```java
import java.util.Collections;

import java.util.Comparator;

import java.util.LinkedList;

import java.util.List;

import java.util.Scanner;

import java.util.Stack;


public class KruskalsAlgorithm{

   private List<Edge> edges;                                   //define a generic of type edge

   private int numberOfVertices;                               //define number of vertices
public static final int MAX_VALUE = 999;                       //define initial weight of edges

private int visited[];                                         //array to store visited vertices

private int spanning_tree[][];                                 //array to store vertices of MST

//method for initializing number of vertices

   public KruskalsAlgorithm(int numberOfVertices) {

      this.numberOfVertices = numberOfVertices;

      edges = new LinkedList<Edge>();

      visited = new int[this.numberOfVertices + 1];

      spanning_tree = new int[numberOfVertices + 1][numberOfVertices + 1];

   }


   //method for implementing

   public void kruskalAlgorithm(int adjacencyMatrix[][])  {

      boolean finished = false;


     for (int source = 1; source <= numberOfVertices; source++)  {

         for (int destination = 1; destination <= numberOfVertices; destination++)

         {

            if (adjacencyMatrix[source][destination] != MAX_VALUE && source != destination)

            {

               Edge edge = new Edge();

               edge.sourcevertex = source;

               edge.destinationvertex = destination;

               edge.weight = adjacencyMatrix[source][destination];

               adjacencyMatrix[destination][source] = MAX_VALUE;
```

```
            edges.add(edge);
        }
    }
}

Collections.sort(edges, new EdgeComparator());
CheckCycle checkCycle = new CheckCycle();
for (Edge edge : edges)
{
    spanning_tree[edge.sourcevertex][edge.destinationvertex] = edge.weight;
    spanning_tree[edge.destinationvertex][edge.sourcevertex] = edge.weight;
    if (checkCycle.checkCycle(spanning_tree, edge.sourcevertex))
    {
        spanning_tree[edge.sourcevertex][edge.destinationvertex] = 0;
        spanning_tree[edge.destinationvertex][edge.sourcevertex] = 0;
        edge.weight = -1;
        continue;
    }
    visited[edge.sourcevertex] = 1;
    visited[edge.destinationvertex] = 1;
    for (int i = 0; i < visited.length; i++)
    {
        if (visited[i] == 0)
        {
            finished = false;
            break;
        } else
        {
            finished = true;
        }
    }
    if (finished)
        break;
}

System.out.println("The spanning tree is ");
```

```
        for (int i = 1; i <= numberOfVertices; i++)
            System.out.print("\t" + i);
        System.out.println();
        for (int source = 1; source <= numberOfVertices; source++)
        {
            System.out.print(source + "\t");
            for (int destination = 1; destination <= numberOfVertices; destination++)
            {
                System.out.print(spanning_tree[source][destination] + "\t");
            }
            System.out.println();
        }
    }

    public static void main(String... arg)
    {
        int adjacency_matrix[][];
        int number_of_vertices;

        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the number of vertices");
        number_of_vertices = scan.nextInt();
        adjacency_matrix = new int[number_of_vertices + 1][number_of_vertices + 1];

        System.out.println("Enter the Weighted Matrix for the graph");
        for (int i = 1; i <= number_of_vertices; i++)
        {
            for (int j = 1; j <= number_of_vertices; j++)
            {
                adjacency_matrix[i][j] = scan.nextInt();
                if (i == j)
                {
                    adjacency_matrix[i][j] = 0;
                    continue;
                }
                if (adjacency_matrix[i][j] == 0)
```

```java
                {
                    adjacency_matrix[i][j] = MAX_VALUE;
                }
            }
        }
        KruskalsAlgorithm kruskalAlgorithm = new KruskalsAlgorithm(number_of_vertices);
        kruskalAlgorithm.kruskalAlgorithm(adjacency_matrix);
        scan.close();
    }
}

class Edge
{
    int sourcevertex;
    int destinationvertex;
    int weight;
}

class EdgeComparator implements Comparator<Edge>
{
    @Override
    public int compare(Edge edge1, Edge edge2)
    {
        if (edge1.weight < edge2.weight)
            return -1;
        if (edge1.weight > edge2.weight)
            return 1;
        return 0;
    }
}


class CheckCycle
{
    private Stack<Integer> stack;
    private int adjacencyMatrix[][];
```

```
public CheckCycle()
{
    stack = new Stack<Integer>();
}

//method for avoid making loops
public boolean checkCycle(int adjacency_matrix[][], int source)
{
    boolean cyclepresent = false;
    int number_of_nodes = adjacency_matrix[source].length - 1;

    adjacencyMatrix = new int[number_of_nodes + 1][number_of_nodes + 1];
    for (int sourcevertex = 1; sourcevertex <= number_of_nodes; sourcevertex++)
    {
        for (int destinationvertex = 1; destinationvertex <= number_of_nodes; destinationvertex++)
        {
            adjacencyMatrix[sourcevertex][destinationvertex] = adjacency_matrix[sourcevertex][destinationvertex];
        }
     }

     int visited[] = new int[number_of_nodes + 1];
     int element = source;
     int i = source;
     visited[source] = 1;
     stack.push(source);

     while (!stack.isEmpty())
     {
         element = stack.peek();
         i = element;
         while (i <= number_of_nodes)
         {
             if (adjacencyMatrix[element][i] >= 1 && visited[i] == 1)
             {
```

```
                if (stack.contains(i))
                {
                    cyclepresent = true;
                    return cyclepresent;
                }
            }
            if (adjacencyMatrix[element][i] >= 1 && visited[i] == 0)
            {
                stack.push(i);
                visited[i] = 1;
                adjacencyMatrix[element][i] = 0;
                adjacencyMatrix[i][element] = 0;
                element = i;
                i = 1;
                continue;
             }
             i++;
        }
        stack.pop();
    }
    return cyclepresent;
  }
}
```