Design Document - CSU22012 Final Project

Anthony O'Connor, 20331833

Repository link: https://github.com/oconna62/algorithms-project.git

Part One: Shortest Paths

I implemented Dijkstra's shortest path algorithm to find the shortest route between two bus stops. I sourced the Dijkstra algorithm from the Algorithms, Fourth Edition book written by Robert Sedgewick and Kevin Wayne (https://algs4.cs.princeton.edu/44sp/).
I decided to use Dijkstra due to its low complexity, with a worst case asymptotic running time of O(V + ElogV), where E is the number of edges in the graph and V is the number of vertices in the graph, considerably better than that of the Bellman-Ford algorithm which has a worst-case asymptotic running time of O(EV), and Floyd-Warshall's brute force approach would not have been suitable for the objective of the program.

To initialise this part of the program I loaded stop details into ArrayLists to compare stop_id's to calculate weights for the various stops, before adding the stop_id's into a DirectedEdge. In the DijkstraSP main I took in the valid user inputs before creating a new DijkstraSP object and determining whether there was a route between the two stops. I then calculated the total route cost which I printed out after each individual stop along the route.

I implemented a number of error handling methods including catching the ArrayIndexOutOfBoundsException if the inputted value is greater than the total number of stops, ensuring that both inputted stop numbers were not the same. I decided to loop the method until both a valid input was found and the user entered 'exit' to prevent a swift exit to the menu.

Part Two: Stop Name Search

As outlined in the project specification document, I used a Ternary Search Tree to store and search for stop information. I again referenced Sedgewick and Wayne in using some of their TST.java approach (https://algs4.cs.princeton.edu/52trie/TST.java.html).

To preprocess the data I loaded it in from the stops.txt file, and within a while loop using scanner.hasNextLine() I split each line into its relevant parts and used substrings to move "FLAGSTOP", "WB", "EB", "NB" and "SB" to the back of the string. I then used a for loop and the TST.put() function to load the relevant information into a TST. For the main I took the input and made it uppercase to match the file data. I then made an ArrayList<String> to store the stops matching the input data, found with the TST.keysWithPrefix(inputName) function, and used their indexes to output the relevant stop information.

TSTs have good space efficiency and their time complexities are proportional to the height of the ternary search tree, with their operations having a worst case run time of O(n + k) for a string of length k, making them suitable for programs like this over alternatives.

This method involved very little error handling, barring skipping the first line of the scanner and catching ParseException errors from preprocessing the text file, mostly due to the simple nature of the input and from primarily using Strings. I again decided to loop the

method until both a valid input was found and the user entered ‘exit’ to prevent a swift exit to the menu.

Part Three: Stop Time Search

To initialise the data for this method I loaded in the text file and created a SimpleDateFormat("HH:mm:ss"). Within a while(scanner.hasNextLine()) loop I loaded in a line, split it to get the time associated with the trip and then parsed it into a date using the SimpleDateFormat. If the loaded time was equal to or less than the max time (23.59.59) I added it to a public ArrayList<String>.

In the main I followed the same idea to change the input String to a date, with a try catch ParseException to stop inputs not in the (HH:mm:ss) form. I used a for loop to compare the input time to the list of validated times, adding the validated time details to another ArrayList<String> if they were equal.

I chose to use iterative mergesort to sort the matching stops by stopID, because it is more stable than most other sorting algorithms, and despite it not using the most efficient, with a worst-case time complexity of O(N*logN) and quicksort being considerably more efficient, it was the easiest to implement as I was able to use Collections.sort(ArrayList).

After this I printed out the list of stops with matching times before asking the user if they would like to continue or return to the menu, having embedded the method in a while loop.

Part Four: UI

For the User Interface I created a main that displayed a welcome sign before initialising the different methods of the project to save time later on. After all the data was successfully loaded in I created a while loop that called the menu boolean method, only ending when the menu returned true, then printing out a goodbye message.

For the menu I printed out a list of options before prompting the user for an input. If the input was an integer and within 0->3, I had switch cases which then called the relevant function in the code. For error handling I looped this code until a valid input was found, with a try catch to ensure that the input was an integer. Throughout my code I printed out a series of “===” lines to make the output more readable.