DESIGN DOCUMENT FOR ALGORITHMS & DATA STRUCTURES FINAL PROJECT

Shortest Path

I chose to use Dijkstra's algorithm to compute the path of lowest cost between two bus stops. Given that the number of bus stops is approximately 8760, the space used is efficient as it is O(V), where V = number of stops (vertices). The worst case time complexity is O((V+E) log (V)) where E = number of edges, in our case about 1.7 million. This is incredibly fast for our dataset. I did not opt for Floyd- Warshall's algorithm because there is no need to compute the path with lowest cost between all bus stops as this would take up much more space than is required given the specification is that the end user inputs two stops at a time. I also considered using the A* algorithm instead but finding the right heuristic is not trivial. I thought of calculating the great-circle distance between the points using Haversine's formula for the heuristic but I could not prove that it is admissible given that the costs are not actually distances so there is no certainty that the heuristic will not overestimate the cost.
I used a binary min heap priority queue in dijkstra as this efficiently gets the min element in O(log(N)) time in the worst case, where N is the number of elements in the queue. I implemented a compareTo method in the Pair class that specified that elements in the queue should be compared by cost. I used a deque to construct the path because I wanted to be able to iteratively add to the start to build the path. This ensured that there was no need to reverse which takes an extra O(N) time in the worst case.

TST

I implemented the TST using the last letter of the bus stop name as the key and a custom Stop object as the value. The Stop object holds details about the bus stop including the bus stop name and stop id. The insertion is done recursively until all the letters of the stop have been added. The find method initially works similarly to the insert method but when the recursion is at the last letter of the prefix that the user has entered, it calls another method to find all bus stop names with that prefix. This method is also recursive and the base case is when the mid is null. This tells us that we are at the end of the word and have added the Stop and the Stops of all sub strings that are also words. During each recursive call, a word is added if node.value is not null (if it was inserted into the TST) if the base case deceived above does not hold, it calls another recursive method to retrieve all the nodes that are next to the current node and loops over them to get the next nodes. Next nodes means the nodes that have a key that is the next letter of the key of the current node. This has to be retrieved recursively because of how keys are stored in a TST. We add the mid value of the current node as part of the next nodes and then perform a depth first search considering only the left and right nodes each recursive call. The base case is when we hit null. All the nodes we find during this search are the next nodes of the current node. In terms of time and space complexity. I did not bother to randomise the bus stop names as they were not sorted. Also, there are only about 8760 bus stops to insert. Shuffling to guarantee that the worst case is O(logN) is not necessary. By practical testing, I found that the search is blazingly fast with my implementation so there was no need for further optimisation.

Arrival Time Matching

I used Linear search which has O(N) worst case time complexity and then Java's sorted method on the stream which has O(Nlog(N)) time complexity in the worst case. Another option was to use Binary search but that would require sorting according to time first. For efficiency, after parsing the stop_times.txt file into edges, we could do the sorting. However,

though the search for arrival times is now log(N) in the worst case, we still have to sort according to the trip id. This makes the worst case time complexity also Nlog(N). Given that they both have the same time complexity, I still opted for the linear search because though the worst case time complexity for binary search is O(N) There still has to be a secondary search around the index given by the binary search to get all the matches.
In my implementation, I store the stop time as an edge from the previous line with the same trip ID, with a cost of 1 as described. The transfers are also stored as edges but with a different constructor that does not accept arrival time. When searching for a matching arrival time, N is actually the number of all the edges including the ones from transfers.txt. Given that the number of lines in stop_times.txt is over 1.7 million and the number of lines in transfers.txt is about 5,000, the extra added edges is negligible in terms of the time complexity. I chose to do this to avoid parsing the stop_times.txt file each time an arrival time is entered by the user or to create another Class to hold stop_times.txt specifically., doing this would require storing 1.7 million objects in a list which is unnecessary.
I sort according to trip id immediately after filtering the edges with arrival times that match. By specifying that Edge implements Comparable<Edge> and overriding the compareTo method in the Edge class, I was able to specify that edges are sorted by trip id.
The space complexity is O(N). This could be O(1) if for each input by the user the stop_times.txt is parsed, but there still needs to be sorting done by the trip id so O(1) space complexity is impossible.

General

The stops.txt, stop_times.txt and transfers.txt files are parsed only once. The graph and TST are also built once before the user selects one of the three options.

NOTE that O is used to represent Big O