CO322 - LAB 08

GRAPHS

WIJERATHNE E.S.G.

E/18/397

SEMESTER 05

22/01/2023

Task 1

Code

```java
import java.util.*;

public class task1{

    public  boolean isBipartite(int[][] graph) {
    int n = graph.length; // number of nodes in the graph
    int colors[] = new int[n]; // array to store the color of each node

    //assigning values to the color array as -1
    for (int i = 0; i < n; i++){
        colors[i] = -1;
    }


    Arrays.fill(colors, -1);
    // initially all nodes are uncolored

    for (int start = 0; start < n; start++) {
        // iterate through all uncolored nodes

        if (colors[start] == -1) {
            // if the node is uncolored

            Queue<Integer> queue = new LinkedList<>();
            // use a queue for BFS
            queue.add(start);
            // add the node to the queue
            colors[start] = 0;
            // assign color 0 to the node

            while (!queue.isEmpty()) {
                int node = queue.poll();
                for (int neighbor = 0; neighbor < n; neighbor++) {
                    if (graph[node][neighbor] == 1 && colors[neighbor] == -1) {
                        // if the neighbor is uncolored
                        queue.add(neighbor);
                        // add the neighbor to the queue
                        colors[neighbor] = 1 - colors[node];
                        // assign the opposite color of the current node to the neighbor
                    } else if (graph[node][neighbor] == 1 && colors[neighbor] == colors[node]) {
                        return false;
                        // if the neighbor is colored and has the same color as the current node, the
graph is not bipartite
                    }
                }
            }
        }
    }
    return true; // if all nodes are visited and colored without conflicts, the graph is bipartite
}
    public static void main(String args[]){

        int[][] bipartiteGraph   = {{0, 1, 0, 1}, {1, 0, 1, 0}, {0, 1, 0, 1}, {1, 0, 1, 0}};
        int[][] nonBipartiteGraph = {{0, 1, 0, 1}, {1, 0, 1, 1}, {0, 1, 0, 1}, {1, 1, 1, 0}};

        task1 a = new task1();
        task1 b = new task1();


        System.out.println(a.isBipartite(bipartiteGraph)); // prints "true"
        System.out.println(b.isBipartite(nonBipartiteGraph)); // prints "false"


    }
}
```

Output

```
PS D:\Semester 5\CO322 - Data Structures and Algorithm\lab\lab 8 - graphs\Task 1>  d:; cd 'd:\Semester 5\CO322 - Data Structures and Algorithm\lab\lab 8 - graphs\Task 1'; & 'C:\Program Files\Eclipse Adoptium\jdk-17.0.4.101-hotspot\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\J-A-R-V-I-S\AppData\Roaming\Code\User\workspaceStorage\31265575628416cd05e7d16f80a5ec75\redhat.java\jdt_ws\Task 1_b40171e8\bin' 'task1'
true
false
PS D:\Semester 5\CO322 - Data Structures and Algorithm\lab\lab 8 - graphs\Task 1>
```

Task 2

Code

```java
import java.util.ArrayList;
import java.util.List;

public class TransitiveClosure {
    private int V; // number of vertices
    private boolean[][] tc; // transitive closure matrix
    private List<Integer>[] adj; // adjacency list representation of the graph

    public TransitiveClosure(int v) {
        V = v;
        tc = new boolean[V][V];
        adj = new ArrayList[V];
        for (int i = 0; i < V; i++) {
            adj[i] = new ArrayList<>();
        }
    }

    // function to add an edge to the graph
    public void addEdge(int v, int w) {
        adj[v].add(w);
    }

    // DFS traversal of the vertices reachable from v
    private void DFS(int v, int u) {
        tc[v][u] = true;
        for (int i = 0; i < adj[u].size(); i++) {
            if (!tc[v][adj[u].get(i)]) {
                DFS(v, adj[u].get(i));
            }
        }
    }

    // function to get transitive closure
    public void getTransitiveClosure() {
        for (int i = 0; i < V; i++) {
            DFS(i, i);
        }
    }

    // prints the transitive closure matrix
    public void printTC() {
        System.out.println("Transitive closure matrix is: ");
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                System.out.print(tc[i][j] + " ");
            }
            System.out.println();
        }
    }


    public static void main(String args[]){

       TransitiveClosure g = new TransitiveClosure(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.getTransitiveClosure();
        g.printTC();

    }
}
```

Output

```
PS D:\Semester 5\CO322 - Da>  d:; cd 'd:\Semester 5\CO322 - Data Structures and Algorith
m\lab\lab 8 - graphs\Task 1'; & 'C:\Program Files\Eclipse Adoptium\jdk-17.0.4.101-hotspo
t\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\J-A-R-V-I-S\Ap
pData\Roaming\Code\User\workspaceStorage\31265575628416cd05e7d16f80a5ec75\redhat.java\jd
t_ws\Task 1_b40171e8\bin' 'TransitiveClosure'
Transitive closure matrix is:
true true true true
true true true true
true true true true
false false false true
PS D:\Semester 5\CO322 - Data Structures and Algorithm\lab\lab 8 - graphs\Task 1>
```

Task 3

Code

Node class

```java
import java.util.Comparator;

class Node implements Comparator<Node> {

    // Member variables of this class
    public int node;
    public int cost;

    // Constructors of this class

    // Constructor 1
    public Node() {
    }

    // Constructor 2
    public Node(int node, int cost) {

        // This keyword refers to current instance itself
        this.node = node;
        this.cost = cost;
    }

    // Method 1
    @Override
    public int compare(Node node1, Node node2) {

        if (node1.cost < node2.cost)
            return -1;

        if (node1.cost > node2.cost)
            return 1;

        return 0;
    }
}
```

Task 3 Algorithm

```java
import java.util.*;


public class DPQ {

    private int dist[];                 // stores the total distance between the node and the
source(total cost in this case)
    private Set<Integer> settled;       // minimum distance(cost) is calculated for these nodes
    private PriorityQueue<Node> pq;     //priority queue for the nodes


    private int V;                      // Number of vertices

    List<List<Node>> adj;               // store the path as a list of lists

    // Constructor
    public DPQ(int V) {
        //initializing member variables
        this.V = V;
        dist = new int[V];
        settled = new HashSet<Integer>();
        pq = new PriorityQueue<Node>(V, new Node());
    }

    //This method implements the Dijkstas algorithm
    //THis method updates the totalDistance(total cost) for the adjacent nodes of a given node
    public void dijkstra(List<List<Node>> adj, int src) {
        this.adj = adj;

        //at start, assigned distance as a max value
        for (int i = 0; i < V; i++)
            dist[i] = Integer.MAX_VALUE;

        // node to the priority queue
        pq.add(new Node(src, 0));

        // initially distance is 0
        dist[src] = 0;

        while (settled.size() != V) {

            //terminated when an empty queue
            if (pq.isEmpty())
                return;

            //remove the minimum distance(cost) node from the priority queue
            int u = pq.remove().node;

            //checking if the node is already in the setteled set
            if (settled.contains(u))
                continue;       //if it is continue to the next iteration

            //otherwise add to the settled list
            settled.add(u);

            //call the neighbours method
            e_Neighbours(u);
        }
    }

    // helper method for dijkstras algorithm
    // updates the total distance for the neighbours of a node
    private void e_Neighbours(int u) {

        int edgeDistance = -1;
        int newDistance = -1;

        // for all the nodes that are connected to the node u
        for (int i = 0; i < adj.get(u).size(); i++) {

            //getting the neighbour node of u
            Node v = adj.get(u).get(i);

            //Check if the node is already proceessed , if not...
            if (!settled.contains(v.node)) {

                //calculating the distance(cost) from the current node
                edgeDistance = v.cost;
                newDistance = dist[u] + edgeDistance;

                //if the distance is lower than previous value,
                //update that as the new distance by updating the array
                if (newDistance < dist[v.node])
                    dist[v.node] = newDistance;

                //priority queue is updated with the current node
                pq.add(new Node(v.node, dist[v.node]));
            }
        }
    }
}
```

Main part to check the code.

```java
public static void main(String arg[]) {

        //create the adjacency list
        int V = 12;
        List<List<Node>> adj = new ArrayList<List<Node>>();

        // assigend the list for each and every node
        for (int i = 0; i < V; i++) {
            List<Node> item = new ArrayList<Node>();
            adj.add(item);
        }

        // initializing the graph
        adj.get(0).add(new Node(2, 1));
        adj.get(0).add(new Node(8, 5));

        adj.get(1).add(new Node(3, 6));

        adj.get(2).add(new Node(5, 10));
        adj.get(2).add(new Node(11, 2));
        adj.get(2).add(new Node(10, 7));
        adj.get(2).add(new Node(3, 11));

        adj.get(3).add(new Node(4, 9));
        adj.get(3).add(new Node(9, 4));

        adj.get(4).add(new Node(1, 10));
        adj.get(4).add(new Node(5, 4));
        adj.get(4).add(new Node(7, 1));

        adj.get(5).add(new Node(0, 3));
        adj.get(5).add(new Node(7, 7));

        adj.get(6).add(new Node(4, 2));

        adj.get(7).add(new Node(6, 12));

        adj.get(8).add(new Node(7, 7));

        adj.get(9).add(new Node(1, 12));

        adj.get(10).add(new Node(3, 5));

        // Calculating the single source the shortest path
        DPQ dpq0 = new DPQ(V);
        dpq0.dijkstra(adj, 0);

        DPQ dpq1 = new DPQ(V);
        dpq1.dijkstra(adj, 1);

        for (int i = 2; i < dpq0.dist.length; i++){
            if(dpq0.dist[i] <= dpq1.dist[i])
                System.out.println("Shortest distance from S0 to D"+i+" = "+dpq0.dist[i]);
            else
                System.out.println("Shortest distance from S1 to D"+i+" = "+dpq1.dist[i]);

        }
    }
```

Output

```
Shortest distance from S0 to D2 = 1
Shortest distance from S1 to D3 = 6
Shortest distance from S1 to D4 = 15
Shortest distance from S0 to D5 = 11
Shortest distance from S0 to D6 = 24
Shortest distance from S0 to D7 = 12
Shortest distance from S0 to D8 = 5
Shortest distance from S1 to D9 = 10
Shortest distance from S0 to D10 = 8
Shortest distance from S0 to D11 = 3
```