Design Document

For reference, I have built this with the intention that it be used by ordinary people who are ONLY interested in getting from bus X to bus Y. This becomes more apparent in the output of messages or the responses of incorrect input.

Part 1:

ALGORITHM: I considered using Greedy or A* but neither could guarantee me the shortest path. Dijksta's could guarantee me the shortest path. Other Algorithms such as Bellman Ford or Floyd Warshall run in O(VE) and O(V^3) while Dijkstra's runs in O(V^2). Bellman Ford also returned more information than was necessary.

DATA STRUCTURE: Adjacency list, needed quick access to the adjacent nodes of a given vertex for Dijkstra's but I didn't want to take up so much space given that the file's are quite large in storage. This is why I chose an adjacency list over an adjacency matrix.

GENERAL INFO: I also read in the graph before the UI instead of reading in the graph everytime I wanted to use dijkstra's. I decided to hash the stopIDs to the integers between 0 and the number of stop ID's, this is because I can work easily with the integers and then translate back.

This allows me to use dijkstra's efficiently, using these integers to represent my stops, saving on both time and space.

I achieved this via two arrays, one being the hash function, takes in input as an index and the integer at that position is the output of the hash function. The other being it's inverse with the same principle to avoid searching through the array every time I wanted to translate back.

Part 2:

DATA STRUCTURE: I used a TST instead of an ordered array or a priority queue, as searching for certain words with a given prefix is faster as they are just the children of the prefix's last character's node in the TST.

ALGORITHM: The code used explores the subtree of the given prefix's last character using preorder traversal, exploring the left subtree, then the middle and then the right. To find all words in the given subtree.

GENERAL INFO: I chose to only return the human readable information, as everything else would not be of interest to a general user and more-so a distraction. This is of better quality to the user.

Part 3:

DATA STRUCTURE: I added it into an array instead of something like a linked-list because the immediate access allows for quicker sorts. As well as that the number of arrival times matching the given arrival time tends to be quite small.

ALGORITHM: I used quicksort instead of merge sort as quick sort tends to work more efficiently on the smaller arrays. Quick sort is implemented via the built-in Arrays.sort() method in code.

GENERAL INFO: Similarly here I have only returned the Trip ID, arrival time, and stop ID as all the other information is not useful to a general user.

Part 4:

GENERAL INFO: I have a response for all invalid inputs, to ensure myself of this, I sectioned the code into smaller pieces each of which work correctly. E.g: The method to check an inputted time makes sense (timeObject.isValid()) is used as well as a try-catch block. The try- catch block is present to ensure I can turn the input into a time object, then isValid() is used to make sure that it is a valid time.

Other features include:

- Basic error handling with try/catch blocks.
- Ability to exit program/subprogram by entering “exit” even with medial capitals at any time in the program.
- Appropriate messages/responses from the computer in the case of incorrect input.