Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2017

Homework Set 3

Due: Start of Class, Monday, April 10, 2017

You may work alone or with a partner on this assignment. However, in order to make sure you learn the material and are well-prepared for the exams, you should work through the problems on your own before discussing them with your partner, should you choose to work with someone. In particular, the "you do these and I'll do these" approach will not prepare you for the exams.

Please submit a hard copy: print your code (consider 2-up double-sided printing), and either a typeset (preferred) or handwritten (must be legible) set of responses for the written problems. Only one submission per group is needed.

There is a significant amount of work to be done here, and you are sure to have questions. It will be difficult if not impossible to complete the homework set if you wait until the last minute. A slow and steady approach will be much more effective.

Written Problems

1. (10 points) In the fraudulent voters problem, the input consists of two arrays: `V[0...n-1]` contains the names of n voters that participated in a recent election, and `IE[0...m-1]` contains the names of m persons ineligible to vote (because of something such as a felony record, deceased, etc...) In this problem, the task is to determine how many of the voters in `V` were actually ineligible to vote. For example, for the two arrays below, 2 should be returned as the answer, because voters "T. Smith" and "J. Doe" were ineligible to vote. You may assume there are no duplicate names in the arrays.

```
V[0...7] = {"A. Turing", "T. Smith", "B. McKeon", "G. Gordon",
   "J. Doe", "M. Taylor"}
IE[0...4] = {"Z. McBeth", "J. Doe", "T. Jones", "T. Smith", "N. Wirth"}
```

You could easily write a brute force algorithm solving this problem in $O(nm)$ time. But the number of voters in both arrays is very large, and so an asymptotically more efficient algorithm is required. Write such an algorithm below, give its worst case running time, and briefly explain how you got it.

```
// Returns the number of voters that were ineligible to vote.
ALGORITHM NumFraudulentVoters(V[0...n-1], IE[0...m-1])
```

2. (15 points) You learned about d-heaps as part of the last lab. You also know about heapsort, which uses a 2-heap as an intermediate representation to sort the contents of an array. Let's consider a generalization of the heapsort idea:

- First, insert the elements to be sorted into a priority queue (PQ).
- Then, remove the elements one by one from the PQ and place them, in that order, into the sorted array.

For heapsort, the PQ is a 2-heap, but any PQ implementation would work (naive array- or list-based with contents either sorted or unsorted, a d-heap, or even a binary search tree). Depending on which underlying PQ is used, the sorting procedure will proceed in a manner similar, in terms of the order in which comparisons occur, to one of the other sorting algorithms we have studied (*e.g.*, selection sort, quicksort, *etc.*). For each of the following underlying PQ structures, state which sorting algorithm proceeds in the manner most similar to the PQ-based sort using that PQ structure, and explain your answer. Each response should be at least a few sentences long, and should discuss how the pattern of comparisons and swaps, and the resulting efficiency relates to the sorting algorithm.

1. 1-heap
2. 3-heap
3. (n-1)-heap
4. binary search tree
5. balanced binary search tree

---

3. (15 points) For the robot coin collection problem described in section 8.1 of Levitin, do the following:

a. Complete the pseudocode below that uses a recursive exhaustive search algorithm to solve it.

```
// Returns the maximum number of coins the robot can collect if she starts
// at row r and column c and only moves once cell right or down in each step
// and stops only when she reaches the lower right corner.

ALGORITHM RobotCoinCollection( r, c, C[1..n,1..m] )
```

b. Now rewrite your algorithm from above so that it uses top-down dynamic programming to get a recursive algorithm that is more efficient.

```
// Assume each entry of sols[1..n][1..m] is initialized to -1 before
// the first call.

ALGORITHM RobotCoinCollection( r, c, C[1..n][1..m], sols[1..n][1..m] )
```

c. What is the worst case running time of your algorithm in part (b)? Explain.

Dijkstra's Road Trip

Your programming task for this assignment is to develop a simplified "driving directions" system based on the mapping data you have been working with. It will be like taking a road trip with Professor Dijkstra himself!

Overview of Basic Requirements

You should use a variant of Dijkstra's Algorithm to compute shortest path from a given starting point (a graph vertex) to a given destination point. The general form of Dijkstra'a Algorithm computes the shortest paths from a starting vertex to all other vertices, but you will be able to stop one you find a shortest path to the specified destination rather than calculating the shortest path to all other places. You will also need to make sure that you can efficiently print/write the computed route in the proper order (starting point to destination point).

Once a shortest path is computed, you will need to be able to output it to the terminal in a human-readable form and to a file in a format plottable by METAL's Highway Data Examiner (HDX).

For example, if you load the `NY-all.tmg` file (available from METAL's graph repository at `http://tm.teresco.org/graphs/`), and compute a shortest path for a few nearby points: `US9/NY2` (Latham Circle). and `NY5/NY32` (next to the Times Union Center in downtown Albany), your path would traverse the following points:

```
US9/NY2, US9/NY155, US9/NY378, US9/NY377,I-90(6)/US9,
US9/US9W, US9/NY32, NY5/NY32
```

Your "human readable" output might look something like this:

```
Travel from US9/NY2 to US9/NY155
 for 0.78 along US9, total 0.78
Travel from US9/NY155 to US9/NY378
 for 2.24 along US9, total 3.02
Travel from US9/NY378 to US9/NY377
 for 2.04 along US9, total 5.06
Travel from US9/NY377 to I-90(6)/US9
 for 0.44 along US9, total 5.50
Travel from I-90(6)/US9 to US9/US9W
 for 0.87 along US9, total 6.38
Travel from US9/US9W to US9/NY32
 for 0.64 along US9, total 7.02
Travel from US9/NY32 to NY5/NY32
 for 0.34 along NY32, total 7.36
```

Your plottable data for HDX should be in a "`.pth`" file. This file format must match the following:

```
START US9/NY2 (42.748115,-73.761048)
US9 US9/NY155 (42.736832,-73.76225)
US9 US9/NY378 (42.704925,-73.754568)
US9 US9/NY377 (42.675873,-73.747659)
US9 I-90(6)/US9 (42.669562,-73.748817)
US9 US9/US9W (42.659938,-73.759975)
US9 US9/NY32 (42.654285,-73.750019)
NY32 NY5/NY32 (42.649869,-73.752787)
```

Here, each line describes one "hop" along the route, consisting of the road name of the segment (*i.e.*, your edge label), the waypoint name (*i.e.*, the label in your vertex), and the coordinates of that point. The exception is the first line, where we substitute `START`, since you don't have to take any road to get to your starting point.

These files should be given a `.pth` extension. Once such a file is created, it can be visualized by directing a browser at `http://courses.teresco.org/metal/hdx/` and uploading the `.pth` file in the file selection box at the top of the page.

Please take advantage of HDX to view the graphs themselves and your computed paths!

Starter Code

To get you started, a small collection of Java classes are provided that expand on the graph structure we worked with earlier in the semester.

In particular, the `HigwayGraph` class, and its auxiliary classes, `HighwayVertex`, `HighwayEdge`, and `LatLng`, provide much of the underlying functionality you will need. Note that most of the fields are declared as `protected`, so you can access them directly from your code as long as it is also in Java's "default" package. This is not ideal from a design perspective, but is done to simplify much of the code so you can focus on the actual algorithm implementation.

There is also a class `Dijkstra` provided that processes the required command-line parameters and sets up the `HighwayGraph` and finds references to the `HighwayVertex` objects for the start and destination of the driving directions request.

Note that there is a named constant `DEBUG` that should be used to turn on or off "debugging" output. It will be much easier to develop your Dijkstra's algorithm implementation step-by-step if you work on a small graph and print lots of information about the vertices and edges being considered at each step by the algorithm. If you encapsulate all of this kind of output inside `if (DEBUG} { ... }` blocks, you can simply set `DEBUG` to `false` when you want to turn it off.

What You Need to Add

Your task is to implement Dijkstra's algorithm to compute and report the shortest path from `start` to `dest`.

You should follow the algorithm as discussed in class, with the following modifications:

- When in debug mode, your program should print out step-by-step information about every significant step the algorithm takes (*e.g.*, add/remove priority queue entries, finding a new shortest path to add to the map/table).
- You should stop your algorithm as soon as you find the shortest path to the destination. This will be the stopping condition on your main loop. You may assume that a path exists between your starting and ending vertices, so you need not be concerned that the priority queue will become empty.
- When we traced the algorithm in class and lab, the values in the table of shortest paths found included both the last edge traversed and the total distance traversed. You will only need to keep the last edge traversed.

When your main loop terminates (because you found the shortest path to `dest`), you will print out the driving directions from `start` to `dest`.

If four command-line parameters were specified (that is, if `args.length == 4`), then `args[3]` will contain the name of a file where you should write a `.pth` file with the route that can be plotted on the map by HDX.

Implementation Details

- You are encouraged to make use of the provided classes as your graph implementation. You should not need to modify those. Check first if you believe you need to do so. These structures are already enhanced from what we studied earlier in the semester to include the information needed by this algorithm.

- Dijkstra's algorithm uses two fundamental data structures: a map/table/dictionary and a priority queue. You may write your own implementations of these, but you are encouraged to find and use appropriate implementations from the standard Java API. See the `Map` interface and various classes that implement it. When you use a generic priority queue, the objects you store must be `Comparable`, so the implementation can determine the correct order to remove highest priority items. In our case, these will be the entries that have the lowest cumulative distance from the start vertex.
- Since we will be working with "collapsed" format graphs, edges can specify intermediate shaping points to improve map accuracy and distances. The majority of your implementation can safely ignore these intermediate shaping points, since they are already considered when edge lengths are computed during graph construction. However, they will come into play if any of the edges on your shortest path being writted to a `.pth` file have such points. In that case, they should be included in your `.pth` file for mapping accuracy.

  For example, if you load the `NY-all.tmg` file and compute the path from `NY29/NY30` (in Vail Mills) to `NY30@AlgDr` (in Wells), your route includes several edges with shaping points. The `.pth` file in this case would be:

```
START NY29/NY30 (43.047397,-74.217067)
NY30 NY30@CR155 (43.051192,-74.215522)
NY30 NY30/NY349 (43.076704,-74.259381)
NY30 NY30_N/NY30A_N (43.094704,-74.279208)
NY30 NY30@HamPoiRd (43.188338,-74.198205)
NY30 NY30@BriSt (43.224055,-74.184923)
NY30 (43.245697,-74.197326) NY30@BenRd (43.255538,-74.225435)
NY30 (43.315726,-74.252729) (43.332128,-74.26981) NY30@PumHolRd (43.353382,-74
NY30 (43.363605,-74.292898) NY30@AlgDr (43.404038,-74.285989)
```

  As seen above, the shaping points are listed between the route name (*i.e.*, edge label) and the vertex label where that edge leads. The coordinates of the destination is always last on the line.

Bonus Opportunties

There are two opportunties to earn bonus points on this homework. Make your suggestions for other bonus ideas and approved ideas will be added here.

1. Directions that mention every intersection, even those where you are simply supposed to continue along in the same direction on your current road, can be a little verbose. For 4 bonus points, compress the human-readable directions to mention only those points where your route changes (*i.e.*, where the edge label changes from one segment to the next). For example, getting directions in `NY-all.tmg` from `US9/NY2` (Latham Circle) to `NY86@MirLakeDr` (Lake Placid) results in a path length of 38. However, since many of those are consecutive points from one I-87 exit to the next or along US 9, the compressed directions simplify to:

```
Travel from US9/NY2 to US9/NY7
 for 0.89 miles along US9, total 0.89
Travel from US9/NY7 to I-87(7)/NY7
 for 0.35 miles along NY7, total 1.24
Travel from I-87(7)/NY7 to I-87@14&NY9P@I-87&NY9PTrkSar@NY9P
 for 22.17 miles along I-87, total 23.41
Travel from I-87@14&NY9P@I-87&NY9PTrkSar@NY9P to
 I-87(15)/NY9PTrkSar/NY29TrkSar/NY50
 for 1.74 miles along I-87,NY9PTrkSar, total 25.15
Travel from I-87(15)/NY9PTrkSar/NY29TrkSar/NY50 to I-87(30)/US9
 for 72.23 miles along I-87, total 97.38
Travel from I-87(30)/US9 to US9/NY73
 for 2.14 miles along US9, total 99.52
Travel from US9/NY73 to NY9N_S/NY73_S
 for 11.19 miles along NY73, total 110.71
Travel from NY9N_S/NY73_S to NY9N_N/NY73_N
 for 1.64 miles along NY9N,NY73, total 112.36
Travel from NY9N_N/NY73_N to NY73/NY86
 for 13.45 miles along NY73, total 125.81
Travel from NY73/NY86 to NY86@MirLakeDr
 for 0.87 miles along NY86, total 126.68
```

2. For 2 bonus points, gracefully handle the situation where the `start` and `dest` vertices are not connected. This could happen, for example, in Hawaii, if you ask for directions between two points on different islands.

Deliverables and Grading Breakdown

The required functionality here is worth 60 points total.

- Correctness of your Dijkstra's algorithm implementation (35 points)
  - Demonstrate this by including a hard copy the debug output for the `NY-all.tmg` graph finding the path from `US9/NY2` to `NY5/NY32`, **and** another short test case of your choice. Consider printing this 2-up and double sided, as it would likely be about 7 pages for the required example alone.
- Correctness of printed human-readable directions (10 points)
  - Demonstrate this by including a hard copy of your detailed directions for the `NM-all.tmg` graph from `NM475/US84/US285` (the center of Santa Fe) to `NM150@ErnBlaRd` (the entrance to the Taos Ski Valley parking lot) **and** another test case of your choice where the shortest path has at least 25 points along the way.
- Correctness of `.pth` files containing shortest path (5 points)

  - Demonstrate this by including printout (screen captures) of your route loaded into HDX for the `usa-all.tmg` graph from `US9/NY2` (Latham Circle again) to `US41@5thAve` (downtown Naples, Florida). Include 3 screen captures: the closest view you can get that has an overview of the entire route, a zoomed-in view showing the route from its starting position to the Kingston area, and a zoomed-in view showing all of the route as it passes through Maryland. **Also** do the same for a test case of your choice where the shortest path has at least 250 points along the way.

- Documentation of your Dijkstra's algorithm implementation (10 points)
  - Include a printout of your `Dijkstra` class and any new classes you add. Again, please consider 2-up and double-sided printing.

- Bonus items: if you complete functionality for bonus credit, include results of test cases that demonstrate that the functionality is correct.