

6. Write a program to find the shortest path between vertices using the Bellman-Ford and path vector routing algorithm

```
import java.util.Scanner;

public class P6 {

    static final int MAX = 999; // Represents infinity (no path)

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

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

        int n = sc.nextInt();

        // Initialize adjacency matrix (1-based indexing)

        int[][] g = new int[n + 1][n + 1];

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

        for (int i = 1; i <= n; i++)

            for (int j = 1; j <= n; j++) {

                int w = sc.nextInt();

                // If diagonal element, distance = 0; else if no edge (0) set to MAX (infinity)

                g[i][j] = (i == j) ? 0 : (w == 0 ? MAX : w);

            }

        System.out.println("Enter the source vertex");

        int src = sc.nextInt();
```

```

int[] dist = new int[n + 1]; // Stores shortest distance estimates

int[] pred = new int[n + 1]; // Stores predecessor of each vertex for path reconstruction

// Initialize distances to MAX and predecessors to -1 (undefined)

for (int i = 1; i <= n; i++) {

    dist[i] = MAX;

    pred[i] = -1;

}

dist[src] = 0; // Distance to source is zero


// Relax edges (n-1) times to find shortest paths

for (int k = 1; k < n; k++)

    for (int u = 1; u <= n; u++)

        for (int v = 1; v <= n; v++)

            if (g[u][v] != MAX && dist[u] != MAX && dist[v] > dist[u] + g[u][v]) {

                dist[v] = dist[u] + g[u][v]; // Update distance

                pred[v] = u; // Update predecessor

            }


// Check for negative weight cycles

for (int u = 1; u <= n; u++)

    for (int v = 1; v <= n; v++)

        if (g[u][v] != MAX && dist[u] != MAX && dist[v] > dist[u] + g[u][v]) {

            System.out.println("The Graph contains a negative edge cycle");
        }

```

```
sc.close();
```

```
return;
```

```
}
```

```
// Print shortest distances and paths from source
```

```
for (int i = 1; i <= n; i++) {
```

```
System.out.print("Distance from source " + src + " to vertex " + i + " is " + dist[i]);
```

```
if (dist[i] != MAX) {
```

```
System.out.print(" | Path: ");
```

```
printPath(pred, i); // Recursive call to print path
```

```
} else
```

```
System.out.print(" | No path");
```

```
System.out.println();
```

```
}
```

```
sc.close(); // Close scanner
```

```
}
```

```
// Recursive method to print path from source to vertex v using predecessor array
```

```
static void printPath(int[] pred, int v) {
```

```
if (v == -1) return; // Base case: no predecessor
```

```
printPath(pred, pred[v]); // Recursive call for predecessor
```

```
System.out.print(v + " "); // Print current vertex
```

```
}
```

}

O/p

Enter the number of vertices

7

Enter the weighted matrix

0 6 5 5 0 0 0

0 0 0 0 -1 0 0

0 -2 0 0 1 0 0

0 0 -2 0 0 -1 0

0 0 0 0 0 0 3

0 0 0 0 0 0 3

0 0 0 0 0 0 0

Enter the source vertex

1

Distance from source 1 to vertex 1 is 0 | Path: 1

Distance from source 1 to vertex 2 is 1 | Path: 1 4 3 2

Distance from source 1 to vertex 3 is 3 | Path: 1 4 3

Distance from source 1 to vertex 4 is 5 | Path: 1 4

Distance from source 1 to vertex 5 is 0 | Path: 1 4 3 2 5

Distance from source 1 to vertex 6 is 4 | Path: 1 4 6

Distance from source 1 to vertex 7 is 3 | Path: 1 4 3 2 5 7

Enter the number of vertices

3

Enter the weighted matrix

1 1 1

1 0 -1

0 1 1

Enter the source vertex

1

Distance from source 1 to vertex 1 is 0 | Path: 1

Distance from source 1 to vertex 2 is 1 | Path: 1 2

Distance from source 1 to vertex 3 is 0 | Path: 1 2 3