Project 4: Graph Algorithm Simulator

DUE: Apr. 11th at 11:59pm

Extra Credit Available for Early Submissions!

Setup

- Download the `p4.zip` and unzip it. This will create a folder `section-yourGMUUserName-p4`.
- Rename the folder in the same way as previous projects. After renaming, your folder should be named something like: `DL1-krusselc-p4`.
- Complete the `readme.txt` file (an example file is included: `exampleReadmeFile.txt`) .

Submission Instructions

- Same as on previous projects!
- Submit to blackboard. DOWNLOAD AND VERIFY WHAT YOU HAVE UPLOADED THE RIGHT THING. ***Submitting the wrong files will result in a 0 on the assignment!***

Basic Procedures

You must:

- Have code that compiles with the command: **`javac -cp .;../310libs.jar *.java`** (Windows) or **`javac -cp .:../310libs.jar *.java`** (Linux/MacOS) in your user directory.
- Have code that runs with the command: **`java -cp .;../310libs.jar SimGUI`** (Windows) or **`java -cp .:../310libs.jar SimGUI`** (Linux/MacOS) in your user directory.
- Have a style (indentation, good variable names, etc.) -- you must pass the style checker!
- Comment your code well in JavaDoc style (no need to overdo it, just do it well) -- you must pass the comments checker!

You may:

- Add additional methods and variables to **`ThreeTenMap`**, **`WeissPriorityQueue`**, or **`ThreeTenGraph`**, however these methods **must be private**.
- Use any of the Java Collections Framework classes in **`java.util`** **already imported for you**. You may not add any imports that do not already exist.

You may NOT:

- Make your program part of a package.
- Add additional public methods or variables.
- Add any additional libraries/packages which require downloading from the internet.
- Use any code from the internet (including the JUNG libary) which was not provided to you in 310libs.jar.
- Import any additional libraries/packages or add any additional import statements (or use the "fully qualified name" to get around adding import statements). So you cannot use any built in Java Collections Framework classes that are not already imported for you.
- Alter any method/class signatures defined in this document of the template code. Note: "throws" is part of the method signature in Java, don't add/remove these.
- Alter provided interfaces/classes that are complete (e.g. **`SimGUI`**, **`ThreeTenNode`**, etc.).
- Add **`@SuppressWarnings`** to any methods unless they are private helper methods for use with a method we provided which already has an **`@SuppressWarnings`** on it.

Grading Rubric

No Credit

- Same as previous projects! (Non submitted assignments. Assignments submitted more than 48 hours late. Non-compiling assignments. Non-independent work. Code that violates and restrictions or "you may not" mandates. "Hard coded" solutions. Code that would win an obfuscated code competition with the rest of CS310 students.)

How will my assignment be graded?

- Automatic Testing (100%): To assess the correctness of programs.
- You CANNOT get points for code that doesn't compile or for submitting just the files given to you.
- Extra credit for early submissions:
  - 1% extra credit rewarded for every 24 hours your submission made before the due time
  - Up to 5% extra credit will be rewarded
  - Your latest submission before the due time will be used for grading and extra credit checking. You CANNOT choose which one counts.

Automated Testing Rubric

The JUnit tests used for grading will NOT be provided for you (you need to test your own programs!), but the tests will be based on what has been specified in the project description and the comments in the code templates. A breakdown of the point allocations is given below:

10pts	WeissPriorityQueue update() method
40pts	ThreeTenMap
50pts	ThreeTenGraph
-5pts ("off the top")	**Not following the submission format** Note: This is very, *very* important for these assignments; the graders need to return grades *very* fast. If you do not follow the submission format (the same one you've been using for P0-3), then they will manually deduct 5pts from your score. No exceptions.
-5 pts ("off the top")	Not passing code style check
-5 pts ("off the top")	Not passing JavaDoc style check

"Off the top" points (i.e. items that will lose you points rather than earn you points).

Assignment Overview

Professional code often uses existing libraries to quickly prototype interesting programs. **You are going to use 3-4 established libraries to develop the internal representation of an advanced data structures (a graph), simulate a simple interactive graph editing program, and provide the support data structures for running Dijkstra's shortest path algorithm**. While Dijkstra's shortest path is very cool, you are not actually implementing it (that's done for you).

The four libraries you are going to use are:

1. The PriorityQueue portion of the Weiss data structures library from your textbook (this is provided to you with your textbook for free on the author's website, if you've never looked).
2. The Set/Map library from this assignment. The map class you are writing, the set class is provided (and is built using your map class).
3. A subset of the Java Collections Framework - This is a collection of existing simple data structures (lists, queues, etc.) which can form the basis for more advanced data structures.
4. A subset of JUNG (Java Universal Network/Graph Framework) - This library provides a lot of cool visualization tools for graphs: automatic layouts for graphs, an easy interface for creating/editing graphs, and much more.

Tasks Breakdown and Sample Schedule

There are **5** tasks in this assignment. It is suggested that you implement these tasks in the given order:

- Task 1: Examine the Library Classes (0%)
- Task 2: Read the Provided Code Base (0%)
- Task 3: Implement the Map Library Class to Support the Graph (40%)
- Task 4: Complete the PriorityQueue Library Class to Support Dijkstra's Shortest Path Alg. (10%)
- Task 5: Implement a Graph Class to Support the Simulator (50%)

See the Examples for Testing section of this document for what the simulator should be able to do when you are done, and then see the Task Details section for a walk-through of each specific task.

Need a schedule?

- You've got 2.5 weeks. Pace yourself. Remember that you have other classes with exams/projects.
- The priority queue class has 1 method you need to write, the map class has 12 methods you need to write. The graph class has 16 methods you need to write. However, most of these methods are only one or two lines of code.
- Assume you want to spend the last half week getting EC or seeking additional help.
- Keeping those things in mind, fill in the following:
    - 3/24-3/25: _____ (first week period)
        - Suggestions: Task 1 and Task 2
    - 3/26-3/28: _____ (first weekend period)
        - Suggestions: JavaDocs for All Classes, Plan/Start Task 3
    - 3/29-4/1: _____ (second week period)
        - Suggestions: Finish Task 3, Start Task 4
    - 4/2-4/4: _____ (second weekend period)
        - Suggestions: Finish Task 4, Task 5
    - 4/5-4/8: _____ (third week period)
        - Suggestions: Thorough Testing and Debugging
    - 4/9-4/11: _____ (third weekend period)
        - Suggestions: Turn in early for Extra Credit

Task Details

There are **5** tasks in this assignment. Again, it is suggested that you implement these tasks in the given order!

Task 1: Examine the Library Classes and Interfaces (0%)

Read and familiarize yourself with the textbook code and the following JCF classes. Some of these are allowed (or required) in certain parts of the code. Below is an overview:

1. Collection - All JCF classes implement this generic interface.
2. ArrayList - Java's list class supported by a dynamic array.
3. Set and Map - The required methods for sets and maps for the JCF.

Where should you start? The Java Tutorials of course! (If you didn't know, Oracle's official Java documentation includes a set of tutorials.) Trail: Collections will provide you with more than enough information on how to use these classes.

Task 2: Read the Provided Code Base (0%)

Read and familiarize yourself with the code. This will save you a lot of time later. An overview of the provided code in is given below, but you need to read the code base yourself.

```
//This class represents a Map (Key->Value pairings) using hashing
//with Hopgood-Davenport probing. You will write ~90% of this class,
//and a template is provided.
class ThreeTenMap {...}

//This class represents a Set (an unordered collection with no
//duplicates). This class is provided in full but needs JavaDocs.
class ThreeTenSet {...}

//This is the Priority Queue code from your textbook. It's complete,
//but for Dijkstra's shortest path algorithm it will need one more
//method added by you (an update method).
class WeissPriorityQueue {...}

//These are part of the textbook library and support the Weiss
//PriorityQueue. They are complete.
class WeissCollection {...}
class WeissAbstractCollection {...}

//Represents a node in a graph used for the GUI. This class is
//provided in full.
class ThreeTenNode {...}

//Represents an edge in a graph used for the GUI. This class is
//provided in full.
class ThreeTenEdge {...}

//This class represents a directed graph formed by connecting
//ThreeTenNodes with ThreeTenEdges. You will write 99% of this
//class, but a template is provided.
class ThreeTenGraph {...}

//These are the simulator, and simulator-related classes. They
//handle the actual algorithm and all the graphical stuff.
//These are provided in full but may be of interest to some people.
class SimGUI {...}
class ThreeTenAlg {...}
class ThreeTenDijkstra {...}
```

You are required to complete the JavaDocs and adhere to the style checker as you have been for all previous projects. The **`checkstyle.jar`** and associated **`.xml`** files are the same as on previous projects. You need to correct/edit the provided style and/or JavaDocs for some classes because the Weiss and JUNG library comments don't quite adhere to the style requirements for this class. Updating/correcting another coder's style is a normal process when integrating code from other libraries – so this is boring but necessary practice for the "real world".

It is HIGHLY RECOMMENDED that you write your JavaDocs for this project *first* and during this stage. That way you will have a full understanding of the code base as you work. Don't overdo it! Remember you can use the inheritdoc comments for inheriting documentation from interfaces and parent classes!

Task 3: Implement the Map Library Class to Support the Graph (40%)

Efficient graph representations require maps and/or sets, as does efficient updates for heaps. The JCF library provides an interfaces for maps: **`Map<K,V>`**. You need to implement a map **`ThreeTenMap<K,V>`** (in **`ThreeTenMap.java`**) which implements the **`Map<K,V>`** interface. Once you have completed class, you will automatically have a working class that implements the JCF's **`Set<E>`** interface (**`ThreeTenSet<E>`** in **`ThreeTenSet.java`**).

The **`ThreeTenMap`** class uses open addressing with Hopgood-Davenport probing. Hopgood-Davenport probing is designed for tables of size 2^x which can handle loads up to a value of 1. In linear probing you add **`+i`** where **`i`** is the number of failed probes. In quadradic probing you add **`+i`**^{2}. With Hopgood-Davenport probing, do a mix of the two such that you add: **`+0.5i+0.5i`**^{2}. This results in a probe sequence of +1, +3, +6, etc. And no, you cannot save the list of +X anywhere, you must calculate it. We'll be testing your code with extremely large tables.

Task 4: Complete the PriorityQueue Library Class to Support Dijkstra's Shortest Path Alg. (10%)

The graph simulator is going to run Dijkstra's shortest path algorithm. This requires a priority queue with the ability to update the priority of individual items. If you look at your textbook, it gives information in Chapter 21 (section 4) about the "decrease key" operation. We want a similar **`update()`** operation which increases (or decreases) the priority of the item. The item (or an equal item) will be provided and it should update the priority queue appropriately.

This has no big-O restrictions, but if you want an efficient method for doing updates, you'll want to use the **`Map`** you just built to keep track of the items and their indexes (trading space for time). Again, it isn't required to make this efficient, but you might want to give this a try if you finish early. Hint for real life coding: get it working first, optimize second.

Task 5: Implement a Graph Class to Support the Simulator (50%)

In order for the graph simulator to work, you need an internal representation of a graph! The JUNG library provides an interfaces for this: **`Graph<V,E>`**. You need to implement the directed graph **`ThreeTenGraph`** (in **`ThreeTenGraph.java`**) which implements the **`Graph<ThreeTenNode,ThreeTenEdge>`** and **`DirectedGraph<V,E>`** interfaces. The **`ThreeTenGraph`** class uses map storage for fast sparse graph representations.

Below is a quick overview of the methods you need to support. Note that in the template, actual JavaDoc comments are provided. That said, the JavaDocs are from the **`Graph<>`** interface and the **`HyperGraph<>`** interface in JUNG. They have been copied from that library for your reference, but are otherwise unaltered. Part of this assignment is to practice reading "real" documentation and understanding what to implement based on the library's requirements.

```
//*******************************
// Graph Editing (~20%)
//*******************************
addEdge() {...}
addVertex() {...}

removeEdge() {...}
removeVertex() {...}

//*******************************
// Graph Information (~30%)
//*******************************

//For a given graph...
getEdges() {...}
getVertices()  {...}

getEdgeCount() {...}
getVertexCount() {...}
```

```
//For a given vertex in a graph...
getPredecessors()
getSuccessors()
getInEdges()
getOutEdges()

//Given two verticies in a graph...
findEdge() {...}

//Given an edge in a graph...
getSource()
getDest()
```

When you are done with this step, you can generate and play with some graphs in the simulator (see the next section).

Hints and Notes

- Read ALL the methods before you decide how to implement any methods, you may need/want to track a lot more things than the simple graphs we covered in class.
- Note that we cannot test editing a graph or getting information about a graph independently of each other. So you cannot get points for completing only the graph editing or only the graph information parts of this interface, you need everything...

Examples for Testing

`ThreeTenMap`, **`ThreeTenSet`**, **`WeissPriorityQueue`**, and **`ThreeTenGraph`** have main methods for testing with a couple of sample "yays". You should thoroughly test this class before trying to run the simulator. If you see errors/weird things in the simulator it means something is wrong in YOUR code. All the simulator does is call your methods! For example:

Issue:

You delete a node, but there are still edges appearing "in the air" on the simulator which should have been removed.

Possible Causes:

Your removeVertex() method might not be working properly.
Your getEdges() or getEdgeCount() methods might not be working properly.
Your map storage class might not be working properly.

Definitely NOT the Cause:

The simulator is broken.

How do you diagnose the problems?

Debugger, breakpoints, print statements...

Displaying a Graph

Once you have the Graph class working, you can run the main program to test and debug. The main program is **`SimGUI`**, which can be compiled and run with the following commands if you are on Windows:

```
javac -cp .;../310libs.jar *.java
java -cp .;../310libs.jar SimGUI
```

or the following commands if you are on Linux/MacOS:

```
javac -cp .:../310libs.jar *.java
java -cp .:../310libs.jar SimGUI
```

Why is there extra stuff? The **`-cp`** is short for **`-classpath`** (meaning "where the class files can be found). The **`.;../310libs.jar`** or **`.:../310libs.jar`** has the following components: **`.`** the current directory, **`;`** or **`:`** the separator for Windows or Linux/MacOS respectively, **`310libs.jar`** the provided jar file which contains the library code for JUNG.

If you run the simulator with the above command, you will get a six node graph with some random edges. Each time you hit "reset" you get another graph, but the same sequence of graphs is always generated (for your testing). However, the simulator can also be run with some additional optional parameters to get some more interesting results: The number of nodes, the likelihood that two nodes have an edge between them, the random seed for the graph generator. The next few tables give examples of what you can do.

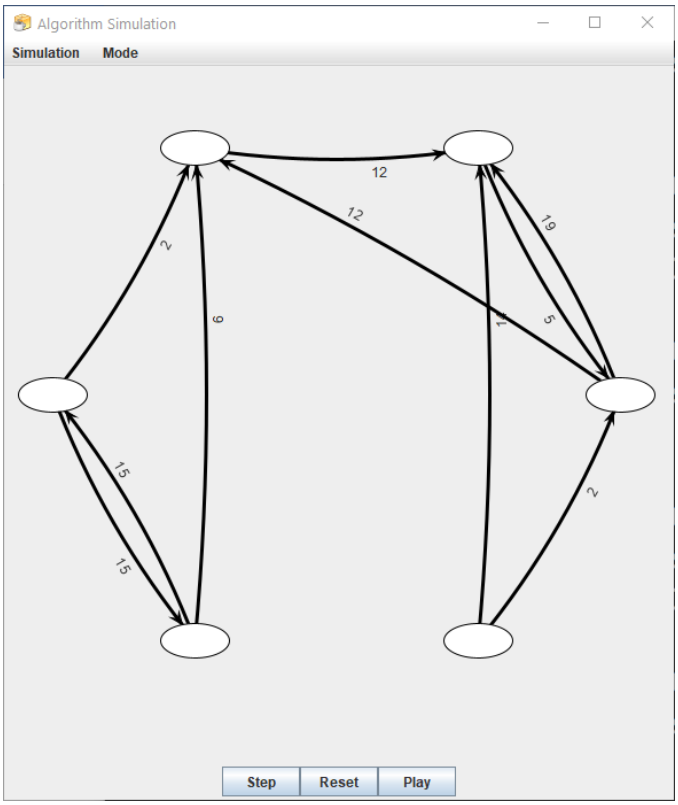
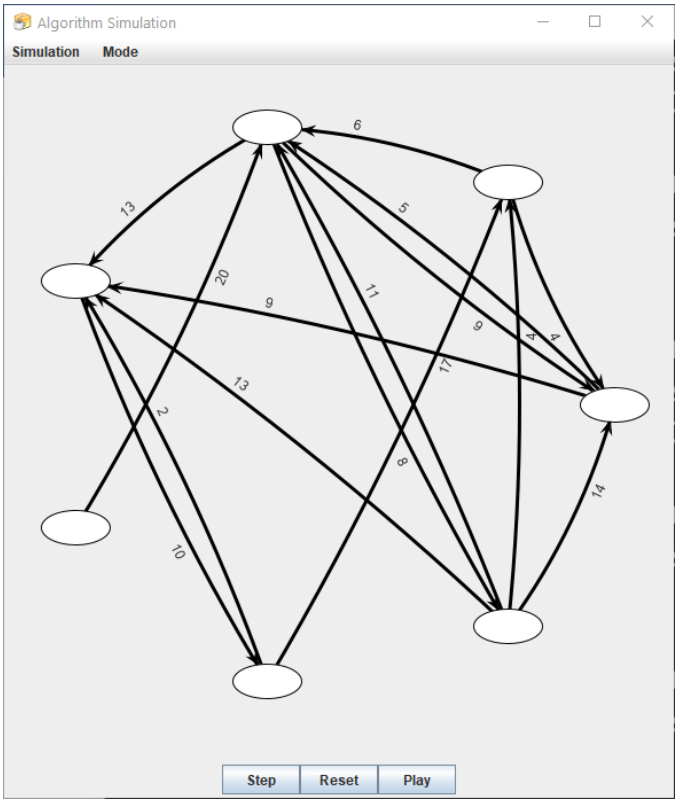
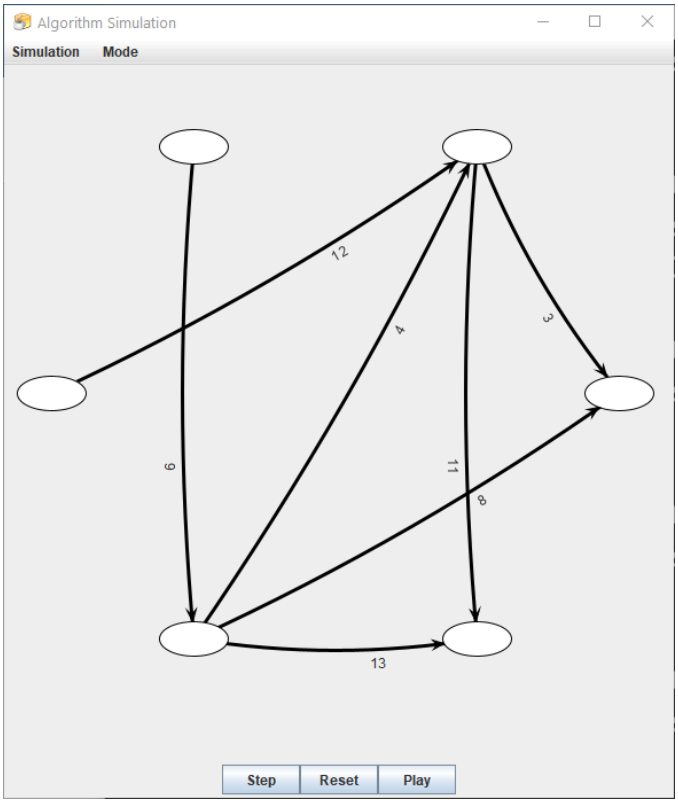
Image	**Command + Explanation**
	**`java -cp .;../310libs.jar SimGUI`** Generate a six node graph, with connection probability of 0.4, and seed 0.

Image	Command + Explanation
	**`java -cp .;../310libs.jar SimGUI 7 0.5`** Generate a seven node graph where nodes have a 50% chance of being connected.
	**`java -cp .;../310libs.jar SimGUI 6 0.4 1123`** Generate a different sequence of graphs using seed 1123.

Adding nodes and edges to a graph

You'll want to test out adding multiple nodes and edges from to make sure you've gotten out all the bugs.

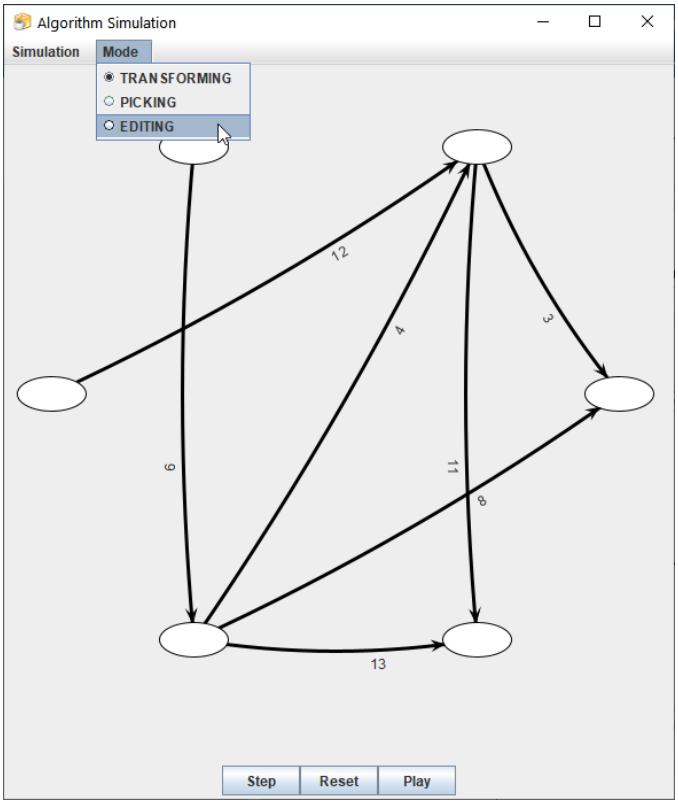
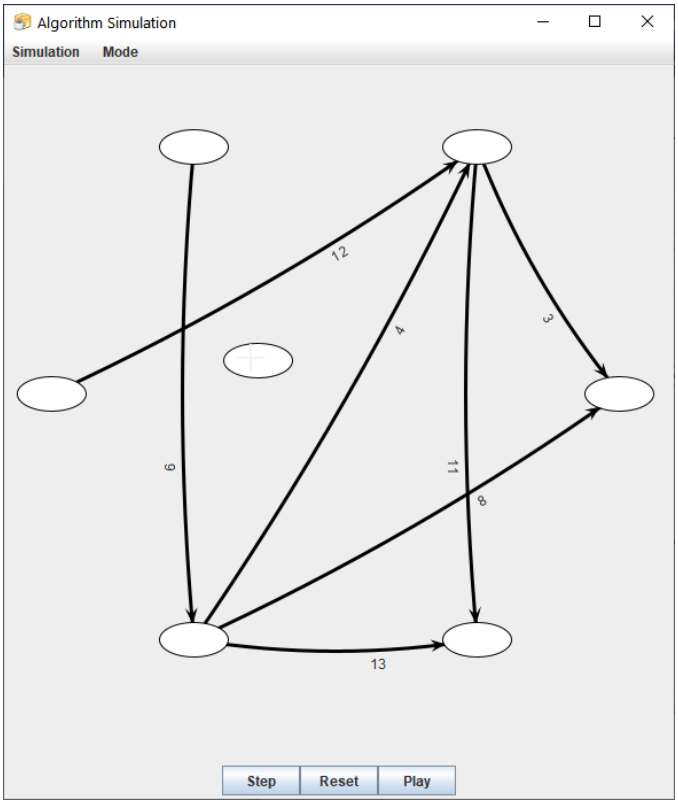
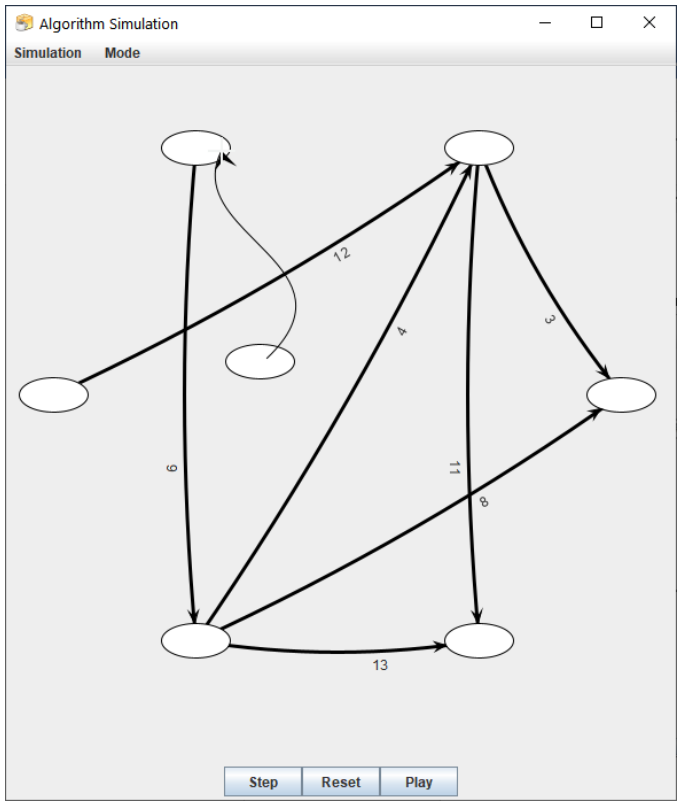
Image	Explanation
	Select "Mode", then "Editing".
	Click Anywhere on the graph surface to add a node.

Image	Explanation
	Drag to another node to add an edge.

Removing a nodes and edges from a graph

You'll want to test out removing multiple nodes and edges to make sure you've gotten out all the bugs.

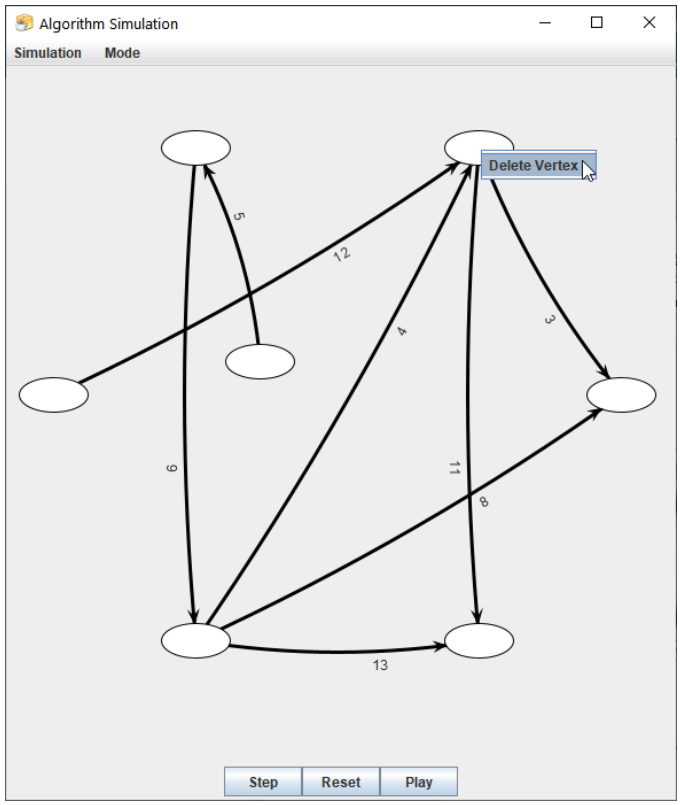
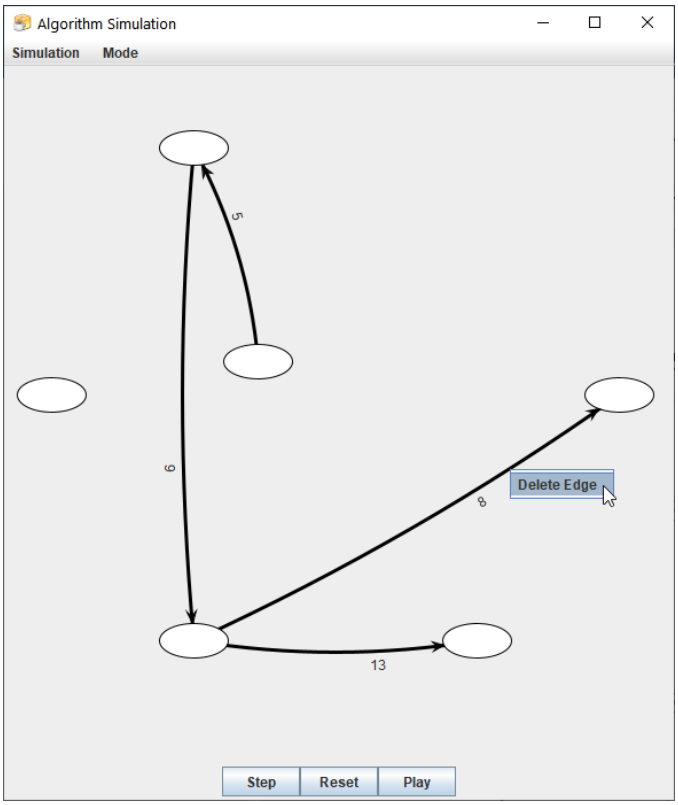
Image	Explanation
	In any mode, right click a node and select "delete vertex".

Image	Explanation
	In any mode, right click an edge and select "delete edge".