Module 4

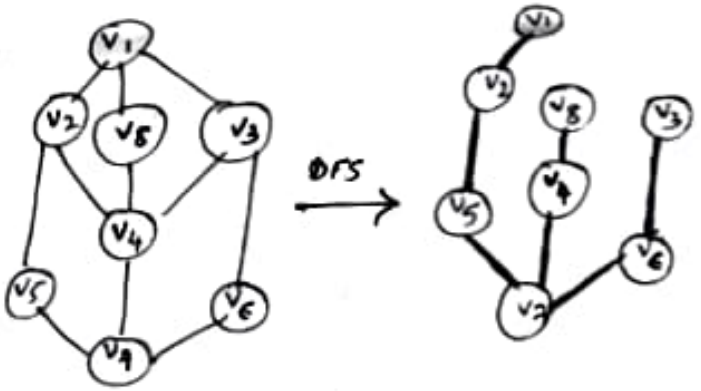
* Graphs.—
  - DFS traversal
  - BFS traversal
  - Complexity
* Spanning trees
  - Minimum Cost Spanning trees (MST)
    - → Prims Algorithm.
    - → Kruskal Algorithm.
  - Single Source Shortest path Algorithms
    - → Bellman Ford's Algorithm
    - → Dijkstra's Algorithm
* Topological Sorting.
* Strongly connected components.
  - Kosaraju's algorithm

GRAPH TRAVERSING

Traversing a graph means visiting all vertices exactly once. Several methods are known to traverse a graph systematically. Out of them 2 methods are accepted as standard. These methods are called Depth First search (DFS) and Breadth First Search (BFS).

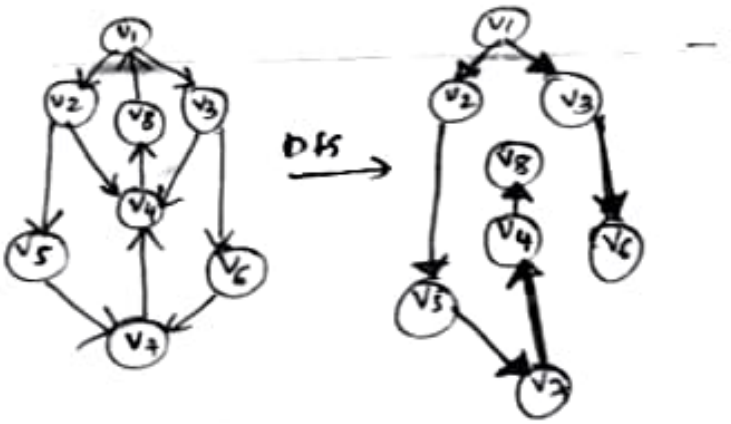
With these traversals, starting from a given node we can visit all nodes which are reachable from starting node.

DFS is similar to <u>inorder traversal</u> of binary tree. Starting from a given node, this traversal visits all nodes upto deepest level and so on.



G_1

Fig1: DFS traversal of undirected graph G_1



G_2

Fig2: DFS of directed graph G_2.

DFS (G_1): $v_1 - v_2 - v_5 - v_7 - v_4 - v_8 - v_6 - v_3$

DFS (G_2): $v_1 - v_2 - v_5 - v_7 - v_4 - v_8 - v_3 - v_6$

It can be noted that the sequence of visit depends on the depth we choose first and hence the order of visiting may not be unique.

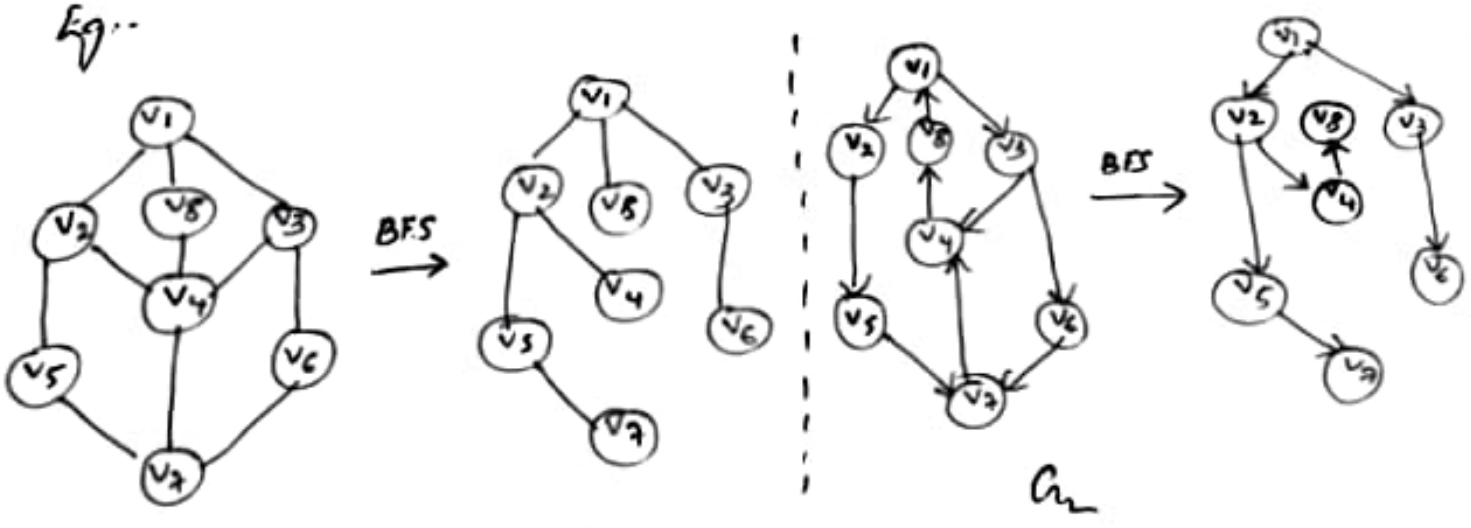
BFS & DFS Traversing

Another standard graph traversal method is Breadth First search (BFS). This traversal is similar to level-by-level traversal of a tree. Here any vertex in level i will be visited only after the visiting of preceding level (level i-1)

BFS is roughly analogous to pre order traversal of a tree. Here, suppose we are to visit the vertex v_i and v_i has $v_{i1}, v_{i2}, v_{i3}, \ldots v_{in}$ as the adjacent vertex of it. The BFS is recursively defined as

```
Traverse (vi)
    process (vi)
    traverse (vi1)
      "      (vi2)
      :
      "      (vin)
End of traversal (vi)
```

Eg:-



G_1 (of BFS)

BFS (G_1) :- $v_1 - v_2 - v_8 - v_3 - v_5 - v_4 - v_6 - v_7$

BFS (G_2) = $v_1 - v_2 - v_3 - v_5 - v_4 - v_6 - v_7 - v_8$.

Q) DFS

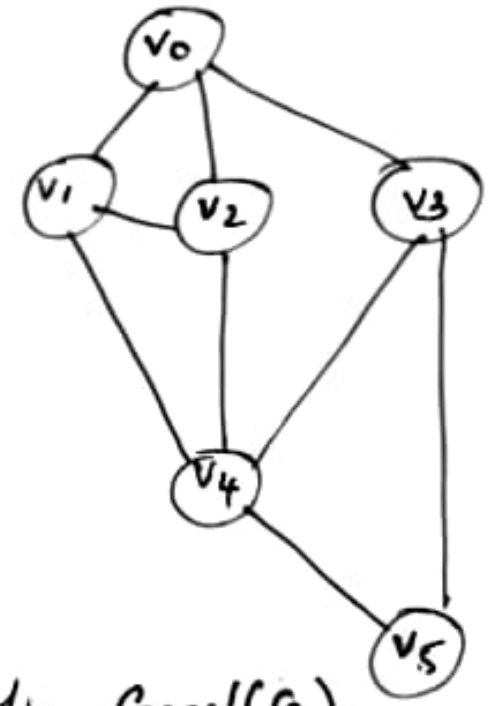


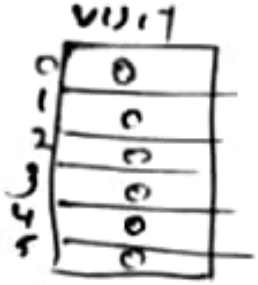
Fig:- Graph(G_1).

Adj Matrix

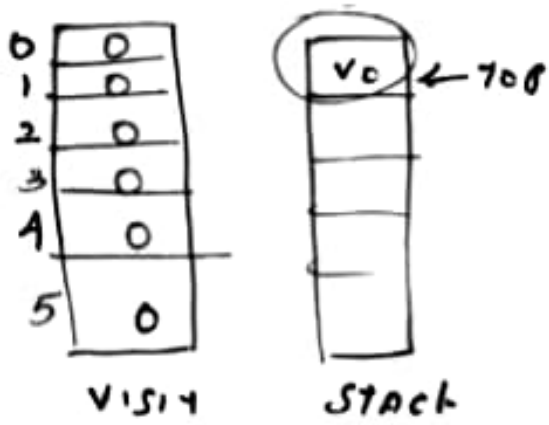
	v_0	v_1	v_2	v_3	v_4	v_5
v_0	0	1	1	1	0	0
v_1	1	0	1	0	1	0
v_2	1	1	0	0	0	0
v_3	1	0	0	0	1	1
v_4	0	1	1	1	0	1
v_5	0	0	0	1	1	0

— Fig(a) —

visit



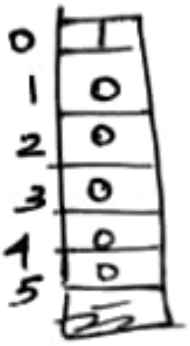
1. Push (v_0) (Slip 7)



while top ≠ -1

v = pop()

pop v_0. ⟶ v_0 print.



visit(v_0) = 1

Adj[v_0] ⇒ v_1, v_2, v_3 → if visit[v_1, v_2, v_3] =

Push v_1, v_2, v_3 to stk.

(4) Stack not empty

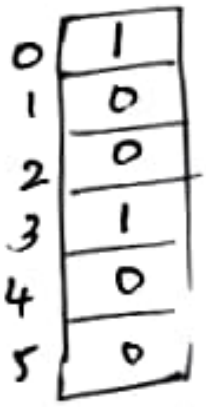
v = pop()

↓

v_3 pop.

{ visit(v_3) = 0
  so print v_3
  visit(v_3) = 1

⇒ Print v_3

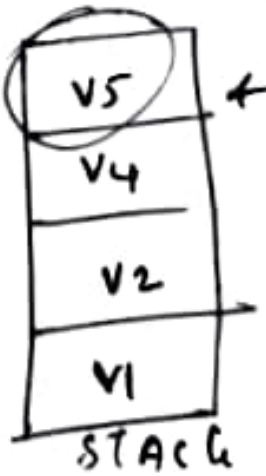


adj [v_3] = v_0, v_4, v_5

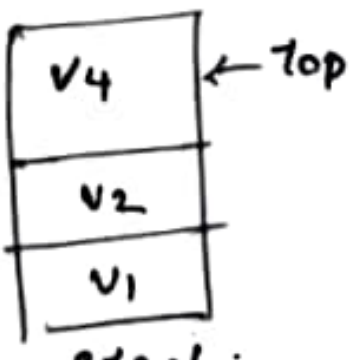
↓

already visited (visit [v_0] = 1)

So take v_4, v_5 → push to st



Pop v_5 →

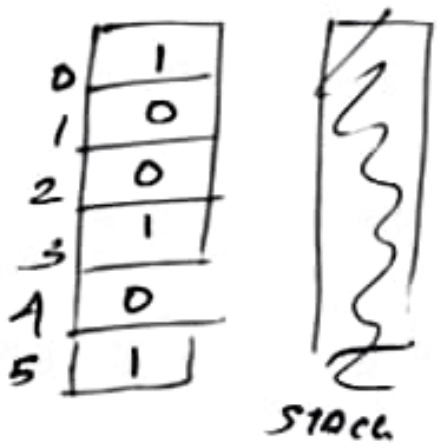


(iii) Pop v_5 — Stack not empty

$v = pop()$

↳ v_5 pop

{ visit(v_5) = 0
  so print v_5
  visit(v_5) = 1



Print v_5

adj[v_5] = v_3, v_4

↳ already visited

So Take v_4.

↳ already in Stack.

(iv) Stack not empty

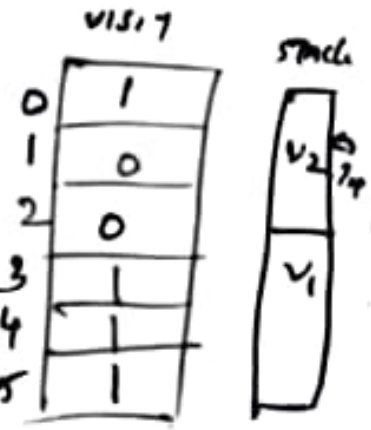
$v = pop()$

↳ v_4 poped

↳ visit(v_4) = 0

so **print v_4**

visit(v_4) = 1

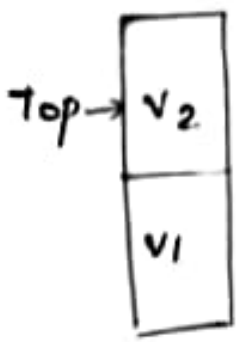


adj [v_4] → v_1, v_2, v_3, v_5

already visited

already in stack.

~~No need to push.~~



Stack.

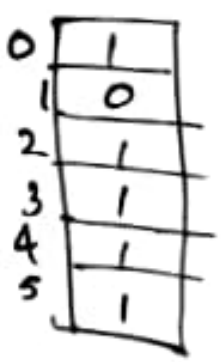
Q. Stack not empty

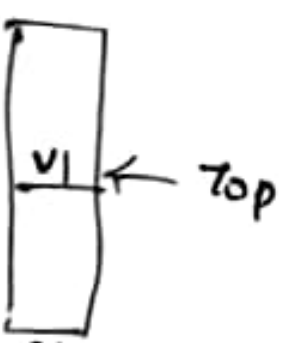
Pop v_2

visit [v_2] = 0

So **print v_2**

Make visit [v_2] = 1





Stack.

Adj [v_2] = (v_0) v_1

↳ already visited

stack not empty

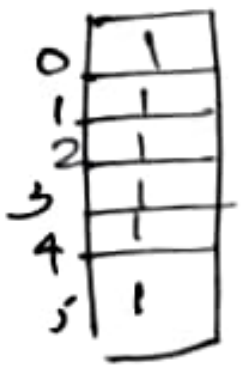
(v)

Pop v_1

↳ visit (v_1) = 0

So print v_1

make visit (v_1) = 1



Printing step → Since visit (v_i) = 1

OP ⇒ v_0, v_3, v_5, v_4, v_2, v_1

Algorithm for DFS

I/p :- V - starting vertex
N - No. of vertices currently present in graph
visit - Info^n about visited vertex, a stack with top

O/p :- Traversal of vertex reaching from V.

DS :- Matrix rep^n of graph.

Algo.

1. start
2. If (N = -1)
    1. Print "Graph is empty"
    2. exit
3. End if
4. i = 0
5. while (i < N)
    1. visit[i] = 0
    2. i = i+1
6. End while
7. push(v)
8. while (top ≠ -1) do
    1. v = pop()
    2. If (visit(v) = 0) then
        1. Print v
        2. visit(v) = 1
        3. i = 0
        4. while (i < N) do
            1. If (adj[v][i] = 1 and visit[i] = 0)
                1. push(i)
            2. Endif
            3. i = i+1
        5. End while
    3. End if
9. End while
10. Stop.

BFS traversing

Algorithm

It is similar to DFS, but instead of stack, queue is used in BFS traversing

Algo

BFS (B)

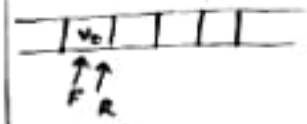
i/p :- V → starting vertex
N → No. of vertices currently present in graph.
VISIT → Info about visited vertex, a queue with REAR.

o/p :- Traversal of vertex reaching from v.

DS :- Matrix rep^n of graph.

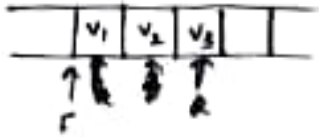
1. Start
2. If (N = -1) Then
    1. print "Graph is empty"
    2. exit
3. End if
4. i = 0
5. while (i < N) do
    1. visit[i] = 0
    2. i = i + 1
6. End while
7. Enqueue (N)
8. while (REAR ≠ -1) do
    1. v = dequeue ()
    2. If (visit(v) = 0) then
        1. print v
        2. visit(v) = 1
        3. i = 0
        4. while (i < N)
            1. If (adj[v][i] = 1 & visit[i] = 0) then
                1. Enqueue (i)
            2. End if
            3. i = i + 1
        5. End while
    3. End if
9. End while
10. Stop.

(Ref adj matrix fig (a))



Print v_0 — (dequeue it)

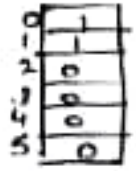
Adjacent of v_0 → v_1, v_2, v_3 → Enqueue it

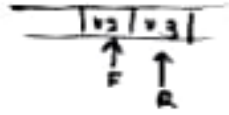


dequeue (v_1)

visit(v_1) = 1

Print v_1



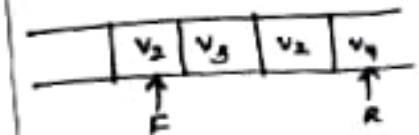


Adj of v_1 = v_0, v_1, v_4

↓

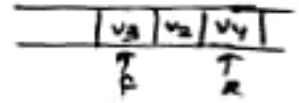
visited

Enqueue v_2, v_4

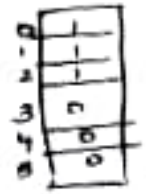


dequeue (v_2)

Print v_2



visit(v_2) = 1



Adj (v_2) = v_0, v_1 → already visited ie, visit(v_0, v_1)

no enqueuing.

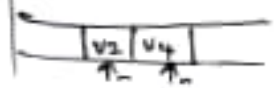
Queue not empty

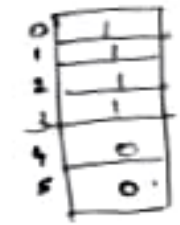
v = dequeue()

→ v_3 dequeue

Print v_3

visit(v_3) = 1

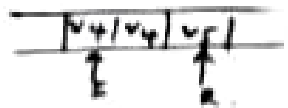




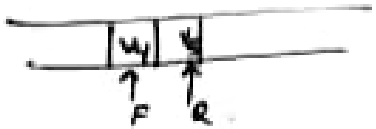
Adjacent of v_3 ⇒ v_0, v_4, v_5
↓
already visited

Enqueuing v_4, v_5

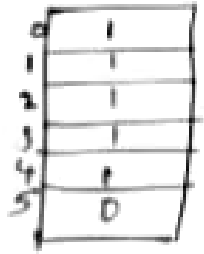
v_3 dequeue
But visit(v_3) = 1



dequeue (v_4)



visit(v_4) : 0
So print v_4
Then visit(v_4) = 1



dequeue

adj (v_4) ⇒ v_1, v_2, v_3, v_5
visited

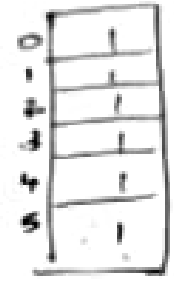
So enqueue v_5

dequeue (v_5) ⇒ Not print
become visit(v_5) = 1

dequeue (v_5)

Print v_5

Make visit(v_5) = 1



visit(v_1)

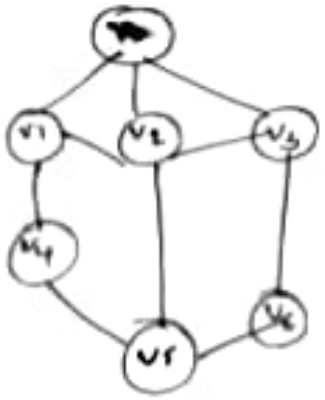
adj (v_5) = v_3, v_4
already visited

Queue also have vertex already visited
So stops.

Ans. v_0, v_1, v_2, v_3, v_4, v_5

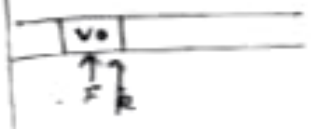
	v_0	v_1	v_2	v_3	v_4	v_5	v_6
v_0	0	1	1	1	0	0	0
v_1	1	0	1	0	1	0	0
v_2	1	1	0	1	0	1	0
v_3	1	0	1	0	0	0	1
v_4	0	1	0	0	0	1	0
v_5	0	0	1	0	1	0	1
v_6	0	0	0	1	0	1	0

Fig(b) Adj matrix



visited

0	0
1	0
2	0
3	0
4	0
5	0
6	0

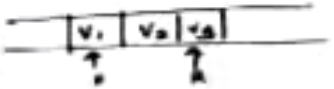


Pop v_0
Print v_0

visited

0	1
1	0
2	0
3	0
4	0
5	0
6	0

adj(v0) (and unvisited) =
v_1 v_2 v_3



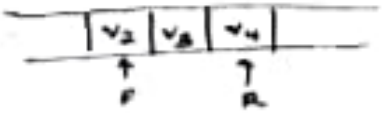
Pop v_1
Print v_1

visited

0	1
1	1
2	0
3	0
4	0
5	0
6	0

adj(v1) → ~~v_0~~, v_2, v_4
already in queue

So add v_4 to queue



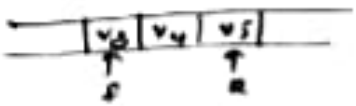
Pop v_2
Print v_2

visited

0	1
1	1
2	1
3	0
4	0
5	0
6	0

adj(v2) → ~~v_0~~, ~~v_1~~, v_3, v_5
already visited, already queue

So add v_5 to queue



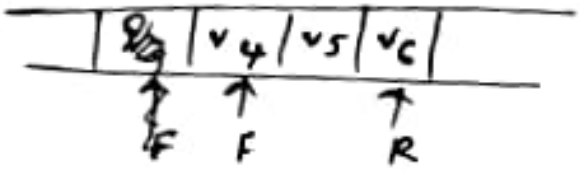
Pop v_3
Print v_3

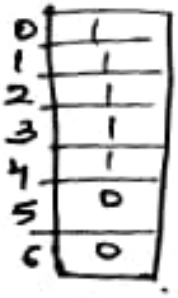
visited

0	1
1	1
2	1
3	1
4	0
5	0
6	0

adj (v_3) ⇒ ~~v_0~~, ~~v_2~~, v_6

add v_6 to queue.



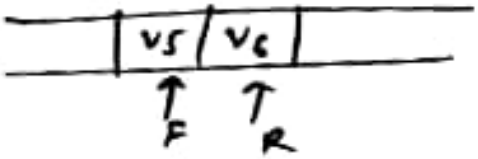


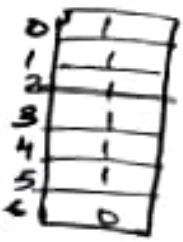
Pop v_4

Print v_4

adj (v_4) ⇒ ~~v_1~~, v_5

already v_5 in queue.





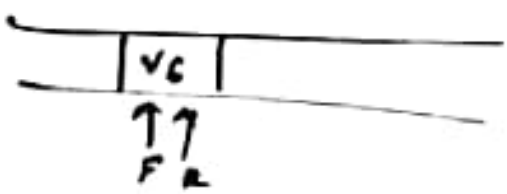
Pop v_5

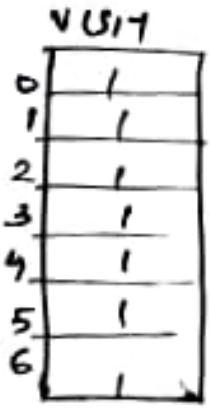
Print v_5

adj (v_5) ⇒ ~~v_3~~, ~~v_4~~, v_6

already visited

already in queue.





Pop v_6

Print v_6

$v_0, v_1, v_2, v_3, v_4, v_5, v_6$

Complexity

BFS => $O(V+E)$

DFS -> $O(V+E)$

V = vertices
E = edges

SPANNING TREES

→ It's a subset of undirected connected graph, $G=(V, E)$, which has all the vertices covered with min. no. of edges.

→ Hence a spanning tree doesnot have cycles and it cannot be disconnected.

* From a complete graph by removing max$(e-n+1)$ edges, we can construct a spanning tree

Here $e = 3$
$n = 3$
so by removing 1 edge $(3-3+1)$ we can construct spanning tree

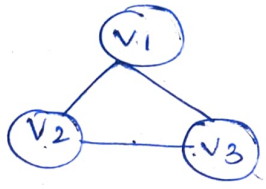
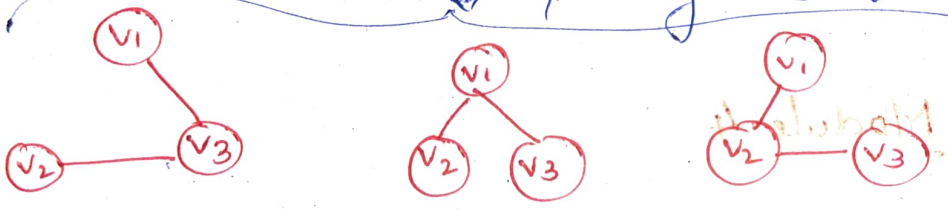


Fig:- Graph G.

⇓ spanning trees of G.



Properties of spanning trees

1) A connected graph G has more than one spanning tree
2) All possible spanning tree of graph G has same no. of edges & vertices.
3) Spanning tree doesn't have cycles (loops)
4) Spanning tree is minimally connected i.e, removing one edge from spanning tree will make graph disconnected.
5) Spanning tree is maximally acyclic i.e, adding one edge will create a loop or cycle in spanning tree
6) Spanning tree has $(n-1)$ edges; where n = no. of nodes in G

→ MINIMUM SPANNING TREE (MST)

In a weighted graph, a MST is a spanning tree that has minimum weights than all other spanning trees of same graph.

- In real world situations, the weight can be measured as
  - → distance
  - → congestion, traffic load or
  - → any arbitrary value denoted to edges.

⇒ MST Algorithms :-

1) Prim's
2) kruskal

1) PRIM'S ALGORITHM

→ It's a greedy algorithm that finds MST for a connected weighted undirected graph.

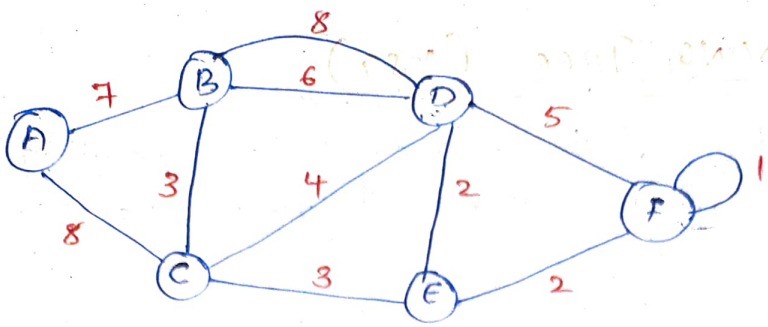
→ It finds a subset of edges that form a tree that includes every vertex, where the total weight of all the edges in a tree is maximized.

1) ~~Remove all the loops & parallel edges.~~

Steps :- (Remove all self loop & parallel edges from graph)

1) Initialize the MST with a vertex chosen at random.
2) Find all the edges that connect the tree to new vertices. Find minimum & add it to the tree
3) Keep repeating step(2) until we get a MST.

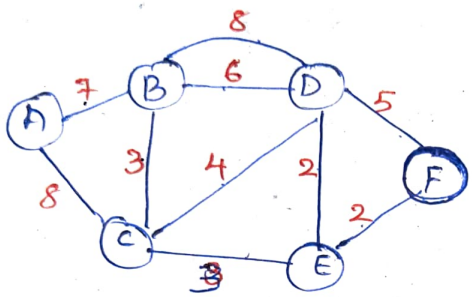
*



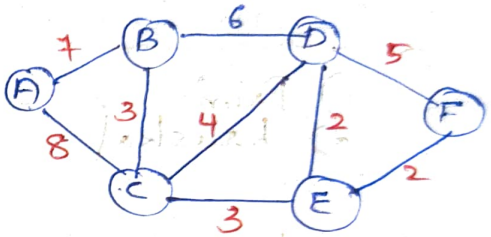
Any)

Graph, G.

- Remove loop



- Remove parallel edges.
(Remove edge with heighest weight)



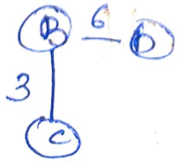
- Choose an arbitrary vertex as starting vertex
Let take A as starting vertex
Notice all outgoing / incident edges from A.
and choose min. weighted edge.

From A, there are 2 edges

1) A to B with weight 7
2) A to C " " 8.

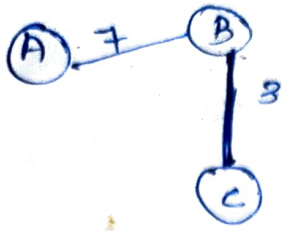
So choose the edge from A to B.

→ Find all edges from B and also from the edge from A which is not taken already.
Choose min. weighted edge.

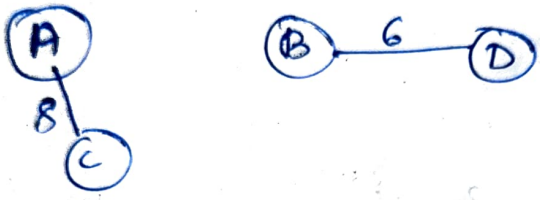


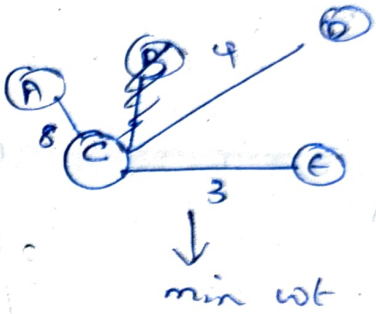
Here min weight edge is 3 (i.e, B to C).

so draw the edge from B to C



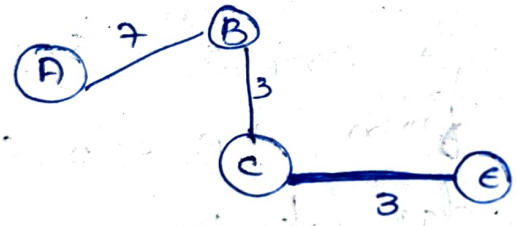
→ Next consider the edges from C. Also from A and B.



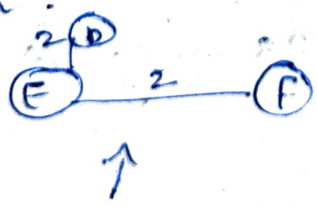


choose C to E.

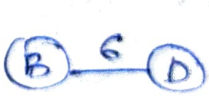
ie,

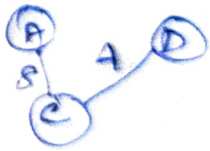


→ Next consider edges from E, also edges from A, B and C which is not taken.



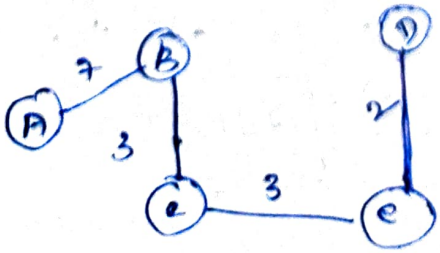
choose E to D
or
E to F



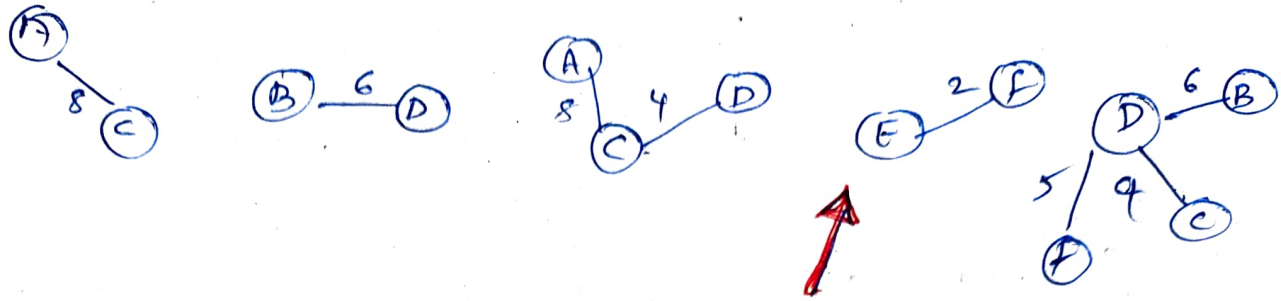


we can choose either E to D or E to F.

Say, E to D

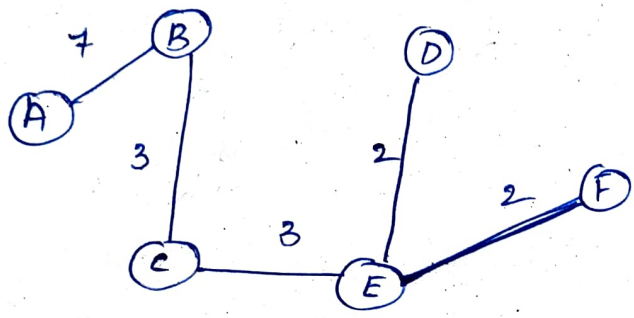


- Consider the edges from D, also from the edges from A, B, C, and E



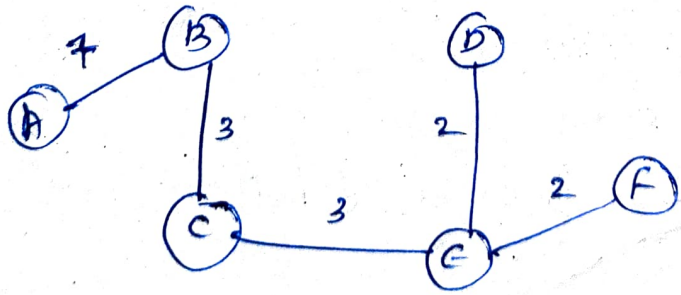
choose this

- So, the the graph will be,



- ~~No new vertex is now to consider.~~ So consider above the edges from A, B, C, D, E, F.
If we draw any edge, it forms a loop (you can check it)

- So this is the final min. spanning tree



- So the spanning tree contains all the vertices of original one (graph G). After removing selfloops & Parallel edges there contains
$n = 6$ (no. of vertices)
$e = 9$
Then no. of edges removed to get spanning tree = $e - n + 1 = 9 - 6 + 1$
$= 4$ edges removed

2) KRUSKAL ALGORITHM

→ To find MST.

→ Uses greedy approach

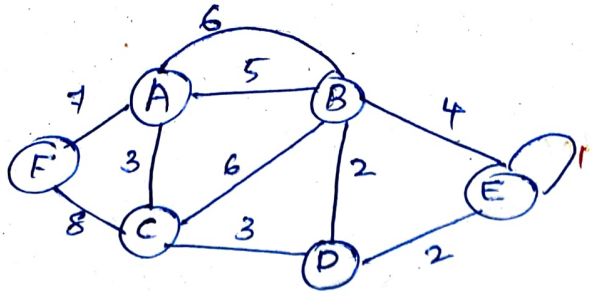
→ This ~~gre~~ algorithm treats the graph as forest and every node it has as an individual tree.

→ A tree connects to another only and only if, it has the least cost among all available options and doesn't violate MST properties.

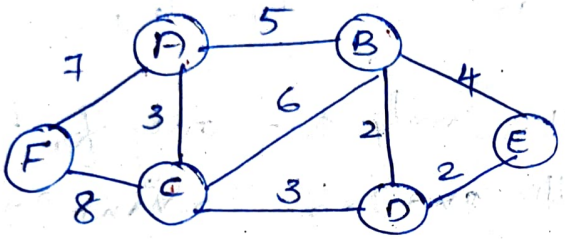
1) Sort all the edges in ~~non-decreasing~~ increasing order of their weight

2) Pick the smallest edge. check if it forms cycle with the spanning tree formed so far. If cycle is not formed, include this edge, else discard it.

3) Repeat step (2) until there are (v-1) edges in the spanning tree.



* Removing self loop and parallel edges (with min edge)



wt
2
3
4
5
6
7
8

* choose the path in the ascending order of weights.

AB = 5
AC = 3
AF = 7

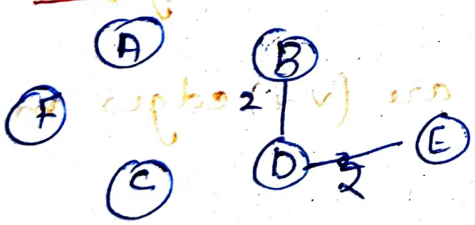
~~BA = 5~~
~~BC = 6~~
BD = 2
~~BE = 4~~

CA = 3
CB = 6
CD = 3
~~CF = 8~~

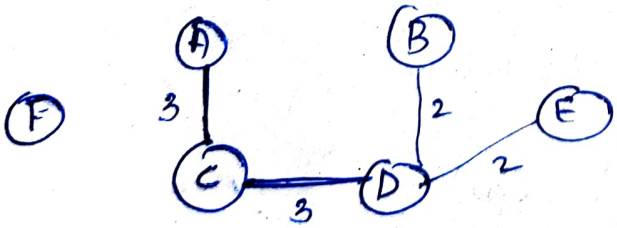
DB = 2
DC = 3
DE = 2

EB = 4
ED = 2
FA = 7
FC = 8

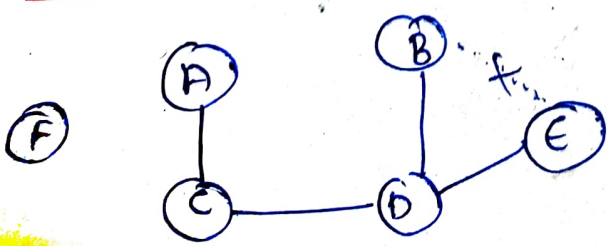
- BD, ~~DB~~, DE, ~~EB~~ — (weight 2)



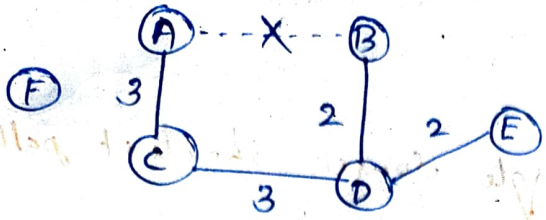
- weight = 3 → AC, CD, ~~DC~~



- weight = 4 → BE, ~~EB~~ → can't draw edge, because it form cycle.

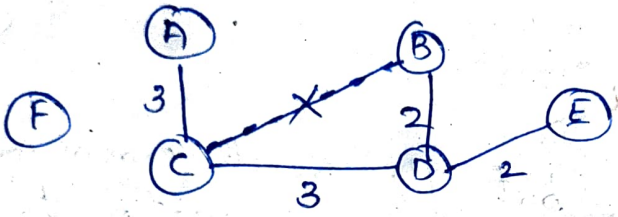


- Weight = 5 → AB, ~~BA~~



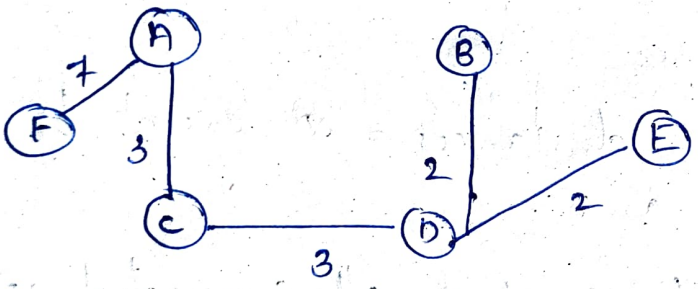
can't draw edge AB, because it forms cycle.

- Weight = 6 → Bc, ~~cB~~

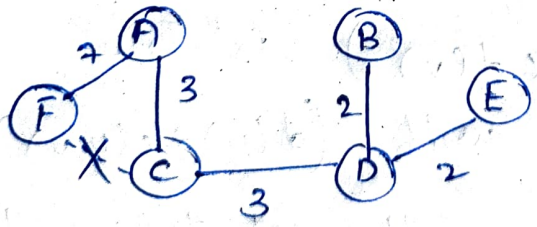


can't draw edge Bc (or cB because it forms cycle.

- Weight = 7 → AF, ~~FA~~

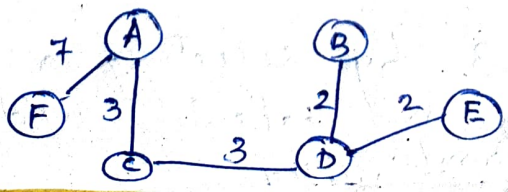


- Weight = 8 → CF, ~~FC~~



can't draw edge, because it forms cycle

→ All weights are thus considered. We consider weight in the increasing order and choose path according to that without forming cycle. Thus the MST is



SINGLE SOURCE SHORTEST PATH

To find a shortest path from a given source vertex, $s \in V$ to every vertex $v \in V$

→ 2 Algorithms for finding Single source shortest path

1. Dijikstra's Algorithm
2. BellmanFord's Algorithm

1) DIJIKSTRA'S ALGORITHM

→ It has many variants, but the most common one is to find shortest path from source vertex to all other vertices in the graph.

Steps:

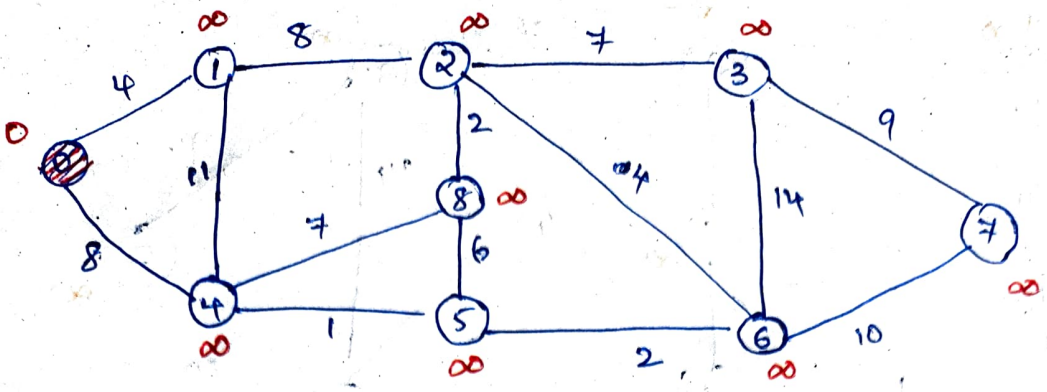
1) Set all vertices distances = ∞ except for source vertex, set it as 0.
2) From source vertex s, find all connected vertex from s.
3) Find the distance from selected/visited vertex by calculating:

   If $(d(u) + c(u,v) < d(v))$ is True, then only update $d(v)$ with $(d(u) + c(u,v))$, otherwise not

   → where u and v are the starting vertex and ending vertex

   → $c(u,v)$ = edge weight or cost between vertex u and v.

4) Repeat step 3 by selecting a vertex that has min distance and ~~calculate~~ update distance by formula.
5) Repeat until all nodes are visited.



We want to find shortest path from source vertex, say 0, to all other vertices in graph G.

Initially put distances of all vertices to ∞ and source to 0 and make source as visited

Then update distance when

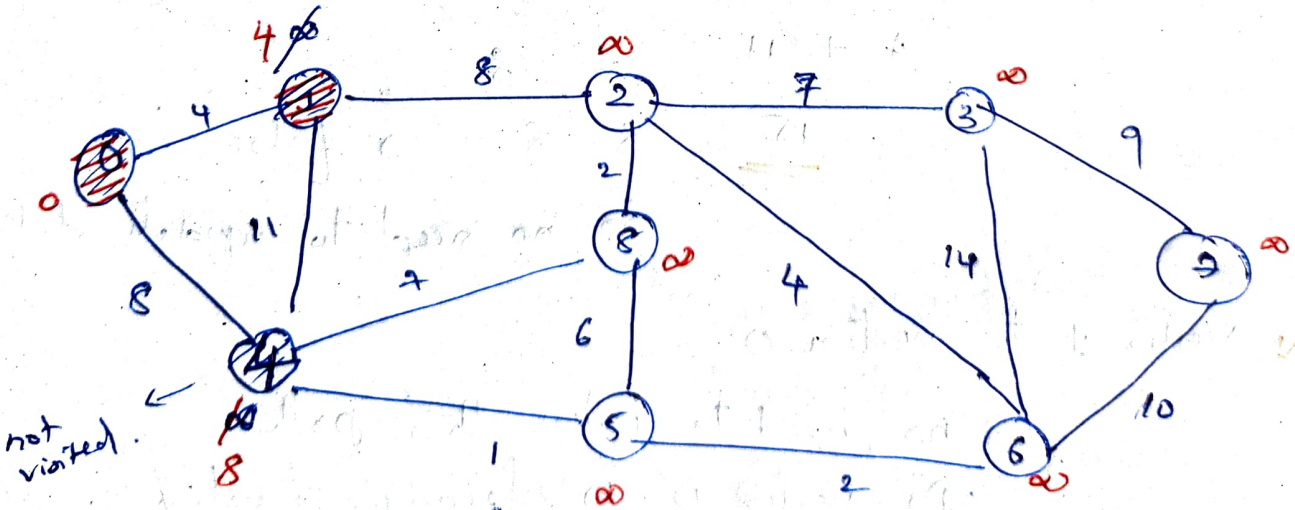
$$d(u) + c(u,v) < d(v)$$

So first find distance from vertex 0, so here

vertex 0 to 1 → vertex 0 = u, vertex 1 = v. } $(d(u) + c(u,v)) < d(v)$

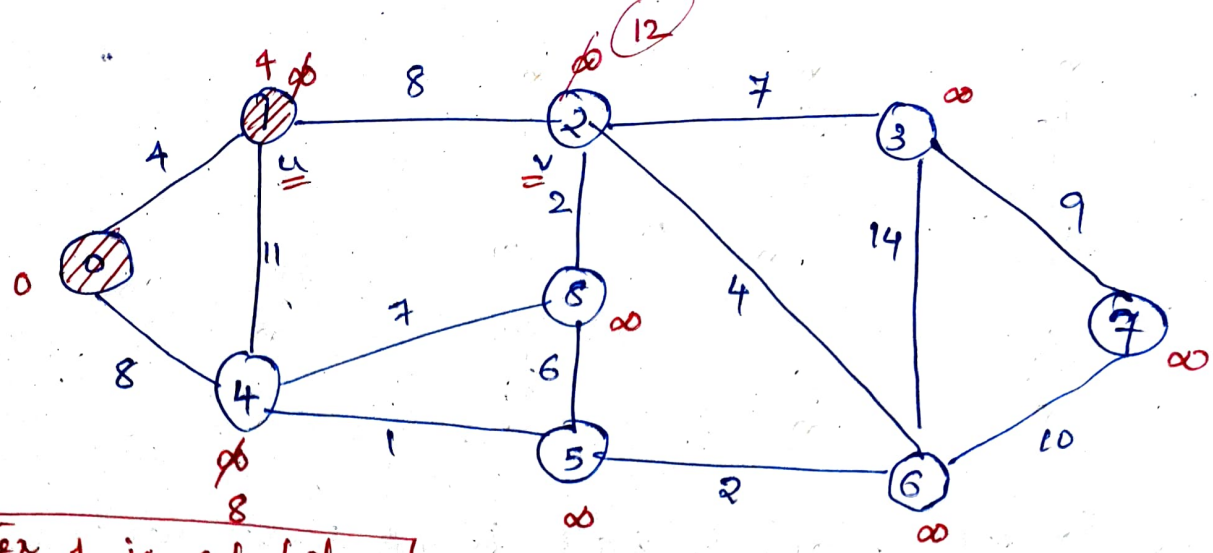
0 to 4 → $0 + 4 < \infty$

$4 < \infty$



Here min of 4 and 8 is 4. and ∞ of other vertex

So make 4 visited



Vertex 1 is selected

* Vertex 1 to vertex 2

$d(u) + c(u,v) < d(v)$

$4 + 8 < \infty$

$12 < \infty \longrightarrow$ true

So make dist (vertex 2) as 12 from ∞

* Vertex 1 to vertex 4

(u to v)

$d(u) + c(u,v) < d(v)$

$4 + 11 < 8$

$15 < 8 \longrightarrow$ false.

So no need to update dist 8 of vertex

* Vertex 1 to vertex 0

→ no need to take this path

As vertex 0 is already selected.

Next check min distance vertex

vertex 0 & 1 already visited / selected.

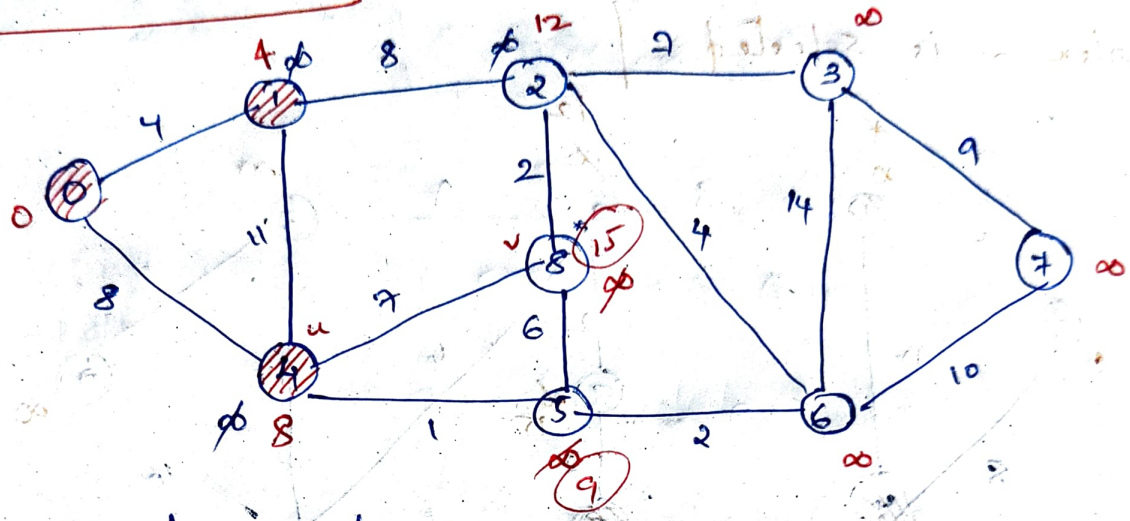
So no need to consider them again.

From 8, 12 and ∞'s

Min = 8.

So select vertex 4

Vertex 4 is selected



Find paths from vertex 4.

✓ vertex 4 to vertex 8

✓ vertex 4 to " 5

vertex 4 to " 1 (already visited vertex 1, so no need to consider vertex 1)

Vertex 4 to vertex 8

$$d(u) + c(u,v) < d(v)$$

$$8 + 7 < \infty$$

$$15 < \infty \longrightarrow \text{true}$$

* So update d(v) as 15

Vertex 4 to vertex 5

$$d(u) + c(u,v) < d(v)$$

$$8 + 1 < \infty$$

$$9 < \infty \longrightarrow \text{true}$$

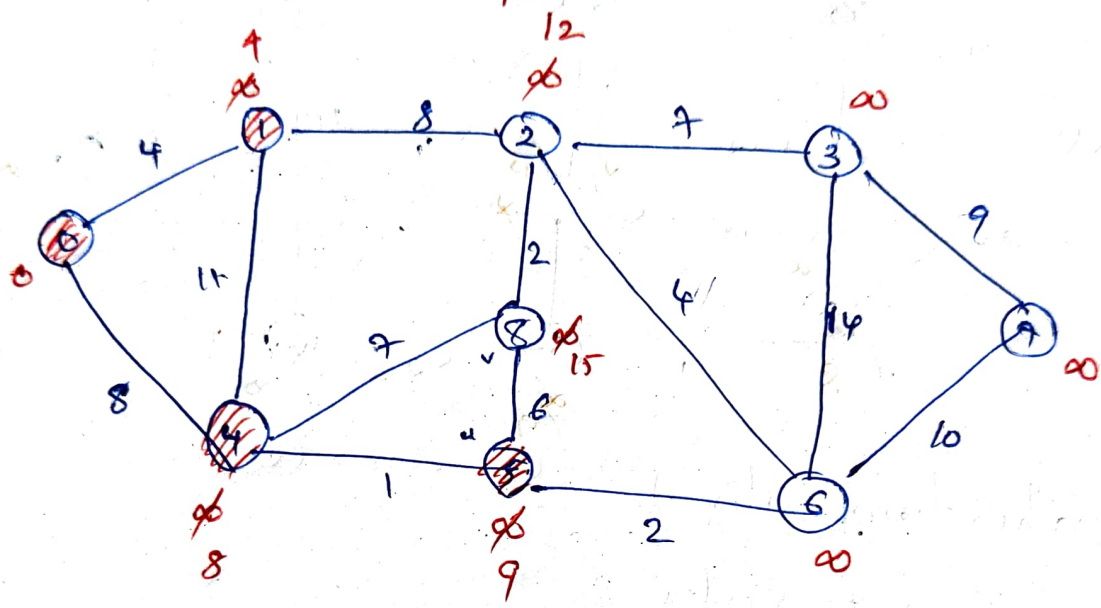
So update d(v) as 9

So here min distance vertex from

12, 15, 9, ∞ ; are → 9.

So choose vertex 5.

Vertex 5 is selected



Consider paths from vertex 5

✓ vertex 5 to vertex 8
✓ vertex 5 to vertex 6
" " " 4 (no need to select vertex 4 since it is visited)

vertex 5 to vertex 8

$$d(u) + c(u,v) < d(v)$$

$$9 + 6 \quad < \quad 15$$

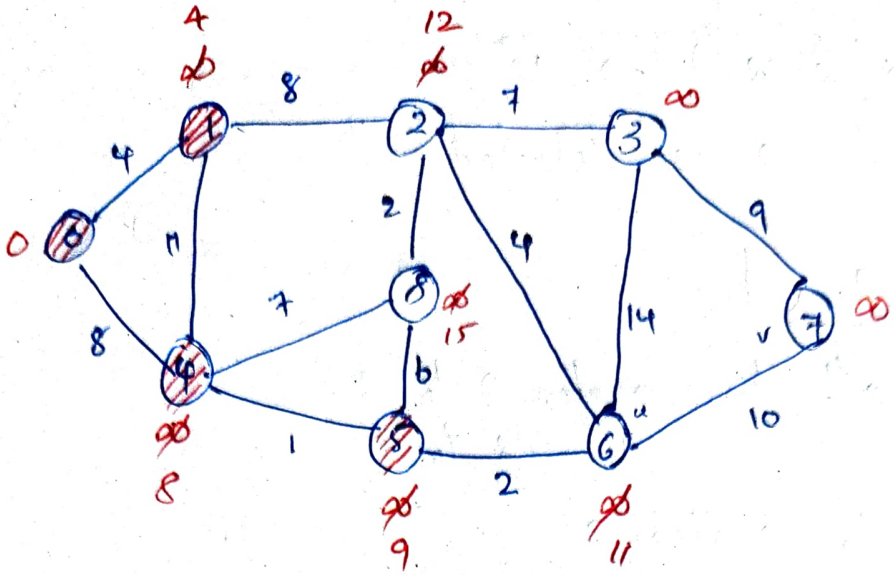
$$15 \quad < \quad 15 \longrightarrow \text{false} \rightarrow \text{no updation for } d(v).$$

vertex 5 to vertex 6

$$d(u) + c(u,v) < d(v)$$

$$9 + 2 < \infty$$

$$11 < \infty \longrightarrow \text{True} \rightarrow \text{make } d(v) \text{ as } 11$$



Here minimum distance vertex is :

$$\min(12, 15, 11, \infty, \infty) \Rightarrow \underline{11}$$

So select vertex 6.

vertex 6 is selected

Consider path from vertex 6 to vertex 7

" 6 " " 3

" 6 " " 2

vertex 6 to vertex 7

$d(u) + c(u,v) < d(v) \Rightarrow 11 + 10 < \infty \Rightarrow 21 < \infty$

update $d(v) = 21$

vertex 6 to 3

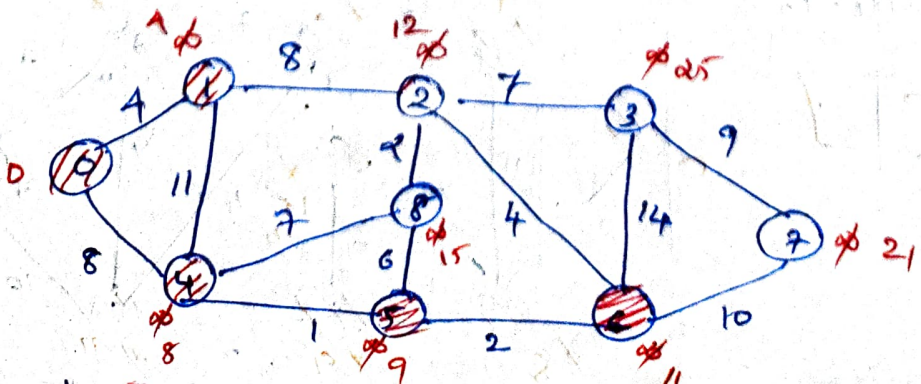
$d(u) + c(u,v) < d(v) \Rightarrow 11 + 14 < \infty \Rightarrow 25 < \infty$

So update $d(v)$ as 25

vertex 6 to 2

$d(u) + c(u,v) < d(v) \Rightarrow 11 + 4 < 12 \Rightarrow 15 < 12 \rightarrow$ false.

So no need to update



min (12, 15, 21, 25) => 12.

So vertex 2 is selected

Select vertex 2

Consider path from vertex 2 to 3
& 2 to 8

→ vertex 2 to vertex 3

$d(u) + c(u,v) < d(v) \Rightarrow 12 + 7 < 25$

$19 < 25 \rightarrow$ true

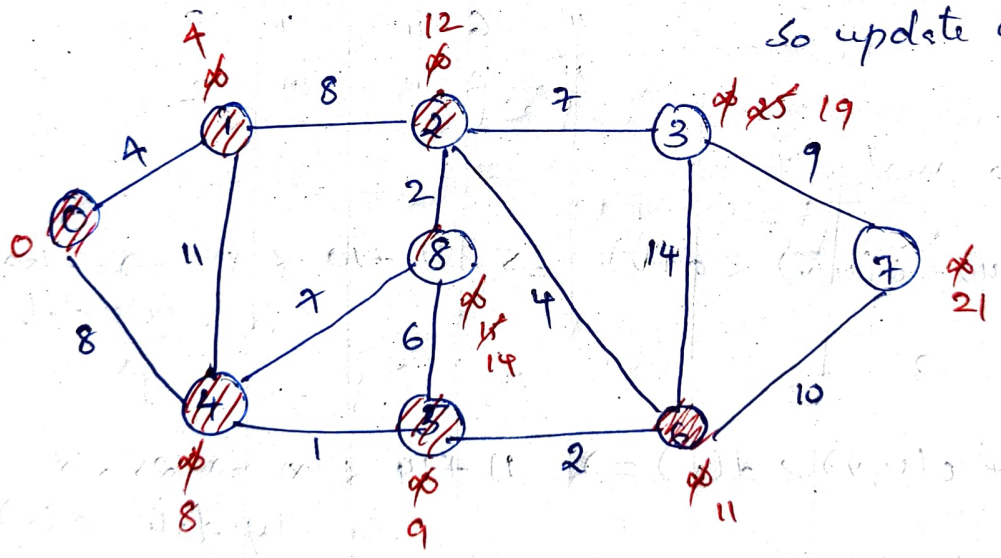
So update d(v) as 19.

→ vertex 2 to vertex 8

$d(u) + c(u,v) < d(v) \Rightarrow 12 + 2 < 15$

$14 < 15 \rightarrow$ true

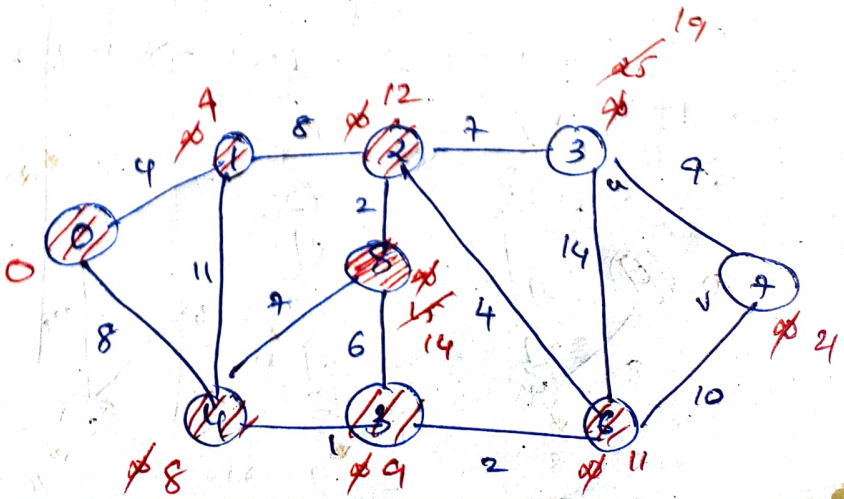
So update d(v) as 14.



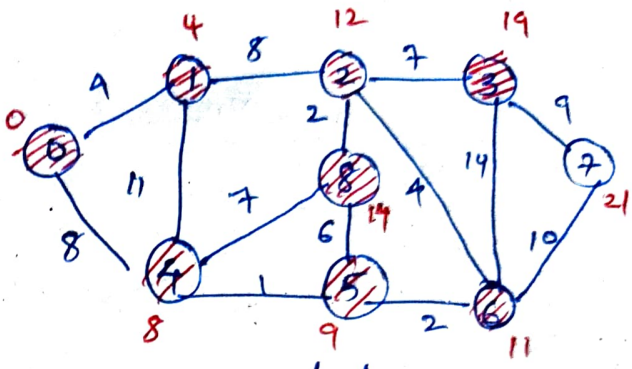
Min(19, 14, 21) => 14.

So, vertex 8 is selected

Select vertex 8



Consider path from vertex 3 to its connected vertices
Since all connected vertices are already visited,
we need to find next min dist. vertex
ie vertex 3 with dist as 19.



Select vertex 3

Consider vertex 3 to 7

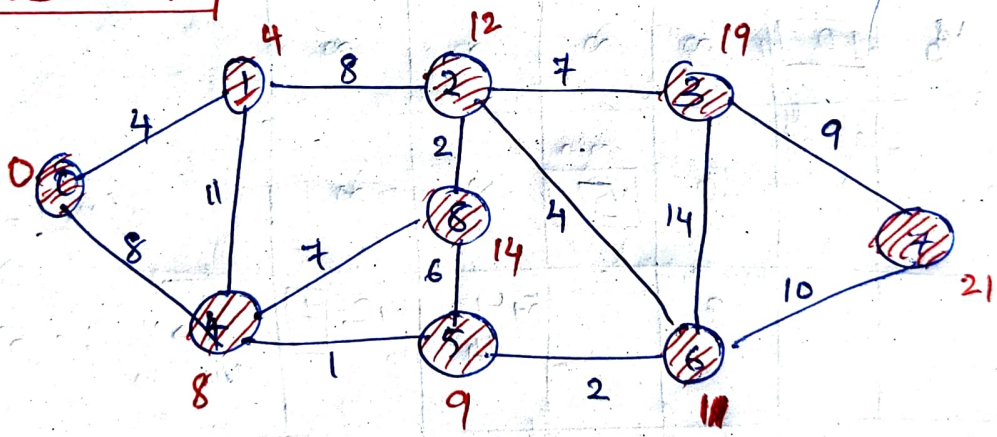
$d(u) + c(u,v) < d(v)$

$19 + 4 < 21 \Rightarrow 23 < 21 \Rightarrow$ false.

no need to update $d(v)$

Finally ∃ vertex 7 only

Select vertex 7



So all nodes are here visited.

Then; from vertex 0 to vertex 1 shortest dist = 4

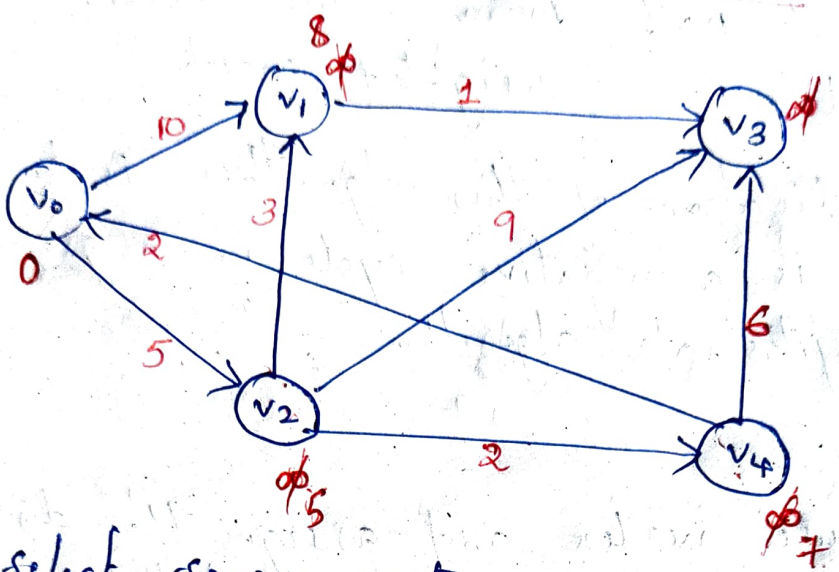
" vertex 0 to " 4 " = 8

vertex 0 to 3 " = 19

⋮

and so on.

DIJIKSTRA'S ALGORITHM IN DIRECTED GRAPH



$d(u) + c(u,v) < d(v)$

Let's select source vertex $\Rightarrow v_0$.

Dijkstra's algorithm may or may not work when there is negative edge weight

	v_0	v_1	v_2	v_3	v_4	
Select v_0	(v_0 to v_0) 0	∞	∞	∞	∞	$d(v_0) = 0$
Selected min is v_2	– (already visited)	(v_0 to v_1) 10	(v_0 to v_2) 5 (min)	∞	∞	$d(v_2) = 5$
Select v_4	–	(v_2 to v_1) 8	–	(v_2 to v_3) 14	(v_2 to v_4) 7	
Select v_1	–	8	–	v_4 to v_3 13	–	
Select v_3	–	–	–	9	–	

$d(v_1) \Rightarrow d(u) + c(u,v) < d(v)$

v_2 to $v_1 \Rightarrow d(v_2) + c(v_2, v_1) < d(v_1)$

$5 + 3 < 10$

$8 < 10$

So update $d(v_1) = 8$

v_2 to $v_3 \Rightarrow 5 + 9 < \infty$

$14 < \infty$

v_2 to $v_4 \Rightarrow 5 + 2 < \infty$

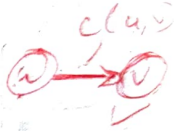
$7 < \infty$

② BELLMAN-FORD ALGORITHM

* To find shortest path from source vertex to all other vertices in a weighted graph
* A main application of this algorithm is to check if there is a negative cycle in a graph.
* Slower than dijikstra's algorithm

→ Steps :-

1) Select a source vertex and assign the distance of source vertex is 0 and the distance of all other vertices as infinity (∞)



2) calculate shortest distance from u to v by checking

if (d(u) + c(u,v) < d(v))

then update d(v) = d(u) + c(u,v)

3) (Do on relaxing with all edges (n-1) times where n = no. of vertices.)

---

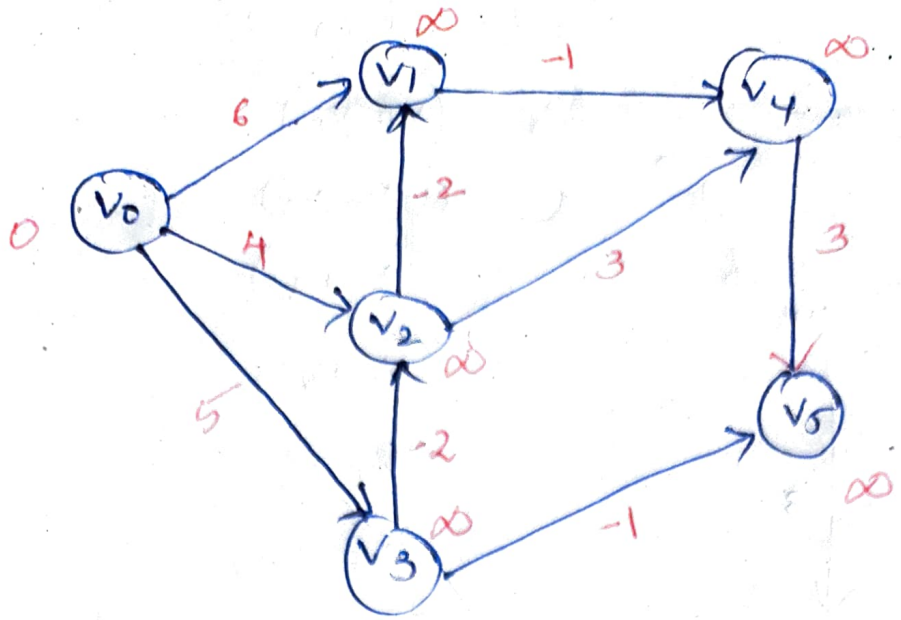
* Drawback

Can't apply if the graph contains negative cycle.

* ALGORITHM

BELLMAN-FORD (G, c, s)

```
1. Initialize-Single-Source(G, s)
2. for i = 1 to |G.V| - 1
3.     for each edge (u,v) ∈ G.E
4.         RELAX (u, v, c)
5. for each edge (u,v) ∈ G.E
6.     if d(u) + c(u,v) > d(v)
7.         return FALSE
8. return TRUE.
```



6-1=5

→ Write all the edges in any order.

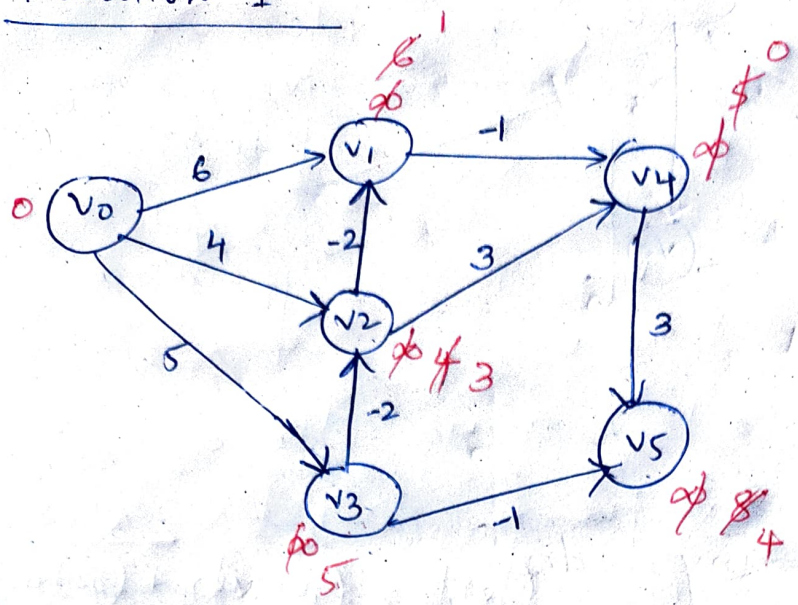
(v_0, v_1), (v_0, v_2), (v_0, v_3), (v_1, v_4), (v_2, v_1), (v_2, v_4), (v_4, v_5), (v_3, v_2), (v_3, v_5)

→ Let v_0 = starting vertex.

→ Let's start our iteration

→ Since there are 6 vertices, there will be (6-1) = 5 iterations.

Iteration 1



The edges are

(v_0, v_1)
(v_0, v_2)
(v_0, v_3)
(v_1, v_4)
(v_4, v_5)
~~(v_1, v_3)~~
(v_3, v_5)
(v_3, v_2)
(v_2, v_1)
(v_2, v_4)

v_0, v_1

v_0 to v_1 $\rightarrow$ $0+6 < \infty$

$d(v_1) = 6$

(v_0, v_2)

$d(v_0) + c(v_0, v_2) < d(v_2)$

$0 + 4 < \infty$

$4 < \infty$

$d(v_2) = 4$

(v_0, v_3)

$d(v_0) + c(v_0, v_3) < d(v_3)$

$0 + 5 < \infty$

$5 < \infty$

ie, $d(v_3) \Rightarrow 5$

(v_1, v_4)

$d(v_1) + c(v_1, v_4) < d(v_4)$

$6 + -1 < \infty$

$5 < \infty$

$\Rightarrow d(v_4) = 5$ //

(v_4, v_5)

$5 + 3 < \infty \Rightarrow 8 < \infty$

ie, $d(v_5) = 8$

(v_3, v_5)

$5 - 1 < 8 \Rightarrow 4 < 8$

~~no updation~~ $d(v_5) = 4$

(v_3, v_2)

$5 - 2 < 4$

$3 < 4$, $d(v_2) = 3$

(v_2, v_1)

$3 - 2 < 6 \rightarrow 1 < 6$

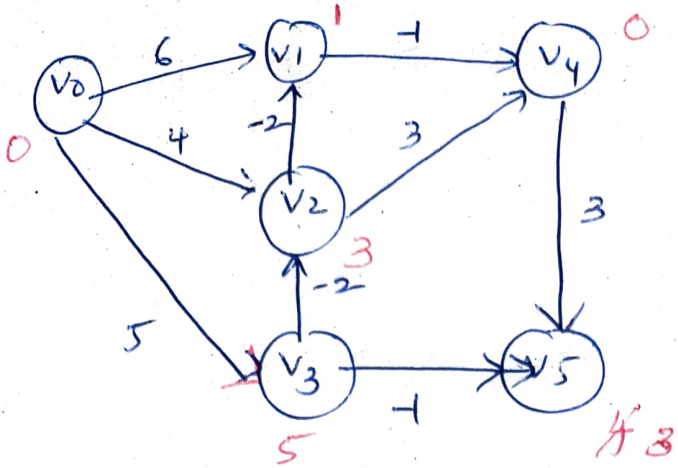
$d(v_1) = 1$

(v_2, v_4)

$3 - 3 < 5 \rightarrow 0 < 5$

$d(v_4) = 0$ //

Iteration 2



(v_0, v_1)

$0 + 6 < 1$

$6 < 1 \rightarrow$ false.

no updation to $d(v_1)$

(v_0, v_2)

$0 + 4 < 3 \rightarrow$ false.

no updation to $d(v_2)$

(v_0, v_3)

$0 + 5 < 5 \rightarrow$ false.

(v_1, v_4)

$1 - 1 < 0 \rightarrow$ false.

(v_4, v_5)

$0 + 3 < 4$

$d(v_5) = 3$

(v_3, v_5)

$5 + 1 < 3$

$1 < 3 \rightarrow$ false.

(v_3, v_2)

$5 + -2 < 3 \rightarrow$ false.

(v_2, v_1)

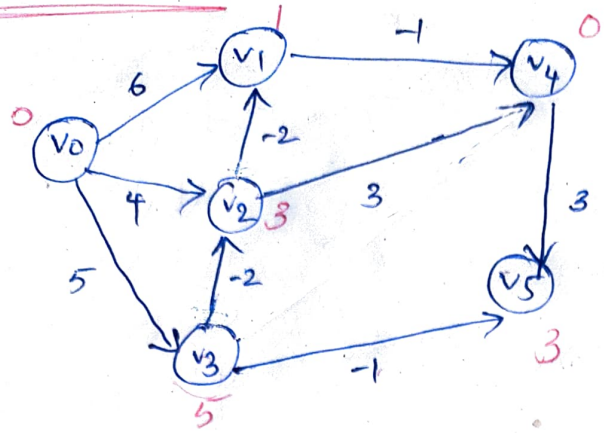
$3 + -2 < 1 \rightarrow$ false

(v_2, v_4)

$3 + 3 < 0$

$6 < 0 \rightarrow$ false.

Iteration 3



(v0, v1)

$d(v_0) + c(v_0, v_1) < d(v_1)$

$0 + 6 < 1$

→ false

(v0, v2)

$0 + 4 < 3$ → false.

(v0, v3)

$0 + 5 < 5$ → false.

(v1, v4)

$1 + -1 < 0$ → false.

(v4, v5)

$0 + 3 < 3$ → false.

(v3, v5)

$5 + -1 < 3$ → false.

(v3, v2)

$5 + -2 < 3$

→ false.

(v2, v1)

$3 + -2 < 1$

→ false

(v2, v4)

$3 + 3 < 0$

→ false

We can see that there is no updation to the distance of any vertex v_i.

So we can stop the iteration. (No need to go till iteration 5).

So the distance:

v0 = 0	v5 = 3
v1 = 1	
v2 = 3	
v3 = 5	
v4 = 0	

⇒ **Time Complexity**

→ $O(E(|V|-1))$

→ $n-1$ iterations

$= O(E \cdot V)$

⇒ In case of complete graph, time complexity

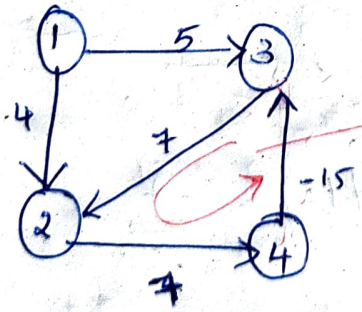
In complete graph, no. of edges $(E) = \frac{n(n-1)}{2}$

(where n => no. of vertices.)

So, $O(E \cdot V)$ becomes => $\left[\frac{n(n-1)}{2} \cdot (n-1)\right]$

(n : no. of vertices.)

* **Negative cycle - Eg:-**



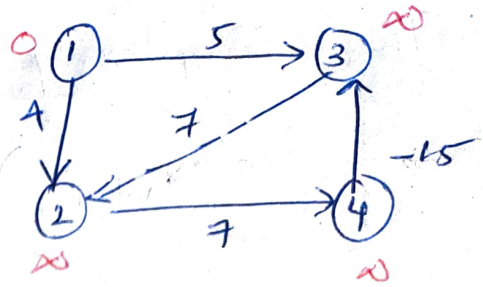
cycle.

7+7-15 = -1

negative cycle here.

We can't apply Bellman-ford algorithm in case if there exist negative cycle in a graph.

Let's check :-



Edges :-

(1,3)
(1,2)
(2,4)
(4,3)
(3,2)

let 1 be the source vertex.

Since there are 4 vertices, there will be

(4-1) = 3 iterations

⇒ Iteration 1

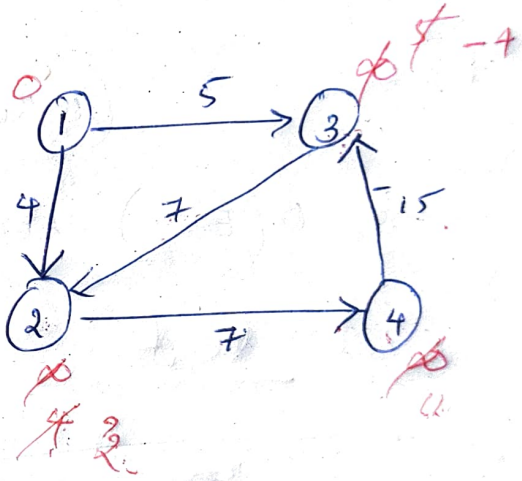
(1,3)

$d(1) + c(1,3) < d(3)$

$0 + 5 < \infty$

$5 < \infty$

$d(3) = 5$



(1,2)

$0 + 4 < \infty$

$d(2) = 4$

(2,4)

$4 + 7 < \infty$

$11 < \infty$; $d(4) = 11$

(4,3)

$11 + -15 < 5$

$-4 < 5$

$d(3) = -4$

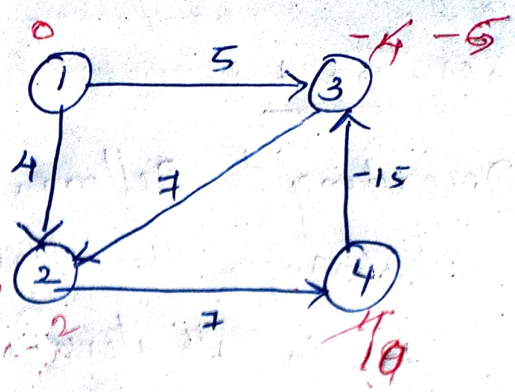
(3,2)

$-4 + 7 < 4$

$3 < 4$

$d(2) = 3$

Iteration 2



(1,3)

$0+5 < -4$

↳ false, not update d(3)

(1,2)

$0+4 < 3$ → false.

not update d(2)

(2,4)

$3+7 < 11$

$10 < 11$ → true.

So d(4) = 10

(4,3)

$10 + -15 < -4$

$-5 < -4$ → true.

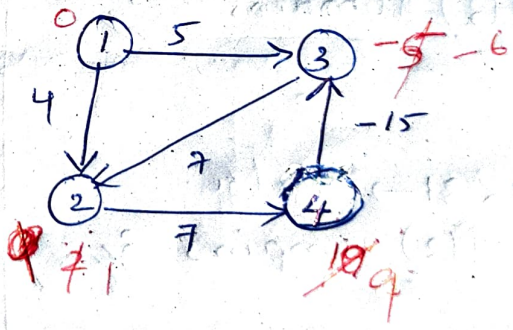
Update d(3) = -5

(3,2)

$-5+7 < 3$

$2 < 3$ → true, d[2] = 2 ~~false~~

Iteration 3



(1,3)

$0+5 < -5$

→ false

(1,2)

$0+4 < 2$

→ false

(2,4)

$7+7 < 10$ → true

$9 < 10$ → true

d(4) = 9

(4,3)

$9 + -15 < -5$

$-6 < -5$ → true

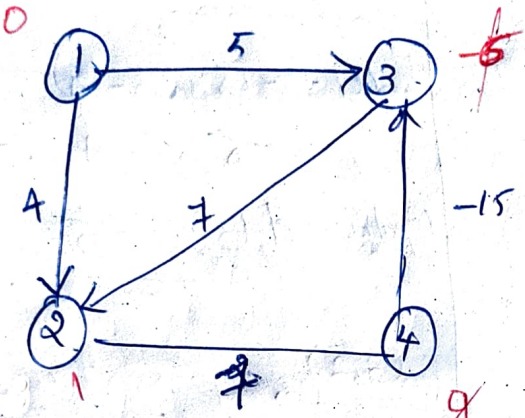
update d(3) = -6

(3,2)

$-6+7 < 3$ → $1 < 3$ → true, d(2) = 1

Iteration 4

According to Bellman ford algorithm there should not change the distance after n-1 iteration
ie, in 4th iteration, if distance doesn't change, we can say that there is no -ve cycle



(1,3)
(1,2)
(2,4)
(4,3)
(3,2)

(1,3)

0+5 < -5 → false
d(3) remains -5

(1,2)

0+4 < 1 → false
d(2) remains 3

(2,4)

1+7 < 10 → 8 < 10 → true
d(4) becomes 8

(4,3)

d8 + -15 < -5 → false
-7 < -5
d(3) ~~remains~~ -7

(3,2)

-5 + 7 < 3
2 < 3 → false

So here distance changed
d(3) = -7, d(4) = 8
So we can see that in (n-1)+1 th iteration, distance changed because there is a negative cycle in graph.

Drawback of Bellmanford

So we can't apply Bellman ford algorithm in negative ~~graph~~ cycled graph.

TOPOLOGICAL SORTING

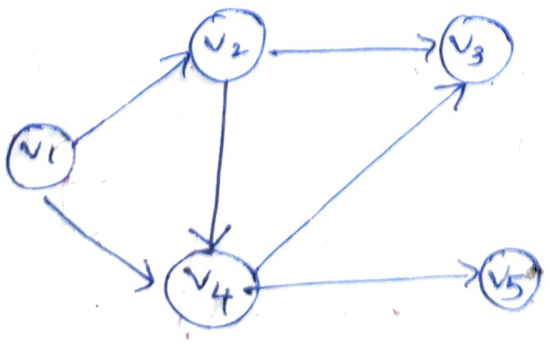
* It's a linear ordering of its vertices such that for every directed edge uv for vertex u to vertex v, u comes before v in the ordering.
* Graph should be Directed Acyclic Graph (DAG)
* Every DAG will have atleast one topological ordering
* Topological sorting is not possible if the graph is not DAG

TOPOLOGICAL_SORT(V, E)

1. Call DFS(V, E) to compute finishing time f[v] (indegree) for all v in V
2. As each vertex is finished, insert it onto the front of linked list
3. Return the linked list of vertices.

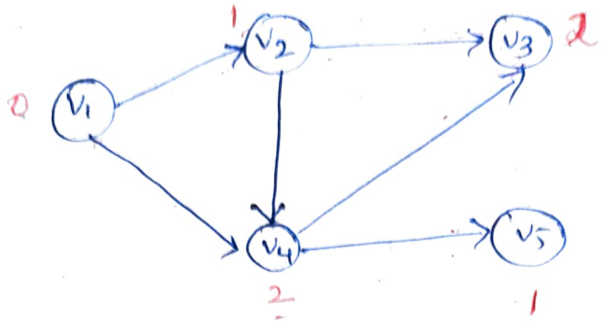
* We can perform a topological sort in O(V+E), since DFS takes O(V+E) time and it takes O(1) time to insert each of vertices v onto the front of linked list.

Eg.-



Indegree (v_1) = 0
Indegree (v_2) = 1
Indegree (v_3) = 2
Indegree (v_4) = 2
Indegree (v_5) = 1

Step 1:- CALCULATE INDEGREE OF ALL VERTICES IN THE GIVEN GRAPH.

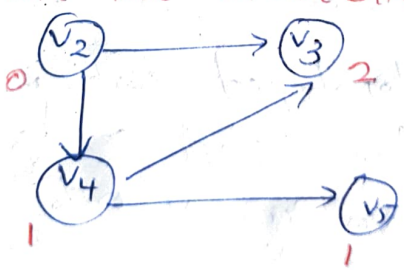


Step 2:- CHOOSE THE VERTEX HAVING INDEGREE 0 AND REMOVE THAT VERTEX FROM GRAPH AND ADD TO THE FRONT OF LINKED LIST.
(Along with vertex, we need to remove the outgoing edges from v)

Here vertex v_1 having indegree 0.
So remove that vertex and outgoing edges.

Then, Add v_1 to list ⇒

Step 3:- UPDATE THE IN-DEGREE OF NEW GRAPH AFTER REMOVING v_1

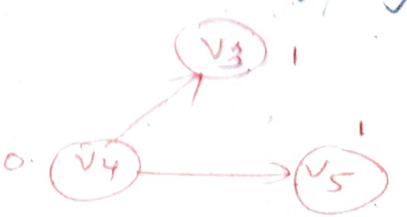


Repeat step 2 until there is no more vertex in graph i

INDEGREE (v_2) = 0

So remove v_2 and its outgoing edges.

Then the graph becomes



Indegree (v_4) = 0

Add v_4 to linked list | v_1 | v_2 | v_4 |

Then the graph becomes

(v_3)°

(v_5)°

We can see that indegree (v_3) = 0
& indegree (v_5) = 0

we can either choose v_3 or v_5 first.

Case 1 :- choose v_3

| v_1 | v_2 | v_4 | v_3 |

~~Case 2 : choose~~ Remaining v_5 there (v_5)°

| v_1 | v_2 | v_4 | v_3 | v_5 |

Case 2 :- choose v_5 first

| v_1 | v_2 | v_4 | v_5 |

Remaining vertex is (v_3)

| v_1 | v_2 | v_4 | v_5 | v_3 |

So this graph have 2 topological sorting

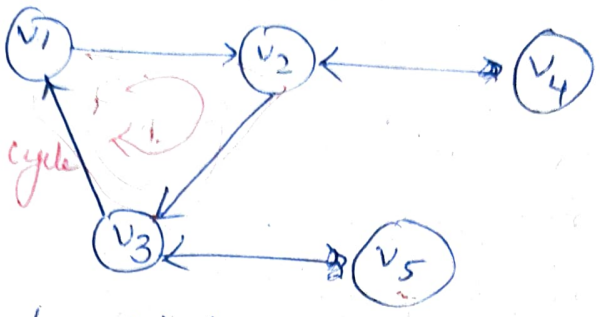
1) v_1 v_2 v_4 v_3 v_5

2) v_1 v_2 v_4 v_5 v_3

{ So a DAG will have 1 or more topological sort

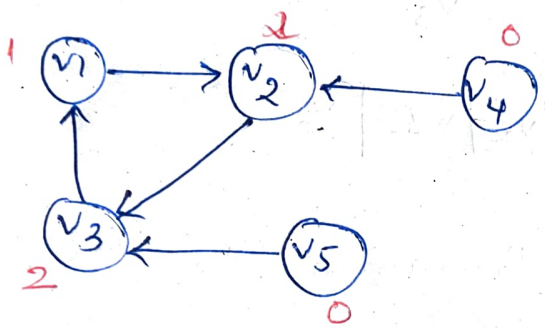
⇒ We cannot find topological sort of a graph which is not DAG.

Eg:-



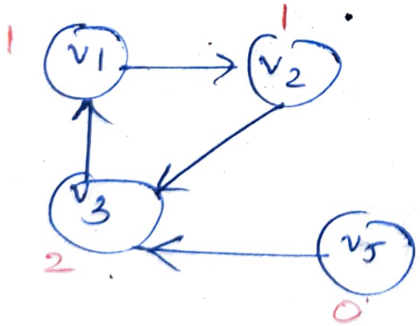
Let's try to find out topological sort of this graph!

Finding indegree



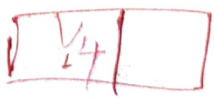
Indegree 0 having vertices ⇒ v_4, v_5

case 1: Remove v_4. (∵ indegree(v_4) = 0)

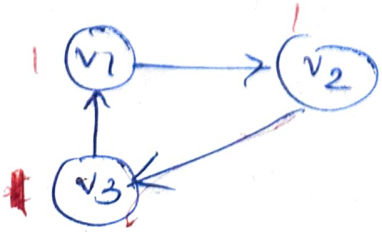


~~add to Linked List~~
~~Print v4~~

add v_4 to linked list



Remove v_5

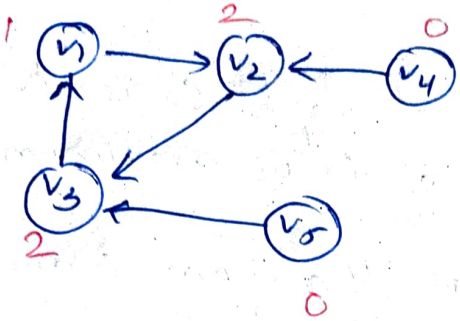


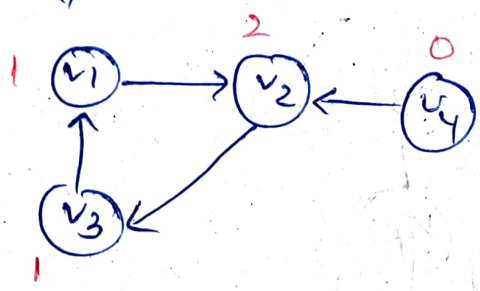
~~Print~~ v_5 ← add to ll.

v_4, v_5

we can see that there is no vertex remaining there having indegree 0.

Case 2 :- Remove v_5 from graph G

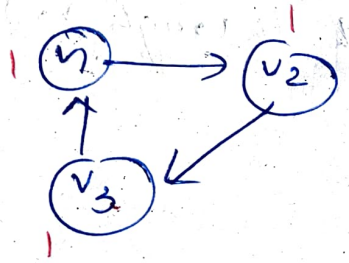




Add ~~Print~~ v_5 to LL.

v_5

Remove v_4



Add v_4 to linked list

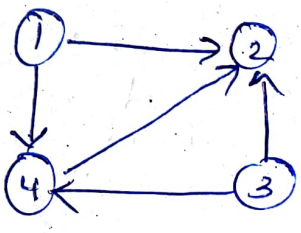
↳ v_5, v_4

we can't remove any ~~ed~~ vertex (vertex with indegree = 0) from this graph.

∴ So we can say that we can't find topological sort on a directed graph having cycle.

Qs1n

Find topological sort of given graph



[Ans :- 1) 1 3 4 2
3) 3 1 4 2]

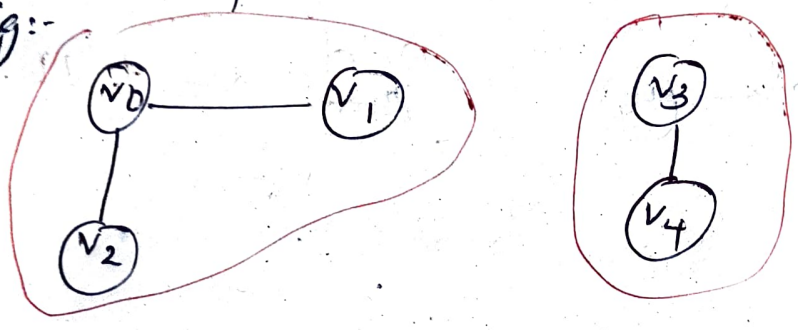
* STRONGLY CONNECTED COMPONENTS (SCC)

→ CONNECTED COMPONENTS

Connected component of a graph G is a connected subgraph of G of maximum size.

A graph may have more than one connected component.

Eg:-



There are 2 connected components in above graph { v_1, v_0, v_2 (path through v_0) ; v_3 v_4 }

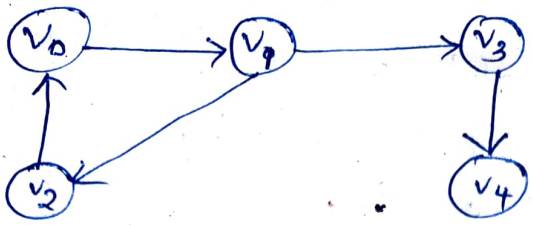
Nodes are reachable.

→ STRONGLY CONNECTED COMPONENTS (SCC)

→ Strongly connectivity applies only to directed graph

→ A scc of a directed graph is a complement such that all vertices in that component is reachable from every other vertex in that component.

Eg..



Need to reachable from other vertices.

$v_0 \Rightarrow v_0 - v_1 - v_2 \rightarrow v_0$.
$v_1 \Rightarrow v_1 \rightarrow v_2 \rightarrow v_0 - v_1$
$v_2 \Rightarrow v_2 \rightarrow v_0 \rightarrow v_1 \rightarrow v_2$.

But v_3 is not reachable. we can't go back to v_1 or v_0 or v_2.

$\Rightarrow$ So 1st SCC = $\{v_0, v_1, v_2\}$.

No states are reachable to v_3 and also to v_4.

$\Rightarrow \{v_3\}$
$\Rightarrow \{v_4\}$.

So there are 3 SCC = $\{\{v_0, v_1, v_2\}, \{v_3\}, \{v_4\}\}$.

SCC Algorithm → kosaraju's Algorithm

→ Algorithm consist of 2 pass.

- Pass 1 ⇒ step: 1 to 4
- Pass 2 ⇒ step: 5 to 7

Pass 1

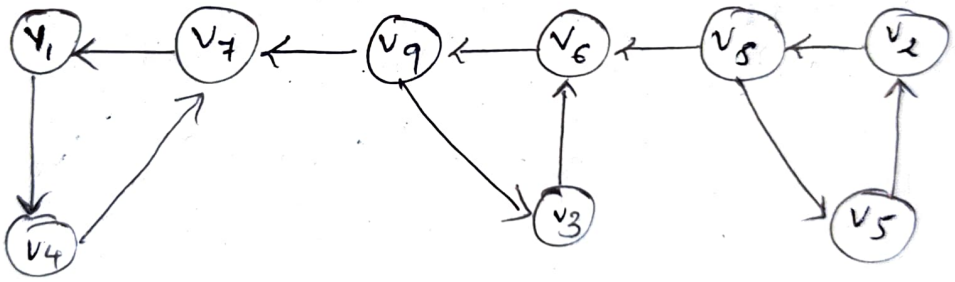
1. set of all vertices of graph G are unvisited
2. Create an empty stack.
3. Do DFS traversal on unvisited vertices & set it as visited
   If a vertex has no unvisited neighbor, Push it to stack.

4. Perform the above step until all vertices are visited.

Pass 2

5. Reverse the graph G
6. Set all nodes are unvisited
7. while stack is not empty:
   1) Pop one vertex v'
   2) if v' is not visited:
      1) set v' as visited
      2) call DFS(v'). It will print strongly connected component of v'

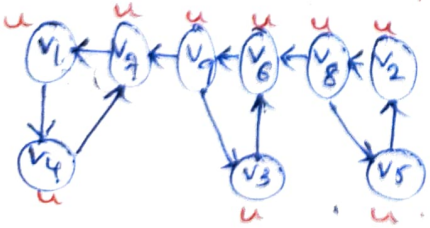
Q. Find scc of the following digraph:

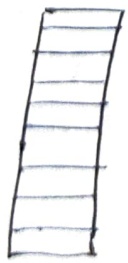


Pass 1

Step 1, 2

*) Make all vertices as unvisited and stack is empty





stack (empty)

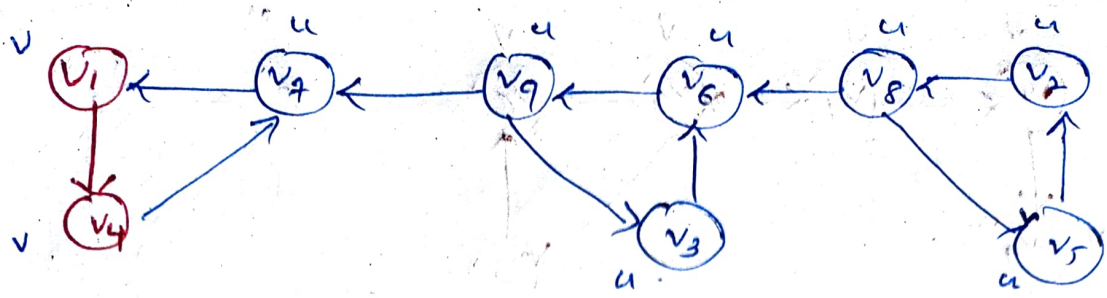
v1
v2
v3
:
v9

} unvisited (4)

Let starting vertex = v_1

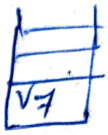
→ Make it visited
→ Then look its unvisited neighbor, here it is v_4. and set it as visited (using DFS)

DFS :- v_1, v_4



unvisited neighbor (v_4) = v_7.

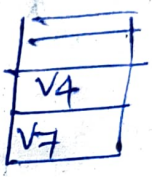
and v_7 has no unvisited neighbor.



So push v_7 into stack.

Then go back. ie v_4.

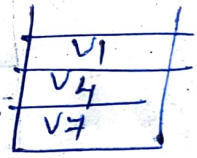
v_4 has no unvisited neighbor, so push v_4 to stack



Go back ie, v_1.

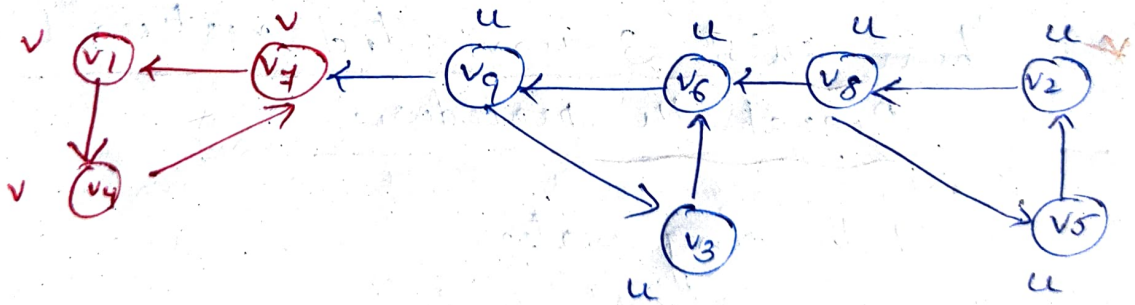
v_1 has no unvisited neighbor.

so push v_1 to stack



These v_7, v_4, v_1 are now visited so, DFS :- v_1, v_4 v_7

⇒



* There are many unvisited nodes.

Choose any vertex (unvisited vertex), say, v_9.

so v_9 visited

unvisited neighbor (v_9) = v_3 → visit v_3.

DFS :- v_1, v_4, v_7, v_9, v_3, v_6

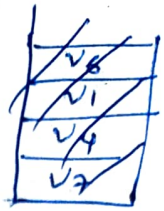
unvisited-neigh (v_3) = v_6.

make v_6 visited.

DFS :- $v_1, v_4, v_7, v_9, v_3, v_6, v_8, v_5, v_2$

1) unvisited neighbor (v_2) = nil → Push v_2 to stack

So go back, v_5



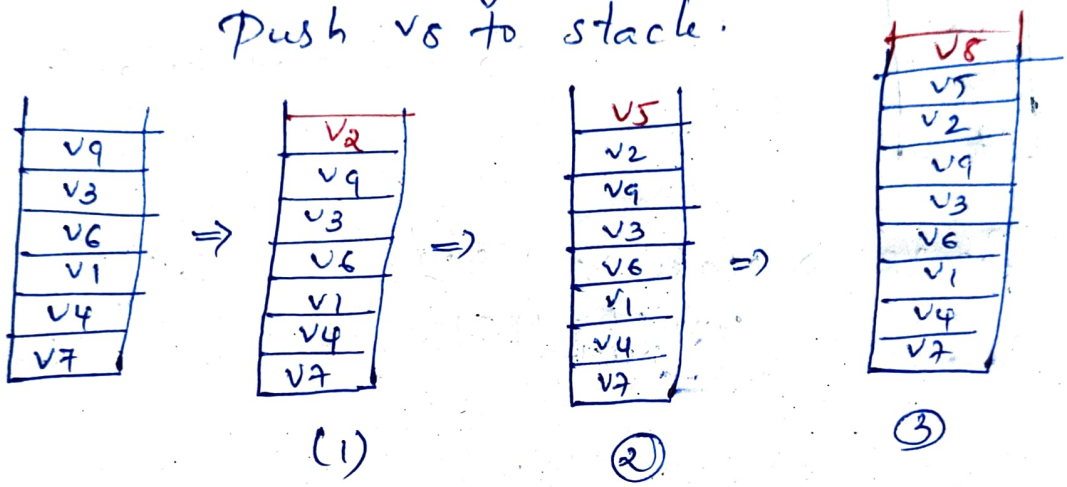
2) unvisited neighbor (v_5) = nil

Push v_5 to stack.

3) Go back, v_8.

unvisited neighbor (v_8) = nil

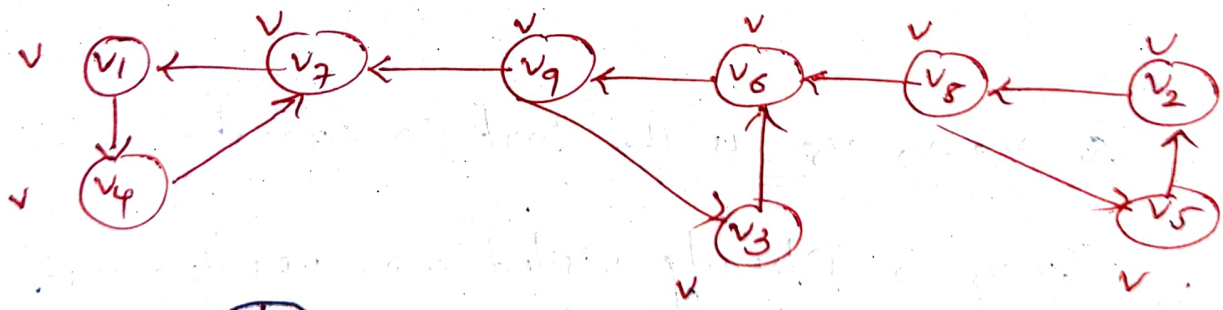
Push v_8 to stack.



↑ (stack construction completed.

Since no unvisited vertices are there,

Pass 1 completed.

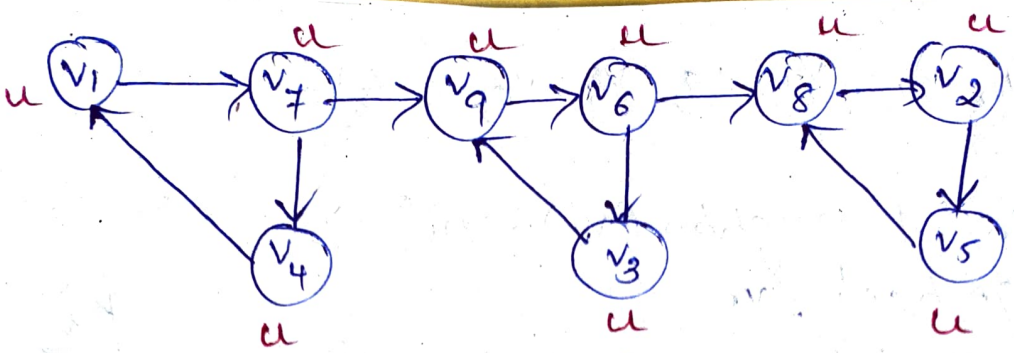


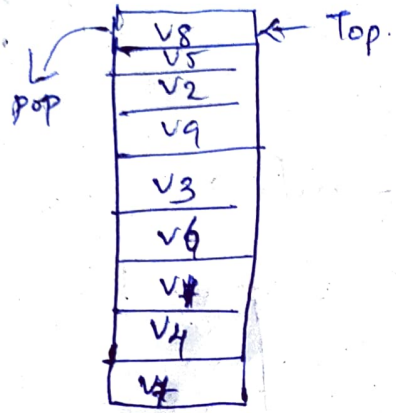
Pass 1 Completed.

Pass 2.

(1) REVERSE THE GRAPH

→ make edges in opposite direction and make all vertices as unvisited.





→ Pop v_8 — visit v_8 ⇒ v_8

unvisited neigh(v_8) = v_2 ⇒ v_2

unvisited neigh(v_2) = v_5 ⇒ v_5

No unvisited neigh(v_5) =

$v_8 - v_2 - v_5$

Pop v_8 → Make v_8 visited

unvisited neigh(v_8) = v_2 → make v_2 visited.

unvisited neig(v_2) = v_5

↓

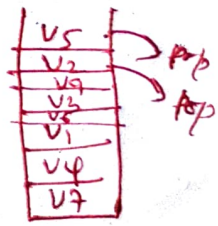
Repeat

unvisited neigh(v_5) = nil

go back.

unvisited neig(v

$v_8 - v_2 - v_5$



So $v_8 - v_2 - v_5$ is the first component.

Pop v_5 → Already visited — no DFS traversal

Pop v_2 → Already visited — "

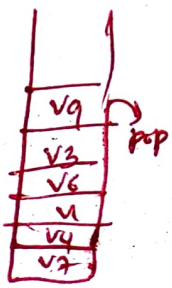
⇒ Next is pop v_9 — v_9 is now unvisited — do DFS

v_9 → make visited

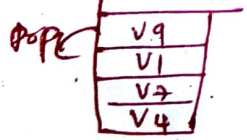
unvisited neigh(v_9) = v_6 → make v_6 visited — do DFS traversal

unvisited neigh(v_6) = v_3 → make v_3 visited — "

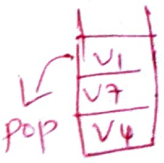
unvisited neigh(v_3) = null → no DFS



So next component ⇒ $v_9 - v_6 - v_3$.



⇒ Next Pop v_1



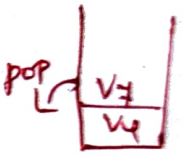
It is unvisited.
Make v_1 visited. ⇒ v_1

Unvisited neigh(v_1) = v_7 ⇒ make v_7 visited & Dfs traversed. } - v_7

Unvisited neigh(v_7) = v_4 ⇒ make v_4 visited & do Dfs traversed } - v_4

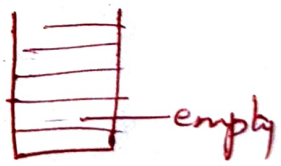
Unvisited neigh(v_4) = null → (Completed thus Dfs)

So new component = $v_7 - v_4 - v_4$



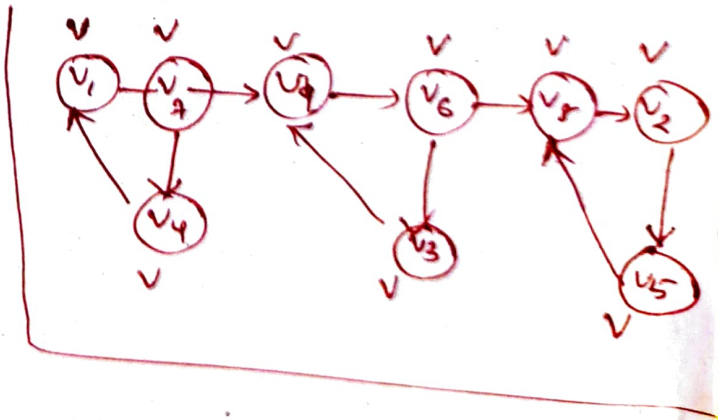
Pop v_4 — already visited → So no Dfs
Pop v_4 — " " → "

Now stack is empty.



⇒ So scc are:-

1) $v_8 - v_2 - v_5$
2) $v_9 - v_6 - v_3$
3) $v_1 - v_7 - v_4$



TIME COMPLEXITY

Time complexity of Dfs (since we're using Dfs) = $O(V+E)$
Reversal of graph = $O(V+E)$ time
Pass 2 take another $O(V+E)$ time.

so total ⇒ $O(V+E)$