CSCI241 Winter 2018: Assignment 4
Due March 13th at 9:59pm

Your submissions, for this and all future homework assignments, must be yours. You may chat with other students on a high-level about topics and concepts, and brainstorm using a white board, but you cannot share, disseminate, co-author, or even view, another student's code. See also the academic honesty guidelines on the syllabus. If any of this is unclear, please ask for further clarification.
If you rely on any external resources (e.g., the internet, other textbooks, etc.), you MUST cite that resource(s) in a comment portion at the top of your `ShortestPaths.java` file. Under no circumstances may you cut-and-paste entire blocks of code—if you do, you will receive an F in the course and be reported to the Dean of Students.

1 Overview

In A4, you will implement Dijkstra's Single-Source Shortest Paths algorithm using a Graph class provided to you. You will write methods in the ShortestPaths class to implement Dijkstra's algorithm, reconstruct shortest paths after the algorithm has executed, and complete a command-line program that computes shortest paths on a graph read from a text file.

As usual, please keep track of the hours you spend on this assignment and include them in your submission as a lone integer in the provided hours.txt file.

2 Getting Started

The Github Classroom invitation link for this assignment is in Assignment 4 on Canvas. Begin by accepting the invitation and cloning a local working copy of your repository as usual.

2.1 `import heap.Heap`

This assignment makes use of the priority queue you implemented in the Heap class in A3, which in turn depends on your HashTable and AList classes. Build your A3 code and copy the `heap.jar` file from `your-a3-repo/build/libs/` into `your-a4-repo/libs`. Gradle will now know to include this jarfile when compiling your A4 code, and you can use the Heap class by importing it at the top of ShortestPaths.java: `import heap.Heap`.

If you are not confident in the correctness of your A3 solution, you may download our `heap.jar` from Canvas and use that instead. Also if you're encountering strange bugs in your shortest

paths algorithm, try plugging in the correct `heap.jar` to make sure the problem isn't originating in your A3 code.

3 Testing A4

Your grade will be partly based on unit tests, but this time around you have not been provided with these tests. **It is your responsibility to test your implementation and be certain that the algorithm is implemented correctly.** ShortestPathsTest.java contains a placeholder test case to get you started writing your own tests. You will lose a few points if you do not write at least a few unit tests in ShortestPathsTest.java, although we will not be grading your test cases for correctness or comprehensiveness.

Some testing advice:

- You should test your code both using JUnit test cases and the command line program implemented in the main method.
- The two graphs given in the in-class exercise from the first Dijkstra lecture (Simple1.txt and Simple2.txt) are provided in src/test/resources/ directory. Run the algorithm by hand to determine the correct answers for these graphs and verify that your implementation arrives at the correct paths and path lengths. Keep in mind that if algorithm works on these simple graphs, it does not necessarily work on all graphs.
- You can write unit tests either by constructing Graph objects from scratch in code, or by parsing graphs from files in the `src/test/resources` directory. Opening these files at test time requires a little bit of gradle trickery, but we've included some example code and a helper method in `ShortestPathsTest.java` that finds the path for a given filename and uses `ShortestPaths.parseGraph` to read a graph from a file located in `src/test/resources`.
- Make sure your algorithm handles edge cases correctly, including behaving as specified when the destination node is unreachable. Test this using the simplest possible test cases (for example, this edge case could be tested using a two-node graph with only an edge from destination to origin).
- The BasicParser class parses a simple edge list from a text file, such as Simple1.txt, Simple2.txt, and FakeCanada.txt. Feel free to write and test using additional graph files in this format.
- See the information in data/db1b_info.txt for how to download a larger, more real-world dataset of flight information and run your algorithm on it.

4 Implementing The Algorithm

The abstract version of Dijkstra's algorithm presented in lecture is as follows:

```
/** compute shortest paths to all ndoes from
  * origin node v */
shortest_paths(v):
  S = { };
  F = {v};
  v.d = 0;
  v.bp = null;
  while  (F != {})  {
    f = node in F with min d value;
    Remove f from F, add it to S;
    for each neighbor w of f {
      if (w not in S or F) {
        w.d =  f.d + weight(f, w);
        w.bp = f;
        add w to F;
      } else if (f.d + weight(f,w) < w.d) {
        w.d = f.d + weight(f,w);
        w.bp = f;
      }
    }
  }
```

Your implementation should follow this abstract algorithm as closely as possible, but the graph representation won't match the pseudocode exactly because we have to deal with practical implementation considerations. The following design decisions have been made for you:

- So that we can efficiently find the node in F with minimum d value, the Frontier set is stored in a min-heap, using d-values as priorities. Because a Node's d-value may change, you will need to use the Heap's changePriority method to keep the priorities updated.
- Instead of the Node class having a field for `d` and `bp`, we store these things separately. The ShortestPaths class maintains a Map that associates each Node with a PathData object, which stores the distance and backpointer for a node.
- The Settled set does not need to be explicitly stored. If a Node has a PathData object associated with it, it is in either the Settled (not in the heap) or Frontier set (in the heap); otherwise it is in the Unexplored set.

5 Main Method Behavior

The main method behavior is specified in the descriptions below and in comments associated with each TODO item. In brief, the program takes 3 or 4 command-line arguments. The first two specify the file type (basic or db1b) and the filename where the graph data is stored. The third is an origin node id, from which shortest paths should be computed. If no fourth argument is supplied, the program should print all reachable nodes and their shortest path distances. If the fourth argument is supplied, it gives a destination node, and the program should print in

order the nodes along the path from the origin to the destination, followed by the length of the path.

Two sample invocations of the program are given below:

```
$ gradle run --args "basic src/test/resources/Simple0.txt A"
Graph has:
  3 nodes.
  3 edges.
  Average degree 1.0
Shortest paths from A:
B: 1.0
C: 2.0
A: 0.0

$ gradle run --args "basic src/test/resources/Simple0.txt A C"
Graph has:
3 nodes.
3 edges.
Average degree 1.0
A C 2.0
```

6 Your Tasks

You will be implementing Dijkstra's algorithm, writing helper methods to reconstruct paths and fetch path lengths, and finishing the main method, all in ShortestPaths.java. As in A3, there are not very many lines of code to write, but you cannot write them without first understanding the algorithm and the codebase you're working in.

0. Begin by looking over the early slides of Lecture 19, where the implementation details are covered. Carefully read and understand the `/** Javadoc comments */` in Graph.java, Node.java, and ShortestPaths.java. Read over the code in these files as well. When you're done, you should be able to answer questions such as:
   (a) What is the purpose of each of the following HashMaps?
      - `Graph`'s `nodes` field
      - `Node`'s `neighbors` field
      - `ShortestPath`'s `paths` field
   (b) Where is the Graph's adjacency list stored, how would you iterate over all edges leaving a given Node, and how would you get the weight of each edge?
   (c) What are types of the Values and Priorities in the min-heap storing F?
   (d) For a given Node object, where are `n.d` and `n.bp` stored?
1. Implement the `compute` method in the ShortestPaths class according to its specification.

2. Implement the `shortestPathLength` method. Notice that this method's precondition states that `compute` has been called with the desired origin node, so the `paths` field should already be filled in with the final shortest paths data.
3. Implement the `shortestPath` method. Once again, `compute` has already been called with the desired origin so you only need to use the data stored in `paths` to reconstruct the path.
4. In the `main` method, create and use an instance of ShortestPaths to compute shortest-paths data from the origin node specified in the command line arguments.
5. If a destination node was **not** specified on the command line, print a table of reachable nodes and their shortest path lengths.
6. If a destination node **was** specified on the command line, print the nodes in the shortest path from the origin to the destination, followed by the length of the path.

7 Enhancements

You'll have noticed based on ShortestPaths' main method that the codebase is able to read graphs from two types of text files, denoted basic and db1b. When a db1b file is given, the DB1BParser class parses a Graph from a csv file with data showing actual flights, their origin and destination, and the number of miles flown. The DB1BCoupon dataset I used is available at `https://www.transtats.bts.gov/Tables.asp?DB_ID=125`.

When I started making plans to see my folks in Vermont for winter break, I wondered about the shortest path from Bellingham to Burlington, Vermont. I downloaded flight data from the Bureau of Transporation Statistics for the first quarter of 2018 and ran the following command:

```
% gradle run --args "db1b data/DB1BCoupon_2018_1.csv BLI BTV"
BLI GEG MSP BUF BTV 2454.0
```

It turns out the shortest route to Vermont, in terms of miles, involves flying from Bellingham to Spokane to Minneapolis to Buffalo to Burlington, a total of 4 flights! Surely that's not the best way to fly there.

In air travel, it's rarely the case that more flights get you there faster than fewer flights—you usually want to get there with the fewest hops possible. For 5 points of enhancement credit, create an `enhancements` branch and extend your ShortestPaths class to also compute the route between a given origin and destination with the fewest flights, regardless of the total number of miles flown. For full credit, compute both the path itself and the number of edges in the path.

How you design your solution is up to you, but you should keep your modifications in ShortestPaths.java and make sure they're in a separate `enhancements` branch. Write a detailed comment at the top of ShortestPaths.java explaining

1. How to use your program (try to keep it similar to the behavior of the base assignment).
2. How you solved the problem and any design decisions you made, including any relation your solution has to algorithms we've discussed in class.

8 How and What to Submit

Check the following things before you submit:

1. Your code follows the style guidelines set out in the rubric and the syllabus.
2. Your submission compiles and runs on the command line in Linux without modification.
3. hours.txt contains a lone integer with the estimated number of hours you spent on this assignment.
4. All code, including unit tests, is committed and pushed to your A4 GitHub repository.

Submit the assignment by pushing your final changes to GitHub (`git push origin master` or just `git push`) before the deadline.

Rubric

You can earn points for the correctness and efficiency of your program, and points can be deducted for errors in commenting, style, clarity, and following assignment instructions. Correctness will be judged on both unit tests on your ShortestPaths class as well as the correct behavior exhibited by the program implemented in the `main` method.

Git Repository	
Code is pushed to github and hours spent appear as a lone integer in hours.txt	1 point
Code : Correctness	
Unit tests of methods in ShortestPaths	21
Correct behavior of the ShortestPaths main method program with no destination specified	5
Correct behavior of the ShortestPaths main method program with a destination specified	5
Code : Unit Tests	
At least a few test test are written in ShortestPathsTest.java	3
Code : Efficiency	
`ShortestPaths.compute` uses a Heap to find the node to move from F to S in O($\log n$).	5
The body of the `for` loop in `ShortestPaths.compute` runs in O($\log n$) expected time.	5
Enhancements	
Correctly computes the fewest number of hops	3
Correctly computes a path with the fewest hops	2
Clarity deductions (up to 2 points each)	
Include author, date and purpose in a comment comment at the top of each file you write any code in	
Methods you introduce should be accompanied by a precise specification	
Non-obvious code sections should be explained in comments	
Indentation should be consistent	
Method should be written as concisely and clearly as possible	
Methods should not be too long - use private helper methods	
Code should not be cryptic and terse	
Variable and function names should be informative	
Total	50 points