Student name: Quan Hong Doan

Student ID: 986956

Assignment: Lab 13 & 14

Week: 13

Due: Jan 13, 2020

**Lab 13**

**Problem 1**:

In this lab you will develop your graph package of software. Based on direction from the slides, finish the implementation of the operations on a Graph:

boolean areAdjacent(Vertex u, Vertex v)

List getListOfAdjacentVerts (Vertex u)

Graph getSpanningTree()

List getConnectedComponents()

boolean isConnected()

boolean hasPathBetween(Vertex u, Vertex v)

boolean containsCycle()

boolean isTree()

boolean isBipartite()

DFS and the spanning tree algorithm have already been implemented. You will need to use observations given in the slides to provide the connected components of the graph, determine whether the graph has a cycle, and to determine if there is a path joining two given vertices.

You will also implement BFS, and, in a subclass, implement the additional work needed to determine if the graph has an odd cycle (so you can determine whether it is a bipartite graph).

Finally, I have provided a second constructor in Graph that accepts an array of Edges (in the form of Objects). One use for this constructor is that it allows you to return a spanning tree as a Graph object after performing your spanning tree algorithm.

The toString method that is provided may not be suitable – you should modify it as necessary so that you can display test results in a useful way.

**Answer**:

**Connected Components**:

Basically, we will modify the additionalProcess() with:

* Remove the visited vertices from the current vertices
* Reset the visited vertices hashmap, and the vertex iterator
* Create a new graph from the visited vertices and edges.
* Reduce the number of vertices

Following is the implementation, not fully optimized, but it follows the above algorithm.

**protected** **void** additionalProcessing() {

List<Edge> lstEdge = **new** ArrayList<Edge>();

**for** (Iterator<Entry<Vertex, Vertex>> it = visitedVertices.entrySet().iterator(); it.hasNext();) {

Map.Entry en = it.next();

**for** (Edge ed : edges) {

**if** (ed.belongTo((Vertex) en.getKey())) {

lstEdge.add(ed);

}

}

vertices.remove(en.getKey());

}

Graph gh = **new** Graph(lstEdge.toArray());

listGraph.add(gh);

numVertices = numVertices - visitedVertices.size();

resetVisitedVertices();

resetVertexIterator();

}

In the ConnectedComponentSearch class:

**public** List<Graph> computeConnectedComponent() {

start();

**return** listGraph;

}

**ExistPathBetween**:

We will utilize the DFS, and just add some conditions to check if we can reach from vertex u to vertex v. The implementation below.

**public** **void** findPath(Vertex u, Vertex v) {

stack.push(u);

visitedVertices.put(u, u);

**while** (!stack.isEmpty()) {

Vertex sv = nextUnvisitedAdjacent(stack.peek());

**if** (sv == **null**) {

stack.pop();

} **else** {

setHasBeenVisited(sv);

stack.push(sv);

**if** (sv.equals(v)) {

**break**;

}

}

}

path = stack.subList(0, stack.size());

}

and Graph class:

**public** **boolean** existsPathBetween(Vertex u, Vertex v) {

FindPath fp = **new** FindPath(**this**);

fp.findPathBetween(u, v);

**this**.path = fp.path;

**return** fp.path.size() > 0;

}

**Check if the graph is connected**:

We also base on the ConnectedComponent (extending from DFS) implementation. If the connectedComponent list has size not equal to 1, we don’t have connected.

**public** **boolean** isConnected() {

**if** (connectedComponents != **null**)

**if** (connectedComponents.size() == 1)

**return** **true**;

**return** **false**;

}

**BFS**: (bfsList is the array list of visited vertices in BFS order)

**protected** **void** bfs() {

Vertex s = nextUnvisited();

**if** (**null** != s) {

setVisited(s);

queue.add(s);

**while** (!queue.isEmpty()) {

Vertex v = queue.poll();

bfsList.add(v);

List<Vertex> listAdj = graph.getListOfAdjacentVerts(v);

Iterator<Vertex> it = listAdj.iterator();

**while** (it.hasNext()) {

Vertex u = it.next();

**if** (!checkVisited(u)) {

setVisited(u);

queue.add(u);

}

}

}

}

}

**Odd Cycle**:

The algorithm bases on the BFS, and we need to check if any vertex of the adjacent vertices of the checking vertex is visited. If visited and not the one from the queue, then we have a loop here, that means we have cycle. During this check, we add counter, so if the counter is odd, we have odd cycle too. The modification from the above bfs() and affected at the checking visited condition.

**if** (!checkVisited(u)) {

setVisited(u);

queue.add(u);

List<Vertex> subList = graph.getListOfAdjacentVerts(u);

**int** counter = 0;

cnt++;

**for** (Vertex sv : subList) {

**if** (checkVisited(sv))

counter++;

**if** (counter % 2 == 0) {

cycle = **true**;

**if** (cnt % 2 != 0) {

oddCycle = **true**;

oddCycleLength = cnt;

}

cnt = 0;

**break**;

}

}

}

**isBipartie**:

If there’s odd cycle, then no bipartie. Therefore, from the above implementation:

**public** **boolean** isBipartie() {

start();

**if** (cycle && !oddCycle)

**return** **true**;

**return** **false**;

}

**Lab 14**

**Problem 1**:

Must every dense graph be connected? Prove your answer.

**Answer**:

We know that an undirected simple graph, the graph density is

A dense graph has number of edges closing to the maximal number of edges.

We also know that the maximal number of edges for an undirected graph is |V|(|V|-1)/2. In other words, the maximal density is 1. Therefore the dense graph will have its density close to 1.

From the above we can see that the number of edges for a dense graph is close to |V|(|V|-1)/2, which is larger than or equal to v – 1.

If v = 2, then v(v-1)/2 = 2(2-1)/2 = 1 = v - 1

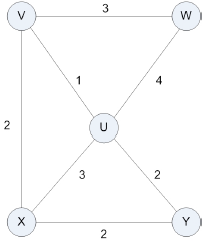
If v = 3, then v(v-1)/2 = 3(3-1)/2 = 3 > 2 = v – 1

Because v is positive integer, therefore v\*(v-1)/2 > v – 1

We also know a graph with minimum v – 1 edges is connected. Therefore, a dense graph is connected.

**Problem 2**:

Carry out the steps of Dijkstra's algorithm to compute the length of the shortest path between vertex V and vertex Y in the graph I gave in class (reproduced below). Display the evolution of the values for D[] in a table.



**Answer**:

Starting from vertex V, our destination is vertex Y, we will go through all vertices, comparing the cost to those adjacent vertices, and pick only the smallest cost. If we don’t reach vertex Y, we come back to the parent, repeat the loop. Therefore the loop can be presented as the table below.

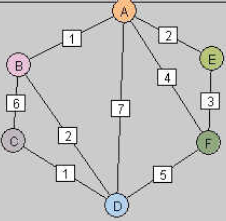
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Iteration** | **Unsettled** | **Settled** | **Evaluation vertex** | **v** | **u** | **x** | **w** | **y** |
| 1 | v | - | v | 0 | v-1 | v-2 | v-3 | - |
| 2 | u, x, w | v | u | 0 | v-1 | u-3 | u-4 | u-2 |
| 3 | y | v, u | y | 0 | v-1 | - | - | u-2 |
| **Final** | - | v, u, y | - | 0 | v-1 | - | - | u-2 |

The shortest path is:

Vertex V 🡪 (cost: 1) 🡪 vertex U 🡪 (cost: 2) 🡪 vertex Y

**Problem 3**:

Carry out the steps of Kruskal's algorithm to compute a minimum spanning tree for the graph shown below. Express the tree as a set of edges, and display the evolution of clusters in a table.



**Answer**:

First, we will create a table with all edges along with their weights.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Edge** | AB | CD | AE | BD | EF | AF | DF | BC | AD |
| **Weight** | 1 | 1 | 2 | 2 | 3 | 4 | 5 | 6 | 7 |

Now we will start from the smallest edge, and keep selecting edges which does not create any cycle with the previous selected edges.

Our selection will be:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Edge** | AB | CD | AE | BD | EF | AF | DF | BC | AD |
| **Weight** | 1 | 1 | 2 | 2 | 3 | 4 | 5 | 6 | 7 |
| **Selection iterations** | | | | | | | | | |
| Loop 1 | yes |  |  |  |  |  |  |  |  |
| Loop 2 |  | yes |  |  |  |  |  |  |  |
| Loop 3 |  |  | yes |  |  |  |  |  |  |
| Loop 4 |  |  |  | yes |  |  |  |  |  |
| Loop 5 |  |  |  |  | yes |  |  |  |  |
| Loop 6 |  |  |  |  |  | No, cycle AEFA |  |  |  |
| Loop 7 |  |  |  |  |  |  | No,  cycle ABDFEA |  |  |
| Loop 8 |  |  |  |  |  |  |  | No,  cycle BCDB |  |
| Loop 9 |  |  |  |  |  |  |  |  | No, cycle ABD and ADF |

So our MST is AB, CD, AE, BD, EF

Cost is 9.