UNIT-5

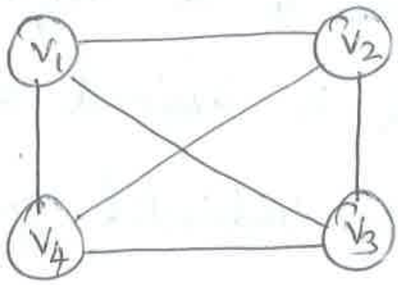
GRAPHS

REPRESENTATIONS OF GRAPHS:

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

Vertices are referred to as nodes and the arc between the nodes are referred to as edges.

Each edge is a pair $(v, w) \in V$. (i.e) $v = V_1$, $w = V_2$



Here V_1, V_2, V_3, V_4 are 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)

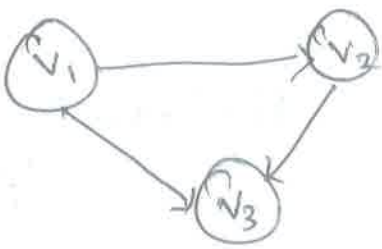
are edges.

DIRECTED GRAPH OR DIGRAPH

-> Directed graph is a graph, which consists of directed edges, where each edge in E is unidirectional

-> Also called as digraph.

-> If (v, w) is a directed edge, then $(v, w) \neq (w, v)$

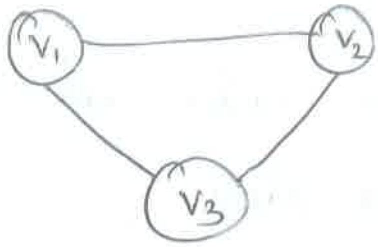


$(V_1, V_2) \neq (V_2, V_1)$

UNDIRECTED GRAPH

An undirected graph is a graph which consists of undirected edges.

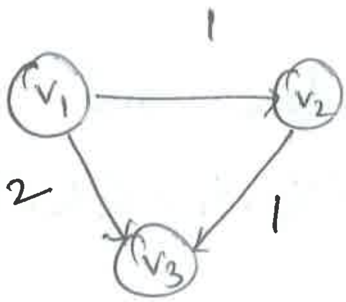
If (v, ω) is an undirected edge, then $(v, \omega) = (\omega, v)$

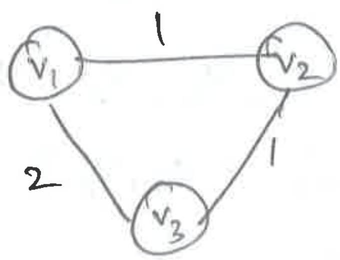


$(v_1, v_2) = (v_2, v_1)$

WEIGHTED GRAPH

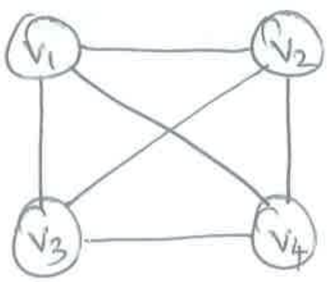
A graph is said to be weighted graph if every edge in the graph is assigned a weight or value. It can be directed or undirected graph.





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 $\frac{n(n-1)}{2}$ edges.



No. of vertices = 4

$$\text{No. of edges} = \frac{4(4-1)}{2}$$

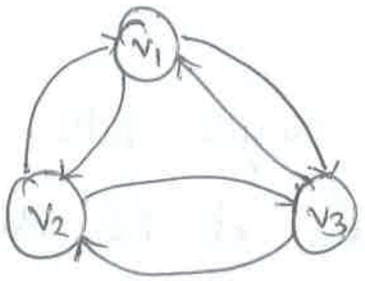
$$= \frac{4 \times 3}{2} = 2 \times 3 = 6.$$

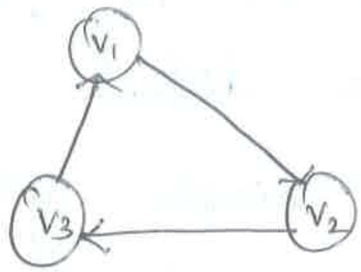
No. of edges = 6.

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.

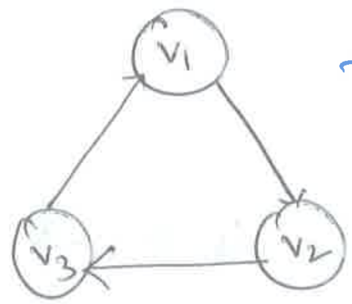
Otherwise, it is said to be weakly connected graph.





PATH

A path in a graph is a sequences of vertices $(\omega_1, \omega_2 \ldots \omega_n)$ such that $\omega_i, \omega_{i+1} \in E$ for $1 \le i \le N$.



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 $\underline{N-1}$, where N represents number of vertices.

The length of path V_1 to V_3 is 2

$(V_1\ V_2)\ (V_2\ V_3)$

LOOP

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

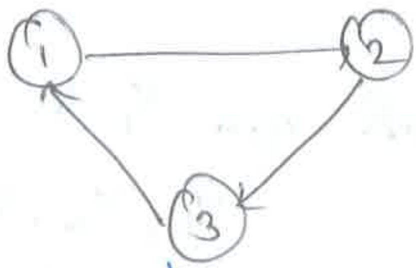
SIMPLE PATH

A simple path is a path, such that all the vertices on the path, except possibly the first and 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 same



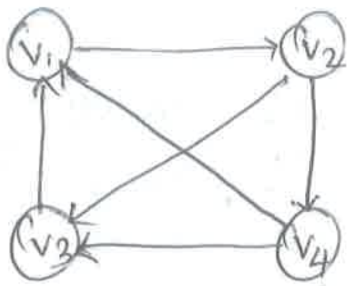
A graph which has cycles is termed to be cyclic graph.

DEGREE: The number of edges, incident on a vertex determines its degree. The degree of vertex v is written as degree(v)

③

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

Outdegree of vertex 'v' is the number of edges exiting from that vertex v.



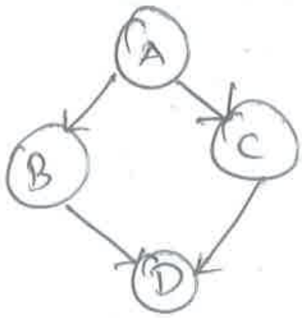
Indegree $(V_1) = 2$

Out degree $(V_1) = 1$

ACYCLIC GRAPH

A directed graph, which has no cycles is referred to as acyclic graph.

Also called as Directed Acyclic graph (DAG)



REPRESENTATION OF GRAPH

Graphs can be represented using

→ Adjacency Matrix

→ Adjacency list.

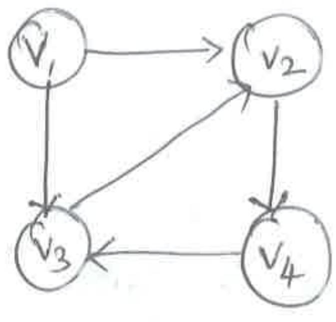
(i) ADJACENCY MATRIX

The adjacency matrix 'A' for a graph $G = (V, E)$ with 'n' vertices is an $n \times n$ matrix, such that

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

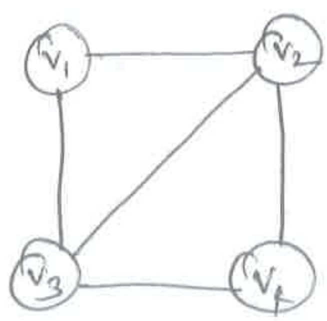
$A_{ij} = 0$, if there is no edge.

ADJACENCY MATRIX FOR DIRECTED GRAPH



$$\begin{array}{c|cccc} & V_1 & V_2 & V_3 & V_4 \\ \hline V_1 & 0 & 1 & 1 & 0 \\ V_2 & 0 & 0 & 0 & 1 \\ V_3 & 0 & 1 & 0 & 0 \\ V_4 & 0 & 0 & 1 & 0. \end{array}$$

ADJACENCY MATRIX FOR UNDIRECTED GRAPH



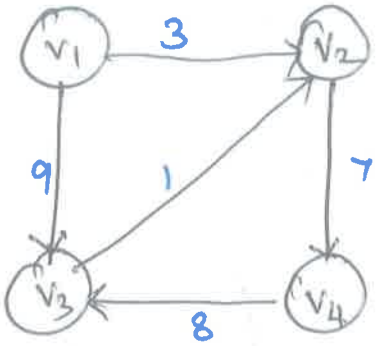
$$\begin{array}{c|cccc} & V_1 & V_2 & V_3 & V_4 \\ \hline V_1 & 0 & 1 & 1 & 0 \\ V_2 & 1 & 0 & 1 & 1 \\ V_3 & 1 & 1 & 0 & 1 \\ V_4 & 0 & 1 & 1 & 0 \end{array}$$

ADJACENCY MATRIX FOR WEIGHTED GRAPH.

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

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

$A_{ij} = \infty$ if $i \neq j$ and no edge bet'n i & j



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

ADVANTAGE.

→ Simple to implement

DISADVANTAGE

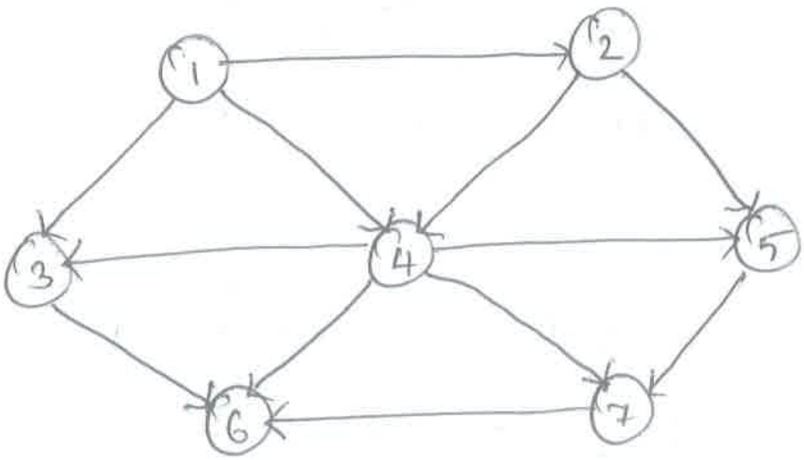
→ Space complexity is $O(n^2)$

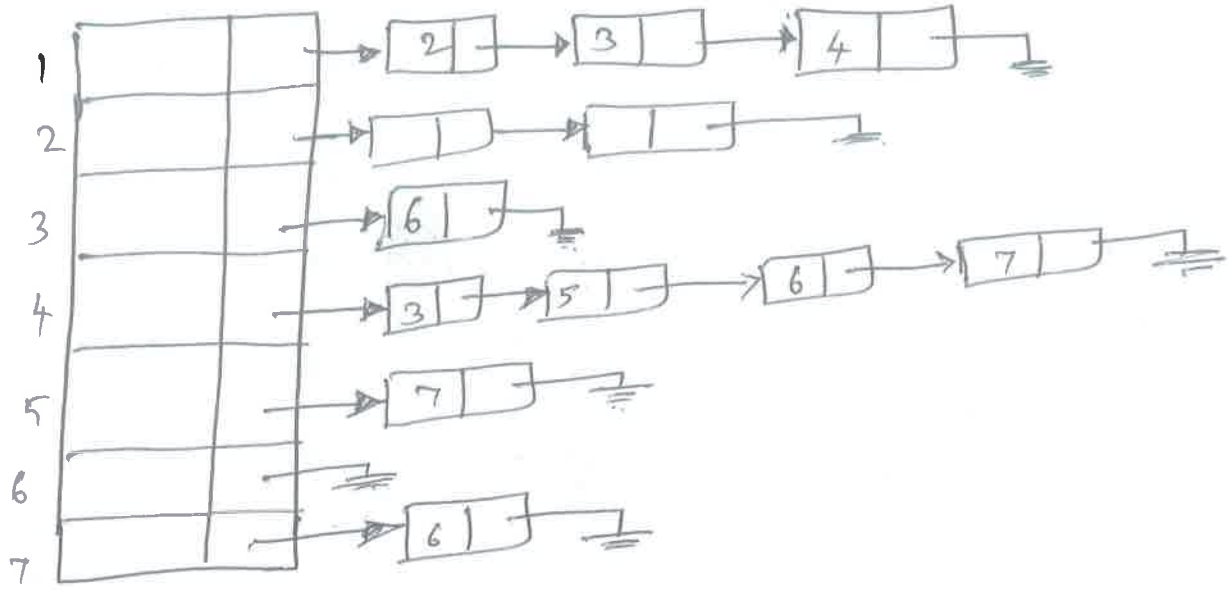
→ Time complexity is $O(n^2)$

(ii) ADJACENCY LIST REPRESENTATION

→ A graph is stored as linked structure.

→ store all vertices in a list and then for each vertex, maintain a linked list of its adjacency vertices.





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 'ω' on the cycle, 'v' precedes 'ω' and 'ω' precedes 'v'.

STEPS.

1. Find the indegree of every vertex.
2. Place the vertices whose indegree is '0' on the empty Queue.
3. Dequeue the vertex 'v' and decrement the indegree's of all its adjacent vertices.
4. Enqueue the Vertex on queue, if indegree falls to

5 Repeat from step 3 until Queue becomes empty.

Topological ordering is the order in which the vertices are dequeued.

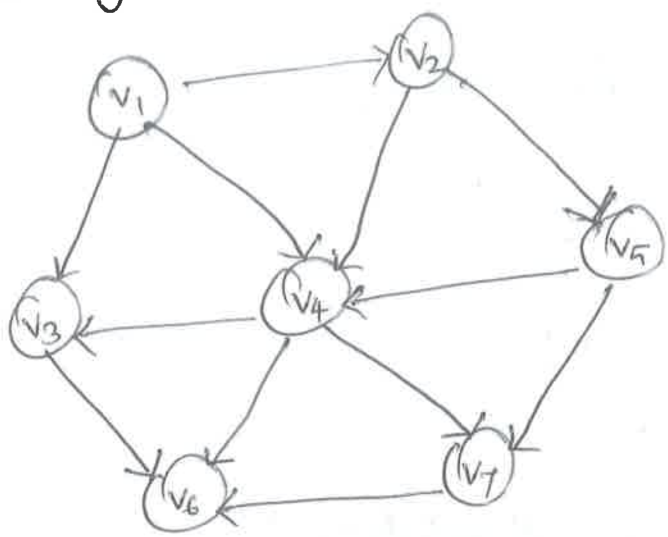
ALGORITHM

```
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)
    }
```

DisposeQueue (Q); // Free the memory.

}

FIND THE TOPOLOGICAL ordering of the following graph.



Step 1: Construct 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 $[V_3] = 2$ Indegree $[V_5] = 1$

Indegree $[V_2] = 1$ Indegree $[V_4] = 3$ Indegree $[V_6] = 3$

Indegree $[V_7] = 2$

Step 2:

Enqueue the vertex, whose indegree is 0

Enqueue : V_1

Dequeue :

Step 3:

Dequeue the vertex V_1 from Queue and decrement the indegree's of its adjacent vertex V_2 V_3 and V_4

Indegree $[V_2] = 0$ Indegree $[V_3] = 1$

Indegree $[V_4] = 2$

Enque the vertex, whose indegree is 0

Enqueue : V_2

Dequeue : V_1

Step 4:

Dequeue Vertex V_2 from Queue and decrement the indegree of its adjacent vertex V_4 & V_5

Indegree $[V_4] = 1$ Indegree $[V_5] = 0$

Enque the vertex V_5, whose indegree falls to 0

Enque : V_5

Deque : V_1 V_2

Step 4 :

Deque the vertex V_5 and decrement the indegree of its neighbors.

Indegree $[V_4] = 0$ Indegree $[V_7] = 1$

Enque vertex V_4, whose indegree falls to 0.

Enque : V_4

Deque : V_1, V_2, V_5, $\cancel{V_4}$.

Step 5 :

~~Enque~~ Deque the vertex V_4 and decrement the indegree of its neighbors.

Indegree $[V_6] = 2$ Indegree $[V_7] = 0$ Indegree $[V_3] = 0$

Enque vertex V_3, V_7, whose indegree falls to 0

Enque : V_3, V_7

Deque : V_1 V_2 V_5 $\cancel{V_4}$ V_4

Step 6 :

Deque vertex V_3 and decrement indegree of its neighbors

Indegree [V_6] = 1

Enque : V_7

Deque : V_1 , V_2 V_5 V_4 , V_3

Step 7 :

Deque vertex V_7 and decrement indegree of its neighbor.

Indegree [V_6] = 0

Enque vertex V_6 whose indegree falls to 0.

Enque : V_6

Deque : V_1 V_2 V_5 V_4 V_3 V_7

Step 8 :

Deque vertex V_6

Deque : V_1 V_2 V_5 V_4 V_3 V_7 V_6

Indegree before Deque.

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

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

ANALYSIS

The running time of algorithm is

$$O(|E| + |V|)$$

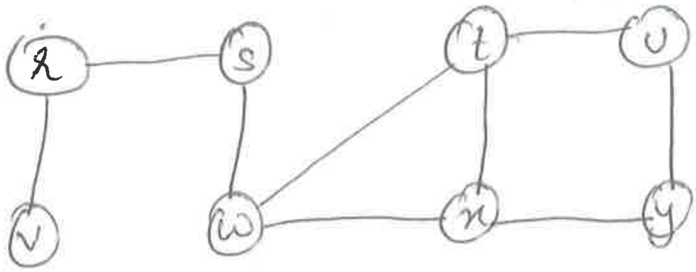
where E represents edges

V represents vertices of graph.

BREADTH FIRST SEARCH

- To find all the vertices reachable from a given source vertex s.
- BFS traverse a connected component of a given graph and defines a spanning tree.
- Data structure used is : Queue.

EXAMPLE



Take 's' as source.

Step 1.

Enqueue : s

step 2 :

deque 's' and enque its neighbours (w, r)

Enque : | w | r |

deque : | S | |

Step 3 :

deque 'w' and enque its neighbours (t, x)

Enque : | r | t | x |

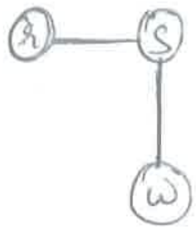
deque : | S | w |

Step 4:

deque 'r' and enque its neighbours (v)

Enque : t x v

deque : s w r

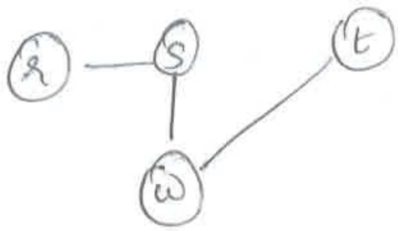


Step 4:

deque 't' and enque its neighbours (u)

Enque : x v u

deque : s w r t

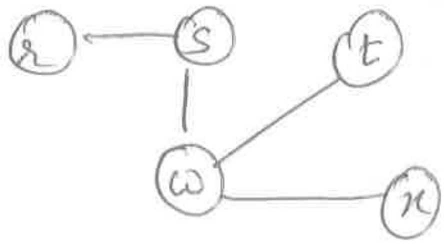


Step 5:

deque x and enque its neighbours (y)

Enque : v u y

deque : s w r t x

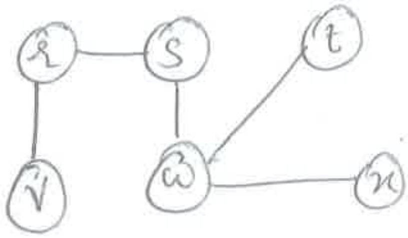


Step 6

deque v and enque its neighbours (No neig

Enque : u y

deque : s w r t x v



Step 7

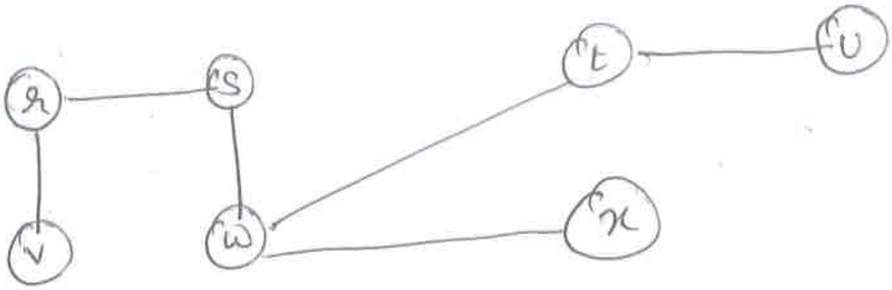
deque u and enque its neighbours

(t, y)

t is already visited. y is already in Queue.

Enque : y

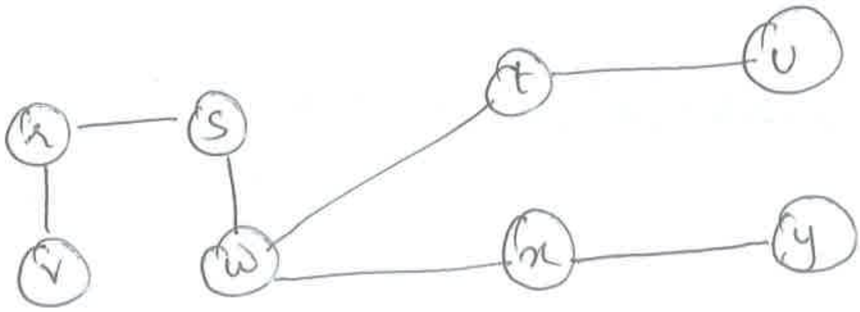
deque : s, w r t x v u



Step 8:

Deque 'y' & enque its neighbors.

neighbours of 'y' (x and t are already visited)



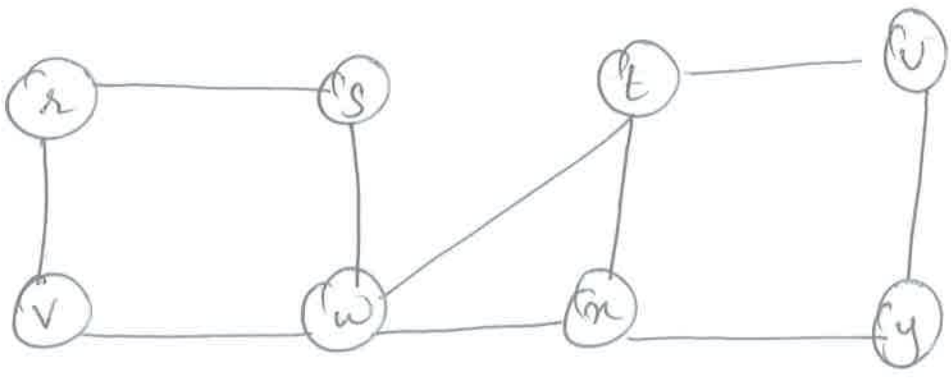
DEPTH FIRST TRAVERSAL

- Exploration of a vertex 'v' is suspended as soon as a new vertex is reached. At this time, the exploration of new vertex 'u' begins. When this new vertex has been explored, the exploration of 'v' continuous. The search process terminates when all reached vertices have been fully explored.

ALGORITHM

```
Algorithm DFS (v)
{
  visited [v] := 1
  for each vertex 'w' adjacent from 'v' do
  {
    if (visited [w] == 0) then
    {
      DFS (w);
    }
  }
}
```

EXAMPLE



Consider the source vertex 's'.

Step 1:

Push s

Pop 's' and push its neighbors

visited : s

Step 2:

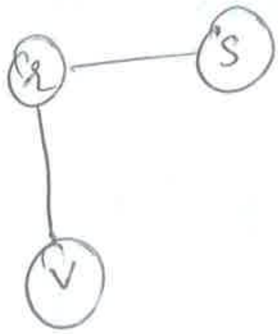
Pop 'r' and push its neighbors (v, w)

visited : s r

Step 3:

Pop 'v' and push its neighbor (w)

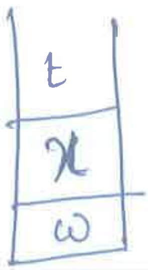
visited : s r v

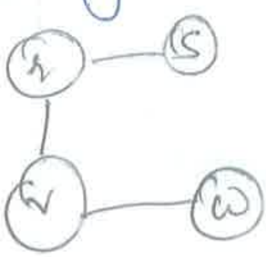


Step 4:

Pop 'w' and push its neighbor (s, t, x)

's' is already visited.

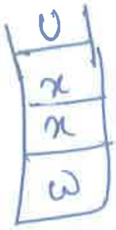




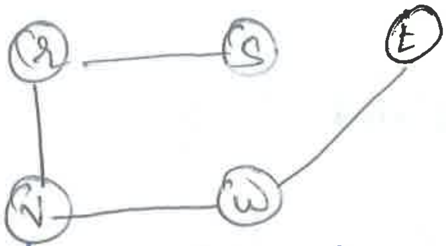
Step 5 :

Pop 't' and push its neighbours (ω, x, υ)

already visited.



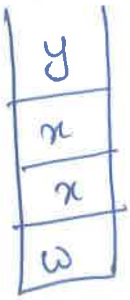
Visited : s r v ω t

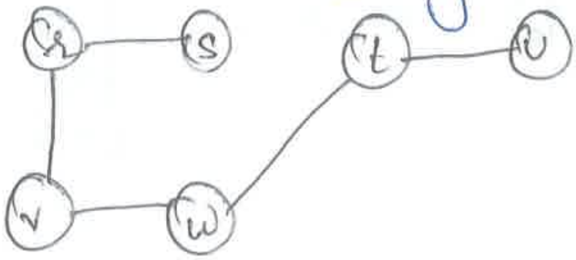


Step 6 :

pop 'υ' and push its neighbours (t, y)

already visited.



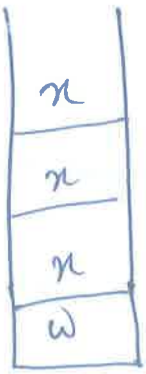


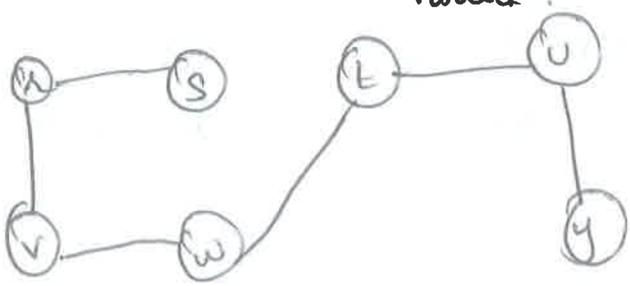
Visited : s r v ω t υ

Step 7 :

pop 'y' and push its neighbours (υ, x)

already visited



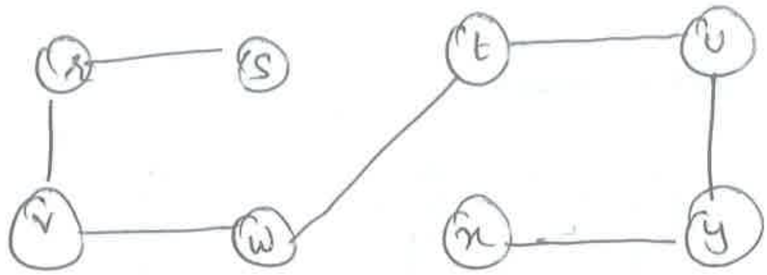


Visited : s r v ω t υ y

Step 8 :

Pop 'x' and push its neighbours. (Its neighbors are already visited)

Visited : s r v w t u y x.



MINIMUM SPANNING TREE

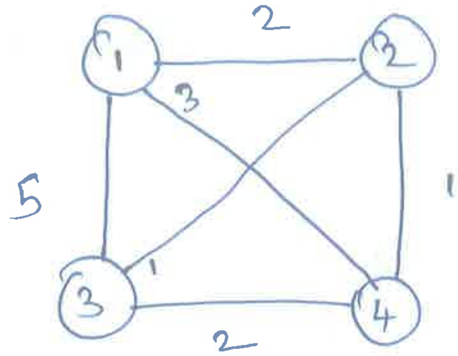
A spanning tree of a connected graph is a connected acyclic subgraph, that contains all the vertices of the graph.

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 weight on all its edges.

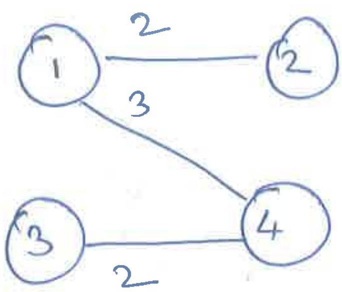
The total number of edges in Minimum Spanning tree is $|V| - 1$

where $|V|$ is the number of vertices in a graph.

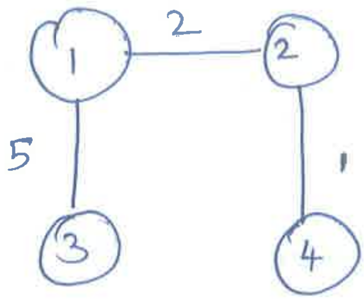
Ex :-



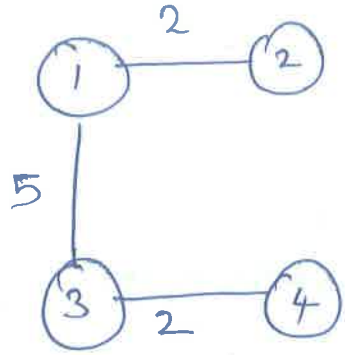
Spanning Tree for the above graph.



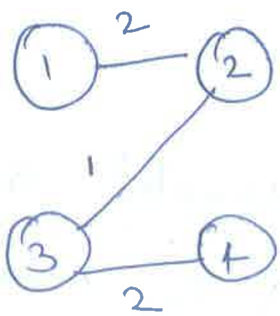
cost = 7



Cost = 8

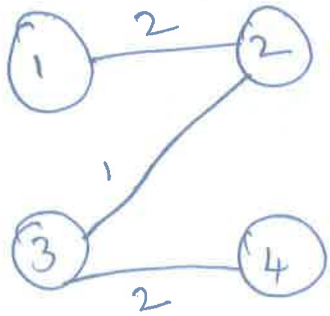


Cost = 9



cost = 5

Thus MST is



Minimum spanning tree can be constructed using

→ PRIM'S Algorithm

→ ~~D~~ Kruskal's Algorithm.

KRUSKAL'S ALGORITHM

Kruskal's algorithm use greedy strategy, to construct a minimum spanning tree. At each step, it select the edge having smallest weight and accept the edge if it does not form a cycle.

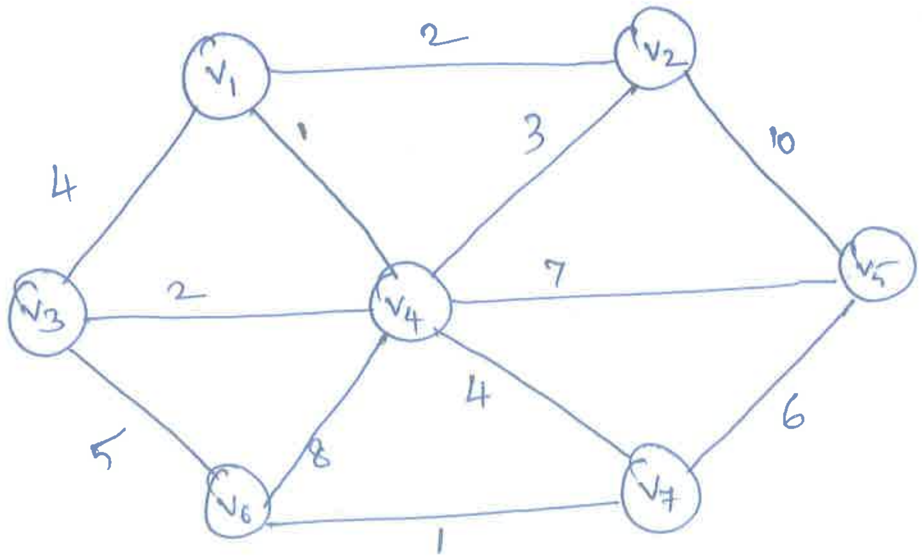
Steps:

→ Arrange the edge in ascending order of weight.

→ choose the edge, in order, if it does not form a cycle.

EXAMPLE:

Construct Minimum Spanning tree for the following graph using Kruskal's algorithm.

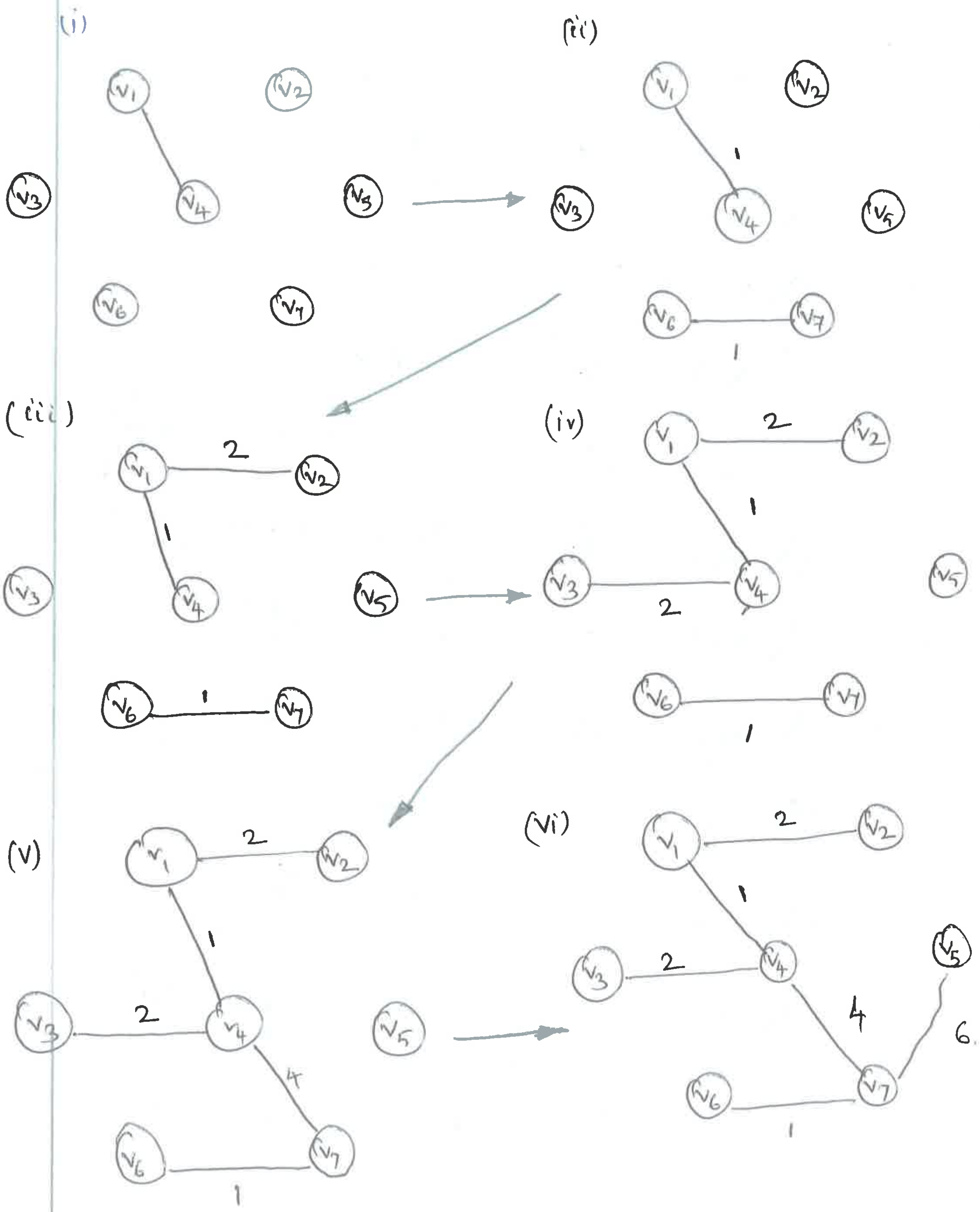


Solu:

EDGE	WEIGHT
$(V_1\ V_2)$	2
$(V_1\ V_4)$	1
$(V_1\ V_3)$	4
$(V_2\ V_4)$	3
$(V_2\ V_5)$	10
$(V_3\ V_4)$	2
$(V_3\ V_6)$	5
$(V_4\ V_6)$	8
$(V_4\ V_5)$	7
$(V_4\ V_7)$	4
$(V_5\ V_7)$	6
$(V_6\ V_7)$	1

Arrange the edges in ascending order of weight.

EDGE	WEIGHT	ACTION
$(v_1\ v_4)$	1	ACCEPT
$(v_6\ v_7)$	1	ACCEPT
$(v_1\ v_2)$	2	ACCEPT
$(v_3\ v_4)$	2	ACEEPT
$(v_2\ v_4)$	3	REJECT (FORMS cycle)
$(v_1\ v_3)$	4	Reject (forms cycle)
$(v_4\ v_7)$	4	ACCEPT
$(v_3\ v_6)$	5	Reject (form cycle)
$(v_5\ v_7)$	6	accept
$(v_4\ v_5)$	7	reject (form cycle)
$(v_4\ v_6)$	8	reject (form cycle)
$(v_2\ v_5)$	10	reject (form cycle)



$$\text{Total cost} = Cost_{v_1 v_4} + Cost_{v_6 v_7} + Cost_{v_1 v_2} + Cost_{v_3 v_4} + Cost_{v_4 v_7} + Cost_{v_5 v_7}$$

$$= 1 + 1 + 2 + 2 + 4 + 6$$

$$= 16$$

PRIM'S ALGORITHM

It grows the tree in successive stages.

STEPS :

(i) For source vertex

Set known to 0

d_v to 0

P_v to 0

(ii) For all vertices, except source

Set known to 0

d_v to α

P_v to 0

(iii) After visiting a vertex v,

set v. known = 1,

if there exists neighbours of 'v' say 'w'

Compute :

$$w.dist = Min\left[w.dist,\ c_{vw} \right]$$

c_{vw} → represent cost bet'n the edge v & w.

if first argument of 'Min' is small

w.path remain same

else

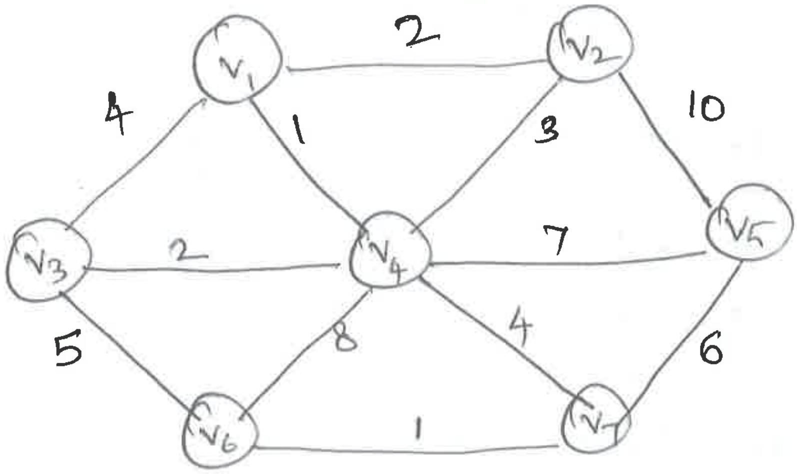
w.path = v

At each step, choose the vertex having min distance.

Repeat step 3 till all the vertices are visited.

Ex:

Construct spanning tree for the graph shown below using Prim's Algorithm.



Step 1: Consider 'v_1' as source vertex.

INITIAL TABLE

VERTEX	KNOWN	d_v	P_v
v_1	0	0	0
v_2	0	α	0
v_3	0	α	0
v_4	0	α	0
v_5	0	α	0
v_6	0	α	0
v_7	0	α	0

Visit : V_1

$$\boxed{V_1 \cdot \text{known} = 1}$$

Neighbours of V_1 are V_2, V_4, V_3

↳ Compute distance ~~of~~ and path, if the neighbours are not visited.

$$V_2 \cdot dist = \text{Min}\left[V_2 \cdot dist,\ C_{V_1 V_2}\right] = \text{Min}[\alpha, 2] = 2$$

$V_2 \cdot path = V_1$

$$V_3 \cdot dist = \text{Min}\left[V_3 \cdot dist,\ C_{V_1 V_3}\right] = \text{Min}[\alpha, 4] = 4$$

$V_3 \cdot path = V_1$

$$V_4 \cdot dist = \text{Min}\left[V_4 \cdot dist,\ C_{V_1 V_4}\right] = \text{Min}[\alpha, 1] = 1$$

$V_4 \cdot path = V_1$

Vertex	known	d_v	P_v
V_1	1	0	0
V_2	0	2	V_1
V_3	0	4	V_1
V_4	0	1	V_1
V_5	0	α	0
V_6	0	α	0
V_7	0	α	0

Spanning tree

Step 2:

The vertex having minimum distance and not yet visited is 'V_4'.

visit : V_4

$\boxed{V_4 \cdot known = 1}$

Neighbors of V_4 are V_1 V_2 V_3 V_5 V_6 V_7.

V_1 is already visited.

$V_2 \cdot dist = Min[V_2 \cdot dist, C_{V_4 V_2}]$
$= Min[2, 3]$
$= 2$

$V_2 \cdot path = V_1$
//first argument of Min is small thus path remains same.

$V_3 \cdot dist = Min[V_3 \cdot dist, C_{V_4 V_3}]$
$= Min[4, 2]$
$= 2$

$V_3 \cdot path = V_4$

$V_5 \cdot dist = Min[V_5 \cdot dist, C_{V_4 V_5}]$
$= Min[\alpha, 7]$
$= 7$

$V_5 \cdot path = V_4$

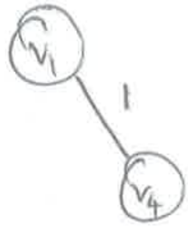
$V_6 \cdot dist = Min[V_6 \cdot dist, C_{V_4 V_6}]$
$= Min[\alpha, 8]$
$= 8$

$V_6 \cdot path = V_4$

$V_7 \cdot dist = Min[V_7 \cdot dist, C_{V_4 V_7}]$
$= Min[\alpha, 4]$
$= 4$

$V_7 \cdot Path = V_4$

Vertex	known	d_v	P_v
V_1	1	0	0
V_2	0	2	V_1
V_3	0	2	V_4
V_4	1	1	V_1
V_5	0	7	V_4
V_6	0	8	V_4
V_7	0	4	V_4



Step 3:

The vertex having minimum distance and not yet visited is V_2.

Visit : V_2

$\boxed{V_2 \cdot known = 1}$

Neighbors of V_2 are V_1, V_4 and V_5

V_1 and V_4 are already visited.

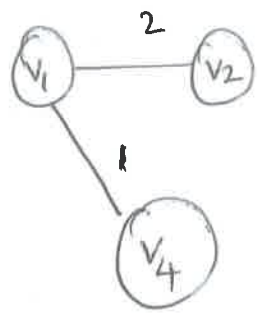
$V_5 \cdot dist = Min\,[V_5 \cdot dist,\ C_{V_2 V_5}]$

$= Min\,(7, 10)$

$= 7$

$V_5 \cdot path = V_4$

Vertex	known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	0	2	V_4
V_4	1	1	V_1
V_5	0	7	V_4
V_6	0	8	V_4
V_7	0	4	V_4



Step 4:

The vertex having minimum distance and not yet visited is V_3.

Visit : V_3

$\boxed{V_3 \cdot \text{known} = 1}$

Neighbours of V_3 are V_4, V_6, V_1

V_4 and V_1 are already visited.

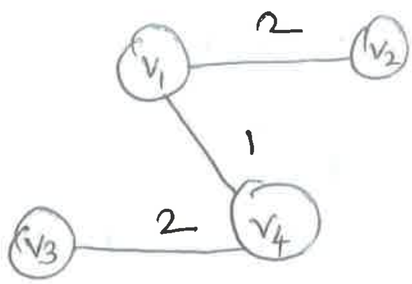
$$V_6 \cdot \text{dist} = \text{Min}\left[V_6 \cdot \text{dist},\ C_{V_3 V_6}\right]$$

$$= \text{Min}\left[8, 5\right]$$

$$= 5$$

$V_6 \cdot \text{path} = V_3$

Vertex	known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	2	V_4
V_4	1	1	V_1
V_5	0	7	V_4
V_6	0	5	V_3
V_7	0	4	V_4



Step 5 :

The Vertex having minimum distance and not yet visited is V_7.

Visit : V_7

$\boxed{V_7 . \text{known} = 1}$

Neighbours of V_7 are V_4, V_5 and V_6.

V_4 is already visited.

$V_5 . dist = Min [V_5 . dist , C_{V_7 V_5}]$

$= Min [7, 6]$

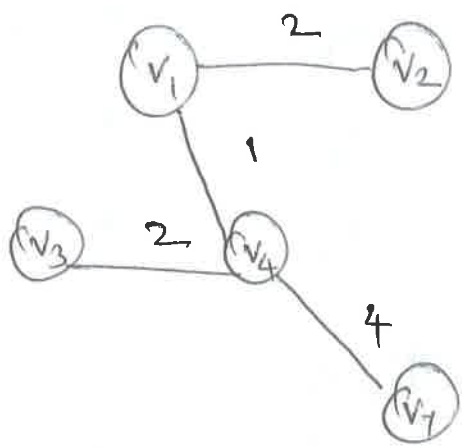
$= 6$

$V_5 . path = V_7$

$$V_6.dist = Min[V_6.dist, C_{V_7V_6}]$$
$$= Min[5, 1]$$
$$= 1$$

$V_6.path = 1$

VERTEX	KNOWN	d_V	P_V
V_1	1	0	0
V_2	1	2	V_1
V_3	1	2	V_4
V_4	1	1	V_1
V_5	0	6	V_7
V_6	0	1	V_7
V_7	1	4	V_4



Step 6 :

The vertex having min distance and not yet visited is V_6.

Visit : V_6

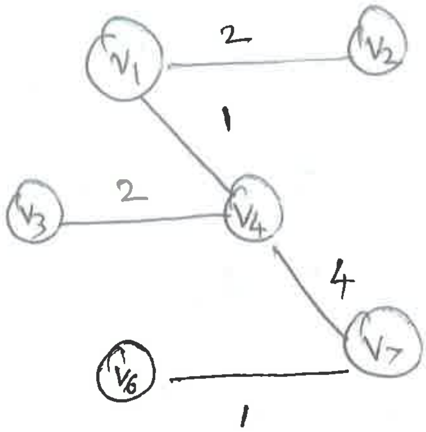
$\boxed{V_6.known = 1}$

Neighbors of V_6 are V_3 and V_7.

Both V_3 and V_7 are already visited.

Thus,

Vertex	known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	2	V_4
V_4	1	1	V_1
V_5	0	6	V_7
V_6	1	1	V_7
V_7	1	4	V_4



<u>step 7.</u>

Remaining unseen vertex is V_5

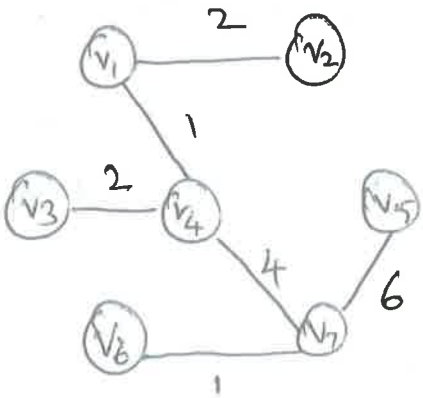
Visit : V_5

$\boxed{V_5 \cdot known = 1}$

Neighbors of V_5 are V_2 , V_4 , V_7

All its Neighbors are already visited.

Vertex	known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	2	V_4
V_4	1	1	V_1
V_5	1	6	V_7
V_6	1	1	V_7
V_7	1	4	V_4



Total cost is

$$\text{Cost}_{V_2V_1} + \text{Cost}_{V_4V_1} + \text{Cost}_{V_5V_7} + \text{Cost}_{V_6V_7} + \text{Cost}_{V_7V_4} + \text{Cost}_{V_3V_4}$$

$$= 2 + 1 + 2 + 4 + 6 + 1$$

$$= 16 \text{ //.}$$

SHORTEST PATH ALGORITHM.

INPUT : A weighted graph.

cost between the edges V_i and V_j is represented as C_{ij}

Consider the path $V_1, V_2 \ldots V_N$ is

$$\sum_{i=1}^{N-1} C_{i,i+1}$$

The above equation is termed to be weighted path length.

DIJKSTRA'S ALGORITHM

Input : Source vertex S.

To find : the shortest path from source to every vertex in a weighted graph.

Weight is assigned to each edge.

Steps :

(i) For Source vertex, set

known to 0

d_v to 0

P_v to 0

(ii) For all vertices,

known is set to 0.

d_v is set to ∞

P_v is set to 0

(iii) When visiting a vertex v,

set v.known = 1.

Say the neighbors of v as w,

$$w.dist = \text{Min}\{ w.dist,\ v.dist + C_{v,w} \}$$

If the first argument of Min, is minimum, then

w.path remains same

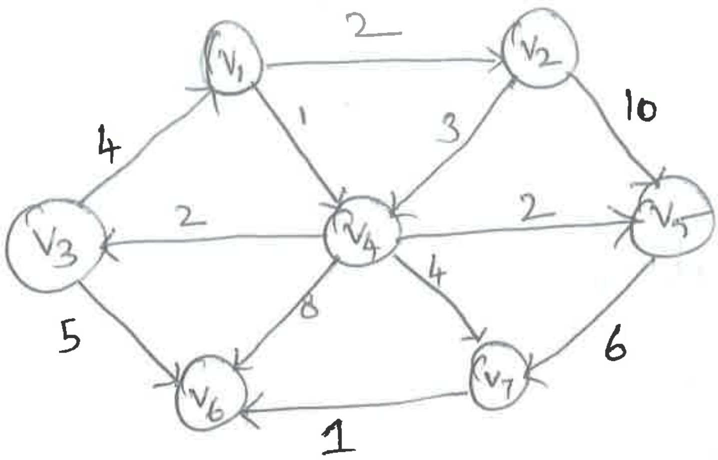
else

w.path = v.

(iv) repeat step (iii), till all the vertices are visited.

EXAMPLE:

Consider the graph, below, find the shortest path using Dijkstra's algorithm.



Soln:

Step 1:

Consider the Source Vertex as V_1

Initial table:

Vertex	Known	d_v	P_v
V_1	0	0	0
V_2	0	∞	0
V_3	0	∞	0
V_4	0	∞	0
V_5	0	∞	0
V_6	0	∞	0
V_7	0	∞	0

Visit : V_1

$\boxed{V_1 \cdot \text{known} = 1}$

Neighbors of V_1 are V_2 and V_4

$$V_2 \cdot dist = Min\left[V_2 \cdot dist,\ V_1 \cdot dist + C_{V_1 V_2}\right]$$

$$= Min\left[\alpha,\ 0+2\right]$$

$$= 2$$

$V_2 \cdot path = V_1$

$V_2 \cdot path = V_1$

$$V_4 \cdot dist = Min\left[V_4 \cdot dist,\ V_1 \cdot dist + C_{V_1 V_4}\right]$$

$$= Min\left[\alpha,\ 1\right]$$

$V_4 \cdot dist = 1$.

$V_4 \cdot path = V_1$

Vertex	known	d_v	P_v
V_1	1	0	0
V_2	0	2	V_1
V_3	0	α	0
V_4	0	1	V_1
V_5	0	α	0
V_6	0	α	0
V_7	0	α	0

Step 2:

The vertex having minimum distance and not yet visited is V_4.

$$\boxed{V_4 \cdot known = 1}$$

Neighbors of V_4 are V_3, V_5, V_6, V_7.

$$V_3 \cdot dist = Min(V_3 \cdot dist,\ V_4 \cdot dist + C_{V_4 V_3})$$
$$= Min(\alpha,\ 1+2)$$
$$= 3$$

$V_3 \cdot path = V_4$

$$V_5 \cdot dist = Min(V_5 \cdot dist,\ V_4 \cdot dist + C_{V_4 V_5})$$
$$= Min(\alpha,\ 1+2)$$
$$= 3$$

$V_5 \cdot path = V_4$

$$V_6 \cdot dist = Min(V_6 \cdot dist,\ V_4 \cdot dist + C_{V_4 V_6})$$
$$= Min(\alpha,\ 1+8)$$
$$= 9$$

$V_6 \cdot path = V_4$

$$V_7 \cdot dist = Min(V_7 \cdot dist,\ V_4 \cdot dist + C_{V_4 V_7})$$
$$= Min(\alpha,\ 1+4)$$
$$= 5$$

$V_7 \cdot path = V_4$

VERTEX	KNOWN	d_v	P_v
V_1	1	0	0
V_2	0	2	V_1
V_3	0	3	V_4
V_4	1	1	V_1
V_5	0	3	V_4
V_6	0	9	V_4
V_7	0	5	V_4

Step 3:

The vertex which is not visited and having minimum distance is V_2

$$\boxed{V_2 \cdot known = 1}$$

Neighbors of V_2 are V_4 and V_5

V_4 is already visited.

$$V_5 \cdot dist = Min\left[V_5 \cdot dist,\ V_2 \cdot dist + C_{V_2 V_5}\right]$$

$$= Min\left[3,\ 2+10\right]$$

$$= 3$$

$V_5 \cdot path = V_4$

// path remains Same.

Vertex	Known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	0	3	V_4
V_4	1	1	V_1
V_5	0	3	V_4
V_6	0	9	V_4
V_7	0	5	V_4

Step 4

The vertex whic is not visited and having minimum distance is V_3

$$\boxed{V_3 \cdot known = 1}$$

Neighbors of V_3 are V_1, V_6

$V_1 \rightarrow$ already visited.

$V_6.dist = Min[V_6.dist, V_3.dist + C_{V_3V_6}]$ | $V_6.path = V_3$

$= Min[9, 3+5]$

$= Min[9, 8]$

$= 8$

Vertex	Known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	0	3	V_4
V_6	0	8	V_3
V_7	0	5	V_4

Step 5:

The vertex having minimum distance and not yet visited is V_5

$\boxed{V_5.known = 1}$

Neighbors of V_5 is V_7

$V_7 . dist = Min \{ V_7 . dist , V_5 . dist + C_{V_5 V_7} \}$ | $V_7 . path = V_4$

$= Min \{ 5, 3+6 \}$

$= 5$

Vertex	Known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	0	8	V_3
V_7	0	5	V_4

Step 6:

The Vertex having minimum distance and not yet visited is V_7.

$\boxed{V_7 . known = 1}$

Neighbors of V_7 is V_6

$V_6 . dist = Min \{ V_6 . dist , V_7 . dist + C_{V_6 V_7} \}$ | $V_6 . path = V_7$

$= Min \{ 8, 5+1 \}$

$= 6$

Vertex	Known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	0	6	V_7
V_7	1	5	V_4

Step 7 :-

Visit the remaining unseen vertex i.e V_6

Visit : V_6

$\boxed{V_6 \cdot known = 1}$

No Neighbors for V_6.

Vertex	Known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	0	6	V_7
V_7	1	5	V_4

BELLMAN FORD ALGORITHM

Algorithm

for each vertex.

V.weight = ∝

V.predecessor = 0

for source vertex 's'

s.weight = 0

s.predecessor = 0

for i = 1 to |v| - 1

{ for each edge (u, v) ∈ E

{

if u.weight + weight (u, v) < v.weight

{

v.weight := u.weight + weight (u, v)

v.predecessor := u ;

}

}

}

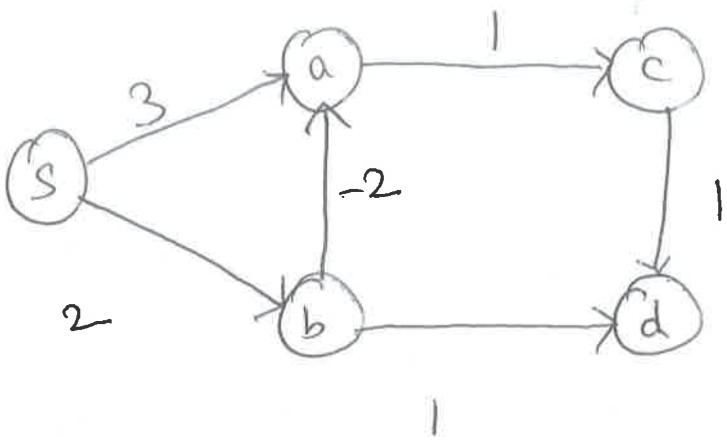
//check for Negative Weight cycles

for each edge (u, v) ∈ E

if u.weight + weight (u, v) < weight [v]

Print graph contain Negative Weight cycle

Solve the following graph using Bellman Ford Algorithm



Soln:

Step 1:

Construction of Initial table

Vertices	Weight	Predecessor
S	0	0
a	α	0
b	α	0
c	α	0
d	α	0

Step 2

$i = 1$ // i.e path length from Source to other vertex is 1.

$S \longrightarrow a$

$S \longrightarrow b$

$S \longrightarrow a$.

$$S.\text{weight} + \text{weight}(S, a) < a.\text{weight}$$

$$0 + 3 < \alpha$$

$$\boxed{\begin{array}{l} a.\text{weight} = 3 \\ a\ \text{predecessor} = S \end{array}}$$

$S \longrightarrow b$.

$$S.\text{weight} + \text{weight}(S, b) < b.\text{weight}$$

$$0 + 2 < \alpha$$

$$2 < \alpha$$

$$\boxed{\begin{array}{l} b.\text{weight} = 2 \\ b.\text{predecessor} = S \end{array}}$$

Vertices	Weight	Predecessor
S	0	0
a	3	S
b	2	S
c	α	0
d	α	0

Step 3.

$i=2$ // path length from s to other vertex is 2

S $\rightarrow$ a $\rightarrow$ c

S $\rightarrow$ b $\rightarrow$ a

S $\rightarrow$ b $\rightarrow$ d

S $\rightarrow$ a $\rightarrow$ c.

(a, c)

a.weight + weight (a, c) $\leq$ c.weight

$3 + 1 \leq \alpha$

$$\boxed{\begin{array}{l} c.\text{weight} = 4 \\ c.\text{predecessor} = a \end{array}}$$

S $\rightarrow$ b $\rightarrow$ d.

(b, d)

b.weight + weight (b, d) $\leq$ d.weight

$2 + 1 \leq \alpha$

$3 \leq \alpha$

$$\boxed{\begin{array}{l} d.\text{weight} = 3 \\ d.\text{predecessor} = b \end{array}}$$

S $\rightarrow$ b $\rightarrow$ a

(b, a) b.weight + weight (b, a) $\leq$ a.weight

$2 + (-2) \leq 3$

$= 0$

$\boxed{\begin{array}{l} a.weight = 0 \\ a.predecessor = b \end{array}}$

VERTICES	WEIGHT	PREDECESSOR
s	0	0
a	0	b
b	2	s
c	4	a
d	3	b

Step 4 :

i=3 // path length from s to other vertex is 3

$s \to b \to a \to c$

~~S+~~

(a,c).

$a.weight + weight(a,c) \leq c.weight$

$0 + 1 \leq 4$

$\boxed{\begin{array}{l} c.weight = 1 \\ c.predecessor = a \end{array}}$

Vertices	Weight	Predecessor
s	0	0
a	0	b
b	2	s
c	1	a
d	3	b

Step 4:

$i = 4$

$s \to b \to a \to c \to d$

(c, d)

$$c.weight + weight(c, d) \le d.weight$$

$$1 + 1 \le 3$$

$$2 \le 3$$

$$\boxed{\begin{array}{c} d.weight = 2 \\ d.predecessor = c \end{array}}$$

Vertices	Weight	Predecessor
s	0	0
a	0	b
b	2	s
c	1	a
d	2	c