Assignment 3 - Analysis of the tasks.

Exercise 1, Undirected graph

This one is class is has an inner class which is a Node class that I wrote, which was the initial idea. We started using a Node[] but later on we changed to using a LinkedList array which was almost the same thing and made DFS algorithms easier,

Constructor

```
public class MyUndirectedGraph implements A3Graph {
    int totalVertices;
    ArrayList<Node> vertices;
    List<List<Integer>> connectedList = new ArrayList<List<Integer>>();
    LinkedList<Integer> [] CycleList;

    public MyUndirectedGraph () {
        this.totalVertices = 0;
        this.vertices = new ArrayList<Node>();

    }
```

```
@Override
public void addVertex(int vertex) {
    Node newNode = new Node(vertex);
    totalVertices++;
    vertices.add(newNode);
    CycleList = new LinkedList[vertices.size()];
    for (int i = 0; i < vertices.size(); i++) {
        CycleList[i] = new LinkedList();
    }

}

@Override
public void addEdge(int sourceVertex, int targetVertex) {
    vertices.get(sourceVertex).addEdge(targetVertex);
    vertices.get(targetVertex).addEdge(sourceVertex);
    CycleList[sourceVertex].addFirst(targetVertex);
    CycleList[targetVertex].addFirst(sourceVertex);
}
```

This is the constructor and the simple addVertex and addEdge methods. Not a lot to say about these, they simply create the data structures and handle the adding of edges and vertices. Edges are being handled on both lists as both are being used.

Node class

```
    class Node {
        public int vertexID;
        List<Integer> edges = new ArrayList<Integer>();
        int currentI = 0;

        public Node(int v, int maxEdges) {
            this.vertexID = v;
        }
        public void addEdge(int target) {
            boolean answer = false;
            if (edges.contains(target)) {
                answer = true;
            }
            if (answer == false) {
                edges.add(target);
            }
        }


    }


}
```

IsACyclic

```java
@Override
public boolean isAcyclic() {
    boolean answer = false;
    boolean[] visitedNodes = new boolean[totalVertices];



    for (int i = 0; i < totalVertices; i++) {
        if (visitedNodes[i] == false) {
            if(traverse(i, -1 ,visitedNodes)) {
                answer = true;
            }
        }

    }


    return answer;
}
```

This method also uses the recursive function traverse.

```java
public boolean traverse (int current, int previous, boolean[] visited) {
    visited[current] = true;


    for (int i = 0; i < CycleList[current].size(); i++) {
        int vertice = CycleList[current].get(i);
        if (vertice != previous) {
            if (visited[vertice]) {
                return true;
            }
            else {
                if (traverse(vertice, current, visited)) {
                    return true;
                }
            }
        }
    }
    return false;

}
```

So here we do a DFS (Depth first search) and traverse the edges. If we find a vertice that has been visited and is not the previous Vertice traversed, we have a cycle.
Time complexity; O(V) + O(2E) = O(V+E)
its 2E since its undirected and all the edges appear twice.

Connected components

```
public List<Integer> connectedDFS(boolean[] visited, List<Integer> connected, Integer current) {
    visited[current] = true;
    if(!connected.contains(current))
        connected.add(current);
    for(int i = 0; i < vertices.get(current).edges.size(); i++) {
        if(!visited[vertices.get(current).edges.get(i)]) {
            connectedDFS(visited, connected, vertices.get(current).edges.get(i));
        }
    }

    return connected;
}

@Override
public List<List<Integer>> connectedComponents() {
    List<List<Integer>> connectedList = new ArrayList<List<Integer>>();
    List<Integer> innerList = new ArrayList<Integer>();

    boolean[] visited = new boolean[totalVertices];

    for (int i = 0; i< vertices.size(); i++) {
        innerList = connectedDFS(visited, innerList, vertices.get(i).vertexID);
        if (!connectedList.contains(innerList))
            connectedList.add(innerList);
    }

    return connectedList;
}
```

Connected components is calculated doing by recursively calling the connectedDFS function multiple times. We simply do a DFS on every vertice and add them to a list. If the ConnectedList already contains this list, we do not add it.

Time complexity.
DFS = O(V+E)
Calling DFS for every Vertice = O(V*(E+V))
Total complexity = O(V*(E+V))

IsConnected()

```
@Override
public boolean isConnected() {
    boolean Answer = false;
    this.connectedList= connectedComponents();
    if (connectedList.get(0).size() == totalVertices) {
        Answer = true;
    }
    return Answer;
}
```

This one was fairly simple. we check if the connectedComponents first index size is equal to the total number of vertices. If so, then it is connected.
Time complexity, same as Connected Components.

Exercise 1B, Directed graph

Constructor

```java
import java.util.*;

public class MyDirectedGraph implements A3Graph {
    int totalVertices = 0;
    ArrayList<Node> vertices = new ArrayList<Node>();
    List<List<Integer>> connectedList = new ArrayList<List<Integer>>();
    ArrayList<Node> inverseVertices = new ArrayList<Node>();

    public MyDirectedGraph() {

    }
    public MyDirectedGraph(int vertices) {
        for (int i = 0; i<vertices; i++) {
            addVertex(i);
        }
    }
```

I chose to have two different constructors for the class. One empty and one where you decide how many vertices you wish to add to the graph. I reused the Node class that I made in the previous task, chose to have a variable for the total vertices in the graph, and then chose to have two array lists with nodes, and a list of arrayLists with integers for the connected components.

Connected components

```java
@Override
public List<List<Integer>> connectedComponents() {
    connectedList = new ArrayList<List<Integer>>();
    for (int i = 0; i<totalVertices; i++) {
        Node currentNode = vertices.get(i);
        for (int j = 0; j < totalVertices; j++) {
            if (j != i) {
                if (connectedDFS(new boolean[totalVertices], j, i, i)) {
                    if (connectedList.size() > 0) {
                        for(int k = 0; k < connectedList.size(); k++) {
                            if (!connectedList.get(k).contains(i) && connectedList.get(k).contains(j)) {
                                connectedList.get(k).add(i);
                                break;
                            }
                            else if (connectedList.get(k).contains(i) && !connectedList.get(k).contains(j)) {
                                connectedList.get(k).add(j);
                                break;
                            }
                        }
                    }
                    else {
                        ArrayList<Integer> list = new ArrayList<Integer>();
                        list.add(i);
                        list.add(j);
                        connectedList.add(list);
                    }
                }
                else {
                    if (connectedList.size() == 0) {
                        ArrayList<Integer> list = new ArrayList<Integer>();
                        list.add(i);
                        connectedList.add(list);
                    }
                    else {
                        boolean contained = false;
                        for (int o = 0; o < connectedList.size(); o++) {
                            if(connectedList.get(o).contains(i)) {
                                contained = true;
                                break;
                            }
                        }
                        if (!contained) {
                            ArrayList<Integer> list = new ArrayList<Integer>();
                            list.add(i);
                            connectedList.add(list);
                        }
                    }
                }
            }
        }
    }
    return connectedList;
}
```

The connectedComponents() requires 2 methods, which are the connectedDFS() and the InvertedConnectedDFS(). We have two arraylist with Nodes, which is the Vertices and inverseVertices. Once we add an edge to a vertice, we add an inverse of the edge in the inverseVertices list. For example, if we have an edge from 1 -> 2, we add an edge from 2->1 in the inverse list. The algorithm idea is to check which of the vertices are strongly connected by testing if every vertice can reach every other vertice by doing different DFS searches. For example, if we start by testing if vertice 1 is connected to 2 and find a connection between 1->2, we will then do a DFS in the inverse tree from Node 1. If we can still find a route that connects us to vertice 2, they are strongly connected. After we find the connections, we will add them to the List of array list (connectedList). First of all we have to check the size of the Linked List, to see if anything has been added previously. If not, we create a new ArrayList with the integers 1 and 2, and add them to the list. If something has already been added, we need to check every index of the connectedList if they contain any or both of the two vertices. If they only have one of the two, we add the other one to the other list, if they do not

have either, we create a new list and add it to the connectedList. If they contain both, we just break the for loop and iterate over the next item.

Also, if we do not find a connection from the first node to something else, and there are no lists in the connectedList, we add a list with just the node.

```
public boolean connectedDFS(boolean[] visited, Integer target, Integer current, Integer start) {
    boolean answer = false;
    visited[current] = true;
    Node currentNode = vertices.get(current);
    if (current == target) {
        if (invertedConnectedDFS(new boolean[totalVertices], target, start, start))
            return true;
    }
    else {
        for (int i = 0; i < currentNode.edges.size(); i++) {
            if (currentNode.edges.size() == 0){
                return false;
            }
            else {
                Integer currentEdge = currentNode.edges.get(i);
                if (currentNode.edges.contains(target)) {
                    answer = invertedConnectedDFS(new boolean[totalVertices], target, start, start);
                    break;
                }
                else if (visited[currentEdge] == false) {
                    answer = connectedDFS(visited, target, currentEdge,start);
                }
            }

        }
    }

    return answer;
}
public boolean invertedConnectedDFS(boolean[] visited, Integer target, Integer current, Integer start) {
    boolean answer = false;
    visited[current] = true;
    Node currentNode = inverseVertices.get(current);
    if (current == target) {
        return true;
    }
    else if (currentNode.edges.size() == 0) {
        return false;
    }
    else {
        for (int i = 0; i < currentNode.edges.size(); i++) {
            Integer currentEdge = currentNode.edges.get(i);
            if (currentNode.edges.contains(target)) {
                //System.out.println("Found it!");
                return true;
            }
            else if (visited[currentEdge] == false) {
                return invertedConnectedDFS(visited, target, currentEdge, start);
            }
        }
    }
    return answer;
}
```

Time complexity:
DFS = **O(V+E)**
Inverse DFS = **O(V+E)**
Worst case, we call 1 DFS and one InverseDFS call per Vertice combination

Calling the DFS for all the vertices and combinations: **O(V*V) = O(V^2)*2(V+E)**
Check for duplicates: Worst case = **O(V)**
Adding to list = **O(N)**
Total Timecomplexity: **O((V^2)*(V+E)) = O(V^3)+(E*V^2)**

IsACyclic

```
@Override
public boolean isAcyclic() {        //returns true if the graph is not cyclic
    boolean answer = true;
    if (totalVertices <=1) {
        return true;
    }
    else {
        connectedComponents();
        for (int i = 0; i < connectedList.size(); i++) {
            if (connectedList.get(i).size() > 1) {
                answer = false;
                break;
            }
        }
    }
    return answer;
}
```

First of all, we check how many total vertices we have. If we have 1 or less, it cannot create a cycle. After that, we use the connectedComponents function to fill the list of arraylists filled. Then we iterate over all the indexes and check if they have more than one item in their list. If they have 2 or more, there has to be a cycle since these nodes are strongly connected. Worst case, we have one index for every vertice, therefore the time complexity is as follows:

Time complexity = **O(ConnectedComponents)+O(V) = O(ConnectedComponents)**

IsConnected

```
@Override
public boolean isConnected() { //MyDirectedGraph, isConnected() returns true if the graph is strongly connected
    boolean answer = false;
    if (totalVertices <= 0) {
        System.out.println("No vertices in the graph");
    }
    else if (totalVertices == 1) {
        answer = true;
    }
    else {
        connectedList = connectedComponents();
        if ( connectedList.get(0).size() == totalVertices) {
            answer = true;
        }
    }
    return answer;
}
```

The plan here is to use the ConnectedComponents function and then check the list of arrayLists. There might also be only one node in the graph, and therefore we need to adress that issue as that one Node is strongly connected to itself.

Otherwise we check the connectedList's first entry and check if the size is equal to the number of vertices in the graph. If not, then they are all not connected, and therefore the graph is not strongly connected.
Time Complexity = **O(ConnectedComponents)+O(N) = O(ConnectedComponents)**

Exercise 2, Eulerpath

HasEulerPath

```java
@Override
public boolean hasEulerPath() {
    boolean answer = false;
    if (isConnected()) {
        int oddVertices = 0;
        for (int i = 0; i<vertices.length; i++) {
            if (vertices[i].edges.size() %2 == 1) {
                oddVertices++;
            }
        }
        if (oddVertices == 2 || oddVertices == 0) {
            answer = true;
        }
    }
    return answer;
}
```

Really simple since the conditions for the euler path to exist is that there have to be either 0 vertices with uneven edges (a single vertice without edges) or exactly two vertices with uneven edges. This can only appear in connected graphs.

Time complexity: Since you told us we could assume that the graph would be connected, the operation itself would be O(V) since we iterate over the vertices and check the size of the children array. If we have to check for the connectivity as well, it will be O(isConnected()) +O(N)

Euler Path

This one is a fairly large task. We need the following functions:

```java
@Override
public List<Integer> eulerPath() {
    int vertice = 0;
    ArrayList path = new ArrayList();
    //first we need to find the first odd vertice in case the odd vertices were 2 (euler path).
    for (int i = 0; i < vertices.length; i++)   {
        if (vertices[i].edges.size() %2 ==1 ) {
            vertice = vertices[i].vertexID;
            break;
        }
    }
    LinkedList<Integer>[] PathList = CycleList.clone();
    path.add(vertice);
    while (PathList[vertice].size() >0 ) {
        int bridges = 0;
        for (int i = 0; i < PathList[vertice].size(); i++) {
            if(!isABridge(PathList[vertice].get(i), vertice, PathList)) {
                int target = PathList[vertice].get(i);
                removeEdge(vertice, target, PathList);
                path.add(target);
                vertice = target;
                break;
            }
            else if (PathList[vertice].size() == 1){
                int target = PathList[vertice].get(i);
                removeEdge(vertice, target, PathList);
                path.add(target);
                vertice = target;
                break;
            }
            else {
                //System.out.println("It is");
                bridges++;
            }
        }
        if (bridges == PathList[vertice].size() && PathList[vertice].size() > 0) {
            System.out.println("They were all bridges?");
            int target = PathList[vertice].getFirst();
            removeEdge(vertice, target, PathList);
            path.add(target);
            vertice = target;
        }
    }
    return path;
}
```

This is the general function and we start by getting the first vertice with uneven edges. If you wish to find an euler path, you must start from one of the uneven vertices with uneven edges. Then you have to iterate over all the edges and remove them, but you cannot remove a “bridge” unless its the only edge you can remove. Since i wish to modify this list, I make a copy of the previous LinkedList array. Then I check the first edge to see if it is a bridge, if not, then i remove that one and change pointer to the targeted node and also add it to the path array. If a vertice only has one edge, it doesnt matter if its the bridge, we remove that one and go on.

```java
public boolean isABridge(int target, int current, LinkedList<Integer>[] list) {
    boolean answer = false;
    boolean [] visited = new boolean [totalVertices];
    int before = DFSPath(target, current, visited, 0, list, 0);

    removeEdge(current, target, list);
    visited = new boolean [totalVertices];
    int after = DFSPath(target, current, visited, 0, list, 1);

    //reverting back
    list[current].add(target);
    list[target].add(current);

    return before > after;
}
```

This is the algorithm to check wethere an edge is a bridge or not. We basicly do a DFS search before the removal of an edge to see how many nodes we can reach. We remove said edge, and then try doing the DFS again. If the result is lower, then it is a bridge.

```java
public int DFSPath(int current, int previous, boolean [] visited, int counter, LinkedList<Integer>[] list, int FirstTime) {
    visited[current] = true;
    int Counter = counter;
    for(int i = 0; i < list[current].size(); i++) {
        int vertice = list[current].get(i);
        if (vertice != previous && FirstTime == 1) {
            if(!visited[vertice]) {
                Counter = DFSPath(vertice, current, visited, Counter, list, 1) +1;
            }
        }
        else if (!visited[vertice] && FirstTime == 0) {
            Counter = DFSPath(vertice, current, visited, Counter, list, 1) +1;
        }
    }
    return Counter;
}
```

This is the DFS search which basicly searches for nodes and increases a counter.

```java
public void removeEdge(int source, int target, LinkedList [] list) {
    list[target].remove((Integer)source);
    list[source].remove((Integer)target);
}
```

the removal of nodes function.

Total time complexity:
Remove Edge: O(1)
DFS : O(E+V)
isABridge: 2*O(E+V), done E times = O(E(E+V))
Finding the first vertice with uneven edges: O(V-1) = O(V)
EulerPath : traversing all the edges = O(E)
= O(1)+3O(E+V)+O(V)+O(E*(E+V)) =O(E*(E+V))

Exercise 3, MySocialNetwork

The general idea was to use another DFS to find the relative depth from a vertice. Once they have all been added, it is fairly easy to use that array to do the rest of the calculations.

Here is the general DFS for this class.

```
public Integer[] DepthDFS(int current, Integer[] levels, int depth, boolean[] visited) {
    visited[current] = true;
    if (levels[current] != null) {
        if (levels[current] > depth) {
            levels[current] = depth;
        }
    }
    else {
        levels[current] = depth;
    }
    for (int i = 0; i < CycleList[current].size(); i++) {
        if(!visited[CycleList[current].get(i)]) {
            levels = DepthDFS(CycleList[current].get(i), levels, depth+1, visited);
        }
        else if( levels[CycleList[current].get(i)] > depth+1 ) {
            levels = DepthDFS(CycleList[current].get(i), levels, depth+1, visited);
        }
    }

    return levels;
}
```

So basically we iterate over all the vertices and add their depth to an array, and add it to the visited array. Since this is a recursive command, we must also check the the depth of each edge, even if has already been visited since the other call might be closer to the vertice which would indicate that it has a lower depth.

Time complexitiy for this one is:
O(V+E)*E

NumberofPeopleAtFriendhipDistance

```
@Override
public int numberOfPeopleAtFriendshipDistance(int vertexIndex, int distance) {
    Integer[] Levels = new Integer[totalVertices];
    boolean[] visited = new boolean[totalVertices];
    Levels = DepthDFS(vertexIndex, Levels, 0, visited);
    int people = 0;
    for (int i = 0; i < Levels.length; i++) {
        if (Levels[i] == distance) {
            people++;
        }
    }

    return people;

}
```

This one uses the previous DFS and then Iterates over the array we send in for the selected distance. **Complexity is O(DepthDFS)+O(V)**

furthestDistanceInFriendshipRelationships

```
@Override
public int furthestDistanceInFriendshipRelationships(int vertexIndex) {
    Integer[] Levels = new Integer[totalVertices];
    boolean[] visited = new boolean[totalVertices];
    Levels = DepthDFS(vertexIndex, Levels, 0, visited);
    int distance = 0;
    for (int i = 0; i < Levels.length; i++) {
        if (Levels[i] > distance) {
            distance = Levels[i];
        }
    }


    return distance;
}
```

This one is pretty much the same. Do the DFS and use the array to look for the highest distance.

Time complextiy O(DepthDFS)+O(V)

Possiblefriends

```
@Override
public List<Integer> possibleFriends(int vertexIndex) {
    Integer[] levels = new Integer[totalVertices];
    boolean[] visited = new boolean[totalVertices];
    levels = DepthDFS(vertexIndex,levels, 0, visited);
    ArrayList possibleFriends = new ArrayList();
    LinkedList friend = CycleList[vertexIndex];
    for (int i = 0; i < levels.length; i++) {
        if (levels[i] ==  2) {
            int friends = 0;
            for (int j = 0; j< CycleList[i].size(); j++) {
                if (friend.contains(CycleList[i].get(j))){
                    friends++;
                }
            }
            if (friends >2) {
                possibleFriends.add(i);
            }
        }
    }


    return possibleFriends;
}
```

Startout with the same DFS and then we then iterate over the levels array for the vertices that are at distance 2 from the starting vertice. Once we found a vertice that matches that criteria, we do a for loop for every edge. If the friend linkedList contains that edge, we increase the friends counter.

Timecomplexity:
Contains: O(N), log E times. (as this this is not all edges, but some)
Iterate over all Vertices: O(V)
Total: O(DepthDFS+V+N*logE)