Project 4: Topological Sorting Simulator

DUE: April 30th, 2023 at 11:59pm

Extra Credit Available for Early Submissions!

Basic Procedures

You must:

- Fill out a readme.txt file with your information (goes in your user folder, an example readme.txt file is provided).
- 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. This will create a folder `section-yourGMUUserName-p4`;
- Rename the folder replacing `section` with the `004` or `006` based on the lecture section you are in;
- Rename the folder replacing `yourGMUUserName` with the first part of your GMU email address;
- After renaming, your folder should be named something like: `000-yzhong-p4.`
- Complete the `readme.txt` file (an example file is included: `exampleReadmeFile.txt`).

Submission Instructions

- Make a backup copy of your user folder!
- Remove all test files, jar files, class files, etc. You should just submit your java files and your readme.txt
- Zip your user folder (not just the files) and name the zip `section-username-p4.zip` (no other type of archive) following the same rules for `section` and `username` as described above. For example:

  -
    ```
    000-yzhong-p4.zip --> 000-yzhong-p4 -->   JavaFile1.java
                                              JavaFile2.java
                                              JavaFile3.java ...
    ```
- Submit to blackboard.

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 (topological sorting). 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 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. Your previous project code - You will need a hash table for part of this project, and you have written one in Project 2 and 3. You're encouraged to reuse your implementation!
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) (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.

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 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. LinkedList - Java's linked list class (https://docs.oracle.com/javase/9/docs/api/java/util/LinkedList.html).
2. HashMap - Java's map class supported by a hash table (https://docs.oracle.com/javase/9/docs/api/java/util/HashMap.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 directed graph.
//You will write 90% of this class, but a template is provided.
class ThreeTenGraph implements Graph<GraphNode,GraphEdge>,
DirectedGraph<GraphNode,GraphEdge> {...}

//This is the Priority Queue code from your textbook. It's complete,
//but to support our Topological Sorting Algorithm, it will need some more work
//(implemented by you).
class WeissPriorityQueue {...}

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

//This interface defines an algorithm that can be simulated with the GUI
interface ThreeTenAlg {...}

//You will be completing this algorithm (a template provided)
class TopologicalSort 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 a Directed 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 directed graph **`(ThreeTenGraph.java)`** which implements the **`Graph<GraphNode,GraphEdge>`** interface.

The method of storage we are going to use is at its core an adjacency matrix, with the following details:

- To store information about the nodes themselves, we keep an array of **`GraphNode`** objects (one per vertex in the graph). Every **`GraphNode`** has an integer ID property and uses that to determine which index is used to keep the **`GraphNode`** object in our storage array. We promise that we will never create or consider a graph with IDs below 0 or above 199 (but ID 199 *is* possible).
- The graph storage will be a 2D array of **`GraphEdge`** objects, rather than a 2D int array. 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 layout of the adjacency matrix we constructed. Check the example below.

A concept image of the graph storage with three vertices (ID 0, 2, and 3) and three edges (ID 1, 2, and 3) is shown below. The vertices with other IDs are not in the graph and therefore not in vertexList. Edge 1 starts with vertex 0 and points to vertex 2; edge 2 has the same starting vertex (vertex 0) but ends at vertex 3. Vertex 3 has an outgoing edge (edge 3) to vertex 2.

vertexList:	**GN-0**	**null**	**GN-2**	**GN-3**	**null**	…	**null**
Index	0	1	2	3	4	…	199

matrix:	0	1	2	3	…
0			**GE-1**	**GE-2**	
1					
2					
3			**GE3**		
…					

NOTE: GN-I=GraphNode object with ID I; GE-J=GraphEdge object with ID J.

Remember that all graph components have IDs, so edges have IDs too (and those IDs do not relate in any way to what vertices they connect, the placement in the adjacency matrix gives you that information).

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<GraphEdge> getEdges() {...}
Collection<GraphNode> getVertices()  {...}

int getEdgeCount() {...}
int getVertexCount() {...}

String depthFirstTraversal() {...}

//For a given vertex in a graph...

boolean containsVertex(GraphNode vertex) {...}

Collection<GraphEdge> getInEdges(GraphNode vertex) {...}
Collection<GraphEdge> getOutEdges(GraphNode vertex) {...}

int inDegree(GraphNode vertex) {...}
int outDegree(GraphNode vertex) {...}

Collection<GraphNode> getPredecessors(GraphNode vertex) {...}
Collection<GraphNode> getSuccessors(GraphNode vertex) {...}

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

//Given two vertices in a graph...
E findEdge(GraphNode v1, GraphNode v2) {...}
boolean isPredecessor(GraphNode v1, GraphNode v2) {...}
boolean isSuccessor(GraphNode v1, GraphNode v2) {...}

//Given an edge in a graph...
GraphNode getSource(GraphEdge directed_edge)
GraphNode getDest(GraphEdge directed_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 4: Implement an Efficient Update Mechanism for a Heap Class

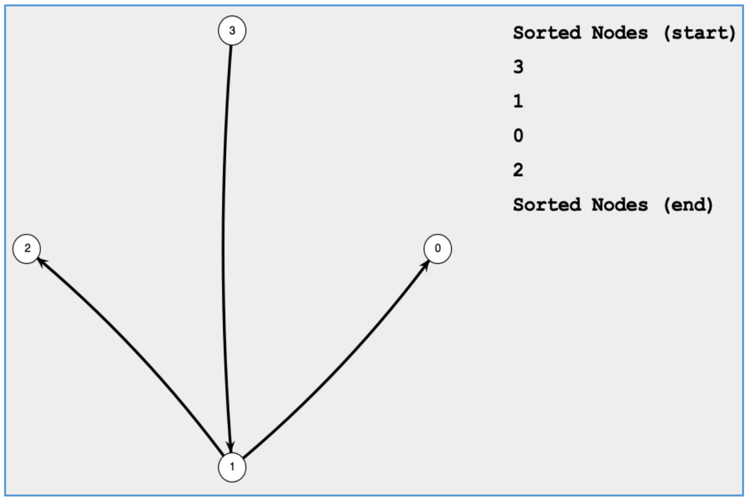
The graph simulator is going to run a topological sorting algorithm, but this requires a priority queue with the ability to *update* the priority of individual items. If you look at your textbook, it gives information in Chapter 21 (section 4) about the "decrease key" operation. We want a similar update() 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.

The only way to do this efficiently is to use a map to map heap items to indexes so that you know where to start the update (in average case O(1) time). Without this, update() would be O(n) and not O(lg n). Therefore, before you implement updating, you need to integrate a map into the Weiss code. Whenever an item is placed, moved, or removed from the heap, the map should be updated to reflect the item's new index (or remove it from the map in the case of removal from the heap). This is actually similar to what we have in Project 3. You are welcome to use the JCF HashMap class for this, or you may use your HashMap implementation from previous projects.

Task 5: Implement Topological Sorting Algorithm in the Simulator

5.1 Topological Sorting

A topological sorting of a directed graph is an ordering of all vertices of the graph such that if there is an edge from node **`u`** to node **`v`**, then **`u`** comes before **`v`** in the ordering. For example, the graph to the right has 4 vertices and 3 edges (3→1, 1→0, and 1→2). One valid topological sorting of these 4 nodes is (3, 1, 0, 2), as shown in the picture.



```
Sorted Nodes (start)
3
1
0
2
Sorted Nodes (end)
```

Topological sorting has multiple algorithms and many applications. For this project, we will simulate Kahn's algorithm for topological sorting in a directed graph. Below is the list of steps we simulate to construct a list **`L`** of ordered vertices:

- Start with an empty list **`L`**.
- Repeat while we still have nodes in graph with no incoming edges:
  - Remove a vertex **`v`** with no incoming edges; use node ID to break the tie. Append **`v`** to **`L`**.
  - "Remove" all outgoing edges from **`v`** and move on to the next round.
    - This will reduce the number of incoming edges of **`v`**'s successors and potentially generate more nodes with no incoming edges.
    - *In simulation*, we do not actually delete edges but we maintain a counter for each node for its incoming edges and update the counter in this step.
- At the end of the loop, if all nodes are added to **`L`**, we complete the topological sorting; otherwise, the graph has at least one cycle and cannot sort all nodes in the topological order.
  - *In simulation*, we still process the remaining nodes (those with incoming edges) one by one until we are done with all vertices. But we will not remove edges for those nodes and we mark them with a red color to indicate topological sorting is not applicable to them.
  - *In our simulator*, an optional simplification can be applied to the original graph which will remove all edges that points from a lower-ID node to a higher-ID node. This will ensure the simplified graph is a DAG and a topological sorting can always be performed.

5.2 Overview of Simulation

The simulator would show the sorting algorithm step by step. **`TopologicalSort`** (in **`TopologicalSort.java`**) will provide the steps for the algorithm, and the main framework is mostly written, but there are some key pieces missing! You're going to implement a few required methods to get it working.

An overview of some key parts of **`TopologicalSort`** class is given below to get you started. Make sure to check the actual file (**`TopologicalSort.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 some methods to make this work
void doNextStep() {...}

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

//--------------------------------------------------------
//You need to complete the methods below
//--------------------------------------------------------

//find out which node has the lowest indegree and highlight it
public void highlightNext(){ ... }

//find the node with the lowest indegree and process it
public GraphNode selectNext (){ ...     }

//update the indgree of the successors of minNode if applicable
public void updateSuccessorCost(GraphNode minNode){ ...}

//simplify the graph by removing all edges that starts with a lower ID node and
//ends with a higher ID node
public boolean simplify{ ... }
```

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`**).

<u>Hints and Notes</u>

- You are not responsible for making the algorithm work if the user edits the graph while sorting 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.

Project Schedule

- You've got 2.5 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:
  - 04/12-04/14: ______________________________ (first week)
    - Suggestions: Task 1 and Task 2
  - 04/15-04/16: ______________________________ (first weekend)
    - Suggestions: Start Task 3, complete JavaDocs for All Classes
  - 04/17-04/21: ______________________________ (second week)
    - Suggestions: Finish Task 3 and start Task 4
  - 04/22-04/24: ______________________________ (second weekend)
    - Suggestions: Finish Task 4 and Task 5
  - 04/25-04/28: ______________________________ (third week)
    - Suggestions: Thorough Testing and Debugging
  - 04/29-04/30: ______________________________ (third weekend)
    - Suggestions: Turn in early for Extra Credit

Testing

There is limited testing code in main methods in the provided template files. You can use command like "`java ThreeTenGraph`" or "`java WeissPriorityQueue`" 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...