Implementing Dijkstra's shortest-path algoritm

Preamble

In this assignment, you use your solution to A6 to implement Dijkstra's shortest-path algorithm. This shortest-path algorithm will then be used in the final project A8. We know your time is limited, so we have cut this assignment to the minimum while still giving you the invaluable experience of implementing Dijkstra's algorithm. Our solution is only 36 lines long.

Keep track of how much time you spend on A7; we will ask for it upon submission.

Read this whole document before beginning to code.

The due date is Sunday, 15 November, but we suggest getting it done *well* before then, so you can study for the prelim and begin thinking about A8.

Collaboration policy and academic integrity

You may do this assignment with one other person. Both members of the group should get on the CMS and do what is required to form a group well before the assignment due date. Both must do something to form the group: one proposes, the other accepts.

People in a group must *work together*. It is against the rules for one person to do some programming on this assignment without the other person sitting nearby and helping. Take turns "driving" —using the keyboard and mouse.

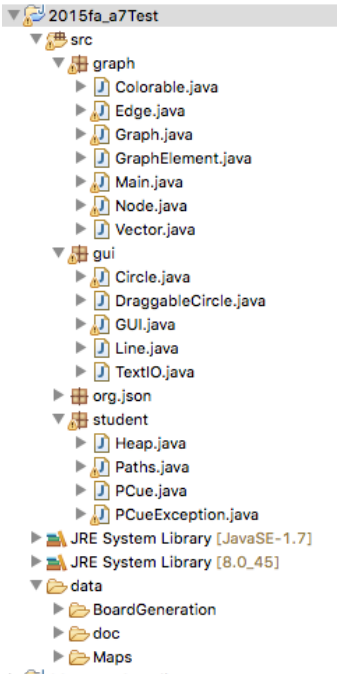
With the exception of your CMS-registered group partner, you may not look at anyone else's code, in any form, or show your code to anyone else (except the course staff), in any form. You may not show or give your code to another student in the class.

Getting help

If you don't know where to start, if you don't understand testing, if you are lost, etc., please SEE SOMEONE IMMEDIATELY —an instructor, a TA, a consultant. Do not wait. A little in-person help can do wonders. See the course homepage for contact information.

The release code

Zip file a7.zip contains a bunch of files and directories. Place them in a new project called a7 (for example) so that the package explorer pane for this project looks like the diagram to the right (your JRE System Libraries may be different). *Then replace the code in Heap.java with the code in your Heap.java —or with our solution*. Be sure to leave the package statement at the top.



Class student.Paths is the only file you have to change and submit.

Running the program

Class graph/Main contains method main. To run the program, open class Main in the Eclipse editor, select class Main in the Package Explorer pane, and choose menu item Run -> Run. Or, do it some other way. A GUI will open with a graph. You can get a new randomly-generated graph using menu item Graph -> New Random Map.You can drag the nodes of the graph around to make it easier to see a part of it.

The text at the bottom of the window tells you what to do: Click a start node, click an end node, and it shows you in red the shortest from from start to end.

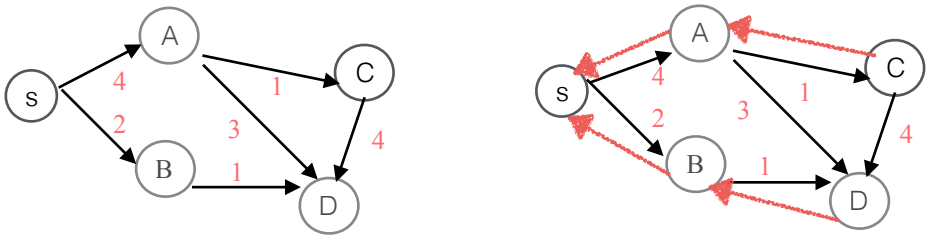
What to do for this assignment

Your job is to implement method Paths.dijkstra. It is marked "TODO". We give you everything else. The body of that method contains some comments, which you must follow.

Backpointers

The basic shortest-path algorithm calculates the shortest path from a start node to an end node. Here, we show, in the context of A7, how to extend the algorithm to also calculate the shortest path itself. Often in programming, we write a basic algorithm and then extend it to produce more information. It's a standard practice/technique.

It is difficult to maintain the shortest path from a start node S to every other node. For example, look at the diagram below and ask yourself: How, in start node S, would you store the shortest paths to all nodes? You would need to store *four* paths, from S to A, to B, to C, and to D. If the graph had 1,000 nodes, you would be storing information for 1,000 paths in S! There *must* be a better way.



We do something else. Look at the diagram to the right. The shortest path from S to D is (S, B, D). Therefore, in node D, store the *back-pointer* on this path, i.e. the *previous* node on this path: B. We show it with a squiggly red arrow. Similarly, the shortest path from S to B is (S, B), so in node B contains a *back-pointer* to S.

As one more example, the shortest path from S to C is (S, A, C), so node C contains *back-pointer* A, A contains *back-pointer* S, and S contains null as its back-pointer.

That's it! With only one extra piece of info in each node, a *back-pointer*, we can store all the information needed to give us the path from S to any node, but in reverse. To find the shortest path from S to some node D, we have to use the back-pointers beginning in node D to construct the path. That takes time proportional to the length of the path. Not bad!

The comments at the beginning of function dijkstra describe how to save back pointers as well as the distances of nodes in the settled and frontier sets.

Read this list carefully

1. Your task it to implement method Paths.dijkstra. It is marked with "// TODO …". It ***must*** be an implementation of the algorithm given on slide 32 (or so) of lecture 20 that is titled "Final algorithm". The algorithm should be refined to meet the specification and environment in which it is being implemented. See below for more info.
2. The final algorithm in the lecture slides stops when shortest paths to *all* nodes from node *start* have been determined. In addition, your algorithm should stop as soon as the shortest path from node *start* to node *end* has been determined; once that is known, the method must *not* continue to calculate shortest paths.
3. Your method will not use array L for distances. Instead, a comment at the beginning of the body of Paths.dijkstra explains how to maintain both the backpointer and the distance for each node in the settled and frontier sets.

4. Do *not* attempt to set each node's distance to ∞ (or Integer.MAX_VALUE) initially. Can you imagine google doing this whenever it has to find a route from Los Angeles to New York? Tens of thousands of nodes, if not more, would have to be dealt with! Instead, use this easier way to know that the node is in the far-off set: It is not in the settled or frontier set.
5. We have provided function Paths.buildPath, which constructs the path from the back-pointers. Use this method, once the desired node has been reached.
6. When testing/debugging, you may want to print out the frontier at each iteration. You can easily do this:

   ```
   System.out.println("frontier is: " + frontier);
   ```

7. When testing/debugging, you will want some small maps to work with. For these, try seeds: 7, 16. 1, 6, 19, 18. You can also change constant Graph.GraphGeneration.MAX_NODES to a small number. (GraphGeneration is a static class within class Graph.)
8. When testing/debugging, try very short paths first. For example, make the start and end be the same! Next, make the end be a neighbor of start.

What to do submit

In class Paths, in the comment at the top, put the hours hh and minutes mm that you spent on this assignment. Write a few lines about what you thought about this assignment. Submit on the CMS (only) file Paths.java. We know that your function dijkstra uses class Heap, but we assume you have not changed its behavior by changing its public methods. We will use *our* correct Heap in testing your function.