Implementing Dijkstra's shortest-path algoritm

1. Introduction

We have cut this assignment to the minimum while still giving you the invaluable experience of implementing the algorithm. Our solution is only 45 lines long, counting the method specification and comments and blank lines. Further, we give you a JUnit testing class. If your method passes all the tests, it should be correct, although we reserve the right to deduct points for fishy things.

Look at the grading guidelines in Section 7. Twenty (20) points ride on using good style!

Keep track of how much time you spend on A6; we will ask for it upon submission.

Read this whole document before beginning to code.

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.

We know the two people in a group are separate and must dialog over the internet. But as much as possible, try to code together. The worst thing is for one person to write a lot of code and other one not look at it. That against academic integrity principles. If one person does a bit of coding, both must talk it through, understand it, and agree on it.

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. You may not look at someone else's code from a similar assignment in a previous semester. This work must be your own (and that of your partner).

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 one-on-one help, even over the internet, can do wonders. Check zoom office hours, including those of the profs.

2. The release code

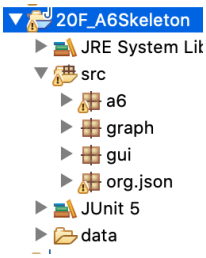
Download file `a6release.zip` from the Canvas Assignment page for this assignment and unzip it. It has two directories, `src` and `data`.

Start a new Eclipse project called, say, a6 (you can name it whatever you want).

Drag directory `src` over the project in the Package Explorer pane. You will be asked whether directory `src` should be overwritten. Yes, it should be. File *A6Test.java* will now show errors, because JUnit 5 is not on the build path.

Add JUnit 5 to the build path. One way to do this is to create a new JUnit testing class (Menu item `File -> New -> Junit Test Case`), then delete it.

Drag directory `data` over the project in the Package Explorer pane. The project should then be similar to the image on the right.



3. 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`. 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 you will see (once you complete the assignment) in red the shortest path from start to end.

4. What to do for this assignment

Your job is to implement method `A6.shortest` in package `a6`. It is marked "TODO". We give you everything else. File `A6.java` is the only file you must change and submit.

5. Submitting the assignment

In class `A6`, change the value of static field `timeSpent` (about line 25) to the amount of time you spent on this assignment. Please be careful and accurate. We use this to give you statistics on the time spent on `A6`.

In the comment at the beginning of file `A6.java`, put in your name(s) and netid(s) and write a few lines about what you thought about this assignment. Submit on the CMS (only) file `A6.java`. Your function `shortest` will use class `Heap`. The version we give you is the skeleton. You can put your own class Heap there, but if it has errors, your testing of method `shortest` won't work. We will make our solution available on Monday, 19 April, after the deadline for late A5s. Do not change `Heap`, and use *only* its public methods.

6. Making your code readable and efficient —20 points.

Your code will be tested for correctness by our grading program. But we will also grade your submission for readability and efficiency, for some of the issues that we have been talking about throughout this course. Specifically, you will lose points for the items listed in Section 7 on grading guidelines. On a positive note, being aware of these items as you program can help you manage the implementation more efficiently and successfully because your code will stay simple and readable. Thus, keep these guidelines under consideration as you program, not just when you think you are "finished".

Read this list carefully

1. Implement method `a6/A6.shortest`. It is marked with "// TODO …". It ***must*** be an implementation of the algorithm given in Section 10. The algorithm should be refined to meet the specification and environment in which it is being implemented. See below for more information.
2. The abstract algorithm in Section 10 stops when shortest paths from node `v` to *all* nodes have been determined. However, your algorithm must stop as soon as the shortest path from node `v` to node end has been determined; once that is known, the method must *not* continue to calculate shortest paths. This must be done by putting an appropriate test near the beginning of the main loop body and returning the result if the test is met. Do not do use a break statement. *It's a 15-point deduction if you use break and if the method doesn't return at this point*. Do not change the return statement that appears at the end of the method.
3. Read the grading guidelines given in Section 7.

4. We have provided function `A6.path`, which builds the path from the back-pointers. Call it *only* once the shortest path the node `end` is detected, in order to build the shortest path to node `end`. Study it to see how the path is constructed.
5. Debugging/testing. When testing/debugging, you may want some small maps to work with. For these, try seeds: 1, 6, 7, 16, 18, 19.

When testing/debugging, you may want to print out the frontier at each iteration. To do that, write a method in class `Heap` to print it out in some suitable form.

6. You can use the GUI to eyeball how your program is doing. However, we have provided complete test cases in JUnit testing class `a6/A6Test`. If your method `A6.shortest` passes those tests, you can consider the method to be correct, although we reserve the right to deduct points if we see something fishy.

7. Grading guidelines

1. 15 points. FIND ONE SHORTEST PATH

Method `shortest` must return as soon as the shortest path from node `v` to node `end` has been determined. In that case, do *not* continue to find shortest paths. You will lose 15 points if you do not do this. This should be done near the beginning of the repetend of the main loop, returning (do not use `break`) as soon as the shortest path to node `end` is known. The theorem proved about the loop invariant is important for this point.

Do *not* change the return statement that appears after the loop. There is no need to change it if you follow the directions in the previous paragraph.

2. 20 points CODE READABILITY AND EFFICIENCY

We have been emphasizing sound programming practices throughout this course. We expect your A6 submission to follow those practices. Points will be deducted for not following the instructions given below.

On a positive note, awareness of these instructions and following them *as you program* (instead of fixing these things when you think you are done) can help you program more effectively and efficiently.

A. You are implementing the abstract algorithm given in *Section 10, The Abstract Algorithm*. Your implementation should be as close as possible to the abstract algorithm in terms of structure and variable names. For example, the names `F`, `f`, and `w` should be used, since the variables appear in both the abstract algorithm and its implementation.

B. You *must* put an appropriate definition of the `HashMap` as a comment. For example, it must mention which nodes are in it.

C. Give mnemonic names to local variables. For example, if you save the `Info` object for node `f`, save it in a variable `fInfo` or something like that. Naming that variable `Mary` or `John` is not good.

D. Place local variable declarations as close to their first use as possible. Do not place them all at the beginning of the method.

E. Placement of assignments: Do not place an assignment to a variable in a loop if it has to be calculated only once and can be done before the loop. Placing an assignment outside a loop may violate point D. That's OK, efficiency is more important.

F. Don't have a method call with the same arguments in two different places. Instead, call it once and save its value in a local variable —-if it is not defined in the loop invariant.

G. Don't have the same expression, e.g. the equivalent of `d[w]= d[f] + wgt(f, w)`, in two different places. instead, evaluate it once and save its value in a local variable.

H. Don't create an `Info` object or put something into a `HashMap` when they are not needed. For example, they are not needed when processing a Node `w` that is in `S` or `F`. Instead, you can change the fields of an existing `Info` object. Remember, in a variable of type `Info`, you have a pointer to an object, not the object itself. This is a good thing here.

8. Suggestions for getting through A6 quickly

We have suggestions about how to proceed with this assignment.

1. It doesn't make sense to proceed until you firmly understand the abstract algorithm.

- (a) Study the invariant.
- (b) Understand the theorem

You should be able to state the theorem and explain what it means whenever anyone asks you!

2. Understand the data structures.

- (a) Use a min-heap for the nodes in the frontier set, with the priority being the shortest known distance to the node, i.e. `d[u]` for a node `u` in the frontier set.
- (b) Understand why we use a `HashMap`. Write a comment that explains exactly which nodes have entries in the `HashMap`. If you do not understand this, you will be lost. This is critical.

3. As you program, follow the required style instructions given in Section 7 on Grading Guidelines. Don't wait until you are "done"; be aware of the style instructions as you program. For example,

- (a) Use names `f`, `F`, and `w` so that the abstract algorithm and its implementation are close as possible.
- (b) See the same expression appearing twice? Immediately introduce a local variable to eliminate duplication.

We force this upon you because we know from experience that:

- you will get the programming done faster,
- you will be more in control,
- you will make fewer mistakes,
- you will avoid complicated uglinesses that veil understanding.

4. Maintain the invariant!

In each of the two cases (`w` is in `S` or `F`; `w` is in the far-off set), you will be changing frontier set `F` and perhaps `S` (in the abstract algorithm only). You must make sure that the definitions of `F` and the `HashMap` are maintained! If you change `F`, you probably have to change the `HashMap` too.

In the late 1970's, the Gries's hosted an Indian student, a grad student in OR. He sat in on Gries's one-credit course on the "Science of Programming". At the end, Gries asked him what he thought about the course. The reply?

"It was dull".

"What? Why?"

"All you have to do is write something to make progress toward termination and then make sure the definitions of the variables are kept true".

And that, in a nutshell, is how you have to think.

9. Backpointers

Backpointers were discussed during the lecture on the shortest path algorithm. To see a two-page introduction to backpointers, visit JavaHyperText, click the link "Shortest Path" in the horizontal navigation bars at the top, scroll down to the text under the videos and the graph, and click on "This pdf file" in the section titled *Calculating the shortest Path as well as their lengths using backpointers*.

10. The abstract algorithm

Your job in assignment A6 is to implement the abstract version of the shortest-path algorithm on an ***undirected*** graph that is given below. In doing so, your implementation should stay as close to the abstract version as possible.

1. It should have the same structure, and if a variable is used in both the abstract version and the implementation, the same name should be used.

2. Set `F` is to be implemented as a min-heap.

3. An instance of class `Info` (already in the release code in class `a6.A6`) contains (1) the shortest distance from the start node to a node and (2) the back pointer on the shortest-distance path.

4. Replace arrays `d` and `bk` and set `S` of the abstract algorithm by a single `HashMap<Node, Info>`, which should contain an entry for each node of Settled set `S` and Frontier set `F`. Do not add any other data structure.

The abstract algorithm

Here's the loop invariant:

(1) For a node `s` in Settled set `S`, a shortest `v --> s` path exists that contains only settled nodes; `d[s]` is its length (distance) and `bk[s]` is `s`'s backpointer.

(2) For each node `f` in Frontier set `F`, a `v --> f` path exists that contains only settled nodes except for `f`; `d[f]` is the length (distance) of the shortest such path and `bk[f]` is `f`'s backpointer on that path.

(3) All edges leaving `S` go to `F`.

Here's the theorem:

For a node `f` in Frontier set `F` with minimum `d`-value (over nodes in `F`), `d[f]` is indeed the length of a shortest `v --> f` path.

Here's the abstract algorithm:

```
S= {}; F= {v}; d[v]= 0;
// invariant: pts (1)..(3) given above
while (F ≠ {}) {
   f= node in F with minimum d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
       if (w is not in S or F) {
          d[w]= d[f] + wgt(f, w);
          add w to F; bk[w]= f;
       } else if (d[f] + wgt(f,w) < d[w]) {
          d[w]= d[f] + wgt(f, w);
          bk[w]= f;
       }
   }
}
```

Class `Info` already appears in the release code as an inner class of class `A6`.

11: Graph methods that you may use

Package `graph` has several classes and lots of methods and fields. In writing A6, you may use *only* the following three graph methods. Points will be deducted if you use others. *Since the graph is undirected*, these are the only ones to use.

- `n.getExits()`: Return a `List<Edge>` of edges that leave `Node n`.
- `e.getOther(n)`: `n` must be one of the nodes of `Edge e`. Return the other `Node`.
- `e.length`: The length of `Edge e`.

12. Testing and the maps in data/Maps

Class `A6Test` contains all the tests you need. If your function `shortestPath` passes all the tests, consider it correct, although we reserve the right to deduct points if we detect something fishy. Below, we explain how maps are constructed and used.

MAPS IN TEXT FORM

In the A6 project in the Eclipse Package Explorer, click on directory `data` and then on `Maps`. Most of the files in directory `Maps` contain graphs in a textual form. For example, double click on file `TwoNodeBoard.txt`; it will open in the editing pane. There, you can see that it has two nodes (node-0 and node-1) and one edge (edge-0).

All the text files in directory `data/Maps` that are undirected graphs are in this form. You can generate a file of this form for any graph that you see in the GUI as follows:

choose menu item `Graph -> PRINT Graph JSON`

It will print the graph in the console, and you can copy it and paste it into some text file.

TESTING SMALL MAPS

Open `A6/A6Test.java` in the editing pane and look at the third testing procedure, beginning on line 56 and called `test30TwoNodeOneEdge`. Look at the method body. It reads in `TwoNodeBoard.txt` and stores the graph in local variable `g`.

Below that first line, you can see three tests,

1. Test shortest path from Ithaca to Ithaca
2. Test shortest path from Ithaca to Truck Depot
3. Test shortest path from Truck Depot to Ithaca

The first three testing procedures test very small maps in this fashion.

TESTING LARGER MAPS

Look at testing procedures `test40MapTestBoard1()` on line 84. From the call `assertEquals(..., pd.size)` you can guess that it is testing a graph of size 3 (line 89) and then checking all shortest paths. (Note: all these mapes and boards are in directory data/Maps.) How do we perform that test?

Earlier, we used our solution for method `shortestPath` to create a file that contains an adjacency matrix that gives the shortest-path distance from each node to all the other nodes. For example, double click on `TestBoard1distances.txt` in the PackageExplorer pane so that that file opens in an editing pane. This file is for the graph given in file `TestBoard1.txt`. You see five lines:

1. The number of nodes
2. The names of the cities
3. Three lines giving the shortest path distances from each node to each other node —this is the adjacency matrix with ints instead of true-false values

Now look method `test40MapTestBoard1` at `A6Test.java`, about line 83. You see code that reads in the graph from `TestBoard1.txt`, reads in `TestBoard1distances.txt`, producing a `PathData` object that contains all the information, asserts that the size is 3 (just a test that we have the correct graph), and then calls `checkAllShortestPath(g, pd)` to check *your* shortest path method against the data in file `TestBoard1distances`.

Look at the body of `checkAllShortestPaths`, about line 132. It has two loops. As it states, the repetend of the inner loop checks the shortest path from node `r` to node `c`:

(1) Store in list the shortest path from the first node to the second —using your method.
(2) Check that the length of that path is what is expected.
(3) Check that the first and last nodes of the path you generated are correct.

This, of course, is not a *complete* check that the path is correct, but it's enough for us to believe the path is correct, based on how we know the path is generated.