package bcimdad;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Map;

import java.util.Set;

import java.util.HashSet;

import java.io.FileReader;

import java.io.InputStreamReader;

import java.io.BufferedReader;

import java.io.IOException;

import java.util.StringTokenizer;

/\*\*

\* The <tt>BreadthFirstPaths</tt> class represents a data type for finding

\* shortest paths (number of edges) from a source vertex <em>s</em>

\* (or a set of source vertices)

\* to every other vertex in an undirected graph.

\* <p>

\* This implementation uses breadth-first search.

\* The constructor takes time proportional to <em>V</em> + <em>E</em>,

\* where <em>V</em> is the number of vertices and <em>E</em> is the number of edges.

\* It uses extra space (not including the graph) proportional to <em>V</em>.

\* <p>

\* For additional documentation, see <a href="/algs4/41graph">Section 4.1</a> of

\* <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.

\*

\* @author Robert Sedgewick

\* @author Kevin Wayne

\*/

public class BreadthFirstPaths {

private static final int INFINITY = Integer.MAX\_VALUE;

private boolean[] marked; // marked[t] = is there an s-t path

private int[] pathCount; // pathCount[t] = how many shortest paths s-t

private Map<Integer, ArrayList<Integer>> edgeTo; // edgeTo.get(x) = previous edges on shortest s-x path

private int[] distTo; // distTo[t] = number of edges shortest s-t path

private Map<Integer, ArrayList<ArrayList<Integer>>> paths; // paths.get(t) = all shortest paths s-t

private int[][] pathsThru; // pathsThruV[v] = num shortest paths s-t through v

/\*\*

\* Computes the shortest path between the source vertex <tt>s</tt>

\* and every other vertex in the graph <tt>G</tt>.

\* @param G the graph

\* @param s the source vertex

\*/

public BreadthFirstPaths(Graph G, int s) {

marked = new boolean[G.V()];

pathCount = new int[G.V()];

distTo = new int[G.V()];

edgeTo = new HashMap<>(G.V());

paths = new HashMap<>(G.V());

pathsThru = new int[G.V()][G.V()];

bfs(G, s);

assert check(G, s);

}

// breadth-first search from a single source

private void bfs(Graph G, int s) {

Queue<Integer> q = new Queue<Integer>();

for (int v = 0; v < G.V(); v++) distTo[v] = INFINITY;

distTo[s] = 0;

marked[s] = true;

q.enqueue(s);

while (!q.isEmpty()) {

int v = q.dequeue();

for (int w : G.adj(v)) {

if (!marked[w]) {

if(edgeTo.get(w) == null) {

edgeTo.put(w, new ArrayList<Integer>());

}

edgeTo.get(w).add(v);

distTo[w] = distTo[v] + 1;

marked[w] = true;

pathCount[w]++;

q.enqueue(w);

} else if (distTo[w] == distTo[v] + 1) {

pathCount[w]++;

edgeTo.get(w).add(v);

}

}

}

backTrackPaths(G, s);

for(int v = 0; v < G.V(); v++)

backTrackPathsThruV(G, s, v);

}

private void backTrackPathsThruV(Graph G, int s, int v) {

for (int t = 0; t < G.V(); t++) {

if (!marked[t]) continue;

if (t==s) continue;

if (t==v) continue; // TODO: Should we count paths that end at v? Currently do not.

for(ArrayList<Integer> p : paths.get(t)) {

for(int x : p) {

if (x==v && x!=s) {

pathsThru[t][v]++;

}

}

}

}

}

private void backTrackPaths(Graph G, int s) {

for (int v = 0; v < G.V(); v++) {

if (!marked[v]) continue;

if (v==s) continue;

paths.put(v, new ArrayList<ArrayList<Integer>>());

ArrayList<Integer> path = new ArrayList<>();

paths.get(v).add(path);

spawnSubPath(v, v, path);

}

}

private void spawnSubPath(int origin, int current, ArrayList<Integer> path) {

if (distTo[current] != 0) {

path.add(current);

for (int p : edgeTo.get(current)) {

ArrayList<Integer> copyPath = new ArrayList(path);

paths.get(origin).add(copyPath);

spawnSubPath(origin, p, copyPath);

}

paths.get(origin).remove(path);

} else {

path.add(current);

}

}

public Map<Integer, ArrayList<ArrayList<Integer>>> getPaths() {

return paths;

}

/\*\*

\* Is there a path between the source vertex <tt>s</tt> (or sources) and vertex <tt>v</tt>?

\* @param v the vertex

\* @return <tt>true</tt> if there is a path, and <tt>false</tt> otherwise

\*/

public boolean hasPathTo(int v) {

return marked[v];

}

public int numPathsTo(int v) {

return pathCount[v];

}

public int pathsThru(int t, int v) {

return pathsThru[t][v];

}

/\*\*

\* Returns the number of edges in a shortest path between the source vertex <tt>s</tt>

\* (or sources) and vertex <tt>v</tt>?

\* @param v the vertex

\* @return the number of edges in a shortest path

\*/

public int distTo(int v) {

return distTo[v];

}

/\*\*

\* Returns a shortest path between the source vertex <tt>s</tt> (or sources)

\* and <tt>v</tt>, or <tt>null</tt> if no such path.

\* @param v the vertex

\* @return the sequence of vertices on a shortest path, as an Iterable

\*/

public Iterable<Integer> pathTo(int v) {

if (!hasPathTo(v)) return null;

Stack<Integer> path = new Stack<Integer>();

int x;

for (x = v; distTo[x] != 0; x = edgeTo.get(x).get(0))

path.push(x);

path.push(x);

return path;

}

// check optimality conditions for single source

private boolean check(Graph G, int s) {

// check that the distance of s = 0

if (distTo[s] != 0) {

System.out.println("distance of source " + s + " to itself = " + distTo[s]);

return false;

}

// check that for each edge v-w dist[w] <= dist[v] + 1

// provided v is reachable from s

for (int v = 0; v < G.V(); v++) {

for (int w : G.adj(v)) {

if (hasPathTo(v) != hasPathTo(w)) {

System.out.println("edge " + v + "-" + w);

System.out.println("hasPathTo(" + v + ") = " + hasPathTo(v));

System.out.println("hasPathTo(" + w + ") = " + hasPathTo(w));

return false;

}

if (hasPathTo(v) && (distTo[w] > distTo[v] + 1)) {

System.out.println("edge " + v + "-" + w);

System.out.println("distTo[" + v + "] = " + distTo[v]);

System.out.println("distTo[" + w + "] = " + distTo[w]);

return false;

}

}

}

// check that v = edgeTo[w] satisfies distTo[w] + distTo[v] + 1

// provided v is reachable from s

for (int w = 0; w < G.V(); w++) {

if (!hasPathTo(w) || w == s) continue;

for (int v : edgeTo.get(w))

if (distTo[w] != distTo[v] + 1) {

System.out.println("shortest path edge " + v + "-" + w);

System.out.println("distTo[" + v + "] = " + distTo[v]);

System.out.println("distTo[" + w + "] = " + distTo[w]);

return false;

}

}

return true;

}

private static void doBFS(Graph G, int s, double [] BC) {

BreadthFirstPaths bfs = new BreadthFirstPaths(G, s);

System.out.println(s + " (t:v:paths-through-V:num-shortest-paths): ");

for(int v = 0; v < G.V(); v++) {

for (int t = 0; t < G.V(); t++) {

System.out.printf("%d:%d:%d:%d\t", t, v, bfs.pathsThru(t, v), bfs.numPathsTo(t));

}

System.out.println("");

}

System.out.println("");

for (int v = s+1; v < G.V(); v++) {

if (bfs.hasPathTo(v)) {

System.out.printf("%d to %d: Len %d, Num: %d ", s, v, bfs.distTo(v), bfs.numPathsTo(v));

for (ArrayList<Integer> p : bfs.getPaths().get(v)) {

for(int x : p) {

if (x == s) System.out.print(x);

else System.out.print(x + "-");

}

System.out.print(", ");

}

System.out.println();

}

else {

System.out.printf("%d to %d (-): not connected\n", s, v);

}

}

for (int v = 0; v < G.V(); v++) {

for (int t = 0; t < G.V(); t++) {

if(bfs.numPathsTo(t) != 0) {

BC[v] += ((double) bfs.pathsThru(t, v)) / ((double) bfs.numPathsTo(t));

}

}

}

}

}