Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2019

Problem Set 6

Due: 4:00 PM, Thursday, April 25, 2019 (no late submissions!)

You may work alone or in a group of size 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(s), should you choose to work in a group. 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, Monday, April 8, 2019. 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 `ps6-yourgitname`, for this problem set. Only one member of the group should follow the link to set up the repository on GitHub, then others will be granted write access.

Submitting

Please submit a hard copy (typeset preferred, handwritten OK but must be legible) for all written questions. Only one submission per group is needed.

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

Generalized Heaps and Heapsort

The heaps you know from data structures and the previous lab, where the heap is represented by a binary tree stored in an array, are one specific case of a more general structure called a *d-heap*. In a d-heap, each node has up to d children. So the binary heaps you would have seen before would be 2-heaps. For the questions below, assume that the minimum value is stored at the root node (*i.e.*, that it is a min-heap).

Question 1:
Show the construction of a 2-heap that results from the following values being inserted: 18, 9, 23, 17, 1, 43, 65, 12. How many comparisons are needed? (2 points)

Question 2:
Show the construction of a 3-heap that results from the following values being inserted: 18, 9, 23, 17, 1, 43, 65, 12. How many comparisons are needed? (2 points)

Question 3:
For the heap element at position i in the underlying array of a 3-heap, what are the positions of its immediate chidren and its parent? (Give formulas in terms of i.) ($\frac{1}{2}$ point)

Question 4:
For the heap element at position i in the underlying array of a d-heap, what are the positions of its immediate chidren and its parent? (Give formulas in terms of i and d.) ($\frac{1}{2}$ point)

Question 5:
Show the construction of a 1-heap that results from the following values being inserted: 18, 9, 23, 17, 1, 43, 65, 12. How many comparisons are needed? (2 points)

Question 6:
Show the construction of a 7-heap that results from the following values being inserted: 18, 9, 23, 17, 1, 43, 65, 12. How many comparisons are needed? (2 points)

You also reviewed heapsort 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. It's not exactly the same, since it uses a secondary data structure, but the general behavior is similar.

- 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.*).

Question 7:
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. (10 points)

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

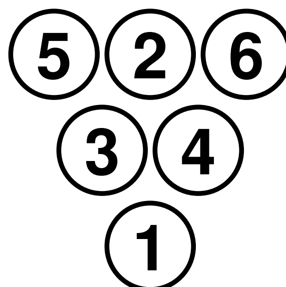
Backtracking Practice

Question 8:
Read Levitin p. 427–428, which describes the subset-sum problem. For each non-solution leaf in Figure 12.4 on p. 428 of Levitin, explain where the numbers in the condition come from and why that means the search can be cut off at that point. (5 points)

Question 9:
Draw a complete state-space tree for a backtracking approach to the subset-sum problem as applied to the example in Levitin Exercise 12.1.8 (a), p. 431. (5 points)

More Backtracking: The Billiard Ball Problem

The *billiard ball problem* consists of arranging a triangle of numbered billiard balls such that the resulting layout has the following arithmetic property: every ball below the top level is the absolute value of the difference between two balls immediately above it. For instance, the following is a solution to the problem when the top row consists of three balls (which requires a total of six balls).



Note that the balls are numbered from 1 to 6 in this case.

For the problems with four and five balls in the top row, there are a total of 10 and 15 balls, respectively.

You can read more about the problem in Martin Gardner's *Penrose Tiles to Trapdoor Ciphers: And the Return of Dr Matrix* (Cambridge Press 1997) on pages 119-120. (`http://books.google.com/books?id=8-FlYl6-ML8C&lpg=PP1&pg=PA119#v=onepage&q&f=false`)

As far as I can tell, there are no known solutions for problems larger than five balls in the top row.

A Backtracking Approach

A backtracking algorithm, very similar to that for n-Queens problem, is one approach to solving this problem.

The idea is that we attempt to build up a candidate solution by adding balls to the top row. Each time a ball is added, we make sure we have not broken any of the rules (in this case, there is just one: there are no repeated numbers in the triangle generated by that top row, including the top row itself). If we have not yet broken a rule, we either have found a solution (if the top row now contains the desired number of balls) or we have a partial candidate solution and we should add another ball. Any time we generate a candidate solution that does violate the rule, we have hit a dead end, so we undo the most recent addition and try the next option. If we ever backtrack all the way to the beginning and have run out of options for our first move, we know no solution exists.

For example, consider a backtracking solution to the problem where there are 2 balls in the top row, and we choose numbers from the largest to the smallest each time we reach a decision point. (Note that this is a good strategy to get a solution more quickly, as larger numbers will tend to be in the top row.)

We start with an empty solution, and we are ready to add the first ball to the top row. Since we are trying numbers from the largest to smallest, we start with 3:

```
(3)
```

This is a legal configuration: there are no repeated digits when we expand this out (in fact, there is no expansion needed for a single ball). So we accept this as a partial solution and move on, trying to add a ball to the second position. The first ball we attempt to place at this position is the highest numbered, 3:

`(3 3)` which expands to

```
(3 3)
 (0)
```

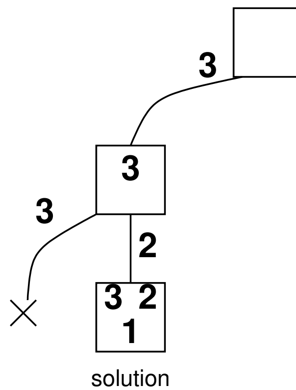
This is not a legal configuration: it includes two 3's. So we backtrack and erase our last move, and instead try the next option, which is to use the 2 ball:

`(3 2)` which expands to

```
(3 2)
 (1)
```

This is a legal solution: no repeats. Plus, we now have filled the top row, so our solution is complete.

We can display this in the format of a state space search diagram, like we did for n-Queens.



Question 10:
Show the state space search diagram for a backtracking solution to the problem with two balls in the top row if we instead chose balls for each position in increasing instead of decreasing numerical order. (4 points)

Now, let's consider the start of the procedure for the much more interesting (and much longer) backtracking computation of a solution to the problem with three balls in the top row. Note that here, we have a total of six balls.

Our first move is to place the largest number into the top row.

```
(6)
```

This is again legal, so we continue by adding a second number.

```
(6 6)
```

This contains a duplicate, so we backtrack and try a 5 in the second position.

```
(6 5)
 (1)
```

This is legal, so we accept the 5 for now, and start working on the third ball. We begin, as before, with the highest numbered ball and work our way down if we encounter illegal moves.

```
(6 5 6)
 (1 1)
  (0)
```

This has duplicates, so it is not legal. In fact, all of our choices for the third ball will result in illegal configurations here:

```
(6 5 6)    (6 5 5)    (6 5 4)    (6 5 3)    (6 5 2)    (6 5 1)
 (1 1)      (1 0)      (1 1)      (1 2)      (1 3)      (1 4)
  (0)        (1)        (0)        (1)        (2)        (3)
```

So this means 5 in the second position of the top row was a dead end. And we backtrack, and try a 4 there instead:

```
(6 4)
 (2)
```

So far so good here, so we move on trying each ball in the 3rd position of the top row...

Question 11:
Draw the state space search diagram for the 6-ball solution that will result from the above procedure. Note that this will not necessarily lead to the sample solution pictured earlier in this document. (8 points)

Question 12:
Write a program to find solutions to the billiard ball problem. 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. (15 points)

Question 13:
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. (2 points)

Question 14:
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? (1 point)

Dijkstra's Road Trip

We've looked at Dijkstra's algorithm for single-source shortest paths. 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 as much fun as 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 `NY2/US9`) to the Times Union Center (graph waypoint `NY32/US9`).

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 15:**
> 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 16:**
> About how many edges are in the PQ when an edge to the destination is first added? (1 point)

> **? Question 17:**
> 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? (1 point)

Now let the simulation run to completion.

> **? Question 18:**
> 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 19:
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? (1 point)

? Question 20:
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 21:
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 (`NY2/US9`and `NY32/US9`) as your start and end. Verify that your answer matches what you saw from the HDX algorithm visualization.

? Question 22:
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 23:
Include your file `latham-albany.pth` and your screen capture in your repository, and give the latter's file name as your answer to this question. (1 point)

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 (**not** HDX!) on this graph to compute the shortest path from `US9@FidLn` (the closest point to Siena) to `US41@5thAve` (downtown Naples, Florida), producing a file `siena-naples.pth`. Create 3 screen captures by loading `siena-naples.pth` into HDX: 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 24:
Include your Siena to Naples screen captures in your repository. Your answer to this question should be the file names. (1 points)

? Question 25:
Repeat the above using a different METAL graph of your choice (see `http://travelmapping.net/graphs/`, and be sure to change the filter settings if you want larger 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. (3 points)

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
4. Random 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 16 points.

> **Question 26:**
> 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. (3 points)

Once you have this working, use your program(s) to compute the paths (in `.pth` files) 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 27:**
> For each of the above examples and for each algorithm, submit the names of the `.pth` file that contain the paths, and the names of the files containing screen captures of your paths as shown 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. 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-region.tmg` from `US9/NY2` (Latham Circle) to `NY86@MirLakeDr` (Lake Placid) results in a path length of 35. 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 NY2/US9 to NY7/US9
```

```
 for 0.89 miles along US9, total 0.89
Travel from NY7/US9 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/NY
 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 NY73/US9
 for 2.14 miles along US9, total 99.52
Travel from NY73/US9 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
```

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.