link (in lakhs of rupees). The Minimal Spanning Tree problem for this situation could be the building a least cost communication network. Ale shortest path problem (One source - all definitions) could be identifying one city and finding out the least cost communication lines from this city to all the other cities. Therefore, Shortest Path trees are rooted, while MST are free trees. Also MSTs are defined only on undirected graphs.

Building a Minimum Spanning Tree

As stated in an earlier paragraph, we need to select n-1 edges in G (of n vertices) such that these form an MST o^ G. We begin by first selecting an edge with least cost. It can between any two vertices of G. Subsequently, from the set of remaining edges, we can select another least cost edge and so on. Each time an edge is picked, we determine whether or not the inclusion of this edge into the spanning tree being constructed creates a cycle. If it does, this edge is discarded. If no cycle is created, this edge is included in the spanning tree being constructed.

Let us work it out through an example. Consider the graph given in Figure 23. The minimum cost edge is that connects vertex A and B. Let us choose that.

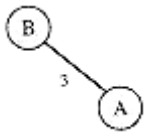


Figure 24(a)

Now we have

Vertices that may be added	**Edge**	**Cost**
F	BF	8
	AF	7
G	BG	9
C	AC	4
D	AD	5

The least cost edge is AC; therefore we choose AC

Now we have

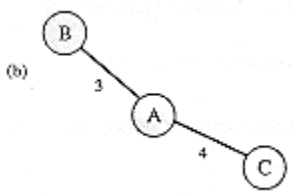


Figure 24 (b)

Vertices that may be added	**Edge**	**Cost**
F	BF	8
	AF	7
G	BG	9
	CG	10
D	AD	5
E	CE	8

The least cost edge is Ad; therefore we choose ad.

Now we have

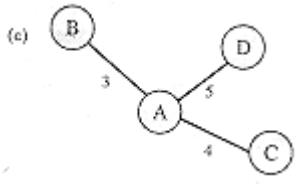


Figure 24 (c)

Vertices that may be added	Edge	Cost
F	BF	8
	AF	8
	DF	10
G	BG	9
	CG	10
E	CE	8
	DE	9

AF is the minimum cost edge; therefore, we add it to the partial tree.

Now we have

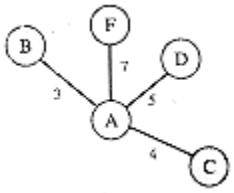


Figure 24 (d)

Vertices that may be added	Edge	Cost
G	BG	9
	CG	10
E	CE	8
	DE	9
	FE	11

Obvious choice is CE

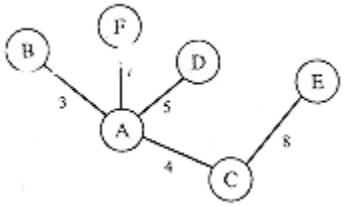


Figure 24 (e)

The only vertex left is G and we have the minimum cost edge that connects it to the tree constructed so far is BG. Therefore we add it and the minimal spanning tree constructed would be of the costs 9+3+7+5+4+8 = 36; and is given in Figure 24(f).

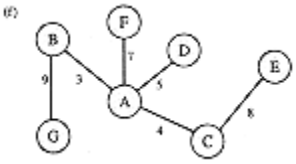
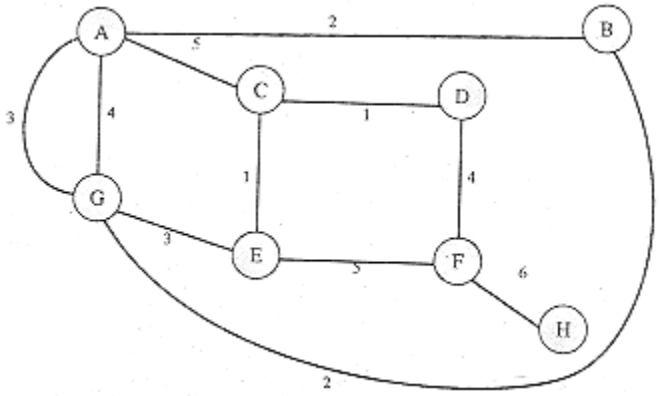


Figure 24 (f)

This method is called the Kruskal's method of creating a minimal spanning tree.

Check Your Progress 5

Question 1: Draw Minimum Cost Spanning Tree for the graph given below and also find its cost.



4.8 SUMMARY

Graphs provide in excellent way to describe the essential features of many applications. Graphs are mathematical structures and are found to be useful in problem solving. They may be implemented in many ways by the use of different kinds of data structures. Graph traversals, Depth First as well as Breadth First, are also required in many applications. Existence of cycles makes graph traversal challenging, and this leads to finding some kind of acyclic subgraph of a graph. That is we actually reach at the problems of finding a shortest path and of finding a minimum cost spanning tree problems.

In this Unit, we have built up the basic concepts of the graph theory and of the problems, the solutions of which may be programmed.

Some graph problems that arise naturally and are easy to state seem to be quite difficult, and no good algorithms are known to solve them. For example, no efficient algorithm is known for finding the minimum-cost tour that visits each vertex in a weighted graph. This problem, called the travelling salesman problem, belongs to a large class of difficult problems.

Other graph problems may well have efficient algorithms, though none have been found. An example of this is the graph isomorphism problem. Determine whether two graphs could be made identical by renaming vertices. Efficient algorithms are known for this problem for many special types of graphs, but the general problem remains open. In short, there is a wide spectrum of problems and algorithms for dealing with graphs. But many relatively easy problems do arise quite often, and the graph algorithms we studied in this unit serve well in a great variety of applications.

Exercises

Question 1: Write a program that accepts adjacency matrix as an input and output a list of edges given as pairs of positive integers.

Question 2: Write a program that accepts as input the edges of a graph and determines whether or not the graph contains an Euler circuit.