SoftEng 281: Object-Oriented Programming
Assignment 4 –Solving Graph Problems (20% of final grade)
Due: 9:00pm, 2nd June 2021 (Week 12)

Partha Roop, Valerio Terragni

Learning outcomes

The purpose of this assignment is to target the following learning outcomes:

- You will be able to represent Graphs using object-oriented concepts.
- You will be able to implement common data structures like LinkedLists and Stacks.
- You will be able to improve your understanding of *unit testing* with JUnit.
- You will be able to implement search operations on Graphs using algorithms such as Breadth-First Search (BFS) and Depth-First Search (DFS) and understand the difference between BFS and DFS.
- You will be able to implement an algorithm for finding the shortest path in a graph using Dijkstra's shortest path algorithm.
- You will be able to perform OO design using the Model View Controller (MVC) design pattern.

1 Introduction

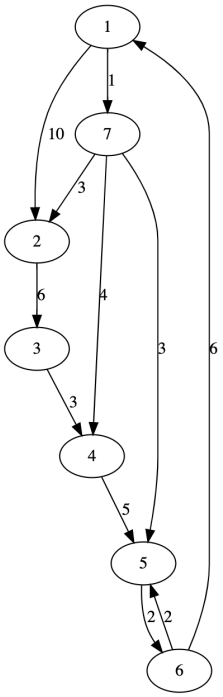
In assignment 3 the information about the set and relation were stored as graphs. Graphs are mathematical objects of the form $< V, E >$ where V denotes a set of vertices (nodes) and E denotes a set of edges. Graphs are used in multitude of applications. A very prominent one is GPS navigation, where the destinations in a city, for example, could be represented as the vertices and the edges between them represent the different paths. Google maps uses graphs for building transportation systems, where intersection of two (or more) roads are vertices and the road connecting two vertices is an edge. Google maps calculates the shortest path between two vertices to find the shortest route. Likewise, the friends suggestion algorithm of Facebook uses graph theory, where nodes are users and edges between two users represents a "friendship" relation. In the Web, web pages are considered to be the vertices. There is an edge from a page p_1 to another page p_2 if there is a link of page p_2 on page p_1. This is the basic idea behind the famous Google Page Ranking Algorithm.

In this assignment we consider Graphs that are directed, fully connected, weighted and rooted, which are defined below.

Directed : A graph that is made up of a set of vertices connected by directed edges.

Fully connected : A graph whose nodes have at least one in-comming or one out-going edge. For example, the edge $a \rightarrow b$ is an out-going edge for node a, and an in-comming edge for node b. Moreover, a fully connected graph is a graph in which it is possible to get from every vertex in the graph to every other vertex through a series of edges, called a path.

Weighted : A graph whose edges have been assigned weights, where a given weight is an integer greater than zero. As such, each edge has a source and a target vertex. For example, $a \rightarrow b$, a is the source and b is the target. This edge is an out-going edge for node a.



(a) Graph visualization.

```
//1,2,3,4,5, 6, 7
digraph testgraph{
   1 -> 2 [ label="10"];
   1 -> 7 [ label="1" ];
   2 -> 3 [ label="6" ];
   3 -> 4 [ label="3" ];
   4 -> 5 [ label="5" ];
   5 -> 6 [ label="2" ];
   6 -> 1 [ label="6" ];
   6 -> 5 [ label="2" ];
   7 -> 2 [ label="3" ];
   7 -> 5 [ label="3" ];
   7 -> 4 [ label="4" ];
}
```

(b) Textual representation.

Figure 1: An example of rooted, weighted, directed graph, which is fully connected

Rooted: A graph in which one vertex has been distinguished as the root. For simplicity, we assume that the root is the source node of the first edge in the Graph. For example, in the Graph of Figure 1b the root is the node "1" because the first edge is $1 \rightarrow 2$.

Consider the example graph shown in Figure 1. Note that the set of vertices V can be considered as the set using which a relation may be created. In this example $V = \{1, 2, 3, 4, 5, 6, 7\}$. The relation captures the edges of the graph. In this example the adjacency relation between vertices may be captured as a relation $R = \{(1, 2)(1, 7)(2, 3)(3, 4)(4, 5)$ $(6, 1)(6, 5)(7, 2)(7, 5)(7, 4)\}$. We can consider the weights being stored in another set called $W = \{10, 1, 6, 3, 5, 6, 2, 3, 3, 4\}$. A question then arises, how to associate the correct weight with an edge? One approach we may adopt is to consider V, R, W as ordered sets. Then, the position in a set can be used to determine the association between an edge and the associated weight. For example, the weight of the edge `(1,2)` is `10` and likewise, the weight of the edge `(7, 5)` is `3`.

Such a relation can be also represented using a specific text format in Figure 1b such as the dot format. Using a popular tool for graphs called Graphviz, a graph is visualised as in Figure 1. With `label`, we can specify the weight of the edge. We assume that edges are positive integers. There is a web application for the Graphviz tool at `www.webgraphviz.com`. The input file can be visualised for manual validation of your GraphCalculator program. Try to visualize the sample input files in the test-cases folder. This can be done simply by copying the entire text in the file to the Graphviz tool.

It is important that the first line in Figure 1b must capture all the nodes in the graph. This is clear because the relations are described only using the existing nodes. However, it is possible that the input file is invalid. For this assignment, *you can assume that only the correct input files are tested during the marking process*.

2 Files provided

This program uses the architectural design pattern, Model View Controller (MVC), and has the following classes. The `GraphUI` class reads a file in dot format. It also has a method to read user commands over this file. An example of an entity class is `Graph`. As an exercise, try to identify which group the other classes belong to.

The `GraphContrtol` is the main controller which has an `execute()` method that computes Graph operations by passing appropriate messages.

`src/main/java/nz/ac/auckland/softeng281/a4/GraphUI.java` This is the view class in this program using MVC. The GraphUI class reads a file in dot format. It also has a method to read user commands over this file. *You should not modify this class*.

`src/main/java/nz/ac/auckland/softeng281/a4/GraphControl.java` . This is the controller class that has a execute() method that joins all the classes by passing appropriate messages. *You should not modify this class*.

`src/main/java/nz/ac/auckland/softeng281/a4/Edge.java` A class that represents edges in the graph *You should not modify this class*.

`src/main/java/nz/ac/auckland/softeng281/a4/Node.java` A class that represents nodes in the graph *You should not modify this class*.

`src/main/java/nz/ac/auckland/softeng281/a4/Path.java` A class that represents a Path in the graph *You should not modify this class*.

`src/main/java/nz/ac/auckland/softeng281/a4/InvalidPositionException.java` An exception for invalid positions *You should not modify this class*.

`src/main/java/nz/ac/auckland/softeng281/a4/NodesStackAndQueue.java` A data structure on objects of type Node that should implement both stack and queue operations. *You should change this class for Task 1*.

`src/main/java/nz/ac/auckland/softeng281/a4/EdgesLinkedList.java` A singly linked list data structure on objects of type Edge that should implement linked lists operations. *You should change this class for Task 2*.

`src/main/java/nz/ac/auckland/softeng281/a4/Graph.java` A class that represents a Graph using an `adjacencyMap`, that for map each node that has at least one out going edge with a `EdgesLinkedList` containing all of its outgoing edges. Note that all nodes with only in-coming edges in the Graph are not keys in the `adjacencyMap`. *You should change this class for Tasks 3, 4*.

The `test` folder contains four test classes:

`src/test/java/nz/ac/auckland/softeng281/a4/NodesStackAndQueueTest.java` This Java file will contain the test cases that you implement for NodeStackAndQueue class.

`src/test/java/nz/ac/auckland/softeng281/a4/EdgesLinkedListTest.java` This Java file will contain the test cases that you implement for EdgesLinkedList class.

`src/test/java/nz/ac/auckland/softeng281/a4/GraphTest.java` This Java file will contain the test cases that you implement for Graph class.

`src/test/java/nz/ac/auckland/softeng281/a4/AllTests.java` , which runs all test cases inside the other three test classes.

The `testcases` folder contains some example files.

`Makefile` This Makefile will allow you to build and run this assignment. You can either build and run the code in your machine via Eclipse or via command line. You can also run your project in replit.com. Ultimately, once you finish your project and before you submit, you will want to **test your project using the command line in replit.com** as this is how your project will be marked. The main commands relevant to this file are:

make dependencies Finds for the commands java and javac, if it does not find them it triggers an error `make: *** [dependencies] Error 1` . In such a case please follow the course help videos to properly setup your machine.

make clean Removes the Java binary files (*.class) in the bin folder.

make build Compiles the Java classes.

make run Runs the *Graph* application. In Eclipse it is equivalent to run `GraphControl.java`.

make test-nodes Compiles the Java classes and runs the NodesStackAndQueueTest class. In Eclipse it is equivalent to run `NodesStackAndQueueTest.java`.

make test-edges Compiles the Java classes and runs the EdgesLinkedListTest class. In Eclipse it is equivalent to run `EdgesLinkedListTest.java`.

make test-graph Compiles the Java classes and runs the GraphTest class. In Eclipse it is equivalent to run `GraphTest.java`.

3 Tasks

Before proceeding with the tasks, carefully review the code and try to run the application (`make run` or in Eclipse run the class `GraphControl`). The console will ask you to insert a command

```
---------------------------------------------------------------------
The Graph Calculator. To know available commands, please type 'help'
---------------------------------------------------------------------
```

You need to first open a file contained in the `testcases` folder. For example, you can open the `g1.txt` file which is the example in Figure 1. Then you can give the command "`list`", which will show the content of the graph.

```
---------------------------------------------------------------------
The Graph Calculator. To know available commands, please type 'help'
---------------------------------------------------------------------
>>open g7.txt
The command is open g7.txt
The current directory is /Users/proo003/code/SE281-A4
The full path name is: /Users/proo003/code/SE281-A4/testcases/g7.txt
>>list
The command is list
The set elements are: {1,2,3,4,5,6,7}
The relational elements are:
   {(1,2)(1,7)(2,3)(3,4)(4,5)(5,6)(6,1)(6,5)(7,2)(7,5)(7,4)}
The weight elements are: {10,1,6,3,5,2,6,2,3,3,4}
>>
```

You can type "`search weight`", where the value of `weight` is a positive integer. Likewise, "`search source target`" may be used to determine the weight of an edge from node `source` to node `target`. This is shown in the listing below.

```
---------------------------------------------------------------------
The Graph Calculator. To know available commands, please type 'help'
---------------------------------------------------------------------
>>open g7.txt
The command is open g7.txt
The current directory is /Users/proo003/code/SE281-A4
The full path name is: /Users/proo003/code/SE281-A4/testcases/g7.txt
>>list
The command is list
The set elements are: {1,2,3,4,5,6,7}
The relational elements are:
   {(1,2)(1,7)(2,3)(3,4)(4,5)(5,6)(6,1)(6,5)(7,2)(7,5)(7,4)}
```

```
The weight elements are: {10,1,6,3,5,2,6,2,3,3,4}
>>search 4
The command is search 4
The edge searched having weight 4 is: 7-->4
>>

------------------------------------------------------------------------
The Graph Calculator. To know available commands, please type 'help'
------------------------------------------------------------------------
>>open g7.txt
The command is open g7.txt
The current directory is /Users/proo003/code/SE281-A4
The full path name is: /Users/proo003/code/SE281-A4/testcases/g7.txt
>>list
The command is list
The set elements are: {1,2,3,4,5,6,7}
The relational elements are:
   {(1,2)(1,7)(2,3)(3,4)(4,5)(5,6)(6,1)(6,5)(7,2)(7,5)(7,4)}
The weight elements are: {10,1,6,3,5,2,6,2,3,3,4}
>>search 6 1
The command is search 6 1
Given the edge from source 6 target 1 has weight: 6
>>
```

One of the key tasks, in addition to searching, is the requirement to implement Dijkstra's shortest path algorithm using the command "path source target" from node source to node target. This command then returns the shortest path between two nodes in the graph, provided the target node is reachable from the source, as shown in the listing below.

```
------------------------------------------------------------------------
The Graph Calculator. To know available commands, please type 'help'
------------------------------------------------------------------------
>>open g7.txt
The command is open g7.txt
The current directory is /Users/proo003/code/SE281-A4
The full path name is: /Users/proo003/code/SE281-A4/testcases/g7.txt
>>list
The command is list
The set elements are: {1,2,3,4,5,6,7}
The relational elements are:
   {(1,2)(1,7)(2,3)(3,4)(4,5)(5,6)(6,1)(6,5)(7,2)(7,5)(7,4)}
The weight elements are: {10,1,6,3,5,2,6,2,3,3,4}
>>path 6 2
The command is path 6 2
The shortest path is: 6 -> 1 -> 7 -> 2 cost: 10
>>
```

3.1 Task 1 - NodeStackAndQueue.java

Implement the methods `isEmpty()`, `push(Edge)`, `Edge pop()`, `Edge peek()`, `append(Edge)`. **The only data structure that you are allowed to use in NodeStackAndQueue is `java.util.ArrayList`.**

3.2 Task 2 - EdgesLinkedList.java

Implement the methods `prepend(Edge)`, `append(Edge)`, `Edge get()`, `insert(int, Edge)`, `remove(int)`, `int size()`. **You are not allowed to use any other pre-defined data structures in Java i.e. you cannot use any other data structure including the data structures in `java.util.*` except the one specified above.**

3.3 Task 3 - Graph.java find methods

Implement the methods `isNodeInGraph(Node)`, `findEdgeByWeight(int)` and `findWeightByEdge(Node, Node)`, please read the JavaDoc for the instructions and what are the allowed data structures for each method. **Note that you cannot add new instance fields to the `Graph` class.**

3.4 Task 4 - Shortest Path

You need to implement the Dijkstra's shortest path algorithm inside the method `Path computeShortestPath(Node source, Node target)` of `Graph.java`. You are allowed to use any `java.util.*` data structure you want and you may create as many helper methods are you need. We assume that the method `Path computeShortestPath(Node source, Node target)` is always invoked with source and destination nodes that are inside the Graph. **If there are shortest paths with same weight just return any one of them. In the implementation we assume that `Integer.MAX_VALUE` represents infinity in Java**.

Important:

- You should write at least two JUnit test cases for each method that we ask you to implement. You might loose up to 20% marks if you do not write enough JUnit tests.

  You can use TDD principles when writing tests and code. However, in this assignment we will not give you point if you used TDD.

- You cannot change add or remove any lines of code in the classes that have

```
//*******************************
//YOU SHOULD NOT MODIFY THIS CLASS
//*******************************
```

- For the other classes that you need to implement you can add all helper methods you want, but do not change the signature of existing methods.

- If you do not comply with the rules of each tasks (e.g., you use data structures that are not allowed) you will loose all marks for that specific task, even if all the test cases are passing.

- You cannot change the constructor of Graph class.

Important: code style

In this assignment your code will **be marked** for good programming practices, you should follow standard Java naming conventions and code style. In particular:

- **method names:** Methods should be verbs, in camel case with the first letter lowercase, with the first letter of each internal word capitalised (e.g., `decideAction`).

- **variable names:** Variable names should be nouns in camel case with the first letter lowercase, and with the first letter of each internal word capitalised (e.g., `initialAmount`). Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed. Variable names should be short yet meaningful. One-character variable names should be avoided except for temporary "throwaway" variables (e.g., iterator `i` of a for loop).

- **correct indentation and brace placement:** You should enforce correct indentation of your Java classes. Eclipse can help you with this (Source -> Format).

- **code comments:** Comments in the code should give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. Do not comment every single line of code (!), only where necessary.
- **avoid duplicated code:** If you find yourself copy-and-pasting your code in different parts of your Java class, consider creating a `private` (`protected` if you want to write unit tests for it) method to avoid duplicated code and promote code reuse. This improves maintainability because when code is copied, bugs need to be fixed at multiple places, which is inefficient and error-prone.

Important: how your code will be marked

- We will not release the test cases that we will use for marking. You can add as many test cases you want but we will use our own test cases to mark your code.
- Your code will be marked using a semi-automated process. If you fail to follow the setup given, your code **will not be marked**. All submitted files must compile without requiring any editing. Use the provided tests and `Makefile` to ensure your code compiles and runs without errors. Any tests that run for longer than 10 seconds will be terminated and will be recorded as failed.
- Although you may add more methods to the classes, you must leave unchanged the signature of the original methods.
- Do not move any existing code files to a new class, file, or directory.

Marking Scheme (total 20%)

- Task1 (max score 2%)
- Task2 (max score 3%)
- Task3 (max score 5%)
- Task4 (max score 8%)
- good code style (max score 2%)

Frequent Submissions (GitHub)

You **must** use GitHub as version control for all assignments in this course, starting with this assignment. To get the starting code for this assignment, you must accept the GitHub Classroom invitation that will be shared with you. This will then clone the code into a **private** GitHub repository for you to use. This repository must remain private during the lifetime of the assignment. When you accept the GitHub assignment, you can clone the repository either to your own computer or work on it via replit.com. For the latter, a Repl badge/image is automatically added to the README.md file when you accept the GitHub invitation. By clicking on this, you will be able to run your code in the online replit.com IDE.

As you work on your assignment, **you must make frequent git commits**. If you only commit your final code, **it will look suspicious and you will be penalised**. In the case of this assignment, you should aim to commit your changes to GitHub every time you write new test cases or all your test cases pass. You can check your commits in your GitHub account to make sure the frequent commits are being recorded. **Using GitHub to frequently commit your changes is mandatory in this course.** In addition to teaching you to use a useful software development skill (version control with git), it will also protect you two very important ways:

1. Should something terrible happen to your laptop, or your files get corrupted, or you submitted the wrong files to Canvas, or you submitted late, etc, then you are protected as the commits in GitHub verify the progress in your assignment (i.e. what you did and when), and

2. If your final code submission looks suspiciously similar to someone else (see academic honesty section below), then GitHub provides a track record demonstrating how you progressed the assignment during that period.

Together, GitHub is there to help you.

Submission

You will submit via Canvas. **Make sure you can get your code compiled and running via command line** when running `make` in replit.com. Submit the following, in a single ZIP archive file:

- A **signed and dated Cover Sheet** stating that you worked on the assignment independently, and that it is your own work. Include your name, ID number, the date, the course and assignment number. You can generate and download this in Canvas, see the Cover Sheet entry.
- The entire contents of the **src** folder you were given at the start of the assignment, including the new code you have written for this assignment.
- Do NOT nest your zip files (i.e. do not put a zip file inside a zip file).

You must double check that you have uploaded the correct code for marking! There will be no exceptions if you accidentally submitted the wrong files, regardless of whether you can prove you did not modify them since the deadline. No exceptions. Get into the habit of downloading them again, and double-checking all is there.

Academic honesty

- The work done on this assignment must be your own work. Think carefully about any problems you come across, and try to solve them yourself before you ask anyone for help (struggling a little with it will help you learn, especially if you end up solving it). If you still need help, check on Canvas (if it is about interpreting the assignment specifications) or ask in the Lab help clinics (if you would like more personal help with Java). Under no circumstances should you take or pay for an electronic copy of someone else's work.
- **All submitted code will be checked using software similarity tools.** Submissions with suspicious similarity will result in an Investigative Meeting and will be forwarded to the Disciplinary Committee.
- Penalties for copying will be severe – to avoid being caught copying, don't do it.
- To ensure you are not identified as cheating you should follow these points:
  - Always do individual assignments by yourself.
  - Never show or give another person your code.
  - Keep your Repl workspace private, and do not share your Repl with anyone.
  - Never put your code in a public place (e.g. Reddit, public GitHub repository, forums, your website).
  - Never leave your computer unattended. You are responsible for the security of your account.
  - Ensure you always remove your USB flash drive from the computer before you log off.
  - Frequently commit your code to GitHub. This provides a track record of your work and will allow the teaching team to follow your footsteps as you completed your assignment. If you do not frequently commit your code, it will look suspicious.

Late submissions

Late submissions will incur the following penalties:

- 15% penalty for zero to 24 hours late
- 30% penalty for 25 to 48 hours late
- 100% penalty for over 48 hours late (dropbox automatically closes)