**CSU22012 Algorithms and Data Structures**

**Final Project**

**Author: Yi Ren 20304391**

**Design & Approach**

**Part 1:**

My idea to solve this part is to first setup the graph using the three files given, then use Dijkstra algorithm to compute the shortest path between two stops. Most of the tasks are completed in the ShortestPath class.

For setting up the graph, I need to generate a set of stops as vertexes and connect the routes between stops as edges in the graph. To assist this, I used two HashMaps, as attributes of the class, to store the stops information since it is convenient when looking for a certain stop with the stop id. I read the stops.txt file in the generateStopName() function, assigned an index for each stop and then stored them into two HashMaps, one with the stop id as the key and the index as the value, and the other with the opposite.

After generating the stops, I used the generateRoutes() method to add directed edges and compute the cost by reading the stop times.txt and transfers.txt files. I placed each pair of sequence and stop id in a new HashMap for neighboring stops with the same trip id, and marked the buffer after reaching the end of one trip. Once all of the trip's edges have been connected and the 2D-arrays have been updated, the buffer will be reset and the next line will be read after the previous line. The values in distTo[][] and edgeTo[][] are set to infinite and -1 for the vertexes that are not linked. The graph setup is now complete.

For the shortest path, I selected the Dijkstra algorithm. The reason here is that, in comparison to Floyd-Warshall, I believe Dijkstra is more time efficient when looking for the shortest path between two given vertexes. As I don't have enough information to implement the A\* algorithm, I didn't take it into consideration.

As the user interface class for Part 1, I created Part1.java. It also served as the main function when testing part 1 individually. The handling of user input is also completed in this class.

**Part2.**

My idea for this part is to first read the file, put the formatted stop names into the Ternary Search Tree, then take in the input and search for the keys with the certain prefix.

In order to put the flag from the beginning of a stop name to the end of it while reading the file,  I created the formatWords() method, which breaks the stop name into two components and then switches their order. Once a name is formatted, the stop name is kept in the tree as a key, and the stop id is saved as the corresponding value.

For the TST part, I used the code from Princeton University for the TST class as a package. I imported the TST.java(from <https://algs4.cs.princeton.edu/52trie/TST.java>) and Queue.java(from <https://algs4.cs.princeton.edu/13stacks/Queue.java>) in order to use the functions in TST class.

In the search() function, I used the keysWithIndex() function in TST class to find all keys whose first few characters match with the input string. Using the keys returned by the function, I can extract their corresponding unique stop id so that the full information can be found by walking through the file to look for the certain stop id in the findInfo() function.

The SearchByName class includes 10 public ArrayLists that are used to hold the 10 different types of stop information. They are re-initialized at the start of the search() method so that the results of the previous search can be rewritten. After the full information is found, it is divided into categories and stored in the ArrayLists so that I can print it in a tabular format in the user interface.

As the user interface class for Part 2, I created Part2.java. It also served as the main function when testing part 2 individually. The handling of user input is also completed in this class.

**Part3.**

For this part, most tasks are simply done in the readFile() function. Similar as in part2, the SearchByTime class includes 9 public ArrayLists that are used to hold the 9 different types of trip information. They are also re-initialized at the start of the readFile() method so that the results of the previous search can be rewritten. The function takes in the user input and uses string comparison to compare the input to the arrival time from each line in stop time.txt. The filter for time over 23:59:59 is done in the user interface.

As the user interface class for Part 3, I created Part3.java. It also served as the main function when testing part 3 individually. The handling of user input is also completed in this class.

**Part4.**

For the user interface, I displayed a welcome message and used a switch statement to simply combine Part1.java, Part2.java, and Part3.java to output all of the data to the console.