CS 345 (Fall 18): Computer Organization

Project #5

Dijkstra's Algorithm

due at 5pm, Tue 27 Nov 2018

1 Introduction

In this project, you'll be implementing Dijkstra's Algorithm. As in previous projects, I've provided a `TestDriver`, an example implementation, and an interface which you must implement. Unlike most projects, I'm going to be pretty flexible about **how** you implement your solution. Use the tricks and algorithms that you've learned in this course - and come up with the best implementation you can.

1.1 What I Provide

I will provide a `TestDriver` class, which will:

- Interpret the command-line arguments
- Create one object - either the example code, or your code.
- Parse the input file. It will call `addNode()` and `addEdge()` as it finds the various pieces of the graph.
- After the entire graph is parsed, it will call `runDijkstra()`, `printDijkstraResults()`, and `writeSolutionDotFile()` on the object.

I will, of course, also provide the interface file (which both the example and student code must implement), as well as .class files for all of the classes used in my solution.

1.2 What You Must Implement

1.2.1 Constructor

Write a class (or likely, a set of classes) which can store a graph. You must support both graphs and digraphs. (`TestDriver` will pass a `boolean` to your constructor; if the `boolean` is true, then this is a digraph.)

1.2.2 `addNode()`

Your class must implement the `addNode()` method. This will be called when `TestDriver` finds a node in the input graph. The parameter is the name of the node. You are allowed to assume that all of the node names are unique; my example code checks to make sure that this is true, but you don't have to do this error-checking.

1.2.3 `addEdge()`

Your class must implement the `addEdge()` method. The first two parameters are nodes (from, then to) and the third is an integer, which is the weight of the edge. You may assume that both nodes have already been declared with `addNode()`; you may also assume that this is not a self-loop; you may also assume that the weight is non-negative; you may also assume that the edge has not previously been declared. (Like above, all of these things are checked in my example solution, and you don't have to check for them.)

If your graph is a digraph, then `addEdge()` represents a single directed edge (and the distinction between 'from' and 'to' is critical). However, if your graph is an undirected graph, then `addEdge()` reprsents a single undirected edge, and the distinction between 'from' and 'to' is meaningless.

1.2.4 `runDijkstra()`

Your class must implement the `runDijkstra()` method. The parameter is the name of the node to start with. By the end of this method, you should know the shortest path to every node in the graph (if some nodes are unreachable, you should know that as well). Your code **should not** write out anything during this function; just determine the solutions.

(Your code may assume that this method will be called **at most once** during the lifetime of your object.)

1.2.5 `printDijkstraResults()`

Your class must implement the `printDijkstraResults()` method. Print out the best path to each node; run the example code to find the correct format. (Pay attention to the fact that sometimes there is no path to a given node.)

You are not required to print out the lines of output in any particular order. My grading script will sort both the example output, and also your output. So long as they have the same set of lines (in any order), you will pass.

However, each line must be exactly as expected. In particular, each line prints out the list of nodes in the given path; the nodes must be in exactly the correct order.

(You may assume that there will never be a situation where there are two equally-good "best" paths to any given node. My example code checks for this, and I won't use testcases which have this issue.)

You are **strongly encouraged** to call `writeSolutionDotFile()` many times as your algorithm runs, to help you debug and visualize it.

1.2.6 `writeSolutionDotFile()`

This function must write out a `.dot` file (you choose the name) which shows the solution visually. It must meet the following requirements:

- If the input graph was an undirected graph, then this must also be an undirected graph (and use undirected edges). Likewise, if the original was a digraph, then this must be as well.
- It must include all of the nodes and edges from the original graph, including listing all of their edge weights.
- Each node must be labeled with both the node name, and also the best distance found by Dijkstra's. (Nodes which are unreachable should only list the node name.)
- The selected edges, which form the tree of paths, must be marked in some way that makes them stand out.

Note that you are **not** required to print out the nodes and edges in the same order they were in the input file. As a result, if you use `dot` to turn both the input, and also your solution, into pictures, then they likely will be "shuffled" quite a bit.

Beyond these requirements, you are allowed to add additional features. Run my example code to get some ideas for possible enhancements.

2 How to Run the Test Driver

2.1 Command-Line Arguments

The `TestDriver` accepts up to 2 arguments. The first is optional, and the second is required.

The first argument, if provided, is simply "example". If you provide this, then the `TestDriver` will run the example code; if not, it will run your code.

The second argument is the name of a node in the input graph. It is the node where Dijkstra's Algorithm will begin.

2.2 Input Graph

Provide the input graph by sending it to `System.in` (a.k.a `stdin`).

2.3 Example

To run the example code on the graph `test_01`, starting at node A, you would type (on the UNIX or Mac command line):

```
java Proj05_TestDriver example A < test_01
```

3 Limitations

Your code:

- Must **not** use any global variables (though instance variables in your classes are **unavoidable**)
- Must **not** use any already-existing PriorityQueue implementation; you must write this yourself.

4 Tips, Tricks, Hints, and Freedoms

4.1 Undirected Edges

You are allowed, in an undirected graph, to treat each undirected edge (internally) as two directed edges. **I encourage you to do this,** although I don't require it. (It will make it easier to write Dijkstra's if you only have to handle one type of edge.)

However, when you write out the solution `.dot` file at the end, if you use this trick, you must still draw out **one undirected edge**, not the two. So this is an **internal** trick, which should not be visible to the user.

4.2 Java Collection Classes

You may use Java collection and utility classes (such as `ArrayList`, `Set`, `HashMap`, etc.) as often as you want. I **encourage** you to use generics to make them easy to use - for instance, my solution used several `HashMap`s that mapped `String` to other types.

However, your Priority Queue must be **implemented by hand.** You may adapt code that you've used earlier in this class (or my solution), but it will definitely need to be updated - since our old design didn't support the ability to change a key, or to hold satellite data associated with each key.

4.3 HINT: How to Update Already-Inserted Key/Value Pairs

How do you implement a Priority Queue that allows the user to update the key after the key/value pair is inserted? There are many solutions which are possible, but I'll give you a hint: my solution had **metadata stored inside the node itself,** which was managed by the binary heap. This information made it possible for me to look at a given node, and quickly determine **where in the array that node is currently stored** - without doing a brute-force search of the array.

The downside, of course, is that this required my Priority Queue implementation to know about the internals of my graph representation. It's probably not the best solution (I would have liked to write a general, generic version), but it's Good Enough.

4.4 Poor Design will be Tolerated

While I **encourage** you to write your Priority Queue as a binary heap stored in an array (just like we've learned about in this class), you are not required to do so. You may use **any data structure,** even if inefficient. For instance, you could use a linked list, sorted by the current best-distance.

However, if you do so, expect that you will not get full credit on the testcases. A few of my testcases (not a lot of them, but some) will use **very** large graphs, and if your implementation is too slow, then your code will time out, and you will fail those testcases.

4.5 Generics

You are not required to implement your Priority Queue as a generic class, although I'll allow it if you want. (In my experience, it's easier to implement Dijkstra's Algorithm if you hard-code a little knowledge about the graph into the Priority Queue implementation.)

4.6 Classes

Your solution must include as many classes as you want; make sure to turn in all of the `.java` files to D2L.

5 Base Code

Download all of the files from the project directory

`http://lecturer-russ.appspot.com/classes/cs345/fall18/projects/proj05/`

If you want to access any of the files from Lectura, you can also find a mirror of the class website (on any department computer) at:

`/home/russelll/cs345f18_website/`

6 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).
- **Not** use any other files (unless allowed by the project spec) - since our grading script won't know to use them.
- Follow the spec precisely (don't change any names, or edit the files I give you, unless the spec says to do so).

- (In projects that require output) match the required output **exactly!** Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I **strongly recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

6.1 Testcases

You can find a set of testcases for this project at
`http://lecturer-russ.appspot.com/classes/cs345/fall18/projects/proj05/`

See the descriptions above for the precise testcase format, and also for information about how to run the two test driver classes.

6.2 Other Testcases

For many projects, we will have "secret testcases," which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcaes of your own, in order to better test your code.** You are also encouraged to share your testcases on Piazza!

6.3 Automatic Testing

We have provided a testing script (in the same directory), named `grade_proj05`. Place this script, all of the testcase files, and your program files in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It **might** also work on your Mac, but no promises!)

7 Turning in Your Solution

Turn in all of the .java files associated with your solution. Do not turn in any copies of the files that I have provided you.

You must turn in your code using D2L, using the Assignment folder for this project. Turn in only your program; do not turn in any testcases or other files.