SIENAcollege
Computer Science

Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2018

Problem Set 7
Due: 4:00 PM, Friday, April 27, 2018

You may work alone or in a group of 2 or 3 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 is sure to leave you unprepared for the exams.

All GitHub repositories must be created with all group members having write access and all group member names specified in the `README.md` file by 4:00 PM, Friday, April 20, 2018. This applies to those who choose to work alone as well!

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 assignment if you wait until the last minute. A slow and steady approach will be much more effective.

Getting Set Up

You will receive an email with the link to follow to set up your GitHub repository, which will be named `ps7-yourgitname`, for this problem set. One member of the group should follow the link to set up the repository on GitHub, then that person should email the instructor with the other group members' GitHub usernames so they can be granted access. This will allow all members of the group to clone the repository and commit and push changes to the origin on GitHub.

Submitting

Your submission requires that all required code and other electronic deliverables are committed and pushed to the master for your repository's origin on GitHub. That's it! If you see everything you intend to submit when you visit your repository's page on GitHub, you're set.

For written questions, you may submit a hard copy (typeset preferred, handwritten OK but must be legible) or you may answer the questions in your repository's `README.md` file.

Only one submission per group is needed.

Backtracking Practice

Question 1:
Write a program to find solutions to the billiard ball problem from lab. It should be very similar to the n-Queens solution (use that as a guide or as a starting point). Include your code in your GitHub repository. Be sure it is well documented and that class, variable, and method names are appropriate for this problem. (20 points)

Question 2:
Using your program, show the solutions computed or indicate that no solution is found for 1 through 9 balls in the top row. Give the elapsed time for each and the number of nontrivial recursive calls made in your repository's `README.md` file. (4 points)

Question 3:
In the worst case, what is the Big O growth rate of this problem in terms of the number of balls in the top row? (2 points)

Dijkstra's Road Trip

We've looked at Dijkstra's algorithm for single-source shortest paths in class and in an earlier lab. We will again work with METAL's visualization of this algorithm and then a Java implementation that can be used on larger examples. That Java program is capable of producing output that can be visualized in METAL's Highway Data Examiner (HDX).

Your repository includes a working implementation of Dijkstra's algorithm that produces 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!

A Closer Look at Dijkstra's Algorithm

Let's experiment with Dijsktra's algorithm on some METAL graph data of the local area. In particular, we'll travel from the Latham Circle (graph waypoint `US9/NY2`, #501) to the Times

Union Center (graph waypoint `US9/NY32`, #552).

Point your browser at HDX at `http://courses.teresco.org/metal/hdx/` and load up the `siena-area50.tmg` graph (in the list as "Siena College (50 mi radius)"). Choose "Dijkstra's Algorithm" and the start and end vertices noted above. Start on a slow setting and watch the progression. Shortly after you start, pause the simulation and zoom in on the area near Siena so you can see what's happening. Resume the simulation and pause it again when you notice the destination vertex is in the priority queue (PQ).

Question 4:
Take a screen capture of the simulation at this point. Include it in your repository and give its file name as your response to this question. (2 points)

Question 5:
About how many edges are in the PQ when an edge to the destination in first added? (2 points)

Question 6:
About how many locations are in the table of shortest paths (which corresponds to the number of blue edges in the graph) found when an edge to the destination is first added? (2 points)

Now let the simulation run to completion.

Question 7:
Take a screen capture of the simulation at this point. Include it in your repository and give its file name as your response to this question. (1 point)

Question 8:
How many entries are in the PQ with cumulative distances less than the length of the shortest path to the destination and how many are greater than the length of the shortest path to the destination? (2 points)

Question 9:
Explain why and how entries whose cumulative distance is greater than the length of the shortest path to the destination get into the PQ. (2 points)

Question 10:
Identify at least one edge whose cumulative distance would be less than the cumulative distance of an edge that remains in the PQ at the end of the simulation, but which has not yet been added to the PQ. Why is it not yet in the PQ at this time? (2 points)

Now we will move on to the Java implementation of Dijkstra's algorithm. First, familiarize yourself with the code. Here are some notes:

- The `HigwayGraph` class, and its auxiliary classes, `HighwayVertex`, `HighwayEdge`, and `LatLng`, provide much of the underlying functionality. Note that most of the fields are declared as `protected`, so they can be accessed directly from code in the `Dijkstra` class since 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 `Dijkstra` can focus on the actual algorithm implementation.
- The provided class `Dijkstra` 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. If four command-line parameters were specified (that is, if `args.length == 4`), then `args[3]` will contain the name of a file where the program writes a `.pth` file with the route that can be plotted on the map by HDX.
- Note that there is a named constant `DEBUG` that is used to turn on or off "debugging" output. Setting this to `true` (really only appropriate when working on a small graph) prints lots of information about the vertices and edges being considered at each step by the algorithm.
- This version of the algorithm follows the pseudocode we considered earlier. It uses a `PriorityQueue` containing `PQEntry` objects, which are `Comparable`. It stops as soon as it finds the shortest path to the destination. This will be the stopping condition on your main loop. As written, it assumes that a path exists between your starting and ending vertices, so it is 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. Here, we only need to keep the last edge traversed. These are kept in a `Map` with strings (vertex/waypoint labels) as keys and `HighwayEdge` objects as values.

The `siena50-area.tmg` graph is also in your repository.

Run `Dijkstra` with this example and with debug mode turned on, using the same points as above (`US9/NY2`and `US9/NY32`) as your start and end. Verify that your answer matches what you saw from the HDX algorithm visualization.

? Question 11:
Capture your output from the above run in a text file named `latham-albany.out`. Include that text file in your submission. (2 points)

Run the same example, now with the option to generate the path file `latham-albany.pth`. View this file in HDX and generate a screen capture.

? Question 12:
Include your file `latham-albany.out` and your screen capture in your repository, and give the latter's file name as your answer to this question. (2 points)

The advantage of the Java implementation over the interactive algorithm visualization is that you can compute shortest paths on much larger graphs.

Also in your repository is a graph of all routes known to METAL in the United States, called `USA-country.tmg`. Run the `Dijkstra` program on this graph to compute the shortest path from `US9@FidLn` (the closest point to Siena) to `US41@5thAve` (downtown Naples, Florida). Create 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.

Question 13:
Include your Siena to Naples screen captures in your repository. Your answer to this question should be the file names. (3 points)

Question 14:
Repeat the above using a different METAL graph of your choice (see `http://tm.teresco.org/graphs/`) to compute the shortest path between two places of interest to you. Answer this question with the name of the graph you chose, the endpoints, and the names of the files that include a few screen shots of your computed route displayed in HDX. (5 points)

Bonus Opportunties

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

1. The first bonus opportunity that previously occupied this slot is cancelled, since the code distributed in your repositories included this! Sorry!
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.

Related Algorithms

We noted in a recent lab the similarities between Dijsktra's algorithm and a few others. Your task here is to make the small changes to the provided Dijkstra's algorithm implementation to make it an implementation of each of the algorithms listed below.

1. Prim's algorithm
2. Breadth-first traversal
3. Depth-first traversal

It is your choice whether you prefer to add command-line parameters to the existing code or make new classes that are almost identical to `Dijkstra` to do each. However, you should make changes *only to the value used as the priority for the* `PQEntry` *class*. Everything else should be unchanged.

This means that you will still have a start and an end location and will stop when you first reach the end location.

Your code submission for this part is worth 20 points.

Question 15:
Describe what the paths computed in each of these cases represents in terms of the original graph. If you were answering this for the original Dijkstra's algorithm implementation, you would say it computes the path from the start to the end such that the total length of the edges is minimized. (4 points)

Once you have this working, use your program(s) to compute the paths that would be computed for each of the earlier examples (Latham Circle to the Times Union Center, Siena College to Naples, and the one you chose). **Note:** the Siena to Naples is done on a large graph, and depth-first and breadth-first are likely to generate really long paths and/or take a very long time to compute. You may replace that with a an additional (different and smaller) example of your choice.

Question 16:
For each of the above examples and for each algorithm, give the shortest path computed, and the names of the files containing screen captures of your paths as shown in HDX. (10 points)

Generalized Heaps and Heapsort

You reviewed heapsort and learned about d-heaps as part of a recent lab. In a sense, heapsort 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 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.*).

Question 17:
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. (15 points)

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