Rajalakshmi Engineering College

Rajalakshmi Nagar, Thandalam, Chennai - 602 105

Department of Computer Science and Engineering

CS19241 - Data Structures

Unit - IV

Lecture Notes

(Regulations - 2019)

Prepared by:

B.BHUVANESWARAN

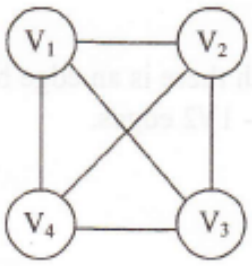
Assistant Professor (SG) / CSE / REC

bhuvaneswaran@rajalakshmi.edu.in

1.1 INTRODUCTION

A graph G = (V, E) consists of a set of vertices, V, and a set of edges, E. Vertices are referred to as nodes. Each edge is a pair (v, w), where v, w ∈ V i.e. v = V_1, w = V_2. Edges are sometimes referred to as arcs.

Example

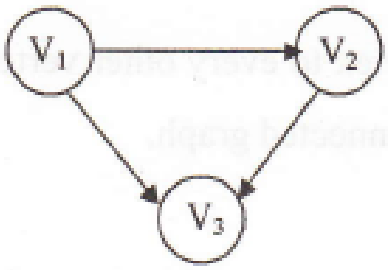


Here V_1, V_2, V_3 and V_4 are vertices and (V_1, V_2), (V_1, V_3), (V_1, V_4), (V_2, V_3), (V_2, V_4) and (V_3, V_4) are edges.

1.2 BASIC TERMINOLOGIES

1.2.1 Directed Graph (or) Digraph

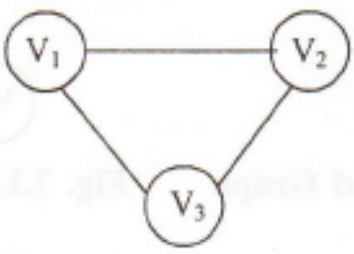
Directed graph is a graph which consists of directed edges, where each edge in E is unidirectional. It is also referred as Digraph. If (v, w) is a directed edge then (v, w) ≠ (w, v).



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

1.2.3 Undirected Graph

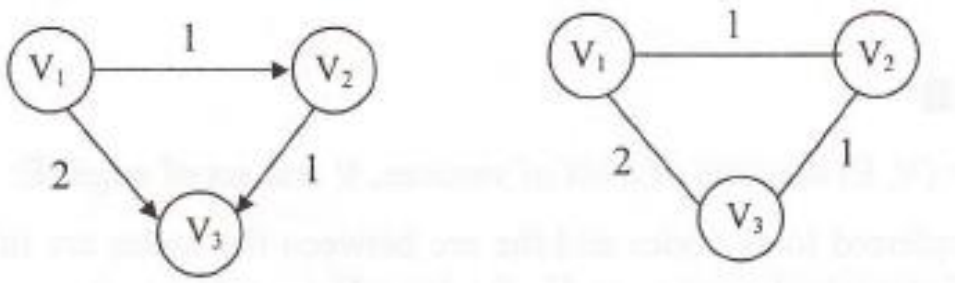
An undirected graph is a graph, which consists of undirected edges. If (v, w) is an undirected edge then (v, w) = (w, v).



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

1.2.4 Weighted Graph

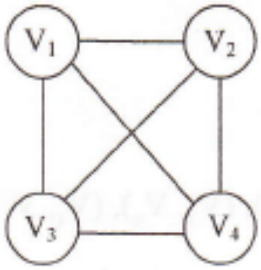
A graph is said to be weighted graph if every edge in the graph is assigned a weight or value. It can be directed or undirected graph.



1.2.5 Complete Graph

An undirected graph, in which every vertex has an edge to all other vertices is called a complete graph. A complete graph with n vertices has n(n-1)/2 edges.

Example

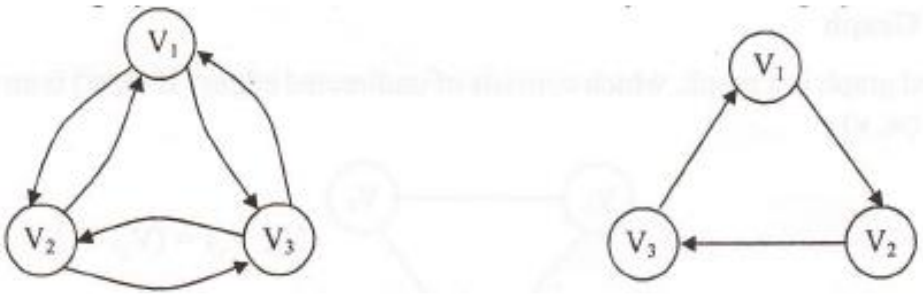


In Fig. Number of vertices = 4
Number of edges = 4(4-1)/2 = 12/6 = 6

1.2.6 Strongly and Weakly Connected Graphs

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.

Example



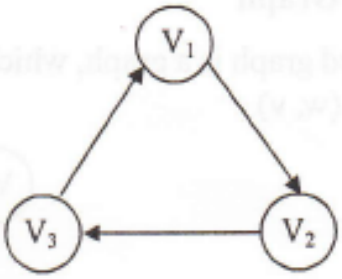
Strongly Connected **Weakly Connected**

1.2.7 Path

A path in a graph is a sequence of vertices $w_1, w_2, w_3, \ldots, w_n$ such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < N$.

Example

In Fig. the path from V_1 to V_3 is V_1, V_2, V_3.

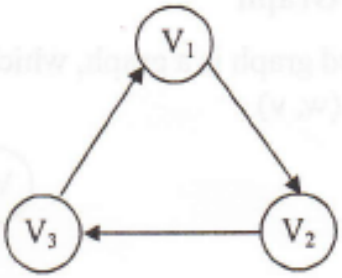


1.2.8 Length

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

Example

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

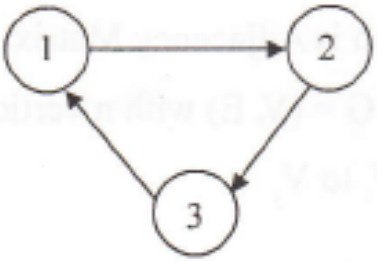


1.2.9 Loop

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

1.2.10 Cycle

A cycle in a graph is a path in which first and last vertex are the same. A graph which has cycles is referred to as cyclic graph.



1.2.11 Degree

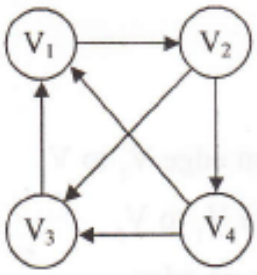
The number of edges incident on a vertex determines its degree. The degree of the vertex V written as degree (V).

1.2.12 Indegree

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

Example

Indegree (V_1) = 2.

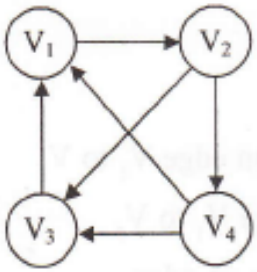


1.2.13 Outdegree

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

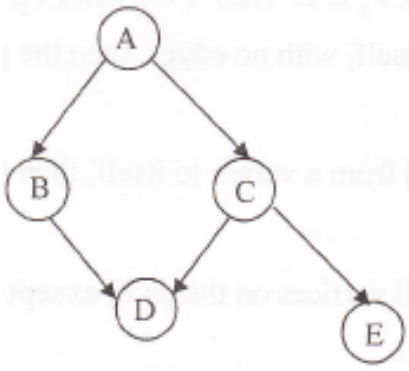
Example

Outdegree (V_1) = 1.



1.2.14 Acyclic

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



2.1 INTRODUCTION

Graph can be represented by:

- Adjacency Matrix
- Adjacency list

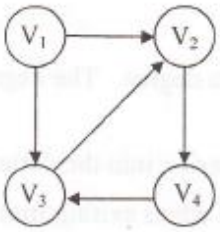
2.2 ADJACENCY MATRIX

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

2.2.1 Adjacency Matrix for Directed Graph

Example-1

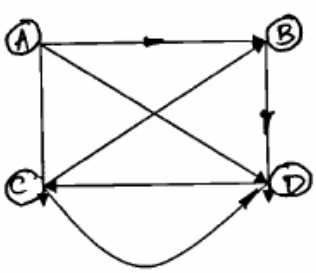


	V_1	V_2	V_3	V_4
V_1	0	1	1	0
V_2	0	0	0	1
V_3	0	1	0	0
V_4	0	0	1	0

$V_{1,2}$, $V_{1,3}$, $V_{2,4}$, $V_{3,2}$ and $V_{4,3}$ = 1 since there is an edge.

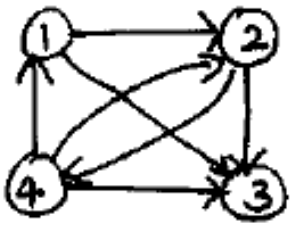
Others 0, there is no edge.

Example-2



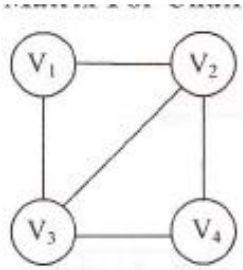
	A	B	C	D
A	0	1	1	1
B	0	0	0	1
C	0	1	0	1
D	0	0	1	0

Example-3



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

2.2.2 Adjacency Matrix for Undirected Graph



	V1	V2	V3	V4
V1	0	1	1	0
V2	1	0	1	1
V3	1	1	0	1
V4	0	1	1	0

2.2.3 Adjacency matrix for weighted graph

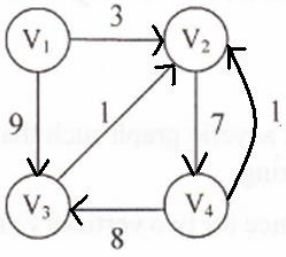
To solve some graph problems, adjacency matrix can be constructed as:

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

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

If there is no arc from i to j, assume C[i, j] = ∞ where i ≠ j.

Example



	V1	V2	V3	V4
V1	0	3	9	∞
V2	∞	0	∞	7
V3	∞	1	0	∞
V4	∞	1	8	0

Advantage

Simple to implement.

Disadvantage

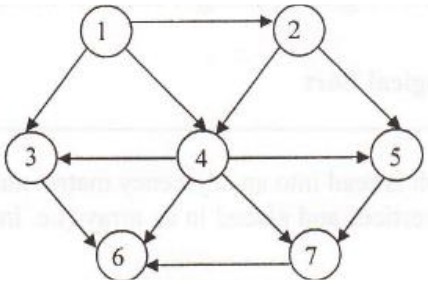
Takes $O(n^2)$ space to represents the graph.

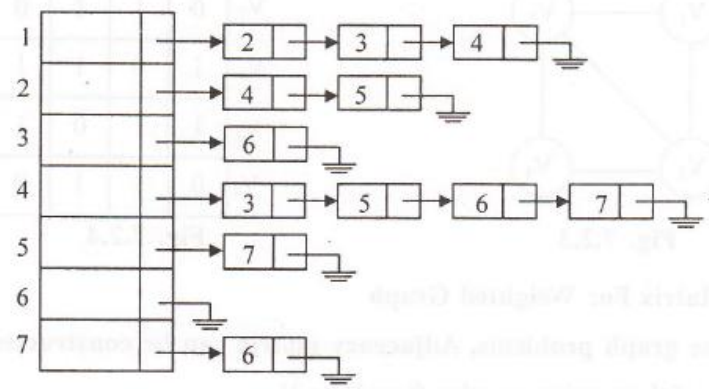
It takes $O(n^2)$ time to solve the most of the problems.

2.3 ADJACENCY LIST REPRESENTATION

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

Example





Disadvantages

It takes 0(n) time to determine whether there is an arc from vertex i to vertex j. Since there can be 0(n) vertices on the adjacency list for vertex i.

3.1 INTRODUCTION

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

- Breadth first traversal
- Depth first traversal

3.2 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.

3.2.1 Algorithm

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.

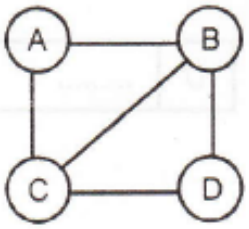
Step 5: This process continues until the queue Q which keeps track of the adjacent nodes is empty.

3.2.2 Routine for BFS

```
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;
                        }
```

```
        }
    }
```

3.2.3 Example



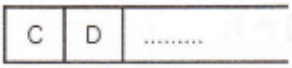
	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	1
D	0	1	1	0

3.2.4 Implementation

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

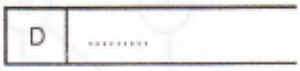
2. Find the adjacent unvisited vertices of 'A' and enqueue then' into the queue. Here B and C are adjacent nodes of A and B and C are enqueued.

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.

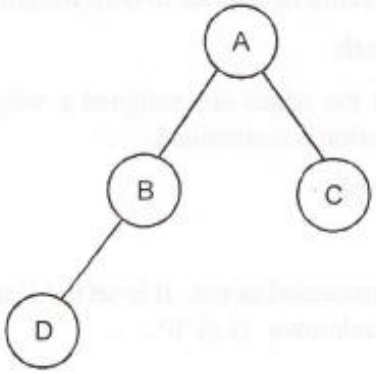
4. Then vertex 'C' is dequeued and its adjacent vertices A, B and D are found out. Since vertices A and B are already visited and vertex D is also in the queue, no enqueue operation takes place.



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.

3.2.5 Breadth First Spanning Tree



3.2.6 Applications of BFS

To check whether the graph is connected or not.

3.3 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.

3.3.1 Algorithm

To implement the Depth First 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.

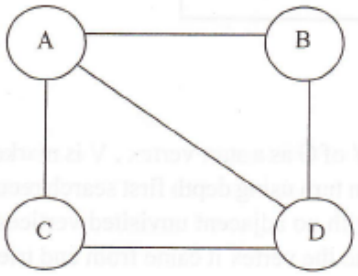
Step 4: When a return to the previous search node in (3) is impossible, the search from the originally chosen search node is complete.

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

3.3.2 Routine for DFS

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

3.3.3 Example

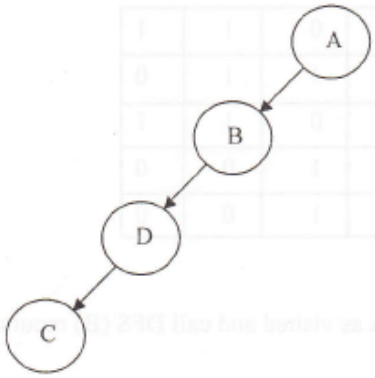


	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	0

3.3.4 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.

3.3.5 Depth First Spanning Tree



3.3.6 Applications of DFS

- To check whether the undirected graph is connected or not.
- To check whether the connected undirected graph is Bioconnected or not.
- To check the Acyclicity of the directed graph.

4.1 INTRODUCTION

A topological sort is an 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 ordering.

A 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. Furthermore, the ordering is not necessarily unique; any legal ordering will do. In the graph in Figure v1, v2, v5, v4, v3, v7, v6 and v1, v2, v5, v4, v7, v3, v6 are both topological orderings.

4.2 ALGORITHM

A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges. We can then print this vertex, and remove it, along with its edges, from the graph. Then we apply this same strategy to the rest of the graph.

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 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 dequeue.

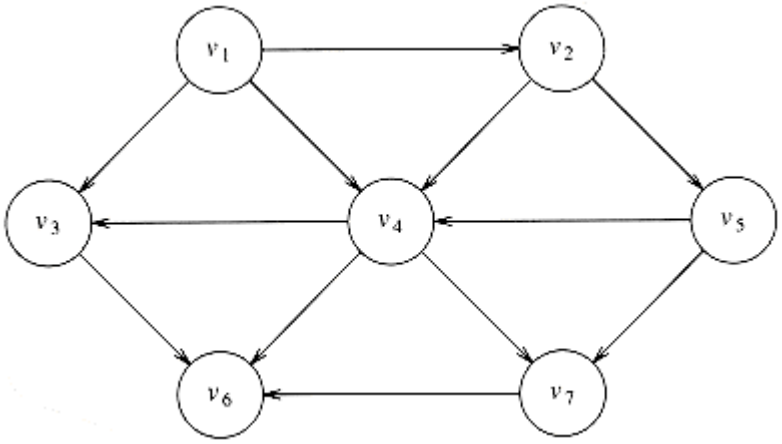
4.3 ROUTINE TO PERFORM TOPOLOGICAL SORT

```
/* Assume thatt 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 [ ] ) */

voidTopsort (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 */
}
```

4.4 EXAMPLE



	v1	v2	v3	v4	v5	v6	v7
v1	0	1	1	1	0	0	0
v2	0	0	0	1	1	0	0
v3	0	0	0	0	0	1	0
v4	0	0	1	0	0	1	1
v5	0	0	0	1	0	0	1
v6	0	0	0	0	0	0	0
v7	0	0	0	0	0	1	0

Step 1:

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

In fig. Indegree [v1] = 0 Indegree [v2] = 1 Indegree [v3] = 2 Indegree [v4] = 3

Indegree [v5] = 1 Indegree [v6] = 3 Indegree [v7] = 2

Step 2:

Enqueue the vertex, whose indegree is 0.

In fig 7.3.1 vertex v1 is 0, so place it on the queue.

Step 3:

Dequeue the vertex v1 from the queue and decrement the indegree of its adjacent vertex v2, v3 and v4.

Hence, Indegree [v2] = 0 Indegree [v3] = 1 Indegree [v4] = 2

Now, enqueue the vertex v2 as its indegree becomes zero.

Step 4:

Dequeue the vertex v2 from the queue and decrement the indegree of its adjacent vertex v4 and v5.

Hence, Indegree [v4] = 1 Indegree [v5] = 0

Now, enqueue the vertex v5 as its indegree falls to zero.

Step 5:

Dequeue the vertex v5 from the queue and decrement the indegrees of its adjacent vertex v4 and v7.

Hence, Indegree [v4] = 0 Indegree [v7] = 1

Now, enqueue the vertex v4 as its indegree falls to zero.

Step 6:

Dequeue the vertex v4 from the queue and decrement the indegrees of its adjacent vertex v3, v6 and v7.

Hence, Indegree [v3] = 0 Indegree [v6] = 2 Indegree [v7] = 0

Now, enqueue the vertex v3 and v7 in any order as its indegree falls to zero.

Step 7:

Dequeue the vertex v3 from the queue and decrement the indegrees of its adjacent vertex v6.

Hence, Indegree [v6] = 1

Step 7:

Dequeue the vertex v7 from the queue and decrement the indegrees of its adjacent vertex v6.

Hence, Indegree [v6] = 0

Now, enqueue the vertex v6 as its indegree falls to zero.

Step 8:

Dequeue the vertex v6.

Step 9:

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

Indegree befoew dequeue #							
Vertex	**1**	**2**	**3**	**4**	**5**	**6**	**7**
V1	0	0	0	0	0	0	0
V2	1	0	0	0	0	0	0
V3	2	1	1	1	0	0	0
V4	3	2	1	0	0	0	0
V5	1	1	0	0	0	0	0
V6	3	3	3	3	2	1	0
V7	2	2	2	1	0	0	0
Enqueue	v1	v2	v5	v4	v3, v7		v6
Dequeue	v1	v2	v5	v4	v3	v7	v6

Analysis

The running time of this algorithm is O (|E| + |V|), where E represents the Edges and V represents the vertices of the graph.

5.1 INTRODUCTION

Given as input a weighted graph, G = (V, E), and a distinguished vertex, s, find the shortest weighted path from s to every other vertex in G.

5.2 DIJKSTRA'S ALGORITHM

The general method to solve the single source shortest path problem is known as Dijkstra's algorithm. This is applied to the weighted graph G.

Dijkstra's algorithm is the prime example of Greedy technique, which generally solve a problem in stages by doing what appears to be the best thing at each stage. This algorithm proceeds in stages, just like the unweighted shortest path algorithm. At each stage, it selects a vertex v, which has the smallest dv among all the unknown vertices, and declares that as the shortest path from S to V and mark it to be known. We should set dw = dv + Cvw, if the new value for dw would be an improvement.

5.3 ROUTINE

```
void Dijkstra (Graph G, Table T)
{
        int i;
        vertex V, W;
        Read Graph (G T) /* Read graph from adjacency list */
        /* Table Initialization */
        for (I = 0; i <Numvertex; i++)
        {
                T[i].known = False;
                T[i].Dist = Ifinity;
                T[i].path=NotA vertex;
        }
        T [start].dist = 0;
        for (; ;)
        {
                V = Smallest unknown distance vertex;
                if (V == NotA vertex)
                        break ;
                T[V].known = True;
                for each W adjacent to V
                        if (!T[W].known)
                        {
                                T[W].Dist = Min(T[W].Dist, T[V].Dist + Cvw);
                                T[W].path = V;
                        }
        }
}
```

5.4 Example

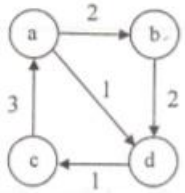


Figure The directed graph G.

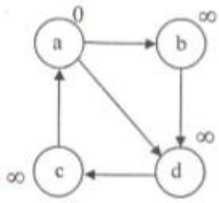


Fig. (a)

V	known	d_v	P_v
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

Initial Configuration

Vertex a is chosen as source and is declared as known vertex.

Then the adjacent vertices of a is found and its distance are updated as follows :

T[b].Dist = Min [T[b].Dist, T[a].Dist + Ca,b] = Min [∞, 0 + 2] = 2

T[d].Dist = Min [T[d].Dist, T[a]. Dist + Ca,d) = Min [∞, 0 + 1] = 1

V	known	d_v	P_v
a	1	0	0
b	0	2	a
c	0	∞	0
d	0	1	a

After a is declared known

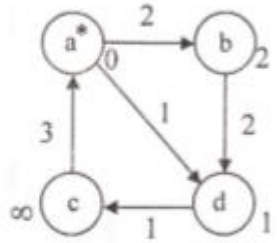


Fig. (b)

Now, select the vertex with minimum distance, which is not known and mark that vertex as visited.

Here d is the next minimum distance vertex. The adjacent vertex to d is c, therefore, the distance of c is updated a follows:

T[c].Dist= Min [T[c].Dist, T[d].Dist+ Cd,c] = Min[∞, 1 +1] = 2

V	known	d_v	P_v
a	1	0	0
b	0	2	a
c	0	2	d
d	1	1	a

After d is declared known

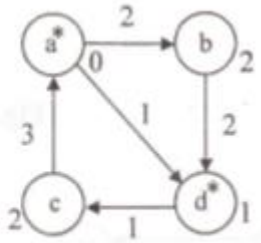


Fig. (c)

The next minimum vertex is band mark it as visited.

Since the adjacent vertex d is already visited, select the next minimum vertex c and mark it as visited.

V	known	d_v	P_v
a	1	0	0
b	1	2	a
c	0	2	d
d	1	1	a

After b is declared known

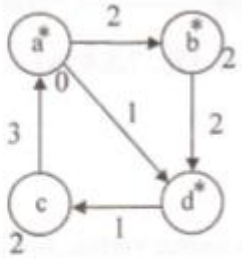


Fig. (d)

V	known	d_v	P_v
a	1	0	0
b	1	2	a
c	1	2	d
d	1	1	a

After c is declared known and algorithm terminates

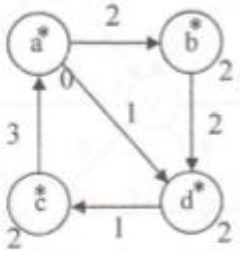


Fig. (e)

6.1 INTRODUCTION

A spanning tree of a connected graph is its connected a cyclic subgraph that contains all the vertices of the graph.

A minimum spanning tree of a weighted connected graph G is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges. The total number of edges in Minimum Spanning Tree (MST) is |V| - 1 where V is the number of vertices. A minimum spanning tree exists if and only if G is connected. For any spanning Tree T, if an edge e that is not in T is added, a cycle is created. The removal of any edge on the cycle reinstates the spanning tree property.

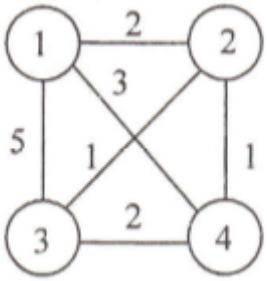
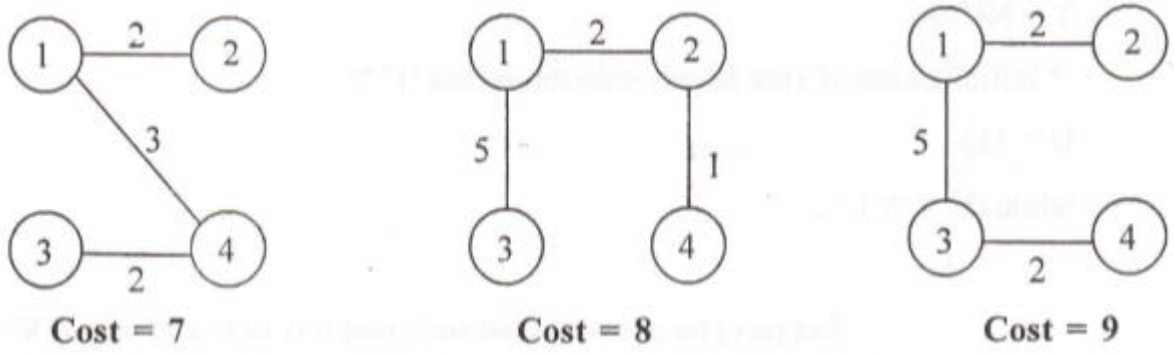


Fig. Connected Graph G



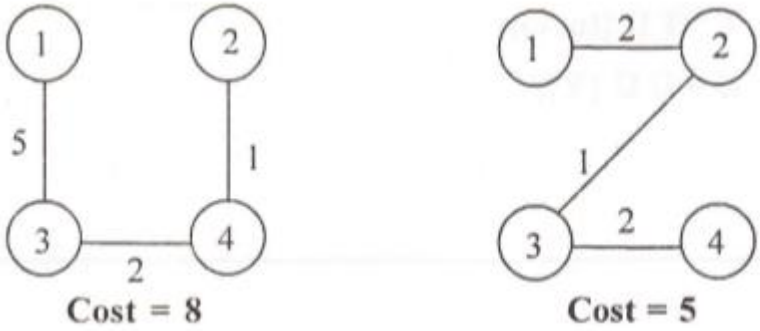
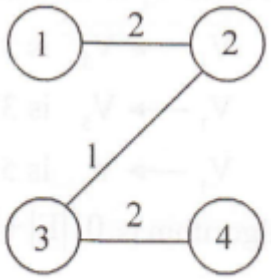


Fig. Spanning Tree for the above Graph

Minimum Spanning Tree



Cost = 5

Fig. Spanning Tree with minimum cost is considered as Minimum Spanning Tree

6.2 PRIM'S ALGORITHM

Prim's algorithm is one of the way to compute a minimum spanning tree which uses a Greedy technique. This algorithm begins with a set U initialised to {1}. It then grows a spanning tree, one edge at a time. At each step, it finds a shortest edge (u, v) such that the cost of (u, v) is the smallest among all edges, where u is in Minimum Spanning Tree and V is not in Minimum Spanning Tree.

5.3 ROUTINE

```
void Prims (Table T)
{
        vertex V, W;
        /* Table initialization */
        for (i = 0; i < Numvertex ; i++)
        {
                T[i].known = False;
                T[i].Dist = lnfinity;
                T[i]. path = 0;
        }
        for (; ;)
        {
                Let V be the start vertex with the smallest distance
                T[V].dist = 0;
                T[V]. known = True;
                for each W adjacent to V
                        If (!T[W].Known)
                        {
                                T[W].Dist = Min (T[W].Dist, Cvw);
                                T[W].path = V;
                        }
        }
}
```

6.4 Example

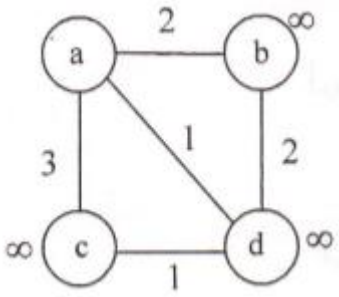


Fig. Undirected Graph G

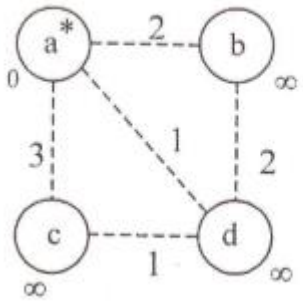


Fig. (a)

V	known	d_v	P_v
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

Initial Configuration

Here, 'a' is taken as source vertex and marked as visited.

Then the distance of its adjacent vertex is updated as follows:

T[b].dist = Min(T[b].Dist, $C_{a,b}$)
= Min(∞, 2)
= 2

T[d].dist = Min(T[d].Dist, $C_{a,b}$)
= Min(∞, 1)
= 1

T[c].dist = Min(T[c].Dist, $C_{a,b}$)
= Min(∞, 3)
= 3

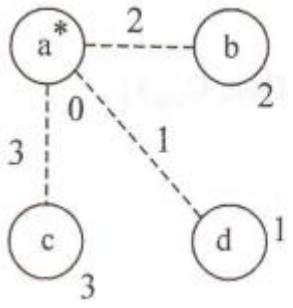


Fig. (a)

V	known	d_v	P_v
a	1	0	0
b	0	2	a
c	0	3	a
d	0	1	a

After the vertex 'a' is marked visited

Next, vertex 'd' with minimum distance is marked as visited and the distance of its unknown adjacent vertex is updated.

T[b].Dist = Min(T[b].Dist, $C_{d,b}$)
= Min(2, 2)
= 2

T[c].dist = Min(T[c].Dist, $C_{d,c}$)
= Min (3, 1)
= 1

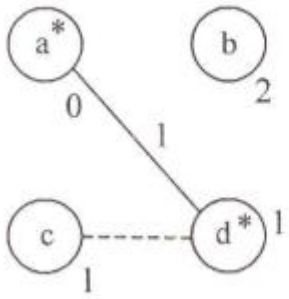


Fig. (a)

V	known	d_v	P_v
a	1	0	0
b	0	2	a
c	0	1	d
d	1	1	a

After the vertex 'd' is marked visited

Next, the vertex with minimum cost 'c' is marked as visited and the distance of its unknown adjacent vertex is updated.

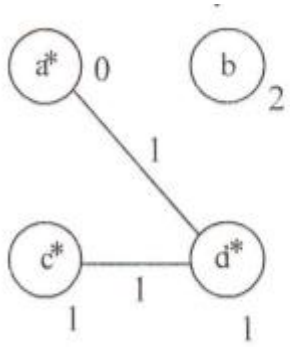


Fig. (a)

V	known	d_v	P_v
a	1	0	0
b	0	2	a
c	1	1	d
d	1	1	a

After the vertex 'c' is marked visited

Since, there is no unknown vertex adjacent to 'c', there is no updation in the distance. Finally, the vertex 'b' which is not visited is marked.

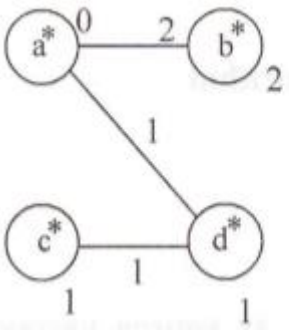


Fig. (a)

V	known	d_v	P_v
a	1	0	0
b	1	2	a
c	1	1	d
d	1	1	1

After the vertex 'b' is marked visited, the algorithm terminates

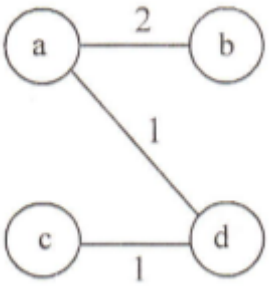


Fig. (f)
Minimum Spanning Tree
Tree is 4 [i.e. $C_{a,b} + C_{a,d} + C_{c,d}$]

RAJALAKSHMI
ENGINEERING COLLEGE

Website : www.rajalakshmi.org

Elearning : www.rajalakshmicolleges.net/moodle

Give a man a fish and you feed him for a day.

Teach him how to fish and you feed him for a lifetime.