Design Document for Final Assignment in Algorithms and Data Structures 2

Part 1: Shortest path between stops function

The function that calculates the shortest path between two stops and the cost of the path works in 2 main parts. The first part is completed when values are loaded into the program from the stops.txt, stop_times.txt and transfers.txt files. The stops.txt file loads in the stops into an ArrayList of type Stop, this type stores all the information needed by the Stop. Once we load in a stop we also give it a pathIndex value that is specific to each stop. Using this pathIndex value we can make a path matrix where each stop has a row of all stops with the associated cost of the travel from that stop to the other stop. The cost of the travel is loaded in from the stop_times.txt and transfers.txt following the guidelines outlined in the assignment document.

The second part is the function itself and the processes that take place when it is called during the program's runtime. With the values loaded in we then use Dijkstra's shortest path algorithm to calculate the lowest cost path to any other stop in the matrix. We only calculate this cost for a specific stop that is called by the function thus saving processing time. I use Dijkstra's algorithm here because it's very good for larger data sets since we only need to calculate the cost of a specific stop's travels rather than all the stops beforehand like in the Floyd Warshall algorithm. The time complexity of Dijkstra's algorithm is ON(logN) which is pretty good given the task.

I use a nested array of floats for the path matrix in this function. Since we calculate most of the values needed before the program allows user input the time it takes for each query to process is relatively low. Despite this the array of float arrays is quite large due to the large number of stops, the size complexity for the array is ON^2 which makes large data sets take up a decent amount of space but it is one of the smallest sizes that allows for the faster processing after the data is loaded.

Part 2: Search by Name function

The function that searches for all the stops with a given name or part of a name works using the values loaded in from the stops.txt file and no other data. This is because the only information needed to find a stop and the information needed from the stop are in the stops.txt file and my function doesn't use anything else because of that.

The loading of the stops.txt file saves each of the stops into an ArrayList of type Stop, the type Stop simply holds all the information for each stop as an object making it much easier to retrieve and print out information into the command line since each stop has a printInfo() function. Once all the stops are loaded into the ArrayList no other information is needed for them so the function itself does most of the work in this situation.

The function takes in a String of any length and then compares it to the name saved in each of the stop's name variables. I also account for the first 2 characters of name String being a prefix in certain situations and move them to the end when outputting the name and comparing the name to the imputed string. Once a match is found the stop gets added to an array of stops that matches the input string and all the matching stops have their printInfo() function called so that we get all the information for each matching stop.

The time complexity for this function is ON and the space complexity is ON also since it is saved in an array and the function just looks through the whole array.

Part 3: Search for Trip by Arrival Time

This function saves all the information in the stop_times.txt file and uses it to find any stops that a trip makes with an arrival Time that matches the imputed arrival time.
This function works by loading in the information from the stop_times.txt file into an ArrayList of type Trip which each have their own ArrayList of type TripNode. I simply look at each line of the stop_times.txt file and take the trip_id value and save it if it's unique to the last trip_id added. From there we add all of those trips nodes until the next line of the file has a new trip_id and then make a new trip and repeat the process till all lines are added. Any invalid times are not added to the node ArrayLists.

Once the information is added to the ArrayLists the function that runs when the request is run just loops through all the nodes for each trip and saves any that match the requested arrival time. Each node that matches the arrival time has its trip added to the array of matching trips and the next trip in the ArrayList is checked until all trips are checked.

The space complexity for the ArrayList of Trips and ArrayList of tripNodes depends on the amount of nodes per trip, when trip count = N and nodes per trip = K then the complexity is O(NK) which could be better but allows for fast and easy loading of the stop_times file.

The time complexity of the loading is simply the size of the file in lines so ON and the time complexity for the function is O(NK) as explained above since we loops through all nodes of all trips.