Computer Science 501
Data Structures & Algorithms
The College of Saint Rose
Fall 2014

Lab 11: Dijkstra's Road Trip

Due: 6:00 PM, Tuesday, December 2, 2014

This lab, your last, and spread over 2 weeks, includes tasks related to trees and graphs.

You may work alone or in groups of 2 or 3 on this lab. Only one submission per group is needed.

Getting Set Up

To get your BlueJ environment set up for this week's lab assignment, start BlueJ and choose "New Project" from the "Project" menu. Navigate to your folder for this course and choose the name "Lab11" (no spaces) for the project.

Create a document where you will record your answers to the lecture assignment and lab questions. If you use plain text, call it "`lab11.txt`". If it's a Word document, you can call it whatever you'd like, but when you submit, be sure you convert it to a PDF document "`lab11.pdf`" before you submit it.

Lecture Assignment Questions

We will usually discuss these questions at the start of class on the lab due date, so no credit can be earned for late submissions of lecture assignment questions.

LA Question 1:
Bailey Problem 13.12, p. 339. (2 points)

LA Question 2:
Bailey Problem 13.20, p. 340. (2 points)

LA Question 3:
Bailey Problem 14.14, p. 365. (2 points)

LA Question 4:
Start with a complete AVL tree containing the values 10, 20, 30, 40, 50, 60, and 70. Insert the values 31, then 32, showing any rotations needed to maintain the AVL condition. (3 points)

LA Question 5:
Now, starting with the original tree, insert 32 then 31. Show any rotations needed to maintain the AVL condition. (3 points)

LA Question 6:

Design an AVL tree of height 4 such that if one more value is inserted, the root is the first unbalanced node on the way back up the tree. Once you have such a tree, insert that value, perform the proper rotation(s) and verify that the AVL condition is once again met. (3 points)

Problem Set Question: Dijkstra's Algorithm Practice

Question 1:

Using the graph from Bailey Problem 16.7, p. 436, use Dijkstra's Algorithm to compute the shortest distance from Dover to Phoenix by filling in the tables below, using the algorithm and notation as shown in the example in the graph notes. (10 points)

The data structures and the implementation of Dijkstra's algorithm are similar to those you will be using for the programming assignment below. Vertex labels are of type `City` and edge labels are of type `TravelLink`. You may assume that `TravelLink` provides methods to retrieve road names, distances, and driving times, but those details will not be important. The priority queue will contain objects of type `ComparableAssociation<Integer,Edge<City,TravelLink>>`, where the keys are the distances or driving times, as appropriate. The algorithm will populate a `Map` of shortest/fastest routes. The `Map`'s keys will be of type `City` and the values will be `ComparableAssociation<Integer,Edge<City,TravelLink>>` objects, from which we can get the total distance/time to the `City` from the starting `City`.

For each case, you are to fill in the given table, representing the `Map`, in the order in which the algorithm fills in entries, and show the values in the priority queue. Do not erase values as they are removed from the priority queue, just cross them out and write a number next to them to indicate the order in which they are removed from the queue. The map and priority queue should indicate their contents at the time the city "Phoenix" is added to the map.

Fill in the following table, which is a `Map` that has `City` objects as keys and `ComparableAssociations` of the shortest distance from Dover to the last edge traversed on that shortest route as values.

It is easiest to specify edges by the labels of their endpoints rather than the edge label itself, which might not be unique.

City	(distance,last-edge)
Dover	(0, `null`)

Also, use the table below to keep track of your priority queue. Remember, don't erase entries when you remove them from the queue, just cross them out and mark them with a number in the "Seq" column of the table entry to indicate the sequence in which the values were removed from the queue.

(distance,last-edge)	Seq

Problem Set Questions: Generalized Heapsort

You learned about d-heaps as you completed the questions in our previous lab. You also have learned 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.*).

Question 2:

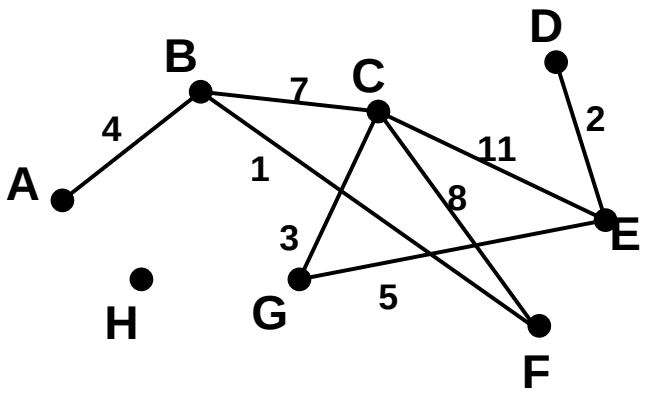
For each of the following underlying PQ structures, state which sorting algorithm proceeds in the manner most similar to the PQ-based sort using that PQ structure, and explain your answer briefly. (10 points)

- 1-heap
- 3-heap
- (n-1)-heap
- binary search tree

Practice Program

Practice Program:

Write a program `LittleGraph.java` that constructs a graph using one of the structure package's `Graph` implementations that represents the graph below, then prints it out using the `Graph`'s `toString` method. (4 points)



Programming Assignment

For your last bigger programming task, you will again be working with the highway graph data files linked from `http://courses.teresco.org/chm/graphs.html`. This time, we will be using the entire file, not just the waypoints.

The Data

Recall that the data is in "`.gra`" files which have the following format:

- The first line consists of two numbers: the number of vertices, $|V|$, (we'll call them "waypoints") and the number of edges, $|E|$, (road segments that connect adjacent waypoints).
- The next $|V|$ lines descibe the waypoints. Each line consists of a string describing a waypoint (its "label"), followed by its latitude and longitude as floating-point numbers.
- The last $|E|$ lines describe the road segments. Each line consists of two numbers specifying the waypoint numbers (0-based and in the order read in from this file) connected by this road segment, followed by a string with the name of the road or roads that form this segment.

Visualization of Graphs and Results

There are several motivations for using the highway data set to help study graphs and graph algorithms. One significant one is the ability to visualize the input data and result data on a map. Our data can be visualized by directing a browser at the "Highway Data Explorer (HDX)" `http://courses.teresco.org/chm/viewer/` and uploading a graph or other file in the file selection box at the top of the page. HDX is built on the Google Maps API.

The list of graphs at `http://courses.teresco.org/chm/graphs.html` has links to load complete graph files and visualize them with HDX.

Your first task is to augment your `WaypointBest` program from a few weeks ago that used your `BestOf` structure to find the collections of waypoints that were the first and last alphabetically by waypoint name, had the longest waypoint name, and were the furthest north, south, east and west. Extend your program so it creates seven output files, one for each of the criteria, in "Waypoint List File (WPL)" format. A WPL file consists of a list of waypoints, one per line, with the waypoint name, followed by a space, followed by the latitude, followed by a space, followed by the longitude. Have your program create each file with a "`.wpl`" extension. Then, from the HDX, to select one of your WPL files and see the points on the map.

? Question 3:

Include screen shots from HDX of the plots of the top 25 waypoints by each of the 7 criteria for the `usa-all.gra` input graph. (7 points)

Building a Graph and Implementing Simple Queries

Your next task is to be able to read the entire contents of a `.gra` file and to create a `Graph` structure using the structure package's graph implementations that represent all of the waypoints (graph vertices) and their connections (graph edges).

? Question 4:

The structure package has 4 implementations of the `Graph` interface: each combination of directed and undirected, and of list-based and matrix-based. Which graph implementation is most appropriate here, and why? (2 points)

Your overall approach will be to develop a Java program or programs that can read in graph data, store it appropriately in memory, and perform a variety of operations on that data. Two "starter" Java classes will be emailed to you and are available on mogul. You should work with these.

- Come up with an appropriate graph structure (something that holds the appropriate data in the vertex and edge labels) to hold this data and write code to construct one from a given `.gra` data file. You must use one of the graph implementations from Java Structures. Big hint: labels need not be simple objects like `String`s or `Integer`s. You can use any object type (including those you define yourself) for those labels. (9 points)

- Print out, in a nice format, a list of all waypoints. This is the `listPlaces` method in the starter code. (3 points)

- Print out, in a nice format, a list of all connections. This is the `listConnections` method in the starter code. (3 points)

- Print out the northernmost and southernmost latitudes, the easternmost and westernmost longitudes among waypoints in the graph, the shortest and longest waypoint names, the lengths of the shortest and longest road segments, and the average road segment length in the graph. Implement this as a new command `Stats`. Do **not** remember these in variables when you are reading the file and creating the graph. Compute them from your graph structure when the command is issued. In the case of ties for the longest and/or shortest names, your command should print all waypoint names of the extreme length. (12 points)

  For example, for the `dc-all.gra` graph, my `Stats` command prints:

```
Lat,Lng extents: (38.792435,-77.070508) to (38.984333,-76.934123)
Shortest waypoint names:
I-395@7
I-395@9
I-295@3
I-395@3
I-295@2
I-395@4
I-395@2
I-395@8
I-295@1
I-395@5
Longest waypoint names:
DC295@I-295/695&I-295@4&I-695@I-295/295
Connection lengths: shortest 0.0665295151, longest 1.69070, average 0.65215
```

Important note: while you might use an array or a `Vector` to track some information during the construction of your graph, the only persistent structure in your implementation should be an instance of a `Graph` class.

Bringing in `BestOf`

Your next task is to add a capability to your program to be able to find and report the n longest or shortest edges in the graph. Implement these as the commands `PrintLongestEdges` and `PrintShortestEdges` in your program.

For example, with the `canyt.gra` graph, the 10 shortest edges:

```
YT6@MacBoatLau to YT6@+X646670 via YT6 length 0.0000
YT1@MorLakeRS to YT1@BC/YT via YT1 length 0.0936
YT1@BeaCrkAir to YT1@CanCus via YT1 length 0.1077
YT2@DukeSt to YT2@GeoBlaFry via YT2 length 0.1979
YT4_S/YT6_S to YT4@OldCanRd&YT6@OldCanRd_S via YT4,YT6 length 0.2299
YT6@+x822 to YT6@+x116 via YT6 length 0.2522
YT6@+X297007 to YT6@+X720169 via YT6 length 0.2704
YT4@+x88 to YT4@+x87 via YT4 length 0.2739
YT2@+X456835 to YT2@CliAgaRd via YT2 length 0.3073
YT6@+X401385 to YT6@+x771 via YT6 length 0.3446
```

and the 10 longest edges:

```
YT1@+X372730 to YT1@+X931620 via YT1 length 10.3368
YT1@+X687758 to YT1@+X612218 via YT1 length 9.8789
YT3@+X260467 to YT3@+x898989 via YT3 length 9.0087
YT2@+X558217 to YT2@GraRd via YT2 length 8.2056
YT1@+X336247 to YT1@+X858279 via YT1 length 7.4635
YT4@+x48 to YT4@+X800213 via YT4 length 7.1541
YT5@+X717826 to YT5@+X920982 via YT5 length 7.0805
YT1@CanCus to YT1@+X680719 via YT1 length 6.9887
YT2@HunCrkRd to YT2@BonCrkRd via YT2 length 6.9588
YT6@+X620794 to YT6@LapCanTr via YT6 length 6.8125
```

A format similar to the above would be appropriate for text output. One way to accomplish this is to create a list of all edges and sort them by edge length. But we can do this more efficiently, as you learned in the `BestOf` lab. For 12 points, compute the set of longest/shortest edges in $\Theta(n|E|)$ time, where n is the number of longest/shortest edges you are looking for and $|E|$ is total the number of edges in the graph. Note that you can earn at most 5 points if your implementation is less efficient than the above, or if it does not compute the sets of edges from the graph in memory (as opposed to pre-computing them while the graph data was being loaded in).

For 5 points, you are to implement a second option, where the lists of longest or shortest edges are written to a file in a particular format. These should be implemented in the commands `MapShortestEdges` and `MapLongestEdges`. Here, each edge in the set of results should be specified by placing the vertex information for each of its endpoints on consecutive lines of a file. For example, the 3 shortest edges from `canyt.gra` would be specified in the file as:

```
YT6@MacBoatLau 62.86788 -130.82806
YT6@+X646670 62.868138 -130.827901
YT1@MorLakeRS 59.998729 -132.116818
YT1@BC/YT 60.000075 -132.117087
YT1@BeaCrkAir 62.407424 -140.860358
YT1@CanCus 62.40891 -140.85935
```

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

Implementing Dijkstra's Algorithm

Your final programming task is to develop a simplified "driving directions" system based on the mapping data you have been working with.

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 in a human-readable form (for the `FindRoute` command) or in a form plottable by HDX (for the `MapRoute` command).

For example, if you load the `ny-all.gra` file, and compute a shortest path for a few nearby points: `US20@WesAve` (the "Y" intersection at Western and Madison right near campus) and `NY2/US9` (Latham Circle), your path would traverse the following points:

```
US20@WesAve, NY443/US9W@US20&US20@US9W, NY5/US9W, US9/US9W
I-90@6/US9, NY377/US9, NY378/US9, NY155/US9 and NY2/US9.
```

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

```
Travel from US20@WesAve to NY443/US9W@US20&US20@US9W
 for 1.56 miles along US20, total 1.56
Travel from NY443/US9W@US20&US20@US9W to NY5/US9W
 for 0.37 miles along US9W, total 1.93
Travel from NY5/US9W to US9/US9W
 for 0.28 miles along US9W, total 2.21
Travel from US9/US9W to I-90@6/US9
 for 0.87 miles along US9, total 3.09
Travel from I-90@6/US9 to NY377/US9
 for 0.44 miles along US9, total 3.53
Travel from NY377/US9 to NY378/US9
 for 2.04 miles along US9, total 5.57
Travel from NY378/US9 to NY155/US9
 for 2.24 miles along US9, total 7.81
Travel from NY155/US9 to NY2/US9
 for 0.78 miles along US9, total 8.59
```

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

```
START US20@WesAve (42.666502,-73.791776)
US20 NY443/US9W@US20&US20@US9W (42.652458,-73.767786)
US9W NY5/US9W (42.656734,-73.763301)
US9W US9/US9W (42.659938,-73.759975)
US9 I-90@6/US9 (42.669562,-73.748817)
US9 NY377/US9 (42.675873,-73.747659)
US9 NY378/US9 (42.704925,-73.754568)
US9 NY155/US9 (42.736832,-73.76225)
US9 NY2/US9 (42.748115,-73.761048)
```

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/chm/viewer/` and uploading the `.pth` file in the file selection box at the top of the page.

Printing human-readable directions is worth 20 points, and generating a `.pth` file is worth 5 points.

Submitting

Before 6:00 PM, Tuesday, December 2, 2014, submit your lab for grading. There are two things you need to do to complete the submission: (i) Copy your file with the answers to the lecture assignment and lab questions into your project directory. Be sure to use the correct file name. If you prepared your answers in Word, export to a PDF file and submit that. (ii) Upload a copy of your lab (a `.7z` or `.zip` file containing your project directory) using Submission Box at `http://sb.teresco.org` under assignment "Lab11".

Grading

This assignment is worth 125 points, which are distributed as follows:

Feature	Value	Score
LA Question 1 (13.12)	2	
LA Question 2 (13.20)	2	
LA Question 3 (14.14)	2	
LA Question 4 (AVL Tree insert 31,32)	3	
LA Question 5 (AVL Tree insert 32,31)	3	
LA Question 6 (AVL Tree example)	3	
Question 1 (Dijkstra's Algorithm Practice)	10	
Question 2 (Generalized Heapsort)	10	
`LittleGraph` Practice Program	4	
Question 3 (`WaypointBest` map screen captures)	7	
Question 4 (which `Graph` and why)	2	
`Mapping` graph construction	9	
`Mapping` `listPlaces` method	3	
`Mapping` `listConnections` method	3	
`Mapping` `Stats` method	12	
`Mapping` `Print`{`Shortest,Longest`}`Edges` commands	12	
`Mapping` `Map`{`Shortest,Longest`}`Edges` commands	5	
`Mapping` `FindRoute` command	20	
`Mapping` `MapRoute` command	5	
`Mapping` style, documentation, and formatting	8	
Total	125	