Project 4: Kruskal's Algorithm Simulator

DUE: November 26th, 2023 at 11:59pm

Extra Credit Available for Early Submissions!

Basic Procedures

You must:

- Have a style (indentation, good variable names, etc.) and pass the provided style checker (See P0).
- Comment your code well in JavaDoc style and pass the provided JavaDoc checker (See P0).
- Implement all required methods to match the expected behavior as described in the given template files.
- For methods that come with a big-O requirement (check the provided template Java files for details), make sure your implementation meet the requirement.
- Have code that compiles with the command below in your user directory without errors or warnings.
  - On a Windows machine: `javac -cp .;../310libs.jar *.java`
  - On a Linux/MacOS machine: `javac -cp .:../310libs.jar *.java`
- Have code that runs with the command below in your user directory.
  - On a Windows machine: `javac -cp .;../310libs.jar SimGUI`
  - On a Linux/MacOS machine: `javac -cp .:../310libs.jar SimGUI`

You may:

- Add any additional private helper methods you would find useful.
- Add any additional classes you have written entirely yourself (such as simple data structures from previous projects).
- Use any of the Java Collections Framework classes already imported for you (others are off limits).

You may NOT:

- Add any additional class or instance variables to existing classes (you MUST use what is there!). Remember local variables are not the same as class/instance variables.
- Alter any method signatures defined in the template code. Note: "throws" is part of the method signature in Java, don't add/remove these.
- 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.
- Alter any fully provided classes (specifically `SimGUI`) or methods that are complete and marked in a "DO NOT EDIT" section.
- Make your program part of a package.
- Add any additional import statements (or use the "fully qualified name" to get around adding import statements).
- Use any code from the internet (including the JUNG libary) which was not provided to you in 310libs.jar.

Setup

- Download the `p4.zip` and unzip it. It contains a template for all the files you must implement.

Submission Instructions

- Make a new temporary folder and copy there your **.java** files. Do not copy the test files, jar files, class files, etc.
- Upload the temporary folder to OneDrive as a backup (this is not your submission, just a backup!)
- Follow the Gradescope link provided in Blackboard>Projects and upload the files from your temporary folder onto Gradescope's submission site. **Do not zip the files or the folder.**

Grading Rubric

Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading, including extra credit for early submissions.

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), and simulate a simple graph algorithm (Kruskal's MST). This GUI can be easily extended to other types of algorithms you might encounter in the future (such as in CS483), and you may find it useful to simulate later algorithms you learn using this same framework. The four libraries you are going to use are:

1. The BinarySearchTree 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. 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.
3. Set and Map code provided in this project. They implement the interfaces defined in JCF.
4. A subset of JUNG (Java Universal Network/Graph Framework) (https://jung.sourceforge.net) - 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 JCF Classes (0%)
- Task 2: Read the Provided Code Base (0%)
- Task 3: Implement the Map Library Class to Support the Graph (20%)
- Task 4: Implement a Graph Class to Support the Simulator (60%)
- Task 5: Implement Kruskal's MST Algorithm in the Simulator (15%)

See the Sample Run 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.

Sample Schedule

- You've got 3 weeks.
- You have other classes with final exams/projects.
- Assume you want to spend the last half week doing the EC and getting additional help.
- Keeping those things in mind, fill in the following:
  - 11/06-11/10: _____ (first week)
    - Suggestions: Task 1 and Task 2
  - 11/11-11/12: _____ (first weekend)
    - Suggestions: Task 3, complete JavaDocs for All Classes
  - 11/13-11/17: _____ (second week)
    - Suggestions: Task 4
  - 11/18-11/19: _____ (second weekend)

- Suggestions: Finish Task 4 and Task 5

- 11/20-11/24: _____ (third week)
  - Suggestions: Thorough Testing and Debugging
- 11/25-11/26: _____ (third weekend)
  - Suggestions: Turn in early for Extra Credit

Tasks Details

Task 1: Examine the Library Classes and Interfaces

Read and familiarize yourself with the JCF classes. You must use these classes in your project, so becoming familiar with them *before* starting is important. Below is an overview of the classes:

1. Set - Java's Set interface (https://docs.oracle.com/javase/9/docs/api/java/util/Set.html).
2. Map - Java's Map interface (https://docs.oracle.com/javase/9/docs/api/java/util/Map.html).
3. Collection - All JCF classes implement this generic interface (https://docs.oracle.com/javase/9/docs/api/java/util/Collection.html).

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

Task 2: Read the Provided Code Base

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 is the parent class of all graph components.
//It is provided and should not be altered.
class GraphComp {...}

//This class represents a node in a graph.
//It is provided and should not be altered.
class GraphNode {...}

//This class represents an edge in a graph.
//It is provided and should not be altered.
class GraphEdge {...}

//This class represents a undirected graph.
//You will write 90% of this class, but a template is provided.
class Graph310 implements Graph<GraphNode,GraphEdge>,
UndirectedGraph<GraphNode,GraphEdge> {...}

//This is the Binary Search Tree code from your textbook. It's 90% complete,
//but to support our project, it will need some more work implemented by you.
class WeissBST {...}

//This is the implementation of a Set class using WeissBST.
//It is provided and should not be altered.
class Set310 {...}

//This is the implementation of a Map class. It's 90% complete - you need to
implement two methods and a template is provided.
class Map310 {...}
```

```
//This interface defines an algorithm that can be simulated with the GUI.
//It is provided and should not be altered.
interface ThreeTenAlg {...}

//You will be completing this algorithm (a template provided).
class Kruskal310 implements ThreeTenAlg {...}

//This is the simulator and handles all the graphical stuff, it is provided.
class SimGUI {...}
```

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 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 Classes to Support the Graph

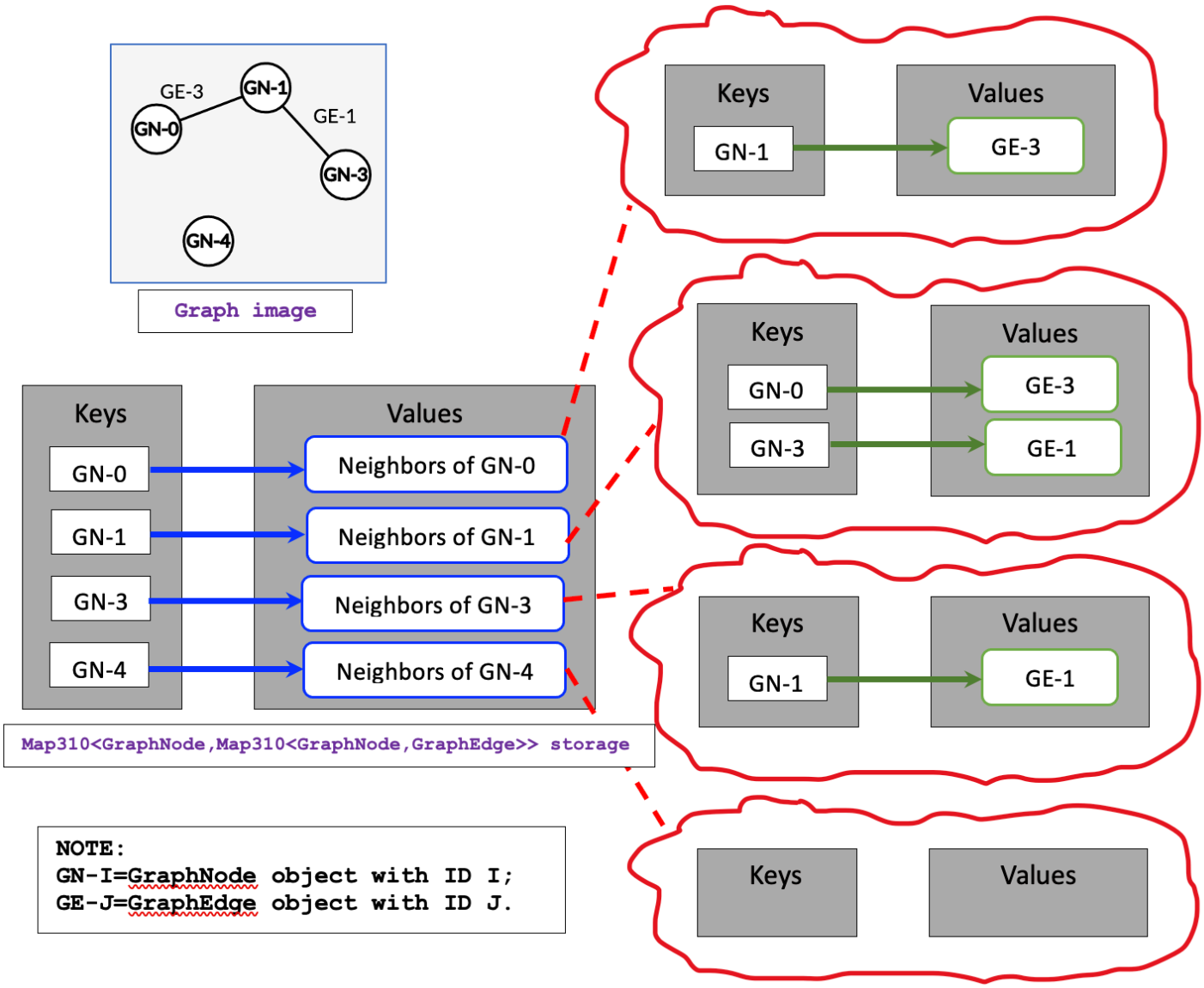
Efficient graph representations require maps and/or sets. We will use binary search trees (**`WeissBST.java`**) to implement both sets and maps for this project. Most methods in **`WeissBST.java`** are written for you (from Weiss textbook), and you will need to add a couple of methods to have it completed (check the provided template file for details). **`Set310.java`** and **`Map310.java`** both use **`WeissBST.java`**. Once you have completed the implementation of **`WeissBST.java`**, you will automatically have a working **`Set310.java`** and an almost working **`Map310.java`.** You will need to write two more methods in the map class to complete the work. **`Set310.java`** and **`Map310.java`** implement the JCF's interfaces `Map<K,V>` and `Set<K>` respectively.

Task 4: Implement an Undirected Graph Class to Support the Simulator

In order for the 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 undirected graph **`(Graph310.java)`** which implements the **`Graph<GraphNode,GraphEdge>`** interface.

The method of storage we use for this project is at its core an adjacency list, with the following details:

- The adjacency list of a graph is maintained as a Map that maps each vertex v to the collection of nodes adjacent to it. Every vertex has a unique integer ID.
- The neighbor collection of any vertex v is also maintained as a Map, which maps each neighbor n to the edge connecting between v and n. Each edge also has a unique integer ID.
- The overall storage is therefore a combination of these two levels of maps: **`Map310<GraphNode,Map310<GraphNode,GraphEdge>>.`**
- We are implementing an **undirected** graph. Self-loops and parallel edges are not permitted.
    - Note that a **`GraphEdge`** object does not keep the information of the nodes adjacent to the edge internally. You will need to record/extract that information using the maps we constructed. Check the example below.



A concept image of the graph storage with four vertices (ID 0, 1, 3, and 4) and two edges (ID 1 and 3) is shown in this example. The vertices/edges with other IDs are not in the graph and therefore not in our storage.

Once you understand how the storage should work, you will need to implement several support methods for the graph visualization. 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 those 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
//********************************

boolean addEdge(GraphEdge e, GraphNode v1, GraphNode v2 {...}
boolean addVertex(GraphNode vertex) {...}

boolean removeEdge(GraphEdge edge) {...}
boolean removeVertex(GraphNode vertex) {...}

//********************************
```

```
// Graph Information
//******************************
//For a given graph...

Collection<GraphNode> getVertices()  {...}
int getVertexCount() {...}

Collection<GraphEdge> getEdges() {...}
int getEdgeCount() {...}

//For a given vertex in a graph...
boolean containsVertex(GraphNode vertex) {...}

Collection<GraphNode> getNeighbors(GraphNode vertex) {...}
int getNeighborCount(GraphNode vertex) {...}

Collection<GraphEdge> getIncidentEdges(GraphNode vertex) {...}

Set310<GraphNode> reachableSet(GraphNode vertex) {...}

//Given two vertices in a graph...
GraphEdge findEdge(GraphNode v1, GraphNode v2) {...}

//Given an edge in a graph...
Pair<GraphNode> getEndPoints(GraphEdge edge) {...}

//Given a vertex and an edge in a graph...
boolean isIncident(GraphNode vertex, GraphEdge edge) {...}
```

When you are done with this step, you can generate and edit graphs in the simulator (check examples in the separate document for sample runs: **project4-sampleruns.pdf**).

Hints and Notes

- Read ALL the methods before you decide how to implement any methods, you may need/want to reuse code from some methods in implementing others.
- 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…

Task 5: Implement Kruskal's MST Algorithm in the Simulator

Now for the fun part! The simulator needs to know what one "step" of Kruskal's Algorithm looks like. `Kruskal310` (in `Kruskal.java`) will provide the steps for the algorithm, and it's mostly written, but there are some key pieces missing! You're going to "fill in the blanks" to get it working. An overview of some key parts of **`Kruskal310`** class is given below to get you started. Make sure to check the actual file (**`Kruskal310.java`**) for detailed description and requirements of each method.

```
//this is the "step" method, which calls other methods you will be completing...
boolean step() {
  if(!started) {
          start();
          return true;
  }

  cleanUpLastStep();
  if(!setupNextStep()) {
          finish();
          return false;
  }
  doNextStep();

  return true;
}
```

```
//this does most of the setup for our algorithm
void start() {...}

//performs any "clean up"; not used by this project
void cleanUpLastStep() {...}

//check whether the algorithm is done
boolean setupNextStep() {...}

//main method for the algorithm
//you will need to complete one step of the algorithm
void doNextStep() {...}

//cleans up after the algorithm finishes
void finish() {...}
```

When you are done with this step, you can play the algorithm in the simulator (see examples in the separate document for sample runs: **`project4-sampleruns.pdf`**).

Hints and Notes

- In the simulation, we will construct a subgraph using graph nodes and edges from the original graph following Kruskal's algorithm. Do NOT remove any nodes or edges from the original graph though – we will change the color of nodes / edges to indicate their status change. Check **`Kruskal310.java`** and the sample runs to see more details.
- A binary search tree (**`WeissBST.java`**) is used to sort / store the weighted edges in our simulation.
- If the original graph is connected, the simulation would identify a minimum, spanning tree; if it is disconnected, the simulation would identify a minimum forest instead.
- You are not responsible for making the algorithm work if the user edits the graph while the algorithm is running. Just assume all editing will take place before hitting "step" or "play" and that, if they want to do the algorithm again, they will hit "reset" and generate a new graph.

Testing

There is limited testing code in main methods in the provided template files. You can use command like "**`java Graph310`**" or "**`java WeissBST`**" to run the testing defined in **`main()`**. You should thoroughly test these classes 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.
- Definitely NOT the Cause: The simulator is broken.
- How do you diagnose the problems?
  - Debugger, breakpoints, print statements.