**Question 1 (20 marks)**

Let edge (a,b) belongs to the minimum spanning tree.

Firstly, calculate the minimum spanning tree. The time complexity of minimum spanning tree is O (E + V \* log (V)) by using Prim’s algorithm with Fibonacci heap.

The time complexity of initial and insert is O(V) which means that just initial and insert number into code, time complexity of Extract-Min is O (V log V) which means that when number after insert into code need to examine each of input is minimum priority or not and delete, then stay and for next code, and the time complexity of Decrease-Priority is O (E\*1). So, the overall running time for minimum spanning tree is O(V^2).

Secondly, using the minimum spanning tree, to find the shortest path eg. Max(a,b) by the Breadth-first search method. Only when satisfy both queues are an empty set and all the vertexes belong to Graph’s vertexes within adjacent c, then the whole Breadth-first search can be accomplished; this will become nested loop.

Inside the loop to find the max(a,b) ---- shortest length of path. In here, the time complexity is O (V^2) for nested loop due to there is a nested loop, and each loop will take O(V).

Thirdly, in the nested loop, there exists a double condition, if-else structure, to examine the a and b is zero or not and a is same as b, and to examine weight of max(a,b) – weight(a,b) is minimum. Due to here is a two conditions structure, the time complexity is O (V^2) and each if statement will take O(V).

Fourth, find the edge (a,b) by using the code in the Breadth-first search method, and add edge(a,b) into the minimum spanning tree, and max(a,b) is the shortest path from a to b which will O(1)

The total time is O(V^2)+ O (V^2)+ O (V^2)+ O(1)= O(3V^2+1).And ignore the constant number.

Overall, the total time is O (V^2)

**Question 2**

Bipartite graph is a pair of G(V,E) that is a vertex and edges which is the partite of it’s vertex set V1 U V2 sets that every edge in the graph will join the vertex from one set to through another. As there won’t be any adjacent vertices in V1 and there won’t be any vertices in V2. However, if there two vertices are adjacent then there is one vertex in V1, and another is in V2, So we can find a partition like this of a graph vertex then it is a bipartite.

Let’s check out an example of a bipartite algorithm, Here it is bipartite graph.

a1

a2

a3

b1

b2

a1

b1

a2

b2

a3

So, in the graph we see that the edges are connecting to one another. a1 is connected to the adjacent b1 and b2. Then, a2 is connected to the adjacent b1. Lastly, a3 is connected to the adjacent b2. So, we see that every ‘a’ set is connected to ‘b’. As we could have separated the partition in this way, this here is a bipartite graph.

Now let’s check out the proof that we have to prove. A graph is bipartite if and only if it contains no odd cycle.

Bipartite means vertices that can't be adjacent with each other in set 1 or in set 2. So, Vertices in set 1 is adjacent with set 2, as well as set 2 is adjacent with set 1. So, when there is a Bipartite graph it doesn't contain an odd cycle.

* V(G) it will divide into two sets v1 and v2. So, every adjacent will of G will join in v1 to v2. Also, there shouldn’t be any connectivity between v1 and v1, so as v2 and v2.
* Also, V as vertices is a circle then there will be a cycle from s1 to s2 then back s1 and then s2. By counting all the cycles the adjacent will be even. For example, if we start from ‘e’, then ‘g’ then ‘i’ then the cycle will come back to ‘e’. So when coming back to ‘e’ then it will be an even cycle. Therefore, the length of the cycle is even.

e

g

i

* So, if the cycle is even, then the graph is Bipartite. We will see a connection with no odd cycle.

x

d

e

c

f

a

x

c

a

d

f

e

* Here we see that ‘x’ is not adjacent to ‘c’ and is not adjacent to ‘a’. Same thing ‘d’ is not adjacent to ‘f’ and is not adjacent to ‘e’.
* If G is not properly colour then it, G will be adjacent with the same colour.

u

v

y

y

p2

p1

x

* P1 = { x……y….u}

P2 = {x……y…..v}

So, we see here that there is x and y common in P1 and P2. Later, ‘y’ there are no common vertices. So, ‘y’ is the last vertex.

P1 = d(x,y)

P2 = d(x,y)

If P1 is the shortest distance between x and u, then P2 is also the shortest distance between x and v. Also, P1(x,y) is less than P2(x,y). So, when we contradict P2 will be the shortest distance. As we know that it will either be an even or odd. Therefore, in total it will be even.

* The time complexity of the bipartite is O | E | where V and E are the total number of vertices and edges in the bipartite graph. The time complexity is O(1).

**Question 3**

**Tester.java file**

// Tester class

public class Tester {

public static void main(String[] args) {

testConnected();

testBipartite();

testCycle();

}

// test for connected graph

private static void testConnected() {

// create graph

Graph graph = new AdjacencyListGraph(6);

graph.addEdge(0, 2);

graph.addEdge(0, 1);

graph.addEdge(1, 3);

graph.addEdge(4, 5);

System.out.println("Is Graph connected? " + graph.isConnected());

graph.addEdge(2, 5);

System.out.println("Is Graph connected? " + graph.isConnected());

}

// tests bipartite

private static void testBipartite() {

Graph graph = new AdjacencyListGraph(5);

// no cycle

graph.addEdge(0, 1);

graph.addEdge(1, 2);

graph.addEdge(2, 3);

graph.addEdge(3, 4);

System.out.println("Is Graph bipartite? " + graph.isBipartite());

// odd length cycle

graph.addEdge(4, 0);

System.out.println("Is Graph bipartite? " + graph.isBipartite());

}

// test if grpah has a cycle

private static void testCycle() {

Graph graph = new AdjacencyListGraph(5);

// create edges

graph.addEdge(0, 1);

graph.addEdge(1, 2);

graph.addEdge(2, 3);

graph.addEdge(3, 4);

System.out.println("Has cycle? " + graph.hasACycle());

// create cycle

graph.addEdge(4, 0);

System.out.println("Has cycle? " + graph.hasACycle());

}

}

**Graph\_1.java**

/\*\*

\* This interface encapsulates the undirected graph ADT. This

\* interface is intended for graphs with a constant number of

\* vertices. Vertices are represented by integers between 0 and the

\* number of vertices - 1 inclusive.

\*/

public interface Graph

{

/\*\*

\* Add an adge between between vertex v1 to v2. If a edge already

\* exists between v1 and v2, do nothing.

\*/

public void addEdge(int v1, int v2);

/\*\*

\* Remove the edge between v1 and v2 if it exists.

\*/

public void removeEdge(int v1, int v2);

/\*\*

\* Return true if there is an edge between v1 and v2 and false otherwise.

\*/

public boolean edgeQuery(int v1, int v2);

/\*\*

\* Return true if this graph is bipartite and false otherwise.

\*/

public boolean isBipartite();

/\*\*

\* Return true if this graph is connected and false otherwise.

\*/

public boolean isConnected();

/\*\*

\* Return true if this graph has a Cycle and false otherwise.

\*/

public boolean hasACycle();

}

**AdjacencyListGraph\_1.java**

import java.util.LinkedList;

import java.util.Iterator;

/\*\*

\* This class is an implementation of the Graph interface using an adjacency

\* list to store edges

\*/

public class AdjacencyListGraph implements Graph {

/\*\*

\* The Vertex class is a private class to provide pointers to the array

\* elements.

\*/

private class Vertex {

int index;

LinkedList<Vertex> adjacencyList;

Vertex(int index\_) {

index = index\_;

adjacencyList = new LinkedList<Vertex>();

}

}

// The number of vertices in the graph

int n;

// The place where the edge lists are stored

Vertex[] vertexList;

public AdjacencyListGraph(int numVertices) {

n = numVertices;

vertexList = new Vertex[n];

for (int i = 0; i < n; i++)

vertexList[i] = new Vertex(i);

}

// Add an edge

// Since this graph is undirected we add the edge to both

// adjacency lists

public void addEdge(int v1, int v2) {

vertexList[v1].adjacencyList.add(vertexList[v2]);

vertexList[v2].adjacencyList.add(vertexList[v1]);

}

// Remove an edge

// Since the graph is undirected, we remove the edge from both

// adjacency lists

public void removeEdge(int v1, int v2) {

for (Iterator<Vertex> i = vertexList[v1].adjacencyList.iterator(); i.hasNext();) {

Vertex v = i.next();

if (v.index == v2) {

vertexList[v1].adjacencyList.remove(v);

break;

}

}

for (Iterator<Vertex> i = vertexList[v2].adjacencyList.iterator(); i.hasNext();) {

Vertex v = i.next();

if (v.index == v1) {

vertexList[v2].adjacencyList.remove(v);

break;

}

}

}

/\*\*

\* Test whether there is an edge between v1 and v2

\*/

public boolean edgeQuery(int v1, int v2) {

for (Iterator<Vertex> i = vertexList[v1].adjacencyList.iterator(); i.hasNext();)

if ((i.next()).index == v2)

return true;

return false;

}

/\*\*

\* Test whether this graph is bipartite

\*/

public boolean isBipartite() {

boolean[] visited = new boolean[vertexList.length];

int[] colors = new int[vertexList.length];

visited[0] = true;

colors[0] = 1;

return isBipartite(0, visited, colors);

}

// bipartite helper

private boolean isBipartite(int vertex, boolean[] visited, int[] colors) {

for (Vertex adjacent : vertexList[vertex].adjacencyList) { // for all adjacent to vertex

if(!visited[adjacent.index]) { // if not visited

// visit

visited[adjacent.index] = true;

// assign opposite color

colors[adjacent.index] = colors[vertex] == 1 ? 2 : 1;

// check if bipartite

if(!isBipartite(adjacent.index, visited, colors))

return false;

} else if(colors[adjacent.index] == colors[vertex]) // if same color

return false;

}

return true;

}

/\*\*

\* Test whether this graph is connected

\*/

public boolean isConnected() {

boolean[] visited = new boolean[vertexList.length];

// start from 0 and count should be same as vertices count

return count(visited, 0) == vertexList.length;

}

// count total nodes using DFS

private int count(boolean[] visited, int vertex) {

if(visited[vertex]) // already visited

return 0;

// visit

visited[vertex] = true;

int count = 1;

// count adjacent

for (Vertex adjacent : vertexList[vertex].adjacencyList) {

if(!visited[adjacent.index]) {

count += count(visited, adjacent.index);

}

}

return count;

}

/\*\*

\* Test whether this graph has any cycles

\*/

public boolean hasACycle() {

boolean[] visited = new boolean[vertexList.length];

for (int i = 0; i < visited.length; i++) {

if(!visited[i] && hasACycle(i, visited, -1))

return true;

}

return false;

}

// detect cycle using DFS

private boolean hasACycle(int vertex, boolean[] visited, int parent) {

visited[vertex] = true; // visit

for (Vertex adjacent : vertexList[vertex].adjacencyList) {

if(!visited[adjacent.index]) { // if not visited

// check if has a cyce

if(hasACycle(adjacent.index, visited, vertex))

return true;

} else if(adjacent.index != parent)

return true;

}

return false;

}

}