COMP 2522
Object Oriented Programming 1
Lab 8

Yue Wang
yue_wang@bcit.ca

Due at or before 23:59:59 on Sunday, April 2, 2023

1 Introduction

For your next lab, I'd like you to write a short program that (a) consumes a graph adjacency matrix and (b) uses depth-first or breadth-first search to determine the shortest distance between any two nodes in the graph.

2 Submission requirements

This lab must be completed and pushed to GitHub Classroom at or before 23:59:59 PM on Sunday, April 2, 2023.

Don't wait until the last minute as you never know what might happen (computer issues, no Internet, power outage, ...). Submit often, and please make sure your comments are short and clear and specific.

You must not ignore Checkstyle warnings or the code style warnings produced by IntelliJ. **You must style your code so that it does not generate any warnings.**

When code is underlined in red, IntelliJ is telling us there is an error. Mouse over the error to (hopefully!) learn more.

When code is underlined in a different colour, we are being warned about something. We can click the warning triangle in the upper right hand corner of the editor window, or we can mouse over the warning to learn more. If the warning seems silly, tell me about it, and I will investigate and possibly ask the class to modify some settings to eliminate it once and for all!

3 Grading scheme

Figure 1: This lab, like all labs, will be graded out of 5

Your lab will be marked out of 5. Consult the rubric available on D2L. On a high level, expect that marks will be assigned for:

1. (3 point) Meeting the functional requirements in this lab, i.e., does the code do what it is supposed to do?

2. (1 point) Meeting non-functional Java OOP design requirements such as encapsulation, information hiding, minimizing mutability, maximizing cohesion and minimizing coupling, making good choices about associations and thinking about Demeter, etc.

3. (1 point) Writing code that is self-documenting and well-styled that generates no warnings from IntelliJ or Checkstyle, with correct and complete Javadocs, using short and atomic methods correctly encapsulated in classes that are well defined.

4 Style Requirements

Continuing with this lab, there are code style requirements that you must observe:

1. Your code must compile.
2. Your code must not generate any IntelliJ code warnings or problems.
3. Your code must execute without crashing.
4. Your code must not generate any Checkstyle complaints (unless they are complains I have specifically said you can ignore).
5. Don't squeeze your code together. Put a blank space on either side of your operands, for example. I will be assessing the readability and clarity of your code.
6. All of your program classes must be in package ca.bcit.comp2522.xxx (replace xxx as required by the assignment, lab, quiz, etc.). For example, today's work should go in the ca.bcit.comp2522.labs.lab0X package.
7. All classes require Javadoc comments including the @author tag AND the @version tag. Class comments go after the package and import statements. There should be no blank lines between a class comment and the class it describes.
8. Public constants require Javadoc comments, private constants do not. Constants should be the first thing in your class.
9. Constants should be static and final, are often public, and should be followed by instance variables.
10. Instance variables have private or protected visibility, never public or default visibility.
11. Public and protected methods require Javadoc comments including @param tag(s)the @return tag, and the @throws tag where needed (we won't worry about throws until we talk about exceptions in depth).
12. A method's comment must begin with verbs describing what the method does, i.e., Calculates, Returns, Sets, Prints, etc. Note that we use present tense in Java – Returns, not Return. Prints, not Print.
13. The @return and @param tags go AFTER the description.
14. Private methods require non-Javadoc comments (the comments that start with a slash and a single asterisk).
15. Do comment complicated logical blocks of code inside methods with sparse, clear inline comments.
16. Do not use magic numbers (you must use constants instead). Remember that a magic number is any numeric literal. A constant can be local in a method (use the final keyword with it) or class-level (make it static and ALL_CAPS).
17. All method parameters that are object references must be made final (so we don't forget parameters are passed by value):
    (a) Nice to prevent erroneous assignments, and necessary if parameter is referenced by inner class, but that is perhaps a little advanced for now.

(b) References made final mean that the reference, once pointing to an object, cannot be changed to point at a different object.

18. Consider making your methods final:
    (a) Making a method final prevents subclasses (those that inherit the method) from changing its meaning.
    (b) Final methods are more efficient (the methods become inline, thus avoiding the stack and generating overhead).
19. Data and methods that work together must be encapsulated in the same class.
20. Code duplication must be minimized.
21. The values of local variables that are primitives are set when they are declared, and local variables are not declared until they are needed.
22. Every class that stores state needs an equals method, a hashCode method, and a toString method.
23. In general, we enforce a fairly hard maximum method length of 20 lines of code. Aim for 10 (excluding braces).

5 Implementation requirements

1. Create a program that determines the shortest simple (no loops) path between any two points on an undirected graph.
2. Recall that a path from node n_1 to node n_k is the sequence of nodes n_1, n_2, ..., n_k where n_{i+1} is the parent of n_i.
3. Recall that the length of the path is the number of edges in the path.
4. Create an object oriented solution. Your program must use classes, encapsulation, and responsibility-driven design to create a simple, elegant and correct program.
5. The program must ask the user for the name of a text file which contains an adjacency matrix (format described below).
6. Open the adjacency matrix and create a graph in memory from it. You may represent the graph in any way you choose. I am personally partial to Node classes that contain lists of neighbour Nodes.
7. Ask the user to identify two nodes in the graph
8. If there is a path between the two nodes, report the length and describe the sequence of nodes in the path.
9. If there is no path between the nodes, report this as well.
10. I understand you learned both depth-first search and breadth-first search in your math class. I recommend you use one of these in your implementation. There is lots of pseudo-code available to help you, particularly on wikipedia at `https://en.wikipedia.org/wiki/Depth-first_search` and `https://en.wikipedia.org/wiki/Breadth-first_search`.
11. For those of you who learn visually, I found this wonderful site too: `https://visualgo.net/en`. Lovely!
12. I have found depth-first search in particular lends itself to an elegant recursive solution:
    (a) Beginning at the source node, create a list called path, add the source node to it, and then make a recursive function call to each neighbour, passing a copy of the path to each call.
    (b) Each neighbour checks if it is the desired destination node. If it is not, make the same recursive call, first adding the node to the path and sending a copy to each call.

(c) When the destination node is reached, it should add itself to the path and return the successful route.

(d) Note that you may uncover or more paths. I would like the shortest only.

(e) Here are some examples:

(a) Suppose we have a graph with two nodes that are connected by a single edge (can you draw it?):

```
01
10
```

i. Your program must respond correctly if the user asks for the distance between node 0 and node 1: "There is a path of length 1 between nodes 0 and 1".

ii. Your program must respond correctly if the user asks for the distance between node 1 and node 0: "There is a path of length 1 between nodes 1 and 0".

iii. If the user asks for anything else, including the path from a node to itself, it is correct to reply "No path found".

(b) Suppose we have a graph with four nodes that are connected by a single edge (can you draw it?):

```
0100
1010
0101
0010
```

i. Your program must respond correctly if the user asks for the distance between the nodes, i.e., node 0 and node 3: "There is a path of length 3 between nodes 0 and 3".

ii. If the user asks for a path that does not exist, including the path from a node to itself, it is correct to reply "No path found".

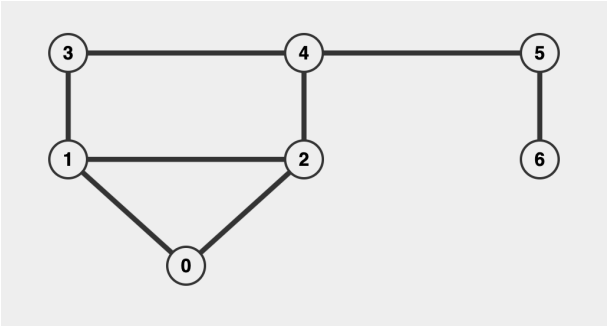

Figure 2: *A graph with seven nodes*

(c) Suppose we have a graph with seven nodes such as the example described in figure 1. Its adjacency matrix file will look like this:

```
0110000
1011000
1100100
0100100
0011010
0000101
0000010
```

i. If the user asks for the shortest path between nodes 0 and 1, your program should provide the route from 0 to 1, not the route from 0 to 2 to 1, or 0 to 2 to 4 to 3 to 1.

ii. If the user asks for the shortest path between nodes 2 and 3, your program should provide two routes, both length 2: the route from 2 to 4 to 3, and the route from 2 to 1 to 3.

That's it! Good luck, and have fun!