COMPUTER SCIENCE 21A (SPRING, 2021)
DATA STRUCTURES AND ALGORITHMS

PROGRAMMING ASSIGNMENT 3

Shortest path

You've just accepted a new internship at Yahoo, in a city not yet supported by Google Maps. You want to find the fastest route to the company from your house, but you don't even have a map! All you know is that your internship probably has a Yahoo sign on its building, and that it's on an intersection. The naïve way to find the shortest path would be to just take a direct route, but you never know if the direct route has the most traffic. Another thing you could do is try every possible path, but the run-time for that would be ridiculous (no pun intended). However, being a data structures and algorithms student, you know that you can find the shortest path from your house to the Yahoo building, even considering the traffic in the city, by using Dijikstra's Algorithm.

WHAT YOU'RE GOING TO BE GIVEN

We've coded a simulator of driving around this city for you. Starting at your home, you'll be able to go north, south, east, or west, and how many seconds it takes to get to the next intersection with traffic. Each intersection will be a GraphNode object with the following methods:

- public int getX()
- public boolean hasNorth()
- public GraphNode getNorth()
- public int northWeight()
- public boolean hasSouth()
- public GraphNode getSouth()
- public int southWeight()
- public boolean hasEast()
- public GraphNode getEast()
- public int eastWeight()
- public boolean hasWest()
- public GraphNode getWest()
- public int westWeight()
- public boolean isGoalNode()
- public String getId()

Basically, each node can have a north node, south node, west node, and east node. For example, if n is a node, if n.hasNorth() returns true, then it will have another node at n.getNorth(), and a "cost" to get to that node.
Each node will have a unique string ID of 36 characters, in a format similar to this one:
5451397c-0d6e-4d7d-985f-bd627dcd2fcc

The characters will be different, but the dashes will always be in the same place. This is called a Universally unique identifier (UUID). You will use this ID to store your nodes in a hash map,

allowing for easy indexing into the heap and so you can know which nodes need to be added to the heap. You can get this id by calling node.getId().

There are also two public fields on the graph nodes (to make it easier). Their functions are described later, but they are:

public int priority
public GraphNode previousNode
public String previousDirection

To get the GraphNode of your home, you need to have the following code:

GraphWrapper gw = new GraphWrapper()
GraphNode home = gw.getHome()

Do not touch the GraphWrapper.java or GraphNode.java files.

WHAT YOU HAVE TO CODE

Min-Priority Queue

This is a data structure that allows you to add items with certain priorities, and then be able to extract the item with the lowest priority in O(1) time. We are looking for the shortest path to work, so we will give priority to exploring the locations with the shortest distance to the starting point. This is normally created using a heap data structure. You can access and change the priority of a graph node by accessing its public "priority" field. Your data structure should support the following operations:

1. insert(GraphNode g) - this should insert g into the queue with its priority.
2. pullHighestPriorityElement() - this should return and remove from the priority queue the GraphNode with the highest priority in the queue.
3. rebalance(GraphNode g) - this calls the heapify method described below.
4. isEmpty() – Returns true if the queue is empty.

To make this easier for you, each node in the graph has a public int "priority" which you can change.

You will also use a HashMap (defined below) internally your heap, mapping the graph nodes to their index in the heap array, so that you can easily index them. It will also allow you to see if a node is in the priority queue. To indicate deletion, you might want to set the value in the HashMap to -1. This will allow you to write the following method in your heap class:

1. heapify(GraphNode g) - once you change the priority of the GraphNode, you should be able to restore the heap-like property of the priority queue at g.

The other heap methods you have to make are left up to you, though it should have all the basic methods (insert and delete min, in addition to the heapify above, and any other methods

to assist those two).

HashMap

This is a data structure that allows you to add items, and then test if they are elements of the heap. This will be useful in your min-priority queue (as described above). It should implement the following methods:

1. void set(GraphNode key, int value) - check the hashmap to see if there is an Entry for the GraphNode "key", if there is, change its value to "value", otherwise, add it to the hashmap with that value.
2. int getValue(GraphNode g) - gets the value for the entry with g as the key.
3. boolean hasKey(GraphNode g) - true if the hashmap has that key.

You can make it generic if you want, but you don't have to. You should be inserting Entries into the HashMap (described below). You should also use closed hashing, and handle collisions.

You'll also have to write your own hash function. This is the hash function you will use to hash the UUIDs into the HashMap. Feel free to use any method discussed in class to generate a hash function you think will evenly distribute through the HashMap (remember that the UUIDs themselves are generated randomly).

Feel free to write other methods to assist in these three.

Entry – Entry.java

The hashmap is a way to store key value pairs. In this case, the keys are GraphNodes and the values are the index of the graph node in the heap array. These key value pairs are stored in an array in the hash map (based on the hash value of the key). In order to store keys and values together, make an Entry class which has a key and a value. Then your hash map can store an array of Entry objects.

Graph Search Algorithm – FindMinPath.java (Main Method)

In class, we have discussed the main algorithm for finding the shortest path to your internship. Here is the psuedocode to get you started, but don't feel obligated to follow each step to the T.

1. Create a priority queue Q.
2. Get the GraphNode for your home, and set its priority to 0 (home.priority = 0)
3. while Q is not empty and you haven't found your goal yet:
    a. GraphNode curr = Q.pullHighestPriorityElement()
    b. if curr.isGoalNode() - **then we found the goal** (save this as answer)
    c. else: for each neighbor of curr that you can access (use the hasNorth, hasSouth, etc. methods):
        i. int x = priority of curr + distance from curr to neighbor
        ii. if neighbor is not in the priority queue, make neighbor.priority = x neighbor.previousNode=curr, neighbor.previousDirection=[the direction you came in], and add it to the priority queue.
        iii. if neighbor is in the priority queue but x is less than the priority of neighbor, then make neighbor.priority = x, re-heapify the priority queue, and make

neighbor.previousNode = curr, neighbor.previousDirection=[the direction you came in.

if Q is empty and you haven't found the goal, then there is no path from start to the goal.

This should eventually get you to the Yahoo office, assuming a path exists. To print out the path you took to get there, you can follow the "previous" field on the answer, all the way until the start node. You can also fill the public field "previousDirection" (which is a String) during your graph tracing algorithm to help you write the path, as described in the algorithm.

Your output should be a file, answer.txt, which gives the directions from going from the start to the goal. For example, if you had to go north 3 times, then west, then north, the output file would look like this.
North
North
North
West
North

How to Debug

We have included a method for you to debug your code, on a simple example. If you create your GraphWrapper like this:

GraphWrapper gw = new GraphWrapper(true);

The console will ask you to enter the names of the files yourself. If you enter the following information to the prompts:

Enter the name of the IDs text file, then press enter.
test_ids.txt
Enter the name of the edge paths text file, then press enter.
test_edge_weights.txt
Enter the homeX, homeY, goalX, and goalY, then press enter.
0 0 1 1

Then the solution should be:
South
East

Last Thoughts and Tips

Be sure to start this early!!! There is a lot of code that needs to be written, and a lot of it is non-trivial, and will need heavy debugging to avoid NullPointerExceptions. If you give yourself time, and sketch out the data structures before you start coding, you'll run into fewer errors. I suggest you first start with the Hash Table, then the heap, then adding the hash table to the heap so that you can do the rebalancing, then creating the priority queue. After each data structure, extensively test it, to make sure it works exactly the way you want it to, so that you can be sure that it's good for the next data structure. This way, if you find a bug, you can be fairly sure it's a bug with what you're working on, not with what you've previously done. Finally, you can use the min-priority queue to implement the Dijikstra's algorithm.

Submission
Submit via Latte a zip file with all the files.

The zip file should be titled <your-username>-PA3.zip - for example, if your brandeis email is dilant@brandeis.edu, your zip file should be called dilant-PA3.zip

IMPORTANT

1. Your code should be well commented:
   - Add your name and email address at the beginning of each .java file.
   - Write comments within your code when needed. You must use Javadoc comments for your classes and methods.
2. Excluding arrays, you may only use data structures that you implement yourself.