CS 5040: Intermediate Data Structures and Algorithms
Virginia Tech
Fall 2020
Dr. Edmison

Programming Assignment 4

Last Updated: 11-17-2020 19:19:53-05:00

Due Wednesday, Dec 9 @ 11:00 PM ET for 125 points
Due Monday, Dec 7 @ 11:00 PM ET for a 10 point bonus
Note: This project also has three intermediate milestones.
See the Piazza forum for details.

Assignment

For this project, you will implement graph algorithms for analyzing road networks. The input file is a real world dataset describing the road networks in the city of Chesapeake, VA[1]. The original data file is available online[2] for educational and research purposes, but the included file `chesapeake-roads.txt` has been reformatted for integration with the project. The dataset contains an **undirected,unweighted graph** (although the algorithms that you will develop could be generalized to weighted or directed cases). Each node represents a road intersection or dead-end, and an edge from node n1 $\rightarrow$ node n2 indicates that there is a single stretch of road connecting locations n1 and n2. The number of intersections and roads in this dataset are provided in the first non-commented line of the file.

You will implement several algorithms that were described in the OpenDSA textbook to analyze the Chesapeake road network. Some of these algorithms are defined for weighted graphs, but can still be applied by *assuming that all edge weights are one*. When deciding which item in a Node's neighbor list to visit first (for any algorithm) you should always choose the neighbor that appears first in that Node object's neighbor list (as returned by the Iterator object). For a directed graph, most of the algorithms from OpenDSA can be extended without modification.

You are provided with `Node` and `Graph` data structures in the files `Node.java` and `Graph.java`. Additional details are provided below. You must use these structures to store the road network, and you may not alter these files in any way. Note, this means that you will not be able to directly use the graph algorithm code available in the OpenDSA textbook, which expects

[1]For those of you not from around those parts: https://en.wikipedia.org/wiki/Chesapeake,$_V irginia$
[2]http://networkrepository.com/road-chesapeake.php

a different interface for the Node and Graph classes, you will be able to use the OpenDSA code as a guide.

Your task is to complete the six Java method stubs in the file `GraphMethods.java`. Each method should solves a specific graphic task. You may write as many helper methods as needed. You will upload your submissions to Web-CAT in a flat source directory containing your implementation in `GraphMethods.java`, plus any additional class files that you have written. You should **not** upload the files RoadMap.java, Graph.java, Node.java, or chesapeake-roads.txt.

Additionally, ***you should not alter RoadMap.java in anyway***. This class has the main() method, as well as the code to hand outputing the data in the expected output format. Altering this file may result in incorrect output and failures of your Web-CAT reference tests.

Design Considerations

There are two provided classes that do much of the heavy lifting:

Node class

A Node is a location, and has zero or more neighbors.

```
\\ key is the ID for this node object
Node n1 = new Node(int key);

\\An existing Node object can be modified/queried using the following methods:

\\ add n2 to the neighbor list for n1
n1.addNeighbor(Node n2);

\\ To get an Iterator object for traversing n1 neighbor list
Iterator iter = n1.getNeighbors();

\\ Get the ID/key value for n1
int id = n1.getID();

\\ print the ID value for n1 and nothing else
System.out.println(n1);
```

Graph class

A Graph is a collection of nodes.

```
\\ graphs have n nodes
\\ NOTE: node IDs range from 0 to n-1

Graph graph = new Graph(int n);

\\ adds a directed edge from node n1 to node n2
graph.addEdge(int n1, int n2);
```

Note: The Chesapeake roads graph is an undirected graph. So, if `(n1, n2)` is in G, *then* ***`(n2, n1)`*** *will also be in G.*

```
\\ returns the node object whose ID value is id int
Node n1 = graph.getNode(int id);

\\ returns the maximum number of nodes for graph
n = graph.getNumNodes();

\\ returns the number of directed edges in graph
int m = graph.getNumEdge();
```

Note: Each directed edge counts as one, so this *will return twice the number of roads in Chesapeake.*

```
\\ prints the number of nodes and edges in graph
System.out.println(graph);
```

Methods to implement

This project is based around real-world data. The problems you are being asked to solved are fairly typical of those associated with delivery routing. In our case, The **Chesapeake Pizza Bakery (CPB)** delivers pizzas from its store, located at intersection **Y**, to locations all over the city. CPB would like to find routes for its delivery drivers, spanning all intersections in the city. Your code will help CPB by providing two methods for traversing all intersections, as described below. (Note that for an unweighted graph, a traversal is equivalent to a minimal-cost spaning tree (MCST)).

breadthFirstTraversal()

Write a graph traversal method that accepts the index of the bakery location (start) and uses a BFS strategy to traverse the road network. For each node, visit its neighbors in the order that they are returned by the `Iterator` object that is returned by `Node.getNeighbors()`. You will return an iterator object over a list of Node objects, which returns every `Node` in the order that they were visited. The signature for this method must be:

```
public static List<Node> breadthFirstTraversal(Graph theGraph, int start)
```

depthFirstTraversal()

Write a graph traversal method that accepts the index of the bakery location (start) and uses a DFS strategy to traverse the road network. For each node, visit its neighbors in the ***reverse*** order that they are returned by the Iterator object that is returned by `Node.getNeighbors()`. You will return an iterator object over a list of Node objects, which returns every Node in the order that they were visited. The signature for this method must be:

```
public static List<Node> depthFirstTraversal(Graph theGraph, int start)
```

shortestDistance()

Add a method that finds the length of the shortest path from an intersection (*start*) to another intersection (*end*), in terms of the fewest road segments traversed. You will return an integer, specifying the number of edges in the shortest path.

Hint: You may be able to solve this problem with a slight modification to one of your traversal methods, but you will need to create a new auxiliary data structure for ordering nodes, for example using a priority queue, in order of their distance from start. The signature for this method must be:

```
public static int shortestDistance(Graph theGraph, int start, int end)
```

longestShortestPath()

CPB really wants to know whether its deliveries can be made on time to **anywhere in Chesapeake**. Assume that each road segment takes roughly the same amount of time to traverse. Use your solution to `shortestDistance()` and write a new method, which accepts the location of a bakery (*start*) and returns the length of the **longest** shortest path from the bakery to **any** destination in Chesapeake. You will return an integer, specifying the number of edges in the longest shortest path. The signature for this method must be:

```
public static int longestShortestPath(Graph theGraph, int start)
```

shortestPath()

Your code has been so successful CPB has expanded! They are looking for a location for their next store. They would like for it to be centrally located, so that they can easily deliver all over Chesapeake.

Provide a method that accepts a bakery location (*start*) and a delivery destination (*end*) and returns an iterator over every node (*including start and end*) along the shortest path between start and end, in the order that they were visited. **You must use Dikjstra's algorithm (assume that each edge has a weight of one)**. When choosing between neighbors, always traverse neighboring nodes in the order that they are returned by the iterator. You may use a Java priority queue as needed.

Hint 1: Keeping track of the order in which the nodes on the shortest path were visited is deceptively hard. One common solution is to keep track of the predecessor for each new node.

Hint 2: The Java priority queue API does **NOT** specify how ties will be handled (i.e., it does not resort to FIFO in the case of a tie). In order to meet the specification that neighbors must be traversed in the order returned by each iterator, you will need some mechanism to break ties. Consider using a secondary key.

The signature for this method must be:

```
public static int shortestPath(Graph theGraph, int start, int end)
```

pathExists()

CPB receives new delivery requests on their website and decides which bakery will fill these requests and how. Provide a method that accepts a list of all bakery locations in an `int` array (start[n]) and a single delivery destination (end). Select the shortest bakery in start that offers the shortest path to end and return an iterator that traverses this path, **using Dikjstra's algorithm**. When there is a tie between multiple bakeries, return the bakery that is ordered at the lowest index in the start array. The signature for this method must be:

```
public static List<Node> pathExists(Graph theGraph, int[] start, int end)
```

Invocation and I/O Files

The program will be invoked from the command-line as:

```
java RoadMap <graph-data-file> <node-count> <method-option>
```

where:

- `RoadMap` is the name of the program. The file where you have your main() method must be called `RoadMap.java`
- `graph-data-file` is the name of the file that holds the graph data. The data file provided will be called chesapeake-roads.txt
- `node-count` is the count of the number of nodes in the file you reading. The chesapeake-roads.txt file has 39 nodes.
- `method-option` is an **OPTIONAL** parameter that allows you to limit which method is executed. By default, if this parameter is not set, or set to a value other than [1-6], all six of the methods you have been asked to create are executed when RoadMap runs. This parameter allows you to specify which method to execute, so that you don't have to scroll through the output of the program each time. Values are:
  1. `breadthFirstTraversal()`
  2. `depthFirstTraversal()`
  3. `shortestDistance()`
  4. `longestShortestPath()`
  5. `shortestPath()`
  6. `pathExists()`

for example:

```
java RoadMap cheaspeake-roads.txt 39
```

is the invocation you will use to execute the full program with the `cheaspeake-roads.txt` datafile, and

```
java RoadMap cheaspeake-roads.txt 39 1
```

would execute only the `breadthFirstTraversal()` method.

Programming Standards

You must conform to good programming/documentation standards. Web-CAT will provide feedback on its evaluation of your coding style, and be used for style grading. Beyond meeting Web-CAT 's checkstyle requirements, here are some additional requirements regarding programming standards.

- You should include a comment explaining the purpose of every variable or named constant you use in your program.
- You should use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc. Use a consistent convention for how identifier names appear, such as "camel casing".
- Always use named constants or enumerated types instead of literal constants in the code.
- Source files should be under 600 lines.
- There should be a single class in each source file. You can make an exception for small inner classes (less than 100 lines including comments) if the total file length is less than 600 lines.

We can't help you with your code unless we can understand it. Therefore, you should no bring your code to the GTAs or the instructors for debugging help unless it is properly documented and exhibits good programming style. Be sure to begin your internal documentation right from the start.

You may only use code you have written, either specifically for this project or for earlier programs, or code provided by the instructor. Note that the OpenDSA code is not designed for the specific purpose of this assignment, and is therefore likely to require modification. It may, however, provide a useful starting point.

Allowed Java Data Structure Classes

You are permitted to use Java classes that implement complex data structures, **except** those that implement graphs, or graph-like structures. You may use the standard array operators, hashs, priority queues, linked lists, etc. You may use typical classes for string processing, byte array manipulation, parsing, etc.

If in doubt about which classes are permitted and which are not, you should ask. There will be penalties for using classes that are considered off limits.

Deliverables

You should implement your project using Eclipse, and you should submit your project using the Eclipse plugin to Web-CAT. Links to the Web-CAT client are posted at the class website. If you make multiple submissions, ***only your last submission will be evaluated***. There is no limit to the number of submissions that you may make.

You are required to submit your own test cases with your program, and part of your grade will be determined by how well your test cases test your program, as defined by Web-CAT's evaluation of code coverage. Of course, your program must pass your own test cases. Part of your grade will also be determined by test cases that are provided by the graders. Web-CAT will report to you which test files have passed correctly, and which have not. Note that you will not be given a copy of these test files, only a brief description of what each accomplished in order to guide your own testing process in case you did not pass one of our tests.

When structuring the source files of your project, use a flat directory structure; that is, your source files will all be contained in the project `src` directory. Any subdirectories in the project will be ignored.

You are permitted to work with a partner on this project. While the partner need not be the same as who you worked with on any other projects this semester, you may only work with a single partner during the course of a project unless you get special permission from the course instructor. When you work with a partner, then ***only one member of the pair will make a submission***. Make sure that you declare your partner in Web-CAT when submitting. Be sure both names are included in the documentation. Whatever is the final submission from either of the pair members is what we will grade unless you arrange otherwise with the GTA.

Honor Code Pledge

Your project submission must include this statement, pledging your conformance to the Honor Code requirements for this course. Specifically, you must include the following pledge statement ***near the beginning of the file containing the function main()*** in your program. The text of the pledge must include:

```
// On my honor:
// - I have not used source code obtained from another student,
//   or any other unauthorized source, either modified or
//   unmodified.
//
// - All source code and documentation used in my program is
//   either my original work, or was derived by me from the
//   source code published in the textbook for this course.
//
// - I have not discussed coding details about this project
//   with anyone other than my partner (in the case of a joint
//   submission), instructor, ACM/UPE tutors or the TAs assigned
//   to this course. I understand that I may discuss the concepts
//   of this program with other students, and that another student
//   may help me debug my program so long as neither of us writes
```

```
//   anything during the discussion or modifies any computer file
//   during the discussion. I have violated neither the spirit nor
//   letter of this restriction.
```

Programs that do not contain this pledge will not be graded.