Student ID: 20333843

Name: Muhammad Omaid Qureshi

Design Document

The following document will explain my design choice of the final project of Algorithms and Data Structures II. Three pieces of functionality were to be implemented in this project and I will explain my design choice of each functionality below.

Functionality 1 – Finding shortest paths between two bus stops

Implemented through userInput.java, ShortestPath.java

This functionality was executed using Dijkstra's algorithm. Since we are only interested in the distance from one node to another, BFS or DFS would not be applicable here. Bellman-Ford is another viable option however it is slower than Dijkstra and works better for negative weighted graphs. The version of Dijkstra implemented in this functionality is different to my Dijkstra class in Assignment 2, due to the need of printing out a path from the nodes. This requires a "parents" array which is being solely used for being able to print out the correct path at the end when Dijkstra's algorithm provides the shortest distances between the nodes. We will need to iterate through all the stops, of which there are precisely 12479 stops. Two other arrays which will be used in Dijkstra, a Boolean and distance array will also have a size of 12479.

The numbers may seem huge, but this is the best space/time trade off you can get since we are using the fastest algorithm for finding shortest distance and the space complexity is due to the file being so large. When reading in the two files for this functionality, a double adjacency matrix of 12479 x 12479 also must be created to store all the weights. However, this is run at the start of the program before the user inputs anything to the program, since it takes 10-30 seconds to initialize the matrix, and we do not want to keep rebuilding this matrix as we will get a memory error from the console.

Both these files are read at the start and utilizing scanners I am able to effectively extract every information and place it into the adjacency matrix.

<u>**Functionality 2 – Searching for a bus stop using a TST**</u>

<u>Implemented through userInput.java, TST.java</u>

The design choice of searching for a bus stop is limited, since we are told to use a TST. The code of TST from the Algorithm's book by Sedgewick is imported into this class, and I utilize a Queue class from the book for TST to work. In the constructor of TST I read in stops.txt file through a scanner, and the information is split up and appended into a StringBuilder object which is then later converted to a String. The stop name and this String are then inserted into the TST using put(). I used The keysWithPrefix() to search for the string in the TST. This will utilize a queue which is the most effective algorithm to use in this instance. Therefore, this is the best space/time trade-off, and this is the simplest solution to adding this functionality, since we are limited to any additional design choices.

<u>**Functionality 3 - Searching for all trips with a given arrival time**</u>

<u>Implemented through userInput.java, ArrivalTimes.java</u>

This is the simplest functionality to add if you utilize the correct classes. The SimpleDateFormat and Comparator class are vital in my code. We will receive a string from the user in the time format of HH:mm:ss (If not in this format, error is produced). We will then parse this string to a Date object. This is because when parsed to a Date object, we can utilize the method of getTime() due to the format being HH:mm:ss. We will error check the date provided by making sure its greater than 00:00:00 and less than 24:00:00. We will have to also read in the stop_times.txt and convert every arrival_time into a Date object and use the getTime() method to check if we have the desired arrival time. If we do, the whole line will be stored into an ArrayList (as opposed to a normal Array where we would have had to specify the space beforehand). The ArrayList is sorted according to trip_id because of the use of Comparator.naturalOrder() on the ArrayList which will take around a worst-case complexity of big O(n log n). At the end, we search through each element of our ArrayList and use a StringBuilder like we did for functionality 2 to split up the information of the trip in my userInput class. This is definitely the best space/time trade off considering the size of the file we read in.