Project for Week 2: Class Design and Graph Search

Getting Set Up

Before you begin this assignment, make sure you check Part 2 in the setup guide to make sure the starter code has not changed since you downloaded it.

In this project, you will design and implement classes to represent a graph that stores street data and implement the breadth-first search (BFS) algorithm on this graph to generate driving directions in the graph.

Open the starter code for this week by expanding the src/roadgraph folder to see the package roadgraph. You will add code to the file MapGraph.java, and you will also create additional classes as needed and place them in this package.

There are several other packages, but you don't need to worry too much about them. Except for GeographicPoint objects (in the geography package) which you will use throughout your code, and the MapLoader (in the util package) which you might use in main, you will not directly use any of the other classes in these packages.

Assignment and Submission Details

Your goals in this assignment are to design and implement a set of classes that will represent the graph structure that stores the street data and to use these classes to implement the missing methods in MapGraph.

MapGraph is a directed graph that represents the road connections between intersections. Vertices in a MapGraph are GeographicPoints (latitude, longitude pairs) that correspond to intersections or dead ends between roads, while the edges are the road segments between these intersections. This graph is directed, so each pair of nodes on a two-way street will have two edges between them: one in each direction.

Complete the implementation of MapGraph

Your goal in this assignment is to complete the implementation of the MapGraph class, implementing all of the methods described below. (Some of the methods in MapGraph will be implemented next week).

In completing this implementation, we expect you do to any or all of the following:

- Add additional classes to the roadgraph package
- Add additional methods (usually private, but perhaps public as well) to the MapGraph class
- Add member variables to the MapGraph class

In completing the implementation here are some things you should not do:

- DO NOT change the signatures or semantics of any of the methods that already exist in MapGraph (these methods are marked with TODO).
- DO NOT change any of the provided code in any other class.
- DO NOT add code or classes to any other package besides the roadgraph package. Do not add code to any of the classes outside the roadgraph package.

The methods you must support in MapGraph are listed below. In this write up we give you a brief description of what each method should do, but the starter code also provides more details about these methods. So before you begin your class design you should not only read the descriptions below, but also open the MapGraph.java file and read the comments in that file.

The required methods are:

public MapGraph() Creates a new empty MapGraph object.

public boolean addVertex(GeographicPoint location) Adds a vertex to the graph at the location specified by the GeographicPoint. Each vertex in the graph represents an intersection at a unique geographic point. Returns true if the vertex was successfully added, and false if it was not (because the vertex representing location was already in the graph) or if the parameter object is null.

public void addEdge(GeographicPoint end1, GeographicPoint end2, String roadName, String roadType, double length) Adds a directed edge between two GeographicPoints (lat, lon pairs) that are already in the graph. The edges represent streets segments, and hence they connect intersections. The roadName is the name of the road (e.g. "Main street") while the roadType is the kind of road (e.g. "residential"). The length is the length of this road segment, in km. This method

should throw an IllegalArgumentException if either of the two points are not in the graph already, if any of the arguments is null, or if length is less than 0.

Note: for this assignment this week you will not use the length of the edges.

public int getNumVertices() returns the number of vertices in the graph

public Set<GeographicPoint> getVertices() returns a set of the vertices in the graph in terms of their GeographicPoint (i.e. the locations of all of the intersections). Hint: A HashSet is a good choice for your set.

public int getNumEdges() returns the number of edges in the graph

public List<GeographicPoint> bfs(GeographicPoint start, GeographicPoint goal,Consumer<GeographicPoint> nodeSearched) Performs bfs starting at start until it reaches goal and returns a list of geographic points along the shortest (unweighted) path from start to goal. This list should include both the start and the goal points and should be ordered from start to goal.

The start and goal parameters are pretty self-explanatory, but the nodeSearched argument is probably a little mysterious. This parameter is a hook that will allow the search process to be visualized in the front-end application. Using it is simple. Every time you explore from a node (which is associated with a GeographicPoint), report that node to the consumer using the following instruction (assuming that next is the GeographicPoint associated with the node currently being explored):

```
nodeSearched.accept(next);
```

If you successfully report each GeographicPoint associated with its node as the is explored to the nodeSearched Consumer, you will be able to run the visualization of your search from the front-end application, as described below.

MapGraph also defines and implements the following method which you should NOT modify which can be useful for testing.

public List<GeographicPoint> bfs(GeographicPoint start, GeographicPoint goal) Creates a dummy Consumer object and calls your bfs method. You do NOT need to modify this method. Useful for testing!

MapGraph also defines the following methods which you should NOT implement this week.

public List<GeographicPoint> dijkstra(GeographicPoint start, GeographicPoint goal,Consumer<GeographicPoint> nodeSearched) Performs Dijkstra's algorithm to search the graph starting at the start until it reaches the goal and returns a list of geographic points along the shortest (unweighted) path from start to goal. You do NOT need to implement this method this week.

public List<GeographicPoint> aStarSearch(GeographicPoint start, GeographicPoint goal, Consumer<GeographicPoint> nodeSearched) Performs the A-Star algorithm to search the graph starting at the start until it reaches the goal and returns a list of geographic points along the shortest (unweighted) path from start to goal. You do NOT need to implement this method this week.

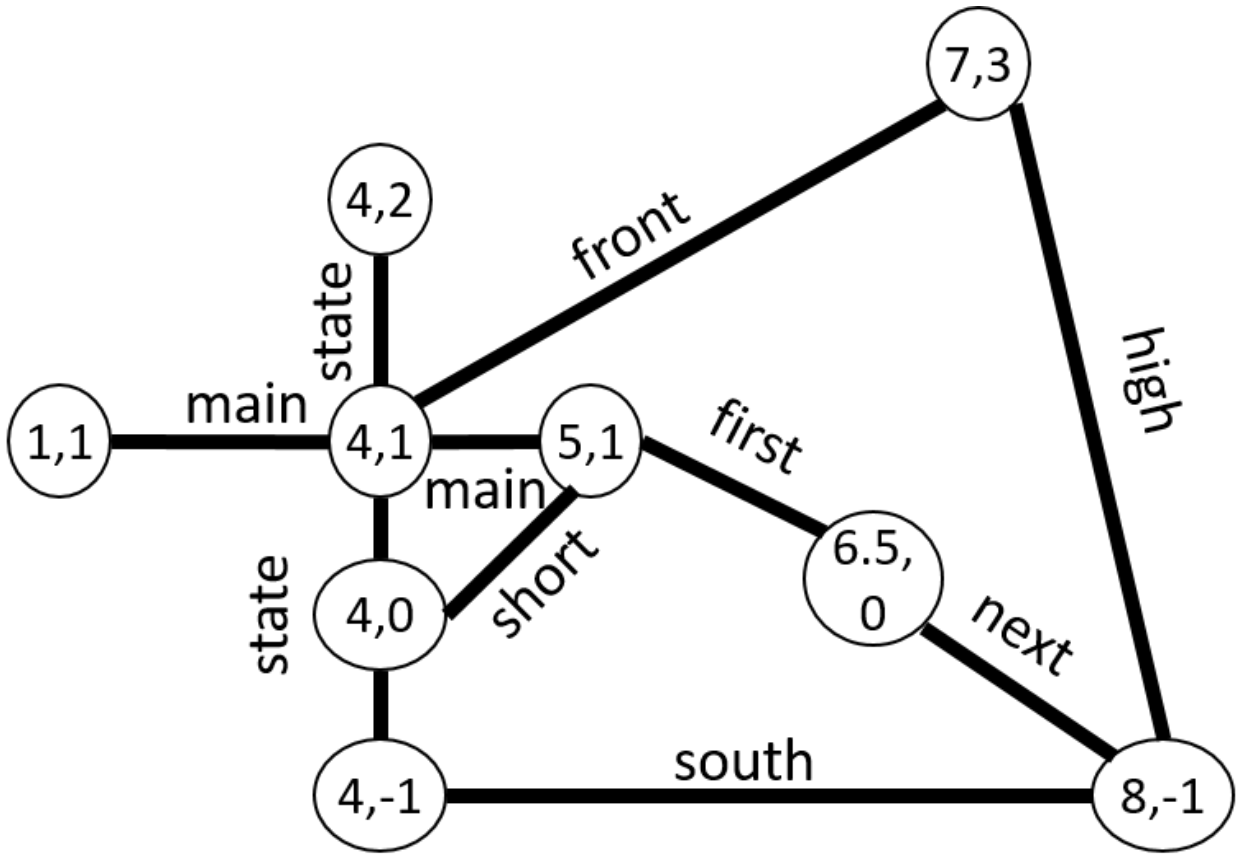
Hints

This is a big, open-ended task, and we have some support videos that will help you with this process. We also have some hints here about how to go about the development process.

1. Decide whether you will want to use an adjacency list or adjacency matrix representation by thinking about the number of neighbors each node will have. Remember that neighbors are intersections that are connected by streets. In general, each intersection is connected to only a handful of other intersections.
2. You'll probably want to include a printGraph method to help you make sure your graph representation is correct before you implement bfs. We've provided you with a small artificial graph file test.map to help you with your debugging. More information about the map this file represents is below.
3. Reading through the code that loads in the graph file can help you understand how your MapGraph class is being used. You don't have to understand the map files themselves, of the processing we do when we load the files.
4. You can use the method provided in MapGraph **public List<GeographicPoint> bfs(GeographicPoint start, GeographicPoint goal)** to debug your BFS algorithm through tester methods or in main, without having to use the provided GUI.

Testing your implementation

We have provided you with a simple test graph called simpletest.map in the data/testdata folder. The map in that graph loops like the following (all streets are two-way):



Obviously the points in that map do not correspond to real latitudes and longitudes, but you can still test well with this graph (absolute distances don't matter this week anyway, just number of hops).

Testing your implementation using the front-end (optional)

We also provide several other test graphs in the data/maps folder. These graphs contain real-world street data from around cities in California. You can use these maps to test your solutions as well, and visualize the paths in the front-end interface. Note that using the front-end application is optional, so if you cannot get it to work or do not want to use it, that's fine.

If you choose to use it, run the front-end application by running the MapApp.java file in the application package (in the src/application folder). Make sure that the search option in the interface is set to BFS. Then load the appropriate data set*, click Show Intersections, select a start and an end intersection, and click Show Route. You should see the route returned by your BFS. After you show the route, you can also show the Visualization which will show you the order in which the nodes were explored during the search, provided you added the points to the Consumer object during the search.

***If you do not see the street data file you want to load in the dropdown menu**, you might need to add it to the file mapfiles.list. First check to make sure the file itself appears in the data/maps folder. Then in the data/maps folder open the mapfiles.list file. If the file you want to load does not appear in that list, simply add it to the list, save the file and relaunch the MapApp application. You should see it in the dropdown menu now.

If you need any help with this process, go back to the project walk through and the project demonstration video from this week and last.

Submission instructions

To submit your assignment, add *all* of the files from your roadgraph package (**do not include any grader files or CorrectAnswer.java**) into a zip file called week2.zip. Use this file as your submission. This should include the MapGraph.java file as well as any other .java files you created. Remember, you should not have modified or created files in any other package.

We will run a set of tests on your implementation and provide you with some feedback about what worked and whether there were any errors. The grader feedback will help direct you to where the errors are, but we also provide you with our actual Java grading files, in the starter code. These files can be found in the roadgraph package. The grader for this week is SearchGrader.java. You can run this file to help you figure out what's going wrong.

Each of the above methods will be tested for correctness when you submit your code. However, because a large portion of this assignment was about class design, there is a peer review assignment that follows this assignment where you will get some feedback on your design.