**Lab Program 5:**

**Write a program to find the shortest path between vertices using bellman-ford algorithm.**

**Code -**

import java.util.Scanner;

public class bellmanford

{

public int distance[]; **// distance array**

public int numb\_vert; **// number of vertices**

public static final int MAX\_VALUE=999; **//represent infinity or an unreachable value**

public bellmanford(int numb\_vert)  **// Constructor**

{

this.numb\_vert = numb\_vert;  **// initializing the number of vertices**

distance = new int[numb\_vert+1]; **allocating memory for the distance array based on the number of vertices.**

}

**/\* implements the core logic of the Bellman-Ford algorithm for finding the shortest paths from a given source vertex to all other vertices in a graph. \*/**

public void BellmanfordpEvaluation (int source,int adj\_matrix[][])

{

for(int node=1; node<=numb\_vert; node++)

**/\*It initializes the distance array for all vertices to MAX\_VALUE \*/**

distance[node]=MAX\_VALUE;

**/\*Sets the distance of the source vertex to 0 \*/**

distance[source]=0;

**/\* Within each iteration, it iterates over all edges of the graph represented by the adjacency matrix \*/**

for(int node=1;node<=numb\_vert-1; node++) **// number of iterations**

{

for(int src\_node=1;src\_node<=numb\_vert;src\_node++)

{

for(int dest\_node=1;dest\_node<=numb\_vert;dest\_node++)

{

if(adj\_matrix[src\_node][dest\_node]!=MAX\_VALUE)

{

if(distance[dest\_node] > distance[src\_node] + adj\_matrix[src\_node][dest\_node])

distance[dest\_node] = distance[src\_node] + adj\_matrix[src\_node][dest\_node];

}

}

}

}

**/\* Negative Cycle Detection:\*/**

for(int src\_node=1;src\_node<=numb\_vert;src\_node++)

{

for(int dest\_node=1;dest\_node<=numb\_vert;dest\_node++)

{

if(adj\_matrix[src\_node][dest\_node]!=MAX\_VALUE)

{

if(distance[dest\_node] > distance[src\_node] +adj\_matrix[src\_node][dest\_node])

{

System.out.println("The graph contains negative edge cycle");

}

}

}

}

**/\* *it prints the routing table for the source router, displaying the shortest distance from the source to each destination vertex.*\*/**

System.out.println("Routing Table for Router " + source+" is");

System.out.println("Destination Distance\t");

for(int vertex=1;vertex<=numb\_vert;vertex++)

System.out.println(+vertex+"\t\t\t"+distance[vertex]);

}

public static void main(String args[])

{

int numb\_vert=0;

int source;

Scanner **scan** = new Scanner(System.in);

**/\*Reads the number of vertices from the standard input using a Scanner.\*/**

System.out.println("Enter the number of vertices");

numb\_vert = scan.nextInt();

**/\* Creates an adjacency matrix\*/**

int adj\_matrix[][] = new int[numb\_vert+1][numb\_vert+1];

System.out.println("Enter the adjacency matrix");

for(int src\_node=1;src\_node<=numb\_vert;src\_node++)

for(int dest\_node=1;dest\_node<=numb\_vert;dest\_node++)

{

adj\_matrix[src\_node][dest\_node] = scan.nextInt();

**/\* If the diagonal elements (where source node equals destination node) are encountered, they are set to 0.\*/**

if(src\_node==dest\_node)

{

adj\_matrix[src\_node][dest\_node]=0;

continue;

}

**/\* If any element is 0 (indicating no direct edge), it's set to MAX\_VALUE, indicating an infinite distance. \*/**

if(adj\_matrix[src\_node][dest\_node]==0)

adj\_matrix[src\_node][dest\_node]=MAX\_VALUE;

} **// end of for loop**

for(int i=1;i<=numb\_vert;i++)

{

**/\* For each vertex, creates an instance of the bellmanford class and runs the BellmanfordpEvaluation method \*/**

bellmanford bellmanford = new bellmanford(numb\_vert);

bellmanford.BellmanfordpEvaluation(i,adj\_matrix);

}

scan.close(); **//Closes the Scanner to release system resources.**

}

}

**Explanation**

* bellmanford Class: This is the main class containing the implementation of the Bellman-Ford algorithm.
* Constructor: Initializes the number of vertices and the distance array.
* BellmanfordpEvaluation Method: This method computes the shortest paths from a given source vertex to all other vertices using the Bellman-Ford algorithm.
* Main Method:
  + Takes input for the number of vertices and the adjacency matrix representing the graph.
  + Initializes an adjacency matrix based on user input.
  + Creates an instance of the bellmanford class for each vertex and calls the BellmanfordpEvaluation method for each vertex.
* Input:
  + The number of vertices in the graph.
  + The adjacency matrix representing the weighted edges of the graph.
* Output:
  + The routing table for each router, showing the shortest distance from the source router to all other routers.

BellmanfordpEvaluation method implements the core logic of the Bellman-Ford algorithm for finding the shortest paths from a given source vertex to all other vertices in a graph. Here's a breakdown of how it works:

* Initialization:
  + It initializes the distance array for all vertices to MAX\_VALUE, representing an infinite distance.
  + Sets the distance of the source vertex to 0, as the distance from the source to itself is 0.
* Relaxation:
  + It iterates numb\_vert - 1 times, where numb\_vert is the number of vertices in the graph. This ensures that the shortest paths are found after at most numb\_vert - 1 iterations.
  + Within each iteration, it iterates over all edges of the graph represented by the adjacency matrix (adj\_matrix).
  + For each edge (src\_node, dest\_node) with weight w, it relaxes the edge if there is a shorter path from the source to dest\_node through src\_node. If distance[dest\_node] > distance[src\_node] + w, it updates distance[dest\_node] to the shorter distance.
* Negative Cycle Detection:
  + After the relaxation step, it performs an additional iteration over all edges to check for negative cycles.
  + If after relaxation, any distance can still be improved, it means there's a negative cycle in the graph, and it prints a message indicating the presence of a negative cycle.
* Output:
  + Finally, it prints the routing table for the source router, displaying the shortest distance from the source to each destination vertex.

Overall, your implementation correctly follows the steps of the Bellman-Ford algorithm. It initializes distances, iterates to relax edges and find shortest paths, checks for negative cycles, and outputs the routing table.