Homework #1: Graph Algorithms for Social Networks

Due Thursday, September 11th 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. You will also use unit tests to verify the correctness of your solution, and use continuous integration tools for the automated building and testing of your code.

Your goals for this assignment are to:

- Understand and apply the concepts of polymorphism and encapsulation, including an appropriate use of Java interfaces.
- Interpret and design software based on informal specifications, demonstrating an understanding of basic software design principles.
- Write unit tests with JUnit, automating builds and tests using Travis-CI.
- Understand the benefits and limitations of code coverage metrics, and interpret the results of coverage metrics.
- Demonstrate good Java coding and testing practices and style.

Instructions

You should start by installing Ant and EclEmma if you have not already done so, and configure Ant to test your homework solution both locally (by running `ant` in your `homework/1` directory) and on Travis-CI (by committing and pushing your homework to GitHub and checking the results on https://travis-ci.com). You should then implement the graph-related and algorithm-related classes described below, and test your solution. We describe each of these steps in turn.

Configuring Ant

Your repository contains a nearly empty `build.xml` file in the `homework/1` directory. You must complete the configuration file so that `ant test` will compile and test your code using the unit tests you will write (described below). We recommend that you examine the sample `build.xml` file provided in recitation 2 and that you consult the Ant manual for more information.

For this assignment we will not build your projects using Eclipse. Instead, we will compile and test your homework using the `build.xml` file that you complete. We encourage, but do not require, that you investigate how you can automate additional tasks, such as generating javadoc files and running coverage and static-analysis tools such as FindBugs and Checkstyle with Ant. (We will discuss some of these tools later in this course.) All your builds with `ant test` must complete within a time limit of one minute; we are enforcing this limitation so that each student can get a fair share of the build-and-test services we provide for this assignment.

Implementing the graphs

Write two classes that implement the `edu.cmu.cs.cs214.hw1.staff.Graph` interface, which represents an undirected graph. Your implementations must be based on two distinct graph representations:

- **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.

Implementing the graph algorithms

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`. The three algorithms you must implement are:

- **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 adjacent 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 adjacent 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 requirement. 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 implementation

Write unit tests for your homework solution, including all code provided by us. Your unit tests should:

- Check the correctness of normal cases as well as edge cases of your algorithms.
- Achieve 100% or nearly 100% line coverage. Depending on your solution, full coverage may not be achievable. If you cannot achieve 100% coverage, explain with a short comment in the uncovered area why you can't achieve 100% coverage.

We recommend that you start writing tests for your solution as you complete your solution; do not delay writing unit tests until after your implementation is complete. It is far easier to test (and find any bugs in) your graph implementations before implementing the algorithms that use your graph implementations. A common strategy is to write your unit tests as you write your code, or to even write your unit tests before you write your code, as you design your software specification.

The source code for unit tests is often organized in a separate `test` (or `tests`) directory within the software project. By mirroring the directory structure of the `src` directory, the `test` directory allows you to place unit tests in the same Java package as the project source code, without placing the test source code in the same directory as the project source.

Evaluation

Overall this homework is worth 120 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 Java built-in arrays.
- Achieve high line coverage of your solution with unit tests. 100% line coverage is not needed for full credit, but your tests and comments should convince us that any lapses are not the result of an inadequate testing.
- Apply best practices to writing unit tests. I.e., write many small, independent tests instead of bunching them all into a single test case method, etc.
- 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. You should 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 homework 0. If you reuse code, please copy your homework 0 code into your homework 1 directory. Do not import your homework 0 files or classes in homework 1.
- 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.
- You should achieve 100% line coverage for your solution, but you do not need to test (or cover) the `.java` files that contain your unit tests themselves. EclEmma can be configured to exclude these packages, if you desire.
- Unit tests should be written using JUnit 4. JUnit is a unit testing framework for Java that is intended to make it easy to implement repeatable tests. Eclipse comes with JUnit installed by default. You can create a new unit test by clicking File → New → JUnit Test Case. You can find additional documentation and starter cookbooks for using JUnit at http://junit.sourceforge.net/.
- Do not modify the `.travis.yml` and `build.xml` files in the root of your repository. We enforce a timeout of 60 seconds for all builds to balance Travis-CI capacity for all students.

We will grade your work approximately as follows:

- Functional correctness of your graph and algorithm implementations: 50 points
- Program design: 10 points
- Unit testing, including adequate coverage and compliance with best practices: 30 points
- Build automation with Ant and Travis-CI: 20 points
- Documentation and style: 10 points

Appendix A

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

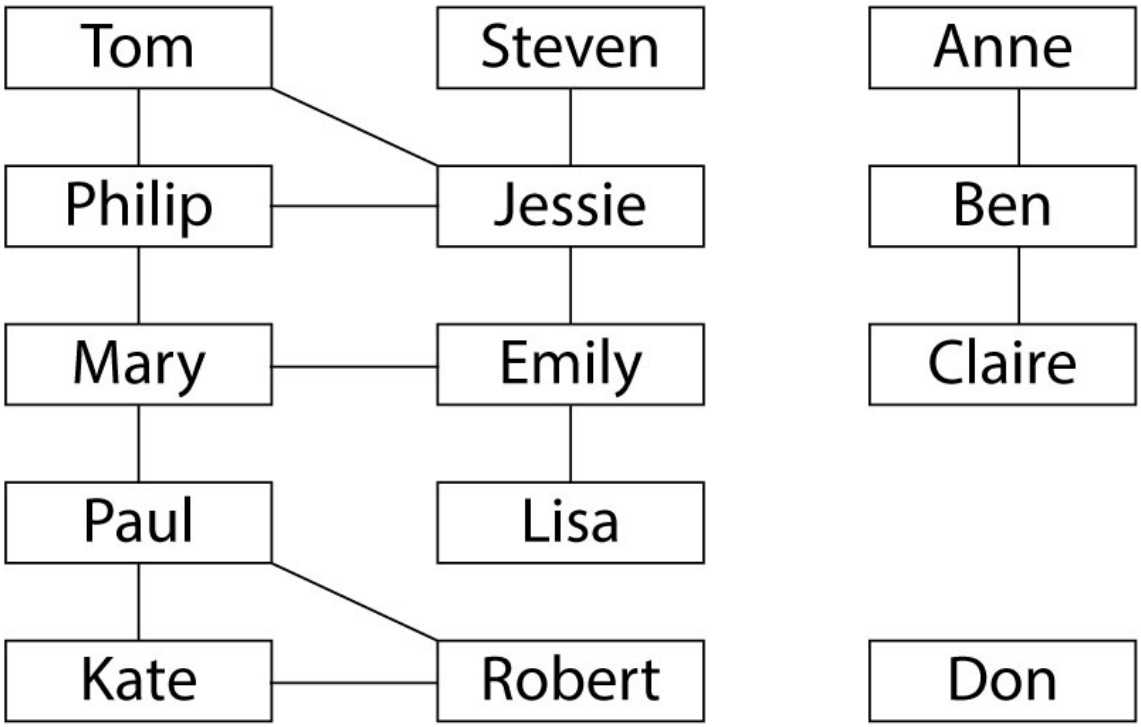

Figure 1: Example Graph