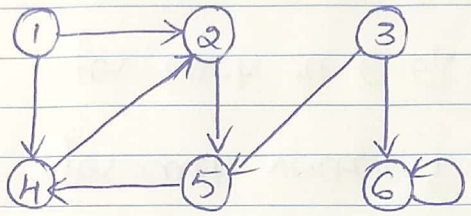
ॐ

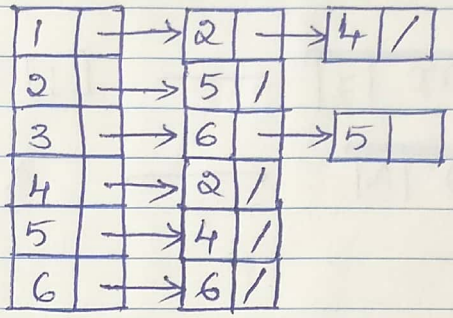
SNEHA VIDHYASHEKAR
ASU ID : 1211274841

ASSIGNMENT-3

1. Adjacency-list representation of a directed graph.



directed graph.



adjacency-list

consider the graph $G(V, E)$

For every vertex V, there is a list in the adjacency list.

∴ There are $|V|$ lists in the adjacency list array, "Adj".

Every vertex that is adjacent to p ⟶ Adj[p].
as shown above.

⟶ Time taken to compute the out-degree of every vertex is, $\Theta(|V|+|E|)$

Pseudo code :-

```
Outdegree (Graph G)
outdegree[v] = {0}   // To store degree of each vertex.
for each vertex i ∈ V   // to traverse all vertices list.
    count = 0 ;
```

Traverses each vertex of the Graph

runs adj[v] times

```
for each u ∈ G.adj[i]
    count += count + 1;
    outdegree[i] = count;
```

outer for loop :-

→ runs |V| times
∵ it traverses every vertex of the graph.

inner for loop

→ runs adj[V] times
↳ (adjacency list)

adj[V] → sum of lengths of each vertex's adjacency list is |E|

∴ |E| times → inner for loop.

∴ $O(|V|+|E|)$

Indegrees of every vertex. $O(|V|+|E|)$

Pseudocode :-

```
indegree[v] = {0};
for each vertex i ∈ V ——→ |V| times
    for each u ∈ G.adj[i] ——→ |E| times.
        indegree[u] = indegree[u] + 1
```

inner loop = |E| times

∵ while traversing a particular node's edges

program simply increment the in-degree of corresponding vertices to which that node headed.

∴ The indegree array will have the in-degree of all the vertices.

∴ Time taken to compute the in-degrees is

$$\boxed{\Theta(|V|+|E|)}$$

2. Algorithm to compute G^T from G

(a) <u>using adjacency list</u> :-

Let, $G(V, E)$

adjacency list adj[u] with length $|V|$

$|V| \rightarrow$ Total no of vertices
$|E| \rightarrow$ Total no of edges.

Adjacency_list_transpose (G_T)

for $u = 1$ to $|V|$ — $|V|$ times (length)

for each vertex v in the list G_T.adj[u] — $|E|$ times

add vertex u to the list G_T^T.adj[v]

* return G_T^T.adj[v]

edges are reversed and added to the list G_T^T.adj[v]

∴ Time taken = $\Theta(|V|+|E|)$.

(b) adjacency matrix :-

Let, $G = (V, E)$.

A → adjacency matrix of G
B → adjacency matrix of G^T (transpose)

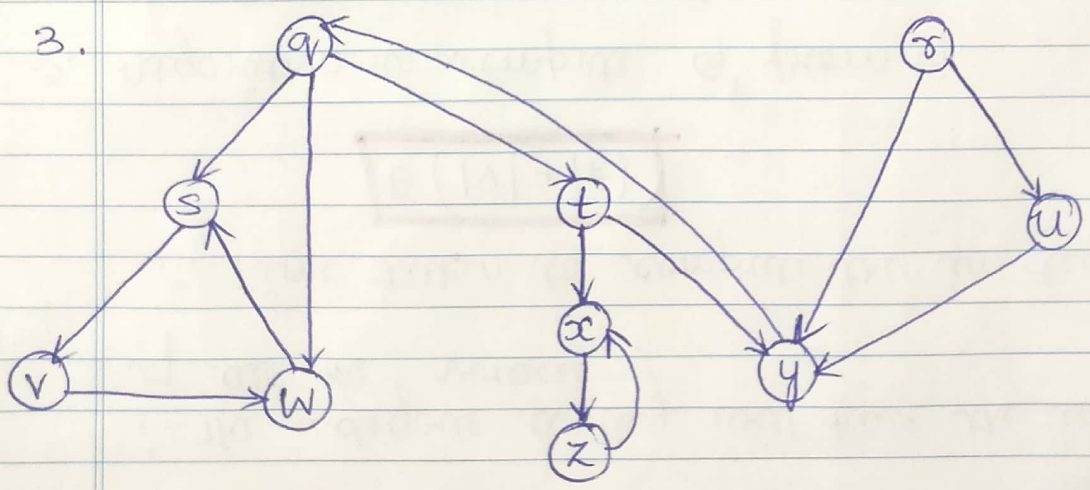
```
Adjacency_matrix_transpose (G, A)
inner loop { for i = 1 to |V|  ⟶ V times
             for j = 1 to |V|  ⟶ V times
                B[j][j] = A[i][j]    // transpose matrix (edges are reversed)
return B
```

→ To scan the matrix (adjacency matrix) the algorithm repeats 2 loops ⟶ $O(V^2)$

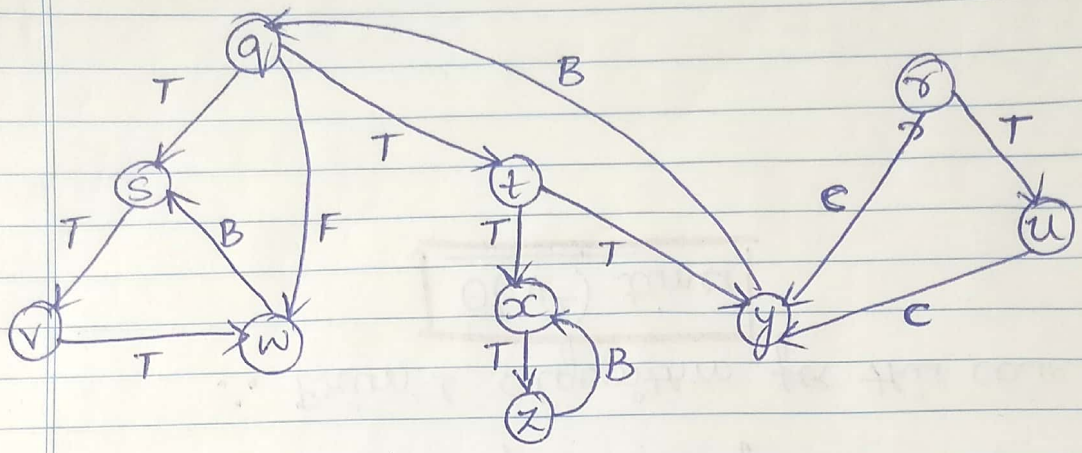
→ // To update new adjacency matrix → $O(1)$

∴ Total time complexity = $O(V^2)$.

3.

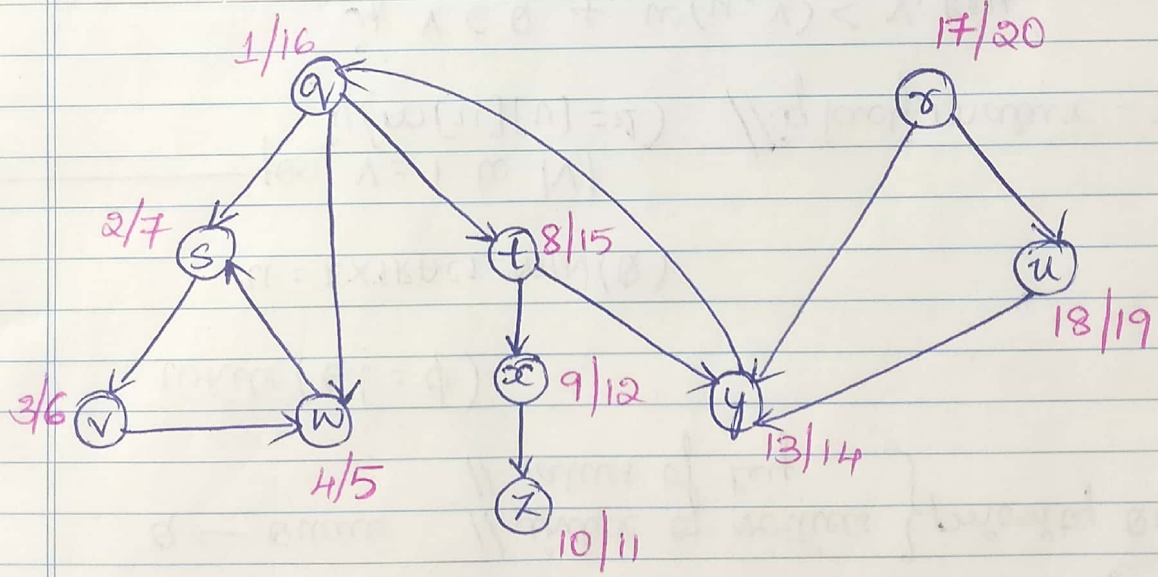


DFS for above graph.



B → Back edges
F → Forward edges
T → True edges
C → Cross edges.

q & r. → are @ the same level.
→ alphabetical order → start at "q".



Above graph shows the discovery and finishing times for each vertex (with the numbers).

Eg:- (q) → 1/16

1st discovered vertex | Finishing time.

4. Prim's algorithm using adjacency matrix $\Rightarrow O(V^2)$

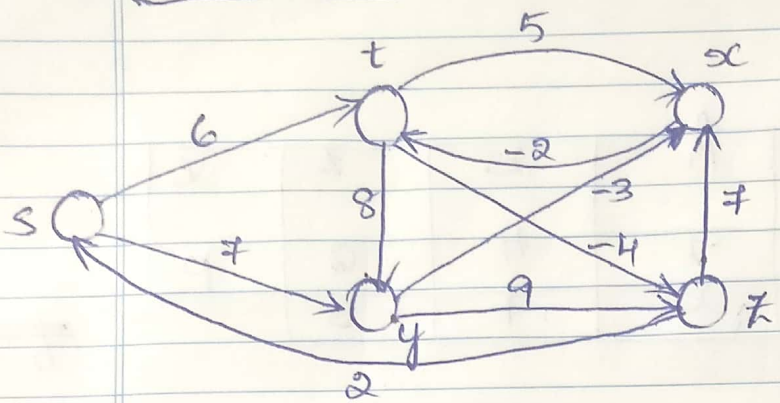
```
Prim(G, w, r)   // G(V, E)   { G - graph
                              { w - weights
                              { r - root
|V| <- for u = 1 to |V|
        u.key = ∞
        u.π = NIL   // π -> parent

r.key = 0   // source - where we start tracing.

Q <- Queue   // index of vertices } priority Queue.
             // values of key.   }

while (Q != ϕ)

    u = EXTRACT_MIN(Q)

|V| <- for v = 1 to |V|
        if (m[u][v] == 1)   // if (adj matrix == 1)

            if v ∈ Q & w(u, v) < v.key
                v.π = u
                v.key = w(u, v)
```

→ It has 2 for loops
→ 2 nested for loops
→ each loop runs from 1 to |V|

∴ Prim's algorithm for this case is

$O(V^2)$ times

5. Bellman-Ford.



Initially setting d & π val:

	s	t	x	y	z
d	∞	∞	∞	∞	0
π	ϕ	ϕ	ϕ	ϕ	ϕ

← ∵ z is source.

← parents = NULL.

Given :- Relaxing order is

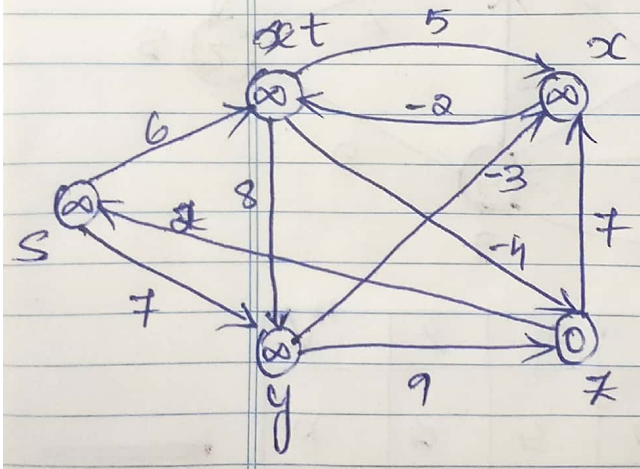
(t, x) (t, y) (t, z) (x, t) (y, x) (y, z) (z, x)

(z, s) (s, t) (s, y)

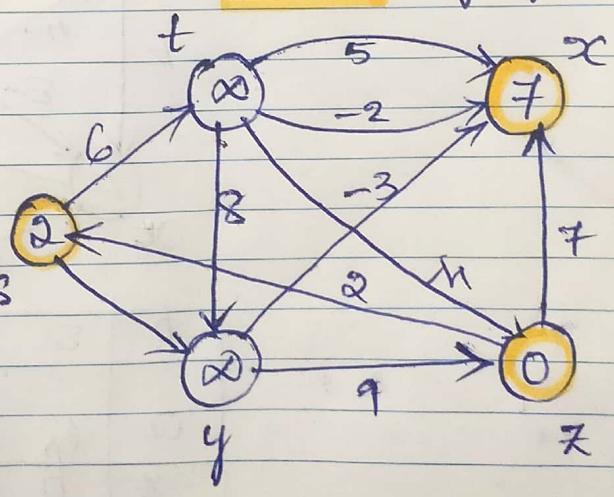
Pass - 1.

	s	t	x	y	z
d	2	∞	7	∞	0
π	z	ϕ	z	ϕ	ϕ

Initial

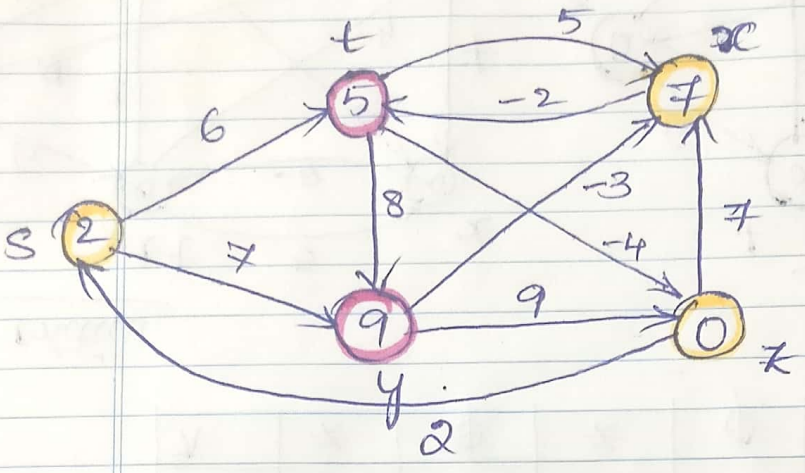


Pass - I (graph).



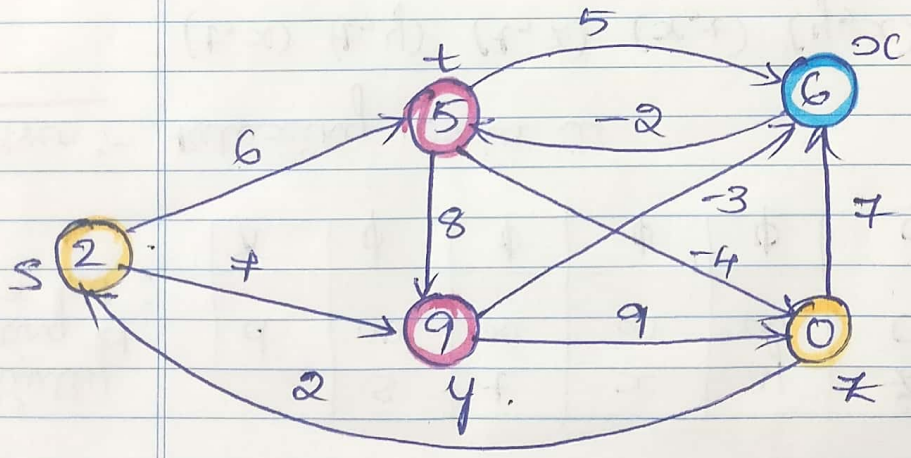
Pass - 2

	S	t	x	y	z
d	2	5	6	9	0
π	z	x	y	s	ϕ



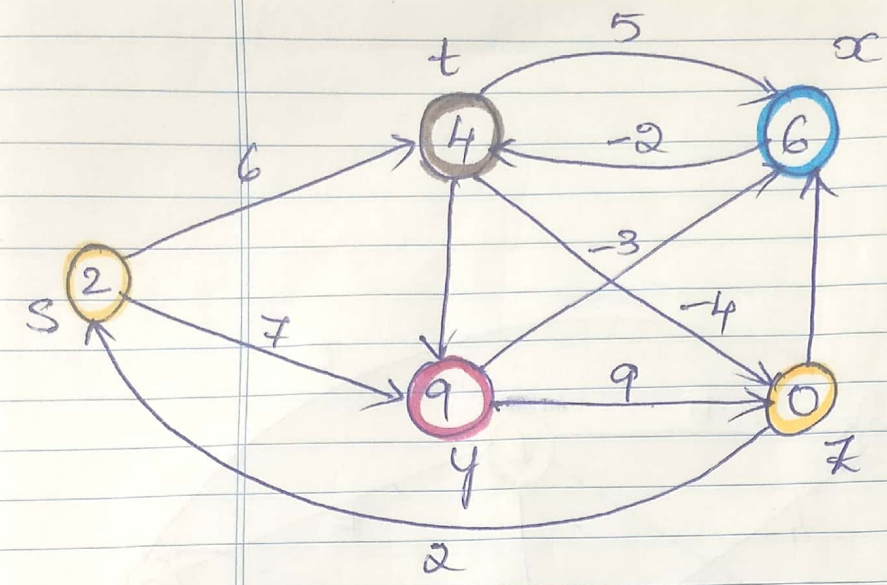
Pass - 3

	S	t	x	y	z
d	2	5	6	9	0
π	z	x	y	s	ϕ



Pass - 4

	S	t	x	y	z
d	2	4	6	9	0
π	z	x	y	s	ϕ



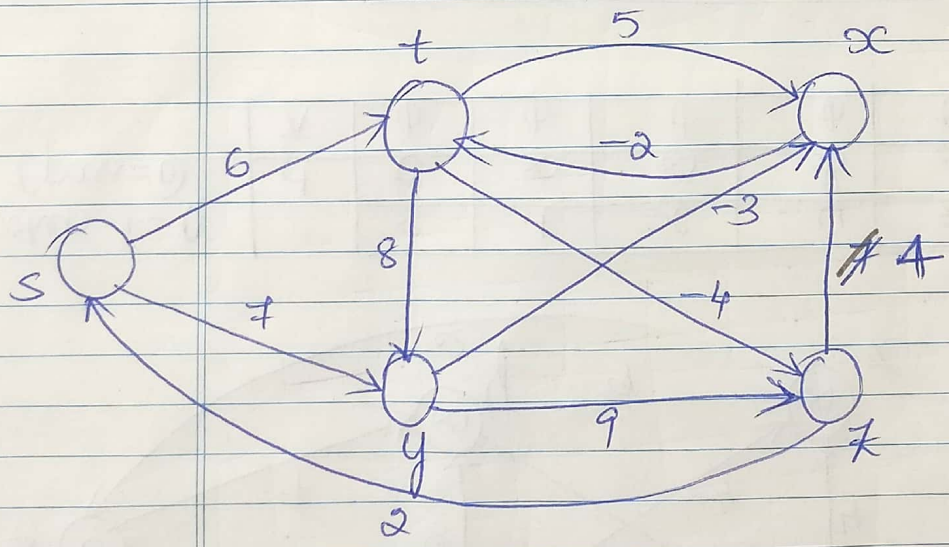
Bellman-Ford takes $|V|-1$ pass.

Here $|V| = 5$

$\therefore$ No of passes $= 5-1 = \underline{\underline{4}}$.

Hence we stop the relaxing process.

Now, by changing weight of edge (z, x) to 4.



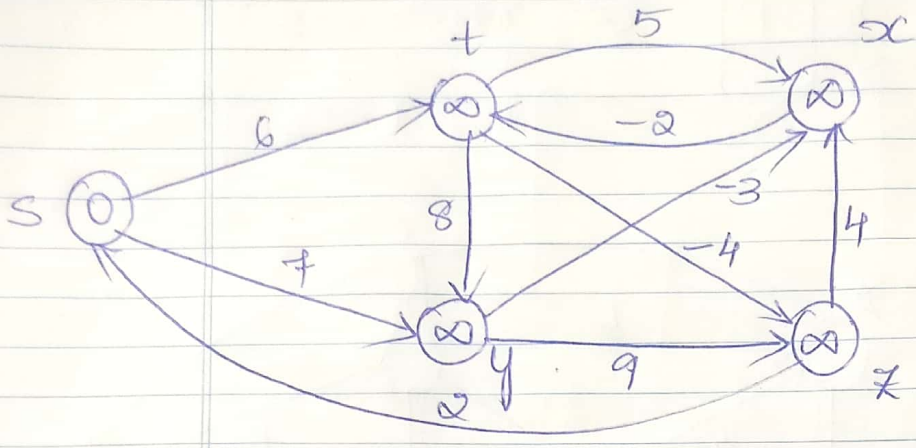
S → source.

$(z, x) = 4$.

S → source $\Rightarrow$

	S
d	0
π	ϕ

Initial graph

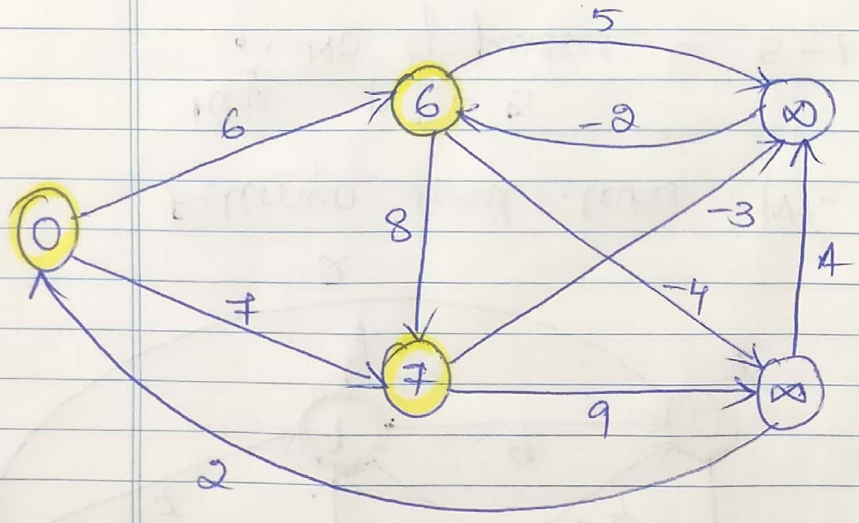


when i = 0
(pass = 0)

	s	t	x	y	z
d	0	∞	∞	∞	∞
π	ϕ	ϕ	ϕ	ϕ	ϕ

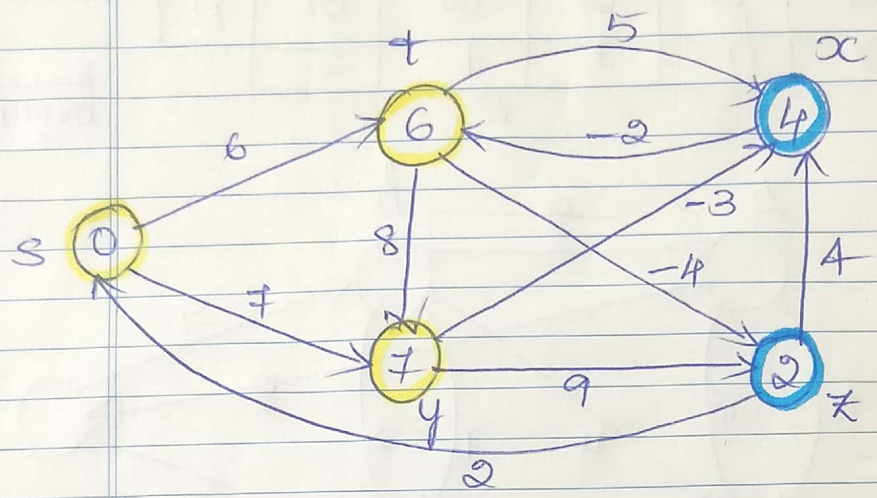
when i = 1
(pass - 1)

	s	t	x	y	z
d	0	6	∞	7	∞
π	ϕ	s	ϕ	s	ϕ



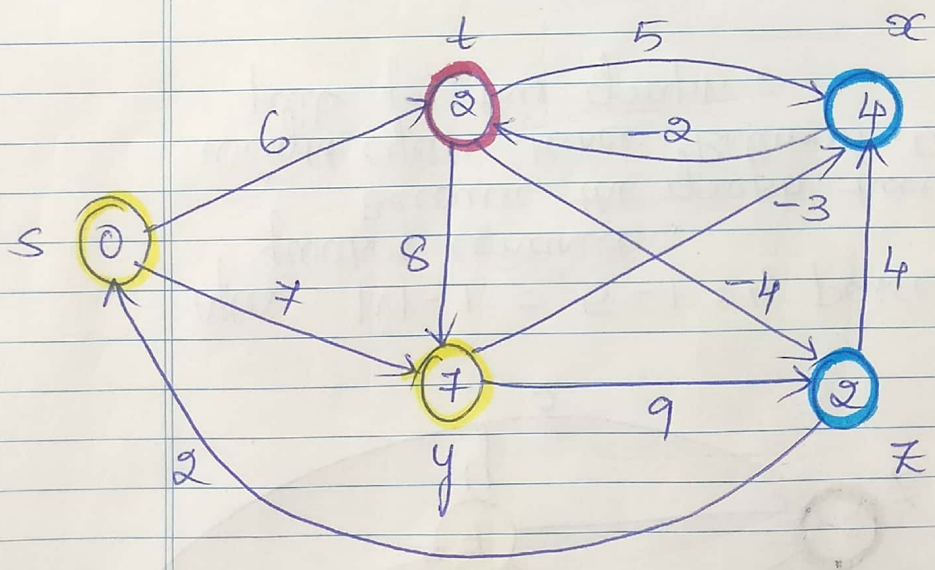
When i=2
(Pass-2)

	s	t	x	y	z
d	0	6	4	7	2
π	ϕ	s	y	s	t



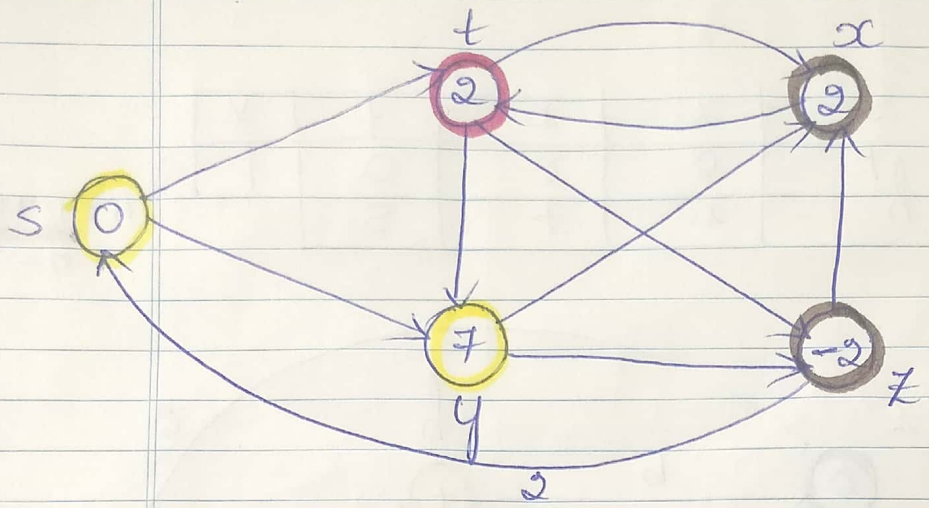
When i=3
(Pass-3)

	s	t	x	y	z
d	0	2	4	7	2
π	ϕ	x	y	s	t



Pass-4
When i=4

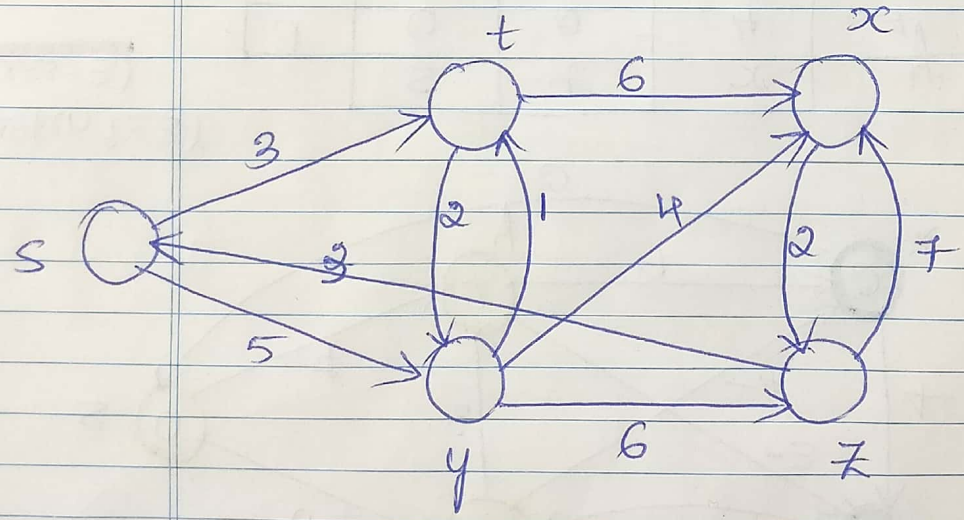
	s	t	x	y	z
d	0	2	2	7	-2
π	ϕ	x	z	s	t



After $|V| - 1 = 5 - 1 = 4$ passes, we cannot further continue.

Because the graph has negative weight cycles, hence Bellman-Ford algorithm fails for this graph.

6. Dijkstra's Algorithm :-



Beginning :-

	d				
	s	t	x	y	z
d	∞	∞	∞	∞	∞
π	ϕ	ϕ	ϕ	ϕ	ϕ

Set $s = \phi$ → empty set.

$Q = \{s, t, x, y, z\}$ // all vertices.

~~Before~~ 1st iteration :- starting with 's' as source

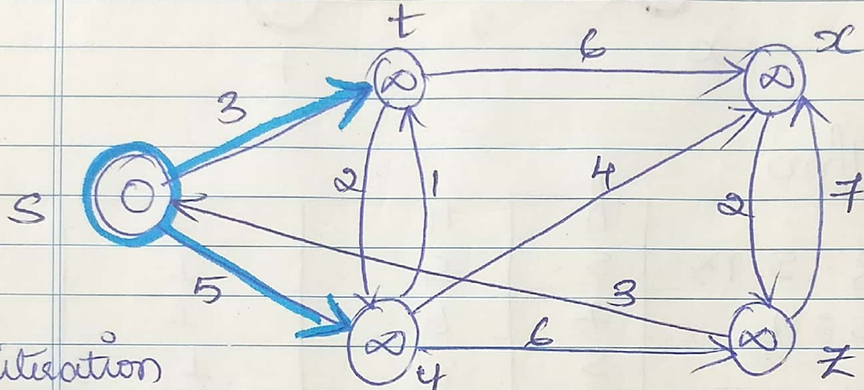
Vertices	visited	d	π
s	T	0	φ
t	F	∞	φ
x	F	∞	φ
y	F	∞	φ
z	F	∞	φ

Q = { s, t, x, y, z }

u = s

S = { s }

adj [s] = (t, y)



2nd iteration

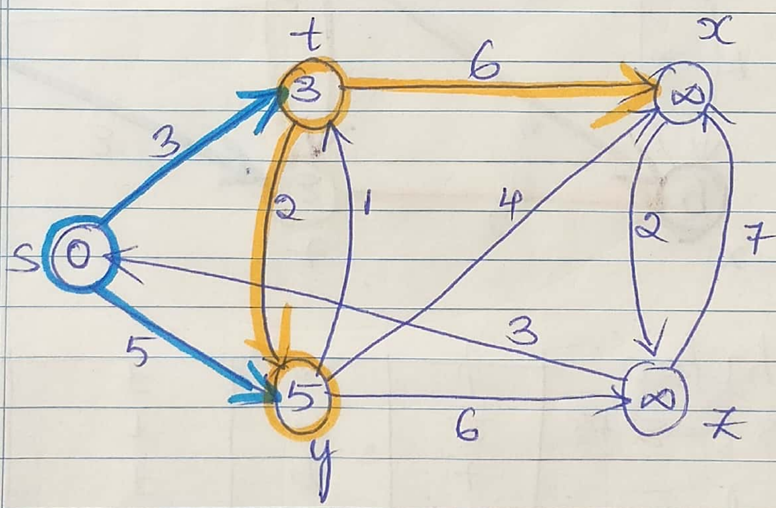
vertex	visited	d	π
s	T	0	φ
t	T	3	s
x	F	∞	φ
y	F	5	s
z	F	∞	φ

Q = { t, x, y, z }

u = t

S = { s, t }

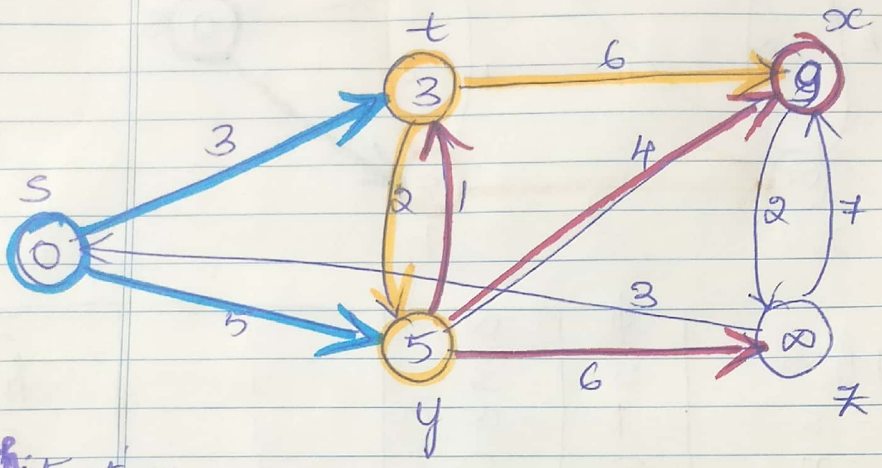
adj [t] = (y, x)



3rd iteration

Vertex	visited	d	π
s	T	0	ϕ
t	T	3	s
x	F	9	t
y	T	5	s
z	F	∞	ϕ

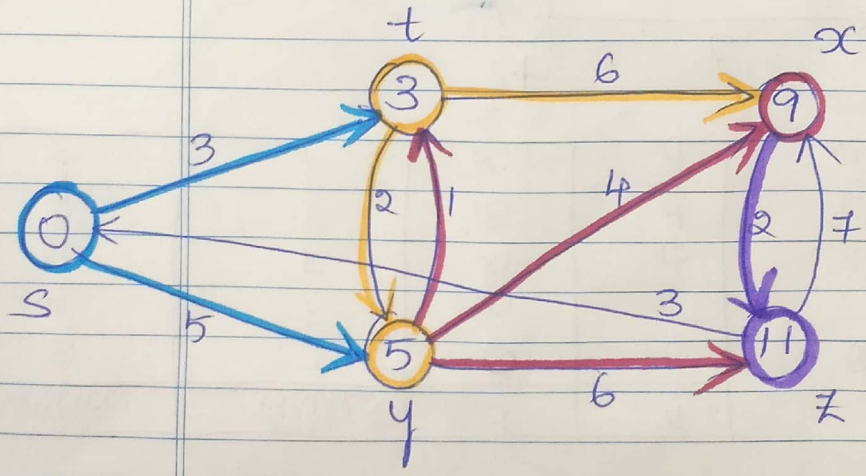
Q = {x, y, z}
u = y.
set S = {s, t, y}
adj[y] = (t, x, z)



4th situation

Vertex	visited	d	π
s	T	0	ϕ
t	T	3	s
x	T	9	t
y	T	5	s
z	F	11	y

Q = {x, z}
u = x.
set S = {s, t, y, z}
adj[x] = z.



5th iteration.

vertex	visited	d	π
s	T	0	ϕ
t	T	3	s
x	T	9	t
y	T	5	s
z	T	11	y

$Q = \{z\}$
$u = z$
$S = \{s, t, y, x, z\}$
$adj[z] = (s, x)$

Finally when all the nodes/vertices are visited (True)

$Q = \phi$ (empty).
Hence while → false.

Thus, no more relaxing the vertices.

∴ Dijkstra's algorithm is successfully applied.