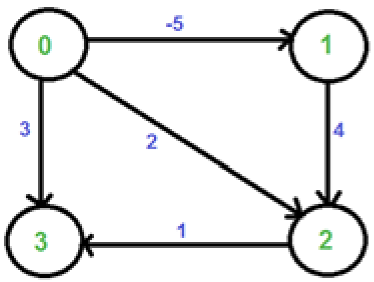
IT300 Assignment 5

NAME: SUYASH CHINTAWAR
ROLL NO.: 191IT109
TOPIC: SHORTEST PATH ALGORITHMS - 2

Q1. Implement Johnson's algorithm for All-pairs shortest paths to find shortest paths between every pair of vertices in a given weighted directed graph and weights may be negative. Show the step by step procedure of finding the all-pairs shortest path of the following graph and compare it with the answer obtained using the program you implemented.



SOLUTION:

Code:

```
/*
This program implements Johnson's Algorithm
with time complexity O(VE+V^2logE)
Adjacency list has been used to store modified graph
NOTE:
1) Vertices must be 0-based
2) Please refrain from giving wrong inputs
*/
#include<bits/stdc++.h>
using namespace std;
#define f first
#define s second
typedef pair<int,int> pii;

//Compute adjacency list of graph
void adjacency_list(int v,int edges,vector<pair<pii,int>>
edge,vector<pii> adj_list[])
{
    for(int i=0;i<edges;i++)
    {
        adj_list[edge[i].f.f].push_back({edge[i].f.s,edge[i].s});
```

```
    }
}

//Perform Bellman-Ford Algorithm
void bellman_ford(int v, int src, vector<pair<pii,int>> edge,
vector<int> &d)
{
     d[src]=0; // distance of source is zero from itself
     int num_edges = edge.size();
     for(int i=0;i<v-1;i++) //run |v|-1 times
     {
          for(int j=0;j<num_edges;j++) // check for updates in each edge
          {
               int s,e;
               int cost;
               s=edge[j].f.f; //starting of edge
               e=edge[j].f.s; // end of edge
               cost=edge[j].s;
               if(d[s]<INT_MAX) //update min cost distance found
               {
                    d[e] = min(d[e],d[s]+cost);
               }
          }
     }
}

//Implement Dijkstra's Algorithm
vector<int> dijkstra(int src,int n, vector<pii> adj_list[])
{
    vector<int> d(n,INT_MAX);//distances/costs of all vertices,
initialized to infinite
    vector<int> found(n,0);//to store shortest distances/costs
    d[src]=0;
    priority_queue<pii,vector<pii>,greater<pii>> q;//min-heap
    q.push({0,src});//pushing starting vertex as distance is zero from
itself
```

```
    while(!q.empty())
    {
        int dv=q.top().first;//distance of shortest distanced vertex
        int v=q.top().second;//vertex with shortest distance
        q.pop();
        if(dv!=d[v]) continue;//occur when path of adjacent vertex 
already is shortest
        found[v]=dv;
        for(auto x:adj_list[v])//check and update adjacent vertices' 
distances
        {
            int adj_x=x.first;
            int weight=x.second;
            if(dv+weight < d[adj_x])//shorter path found
            {
                d[adj_x]=dv+weight;
                q.push({d[adj_x],adj_x});
            }
        }
    }
    return d;
}

int main()
{
     //Take input from user
     int v,edges;
     vector<pair<pii,int>> edge;
     cout<<"Enter number of vertices: ";
     cin>>v;
     cout<<"Enter number of edges: ";
     cin>>edges;
     cout<<"Enter "<<edges<<" edges (each line format: v1 v2 
cost):\nNOTE: vertices must be 0-based\n";
     for(int i=0;i<edges;i++)
     {
```

```
        int s,e,cost;
        cin>>s>>e>>cost;
        edge.push_back({{s,e},cost});
    }

    //distances we get from Bellman Ford Algorithm
    vector<int> d(v+1,INT_MAX);

    //add new vertex and generate edges to all vertices with weight 0
    for(int i=0;i<v;i++)
    {
        edge.push_back({{v,i},0});
    }

    // Call bellman ford with source vertex as the new vertex
    bellman_ford(v+1,v,edge,d);

    //remove the extra added edges after Bellman Ford is finished
    edge.erase(edge.begin()+edges,edge.end());

    //update edge costs so that no negative edge weights remain
    for(int i=0;i<edges;i++)
    {
        int s,e;
        int cost;
        s=edge[i].f.f; //starting of edge
        e=edge[i].f.s; // end of edge
        cost=edge[i].s;
        edge[i].s=cost+d[s]-d[e];
    }

    cout<<"\nModified graph(edge list):\n";
    for(int i=0;i<edges;i++)
    {
        cout<<edge[i].f.f<<" "<<edge[i].f.s<<" "<<edge[i].s<<endl;
    }
```

```
     //storing adjacency list
     vector<pii> adj_list[v];
    adjacency_list(v,edges,edge,adj_list);

    //Stores our final distance matrix
    vector<vector<int>> dist_matrix;

     //dijkstra on all vertices
    for(int i=0;i<v;i++)
    {
     vector<int> temp=dijkstra(i,v,adj_list);
     dist_matrix.push_back(temp);
    }

    cout<<"\n*****FINAL DISTANCE MATRIX*****\n";
    for(int i=0;i<v;i++)
    {
     for(int j=0;j<v;j++)
     {
          if(dist_matrix[i][j]!=INT_MAX)
               dist_matrix[i][j]=dist_matrix[i][j]-d[i]+d[j];
          if(dist_matrix[i][j]!=INT_MAX)
               cout<<setw(3)<<dist_matrix[i][j]<<" ";
          else cout<<"INF ";

     }
     cout<<endl;
    }
}
```

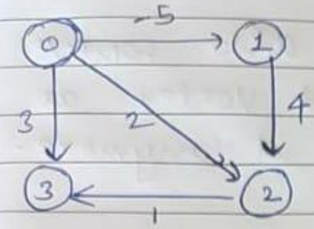
Output:

```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT300/Assignment5$ g++ johnsons-algo.cpp
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT300/Assignment5$ ./a.out
Enter number of vertices: 4
Enter number of edges: 5
Enter 5 edges (each line format: v1 v2 cost):
NOTE: vertices must be 0-based
0 1 -5
0 2 2
0 3 3
1 2 4
2 3 1

Modified graph(edge list):
0 1 0
0 2 3
0 3 3
1 2 0
2 3 0

*****FINAL DISTANCE MATRIX*****
  0  -5  -1   0
INF   0   4   5
INF INF   0   1
INF INF INF   0
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT300/Assignment5$
```

The time complexity of this algorithm is $O(VE+V^2logE)$ because adjacency lists have been used for Dijsktra's Algorithm.

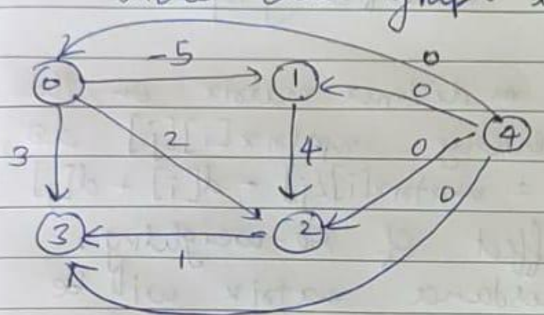
Steps:

(continued...)



Original graph

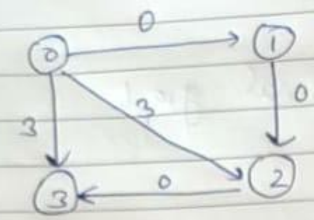
<u>Step 1</u>: We add a new vertex, and add edges that connect this new vertex to all other vertices with edge weight 0.
Now our graph is,



<u>Step 2</u>: Now we perform Bellman-Ford on this modified graph ~~and~~ with the new vertex as source vertex. We will get a distance vector 'd' for all vertices d[0...v].

<u>Step 3</u>: Change the edge weights for all edges. We do this because in the next step we will do Dijkstra's on each vertex & it doesn't support negative weight edges.
~~Ea~~ for every edge (u, v) do,
cost (u, v) = cost (u, v) + d[u] - d[v].

Now, our graph becomes.



We also remove the added vertex as we don't need it anymore

Step 4: Run dijkstra with every vertex as source vertex. Store these individual arrays obtained and we get the distance matrix on the modified graph.

Step 5: To get distance matrix on the original graph, for every matrix[i][j] do,

matrix[i][j] = matrix[i][j] - d[i] + d[j]

to cancel the effect of re-weighting.

Hence, final distance matrix will be

0	-5	-1	0
INF	0	4	5
INF	INF	0	1
INF	INF	INF	0

which matches the program output.

Time complexity:

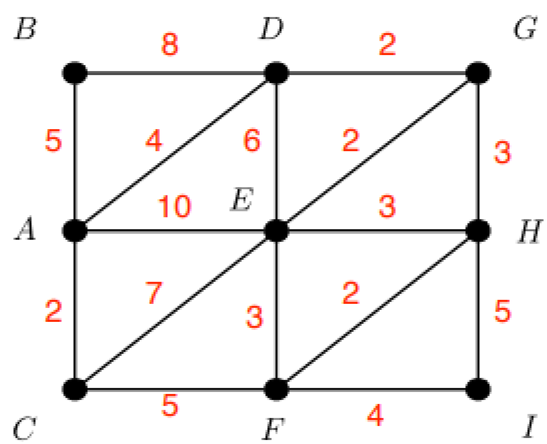
$O(VE)$ for Bellman ford +

$O(V) * O(V \log E)$ for Dijsktra on each vertex

$=$

$\boxed{O(V^2 \log E + VE)}$.

Q2. (a) Apply the Floyd-Warshall algorithm to the graph in the following graph. Write a program to find the shortest path from A to H and display the initial values of the d(i, j), i, j = 1, 2, . . . , 9; the values after k = 1, k = 3, and k = 5; and the final values.



SOLUTION:

Code:

```
/*
This program solves all pair shortest path problem
using Floyd-Warshall Algorithm
NOTE:
1) Vertices must be 1-based
2) Please refrain from giving wrong inputs
*/
#include<bits/stdc++.h>
using namespace std;
#define f first
#define s second
typedef pair<int,int> pii;

//Printing the distance matrix
void print(vector<vector<int>> d)
{
     for(int i=0;i<d.size();i++)
     {
```

```
        for(int j=0;j<d[0].size();j++)
        {
            if(d[i][j]!=INT_MAX) cout<<setw(3)<<d[i][j]<<" ";
            else cout<<"INF ";
        }
        cout<<endl;
    }
}

//Floyd-Warshall Algorithm
void floyd_warshall(int v, int edges, vector<pair<pii,int>> edge,
vector<vector<int>> &d)
{
    //Initialize distance matrix with infinite
    for(int i=0;i<v;i++)
    {
        for(int j=0;j<v;j++) d[i][j]=INT_MAX;
    }

    //As there would be no loops, main diagonal will be zeros
    for(int i=0;i<v;i++) d[i][i]=0;

    //Fill the initial state using the edges in graph
    for(int i=0;i<edges;i++)
    {
        d[edge[i].f.f-1][edge[i].f.s-1] = edge[i].s;
        d[edge[i].f.s-1][edge[i].f.f-1] = edge[i].s;
    }

    cout<<"\n***INITIAL STATE OF DISTANCE MATRIX"<<":***\n";
    print(d);

    //Floyd-Warshall main code
    for(int k=0;k<v;k++) //k will be our intermediate vertex
    {
        //run for each d[i][j] via vertex 'k'
```

```
		for(int i=0;i<v;i++)
		{
			for(int j=0;j<v;j++)
			{
				if(d[i][k]<INT_MAX and d[k][j]<INT_MAX)
				{
					//update distance
					d[i][j]=min(d[i][j],d[i][k]+d[k][j]);
				}
			}
		}
		//print matrix as stated in question
		if(k==0 or k==2 or k==4)
		{
			cout<<"\n***STATE OF DISTANCE MATRIX AT K = 
"<<k+1<<":***\n";
			print(d);
		}
	}
}

int main()
{
	//Take input from user
	int v,edges;
	vector<pair<pii,int>> edge;
	cout<<"Enter number of vertices: ";
	cin>>v;
	cout<<"Enter number of edges: ";
	cin>>edges;
	cout<<"Enter "<<edges<<" edges (each line format: v1 v2 
cost):\nNOTE: vertices must be 1-based\n";
	for(int i=0;i<edges;i++)
	{
		int s,e,cost;
		cin>>s>>e>>cost;
```

```
        edge.push_back({{s,e},cost});
    }

    //Distance matrix
    vector<vector<int>> d(v,vector<int> (v,INT_MAX));

    //Call floyd-warshall algorithm
    floyd_warshall(v,edges,edge,d);

    //Print final state
    cout<<"\n***FINAL STATE OF DISTANCE MATRIX"<<":***\n";
    print(d);

}
```

<u>Output:</u>

```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT300/Assignment5$ g++ floyd-warshall.cpp
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT300/Assignment5$ ./a.out
Enter number of vertices: 9
Enter number of edges: 16
Enter 16 edges (each line format: v1 v2 cost):
NOTE: vertices must be 1-based
1 2 5
1 4 4
1 3 2
1 5 10
2 4 8
3 5 7
3 6 5
4 5 6
4 7 2
5 6 3
5 7 2
5 8 3
6 9 4
6 8 2
7 8 3
8 9 54

***INITIAL STATE OF DISTANCE MATRIX:***
  0   5   2   4  10 INF INF INF INF
  5   0 INF   8 INF INF INF INF INF
  2 INF   0 INF   7   5 INF INF INF
  4   8 INF   0   6 INF   2 INF INF
 10 INF   7   6   0   3   2   3 INF
INF INF   5 INF   3   0 INF   2   4
INF INF INF   2   2 INF   0   3 INF
INF INF INF INF   3   2   3   0  54
INF INF INF INF INF   4 INF  54   0

***STATE OF DISTANCE MATRIX AT K = 1:***
  0   5   2   4  10 INF INF INF INF
  5   0   7   8  15 INF INF INF INF
  2   7   0   6   7   5 INF INF INF
  4   8   6   0   6 INF   2 INF INF
 10  15   7   6   0   3   2   3 INF
INF INF   5 INF   3   0 INF   2   4
INF INF INF   2   2 INF   0   3 INF
INF INF INF INF   3   2   3   0  54
INF INF INF INF INF   4 INF  54   0
```

```
***STATE OF DISTANCE MATRIX AT K = 3:***
  0   5   2   4   9   7 INF INF INF
  5   0   7   8  14  12 INF INF INF
  2   7   0   6   7   5 INF INF INF
  4   8   6   0   6  11   2 INF INF
  9  14   7   6   0   3   2   3 INF
  7  12   5  11   3   0 INF   2   4
INF INF INF   2   2 INF   0   3 INF
INF INF INF INF   3   2   3   0  54
INF INF INF INF INF   4 INF  54   0

***STATE OF DISTANCE MATRIX AT K = 5:***
  0   5   2   4   9   7   6  12 INF
  5   0   7   8  14  12  10  17 INF
  2   7   0   6   7   5   8  10 INF
  4   8   6   0   6   9   2   9 INF
  9  14   7   6   0   3   2   3 INF
  7  12   5   9   3   0   5   2   4
  6  10   8   2   2   5   0   3 INF
 12  17  10   9   3   2   3   0  54
INF INF INF INF INF   4 INF  54   0

***FINAL STATE OF DISTANCE MATRIX:***
  0   5   2   4   8   7   6   9  11
  5   0   7   8  12  12  10  13  16
  2   7   0   6   7   5   8   7   9
  4   8   6   0   4   7   2   5  11
  8  12   7   4   0   3   2   3   7
  7  12   5   7   3   0   5   2   4
  6  10   8   2   2   5   0   3   9
  9  13   7   5   3   2   3   0   6
 11  16   9  11   7   4   9   6   0
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT300/Assignment5$
```

As we can see, the shortest path cost from vertex A(or 1) to vertex H(or 8) is d[0][7]=9. The time complexity of the Floyd-Warshall Algorithm here is O(V^3).

(b) How can the output of the Floyd-Warshall algorithm be used to detect the presence of a negative-weight cycle? Explain your answer.
SOLUTION:

We can find negative weight cycles if the distance of a vertex from itself is less than 0. This implies that if any of the diagonal elements in the distance matrix is less than zero, there exists a negative weight cycle in the graph.

THANK YOU