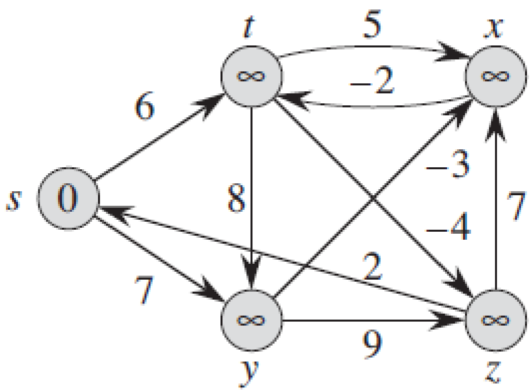
Name: Khushi Nitinkumar Patel

PRN: 2020BTECS00037

Batch: T5

Assignment no 9: Dynamic programming.

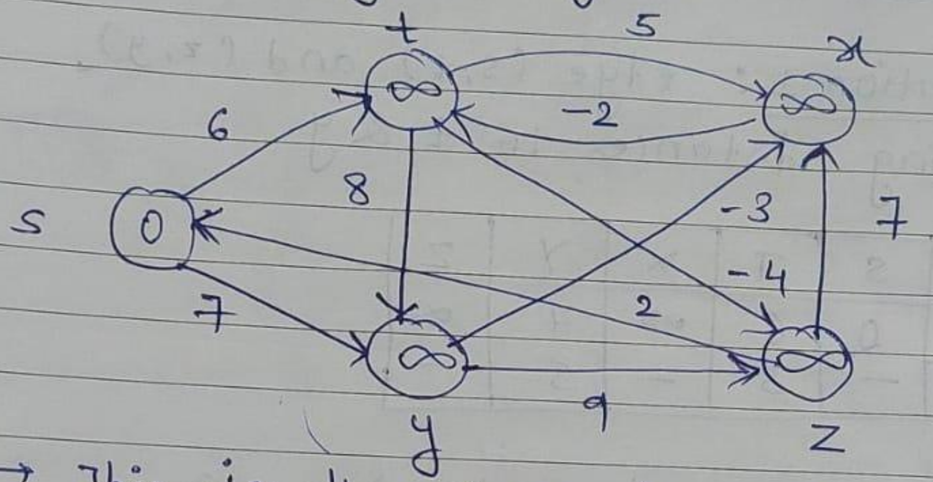
1) From a given vertex in a weighted connected graph, Implement shortest path finding Bellman-Ford algorithm.



Assignment 9

2020BTECS00037.

Dynamic programming.



→ This is the given directed graph.

$(s,t) = 6 \quad (y,x) = -3$.

$(s,y) = 7 \quad (y,z) = 9$.

$(t,y) = 8 \quad (x,t) = -2$

$(t,z) = -4 \quad (z,x) = 7$

$(t,x) = 5 \quad (z,s) = 2$.

Using vertex "s" as source, we initializ all other distance as infinity.

MATRIKAS

Date: / /

	S	T	X	Y	Z
distance	0	∞	∞	∞	∞
Path	–	–	–	–	–

① Iteration 1: edge (s, t) and (z, y), updating distances to t & y.

	S	T	X	Y	Z
distance	0	6	∞	7	∞
Path	–	S	–	S	–

② Iteration 2: edge (t, z), x & z values update.

	S	T	X	Y	Z
distance	0	6	4	7	2
path	–	S	Y	S	T

③ Iteration 3: value of t updated b relaxing (x, t).

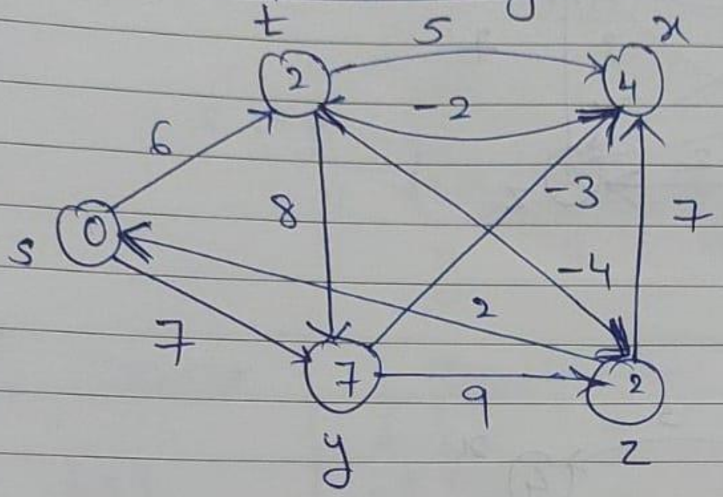
	S	T	X	Y	Z
distance	0	2	4	7	2
path	–	X	Y	S	T

MATRIKAS

Date: / /

(4) Iteration 4: Value of z updated.

Final path diagram.

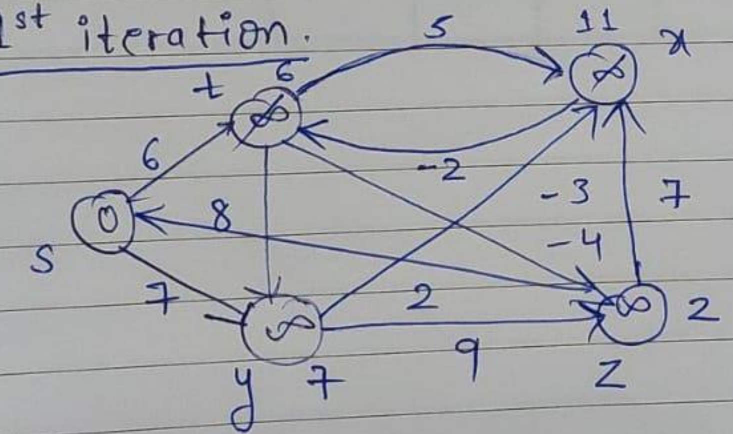


Shortest path for vertices from source

$s \rightarrow t = 2$
$s \rightarrow y = 7$
$s \rightarrow z = -2$
$s \rightarrow x = 4.$

Hence, it takes 4 iterations to find shortest path to every node from source node "s."

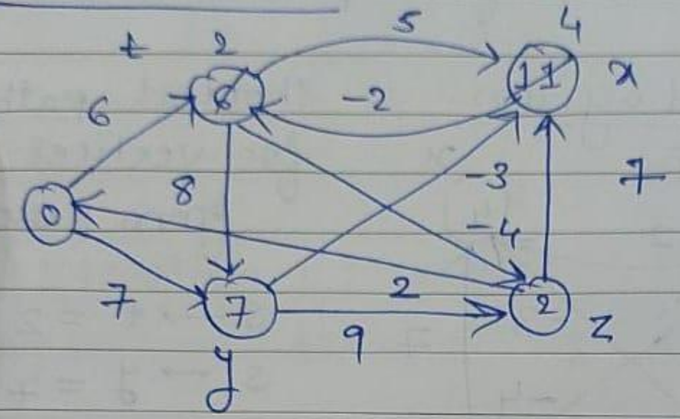
1st iteration.



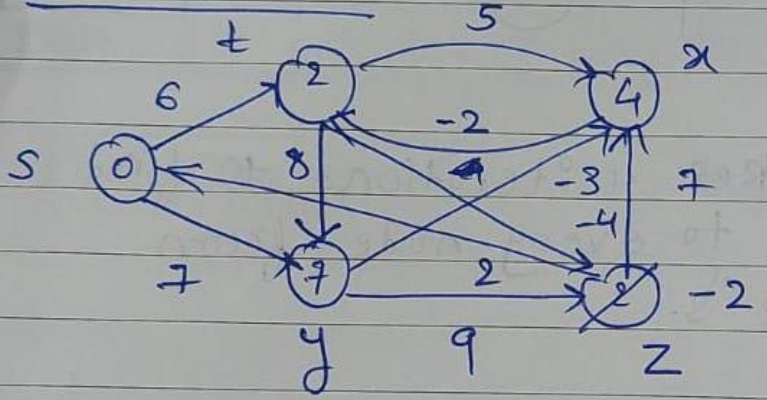
MATRIKAS

Date: / /

2nd iteration.

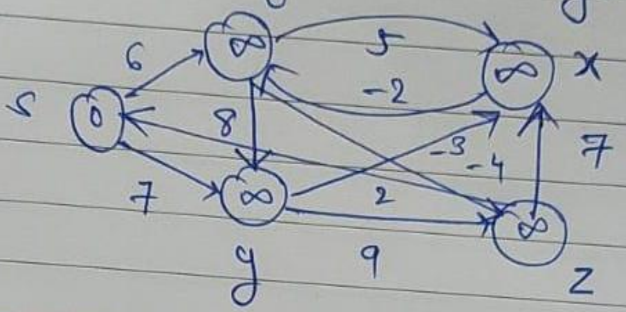


3rd iteration.



Q) Show that Dijkstra’s algorithm doesn’t work for above graph

Q. show that Dijktra's algorithm doesn't work for above graph.



Source	Destination			
	t	x	y	z
{s}	6*	∞	7	∞
{s, t}	6*	11	7	2*
{s, t, z}	6*	9	7*	2*
{s, t, z, y}	6*	4	7*	2*
{s, t, z, y, x}				

But this is not the shortest path. All the vertices can be reached from source, but not with min distance due to presence of some negative edges 't' can be reached with min distance of 2 & z with -2 but with Dijkstra's algo. it doesn't happen ∴ Dijkstra's algo. fails for above graph

MATRIKAS

Q)

Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let m be the maximum over all vertices $v \in V$ of the minimum number of edges in a shortest path from the source s to v. (Here, the shortest path is by weight, not the number of edges.) Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m + 1$ passes, even if m is not known in advance.

Path relaxation property of Bellman-ford implies that every vertex in the graph has achieved shortest path weight in "v.d" after m-iterations. But we don't know for sure that no d value will change in (m+1)th iteration so we cannot terminate it at min iteration. So we can make a Bellman-ford algorithm such that it will stop when nothing changes after (m+1)th iteration.

The change to the Bellman-Ford algorithm to implement this optimization is:

Check if v was relaxed or not.

If v is relaxed then we wait to see if v was updated (which means being relaxed again).

If v was not updated, then we would stop

CODE:

```
#include <bits/stdc++.h>
using namespace std;
struct Edge
{
int src, dest, weight;
};
struct Graph
{
int V, E; // V & E: No. of vertices and edges resp
struct Edge *edge;
};
struct Graph *create_graph(int V, int E)
{
struct Graph *graph = new Graph;
graph->V = V;
graph->E = E;
graph->edge = new Edge[E];
return graph;
};
void printArray(int dist[], int n)
{
printf("Vertex \t Distance from Source\n");
for (int i = 0; i < n; ++i)
{
printf("%d \t\t %d\n", i, dist[i]);
}
};
void Bellman_Ford(struct Graph *graph, int src)
{
int V = graph->V;
int E = graph->E;
int dist[V];
for (int i = 0; i < V; i++)
{
dist[i] = INT_MAX;
}
dist[src] = 0;
for (int i = 1; i <= V - 1; i++)
{
for (int j = 0; j < E; j++)
{
for (int j = 0; j < E; j++)
{
int u = graph->edge[j].src;
int v = graph->edge[j].dest;
int weight = graph->edge[j].weight;
if (dist[u] != INT_MAX && dist[u] + weight < dist[v]){
```

```
dist[v] = dist[u] + weight;
}
}
}
}
printArray(dist, V);
return;
}
int main()
{
int V = 5;
int E = 8;
struct Graph *graph = create_graph(V, E);
graph->edge[0].src = 0;
graph->edge[0].dest = 1;
graph->edge[0].weight = -1;
graph->edge[1].src = 0;
graph->edge[1].dest = 2;
graph->edge[1].weight = 4;
graph->edge[2].src = 1;
graph->edge[2].dest = 2;
graph->edge[2].weight = 3;
graph->edge[3].src = 1;
graph->edge[3].dest = 3;
graph->edge[3].weight = 2;
graph->edge[4].src = 1;
graph->edge[4].dest = 4;
graph->edge[4].weight = 2;
graph->edge[5].src = 3;
graph->edge[5].dest = 2;
graph->edge[5].weight = 5;
graph->edge[6].src = 3;
graph->edge[6].dest = 1;
graph->edge[6].weight = 1;
graph->edge[7].src = 4;
graph->edge[7].dest = 3;
graph->edge[7].weight = -3;
Bellman_Ford(graph, 0);
return 0;
}
```

OUTPUT

```
Vertex   Distance from Source
0                0
1                -1
2                2
3                -2
4                1
```