Homework #1: Graph Algorithms for Social Networks

Due Tuesday, January 28th at 11:59 p.m.

In this assignment, you will implement a `Graph` interface using two different graph representations. You will then develop several algorithms that use the `Graph` interface that might be used in a social network.

Your goals for this assignment are to:

- Understand and apply the concepts of polymorphism and encapsulation
- Understand interfaces
- Familiarize yourself with Java and Eclipse
- Learn to compile, run, and debug Java programs
- Practice good Java coding practices and style

Instructions

Graph Implementations

First, write two classes that implement the `edu.cmu.cs.cs214.hw1.staff.Graph` interface, which represents an undirected graph.

- **Adjacency List:** Inside the package `edu.cmu.cs.cs214.hw1.graph`, implement the `AdjacencyListGraph` class. Your implementation must internally represent the graph as an adjacency list. Your class should provide a constructor with a single `int` argument, `maxVertices`. Your implementation must then support a graph containing as many as `maxVertices` vertices. Your implementation may behave arbitrarily if more than `maxVertices` vertices are added to the graph because we have not discussed Exceptions and other rigorous error-handling techniques. If you are not familiar with the adjacency list representation of graphs, see the Wikipedia page on the adjacency list representation as a reference.
- **Adjacency Matrix:** Next, implement the `AdjacencyMatrixGraph` class in the `edu.cmu.cs.cs214.hw1.graph` package. Your implementation must internally represent the graph as an adjacency matrix. Your class should provide a constructor with a single `int` argument, `maxVertices`, as described above. If you are not familiar with the adjacency matrix representation of graphs, see the Wikipedia page on the adjacency matrix representation as a reference.

Algorithm Implementations

For this part of the assignment, you will write three algorithms that might be used in a social network using your graph implementations. Your algorithms must use only the methods provided in the interface, and can not use any features specific to the implementation of `Graph` being used. Your algorithms must work correctly on any correct implementation of a `Graph`, including your `AdjacencyMatrixGraph` and `AdjacencyListGraph`.

- **Shortest Distance:** Implement the `shortestDistance` method in the `Algorithms` class. This method is conceptually equivalent to the `getDistance` method you implemented for Homework 0. Given a graph G and two vertices $a, b \in G$, your implementation should return the length of the shortest path between a and b, or -1 if the two vertices are not connected. The distance between a vertex and itself is 0.

- **Common Friends:** Implement the `commonFriends` method in the `Algorithms` class. Given a graph G and two vertices a and b in G, your implementation should return an array of all vertices that are connected to both a and b. The ordering of the vertices in the array does not matter, but your array cannot contain nulls. If a and b have no neighbors in common, return an empty array (an array of size 0).

  In a social network, this method might be used to determine the number of mutual friends between two people.

- **Suggest Friend:** Implement the `suggestFriend` method in the `Algorithms` class that will return a good friend suggestion for a user. Given a graph G and a vertex s, you should return a vertex d, such that $d \neq s$, d is not adjacent to s, and d and s share as many adjancent vertices as possible (i.e., the intersection of the neighborhood of d and the neighborhood of s is as large as possible). If multiple vertices satisfy the above requirement, you may return an arbitrary vertex that satisfies the requriement. If such a d does not exist or if the size of the maximum intersection is 0, then return null.

  In a social network, this might be used to provide friend suggestions because you are likely to know someone who has many mutual friends with you.

Testing Your Code

To check the correctness of your code, you must write at least three more methods to test your graph and algorithm implementations. We have provided some code in the `Main` class to help you start testing your implementations; please add a few more test cases to each method. Then, write at least three more of your own methods to test the other parts of your graph and algorithm implementations. Try to write tests that check the correctness of normal cases as well as edge cases of your algorithms.

Evaluation

Overall this homework is worth 100 points. To earn full credit you must:

- Properly encapsulate your implementation. Use the most restrictive access level that makes sense for each of your fields and methods (i.e. use `private` unless you have a good reason not to). Instead of manipulating class fields directly, make them `private` and implement getter and setter methods to manipulate them from outside of the class. See Controlling Access to Members of a Class for a reference.
- Not use the `java.util.*` libraries. An important aspect of this assignment is to familiarize you with arrays and writing your own data structures in Java. This means you may not use ArrayLists or Dictionaries or other built in data structures, except for regular Arrays.
- Not edit any files in the `edu.cmu.cs.cs214.hw1.staff` package or any of the method declarations we've initially provided for you.
- Make sure your code is readable. Use proper indentation and whitespace, abide by standard Java naming conventions, and add additional comments as necessary to document your code.
- Follow the Java code conventions, especially for naming and commenting. Hint: use **Ctrl + Shift + F** to auto-format your code!

Additional hints:

- You may create helper classes and helper methods to help you with the assignment, as long as your code is compatible with the provided staff interfaces.
- The tasks may be underspecified. In case of doubt, use your judgment or ask a question on Piazza. If you want to communicate your assumptions, use comments in the source code.
- You may reuse your code from hw0. If you decide to do so, please copy your hw0 code into your hw1 directory. Do not import your hw0 files or classes in hw1.
- You are not required to throw any exceptions on faulty input arguments for this assignment. Your implementation may behave arbitrarily on incorrect input. We will learn more about `Exception`s later. You can assume that when we grade your code, our method arguments will conform to the specifications described in this handout and our comments in the code.

- As long as your code runs in a reasonable amount of time, and returns the correct values, you do not need to worry about the complexity of your algorithms.

We will grade your work approximately as follows:

- Correct graph implementations: 40 points
- Correct algorithm implementations: 40 points
- Good style and program design: 15 points
- At least three additional test methods: 5 points

Appendix A

This is the representation of the graph provided in the `Main` class.

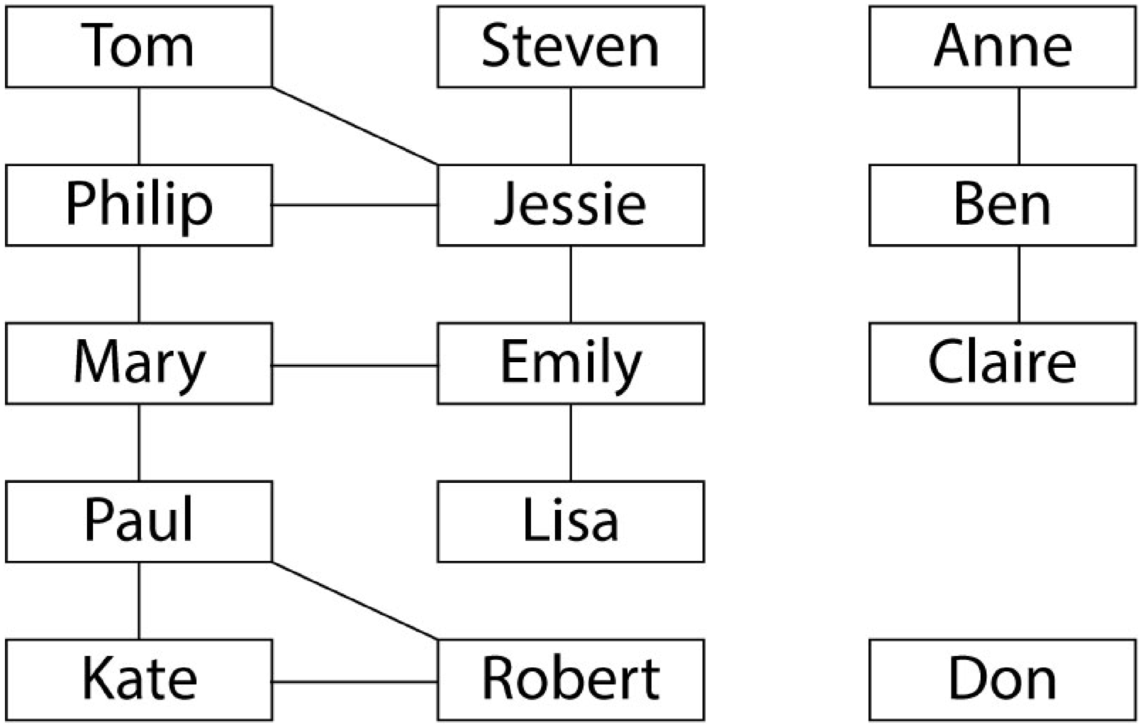

Figure 1: Example Graph