

Program 5:

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

program:

Distance Vector Algorithm is a decentralized routing algorithm that requires that each router simply inform its neighbors of its routing table. For each network path, the receiving routers pick the neighbor advertising the lowest cost, then add this entry into its routing table for re-advertisement. To find the shortest path, Distance Vector Algorithm is based on one of two basic algorithms: the Bellman-Ford and the Dijkstra algorithms.

Routers that use this algorithm have to maintain the distance tables (which is a one- dimension array -- "a vector"), which tell the distances and shortest path to sending packets to each node in the network. The information in the distance table is always up date by exchanging information with the neighboring nodes. The number of data in the table equals to that of all nodes in networks (excluded itself). The columns of table represent the directly attached neighbors whereas the rows represent all destinations in the network. Each data contains the path for sending packets to each destination in the network and distance/or time to transmit on that path (we call this as "cost"). The measurements in this algorithm are the number of hops, latency, the number of outgoing packets, etc.

The Bellman–Ford algorithm is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. It is slower than Dijkstras algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm. If a graph contains a "negative cycle" (i.e. a cycle whose edges sum to a negative value) that is reachable from the source, then there is no cheapest path: any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman–Ford algorithm can detect negative cycles and report their existence

Source code:

```
import java.util.Scanner;

public class BellmanFord
{
    private int D[];
    private int num_ver;
    public static final int MAX_VALUE = 999;

    public BellmanFord(int num_ver)
    {
        this.num_ver = num_ver;
        D = new int[num_ver + 1];
    }
}
```

```

public void BellmanFordEvaluation(int source, int A[][])
{
    for (int node = 1; node <= num_ver; node++)
    {
        D[node] = MAX_VALUE;
    }
    D[source] = 0;
    for (int node = 1; node <= num_ver - 1; node++)
    {
        for (int sn = 1; sn <= num_ver; sn++)
        {
            for (int dn = 1; dn <= num_ver; dn++)
            {
                if (A[sn][dn] != MAX_VALUE)
                {
                    if (D[dn] > D[sn] + A[sn][dn])
                        D[dn] = D[sn] + A[sn][dn];
                }
            }
        }
    }
    for (int sn = 1; sn <= num_ver; sn++)
    {
        for (int dn = 1; dn <= num_ver; dn++)
        {
            if (A[sn][dn] != MAX_VALUE)
            {
                if (D[dn] > D[sn] + A[sn][dn])
                    System.out.println("The Graph contains negative
egde cycle");
            }
        }
    }
    for (int vertex = 1; vertex <= num_ver; vertex++)
    {
        System.out.println("distance of source " + source + " to
        "+ vertex + "is " + D[vertex]);
    }
}

public static void main(String[ ] args)
{
    int num_ver = 0;
    int source;

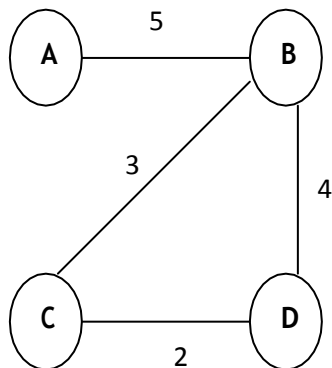
```

```

Scanner scanner = new Scanner(System.in);
System.out.println("Enter the number of vertices");
num_ver = scanner.nextInt();
int A[][] = new int[num_ver + 1][num_ver + 1];
System.out.println("Enter the adjacency matrix");
for (int sn = 1; sn <= num_ver; sn++)
{
    for (int dn = 1; dn <= num_ver; dn++)
    {
        A[sn][dn] = scanner.nextInt();
        if (sn == dn)
        {
            A[sn][dn] = 0;
            continue;
        }
        if (A[sn][dn] == 0)
        {
            A[sn][dn] = MAX_VALUE;
        }
    }
}
System.out.println("Enter the source vertex");
source = scanner.nextInt();
BellmanFord b = new BellmanFord (num_ver);
b.BellmanFordEvaluation(source, A);
scanner.close();
}
}

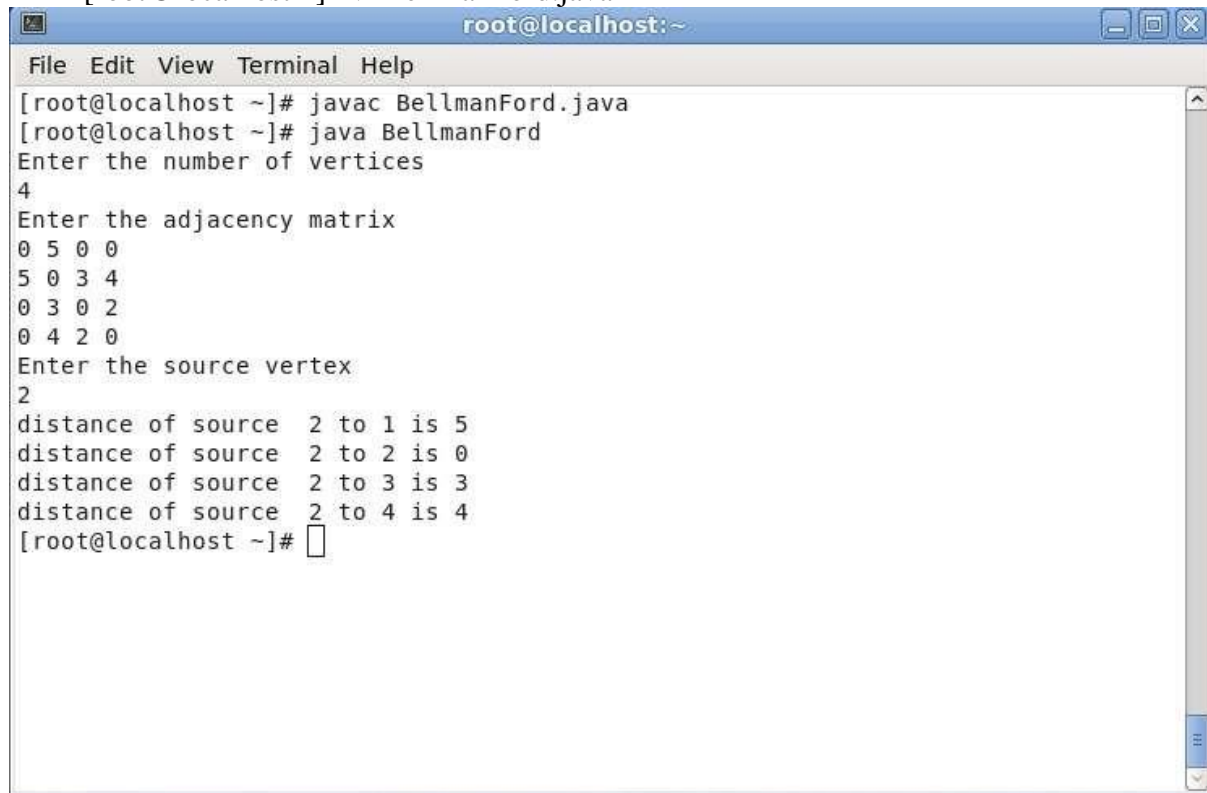
```

Input graph:



Output:

[root@localhost ~]# vi BellmanFord.java



```
File Edit View Terminal Help
[root@localhost ~]# javac BellmanFord.java
[root@localhost ~]# java BellmanFord
Enter the number of vertices
4
Enter the adjacency matrix
0 5 0 0
5 0 3 4
0 3 0 2
0 4 2 0
Enter the source vertex
2
distance of source 2 to 1 is 5
distance of source 2 to 2 is 0
distance of source 2 to 3 is 3
distance of source 2 to 4 is 4
[root@localhost ~]#
```

Explanation for the code:

1. `import java.util.Scanner;`: Imports the Scanner class from the `java.util` package, which is used for reading input from the user.
2. `public class BellmanFord`: Defines a class named `BellmanFord`.
3. `private int D[];`: Declares an integer array `D` to store the distances from the source to each vertex.
4. `private int num_ver;`: Declares an integer variable `num_ver` to store the number of vertices in the graph.
5. `public static final int MAX_VALUE = 999;`: Declares a constant integer variable `MAX_VALUE` and initializes it to 999. It represents the maximum value for distances, used as a placeholder for infinity.
6. `public BellmanFord(int num_ver)`: Constructor for the `BellmanFord` class, which takes the number of vertices as an argument and initializes the `num_ver` and `D` array.
7. `public void BellmanFordEvaluation(int source, int A[][])`: Method to perform the Bellman-Ford algorithm. It takes the source vertex and the adjacency matrix `A` as parameters.

8. `for (int node = 1; node <= num_ver; node++)`: Initializes the distance array ``D`` with maximum values for all vertices.
9. `D[source] = 0;`: Sets the distance from the source vertex to itself to 0.
10. `for (int node = 1; node <= num_ver - 1; node++)`: Outer loop for the relaxation process, iterating for ``num_ver - 1`` times.
11. `for (int sn = 1; sn <= num_ver; sn++)`: Inner loop for the source vertex.
12. `for (int dn = 1; dn <= num_ver; dn++)`: Innermost loop for the destination vertex.
13. `if (A[sn][dn] != MAX_VALUE)`: Checks if there is an edge between the source and destination.
14. `if (D[dn] > D[sn] + A[sn][dn])`: Checks if the distance to the destination can be reduced by going through the source.
15. `D[dn] = D[sn] + A[sn][dn];`: Updates the distance if a shorter path is found.
16. The next block of code (lines 19-27) checks for negative edge cycles in the graph.
17. `for (int vertex = 1; vertex <= num_ver; vertex++)`: Prints the distances from the source to all vertices after the Bellman-Ford algorithm is executed.
18. `public static void main(String[] args)`: The main method where the execution of the program starts.
19. `int num_ver = 0;`: Initializes the variable ``num_ver``.
20. `int source;`: Declares the variable ``source`` to store the source vertex.
- 21-25. Takes input from the user for the number of vertices and the adjacency matrix.
26. `source = scanner.nextInt();`: Takes input for the source vertex.
27. Creates an instance of the ``BellmanFord`` class and calls the ``BellmanFordEvaluation`` method with the source and adjacency matrix as parameters.
28. `scanner.close();`: Closes the Scanner to release resources.