Project: Shortest Path I Coursera

Deadline
Pass this assignment by March 6, 11:59 PM PT

Project for Week 3: Extending and Improving 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 augment your classes to support the execution of Dijkstra's algorithm to find the shorted path through a weighted graph and to support execution of the A Star Search algorithm which optimizes search for our project.

You will be working in the same code as you did in Week 2, specifically you will be completing two methods in the MapGraph.java file in the roadmap package. You may need to revise (or add) other classes to your class organization from Week 2 to support the execution of these methods.

As with Week 2, there are several other packages, but you don't need to worry too much about them. Except for GeographicPoint objects (in the geography package) 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.

If you run into problems with your Week 2 class layout and would like to either refer to our Week 2 solution or use our solution to start for Week 3, the following three files are our solutions:

Assignment and Submission Details
Your goals in this assignment are to author the two search methods we learned this week: Dijkstra's Algorithm and A Star Search. Although both methods will find you the same shortest path on a weighted graph, A Star Search will likely do so in significantly fewer steps.

The two methods you will write in MapGraph are:

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 (weighted) path from start to goal. Explored nodes are reported to the Consumer object for search visualization.

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 (weighted) path from start to goal. Explored nodes are reported to the Consumer object for search visualization.

For testing purposes, we've provided you with two methods which will create an artificial Consumer object, then call the corresponding method. These methods are called:

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

public List<GeographicPoint> aStarSearch(GeographicPoint start, GeographicPoint goal) Creates a dummy Consumer object and calls your dijkstra method. You do NOT need to modify this method.

Step 1: Author Dijkstra's Algorithm
Please refer back to the videos this week for Dijkstra Algorithm pseudo-code. You'll likely be able to use your bfs method from Week 2 as a starting point. Although there are critical changes to make going from bfs to Dijkstra in the method itself, there are two other changes you'll need to make (and read both as they are related to each other):

1. You will likely need to modify your MapGraph class, support classes, and/or add a new support class to help facilitate the search. Specifically, you'll need to be sure your current graph representation stores the length of edges between nodes. You'll also need to add a way of keeping a "current distance" from your start node to each node as your search progresses. This is the distance you'll initialize to positive infinity for all vertices at the start and update as your algorithm progresses.
2. This may be your first time working with a PriorityQueue (See Javadoc here). When you insert items into the queue, there needs to be an ordering of those items based on priority (in this case, their distance from the start node). So the class representing items you insert into the Priority Queue will need to implement the interface Comparable which requires you author the **compareTo** method.

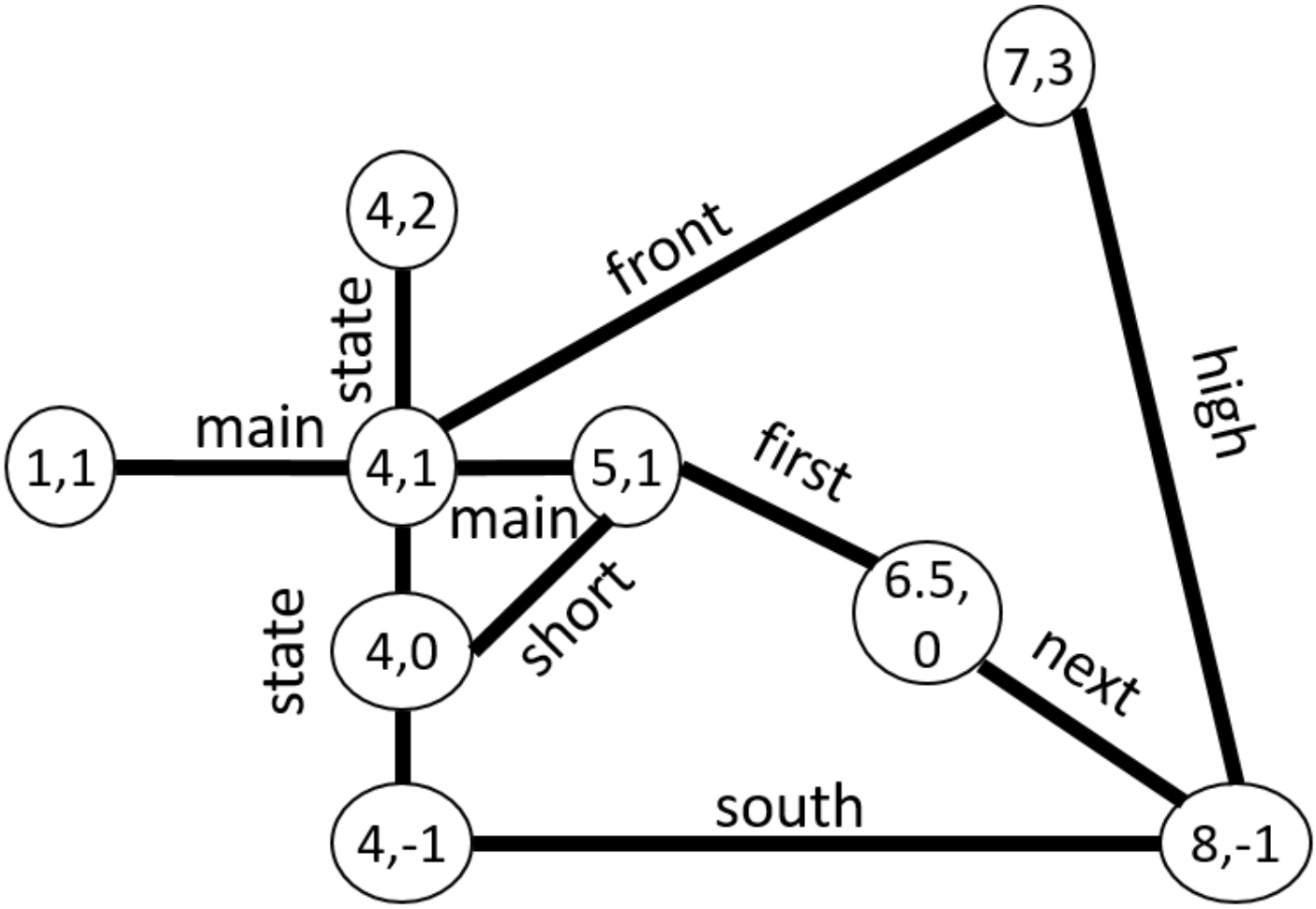
If you wish to use the visualization tool in the provided GUI interface to see how Dijkstra explores the graph, you'll need to report when you remove a node from the queue to the **Consumer** object . To do this, you'll use the Consumer method:

accept(GeographicPoint p)
For example, you might use the following line:

```
// hook for visualization
// assume variable next is the geographic point corresponding
// to the element removed from the priority queue
nodeSearched.accept( next );
```

Testing Dijkstra

After coding Dijkstra, we encourage you to test it on a number of small test cases. In the example from Week 3 below, what should Dijkstra return as it's shortest path?



The path should be (1,1), (4,1), (5,1), (6.5,0), and (8,-1). So you can run Dijkstra on the graph above (simpletest.map in the data/testdata folder) going from (1,1) to (8,-1) it should return a list corresponding to these points above. We encourage you to try additional test cases to verify your algorithm is working properly.

Note that just like in Week 2, you don't have to deal with creating a Consumer object if you want to just test in the main method of a Tester class or MapGraph. Just use the method:

public List<GeographicPoint> dijkstra(GeographicPoint start, GeographicPoint goal)

For larger-scale testing or to see the search visualization, you can run the MapApp application with the real-world road data. Just make sure to select Dijkstra as your search algorithm.

Step 2: Author AStarSearch

Please refer back to the videos this week for A Star Search pseudo-code. You'll likely be able to use your Dijkstra method as a starting point. Although there are critical changes to make going from Dijkstra to AStarSearch, you will also likely need to make an additional modification and/or add a new support class to

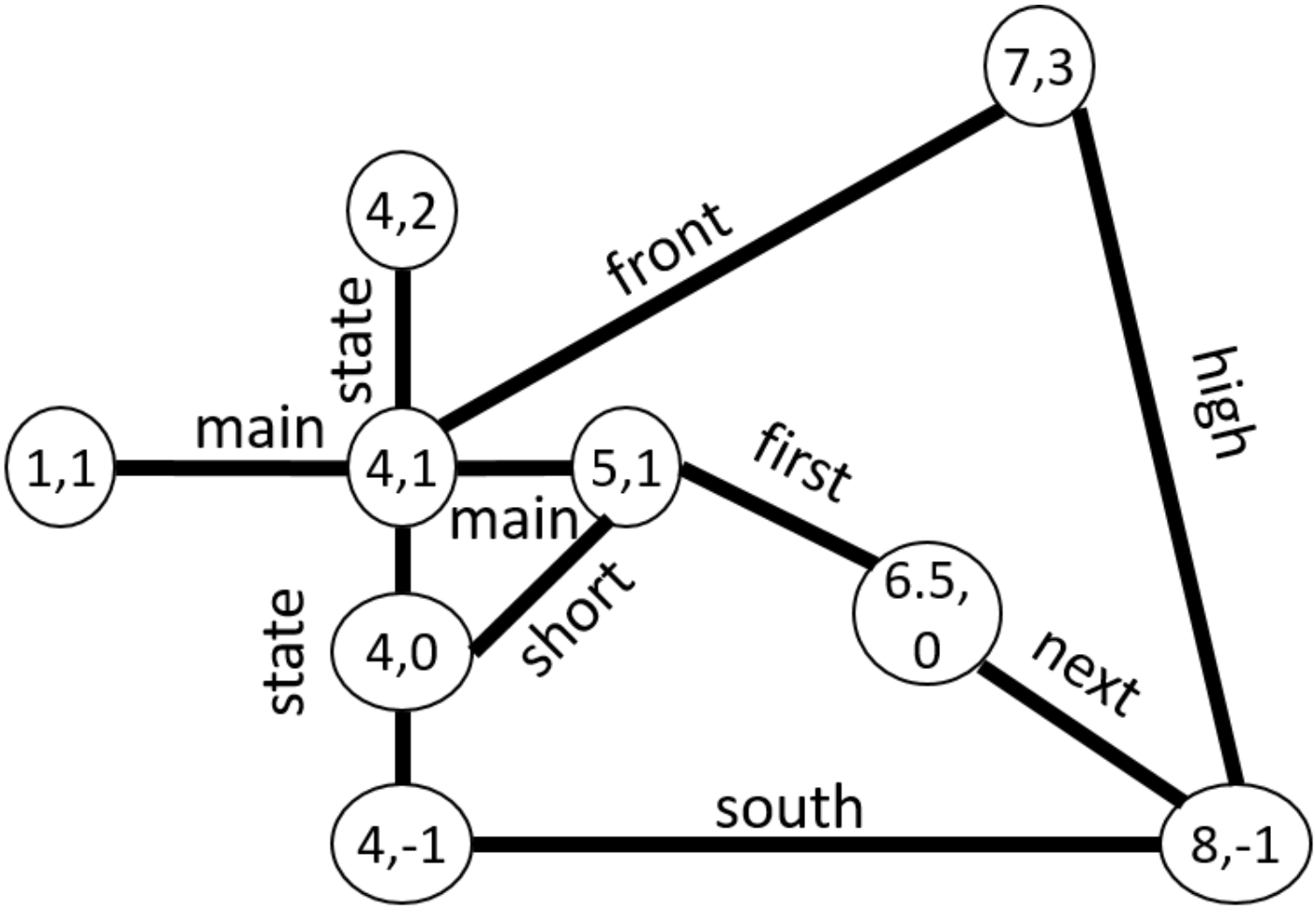
help facilitate this search. Specifically, you'll need to be able to store two distance values. You'll want to keep the actual distance from the start node (just as we did in Dijkstra) but also keep a predicted distance (actual distance + straight line distance) which will be used by the priority queue to prioritize "better" paths. **Hint:** Be sure to initialize both actual and predicted to positive infinity at the start and set both to be zero for the start node.

As with Dijkstra, If you wish to use the visualization tool in the provided GUI interface to see how Dijkstra explores the graph, you'll need to report when you remove a node from the queue to the **Consumer** object . To do this, you'll use the Consumer method:

accept(GeographicPoint p)
For example, you might use the following line:

```
// hook for visualization
// assume variable next is the geographic point corresponding
// to the element removed from the priority queue
nodeSearched.accept( next );
```

Testing AStarSearch
AStarSearch just returns the same answer as Dijkstra, so the first step is to verify that it finds the same path. The big difference between AStarSearch and Dijkstra is the nodes visited on the path to finding the solution. You can add a "count" variable to count the number of vertices or print out nodes as they are visited to observe the different behaviors. Using the same example as before, which nodes will be **visited** by Dijkstra and which will be visited using AStarSearch? (Again, we're just examining at the nodes "visited" as the method runs, both methods should return the same solution of (1,1,), (4,1), (5,1), (6.5,0), (8,-1)).



Dijkstra will visit 9 nodes in this order:

(1,1), (4,1), (4,0), (4,2), (5,1), (4,-1), (6.5,0), (7,3), and (8,-1)

AStarSearch will visit 5 nodes in this order:

(1.0, 1.0), (4.0, 1.0), (5.0, 1.0), (6.5, 0.0), and (8.0, -1.0)

We encourage you to run your code on this example and make at least one other example to illustrate the value of AStarSearch.

And once again, we encourage you to visualize this search in the MapApp application.

Submission instructions

Similarly to module 2, zip all the files in the roadgraph package that you use (except for any grader-related files) into a zipfile called "mod3.zip". Use this file as your submission for both parts 1 and 2.

Each of the above methods will be tested for correctness when you submit your code. To test the efficiency of AStarSearch versus Dijkstra, you'll need to report on the number of nodes visited during the end of week quiz.

If you find any errors in your code reported by our graders, you can run our tester code using the files

AStarGrader.java and DijkstraGrader.java in the mapgraph package.

How to submit

When you're ready to submit, you can upload files for each part of the assignment on the "My submission" tab.