Andrew Donley
July 7th, 2013
ThoughtWorks Train Problem

Documentation

Assumptions:
There should be no negative weight edges since a train track cannot extend a negative distance.
If there is a station A and a station C, it does not automatically follow that there should be a station B (this matters in the matrix multiplication algorithm).

Design:

For the overall design, I used the MVC paradigm to enable the program to evolve and its code to be maintained. This approach helped me decouple the algorithms from the views and enabled me to isolate problems within the logic of the code.

There are two main design features I would change in this program if I were to continue to work on it in the future: In the beginning the choice was made to go with LinkedHashSets in order to achieve a linked implementation of a graph and to deal with the issue of duplicate entries, but I failed to realize that there was no way to access the elements in constant time in the same manner as a HashMap. Instead, next time I would create my own abstract set class where the driving datastructure is a HashMap so that I could access the elements in constant time. Having the graph stored in LinkedHashSet fashion deteriorates from the runtime of a few graph algorithms. The second major idea I would change involves inheritance. Many of my GUI classes use the same functions and ActionListeners. In order to reduce the amount of redundant code, I would create superclasses where these functions were inherited instead of rewriting them.

Here is a quick overview of the algorithms I employed to solve different types of graph theory problems:

Iteration was used in order to solve the distance of a specific route. This involves following the path and returning the distance, if it exists.

Modified Dijkstra's Algorithm was used for calculation of shortest route from one node to another (different nodes). This algorithm terminates early once the shortest route to the terminal node has been established.

In order to find the shortest path from a node to itself, I **found back-edges to that node and then performed Dijkstra's Algorithm** on the graph. This way I could select the shortest back-edge plus Dijkstra value and have the shortest path from a node to itself.

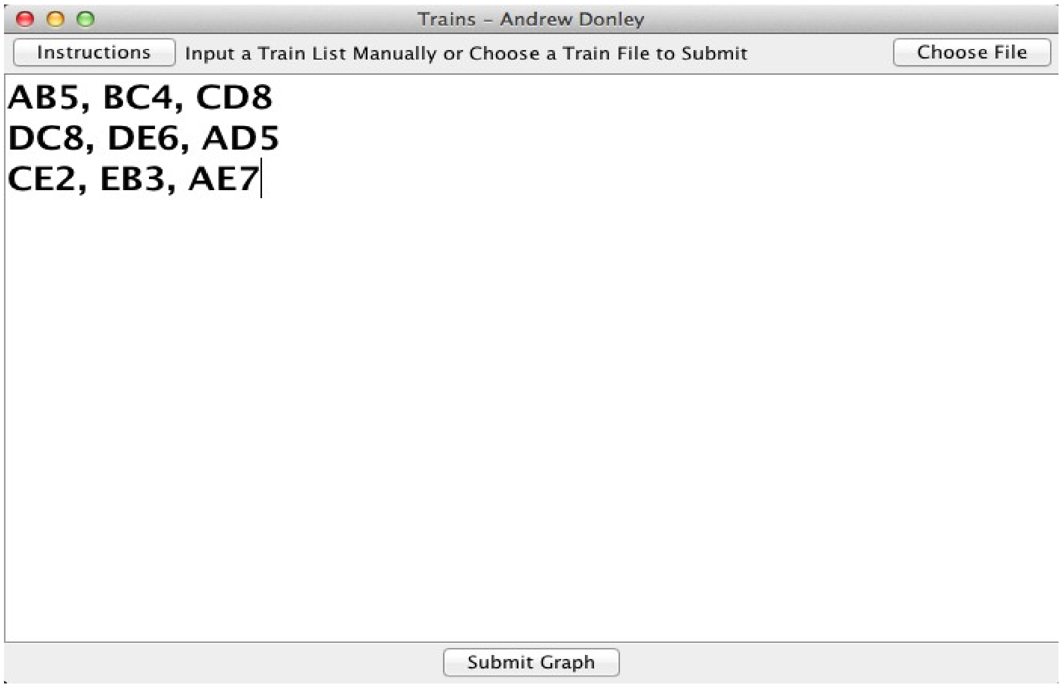
Matrix Squaring was used to calculate how many paths exist from one node to another node with at most or with a specific amount of stops.

Last, I used **recursive brute force** to find the number of paths from one node to another node with less than a certain distance. I spent the most time on this particular subproblem trying to identify a better solution, but all the alternatives ended up having equal time complexity.

Note: I used the eclipse compiler to compile this project.

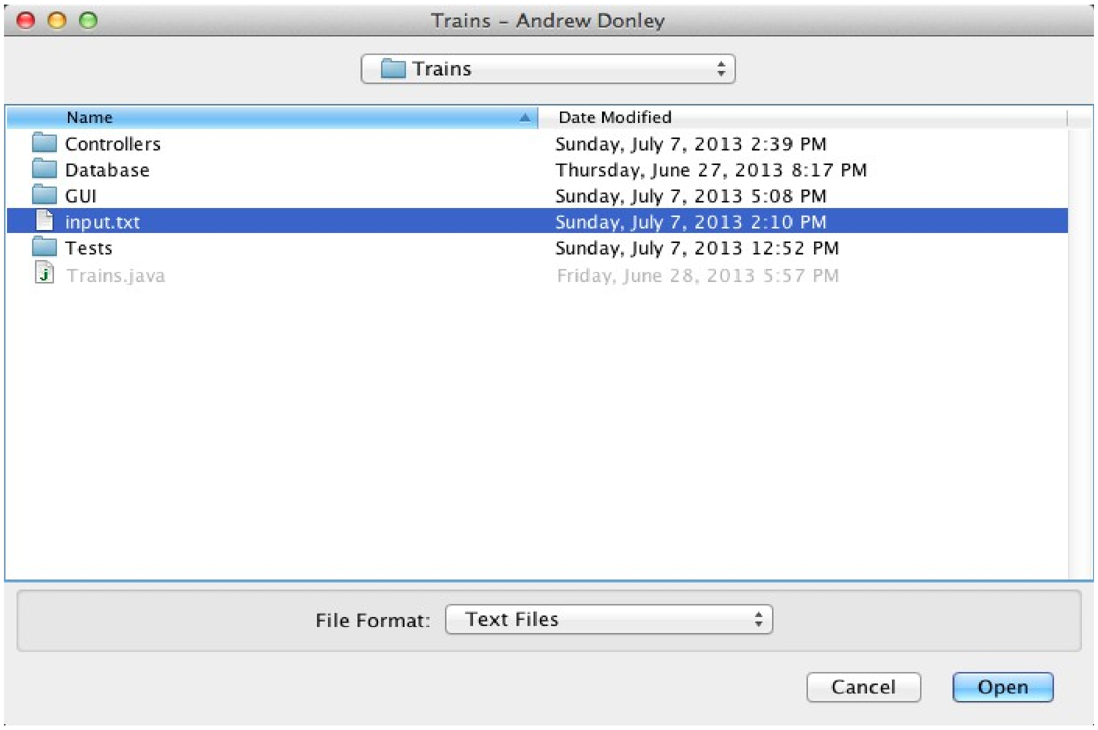
Usage:

When the program is opened you will be prompted by a screen that looks like this:


Trains - Andrew Donley
Instructions
Input a Train List Manually or Choose a Train File to Submit
Choose File
Submit Graph

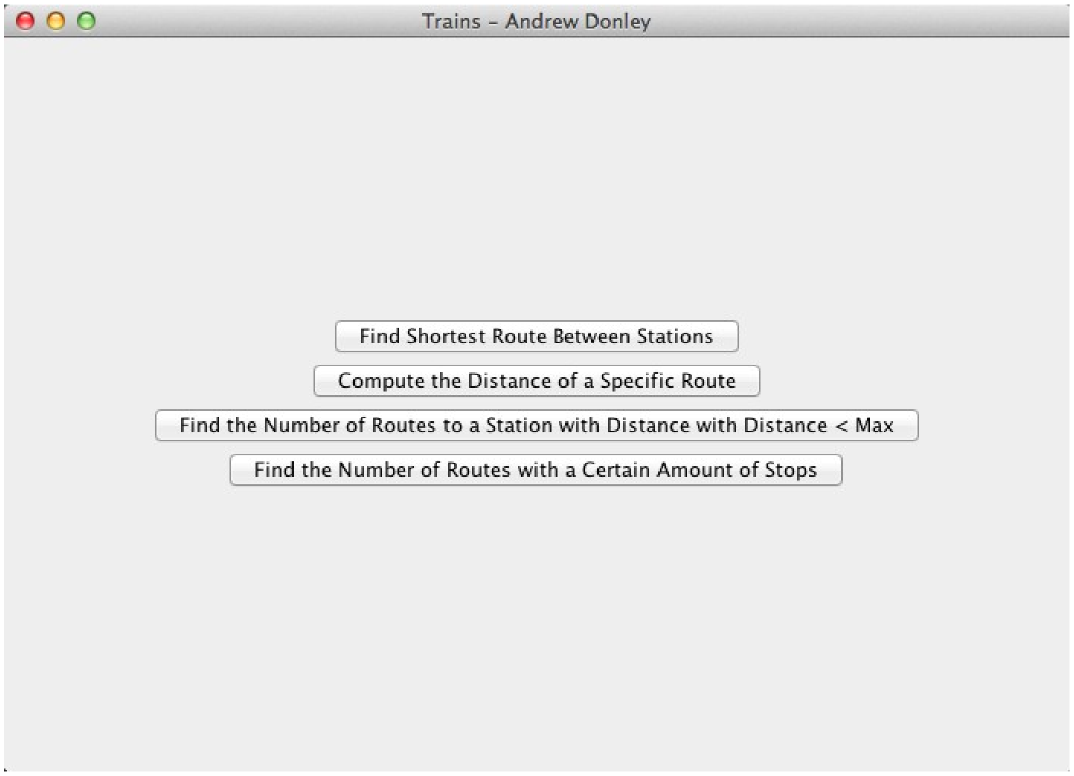
```
AB5, BC4, CD8
DC8, DE6, AD5
CE2, EB3, AE7
```

A valid graph is entered manually like the one above. It contains commas or new line characters to separate the entries, but it does not use both (a comma and a new line to separate entries). A file can be read in by clicking 'Choose File' instead. The file must follow the same formatting guidelines as the input box on this screen.



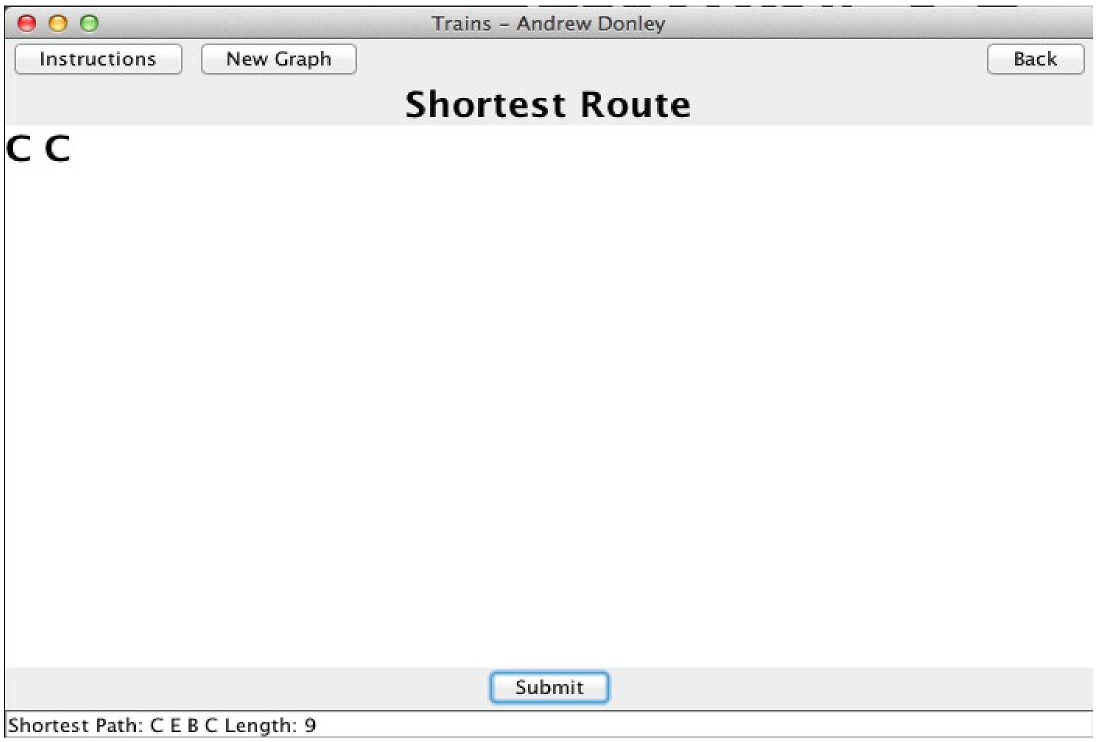
Name	Date Modified
Controllers	Sunday, July 7, 2013 2:39 PM
Database	Thursday, June 27, 2013 8:17 PM
GUI	Sunday, July 7, 2013 5:08 PM
input.txt	Sunday, July 7, 2013 2:10 PM
Tests	Sunday, July 7, 2013 12:52 PM
Trains.java	Friday, June 28, 2013 5:57 PM

The program will change to the solution choice screen if a graph of the proper format was submitted:


Trains – Andrew Donley
Find Shortest Route Between Stations
Compute the Distance of a Specific Route
Find the Number of Routes to a Station with Distance with Distance < Max
Find the Number of Routes with a Certain Amount of Stops

This screen will allow the user to select the type of solution that the user wants to perform. There are four solution types outlined on the prompt.

Shortest Route:

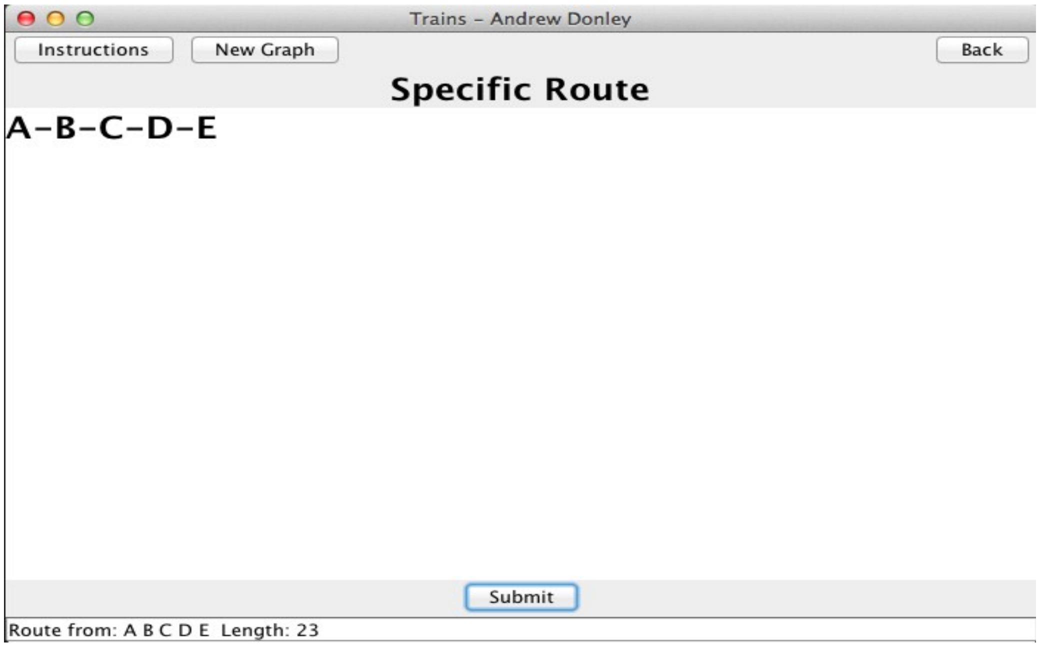


C C

Shortest Path: C E B C Length: 9

Valid input into the shortest path includes two characters with a dash or space in-between them. Clicking 'Submit' generates the answer that will appear at the bottom, underneath the 'Submit' button. 'New Graph' resets the database and returns the user to the initial screen, waiting for a new graph input. 'Back' will return the user to the select solution screen where the user can choose a different solution type for the graph in memory.

Specific Route:



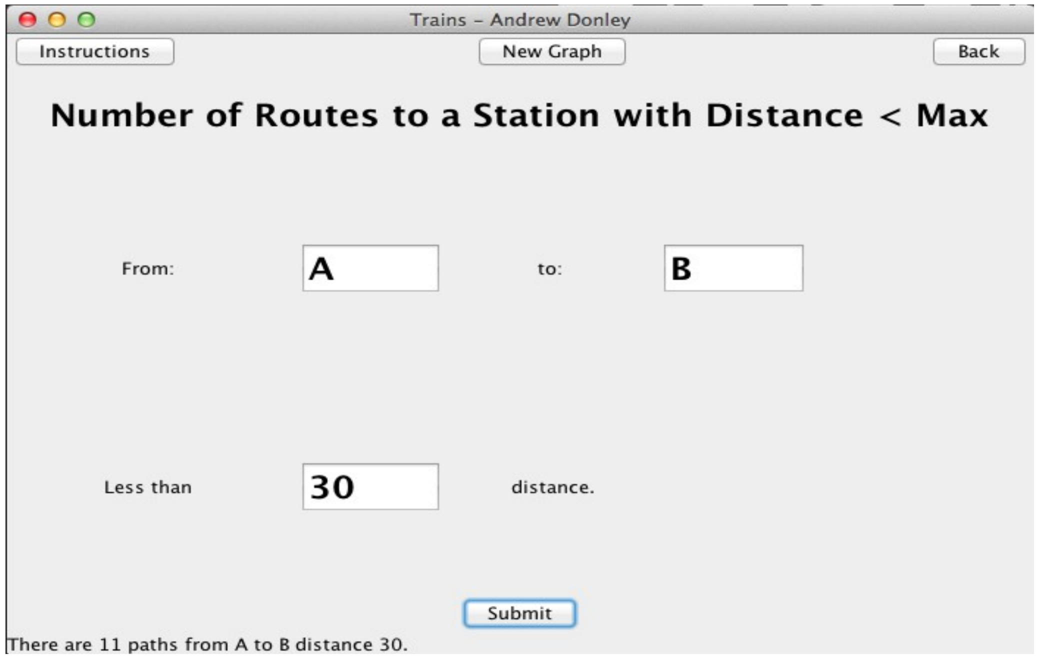
Specific Route

A–B–C–D–E

Route from: A B C D E Length: 23

Valid input for specific route calculation is performed by inserting characters with dashes in-between, representing the sequential route.

Number of Routes with Distance < Max:



Number of Routes to a Station with Distance < Max

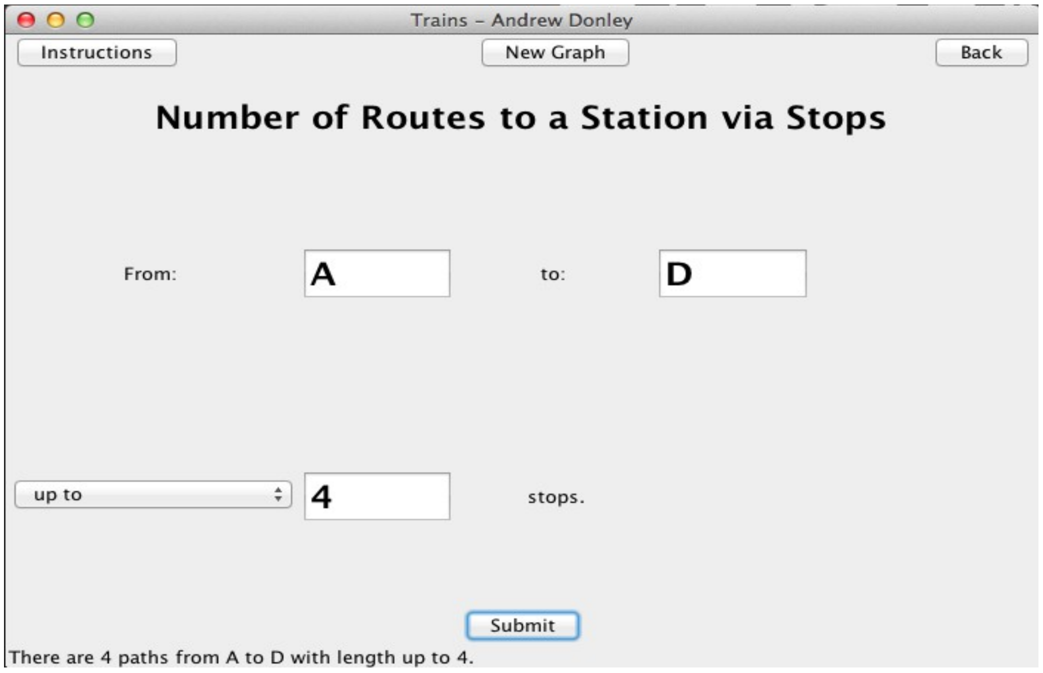
From: A to: B

Less than 30 distance.

There are 11 paths from A to B distance 30.

Enter the starting point station into the input box next to 'From:', and enter an ending point station into the input box next to 'to:'. These stations must be valid stations, represented by a single character (capitalization does not matter). Enter the upper bound max distance into the field next to 'Less than:'. The answer is displayed underneath the submit button.

Number of Routes via Number of Stops:

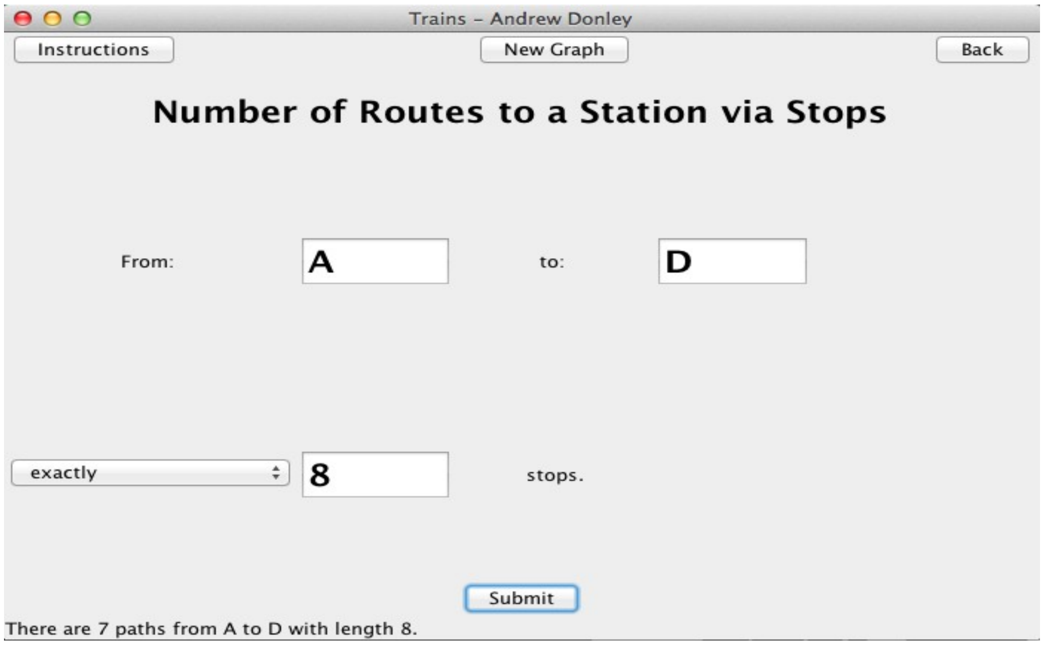


Number of Routes to a Station via Stops

From: A to: D

up to 4 stops.

There are 4 paths from A to D with length up to 4.



Number of Routes to a Station via Stops

From: A to: D

exactly 8 stops.

There are 7 paths from A to D with length 8.

The input for this is the same as the previous example, except for the drop down menu. Use the drop down menu to make a selection including all of the train routes up to any certain number, or alternatively, to make a selection of an exact number of routes. Last, enter the number of train stops into the input box located next to the drop down menu.