UNIT IV GRAPHS 9

Graph – Definition and Terminology – Representation of Graphs - Types of Graph – Graph Traversal: Breadth -first Search, Depth-first Search - Connectivity, strong connectivity – Topological Sort - Minimum Spanning Tree: prim's algorithm, Kruskal's algorithm- Single Source Shortest Path: Dijkstra's algorithm.

4.1 DEFINITION

A graph G = (V, E) consists of a set of vertices, V and set of edges E.

Vertics are referred to as nodes and the arc between the nodes are referred to as Edges. Each edge is a pair (v, w) where v, w $\in$ V. (i.e.) v = V_1, w = V_2...

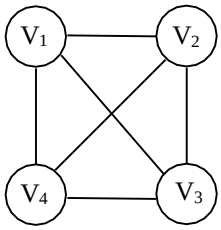


Fig. 4.1

Here V_1, V_2, V_3, V_4 are the vertices and (V_1, V_2), (V_2, V_3), (V_3, V_4), (V_4, V_1), (V_2, V_4), (V_1, V_3) are edges.

4.2 REPRESENTATION OF GRAPH

Graph can be represented by Adjacency Matrix and Adjacency list.

One simple way to represents a graph is Adjacency Matrix.

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

A_{ij} = 1, if there is an edge V_i to V_j

A_{ij} = 0, if there is no edge.

Adjacency Matrix For Directed Graph

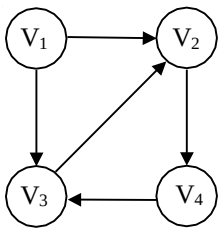


Fig. 4.2.1

	V_1	V_2	V_3	V_4
V_1	0	1	1	0
V_2	0	0	0	1
V_3	0	1	0	0
V_4	0	0	1	0

Fig. 4.2.2

Example $V_{1,2} = 1$ Since there is an edge V_1 to V_2

Similarly $V_{1,3} = 1$, there is an edge V_1 to V_3

$V_{1,1}$ & $V_{1,4} = 0$, there is no edge.

Adjacency Matrix For Undirected Graph

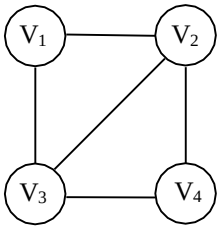


Fig. 4.2.3

	V_1	V_2	V_3	V_4
V_1	0	1	1	0
V_2	1	0	1	1
V_3	1	1	0	1
V_4	0	1	1	0

Fig. 4.2.4

Adjacency Matrix For Weighted Graph

To solve some graph problems, Adjacency matrix can be constructed as

$A_{ij} = C_{ij}$, if there exists an edge from V_i to V_j

$A_{ij} = 0$, if there is no edge & i = j

If there is no arc from i to j, Assume C[i, j] = ∞ where $i \neq j$.

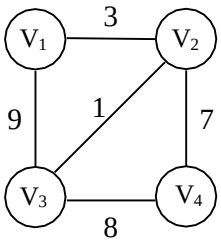


Fig. 4.2.5

	V_1	V_2	V_3	V_4
V_1	0	3	9	
V_2		0		7
V_3		1	0	
V_4		1	8	0

Fig. 4.2.6

Advantage

* Simple to implement.

Disadvantage

* Takes $O(n^2)$ space to represents the graph

* It takes $O(n^2)$ time to solve the most of the problems.

Adjacency List Representation

In this representation, we store a graph as a linked structure. We store all vertices in a list and then for each vertex, we have a linked list of its adjacency vertices

Adjacency List

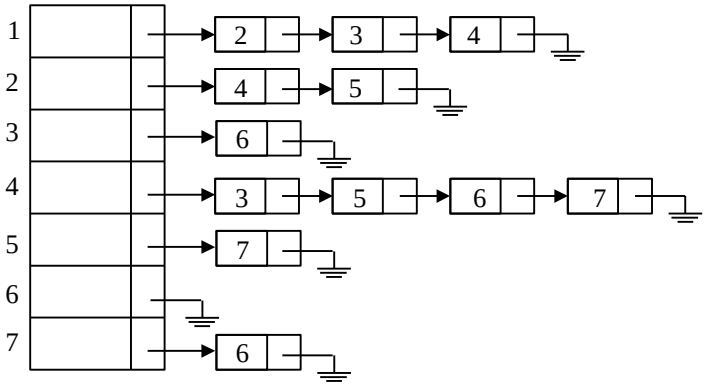
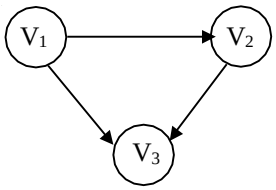


Fig. 4.2.7

4.3 TYPES OF GRAPH

Directed Graph (or) Digraph

Directed graph is a graph whichconsists of directed edges, where each edge in E is unidirectional. It is also referred as Digraph. If (v, w) is a directed edge then (v, w) # (w, v)

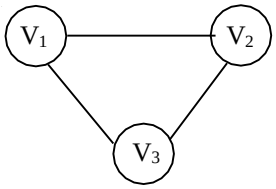


$(V_1, V_2) \neq (V_2, V_1)$

Fig. 4.3.1

Undirected Graph

An undirected graph is a graph, which consists of undirected edges. If (v, w) is an undirected edge then (v,w) = (w, v)



$(V_1, V_2) = (V_2, V_1)$

Fig. 4.3.2

Weighted Graph

A graph is said to be weighted graph if every edge in the graph is assigned a weight or value. It can be directed or undirected graph.

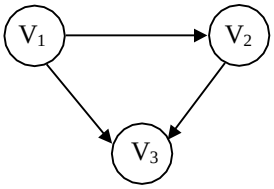


Fig. 4.3.4 (a)

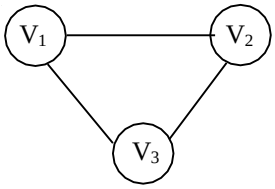


Fig. 4.3.4(b)

Complete Graph

A complete graph is a graph in which there is an edge between every pair of vertices. A complete graph with n vertices will have n (n - 1)/2 edges.

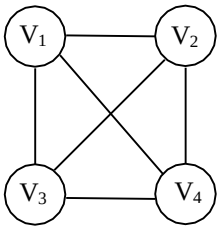


Fig. 4.3.5

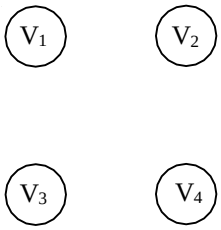


Fig. 4.3.5 (a) Vertices of a graph

In fig. 4.3.5

Number of vertics is 4

Number of edges is 6

(i.e) There is a path from every vertex to every other vertex.

A complete digraph is a strongly connected graph.

Strongly Connected Graph

If there is a path from every vertex to every other vertex in a directed graph then it is said to be strongly connected graph. Otherwise, it is said to be weakly connected graph.

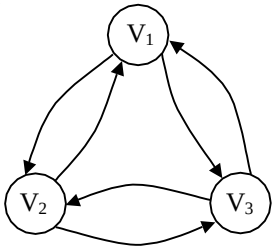


Fig. 4.3.6 Strongly Connected Graph

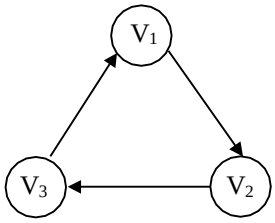


Fig. 4.3.7 Weakly Connected Graph

Path

A path in a graph is a sequence of vertices $\omega_1, \omega_2, \ldots \omega_n$ such that $\omega_i, \omega_{i+1} \in$ E for $1 \le i \le N$. Referring the Fig. 4.3.7 the path from V_1 to V_3 is V_1, V_2, V_3.

Length

The length of the path is the number of edges on the path, which is equal to N-1, where N represents the number of vertices.

The length of the above path V_1 to V_3 is 2. (i.e) (V_1, V_2), (V_2, V_3).

If there is a path from a vertex to itself, with no edges, then the path length is 0.

Loop

If the graph contains an edge (v, v) from a vertex to itself, then the path is referred to as a loop.

Simple Path

A simple path is a path such that all vertices on the path, except possibly the first and the last are distinct.

A simple cycle is the simple path of length atleast one that begins and ends at the same vertex.

Cycle

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

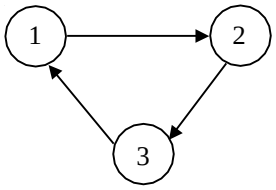


Fig. 4.3.8

A graph which has cycles is referred to as cyclic graph.

Degree

The number of edges incident on a vertex determines its degree. The degree of the vertex V is written as degree (V).

The indegree of the vertex V, is the number of edges entering into the vertex V.

Similarly the out degree of the vertex V is the number of edges exiting from that vertex V.

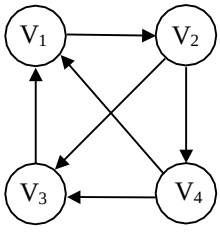


Fig. 4.3.9

In fig. 7.1.9

Indegree $(V_1) = 2$

Outdegree (V_1) = 1

ACyclic Graph

A directed graph which has no cycles is referred to as acyclic graph. It is abbreviated as DAG.

DAG - Directed Acyclic Graph.

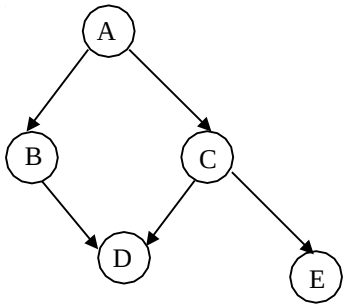


Fig. 4.3.10

4.4 GRAPH TRAVERSAL

A graph traversal is a systematic way of visiting the nodes in a specific order. There are two types of graph traversal namely,

- Depth first traversal
- Breadth first traversal

Breadth First Traversal

Breadth First Search (BFS) of a graph G starts from an unvisited vertex u. Then all unvisited vertices v_i adjacent to u are visited and then all unvisited vertices w_j adjacent to v_i are visited and so on. The traversal terminates when there are no more nodes to visit. Breadth first search uses a queue data structure to keep track of the order of nodes whose adjacent nodes are to be visited.

Steps to implement breadth first search

Step 1: Choose any node in the graph, designate it as the search node and mark it as visited.

Step 2: Using the adjacency matrix of the graph, find all the unvisited adjacent nodes to the search node and enqueue them into the queue Q.

Step 3: Then the node is dequeued from the queue. Mark that node as visited and designate it as the new search node.

Step 4: Repeat step 2 and 3 using the new search node.

Step 5: This process continues until the queue Q which keeps track of the adjacent nodes is empty.

Routine for breadth first search

```
Void BFS (vertex u)
{
   Initialize queue Q;
   visited [u] = 1;
   Enqueue (u, Q);
   while (! Isempty(Q))
      {
         u = Dequeue (Q);
         print u;
         for all vertices v adjacent to u do
            if (visited [v] = = 0) then
               {
                  Enqueue (v, Q)
                  visited [v] = 1;
               }
      }
   }
```

Example:

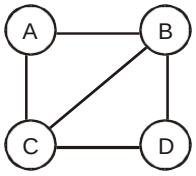


Figure 4.4.1

Adjacency matrix

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

Figure 4.4.2

Implementation

1. Let ‘A’ be the source vertex. Mark it to as visited.

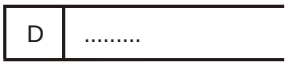
2. Find the adjacent unvisited vertices of ‘A’ and enqueue then into the queue. Here B and C are adjacent nodes of A

and B and C are enqueued.

3. Then vertex ‘B’ is dequeued and its adjacent vertices C and D are taken from the adjacency matrix for enqueuing. Since vertex C is already in the queue, vertex D alone is enqueued.

Here B is dequeued, D is enqueued.

4. Then vertex ‘C’ is dequeued and its adjacent vertices A, B and D are found out. Since vertices A and B are already visited and vertex D is also in the queue, no enqueue operation takes place.



Here C is dequeued

5. Then vertex ‘D’ is dequeued. This process terminates as all the vertices are visited and the queue is also empty.

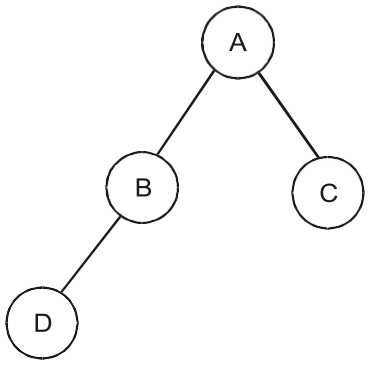


Figure 4.4.3: Breadth first spanning tree

Applications of breadth first search

1. To check whether the graph is connected or not.

4.5 DEPTH FIRST SEARCH

Depth first works by selecting one vertex V of G as a start vertex ; V is marked visited. Then each unvisited vertex adjacent to V is searched in turn using depth first search recursively. This process continues until a dead end (i.e) a vertex with no adjacent unvisited vertices is encountered. At a deadend, the algorithm backsup one edge to the vertex it came from and tries to continue visiting unvisited vertices from there.

The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end. By then, all the vertices in the same connected component as the starting vertex have been visited. If unvisited vertices still remain, the depth first search must be restarted at any one of them.

To implement the Depthfirst Search perform the following Steps :

Step : 1 Choose any node in the graph. Designate it as the search node and mark it as visited.

Step : 2 Using the adjacency matrix of the graph, find a node adjacent to the search. node that has not been visited yet. Designate this as the new search node and mark it as visited.

Step : 3 Repeat step 2 using the new search node. If no nodes satisfying (2) can be found, return to the previous search node and continue from there.

Step : 4 When a return to the previous search node in (3) is impossible, the search from the originally choosen search node is complete.

Step : 5 If the graph still contains unvisited nodes, choose any node that has not been visited and repeat step (1) through (4).

Routine for Depth First Search

```
Void DFS (Vertex V)
{
        visited [V] = True;
        for each W adjacent to V
                if (! visited [W])
                        Dfs (W);
}
```

Example : -

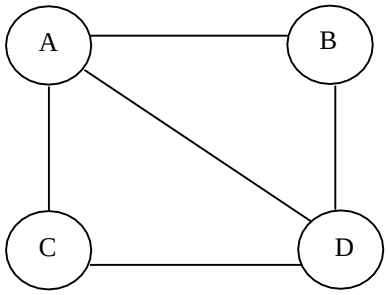


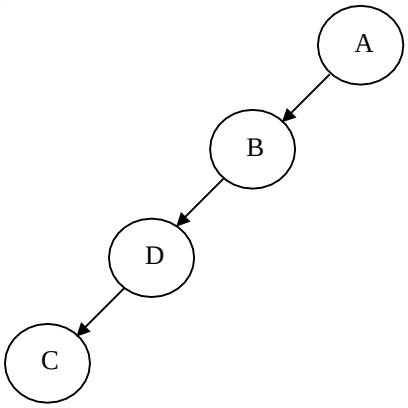
Fig. 4.5

Adjacency Matrix

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

Implementation

1. Let ‘A’ be the source vertex. Mark it to be visited.
2. Find the immediate adjacent unvisited vertex ‘B’ of ‘A’ Mark it to be visited.
3. From ‘B’ the next adjacent vertex is ‘d’ Mark it has visited.
4. From ‘D’ the next unvisited vertex is ‘C’ Mark it to be visited.



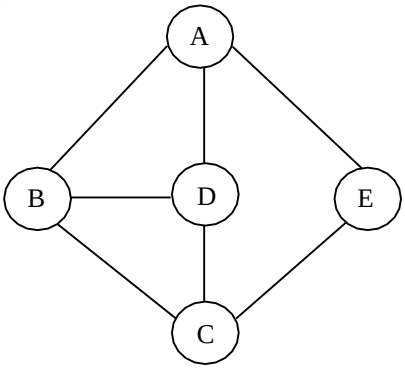
Depth First Spanning Tree

Applications of Depth First Search

1. To check whether the undirected graph is connected or not.
2. To check whether the connected undirected graph is Bioconnected or not.
3. To check the a Acyclicity of the directed graph.

4.5.1 Undirected Graphs

A undirected graph is 'connected' if and only if a depth first search starting from any node visits every node.



An Undirected graph

Adjacency Matrix

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

Implementation

We start at vertex 'A'. Then Mark A as visited and call DFS (B) recursively, Dfs (B) Marks B as visited and calls Dfs(c) recursively.

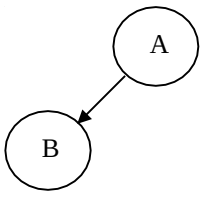


Fig. 4.5.1 (a)

Dfs (c) marks C as visited and calls Dfs (D) recursively. No recursive calls are made to Dfs (B) since B is already visited.

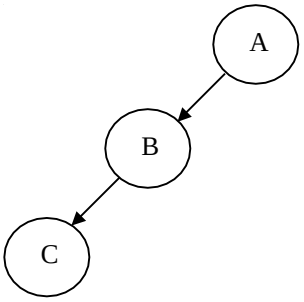


Fig. 4.5.1 (b)

Dfs(D) marks D as visited. Dfs(D) sees A,B,C as marked so no recursive call is made there, and Dfs(D) returns back to Dfs(C).

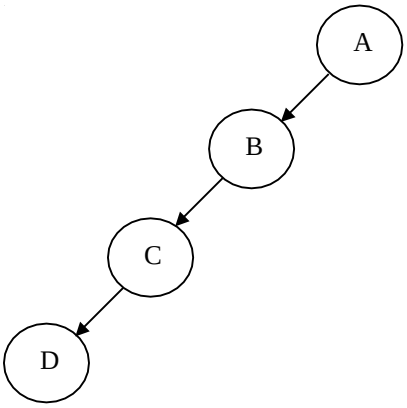


Fig. 4.5.1 (c)

Dfs(C) calls Dfs(E), where E is unseen adjacent vertex to C.

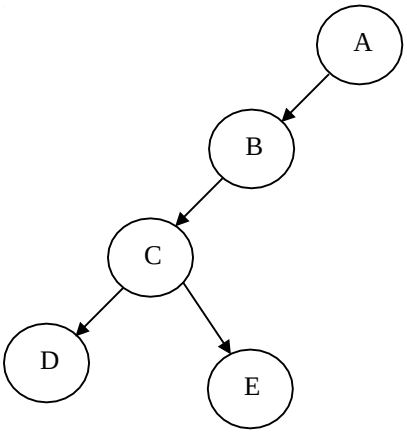


Fig. 4.5.1 (d)

Since all the vertices starting from 'A' are visited, the above graph is said to be connected. If the graph is not connected, then processing all nodes requires reversal calls to Dfs, and each generates a tree. This entire collection is a depth first spanning forest.

4.6 TOPOLOGICAL SORT

A **topological sort** is a linear ordering of vertices in a directed acyclic graph such that if there is a path from V_i to V_j, then V_j appears after V_i in the linear ordering.

Topological ordering is not possible. If the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v.

To implement the topological sort, perform the following steps.

Step 1 : - Find the indegree for every vertex.

Step 2 : - Place the vertices whose indegree is '0' on the empty queue.

Step 3 : - Dequeue the vertex V and decrement the indegree's of all its adjacent vertices.

Step 4 : - Enqueue the vertex on the queue, if its indegree falls to zero.

Step 5 : - Repeat from step 3 until the queue becomes empty.

Step 6 : - The topological ordering is the order in which the vertices dequeued.

Routine to perform Topological Sort

```
/* Assume that the graph is read into an adjacency matrix and that the indegrees are
 computed for every vertices and placed in an array (i.e. Indegree [ ] ) */
        void Topsort (Graph G)
        {
            Queue Q ;
            int counter = 0;
            Vertex V, W ;
            Q = CreateQueue (NumVertex);
            Makeempty (Q);
            for each vertex V
                if (indegree [V] = = 0)
                Enqueue (V, Q);
            while (! IsEmpty (Q))
            {
                V = Dequeue (Q);
                TopNum [V] = + + counter;
```

```
                    for each W adjacent to V
                        if (--Indegree [W] = = 0)
                            Enqueue (W, Q);
                }
        if (counter ! = NumVertex)
            Error (" Graph has a cycle");
        DisposeQueue (Q);        /* Free the Memory */
        }
```

Note :

Enqueue (V, Q) implies to insert a vertex V into the queue Q.

Dequeue (Q) implies to delete a vertex from the queue Q.

TopNum [V] indicates an array to place the topological numbering.

Example 1 :

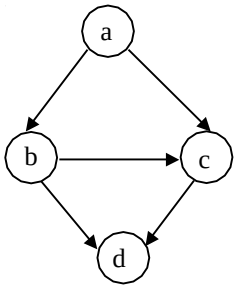


Fig. 4.6.1

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

Adjacency Matrix

Step 1

Number of 1's present in each column of adjacency matrix represents the indegree of the corresponding vertex.

In fig 4.6.1 Indegree [a] = 0 Indegree [b] = 1

Indegree [c] = 2 Indegree [d] = 2

Step 2

Enqueue the vertex, whose indegree is '0'

In fig 4.6.1 vertex 'a' is 0, so place it on the queue.

Step 3

Dequeue the vertex 'a' from the queue and decrement the indegree's of its adjacent vertex 'b' & 'c'

Hence,Indegree [b] = 0 and Indegree [c] = 1

Now, Enqueue the vertex ‘b’ as its indegree becomes zero.

Step 4

Dequeue the vertex ‘b’ from Q and decrement the indegree’s of its adjacent vertex ‘c’ and ‘d’.

Hence, Indegree [c] = 0 and Indegree [d] = 1

Now, Enqueue the vertex ‘c’ as its indegree falls to zero.

Step 5

Dequeue the vertex ‘c’ from Q and decrement the indegree’s of its adjacent vertex ‘d’.

Hence, Indegree [d] = 0

Now, Enqueue the vertex ‘d’ as its indegree falls to zero.

Step 6

Dequeue the vertex ‘d’.

Step 7

As the queue becomes empty, topological ordering is performed, which is nothing but, the order in which the vertices are dequeued.

VERTEX	**1**	**2**	**3**	**4**
a	[0]	0	0	0
b	1	[0]	0	0
c	2	1	[0]	0
d	2	2	1	[0]
ENQUEUE	a	b	c	d
DEQUEUE	a	b	c	d

Result of Applying Topological Sort to the Graph in Fig. 4.6.1

Example 2 :

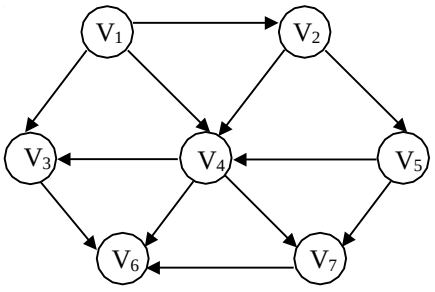


Fig 4.6.2

Adjacency Matrix :-

	V_1	V_2	V_3	V_4	V_5	V_6	V_7
V_1	0	1	1	1	0	0	0
V_2	0	0	0	1	1	0	0
V_3	0	0	0	0	0	1	0
V_4	0	0	1	0	0	1	1
V_5	0	0	0	1	0	0	1
V_6	0	0	0	0	0	0	0
V_7	0	0	0	0	0	1	0
INDEGREE	0	1	2	3	1	3	2

Indegree $[V_1] = 0$ Indegree $[V_2] = 1$ Indegree $[V_3] = 2$
Indegree $[V_4] = 3$ Indegree $[V_5] = 1$ Indegree $[V_6] = 3$
Indegree $[V_7] = 2$

INDEGREE BEFORE DEQUEUE #

VERTEX	1	2	3	4	5	6	7
V_1	0	0	0	0	0	0	0
V_2	1	0	0	0	0	0	0
V_3	2	1	1	1	0	0	0
V_4	3	2	1	0	0	0	0
V_5	1	1	0	0	0	0	0
V_6	3	3	3	3	2	1	0
V_7	2	2	2	1	0	0	0
ENQUEUE	V_1	V_2	V_5	V_4	V_3,	V_7	V_6
DEQUEUE	V_1	V_2	V_5	V_4	V_3	V_7	V_6

Result of Applying Topological Sort to the Graph in Fig. 4.6.2

The topological order is V_1, V_2, V_5, V_4, V_3, V_7, V_6

Analysis

The running time of this algorithm is $0(|E| + |V|)$. where E represents the Edges & V represents the vertices of the graph.

4.7 BICONNECTIVITY

A connected undirected graph is biconnected if there are no vertices whose removal disconnects the rest of the graph.

4.8 ARTICULATION POINTS OR CUT VERTEX

The vertices whose removal would disconnect the graph are known as articulation points.

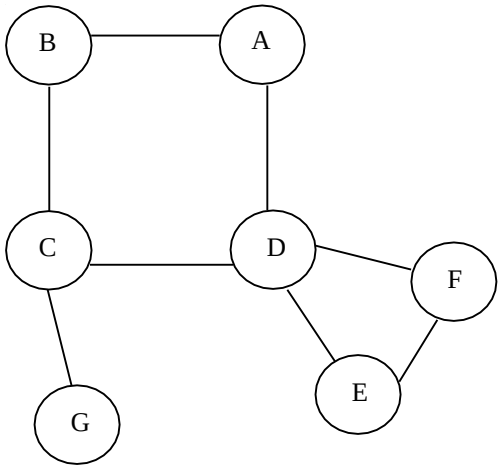


Fig. 4.8 Connected Undirected Graph

Here the removal of 'C' vertex will disconnect G from the graph.

Similarly removal of 'D' vertex will disconnect E & F from the graph. Therefore 'C' & 'D' are articulation points.

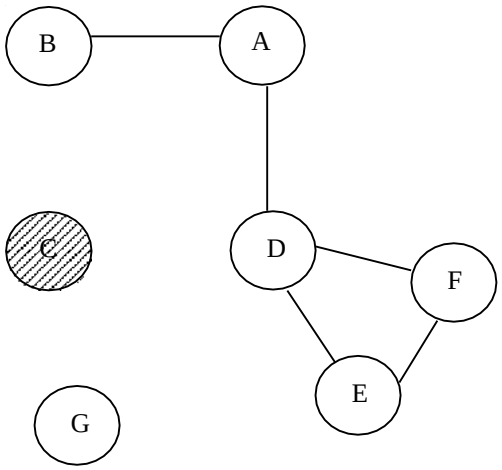


Fig. 4.8 (a) Removal of vertex 'C'

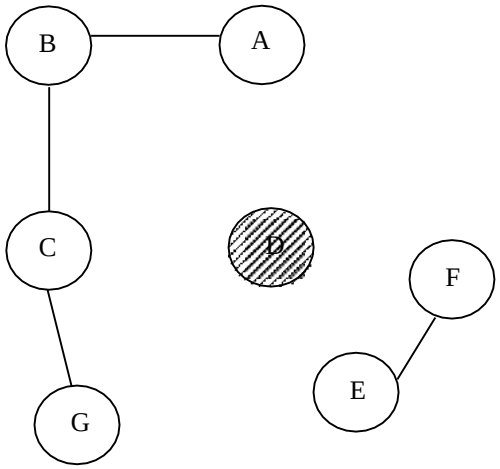


Fig. 4.8 (b) Removal of vertex 'D'

The graph is not biconnected, if it has articulation points.

Depth first search provides a linear time algorithm to find all articulation points in a connected graph.

Steps to find Articulation Points :

Step 1 : Perform Depth first search, starting at any vertex

Step 2 : Number the vertex as they are visited, as Num (v).

Step 3 : Compute the lowest numbered vertex for every vertex v in the Depth first spanning tree, which we call as low (w), that is reachable from v by taking zero or more tree edges and then possibly one back edge. By definition, Low(v) is the minimum of

(i) Num (v)

(ii) The lowest Num (w) among all back edges (v, w)

(iii) The lowest Low (w) among all tree edges (v, w)

Step 4 : (i) The root is an articulation if and only if it has more than two children.

(ii) Any vertex v other than root is an articulation point if and only if v has same child w such that Low (w) $\geq$ Num (v), The time taken to compute this algorithm an a graph is $0(|E|+|V|)$.

Note

For any edge (v, w) we can tell whether it is a tree edge or back edge merely by checking Num (v) and Num (w).

If Num (w) > Num (v) then the edge is a back edge.

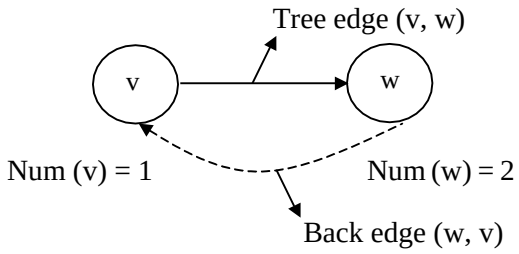


Fig. 4.8.3

Routine to compute low and test for articulation points

```
void AssignLow (Vertex V)
{
    Vertex W;
    Low [V] = Num [V]; /* Rule 1 */
    for each W adjacent to V
    {
        If (Num [W] > Num [V]) /* forward edge */
        {
            Assign Low (W);
            If (Low [W]> = Num [V])
                Printf ("% V is an articulation pt \n", V);
            Low[V] = Min (Low [V],    Low[W]); /* Rule 3*/
        }
        else
        if (parent [V] ! = W) /* Back edge */
            Low [V] = Min (Low [V],  Num [W])); /* Rule 2*/
    }
}
```

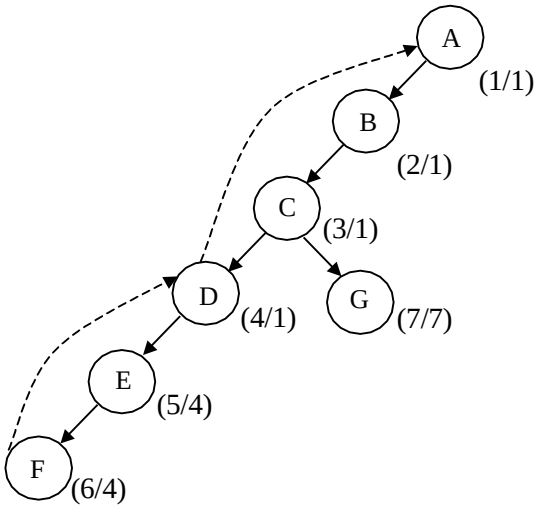


Fig. 4.8.4 Depth First Tree For Fig (4.8) With Num and Low.

Low can be computed by performing a postorder traversal of the depth - first spanning tree. (ie)

Low (F) = Min (Num (F), Num (D))

/* Since there is no tree edge & only one back edge */

= Min (6, 4) = 4

Low (F) = 4

Low (E) = Min (Num (E), Low (F))

/* there is no back edge */.

= Min (5, 4) = 4

Low (D) = Min (Num (D), Low (E), Num (A))

= Min (4,4,1) = 1

Low (D) = 1

Low (G) = Min (Num (G)) = 7 /* Since there is no tree edge & back edge */

Low (C) = Min (Num (C), Low (D), Low (G))

= Min (3,1,7) = 1

Low (C) = 1 .

lll^{ly} Low (B) = Min (Num (B), Low (C))

= Min (2,1) = 1

Low (A) = Min (Num (A), Low (B))

= Min (1, 1) = 1

Low (A) = 1.

From fig (4.8) it is clear that Low (G) > Num (C) (ie) 7 > 3 /* if Low (W) ≥ Num (V)*/ the 'v' is an articulation pt Therefore 'C' is an articulation point.

lll[ly] Low (E) = Num (D), Hence D is an articulation point.

4.9 EULER'S CIRCUIT

Euler path

A graph is said to be containing an Euler path if it can be traced in 1 sweep without lifting the pencil from the paper and without tracing the same edge more than once. Vertices may be passed through more than once. The starting and ending points need not be the same.

Euler circuit

An Euler circuit is similar to an Euler path, except that the starting and ending points must be the same.

It is interesting that Euler never published an algorithm for finding an Euler circuit, but only provided a method of determining if one existed or not. In a note from Ed Sandifer he states, "In his paper on the Konigsberg Bridge Problem, all he says about finding such paths is that if you remove all double edges, then it will be easy to find a solution".

Euler went on to generalize this mode of thinking, laying a foundation for graph theory. Using modern vocabulary, we make the following definitions and prove at theorem:

Definition:

A network is a figure made up of points (vertices) connected by non-intersecting curves (arcs).

Definition:

A vertex is called odd if it has an odd number of arcs leading to it, otherwise it is called even.

Definition:

An Euler path is a continuous path that passes through every are once and only once.

Theorem:

If a network has more than two odd vertices, it does not have an Euler path.

Euler also proved this:

Theorem:

If a network has two or zero odd vertices, it has atleast one Euler path. In particular, if a network has exactly two odd vertices, then its Euler paths can only start on one of the odd vertices, and end on the other -- a type of Euler path called an Euler circuit.

Problem

The seven bridges of konigsberg

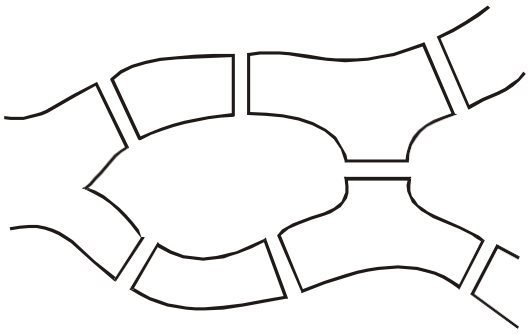


Fig. 4.9

The river Pregel separates the city of Konigsberg into 4 separate regions and the regions are connected by 7 bridges. In the summer evenings, the citizens of the country would like to have a walk around the whole city. Some curious citizens wondered whether it is possible to begin at one of the regions, cross each bridge exactly once and return to the same starting point. Can the citizen's suggestion be made possible?

Solution:

We can observe that each vertex has an odd number of edges. For example, vertex A is of degree 5 and vertex B is of degree 3. Therefore the citizen's suggestion is impossible. As each edge can be used only once and all vertices are odd, it is impossible to re-enter any vertex again after leaving it, and this makes starting and ending at the same point impossible.

Problems

For each of the networks below, determine whether it has an Euler path. If it does, find one.

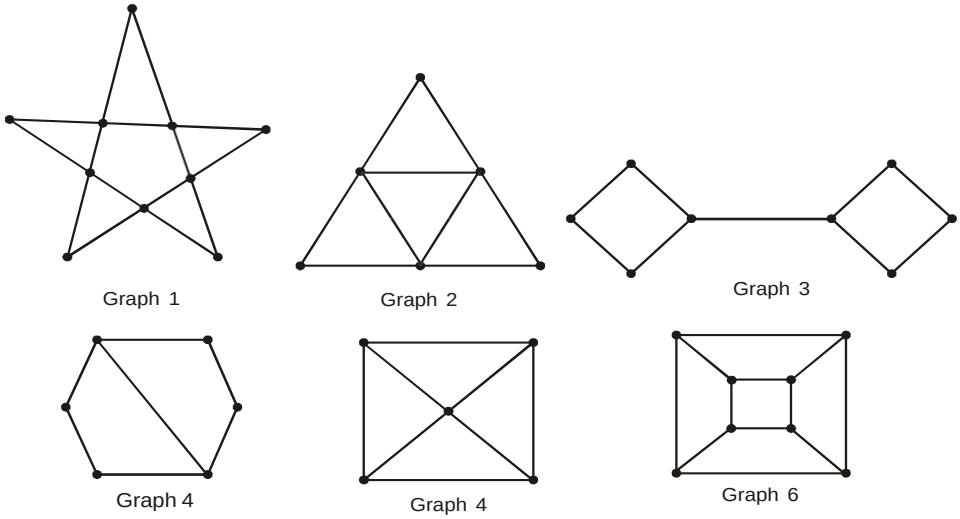


Fig. 4.9

Graph	*Number of odd vertices(vertices connected to an odd number of edges)*	*Number of even vertices (vertices connected to an even number of edges*	*What does the path contain? (Euler path = P; Euler circuit = C; Neither = N)*
1	0	10	C
2	0	6	C
3	2	6	P
4	2	4	P
5	4	1	N
6	8	0	N

From the above table, we can observe that:

1. A graph with all vertices being even contains an Euler circuit.
2. A graph with 2 odd vertices and some even vertices contains an Euler path.
3. A graph with more than 2 odd vertices does not contain any Euler path or circuit.

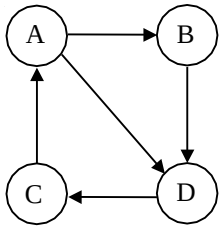
4.10. APPLICATIONS OF GRAPHS

1. **Social network graphs**: to tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in socialstructures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.
2. **Transportation networks**. In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.
3. **Utility graphs**. The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.
4. **Document link graphs**. The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.

5. **Protein-protein interactions graphs**. Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process. Humans have over 120K proteins with millions of interactions among them.
6. **Network packet traffic graphs**. Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.
7. **Scene graphs.** In graphics and computer games scene graphs represent the logical or spacial relationships between objects in a scene. Such graphs are very important in the computer games industry.
8. **Finite element meshes**. In engineering many simulations of physical systems, such as the flow of air over a car or airplane wing, the spread of earthquakes through the ground, or the structural vibrations of a building, involve partitioning space into discrete elements. The elements along with the connections between adjacent elements forms a graph that is called a finite element mesh.
9. **Robot planning**. Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous vehicles.
10. **Neural networks.** Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 1011 neurons

PART - A

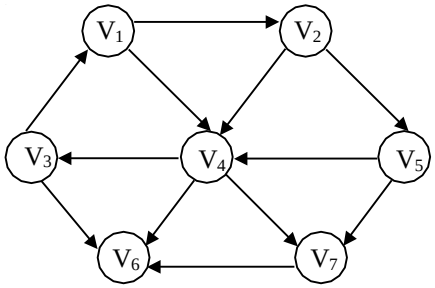
1. Define a graph
2. Compare directed graph and undirected graph
3. Define path, degree and cycle in a graph
4. What is an adjacency matrix?
5. Give the adjacency list for the following graph



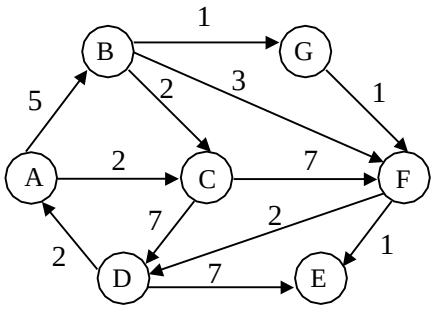
6. Define Topological Sort
7. Define Shortest path problem. Give examples
8. Define Minimum Spanning Tree and write its properties.
9. What is DAG? Write its purpose
10. What are the different ways of traversing a graph?
11. What are the various applications of depth first search?
12. What is an articulation point?
13. When a graph is said to be bi-connected?
14. Write down the recursive routine for depth first search.
15. Write a procedure to check the biconnectivity of the graph using DFS.
16. Define Class_NP
17. What is meant by NP_Complete problem?
18. Define Euler oath and Euler circuit.
19. Write the routine for Breadth first traversel.

PART - B

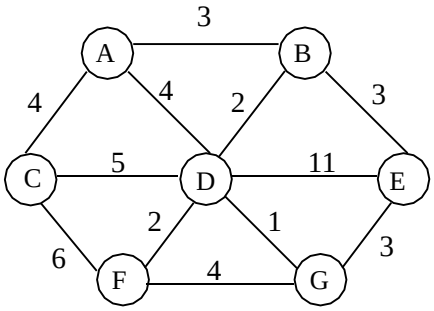
1. What is Topological Sort? Write down the pseudocode to perform topological sort and apply the same to the following graph.



2. Explain the Dijkstra's algorithm and find the shortest path from A to all other vertices in the following graph.



3. Explain Prim's and Kruskal's algorithm in detail and find the minimum spanning tree for the following graph.



4. Find all the articulation points in the given graph. Show the depth_first spanning tree and the values of Num and Low for each vertex.